Java

למקצוענים

שים לב!

יש להתעלם מהוראות התקנת התקליטור

הקבצים נמצאים באתר הוד-עמי בקטגוריה

"קבצי תרגול לספרים"

קוד הספר 59345

הקבצים לא כוללים תוכנה

עורך ראשי: **יצחק עמיהוד**

עורך מקצועי: **משה יצחקי**

עריכה מקצועית: **זהר עמיהוד**

תרגום: **חן קורן**

עריכה ועיצוב: **רמה שנקלר**

ענבל אילני

טליה טופז

עיצוב עטיפה: **שרון רז**

שמות מסחריים

שמות המוצרים והשירותים המוזכרים בספר הינם שמות מסחריים רשומים של החברות שלהם. הוצאת הוד-עמי והוצאת Addison Wesley Longman עשו כמיטב יכולתן למסור מידע אודות השמות המסחריים המוזכרים בספר זה ולציין את שמות החברות, המוצרים והשירותים. שמות מסחריים רשומים (registered trademarks) המוזכרים בספר צוינו בהתאמה.

Java הינו מוצר רשום של חברת SUN.

הודעה

ספר זה מיועד לתת מידע אודות מוצרים שונים. נעשו מאמצים רבים לגרום לכך שהספר יהיה שלם ואמין ככל שניתן, אך אין משתמעת מכך כל אחריות שהיא.

המידע ניתן "כמות שהוא" ("as is"). הוצאת הוד-עמי והוצאת Addison Wesley Longman **אינן אחראיות כלפי יחיד או ארגון עבור כל אובדן או נזק אשר ייגרם, אם ייגרם, מהמידע שבספר זה, או מהתקליטור שמצורף לו.**

לשם שטף הקריאה כתוב ספר זה בלשון זכר בלבד. ספר זה מיועד לגברים ונשים כאחד ואין בכוונתנו להפלות או לפגוע בציבור המשתמשים/ות.

- **טלפון: 09-9564716**
- **פקס: 09-9571582**
- **דואר אלקטרוני: info@hod-ami.co.il**
- **אתר באינטרנט: www.hod-ami.co.il**

Java

למקצוענים

Joshua Bloch

Effective Java Programming Language Guide

By Joshua Bloch

Editor: **I. Amihud**

Authorized translation from the English language edition, entitled Effective Java™ Programming Language Guide, First Edition by Joshua Bloch, published by Pearson Education, Inc, publishing as ADDISON WESLEY LONGMAN, Copyright © 2001 Sun Microsystems, Inc.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

Hebrew language edition published by Hod-Ami Computer Book Publisher Ltd.
Copyright © 2002.

©

כל הזכויות שמורות

הוצאת הוד-עמי לספרי מחשבים בע"מ

ת.ד. 6108 הרצליה 46160

טלפון: 09-9564716 פקס: 09-9571582

info@hod-ami.co.il

תרגום מורשה של ההוצאה המקורית באנגלית כמפורט למעלה, שהודפסה על ידי
ADDISON WESLEY LONGMAN, Copyright © 2001 Sun Microsystems, Inc.

אין להעתיק או לשדר בכל אמצעי שהוא ספר זה או קטעים ממנו בשום צורה ובשום אמצעי אלקטרוני או מכני, לרבות צילום והקלטה, אמצעי אחסון והפצת מידע, ללא אישור בכתב מאת ההוצאה, אלא לשם ציטוט קטעים קצרים בציון שם המקור.

הודפס בישראל 2002

All Rights Reserved
HOD-AMI Ltd.
P.O.B. 6108, Herzliya
ISRAEL, 2002

ISBN 965-361-314-6 מסת"ב

תוכן עניינים מקוצר

מבוא..... xi

פרק 1: הקדמה 1

פרק 2: יצירה ופירוק של אובייקטים..... 5

פרק 3: שיטות המשותפות לכל האובייקטים 25

פרק 4: מחלקות וממשקים (Classes and Interfaces) 59

פרק 5: תחליפים למבני נתונים בשפת C 97

פרק 6: שיטות (Methods) 119

פרק 7: כללי תכנות כלליים..... 141

פרק 8: חריגים (Exceptions) 169

פרק 9: מטלות (Threads)..... 189

פרק 10: Serialization 213

נספח: התקליטור המצורף..... 233

באנגלית:

References..... 1

Index of Patterns and Idioms 6

Index 10

תוכן העניינים

מבוא..... xi

פרק 1: הקדמה 1

פרק 2: יצירה ופירוק של אובייקטים..... 5

נושא 1: שקול שימוש בשיטות יוצרות סטטיות במקום בבנאים 5

נושא 2: אכיפת תכונת singleton בעזרת בנאי פרטי 10

נושא 3: מנע באמצעות בנאי פרטי את היכולת ליצור מופע..... 12

נושא 4: הימנע מיצירת אובייקטים כפולים..... 13

נושא 5: סלק ייחוסים ישנים לאובייקטים..... 17

נושא 6: הימנע ממסיימים (finalizers) 20

פרק 3: שיטות המשותפות לכל האובייקטים 25

נושא 7: ציית להסכם הכללי בעת דריסת equals 25

נושא 8: דרוס תמיד את hashCode כאשר אתה דורס את equals 36

נושא 9: דרוס תמיד את toString..... 42

נושא 10: דרוס את Clone בתבונה..... 45

נושא 11: שקול לממש את ממשק Comparable 53

פרק 4: מחלקות וממשקים (Classes and Interfaces)..... 59

נושא 12: מזער את הנגישות אל מחלקות ואיברים..... 59

נושא 13: העדף להשתמש במחלקות שאינן ניתנות לשינוי (immulable class)
כאשר הדבר אפשרי 63

נושא 14: העדף הרכבה (composition) על פני הורשה 71

נושא 15: תכנן ותעד ביצוע הורשה או שתאסור שימוש בה..... 78

נושא 16: העדף ממשקים על פני מחלקות מופשטות 84

נושא 17: השתמש בממשקים רק כדי להגדיר טיפוסים..... 89

נושא 18: העדף מחלקות מקוננות סטטיות על פני כאלו שאינן סטטיות 91

פרק 5: תחליפים למבני נתונים בשפת C 97

נושא 19: החלף מבנים במחלקות 97
נושא 20: החלף איחוד (union) בהיררכיות מחלקה 100
נושא 21: החלף את מושגי enum במחלקות 104
נושא 22: החלף מצביעי פונקציות במחלקות וממשקים 115

פרק 6: שיטות (Methods) 119

נושא 23: בדוק את תוקפם של פרמטרים 119
נושא 24: צור בעת הצורך עותקים הגנתיים 122
נושא 25: תכנן בזהירות חתימות שיטה 126
נושא 26: השתמש בשיקול דעת בהעמסה 128
נושא 27: החזר מערכים באורך אפס, לא nulls 134
נושא 28: כתוב הערות תיעוד לכל מרכיבי ה- API הגלויים 136

פרק 7: כללי תכנות כלליים 141

נושא 29: מזער את טווח ההכרה של משתנים מקומיים 141
נושא 30: הכר והשתמש בספריות הקוד של Java 145
נושא 31: הימנע מ- float ומ- double אם נדרשות תשובות מדויקות 149
נושא 32: הימנע ממחרוזות כאשר טיפוסים אחרים יותר מתאימים 152
נושא 33: היזהר משרשור מחרוזות 155
נושא 34: התייחס לאובייקטים בעזרת הממשקים שלהם 156
נושא 35: העדף ממשקים על פני השתקפות (Reflaction) 158
נושא 36: השתמש בשיטות native בתבונה 161
נושא 37: בצע אופטימיזציה בתבונה 162
נושא 38: דבוק במוסכמות מתן שמות מקובלות 165

פרק 8: חריגים (Exceptions) 169

נושא 39: השתמש בחריגים רק במצבים יוצאי דופן 169
נושא 40: השתמש בחריגים מסומנים למצבים בני-התאוששות
ובחריגי זמן ריצה לשגיאות תכנות 172
נושא 41: הימנע משימוש מיותר בחריגים מסומנים 174
נושא 42: העדף שימוש בחריגים סטנדרטיים 176
נושא 43: עורר חריגים שמתאימים להפשטה 178
נושא 44: תעד את כל החריגים שכל שיטה מעוררת 181
נושא 45: כלול מידע לכידת כישלון בהודעות המייצגות את החריג 183

נושא 46: שאף ל- failure atomicity 185
נושא 47: אל תתעלם מחריגים 187

פרק 9: מטלות (Threads) 189

נושא 48: סנכרן גישה לנתונים משתנים משותפים 189
נושא 49: הימנע מסינכרוניזציה מוגזמת 196
נושא 50: לעולם אל תקרא ל- wait מחוץ ללולאה 201
נושא 51: אל תהיה תלוי במתזמן המטלות 204
נושא 52: תעד בטיחות מטלה 208
נושא 53: הימנע מקבוצות מטלה 211

פרק 10: Serialization 213

נושא 54: ממש העברה טורית בתבונה 213
נושא 55: שקול שימוש בהעברה טורית מותאמת אישית 218
נושא 56: כתוב שיטות readObject הגנתיות 224
נושא 57: ספק את השיטה readResolve בעת הצורך 230

נספח: התקליטור המצורף 233

הפעלת התקליטור 234
קטלוג ספרים 234
התקנת תוכנות 235
התוכנות המומלצות להתקנה 235
Adobe Acrobat Reader 235
קוד המקור שבספר 236
Microsoft Internet Explorer 6 - התקנת הגירסה המצורפת 237
שאלות נפוצות 238
הערה חשובה! 238
צור קשר 238

באנגלית:

References 1

Index of Patterns and Idioms 6

Index 10

מבוא

I sure wish I had this book ten years ago. Some might think that I don't need any Java books, but I need this one."
James Gosling, Fellow and Vice President, Sun Microsystems, Inc.

אם עמית למקצוע היה אומר לך, "בן זוג שלי הלילה הזה היום מייצר ארוחה הלא רגילה בבית. האם אתה מצטרפת?" סביר להניח ששלושה דברים יעברו בראשך: השלישי הוא שהוזמנת לארוחה; השני, עברית אינה שפת האם של עמיתך; הראשון, לא מעט מבוכה.

אם אי פעם למדת שפה שנייה בעצמך, וניסית להשתמש בה מחוץ לכיתת הלימוד, אז אתה יודע שיש שלושה דברים שאתה חייב להיות בקי בהם: כיצד השפה בנויה (**דקדוק**), כיצד לכנות דברים שאתה רוצה לדבר עליהם (**אוצר מילים**) והדרכים השימושיות והנהוגות לומר דברים יומיומיים (**שימוש**). לרוב, שני המרכיבים הראשונים הם אלה שנלמדים בכיתה ואילו החלק ה**שימושי** קצת מוזנח, ואז אנשים שהשפה שלמדת היא שפתם המדוברת מתאמצים לכבוש את הצחוק שלהם בכל פעם שאתה משוחח אתם.

הדבר נכון מאוד גם לגבי שפת תכנות. אתה צריך להבין את לב השפה: האם היא אלגוריתמית, מבנית-פרוצדורלית, מוכוונת עצמים? אתה צריך גם ללמוד את אוצר המילים: אילו מבני נתונים, פעולות ואמצעים מסופקים על ידי הספריות הסטנדרטיות? ואחרון חביב, אתה חייב גם להיות בקי בדרכים היעילות והמקובלות לבניית הקוד שלך. לעיתים קרובות, ספרים בנושא שפות תכנות מכסים רק את שני הנושאים הראשונים או שהם דנים ב**שימוש** בצורה לא אחידה. אולי הסיבה היא שקל יותר לכתוב על שני הנושאים הראשונים. **דקדוק** ו**אוצר מילים** הם מאפיינים של השפה עצמה אבל **שימוש** הוא מאפיין של הקהילה שמשתמשת בשפה.

שפת התכנות Java, לדוגמה, היא שפה מוכוונת עצמים עם הורשה אנכית (כל מחלקה יכולה לרשת ממחלקה אחת בלבד, בניגוד להורשה רוחבית בה מחלקה יכולה לרשת ממספר מחלקות במקביל) ותומכת בסגנון קוד מונחה הצהרה מתוך כל שיטה. הספריות מציעות תמיכה בתצוגה גרפית, עבודה ברשת, חישוב מבוזר ובטיחות. אבל מהו האופן המעשי הטוב ביותר לשימוש בשפה?

ישנה נקודה נוספת. לכל תוכנית מחשב ישנה סבירות גבוהה שהיא תשונה או תעודכן במשך הזמן. לכן, לא מספיק לייצר קוד שפועל באופן יעיל ושמתכנתים אחרים יכולים להבין אותו היטב. הקוד צריך גם להיות מאורגן באופן כזה שיהיה אפשר לשנות אותו בקלות במרוצת הזמן. יכולות להיות עשר דרכים לכתוב קוד עבור משימה כלשהי.

מתוך עשר הדרכים האלו, שבע תהיינה לא נוחות, לא יעילות או מביכות. נשאלת השאלה: מתוך שלוש הדרכים האחרות, איזו דרך סביר שתהיה דומה לקוד שיבצע את אותה המשימה גם בגרסת התוכנה העתידית?

יש ספרים רבים שיכולים ללמד אותך את ה**דקדוק** של שפת Java, קרא את הספר "**The Java Tutorial סדנת לימוד**" בהוצאת **הוד-עמי** ועיין ברשימה הביבליוגרפית שבסוף הספר. באותו אופן, יש ספרים רבים על הספריות וה-APIs שקשורים לשפת Java.

ספר זה מספק את המרכיב השלישי של הלימוד: **שימוש**. הספר ילמד אותך שימוש יעיל ומקובל בשפת Java. Joshua Bloch עסק שנים בהרחבה, מימוש ושימוש בשפת Java בחברת Sun Microsystems, המפתחת של שפת Java; הוא גם קרא קטעי קוד רבים של אנשים אחרים, כולל קודים שלי. בספר זה הוא מציע עצות טובות שמאורגנות באופן שיטתי בנוגע לכיצד לכתוב את הקודים שלך כך שהם יעבדו כראוי, אנשים אחרים יוכלו להבין אותם, שיפורים ושינויים עתידיים בהם יגרמו לפחות כאבי ראש ואפילו כך שהתוכניות שלך תהיינה נוחות, אלגנטיות ומלאות חן.

Guy L. Steele Jr.

co-author of the book: The Java Language Specification

Burlington, Massachusetts

April 2001

פרק 1

הקדמה

ספר זה נועד לעזור לך לתכנת באופן היעיל ביותר בשפת Java™ ולעשות את השימוש האפקטיבי ביותר בספריות הבסיסיות שלה, java.util, java.lang, ובמידה קטנה יותר ב-java.io. הספר דן גם בספריות אחרות, אבל אין הוא עוסק בתכנות ממשק משתמש גרפי (GUI) או בתכנות ממשק יישומים (API).

ספר זה מורכב מ-57 נושאים, שכל אחד מהם עוסק בכלל יחיד הקשור בתכנות ב-Java. הכללים מתמצתים נוהגי תכנות שנחשבים כמועילים ויעילים על ידי המתכנתים הטובים והמנוסים ביותר. הנושאים מאורגנים בתשעה פרקים, שכל אחד מהם עוסק בהיבט הרחב של עיצוב תוכנה. אינך חייב לקרוא את הספר כולו לפי הסדר, מכיון שכל נושא עומד במידה רבה בפני עצמו. להסברים צורפו התייחסויות רבות, כך שתוכל בקלות לבנות לעצמך מסלול למידה לפי רצונך.

ברוב הנושאים כלולה גם תוכנית דוגמה. תכונה חשובה של הספר היא שהוא מכיל קטעי קוד כדי להסביר **תבניות עיצוב** (design patterns) וקודים יחודיים. חלק מהם מוכרים וידועים, כמו Singleton (נושא 2); חלק מהם חדשים, כמו Finalizer Guardian (נושא 6) ו- Defensive readResolve (נושא 57). אינדקס נפרד מספק גישה נוחה לתבניות וקטעי קוד אלה. בעת הצורך, הם מלווים בהתייחסויות לעבודה הסטנדרטית בתחום זה [Gamma95].

נושאים רבים מתארים דוגמאות קוד שיש להימנע משימוש בהם. דוגמאות אלו, המכונות לעיתים גם **אנטי-תבניות** (antipatterns), מסומנות באופן ברור בהערה כמו "// Never do this!". בכל מקרה, נלווה לכך גם הסבר מדוע אין לנהוג כך, ומוצעת הדרך התקינה לתכנת.

ספר זה אינו מיועד למתחילים: הוא מניח שיש לקורא ידע מוקדם בתכנות בשפת Java. אם אין לך רקע ב-Java, קרא את הספר "**The Java Tutorial סדנת לימוד**" בהוצאת **הוד-עמי** ועיין ברשימה הביבליוגרפית שבסוף הספר. הספר אמנם מיועד לכל מתכנת Java, אך הוא בהחלט יכול לספק חומר למחשבה גם למתכנתים מתקדמים העוסקים בשפות אחרות, כמו C++, למשל.

רוב הכללים בספר נובעים ממספר מועט של עקרונות יסוד. פשטות ובהירות קודמות לכל. המשתמש במודול קוד כלשהו אינו צריך להיות מופתע מההתנהגות שלו.

מודולים צריכים להיות קטנים ככל האפשר, אבל לא קטנים מדי (המונח **מודול**, module, בספר יכול להתייחס לכל רכיב תוכנה הניתן לשימוש חוזר, החל משיטה יחידה וכלה במערכת מורכבת הבנויה ממספר חבילות תוכנה). קוד צריך להיות ניתן לשימוש חוזר יותר מאשר להיות מועתק פעמים אחדות בתוכנית. התלות בין מודולים צריכה להיות מינימלית. יש לאבחן טעויות קרוב ככל האפשר לזמן בו נוצרו, והטוב ביותר - בעת הידור תוכנית המקור.

בעוד שהכללים בספר זה אינם חלים תמיד, הם אכן מובילים ברוב המקרים לתכנות טוב יותר. אין אתה חייב להיצמד לכללים אלה באופן מוחלט, אך תצטרך גם סיבה טובה כדי להתעלם מהם. לימוד תכנות, כמו מיומנויות רבות אחרות, מצריך קודם כל היכרות טובה עם הכללים, ולאחר מכן לימוד המקרים בהם אפשר, ולעיתים אף צריך להתעלם מהם.

ספר זה אינו דן בביצועים (performance). הוא מכוון לכתיבת תוכניות שהן בהירות, נכונות, שימושיות, איתנות, גמישות וניתנות לתחזוקה. אם אתה מסוגל להשיג את כל אלה, תמצא שתוכל להשיג את רמת הביצועים שאתה רוצה בה (נושא 37). חלק מהנושאים בכל זאת דנים בביצועים וחלק מהם אף מספקים נתוני ביצועים. לנתונים אלה, המובאים בצירוף הביטוי "במחשב שלי", יש להתייחס כהערכה בלבד, כיון שנתונים אלה רלוונטיים לתצורת המחשב הספציפי שלי, ואין כל הבטחה לכך שאותן תוצאות יושגו במחשב אחר כלשהו.

לשם השוואה, המחשב שבו עבדתי הוא מחשב "מזדקן" בעל מעבד פנטיום II 400MHz עם 128MB זיכרון, שפועלת בו גרסת SDK 1.3 של Sun, על תשתית Windows NT 4.0 של Microsoft. ב-SDK זה כללתי את Java HotSpot Client VM של Sun, המימוש העדכני של JVM שמיועד לשימוש לקוח (client).

כאשר דנים בתכונות שפת התכנות Java ושימוש בספריות שלה, לעיתים יש צורך להתייחס לגרסאות מסוימות. בקצרה, ספר זה דן בכמה גרסאות, תוך מתן עדיפות לגרסאות הרשמיות. טבלה 1.1 מראה את ההתאמה בין שמות הגרסאות השונות המקובלות:

טבלה 1.1: גרסאות Java

Official Release Name	Engineering Version Number
JDK 1.1.x / JRE 1.1.x	1.1
Java 2 Platform, Standard Edition, v 1.2	1.2
Java 2 Platform, Standard Edition, v 1.3	1.3
Java 2 Platform, Standard Edition, v 1.4	1.4

בעוד שתכונות שהוצגו לראשונה בגרסת 1.4 מתוארות בכמה נושאים, לא השתמשנו בתכונות חדשות אלו בדוגמאות התוכניות, להוציא מספר קטן מאוד של יוצאי דופן. דוגמאות התוכניות נבחנו בגירסה 1.3. מרביתן, אם כי לא כולן, יכולות לפעול ללא שינוי גם בגירסה 1.2.

דוגמאות התוכניות שלמות למדי, אך העדפתי להציגן בצורה שתהיה קריאה וברורה לכל אחד, גם אם פגעתי בשלמות. בדוגמאות הקוד השתמשתי באופן חופשי במחלקות מחבילות התוכנה java.util ו-java.io. כדי להדר דוגמאות אלו, ייתכן שתצטרך להוסיף את אחד ממשפטי הייבוא הבאים, או אף את שניהם:

```
import java.util.*;
import java.io.*;
```

לעיתים השמטתי משפטים מסוג זה ודומיהם, כדי לפשט את קריאת התוכנית. אתר הספר, http://java.sun.com/docs/books/effective מכיל גרסאות מורחבות של כל דוגמה שתוכל להדר ולהריץ. בספר זה השתמשתי במונחים טכניים כפי שהם מוגדרים ב- The Java Language Specification, Second Edition[JLS]. מונחים מסוימים ראויים להתייחסות מיוחדת. השפה תומכת בארבעה סוגי טיפוסי נתונים: **ממשקים** (interfaces), **מחלקות** (classes), **מערכים** (arrays) ו**בסיסיים** (primitives). שלושת הראשונים ידועים כ**טיפוסי ייחוס** (reference types). מופעי מחלקה (class instances) ומערכים (arrays) הם **אובייקטים**; ערכים בסיסיים (primitive values) אינם אובייקטים. **איברי מחלקה** (class's members) מורכבים מ**שדות** (fields), **שיטות** (methods), **מחלקות חברות** (member classes) ו**ממשקים חברים** (member interfaces). **חתימה של שיטה** (method's signature) מורכבת מהשם שלה ומהטיפוסים של הפרמטרים הפורמליים שלה. החתימה **אינה** כוללת את הסוג המוחזר של השיטה.

בספר זה השתמשתי במספר מונחים באופן שונה מהמקובל בספר The Java Language Specification. לדוגמה, השתמשתי במונח **הורשה** (inheritance) כמילה נרדפת ליצירת **תת-מחלקות** (subclassing). במקום להשתמש במונח הורשה לממשקים, הצגתי הסבר ש**מימוש** של ממשק במחלקה יכול להיעשות על ידי **הרחבה** של ממשק אחר. כדי לתאר את **הרשאת ברירת המחדל** (המוחלת כשאיננו מציינים באופן מפורש הרשאה אחרת), השתמשתי במונח התיאורי **הרשאת פרטית בחבילה** (package-private) במקום במונח הרשמי **הרשאת ברירת מחדל** (dafault access [JLS, 6.6.1]).

בספר השתמשתי במספר קטן של מונחים שאינם מוגדרים בספר The Java Language Specification. המונח **API מיובא** (imported API) או בקיצור **API** מתייחס למחלקות, ממשקים, בנאים, איברים וטפסים מסודרים (serialized forms), המאפשרים למתכנת לגשת למחלקה, ממשק או חבילת תוכנה. המונח API (**A**pplication **P**rogramming **I**nterface, ממשק תכנות יישומים) משמש כאן במקום המונח **ממשק** בלבד, כדי להימנע מבלבול עם המונח הדומה בשפה, אך השונה ממנו במשמעות. המתכנת הכותב תוכנית שמשתמשת ב-API, מתואר כ**משתמש** ב-API. מחלקה שהמימוש שלה משתמש ב-API מתוארת כ**לקוחה** (client) של ה-API.

מחלקות, ממשקים, בנאים, איברים וטפסים מסודרים, ידועים כולם כ**אלמנטים של API**. API מיובא מורכב מאלמנטים של API הנגישים מחוץ לחבילת התוכנה המגדירה את ה-API. אלה אלמנטים של API שכל לקוח יכול להשתמש בהם, ומי שיוצר API אמור לתמוך בהם. לא במקרה, הם גם האלמנטים שלמענם שגרת Javadoc יוצרת תיעוד כמצב ברירת המחדל שלה. למעשה, ה-API המיובא של חבילת תוכנה מורכב מהאיברים הציבוריים והמוגנים, ומהבנאים של כל מחלקה ציבורית או ממשק ציבורי שבחבילה.

פרק 2

יצירה ופירוק של אובייקטים

בפרק זה אתייחס לשאלות הקשורות ליצירה ופירוק של אובייקטים: מתי וכיצד ליצור אובייקטים, מתי וכיצד להימנע מיצירת אובייקטים, כיצד להבטיח שאובייקטים יפורקו בעיתוי הנכון וכיצד לשלוט בפעולות "ניקוי" אשר חייבות להתבצע לפני פירוק אובייקטים.

נושא 1: שקול שימוש בשיטות יוצרות סטטיות במקום בבנאים

הדרך המקובלת שבה מחלקה מאפשרת למשתמש לקבל **מופע** (instance), היא באמצעות **בנאי ציבורי** (public constructor). יש טכניקה אחרת, פחות ידועה, שכדאי שתהיה חלק מערכת כלי הפיתוח של כל מתכנת. מחלקה יכולה לספק **שיטה יוצרת ציבורית סטטית** (public static factory method), שלמעשה היא שיטה סטטית המחזירה מופע של המחלקה. להלן דוגמה פשוטה של המחלקה Boolean (המחלקה העוטפת את הטיפוס הפרימיטיבי boolean). שיטה יוצרת זו, אשר התווספה לגרסת 1.4, ממירה ערך בוליאני בסיסי ל**אובייקט ייחוס בוליאני** (boolean object reference):

```
public static Boolean valueOf(boolean b) {
  return (b ? Boolean.TRUE : Boolean.FALSE);
}
```

מחלקה יכולה לספק למשתמשים בה שיטות יוצרות סטטיות במקום, או בנוסף, לבנאים. לשימוש בשיטות יוצרות סטטיות במקום בנאים ציבוריים יש יתרונות, כמו גם חסרונות אחדים.

יתרון אחד של שיטות יוצרות סטטיות הוא שיש להן שמות, בניגוד לבנאים שאין להם שמות. אם הפרמטרים הנשלחים לבנאי אינם מתארים כשלעצמם את האובייקט המוחזר מהבנאי, הרי ששיטה יוצרת סטטית בעלת שם שנבחר בקפידה, תקל על השימוש במחלקה והקוד יהיה קריא יותר. לדוגמה, BigInteger(int, int, Random) הוא בנאי המחזיר BigInteger, שקרוב לוודאי הוא ערך ראשוני.

טוב שבנאי כזה יהיה מוגדר כשיטה יוצרת סטטית בשם BigInteger.probablePrime (שיטה יוצרת סטטית זו הוכנסה לבסוף לגירסה 1.4).

מחלקה יכולה לכלול בנאי יחיד בלבד, בעל חתימה נתונה. מתכנתים נוהגים לעקוף מגבלה זו על ידי שימוש בשני בנאים שכל ההבדל ביניהם הוא הסדר השונה שבו רשומים טיפוסי הארגומנטים שהם מקבלים. זהו נוהג רע. המשתמש ב-API כזה לעולם לא יהיה מסוגל לזכור לאיזה בנאי הוא התכוון לפנות, ובסופו של דבר ישגה ויפנה לבנאי הלא-מתאים. קוראים שיעיינו בקוד תוכנית כזה, לא ידעו מה הקוד מבצע ללא עיון בתיעוד המחלקה.

מכיון שלשיטות יוצרות סטטיות יש שמות, הן אינן כפופות למגבלה שלפיה למחלקה יכול להיות רק בנאי יחיד נתון עם חתימה ייחודית שלו. במקרים בהם נראה שדרושים למחלקה מספר בנאים בעלי חתימה (שם, בעיקר) זהה, כדאי לשקול החלפת בנאי אחד או יותר בשיטות יוצרות סטטיות ששמותיהן, שנבחרו בקפידה, מבליטים את ההבדלים ביניהן.

יתרון שני של שיטות יוצרות סטטיות הוא שבניגוד לבנאים, הן אינן צריכות ליצור אובייקט חדש בכל פעם שהן נקראות. הדבר מאפשר למחלקות בלתי משתנות (immutable classes) (נושא 13), להשתמש במופעים בנויים מראש, או להסתיר מופעים ברגע שהם נבנים ולסלק מופעים אלה שוב ושוב, כמו גם להימנע מיצירת עותקים לא רצויים של אובייקטים. השיטה Boolean.valueOf(boolean), מדגימה טכניקה זו: היא לעולם אינה יוצרת אובייקט. טכניקה זו יכולה לשפר ביצועים באופן ניכר כאשר צריך ליצור אובייקטים שווי ערך לעיתים קרובות, במיוחד אם יצירת אובייקטים אלה גוזלת משאבים רבים, כמו זמן מעבד וזיכרון.

יכולת זו של שיטות היצירה הסטטיות להחזיר את אותו האובייקט כתוצאה מקריאות חוזרות, יכולה גם לשמש לשמירה קפדנית על קיום או אי-קיום של מופעים שונים בכל רגע נתון. יש שתי סיבות לצורך זה. ראשית, הדבר מאפשר למחלקה להבטיח את הייחודיות שלה (נושא 2). שנית, מחלקה שאינה ניתנת לשינוי יכולה להבטיח ששני מופעים זהים לעולם לא יתקיימו בעת ובעונה אחת: אפשר לכתוב a.equals(b) אם, ורק אם, a==b. אם מחלקה מבטיחה זאת, אז המשתמשים בה יכולים להשתמש באופרטור == ולא בשיטה equals(Object) - מה שיגרום לשיפור ניכר בביצועים. מימוש של שיפור זה מוצג על ידי התבנית typesafe enum המתוארת בנושא 21, ועל ידי השיטה String.intern באופן מצומצם יותר.

יתרון שלישי של שיטות היצירה הסטטיות הוא שבניגוד לבנאים, הן יכולות להחזיר אובייקט מכל תת-טיפוס של טיפוס הערך המוחזר שלהן. הדבר מעניק גמישות רבה בבחירת המחלקה של האובייקט המוחזר.

יישום אחד של גמישות זו מתבטא בכך שה-API יכול להחזיר אובייקטים ללא הפיכת המחלקות שלהם לציבוריות. הסתרת מחלקות מימוש בדרך זו יכולה ליצור API מאוד קומפקטי. טכניקה זו מתאימה לממשק מבוסס סביבות עבודה (frameworks), שבהן הממשקים מספקים טיפוסים חוזרים טבעיים (natural return types), לשיטות היצירה הסטטיות.

לדוגמה, לסביבת האוספים (collections Framework), יש עשרים מימושים נוחים של אוסף הממשקים שלה, המספקים אוספים בלתי ניתנים לשינוי (immutable), אוספים מסונכרנים וכדומה. הרוב הגדול של יישומים אלה מיוצא באמצעות שיטות יוצרות סטטיות במחלקה בודדת אשר לא ניתן ליצור מופעים ישירות ממנה (noninstantiable): java.util.Collections. המחלקות של האובייקטים המוחזרים הן כולן לא-ציבוריות.

ה-API של סביבת האוספים הוא קטן הרבה יותר מאשר אם היה מייצא עשרים מחלקות ציבוריות נפרדות עבור המימושים הנוחים. לא רק שנפח ה-API קטן, אלא גם ה"משקל המושגי" שלו. המשתמש יודע שלאובייקט המוחזר יש בדיוק את ה-API שצוין על ידי הממשק הרלוונטי, כך שאין צורך לקרוא חומר תיעוד נוסף על המחלקות. יתר על כן, שימוש בשיטה יוצרת סטטית שכזו גורמת למשתמש להתייחס לאובייקט המוחזר לפי הממשק שלו, יותר מאשר לפי המחלקה המממשת אותו - נוהג שהוא בדרך כלל טוב (נושא 34).

לא רק שהמחלקה של אובייקט שמוחזר על ידי שיטה יוצרת סטטית ציבורית יכולה להיות לא ציבורית, אלא שהמחלקה יכולה להשתנות מקריאה לקריאה, בהתאם לערכים של הפרמטרים של היצירה הסטטית. כל מחלקה שהיא תת-טיפוס של הסוג המוחזר המוצהר, מותרת. המחלקה של האובייקט המוחזר יכולה גם להשתנות מגירסה לגירסה, כדי לאפשר אחזקה טובה יותר של התוכנה.

המחלקה של האובייקט המוחזר על ידי השיטה היוצרת הסטטית, אפילו לא חייבת להיות קיימת בזמן שהמחלקה המכילה את השיטה היוצרת הסטטית נכתבת.
שיטות יוצרות סטטיות גמישות כאלו הן הבסיס של **סביבת עבודה של ספקי שירות** (service provider framework) כמו ה-JCE (Java Cryptography Extension). סביבת עבודה של ספק שירות היא מערכת שבה ספקים גורמים ליישומים כפולים של API להיות זמינים למשתמשים מרובים של סביבת העבודה. לרישום היישומים האלה מסופק מנגנון, ההופך אותם זמינים לשימוש. לקוחות של סביבת העבודה משתמשים ב-API בלי לדאוג למימוש שהם עוסקים בו.

ב-JCE, מנהל המערכת משלב מחלקת מימוש על ידי עריכת קובץ מאפיינים מוגדר, תוך הוספת כניסה שממפה מחרוזת מפתח אל שם המחלקה המתאימה. לקוחות משתמשים בשיטה היוצרת הסטטית שלוקחת את המפתח כפרמטר. השיטה היוצרת הסטטית מחפשת את מחלקת האובייקט במפה שמאותחלת מקובץ המאפיינים וקוראת למחלקה, תוך שימוש בשיטה Class.newInstance.

המימוש הבא מתאר דוגמה של טכניקה זו:

```
// Provider framework sketch
public abstract class Foo {
   // Maps String key to corresponding Class object
   private static Map implementations = null;

   // Initializes implementations map the first time it's called
   private static synchronized void initMapIfNecessary() {
      if (implementations == null) {
         implementations = new HashMap();

         // Load implementation class names and keys from
         // Properties file, translate names into Class
         // objects using Class.forName and store mappings.
          ...
      }
   }

   public static Foo getInstance(String key) {
      initMapIfNecessary();
      Class c = (Class) implementations.get(key);
      if (c == null)
         return new DefaultFoo();

      try {
         return (Foo) c.newInstance();
      } catch (Exception e) {
         return new DefaultFoo();
      }
   }
}
```

החיסרון העיקרי של שיטות היצירה הסטטיות הוא בכך שלא ניתן לרשת (ליצור תת-מחלקה) מתוך מחלקה שאינה מכילה בנאים ציבוריים או מוגנים. הדבר נכון גם לגבי מחלקות לא ציבוריות המוחזרות על ידי "יצרנים" סטטיים ציבוריים. לדוגמה, בלתי אפשרי להגדיר תת-מחלקות אף לא לאחת ממחלקות המימוש הנוחות שבסביבת האוספים. אולם, חיסרון זה נתון לוויכוח, כיון שבאופן מוסווה, הדבר מעודד מתכנתים להשתמש **בהרכבה** (composition) ולא **בהורשה** (inheritance) (נושא 14).

חיסרון שני של שיטות היצירה הסטטיות הוא בכך שקשה להבחין ביניהן לבין שיטות סטטיות אחרות. הן אינן בולטות בתיעוד של ה-API כמו הבנאים. יתר על כן, שיטות יוצרות סטטיות מציעות חריגה מהנורמה המקובלת.

לכן, ייתכן שיהיה קשה להבין מתוך התיעוד של המחלקה כיצד להדגים מחלקה המספקת שיטות יוצרות סטטיות במקום בנאים. אפשר לצמצם חיסרון זה באמצעות הקפדה על מוסכמות למתן שמות סטנדרטיים. מוסכמות אלו עדיין בתהליך הגדרה, אבל שני שמות לשיטות יוצרות סטטיות הולכים ונעשים מקובלים:

- valueOf - השיטה מחזירה מופע שיש לו את אותו הערך כמו הפרמטרים שלו. שיטות יוצרות סטטיות בעלות שם זה הן אופרטורים יעילים להמרת טיפוסים.
- getInstance - השיטה מחזירה מופע שמתואר על ידי הפרמטרים שלו, אבל אי אפשר לומר שיש לו את אותו הערך. במקרה של singleton (מחלקה שיכול להיות לה מופע אחד בלבד), השיטה מחזירה את המופע הבסיסי היחיד. שם זה מקובל בסביבות עבודה של ספקים.

לסיכום, לשיטות יוצרות סטטיות ולבנאים ציבוריים יש תחומי שימוש שונים ורצוי להבין את יתרונותיהם היחסיים. הימנע מהדחף הטבעי לספק בנאים, בלי לשקול תחילה שימוש בשיטות יוצרות סטטיות, מכיון ששיטות כאלו עשויות להיות מתאימות יותר במקרים רבים. אם שקלת את שתי הדרכים ולא מצאת שאחת מהן עדיפה על חברתה, ייתכן שעדיף שתשתמש בבנאי מכיון שזו הנורמה המקובלת.

נושא 2: אכיפת תכונת singleton בעזרת בנאי פרטי

singleton היא למעשה מחלקה שיכול להיות לה מופע אחד בלבד [Gamma95]. בדרך כלל, מחלקות מסוג singleton מייצגות מרכיבי מערכת יחודיים באופן מהותי, כמו תצוגת וידאו או מערכת קבצים.

יש שתי גישות למימוש מחלקות singleton. שתיהן מבוססות על הגדרת הבנאי כפרטי ויצירת איבר סטטי סופי, כדי לאפשר ללקוח גישה אל המופע היחיד של המחלקה. לפי גישה אחת, האיבר הסטטי הסופי הוא בעל הרשאה ציבורית:

```
// Singleton with final field
public class Elvis {
   public static final Elvis INSTANCE = new Elvis();
private Elvis() {
        ...
   }
    ...  // Remainder omitted
}
```

הבנאי הפרטי נקרא פעם אחת בלבד, כדי לאתחל את השדה הסטטי הציבורי הסופי Elvis.INSTANCE. המחסור בבנאים ציבוריים או מוגנים מבטיח עולם "מונואלביסטי" ('monoelvistic' universe): ברגע שהמחלקה Elvis מאותחלת, נוצר מופע אחד בלבד של Elvis, לא פחות ולא יותר. הלקוח לא יכול לעשות דבר כדי לשנות זאת.

לפי גישה שנייה, השדה הסטטי הסופי מקבל הרשאה פרטית, והגישה אליו מתבצעת בעזרת שיטה יוצרת סטטית ציבורית:

```
// Singleton with static factory
public class Elvis {
   private static final Elvis INSTANCE = new Elvis();

   private Elvis() {
        ...
   }

   public static Elvis getInstance() {
      return INSTANCE;
   }

   ...  // Remainder omitted
}
```

כל הקריאות לשיטה הסטטית Elvis.getInstance, יחזירו את אותו ייחוס לאובייקט, ושום מופע אחר של Elvis לא ייווצר לעולם.

היתרון העיקרי של הגישה הראשונה הוא בכך שההצהרה על האיברים הכלולים במחלקה מבהירה שהמחלקה היא מסוג singleton: השדה הסטטי ציבורי הוא סופי. לכן, השדה תמיד יכלול את אותו ייחוס לאובייקט. ייתכן גם יתרון קטן בביצועים של הגישה הראשונה, אבל מימוש טוב של JVM אמור להתגבר על כך על ידי הכללה בקוד (inline) של הקריאה לשיטה היוצרת הסטטית, בגישה השנייה.

היתרון העיקרי של הגישה השנייה הוא בגמישות שמתקבלת, כדי שהמתכנת יוכל לשנות את דעתו אם הוא רוצה שהמחלקה תהיה singleton, בלי לשנות את ה-API. השיטה היוצרת הסטטית עבור singleton מחזירה את המופע היחיד האפשרי של המחלקה, אבל היא יכולה בקלות לעבור שינוי, כך שתחזיר מופע יחיד לכל מטלה אשר קוראת לשיטה.

אם כן, הגיוני יותר להשתמש בגישה הראשונה אם אתה בטוח לחלוטין שהמחלקה תמיד תישאר מסוג singleton, שניתן להפיק ממנה מופע אחד בלבד. השתמש בגישה השנייה אם אתה רוצה להשאיר למתכנת שיקול דעת בנושא.

כדי להפוך מחלקת singleton ל-serializable (פרק 10), כך שניתן יהיה לשלוח מופעים שלה בזרמים (Streams), אין זה מספיק רק להוסיף את המילים implements Serializable להצהרה שלה. כדי להבטיח מצב singleton, עליך גם לספק את השיטה ReadResolve. אחרת, בכל פעם שנבצע קריאה מתוך הזרם של מופע שעבר פירוק לצורך העברה טורית בזרם, תבוצע הרכבה לאובייקט מופע חדש, דבר הנוגד את עיקרון ה-singleton שמטרתו למנוע יצירה של מופעים חדשים למחלקה.

בדוגמה שלנו, נוסיף את השיטה readResolve למחלקה Elvis, כדי למנוע יצירת "זיופי Elvis" בעת הקריאה מתוך הזרם.

```
// readResolve method to preserve singleton property
  private Object readResolve() throws ObjectStreamException {
    /*
     * Return the one true Elvis and let the garbage collector
     * take care of the Elvis impersonator.
     */
    return INSTANCE;
  }
```

דבר משותף נמצא ביסודו של נושא זה ונושא 21 המתאר תבנית מנייה מובטחת-סוג (typesafe enum). בשני המקרים, בנאים פרטיים משמשים בצירוף עם איברים סטטיים ציבוריים, כדי להבטיח ששום מופע חדש של המחלקה הרלוונטית לא ייווצר אחרי שהוא מאותחל. במקרה של נושא זה, נוצר רק מופע יחיד של המחלקה; בנושא 21, מופע אחד נוצר לכל איבר של הסוג הנספר. בנושא הבא (נושא 3), גישה זו מביאה אותנו צעד אחד קדימה: היעדר בנאי ציבורי משמש כדי להבטיח שבשום אופן לא ייווצר מופע כלשהו של מחלקה.

נושא 3: מנע באמצעות בנאי פרטי את היכולת ליצור מופע

לעיתים תרצה ליצור מחלקה שהיא למעשה קבוצה של שיטות סטטיות ושדות סטטיים. למחלקות כאלו יצא שם רע, מכיון שמתכנתים אחדים השתמשו בהן לרעה לכתיבת תוכניות פרוצדורליות בשפות מוכוונות עצמים, אבל יש להן גם שימושים טובים. הן יכולות לשמש לקיבוץ שיטות קרובות על ערכים פשוטים או מערכים, כמו למשל java.lang.Math או java.util.Arrays או לקיבוץ שיטות סטטיות עבור אובייקטים שיש בהם ממשק ייחודי, באופן של java.util.Collections. הן גם יכולות לשמש לקיבוץ שיטות על מחלקה סופית, במקום להגדיל את המחלקה.

מחלקות עזרים כאלו לא תוכננו להיות מופעים: המופע במקרה זה יהיה אבסורדי. בהיעדר של בנאים מפורשים, המהדר מספק בכל מקרה בנאי ברירת מחדל ציבורי וחסר פרמטרים. מבחינת המשתמש, אי אפשר להבדיל בין בנאי זה לבין בנאים אחרים. אין זה נדיר לראות מופעים של מחלקות שהופקו בשוגג ב-API.

הניסיון למנוע יצירת מופעים על ידי הפיכת מחלקה למופשטת לא יצליח. ניתן ליצור תת-מחלקות של מחלקה, שתוחזרנה כמופע. מלבד זאת, הדבר מטעה את המשתמש לחשוב שהמחלקה עוצבה לצורך הורשה (נושא 15). בכל אופן, יש ביטוי פשוט כדי להבטיח שלא ייווצר מופע נוסף. בנאי ברירת מחדל נוצר רק אם המחלקה אינה מכילה בנאים מפורשים, **ולכן אפשר למנוע ממחלקה יצירה של מופע נוסף על ידי הכללת בנאי פרטי מפורש יחיד**:

```
// Noninstantiable utility class
public class UtilityClass {

   // Suppress default constructor for noninstantiability
   private UtilityClass() {
      // This constructor will never be invoked
   }

   ...  // Remainder omitted
}
```

מכיון שהבנאי המפורש הוא פרטי, הוא אינו נגיש מחוץ למחלקה. לכן מובטח שלעולם לא ייווצר מופע של המחלקה, בהנחה שהבנאי לא נקרא מתוך המחלקה עצמה. הביטוי אינו אינטואיטיבי, מכיון שהבנאי מסופק במפורש באופן שאי אפשר יהיה לקרוא לו. לכן, יהיה זה נבון לכלול הערה המתארת את מטרת הבנאי.

כתופעת לוואי, הביטוי גם מונע יצירת תת-מחלקות של המחלקה שבה עוסקים. כל הבנאים חייבים לקרוא לבנאי נגיש של מחלקת העל, באופן מרומז או מפורש. במקרה שלנו, אם נגדיר למחלקה בה אנו עוסקים תת-מחלקה, לבנאי של תת-המחלקה לא יהיה בנאי אב נגיש שאפשר לקרוא לו.

נושא 4: הימנע מיצירת אובייקטים כפולים

לעיתים קרובות, מתאים יותר להשתמש שוב באובייקט יחיד במקום ליצור אובייקט חדש ובעל תפקוד שווה ערך, בכל פעם שיש בכך צורך. השימוש החוזר מהיר יותר ומסוגנן יותר. אפשר לעשות שימוש חוזר באובייקט אם הוא **שאינו ניתן לשינוי** (immutable) (נושא 13). כדוגמה קיצונית לפעולה שאסור לעשות, בחן את שורת הקוד הזו:

```
String s = new String("silly");      // DON'T DO THIS !
```

שורת הקוד יוצרת מופע חדש של String בכל פעם שהיא מבוצעת, וכל אחת מיצירות האובייקט הזה היא מיותרת. הארגומנט ("silly") לבנאי String הוא בעצמו מופע של String, בעל תפקוד שווה ערך לכל אחד מהאובייקטים שנוצרו על ידי הבנאי. אם הקריאה לבנאי נמצאת בלולאה או בשיטה שנקראת בתדירות גבוהה, מיליוני מופעי String עלולים להיווצר ללא צורך.

שורת קוד משופרת תיראה כך:

```
String s = "No longer silly";
```

גירסה שונה זו משתמשת במופע String יחיד, במקום ליצור מופע חדש בכל פעם שהיא מבוצעת. יתרה מזאת, מובטח שייעשה באובייקט שימוש חוזר על ידי כל קוד אחר שמורץ באותה מכונה וירטואלית שמכילה את אותו ביטוי מחרוזת [JLS, 3.10.5].

לעיתים קרובות, תוכל להימנע מיצירת אובייקטים כפולים על ידי שימוש ב**שיטות יוצרות סטטיות** (static factory methods) (נושא 1) ולא בבנאים, במחלקות בלתי משתנות שמספקות את שניהם.

לדוגמה, השיטה היוצרת הסטטית Boolean.valueOf(String) מועדפת כמעט תמיד על פני הבנאי Boolean(String). הבנאי יוצר אובייקט חדש בכל פעם שהוא נקרא, בעוד שהשיטה היוצרת הסטטית אינה צריכה לעשות זאת אף פעם.

בנוסף לשימוש החוזר באובייקטים הבלתי משתנים (immutable objects), תוכל גם להשתמש באופן חוזר באובייקטים משתנים (mutable) שאתה יודע שלא תשנה אותם. להלן דוגמה מעט יותר עדינה והרבה יותר נפוצה על מה שלא רצוי לעשות. הדוגמה כוללת אובייקטים משתנים שאינם משתנים יותר לאחר שערכיהם כבר חושבו:

```
public class Person {
   private final Date birthDate;
   // Other fields omitted

   public Person(Date birthDate) {
      this.birthDate = birthDate;
   }
```

```
    // DON'T DO THIS!
    public boolean isBabyBoomer() {
        Calendar gmtCal =
            Calendar.getInstance(TimeZone.getTimeZone("GMT"));
        gmtCal.set(1946, Calendar.JANUARY, 1, 0, 0, 0);
        Date boomStart = gmtCal.getTime();
        gmtCal.set(1965, Calendar.JANUARY, 1, 0, 0, 0);
        Date boomEnd = gmtCal.getTime();
        return birthDate.compareTo(boomStart) >= 0 &&
               birthDate.compareTo(boomEnd)   <  0;
    }
}
```

השיטה isBabyBoomer יוצרת שלא לצורך מופעי TimeZone, Calendar ושני מופעי Date חדשים בכל פעם שהיא נקראת. הגירסה שלהלן נמנעת מחוסר יעילות זה על ידי שימוש במאתחל סטטי:

```
class Person {
    private final Date birthDate;

    public Person(Date birthDate) {
        this.birthDate = birthDate;
    }

    /**
     * The starting and ending dates of the baby boom.
     */
    private static final Date BOOM_START;
    private static final Date BOOM_END;

    static {
        Calendar gmtCal =
            Calendar.getInstance(TimeZone.getTimeZone("GMT"));
        gmtCal.set(1946, Calendar.JANUARY, 1, 0, 0, 0);
        BOOM_START = gmtCal.getTime();
        gmtCal.set(1965, Calendar.JANUARY, 1, 0, 0, 0);
        BOOM_END = gmtCal.getTime();
    }

    public boolean isBabyBoomer() {
        return birthDate.compareTo(BOOM_START) >= 0 &&
               birthDate.compareTo(BOOM_END)   <  0;
    }
}
```

הגירסה המשופרת של המחלקה Person יוצרת מופעי Calendar, TimeZone ו-Date פעם אחת בלבד, כאשר היא מאותחלת, במקום ליצור אותם בכל פעם ש-isBabyBoomer נקראת. הדבר מביא לשיפור משמעותי בביצועים, כאשר השיטה נקראת בתדירות גבוהה. במחשב שלי, דרושים 36,000 מילי-שניות להרצת הגירסה המקורית על כל מיליון קריאות, בעוד שלהרצת הגירסה המשופרת דרושים 370 מילי-שניות - זמן מהיר פי 100. לא רק שהביצועים השתפרו, אלא גם בהירות הקוד. השינוי של boomStart ו-boomEnd ממשתנים מקומיים לשדות סטטיים סופיים מבהיר שההתייחסות לתאריכים אלה היא כאל קבועים, דבר שעושה את הקוד למובן יותר. למען האמת, החיסכון שמושג על ידי סוג כזה של שיפור לא יהיה תמיד כל כך קיצוני, כיון שמופעי Calendar הם יקרים ליצירה באופן מיוחד.

אם לעולם לא תתבצע קריאה לשיטה isBabyBoomer, הגירסה המשופרת של המחלקה Person תאתחל את השדות BOOM_START ו-BOOM_END ללא צורך. אפשר למנוע את האתחולים המיותרים על ידי **אתחול עצל** (lazily initializing) של שדות אלה (נושא 48) בפעם הראשונה שהשיטה isBabyBoomer נקראת, אך הדבר אינו מומלץ. כמו שלעיתים קרובות קורה באתחול עצל, תגרום לסיבוך היישום, ולא סביר שתשיג שיפור בולט בביצועים (נושא 37).

בכל הדוגמאות הקודמות שהופיעו בנושא זה, היה ברור שאפשר לעשות באובייקטים שימוש חוזר, מכיון שהם אינם משתנים. יש מצבים אחרים, בהם זה פחות מובן מאליו. חשוב על המקרה של **מתאמים** (adapters) [Gamma95] הידועים גם בשם **תצוגות** (views). מתאם הוא אובייקט שסמוך לאובייקט נתמך ומספק ממשק חילופי לאובייקט הנתמך. מכיון שלמתאם אין תפקיד נוסף, פרט לסמיכותו לאובייקט נתמך, אין כל צורך ליצור יותר ממופע אחד של מתאם עבור אובייקט נתון.

לדוגמה, השיטה keySet של הממשק Map מחזירה מתאם Set של האובייקט Map, שמורכב מכל המפתחות במפה. עלול להיווצר הרושם המוטעה שכל קריאה ל-keySet תצטרך ליצור מופע חדש של Set, אבל כל קריאה ל-keySet באובייקט Map נתון, אמורה להחזיר את אותו מופע של Set. למרות שהמופע המוחזר של Set עשוי להשתנות בדרך כלל, כל האובייקטים המוחזרים הם שווי ערך בתפקודם: אם אובייקט מוחזר אחד ניתן לשינוי, כל השאר ישתנו גם הם, כי כולם מגובים על ידי אותו מופע של Map.

אין להבין בטעות שנושא זה רומז כי יצירת אובייקטים היא יקרה ויש להימנע ממנה. להיפך, היצירה וההשבחה של אובייקטים קטנים שהבנאים שלהם עושים מעט עבודה מפורשת היא זולה, במיוחד ביישומי JVM מודרניים. יצירת אובייקטים נוספים כדי להגדיל את הבהירות, הפשטות או העוצמה של תוכנית היא בדרך כלל דבר טוב.

דווקא הימנעות מיצירת אובייקטים על ידי שמירת **אוסף אובייקטים** (object pool) משלך היא רעיון רע, אלא אם כן האובייקטים באוסף הם כבדים באופן קיצוני. דוגמה טיפוסית לאובייקט שמצדיק אוסף אובייקטים היא קישור למסד נתונים. המחיר של יצירת הקישור גבוה דיו כדי להצדיק שימוש חוזר באובייקטים אלה. בכל מקרה, ככלל, שמירת אוסף האובייקטים שלך יוצרת קוד מבלבל, מגדילה את תפוסת הזיכרון שלו ופוגעת בביצועים. ליישומי JVM מודרניים יש אספני אשפה יעילים מאוד שהופכים בקלות אוספי אובייקטים כאלה לאובייקטים קלי משקל.

הנושא המנוגד לנושא זה הוא נושא 24 העוסק ב**העתקה הגנתית** (defensive copying). הנושא הנוכחי אומר: "אל תיצור אובייקט חדש, כאשר באפשרותך לעשות שימוש חוזר באובייקט קיים", בעוד שנושא 24 אומר: "אל תעשה שימוש חוזר באובייקט קיים, כאשר אתה יכול ליצור אובייקט חדש". שים לב לכך שה"עונש" על שימוש חוזר באובייקט, כאשר נעשה שימוש בהעתקה הגנתית, הוא גדול בהרבה מה"עונש" על יצירת אובייקט כפול שלא לצורך. כישלון ביצירת עותקים הגנתיים, כאשר צריך לעשות זאת, יכול להוביל לבאגים נסתרים ולפרצות באבטחה. יצירת אובייקטים כפולים שלא לצורך רק תשפיע על הביצועים והעיצוב.

נושא 5: סלק ייחוסים ישנים לאובייקטים

כאשר אתה עובר משימוש בשפה עם טיפול ידני בזיכרון, כמו C או C++, אל שפה שאוספת אשפה בעצמה, עבודתך כמתכנת קלה הרבה יותר, בשל העובדה שהאובייקטים שלך משתחררים באופן אוטומטי לאחר סיום השימוש בהם. הדבר נראה כמו קסם כאשר אתה חווה זאת בפעם הראשונה, ואתה עלול בקלות לקבל את הרושם שאינך צריך להקדיש מחשבה לטיפול בזיכרון, אבל אין זה נכון לגמרי.

בחן את מימוש המחסנית הפשוט הבא:

```
// Can you spot the "memory leak"?
public class Stack {
    private Object[] elements;
    private int size = 0;

    public Stack(int initialCapacity) {
        this.elements = new Object[initialCapacity];
    }

    public void push(Object e) {
        ensureCapacity();
        elements[size++] = e;
    }

    public Object pop() {
        if (size==0)
            throw new EmptyStackException();
        return = elements[--size];
        Object result = elements[--size];
        elements[size] = null; // Eliminate obsolete reference
        return result;
    }

    /**
     * Ensure space for at least one more element, roughly
     * doubling the capacity each time the array needs to grow.
     */
    private void ensureCapacity() {
        if (elements.length == size) {
            Object[] oldElements = elements;
            elements = new Object[2 * elements.length + 1];
            System.arraycopy(oldElements, 0, elements, 0, size);
        }
    }
}
```

בתוכנית זו אין כל שגיאה הנראית לעין. תוכל לבחון אותה באופן ממצה והיא תעבור כל מבחן בקלות, אבל יש בה בעיה סמויה.

אפשר לומר שלתוכנית יש בעיה של "דליפת זיכרון", שיכולה להתבטא בביצועים ירודים, בשל פעילות מוגברת של אספן האשפה או שימוש מוגבר בזיכרון. במקרים קיצוניים, דליפות זיכרון כאלו יכולות לגרום לגדישת הדיסק או אפילו לנפילת התוכנית, תוך הצגת ההודעה OutOfMemoryError - אבל נפילות כאלו הן נדירות באופן יחסי.

אם כן, היכן היא דליפת הזיכרון? אם מחסנית גדלה ואחר כך מתכווצת, האובייקטים שסולקו ממנה לא ייאספו על ידי אספן האשפה, אפילו אם התוכנית המשתמשת במחסנית אינה זקוקה להם עוד ואינה מתייחסת אליהם. הדבר נובע מכך שהמחסנית שומרת **התייחסויות ישנות** (obsolete references) או מבוטלות שפנו אל אובייקטים אלה. התייחסות ישנה היא, למעשה, התייחסות שלעולם לא יהיה בה שימוש נוסף. במקרה שלנו, כל ההתייחסויות מחוץ לחלק הפעיל (האלמנטים שהאינדקס שלהם הוא פחות מ-size) של המערך הן ישנות ולא ניתן יהיה לבצע בהם שימוש נוסף.

דליפות זיכרון בשפות אוספות אשפה (או יותר נכון, שפות שידועות בכינוי "שומרות אובייקטים בטעות", unintentional object retentions) הן סמויות. אם ייחוס לאובייקט נשמר שלא בכוונה, לא רק שהאובייקט אינו נכלל באוסף האשפה, אלא גם כל האובייקטים שיוחסו על ידי אובייקט זה וכן הלאה. אפילו אם רק כמה ייחוסי אובייקטים נשמרים שלא בכוונה, אובייקטים רבים עלולים שלא להיאסף על ידי אוסף האשפה, עם השפעה אפשרית גדולה על הביצועים.

הפתרון לבעיה זו פשוט למדי: **סלק** (null out) התייחסויות ברגע בו הן מתיישנות ואינן דרושות עוד. במקרה של המחלקה Stack שלנו, ההתייחסות לפריט מתיישנת ברגע שהוא נשלף מהמחסנית. הגירסה המתוקנת של השיטה pop תיראה כך:

```
public Object pop() {
    if (size==0)
        throw new EmptyStackException();
    Object result = elements[--size];
    elements[size] = null; // Eliminate obsolete reference
    return result;
}
```

יתרון נלווה לסילוק התייחסויות ישנות הוא שאם בוטל הייחוס שלהן לאחר מכן בטעות, התוכנית תסתיים מיידית עם החריג NullPointerException, במקום שתעשה את הדבר השגוי. דרך פעולה זו תמיד מועילה לזיהוי טעויות תכנות מהר ככל האפשר.

כאשר מתכנתים נתקלים לראשונה בבעיה כזו, הם נוטים להגזים על ידי סילוק כל ייחוס לאובייקט ברגע שלתוכנית אין עוד עניין בו. הדבר אינו דרוש ואף אינו רצוי מכיון שזה משבש את התוכנית שלא לצורך ויכול בהחלט להפחית ביצועים. סילוק התייחסויות לאובייקטים צריך להיות נוהג חריג, ולא דרך פעולה מקובלת.

הדרך הטובה ביותר לסלק ייחוס ישן היא להשתמש שוב במשתנה שבו הוא היה כלול, או לתת לו להשתחרר אל מחוץ לטווח ההכרה. הדבר יתרחש באופן טבעי אם תגדיר כל משתנה בטווח ההכרה המצומצם ביותר האפשרי (נושא 29). ראוי לציין שביישומי JVM של היום אין זה מספיק סתם לצאת מהבלוק בו משתנה הוגדר, חובה לצאת מהשיטה הכוללת כדי שההתייחסות תיעלם.

אם כן, מתי עליך לסלק התייחסות? איזה היבט של המחלקה Stack גורם לה להיות רגישה לדליפות זיכרון? במילים פשוטות, המחלקה Stack **מנהלת את הזיכרון שלה בעצמה**. אוסף הזיכרון (storage pool) מורכב מאלמנטים של המערך items (התאים של ייחוסי האובייקטים ולא האובייקטים עצמם). האלמנטים בחלק הפעיל של המערך (כמו שהוגדר קודם) **מוקצים** והאלמנטים ביתרת המערך **משוחררים**. לאוסף האשפה אין שום דרך לדעת זאת. לגבי אוסף האשפה, כל ההתייחסויות לאובייקטים במערך items תקפות באותה מידה. רק המתכנת יודע שהחלק הלא פעיל שבהקצאה של המערך אינו חשוב. המתכנת מעביר עובדה זו באופן יעיל לאוסף האשפה על ידי סילוק ידני של אלמנטים במערך, ברגע שהם משתייכים לחלק הלא פעיל.

ככלל, בכל פעם שמחלקה מנהלת את הזיכרון שלה, על המתכנת להיות ערני לדליפות זיכרון. בכל פעם שאלמנט משוחרר, כל ייחוס אל אובייקט שאוחסן באלמנט צריך להיות מסולק. כלומר, להימחק.

מקור נפוץ אחר לדליפות זיכרון הוא **מטמוני זיכרון** (caches). לאחר ששמת ייחוס אובייקט במטמון, קל לשכוח שהוא שם ולהשאיר אותו במטמון גם לאחר שהוא נעשה לא רלוונטי. יש שני פתרונות אפשריים לבעיה זו. אם אתה מספיק בר מזל כדי ליישם מטמון באופן בו האלמנטים שבתוך המטמון רלוונטיים, רק כל עוד יש להם ייחוסים גם מחוץ למטמון, יצג את המטמון כ-WeakHashMap. הייחוסים שבמטמון יוסרו באופן אוטומטי כאשר הם יתיישנו. אולם, בדרך כלל המטמון ממומש באופן שבו משך הזמן שבו ייחוס במטמון הוא רלוונטי אינו מוגדר היטב, והייחוסים שבתוכו נעשים פחות ופחות שימושיים עם הזמן. בנסיבות אלו, המטמון צריך להיות מנוקה מדי פעם מייחוסים שנעשו בלתי שמישים. ניקוי זה יכול להיעשות על ידי מטלת (thread) רקע (אפשרי באמצעות API בשם java.util.Timer), או כתופעת לוואי של הוספת כניסות חדשות למטמון. המחלקה java.util.LinkedHashMap, שהתווספה בגירסה 1.4, מסייעת לגישה השנייה עם השיטה removeEldestEntry שלה.

דליפות זיכרון לא מודיעות על עצמן ככשלים, ולכן הן יכולות להתקיים במערכת במשך שנים. באופן טיפוסי הן מתגלות לאחר בחינה דקדקנית של הקוד או בעזרת כלי ניפוי באגים שידוע בשם **heap profiler** (פרופיל הערימה). לכן, מאוד רצוי ללמוד כיצד לצפות מראש בעיות כאלו לפני התרחשותן, ולמנוע מהן לקרות.

נושא 6: הימנע ממסיימים (finalizers)

מסיימים (finalizers) הם בלתי ניתנים לחיזוי מראש, לעיתים קרובות הם מסוכנים ובדרך כלל בלתי נחוצים. השימוש בהם עלול לגרום התנהגות לא יציבה, לביצועים דלים ולבעיות ניידות (portability). למסיימים יש מעט שימושים תקפים, אותם נכסה מאוחר יותר בנושא הזה, אך כעיקרון מנחה, יש להימנע מהם.

מתכנתי C++ מוזהרים לשים לב לכך שמסיימים אינם האנלוגיה למפרקים, של C++ (destructors). בשפת C++, מפרקים הם הדרך המקובלת להחזיר את המשאבים שהוקצו לאובייקט, והם בני זוג הפוכים והכרחיים לבנאים. ב-Java, אוסף האשפה מחזיר לשימוש את המשאבים שקושרו לאובייקט, כאשר האובייקט נעשה בלתי נגיש. הדבר אינו דורש שום מאמץ מיוחד מצד המתכנת. המפרקים של C++ משמשים להחזרת משאבים נוספים, שאינם בהכרח משאבי זיכרון. בשפת Java, הבלוק try-finally משמש בדרך כלל למטרה זו.

אין כל ערובה שמסיימים יפעלו מייד [JLS, 12.6]. ייתכן שיעבור זמן כלשהו מהרגע בו אובייקט נעשה בלתי נגיש ועד לרגע בו המסיים מופעל. משמעות הדבר היא ש**אין לבצע באמצעות המסיים אף משימה קריטית התלויה בזמן**. לדוגמה, תהיה זו טעות חמורה לסמוך על מסיים כדי לסגור קבצים פתוחים, מכיון שהמתארים של קבצים פתוחים מוגבלים במשאבים. אם קבצים רבים נותרו פתוחים כי ה-JVM מעכב את הפעלת המסיימים, התוכנית עלולה לקרוס מכיון שלא תוכל לפתוח קבצים.

המיידיות שבה מופעלים מסיימים היא בעיקר תוצאה של אלגוריתם אוסף האשפה, אשר אינו אחיד במימושים השונים של JVM. ההתנהגות של תוכנית התלויה בביצוע מיידי של מסיימים יכולה גם כן להשתנות. בהחלט אפשרי שתוכנית כזו תפעל באופן מושלם ב-JVM שבו בחנת אותה, אבל תיכשל באופן אומלל ב-JVM החביב על הלקוח החשוב ביותר שלך.

מסיימים שביצועם מתעכב אינם רק בעיה תיאורטית. אספקת מסיים למחלקה יכולה באופן שרירותי ובמצבים נדירים לעכב את איתור המופעים השונים שלה. עמית שלי ניפה לאחרונה שגיאות ביישום של ממשק משתמש גרפי (GUI) שמשך ביצועו ארוך, אשר היה דועך באופן מסתורי עקב OutOfMemoryError. ניתוח גילה שבזמן דעיכת היישום, היו ליישום אלפי אובייקטים גרפיים במחסנית המסיים שרק חיכו לשלב הסיום שהתעכב ושמשאביהם היו צריכים לחזור למערכת. לרוע המזל, מטלת המסיים רצה בעדיפות נמוכה יותר מאשר מטלה אחרת ביישום, כך שאובייקטים לא עברו לתהליך הסיום באותו קצב שבו הם היו מוכנים לכך. ה-JLS אינו מבטיח דבר בנוגע לאופן שבו מטלה תבצע מסיימים, כך שאין שום דרך למנוע מסוג זה של בעיות להתרחש, חוץ מאשר להימנע משימוש במסיימים.

לא רק שה-JLS אינו מבטיח דבר בנוגע לביצוע מהיר של מסיימים, הוא גם אינו מבטיח שהם אכן יתבצעו. לגמרי אפשרי, ואפילו סביר, שתוכנית תסתיים בלי לבצע מסיימים עבור כמה אובייקטים שכבר אינם נגישים. המסקנה היא ש**לעולם אל תהיה תלוי במסיים כדי לעדכן מצב קריטי מתמשך**. לדוגמה, דרך טובה להביא את כל המערכת המופצת שלך לעצירה מתסכלת, היא להיות תלוי במסיים כדי לשחרר נעילה עקשנית של משאב משותף, כמו מסד נתונים.

אל תתפתה להפעיל את השיטות System.gc ו-System.runFinalization. הן עשויות להגדיל את הסבירות שמסיימים יתבצעו, אך **אינן** מבטיחות זאת. השיטות היחידות שמתיימרות להבטיח ביצוע מסיימים הן System.runFinalizersOnExit והתאומה "המרושעת" שלה, Runtime.runFinalizersOnExit. שיטות אלו אינן טובות ואפילו גרועות, וכבר הוצאו מכלל שימוש (deprecated).

במקרה שעדיין לא השתכנעת שיש להימנע משימוש במסיימים, הנה עוד מידע הראוי להתייחסות: אם במהלך ביצוע מסיים מעוררים חריג שלא נתפס, המערכת תתעלם מהחריג ותהליך הסיום של האובייקט הזה יופסק [JLS, 12.6]. חריגים שלא נתפסו עלולים להשאיר אובייקטים במצב מקולקל. אם מטלה אחרת מנסה להשתמש באותו אובייקט מקולקל, צפה לביצוע שרירותי שלו שאינו ניתן לחיזוי מראש. בדרך כלל, חריג שלא נתפס יחסל את המטלה וידפיס את תוכן המחסנית, אבל הדבר לא יקרה אם זה קורה במסיים. במקרה כזה לא תודפס אפילו אזהרה.

אם כן, מה עליך לעשות במקום להשתמש במסיימים עבור מחלקה שהאובייקטים שלה מסתירים משאבים הדורשים סיום, כמו קבצים או מטלות? כל שעליך לעשות הוא **לספק שיטת סיום מפורשת**, ולדרוש מלקוחות לקרוא לשיטה זו עבור כל מופע שאין בו עוד צורך. פרט אחד שראוי להזכירו: שיטת הסיום המפורשת חייבת לרשום בשדה פרטי שהאובייקט אינו בר-תוקף עוד. שיטות אחרות חייבות לבדוק שדה זה ולעורר את החריג IllegelStateException, אם הן נקראות לאחר שהאובייקט חוסל.

דוגמה טיפוסית של שיטת סיום מפורשת היא השיטה close שב-InputStream וב-OutputStream. דוגמה אחרת היא השיטה cancel שב-java.util.Timer, המבצעת את שינויי המצב הרצויים, כדי לגרום למטלה שקשורה למופע Timer להסתיים כראוי. דוגמאות מ-java.awt כוללות את Graphics.dispose ואת Window.dispose. לעיתים קרובות מגזימים בחשיבות של שיטות אלו עבור ביצועים צפויים. שיטה קרובה היא Image.flush, המשחררת את כל ההקצאות שהיו קשורות למופע Image, אך משאירה אותו במצב בו הוא עדיין ניתן לשימוש, תוך הקצאה חוזרת של המשאבים, אם יש בכך צורך.

לעיתים קרובות משתמשים בשיטות סיום מפורשות בצירוף עם התבנית try-finally, כדי להבטיח סיום מהיר. קריאה לשיטת הסיום המפורשת בתוך בלוק finally, מבטיחה שהיא תתבצע אפילו אם עוררו את החריג בזמן השימוש באובייקט:

```
//try-finally block tuarantees execution of termination method
Foo foo = new Foo(...);
try {
  // Do what must be done with foo
  ...
} finally {
  foo.terminate();          // Explicit termination method
}
```

אם כך הדבר, לשם מה בכלל צריך מסיימים? יש להם שני שימושים לגיטימיים. הראשון, לתפקד כ"רשת ביטחון" במקרה שבעליו של האובייקט שכח לקרוא לשיטת הסיום המפורשת שסיפקת, בהמשך לעצה שנתתי בפיסקה הקודמת. למרות שאין שום ערובה לכך שהמסיים ייקרא במהירות, עדיף לשחרר את המשאב הקריטי מאוחר מאשר לעולם לא, במקרים (נקווה שיהיו נדירים) שבהם הלקוח נכשל במימוש החלק שלו בעסקה על ידי קריאה לשיטת הסיום המפורשת. לשלוש המחלקות שהוזכרו כדוגמה לתבנית שיטת הסיום המפורשת (OutputStream, InputStream ו-Timer) יש גם מסיימים המתפקדים כרשתות ביטחון למקרה ששיטות הסיום שלהם אינן נקראות.

שימוש לגיטימי שני של מסיימים נוגע לאובייקטים עם **עמיתים מקומיים** (native peers). עמית מקומי הוא אובייקט מקומי (native) שאובייקט רגיל מאציל לו סמכויות באמצעות שיטות מקומיות. מכיון שעמית מקומי אינו אובייקט רגיל, אוסף האשפה אינו יודע על קיומו והוא אינו יכול לדרוש אותו כאשר העמית הרגיל שלו נדרש. מסיים הוא מכשיר מתאים לביצוע משימה זו, **בהנחה שהעמית המקומי אינו מחזיק במשאבים חיוניים**. אם העמית המקומי מחזיק משאבים שחייבים להשתחרר במהירות, המחלקה תצטרך לכלול שיטת סיום מפורשת, כמו שתוארה לעיל. שיטת הסיום אמורה לעשות את כל הדרוש כדי לשחרר את המשאב החיוני. שיטת הסיום יכולה להיות שיטה מקומית, או שהיא יכולה לקרוא לאחת כזו.

חשוב לציין ש"שירשור מסיימים" ("finalizer chaining") אינו מתבצע באופן אוטומטי. אם למחלקה (פרט למחלקה Object) יש מסיים, ותת-מחלקה דורסת אותו, המסיים של תת-המחלקה חייב לקרוא למסיים של מחלקת-העל באופן מפורש. אתה אמור לסיים את תת-המחלקה בבלוק try ולקרוא למסיים של מחלקת-העל בבלוק finally המתאים.

הדבר מבטיח שהמסיים של מחלקת-העל יופעל, אפילו אם המסיים של מחלקת-העל מעורר חריג, ולהיפך:

```
// Manual finalizer chaining
protected void finalize() throws Throwable
  try {
    // Finalize subclass state
    ...
  } finally {
    super.finalize();
  }
}
```

אם יישום של תת-מחלקה דורס מסיים של מחלקת-על, אך שוכח לקרוא למסיים של מחלקת-העל באופן מפורש (או בוחר שלא לעשות זאת), המסיים של מחלקת-העל לעולם לא ייקרא. אפשר להתגונן מפני תת-מחלקה לא זהירה או זדונית שכזו, במחיר של יצירת אובייקט נוסף עבור כל אובייקט שצריך להסתיים. במקום להציב את המסיים במחלקה שדורשת סיום, מקם אותו במחלקה אנונימית (נושא 18) שמטרתה היחידה היא לסיים את המופע המצורף שלה. מופע יחיד של המחלקה האנונימית, שמכונה "**משגיחת המסיימים**" ("finalizer guardian"), נוצר עבור כל מופע של המחלקה הסוגרת (enclosing class). המופע הסוגר (enclosing instance) שומר את הייחוס היחיד למשגיח המסיימים שלו בשדה של מופע פרטי, כך שמשגיח המסיימים נעשה כשיר לסיום בדיוק לפני שהמופע הסוגר נעשה כשיר לסיום. כאשר מסיימים את המשגיח הזה, הוא מבצע את פעילות הסיום המבוקשת עבור המופע הרצוי, כאילו המסיים שלו היה שיטה של המחלקה הסוגרת הזו:

```
// Finalizer Guardian idiom
public class Foo {
  // Sole purpose of this object is to finalize outer Foo object
  private final Object finalizerGuardian = new Object()  {
    protected void finalize() throws Throwable {
      // Finalize outer Foo object
      ...
    }
  };
  ...     // Remainder omitted
}
```

שים לב שלמחלקה הציבורית Foo אין שום מסיים (פרט למסיים הפשוט שהיא יורשת מ-Object), כך שאין זה משנה אם מסיים של תת-מחלקה קורא ל-super.finalize או לא. יש לשקול שימוש בטכניקה זו עבור כל מחלקה ציבורית לא סופית (nonfinal public class) שיש לה מסיים.

לסיכום, אל תשתמש במסיימים, פרט למקרה שעליהם לתפקד כרשת ביטחון, או כדי לסיים משאבים מקומיים שאינם קריטיים. במקרים הנדירים בהם אתה משתמש במסיים, זכור לקרוא ל-super.finalize. לבסוף, אם אתה צריך לשייך מסיים למחלקה ציבורית לא סופית, שקול שימוש במשגיח מסיימים, כדי להבטיח שהמסיים יופעל, אפילו אם מסיים של תת-מחלקה אינו מצליח לקרוא ל-super.finalize.

פרק 3

שיטות המשותפות לכל האובייקטים

Object היא מחלקה מגובשת וברורה, אך למרות זאת היא תוכננה מלכתחילה להרחבה. לכל השיטות הלא סופיות שלה (equals, hashCode, toString, clone, finalize) יש הסכמים כלליים (general contracts) מפורשים, מכיון שעוצבו כך שניתן יהיה לדרוס אותן. כל מחלקה שדורסת את השיטות האלו חייבת לציית להסכמים שלהן. אי ציות להסכמים אלה ימנע ממחלקות אחרות התלויות בהסכמים אלה לתפקד באופן תקין בשיתוף עם המחלקה.

פרק זה ידריך אותך מתי וכיצד לדרוס את השיטות הלא סופיות של Object. השיטה finalize הושמטה מפרק זה, מכיון שנושא 6 בפרק 2 דן בה. כמו כן, למרות שהשיטה Comparable.compareTo אינה של Object, פרק זה כן דן בה, מכיון שיש לה אופי דומה.

נושא 7: ציית להסכם הכללי בעת דריסת equals

דריסת השיטה equals נראית פשוטה, אבל יש דרכים רבות לעשות זאת באופן שגוי, והתוצאות במקרה זה עלולות להיות הרות אסון. הדרך הקלה ביותר להימנע מבעיות היא לא לדרוס את השיטה equals, במקרה שבו כל מופע שווה לעצמו בלבד. זה הדבר הנכון לעשות במקרה שבו מתקיים אחד מהתנאים הבאים:

- **כל מופע של המחלקה הוא ייחודי ביסודו.** הדבר נכון עבור מחלקות המייצגות ישויות אקטיביות יותר מאשר ערכים, כמו Thread (מטלה). למימוש equals שמסופק על ידי Object יש בדיוק התנהגות המתאימה עבור מחלקות אלו.
- **אינך חושב שישנו צורך שהמחלקה תספק "שוויון לוגי"**. לדוגמה, java.util.Random היתה יכולה לדרוס את equals כדי לבדוק אם שני מופעי Random יפיקו את אותו רצף של מספרים אקראיים עוקבים בסדר עולה, אבל המתכנן לא חשב שהלקוחות יזדקקו ליכולת זו או שיהיו מעוניינים בה. לאור נסיבות אלו, מספיק המימוש equals שהורש מ-Object.

- **מחלקת-העל כבר דרסה את equals וההתנהגות שהורשה ממחלקת-העל מתאימה למחלקה זו**. לדוגמה, רוב מימושי Set יורשים את מימוש equals שלהם מ-AbstractSet, מימושי List יורשים מ-AbstractList ומימושי Map יורשים מ-AbstractMap.
- **המחלקה אינה ציבורית ואתה בטוח שלעולם לא תתבצע קריאה לשיטה equals שלה**. אך קריטריון זה נתון לוויכוח כיון שיש הטוענים שהשיטה equals **חייבת** להידרס בנסיבות אלו, למקרה שתבוצע קריאה שלה בטעות, ולכן יש לדרוס אותה כך:

```
public boolean equals(Object o)  {
        throw new UnsupportedOperationException();
}
```

אם כך, מתי ראוי לדרוס את Object.equals? נדרוס את equals כאשר אנו חושבים שיש צורך שהמחלקה שלנו תספק **שוויון לוגי** - logical equality (המשווה את תוכנם הלוגי של האובייקטים בניגוד ל**השוואת זהות אובייקטים** רגילה), ומחלקת-על עדיין לא דרסה את equals כדי לממש את ההתנהגות הרצויה. זהו בדרך כלל המקרה עבור **מחלקות ערך** (value classes), כמו Integer או Date. מתכנת המשווה ייחוסים לאובייקטי ערך על ידי שימוש בשיטה equals, מצפה לגלות אם הם שווים באופן לוגי (כלומר מכילים אותו ערך), ולא אם הם מתייחסים לאותו אובייקט. לא רק שדריסת השיטה equals חיונית כדי לספק את ציפיות המתכנת, אלא שהדבר גם מאפשר למופעים של המחלקה לתפקד כמפתחות מפה (map keys), או להעניק לאלמנטים התנהגות צפויה ורצויה.

סוג אחד של מחלקת ערך אשר **אינו** דורש את דריסת השיטה equals הוא **typesafe enum** (נושא 21). מכיון שמחלקות typesafe enum מבטיחות שלכל היותר אובייקט אחד יתקיים עם כל ערך, השיטה equals של Object היא שוות ערך לשיטה equals לוגית עבור מחלקות מסוג זה.

כאשר אתה דורס את השיטה equals, אתה חייב לציית להסכם הכללי שלה. להלן ההסכם, אשר הועתק מההוראות של java.lang.Object:

השיטה **equals** מממשת **יחסי שקילות** (equivalence relation) עם מספר תכונות:

- **רפלקסיביות** (reflexive): עבור כל ערך ייחוס x, חייב x.equals(x) להחזיר true.
- **סימטריות** (symmetric): עבור כל ערכי ייחוס x ו-y, מחזירה x.equals(y) ערך true אם ורק אם y.equals(x) מחזירה true.
- **טרנזיטיביות** (transitive): עבור כל ערכי ייחוס x, y ו-z, אם x.equals(y) מחזירה true ו-y.equals(z) מחזירה true, אז x.equals(z) חייבת להחזיר true.
- **עקביות** (consistent): עבור כל ערכי ייחוס x ו-y, קריאות כפולות של x.equals(y) מחזירות true באופן קבוע או מחזירות false באופן קבוע, בהנחה שהאובייקטים לא שונו.
- "**אי-איפוס**" (non-null). עבור כל ערך ייחוס x שאינו null, חייבת x.equals(null) להחזיר false.

אם אינך בעל נטיות מתמטיות, הדבר עלול להיראות לך קצת מפחיד, אבל אל תתעלם מכך! אם תפר את הכללים, אתה בהחלט עלול לגלות שהתוכנית שלך מתנהגת באופן לא סדיר ואף עלולה לקרוס. אתה עלול למצוא שקשה מאוד לזהות בדייקנות את מקור הכישלון. כמו דבריו של John Donne - מחלקה אינה אי. מופעים של מחלקה אחת שמועברים אל מחלקה אחרת הם דבר נפוץ. מחלקות רבות, וביניהן כל מחלקות האוספים, תלויות באובייקטים שמועברים אליהן ואשר מצייתים להסכם equals.

כעת, לאחר שאתה כבר ער לסכנות שבהפרת הסכם equals, הבה נבדוק הסכם זה בדקדקנות. החדשות הטובות הן שלמרות הרושם הכללי, ההסכם באמת אינו מסובך כל כך וברגע שהבנת אותו, אין כל קושי לדבוק בו. הבה נבחן את חמש הדרישות לפי הסדר:

רפלקסיביות - הדרישה הראשונה אומרת שאובייקט חייב להיות שווה לעצמו. קשה להעלות על הדעת הפרה של דרישה זו שלא בכוונה. אם תנסה להפר זאת ואחר כך להוסיף מופע של המחלקה שלך לאוסף, השיטה contains של האוסף כמעט בוודאות תאמר שהאוסף אינו מכיל את המופע שזה עתה הוספת (כיון שכדי לבדוק האם הוא נמצא באוסף היא משתמשת בשיטה equals).

סימטריות - הדרישה השנייה אומרת שכל שני אובייקטים חייבים להסכים על עובדת היותם שווים. שלא כמו הדרישה הראשונה, אין כל קושי לדמיין מקרה שבו דרישה זו תופר שלא בכוונה. לדוגמה, חשוב על המחלקה הבאה:

```
/**
 * Case-insensitive string. Case of the original string is
 * preserved by toString, but ignored in comparisons.
 */
public final class CaseInsensitiveString {
    private String s;

    public CaseInsensitiveString(String s) {
        if (s == null)
            throw new NullPointerException();
        this.s = s;
    }
    // Broken - violates symmetry!
    public boolean equals(Object o) {
        if (o instanceof CaseInsensitiveString)
            return s.equalsIgnoreCase(
                ((CaseInsensitiveString)o).s);
       if (o instanceof String)  // One-way interoperability!
          return s.equalsIgnoreCase((String)o);
        return false;
    }
     ...  // Remainder omitted
}
```

שיטת equals במחלקה זו מנסה בתמימות לפעול באופן משולב עם מחרוזות רגילות. נניח שיש לנו מחרוזת אחת שאינה רגישה לרישיות (Case Insensitive), ומחרוזת רגילה אחת:

```
CaseInsensitiveString cis = new CaseInsensitiveString("Polish");
String s = "polish";
```

כצפוי, cis.equals(s) מחזירה true. אך יש בעיה: השיטה equals ב-CaseInsensitiveString מכירה במחרוזות רגילות, אבל השיטה equals ב-String אינה מכירה במחרוזות case sensitive. לכן, s.equals(cis) מחזירה false, שזו הפרה ברורה של עיקרון הסימטריות. נניח שאתה שם באוסף מחרוזת אחת שרגישה לאותיות רישיות:

```
List list = new ArrayList();
list.add(cis);
```

מה מחזירה list.contains(s) בנקודה זו? מי יודע? במימוש הנוכחי של Sun היא מחזירה False, אבל במימוש אחר היא יכולה להחזיר true או לעורר חריג בזמן ריצה. ברגע שהפרת את ההסכם של equals, לא תדע כיצד אובייקטים אחרים יתנהגו כאשר ייפגשו באובייקט שלך.

כדי להתגבר על בעיה זו, עליך לנטוש את הרעיון המוטעה לפעול במשולב עם String מתוך השיטה equals. לאחר שתעשה זאת, תוכל לבנות את השיטה כדי לגרום לה להחזיר ערך יחיד:

```
public boolean equals(Object o){
    return o instanceof CaseInsensitiveString &&
        ((CaseInsensitiveString)o).s.equalsIgnoreCase(s);
}
```

טרנזיטיביות - הכלל השלישי של ההסכם של equals אומר שאם אובייקט אחד שווה לאובייקט שני והאובייקט השני שווה לאובייקט שלישי, אז האובייקט הראשון חייב להיות שווה לאובייקט השלישי. גם כאן אין כל קושי לתאר מקרה בו כלל זה מופר שלא בכוונה. חשוב על מקרה בו מתכנת יוצר תת-מחלקה שמוסיפה משמעות נוספת למחלקת-העל שלו.

במילים אחרות, תת-המחלקה מוסיפה פיסת מידע המשפיעה על השוואת equals.
הבה נתחיל עם מחלקה Point (נקודה) דו-מימדית שאינה ניתנת לשינוי:

```
public class Point {
   private final int x;
   private final int y;
   public Point(int x, int y) {
      this.x = x;
      this.y = y;
   }

   public boolean equals(Object o) {
      if (!(o instanceof Point))
         return false;
      Point p = (Point)o;
      return p.x == x && p.y == y;
   }

    ...  // Remainder omitted
}
```

נניח שברצונך להרחיב את המחלקה על ידי הוספת תיאור של צבע לנקודה:

```
public class ColorPoint extends Point {
   private Color color;

   public ColorPoint(int x, int y, Color color) {
      super(x, y);
      this.color = color;
   }

   ... // Remainder omitted
}
```

כיצד תיראה השיטה equals? אם תשמיט אותה לגמרי, המימוש בא בהורשה מ-Point, והמידע לגבי הצבע אינו נלקח בחשבון בהשוואות של equals. למרות שאין זה מפר את ההסכם של equals, ברור לחלוטין שאי אפשר לקבל זאת. נניח שאתה כותב שיטה equals המחזירה true רק אם הארגומנט שלה הוא נקודת צבע אחרת, עם אותו צבע ומיקום:

```
// Broken - violates symmetry!
   public boolean equals(Object o) {
      if (!(o instanceof ColorPoint))
         return false;
      ColorPoint cp = (ColorPoint)o;
      return super.equals(o) && cp.color == color;
   }
```

הבעיה עם השיטה הזאת היא בכך שאתה עלול לקבל תוצאות שונות כאשר תשווה נקודה לנקודת צבע ולהיפך. ההשוואה הראשונה מתעלמת מצבע ואילו ההשוואה השנייה תמיד מחזירה false כי טיפוס הארגומנט אינו נכון. כדי להבהיר את העניין, הבה ניצור נקודה אחת ונקודת צבע אחת:

```
Point p = new Point(1, 2);
ColorPoint cp =new ColorPoint(1, 2, Color.RED);
```

אם כך, p.equals(cp) מחזירה true בעוד ש-cp.equals(p) מחזירה false. כלומר, השיטה equals אינה עומדת בדרישת הסימטריות. תוכל לנסות לתקן את הבעיה בכך שתגרום ל-ColorPoint.equals להתעלם מצבע, כאשר היא מבצעת "השוואות מעורבות":

```
// Broken - violates transitivity.
  public boolean equals(Object o) {
    if (!(o instanceof Point))
      return false;
    // If o is a normal Point, do a color-blind comparison
    if (!(o instanceof ColorPoint))
      return o.equals(this);
    // o is a ColorPoint; do a full comparison
    ColorPoint cp = (ColorPoint)o;
    return super.equals(o) && cp.color == color;
  }
```

גישה זו אמנם שומרת על עיקרון הסימטריות, אבל עושה זאת על חשבון עיקרון הטרנזיטיביות:

```
ColorPoint p1 =new ColorPoint(1, 2, Color.RED);
Point p2 =new Point(1, 2);
Color Point p3 =new Color Point(1, 2, Color.BLUE);
```

בנקודה זו, p1.equals(p2) ו-p2.equals(p3) מחזירות true, בעוד ש-p1.equals(p3) מחזירה false. זו הפרה ברורה של עיקרון הטרנזיטיביות. שתי ההשוואות הראשונות הן "עיוורות צבעים", בשעה שההשוואה השלישית לוקחת בחשבון את הצבע.

אם כן, מה הפתרון? מתברר שזו בעיה עקרונית של יחסים שקולים בשפות מוכוונות עצמים. **למעשה, אין דרך להרחיב מחלקה מופעית ולהוסיף היבט תוך כדי שמירה על הסכם equals**. בכל אופן, יש דרך טובה לעקוף את הבעיה, בלי לתקן את הבעיה עצמה. פעל על פי העצה שבנושא 14.

במקום לתת ל-ColorPoint להרחיב את Point, תן ל-ColorPoint שדה Point פרטי ושיטה **view** (תצוגה) ציבורית (נושא 4) המחזירה את הנקודה באותו מצב כמו נקודת הצבע הזאת:

```
// Adds an aspect without violating the equals contract
public class ColorPoint2 {
   private Point point;
   private Color color;

   public ColorPoint2(int x, int y, Color color) {
      point = new Point(x, y);
      this.color = color;
   }

   /**
    * Returns the point-view of this color point.
    */
   public Point asPoint() {
      return point;
   }

   public boolean equals(Object o) {
      if (!(o instanceof ColorPoint2))
         return false;
      ColorPoint2 cp = (ColorPoint2)o;
      return cp.point.equals(point) && cp.color.equals(color);
   }

    ...  // Remainder omitted
}
```

יש כמה מחלקות בספריות של פלטפורמת Java אשר יוצרות תת-מחלקות למחלקה מופעית ומוסיפות היבט (aspect). לדוגמה, java.sql.Timestamp יוצרת תת-מחלקות ל-java.util.Date, תוך הוספת שדה nanoseconds. המימוש של equals עבור Timestamp מפר את עיקרון הסימטריות ויכול לגרום להתנהגות לא יציבה, אם Timestamp ואובייקט Date משמשים באותו אוסף או אם הם מעורבבים. למחלקה Timestamp יש התראה למתכנת המזהירה אותו מפני ערבוב dates ו-timestamps. למרות שלא תתעוררנה בעיות אם לא תערבב אותם, אין דבר המונע ממך לעשות זאת, והתקלות שתיגרמנה עלולות להיות קשות לניפוי. המחלקה Timestamp הינה יוצאת דופן, ורצוי לא לחקות אותה.

שים לב לכך שאתה **יכול** להוסיף היבט לתת-מחלקה של מחלקה **מופשטת** (abstract) ללא הפרת הסכם equals. הדבר חשוב בשביל סוג ההיררכיות המחלקתיות שתקבל אם תפעל לפי העצה בנושא 20. לדוגמה, יכולה להיות לך מחלקה מופשטת בשם Shape (צורה) ללא שום היבטים, תת-מחלקה בשם Circle (עיגול) שמוסיפה שדה radius ותת-מחלקה בשם rectangle (מלבן) שמוסיפה שדות length (אורך) ו-width (רוחב).

בעיות מהסוג שזה עתה תואר לא תתעוררנה כל עוד אי אפשר ליצור מופע של מחלקת-העל.

עקביות - העיקרון הרביעי של הסכם equals אומר שאם שני אובייקטים שווים, עליהם להישאר שווים כל הזמן, אלא אם כן אחד מהם (או שניהם) עובר שינוי כלשהו. אין זה ממש עיקרון, אלא יותר תזכורת לכך שאובייקטים משתנים יכולים להיות שווים לאובייקטים שונים בזמנים שונים, בעוד שאובייקטים בלתי משתנים אינם יכולים. כאשר אתה כותב מחלקה, חשוב באופן מעמיק אם היא צריכה להיות מחלקה שאינה ניתנת לשינוי (נושא 13). אם הגעת למסקנה שעליה להיות מחלקה שאינה ניתנת לשינוי, הקפד על כך שהשיטה equals שלך אוכפת את ההגבלה לפיה אובייקטים שווים נשארים שווים ואובייקטים שאינם שווים נשארים לא-שווים כל הזמן.

אי-איפוס - העיקרון האחרון, אשר בשל היעדר שם קראתי לו "אי-איפוס", אומר שכל האובייקטים לא יהיו שווים ל-null. אמנם קשה לדמיין החזרת true כתגובה לקריאה o.equals(null), אבל קל לדמיין התעוררות של חריג NullPointerException בטעות. ההסכם הכללי אינו מרשה זאת. למחלקות רבות יש שיטות equals אשר שומרות מפני מצב זה בעזרת השוואה מפורשת ל-null:

```
public boolean equals(Object o){
  if (o ==null)
    return false;
  ...
}
```

ההשוואה הזו אינה הכרחית. כדי לבחון שוויון של הארגומנטים, השיטה equals חייבת תחילה להמיר את הארגומנט לטיפוס מתאים כדי ששיטות הגישה שלו (accessors) ייקראו, או כדי שהשדות שלו יהיו נגישים. לפני ביצוע ההמרה, השיטה חייבת להשתמש באופרטור instanceof כדי לבדוק אם הארגומנט שלה הוא מהטיפוס הנכון:

```
public boolean equals(Object o) {
  if (!(o instanceof MyType))
    return false;
  ...
}
```

אם לא התבצעה בדיקה זו והשיטה equals קיבלה ארגומנט מטיפוס לא נכון, היא עשויה לעורר חריג ClassCastException, המפר את הסכם equals. אבל, האופרטור instanceof מוגדר כך שיחזיר false אם האופרנד הראשון שלו הוא null ואין זה משנה כלל מהו האופרנד השני שלו [JLS, 15.19.2]. לכן, בדיקת הטיפוס תחזיר false אם הועבר null, כך שאינך צריך בדיקת null נוספת. נצרף את הכל יחד, והרי לנו מרשם לשיטה equals איכותית:

1. **השתמש באופרטור == כדי לבדוק אם הארגומנט הוא ייחוס לאובייקט הזה**. אם כן, החזר true. זהו בסך הכל שיפור של הביצועים, אך שווה לבצע אותו אם ההשוואה עלולה להיות יקרה.

2. **השתמש באופרטור instanceof כדי לבדוק אם הארגומנט הוא מהטיפוס הנכון**. אם הארגומנט אינו מהטיפוס הנכון, החזר false. בדרך כלל, "הטיפוס הנכון" הוא המחלקה שבה השיטה מתבצעת. לפעמים הטיפוס שאותו נבדוק יהיה הממשק אותו המחלקה מממשת. נעשה זאת כאשר נרצה לאפשר השוואות בין אובייקטים ממחלקות שונות הממשות ממשק משותף. לדוגמה, למשקי האוספים Set, List, Map ו-Map.Entry יש את התכונה הזאת.

3. **המר את הארגומנט לטיפוס הנכון**. מובטח שהמרה זו תצליח, כיון שקדם לה מבחן instanceof.

4. **עבור כל שדה "משמעותי" במחלקה, בדוק האם שדה הארגומנט מתאים לשדה המקביל של האובייקט הזה**. אם כל הבדיקות האלו מצליחות, החזר true. אחרת, החזר false. אם הטיפוס בשלב 2 הוא ממשק, אתה חייב לגשת לשדות הייחודיים של הארגומנט דרך שיטות ממשק. אם הטיפוס הוא מחלקה, ייתכן שתוכל לגשת אל השדות ישירות, על פי הנגישות שלהם. לשדות פשוטים שהטיפוס שלהם אינו float או double, השתמש באופרטור == להשוואות. לשדות ייחוס אובייקטים, קרא לשיטה equals באופן רקורסיבי. לשדות float, תרגם לערכי int על ידי שימוש ב-Float.floatToIntBits והשווה את ערכי int על ידי שימוש באופרטור ==.
שדות double, תרגם לערכי long על ידי שימוש ב-Double.doubleToLongBits והשווה את ערכי long על ידי שימוש באופרטור == (הטיפול המיוחד בשדות float ו-double דרוש בגלל הקיום של Float.NaN, 0.0f- וקבועי double המקבילים. עיין בתיעוד של Float.equals לקבלת מידע נוסף). לשדות מערכים, השתמש בקווים מנחים אלה עבור כל איבר. ייתכן שכמה שדות ייחוס של אובייקט יכילו null, אך זה לגיטימי.

כדי להימנע מהאפשרות של NullPointerException, השתמש בשורת הקוד הבאה כדי להשוות שדות כאלה:

```
(field == null ? o.field == null : field.equals(o.field))
```

שורת הקוד החילופית הבאה יכולה לפעול מהר יותר אם field ו-o.field הם לעיתים קרובות ייחוסי אובייקטים זהים:

```
(field == o.field || (field != null && field.equals(o.field)))
```

עבור כמה מחלקות, כמו CaseInsensitiveString שהוצגה כבר, השוואת השדות מורכבת יותר מאשר בדיקות השוואה פשוטות. אם זהו המקרה, הדבר אמור להיות גלוי מהמפרט של המחלקה.

אם כך, ייתכן שתרצה לשמור את הצורה **הקאנונית** (canonical form) בכל אובייקט, כך שהשיטה equals תוכל לבצע השוואות מדויקות וזולות על הצורות הקאנוניות, במקום השוואות יקרות ולא מדויקות. טכניקה זו הכי מתאימה למחלקות ש**אינן ניתנות לשינוי** (immutable) (נושא 13), כיון שהטיפוס הקאנוני יצטרך להישמר מעודכן באובייקט שיכול להשתנות.

הביצועים של השיטה equals יכולים להיות מושפעים מהסדר שבו השדות מושווים. לביצועים מיטביים, עליך להשוות תחילה שדות שסביר יותר שהם שונים, או שהם פחות יקרים להשוואה. במצב האידיאלי, שני תנאים אלה מתקיימים. אל תשווה שדות שאינם חלק ממצב לוגי של אובייקט, כמו שדות Object אשר משמשים לסינכרון פעולות. אין צורך שתשווה **שדות יתירים** (redundant fields) היכולים להיות מחושבים מ"שדות משמעותיים", אבל אם תעשה זאת, ייתכן שתשפר את ביצועיה של השיטה equals. אם השדה היתיר מוסיף לתיאור המסכם של האובייקט בשלמותו, השוואת שדה זה תחסוך את העלות של השוואת הנתונים עצמם במקרה שההשוואה נכשלת.

5. **כאשר סיימת לכתוב את השיטה equals שלך, שאל את עצמך שלוש שאלות**: **האם היא סימטרית, האם היא טרנזיטיבית והאם היא עקבית?** (בדרך כלל, אין צורך לדאוג לשני העקרונות האחרים). אם לא ענית בחיוב על שלוש שאלות אלו, נסה לחשוב מדוע כללים אלה אינם מתקיימים והגדר את השיטה בהתאם.

דוגמה מעשית של השיטה equals שנבנתה בהתאם למרשם זה תמצא בנושא 8 ב- PhoneNumber.equals. להלן כמה אזהרות לסיכום הנושא:

- **דרוס תמיד את hashCode כאשר אתה דורס equals** (נושא 8).
- **אל תתחכם**. אם אתה רק בודק שוויון של שדות, אין כל קושי לדבוק בהסכם של equals. אם אתה יותר מדי אגרסיבי בחיפוש אחר שוויון, יהיה לך קל להסתבך בצרות. בדרך כלל זהו רעיון רע לקחת בחשבון כל צורה שהיא של שמות נוספים או כינויים. לדוגמה, המחלקה File אינה צריכה לנסות להשוות קישורים סימליים המתייחסים לאותו קובץ. למרבה המזל, היא אינה עושה זאת.
- **אל תכתוב שיטה equals המסתמכת על משאבים לא מהימנים**. יהיה קשה לשמור על עיקרון העקביות אם תנהג כך. לדוגמה, השיטה equals של java.net.URL מסתמכת על כתובות IP של URL מארחים שמושווים. תרגום שם של מארח לכתובת IP יכול לדרוש גישה לרשת, ואין הוא מבטיח את אותן התוצאות לאורך זמן. הדבר עשוי לגרום לשיטה equals של ה-URL להפר את הסכם equals והדבר גרם כבר בעיות בפועל (לרוע המזל, התנהגות זו אינה יכולה להשתנות בגלל דרישות תאימות). עם מעט יוצאים מן הכלל, שיטות equals אמורות לבצע חישובים דטרמניסטיים על אובייקטים מקומיים בזיכרון.

- **אל תמיר את Object בטיפוס אחר בהצהרה של equals**. אין זה נדיר לראות מתכנת כותב שיטה equals שנראית כמו זו שלהלן, ואחר כך מבזבז שעות בניסיון להבין מדוע היא אינה פועלת כנדרש:

```
public boolean equals(MyClass o)   {
    ...
}
```

הבעיה היא ששיטה זו אינה **דורסת** את Object.equals, אשר הארגומנטים שלה הם מהטיפוס Object, אלא **מעמיסה** אותה (נושא 26). מקובל לספק שיטה equals בעלת "טיפוסיות חזקה" **בנוסף** לשיטה הרגילה, כל עוד שתי השיטות מחזירות את אותה התוצאה, אבל אין שום סיבה מחייבת לעשות זאת. ייתכן שבנסיבות מסוימות הדבר יביא לשיפור קל בביצועים, אבל אין זה שווה את המורכבות שנוספה (נושא 37).

נושא 8: דרוס תמיד את hashCode כאשר אתה דורס את equals

סיבה נפוצה לבאגים היא כישלון בדריסת השיטה hashCode. **אתה חייב לדרוס את hashCode בכל מחלקה שדורסת את equals**. אם לא תעשה זאת, תגרום להפרת ההסכם הכללי של Object.hashCode - דבר אשר ימנע מהמחלקה שלך לתפקד באופן תקין בצירוף עם כל האוספים מבוססי ערבול (hash-based), כולל HashMap, HashSet ו-Hashtable.

להלן ההסכם, אשר תורגם מהמפרט של java.lang.Object:

- בכל פעם שהשיטה hashCode נקראת עבור אותו אובייקט יותר מפעם אחת במהלך ביצוע יישום, היא חייבת להחזיר את אותו ערך שלם בעקביות, בהנחה שלא השתנו נתונים שהשתמשו בהם בהשוואות equals עבור אותו אובייקט. שלם זה אינו צריך להישאר עקבי מביצוע אחד של יישום לביצוע אחר של אותו היישום.
- אם שני אובייקטים שווים לפי תוצאת השיטה equals(Object), אז קריאה לשיטה hashCode עבור כל אחד משני האובייקטים חייבת להחזיר את אותו ערך שלם של התוצאה.
- אם שני אובייקטים אינם שווים לפי השיטה equals(Object), אז קריאה לשיטה hashCode עבור כל אחד משני האובייקטים לא תהיה חייבת להחזיר את אותו ערך שלם של התוצאה. עם זאת, המתכנת צריך להיות ער לכך שהחזרת ערכים שלמים שונים עבור אובייקטים שונים יכולה לשפר את הביצועים של טבלאות הערבול (hash tables).

הכלל העיקרי המופר כאשר אתה נכשל בדריסת hashCode הוא השני: לאובייקטים שווים חייבים להיות קודי ערבול (hash codes) שווים. שני מופעים שונים יכולים להיות שווים מבחינה לוגית לפי השיטה equals של המחלקה, אבל מבחינתה של השיטה hashCode של המחלקה Object הם בסך הכל שני אובייקטים ללא שום דבר מיוחד משותף ביניהם. לכן, השיטה hahsCode של האובייקט מחזירה שני מספרים שהם לכאורה אקראיים, במקום שני מספרים שווים כפי שדורש ההסכם.

לדוגמה, חשוב על המחלקה הפשוטה PhoneNumber, אשר השיטה equals שלה בנויה לפי המרשם שהוצג בנושא 7:

```
public final class PhoneNumber {
    private final short areaCode;
    private final short exchange;
    private final short extension;

    public PhoneNumber(int areaCode, int exchange, int extension) {
        rangeCheck(areaCode,   999, "area code");
        rangeCheck(exchange,   999, "exchange");
        rangeCheck(extension, 9999, "extension");
```

```
        this.areaCode  = (short) areaCode;
        this.exchange  = (short) exchange;
        this.extension = (short) extension;
    }

    private static void rangeCheck(int arg, int max, String name) {
        if (arg < 0 || arg > max)
          throw new IllegalArgumentException(name +": " + arg);
    }

    public boolean equals(Object o) {
        if (o == this)
            return true;
        if (!(o instanceof PhoneNumber))
            return false;
        PhoneNumber pn = (PhoneNumber)o;
        return pn.extension == extension &&
               pn.exchange  == exchange  &&
               pn.areaCode  == areaCode;
    }

    // No hashCode method!
    ... // Remainder omitted
}
```

נניח שאתה מנסה להשתמש במחלקה זו עם HashMap:

```
Map m = new HashMap();
m.put(new PhoneNumber(408, 867, 5309), "Jenny");
```

בשלב זה, אתה עשוי לצפות שהשיטה m.get(new PhoneNumber(408, 867, 5309)) תחזיר "Jenny", אבל היא מחזירה null. שים לב לכך ששני מופעי PhoneNumber מעורבים: אחד משמש להכנסה לתוך HashMap והשני, מופע שווה, משמש להחזרה (שלא הצליחה). כישלונה של המחלקה PhoneNumber לדרוס את hashCode גורם לשני המופעים השווים להחזיק בקודי ערבול שאינם שווים, והדבר מפר את הסכם hashCode. לכן, השיטה get מחפשת מספר טלפון במנת ערבול שונה מהמנה שבה הוא אוחסן על ידי השיטה put. תיקון בעיה זו פשוט, כמו לספק שיטה hashCode נכונה למחלקה PhoneNumber.

אם כן, כיצד צריכה השיטה hashCode להיראות? אפשר בקלות לכתוב שיטה כזו שתהיה חוקית, אבל לא טובה. הדוגמה הבאה, למשל, תמיד חוקית אבל לעולם אל תשתמש בה:

```
// The worst possible legal hash function - never use !
public int hashCode()  { return 42; }
```

שיטה זו חוקית כי היא מבטיחה שלאובייקטים שווים יהיה אותו קוד ערבול. היא אינה טובה, ואפילו רעה, בכך שהיא מבטיחה **שלכל** אובייקט יהיה אותו קוד ערבול. לכן, כל אובייקט מעורבל לאותה המנה וטבלאות ערבול מידרדרות לרשימות מקושרות. תוכניות שאמורות לרוץ בזמן לינארי, תפעלנה במקום זה בזמן כפול ומכופל. עבור טבלאות ערבול גדולות זהו ההבדל בין פעולה לאי-פעולה. פונקציית ערבול טובה נוטה לספק קודי ערבול לא שווים עבור אובייקטים לא שווים. לזה בדיוק מכוונת הדרישה השלישית של הסכם hashCode. במצב האידיאלי, פונקציית ערבול אמורה לחלק כל אוסף הגיוני של מופעים לא שווים בצורה אחידה על פני כל ערכי הערבול האפשריים. הגשמת מצב אידיאלי זה יכולה להיות קשה ביותר. למרבה המזל, אין זה קשה יתר על המידה להשיג תחליף סביר. הנה מרשם פשוט:

1. אחסן ערך קבוע כלשהו שאינו אפס, נניח 17, במשתנה int שייקרא result.
2. עבור כל שדה f משמעותי באובייקט שלך (כלומר, כל שדה שנלקח בחשבון על ידי השיטה equals), בצע את הפעולות הבאות:
   א. חשב קוד ערבול c מטיפוס int עבור השדה:
      1) אם השדה הוא boolean, חשב: (f ? 0 : 1).
      2) אם השדה הוא byte, char, short או int, חשב: (int)f.
      3) אם השדה הוא long, חשב: ((int)(f ^ (f>>>32)).
      4) אם השדה הוא float, חשב: Float.floatToIntBits(f).
      5) אם השדה הוא double, חשב: Double.doubleToLongBits(f) וערבל את ערך long המתקבל, כמו שנאמר בשלב 3, סעיף 2א'.
      6) אם השדה הוא ייחוס אובייקט ושיטה equals של מחלקה זו משווה אותו על ידי קריאה רקורסיבית של equals, הפעל את hashCode באופן רקורסיבי על השדה. אם נדרשת השוואה יותר מורכבת, חשב "ייצוג קאנוני" עבור השדה והפעל את hashCode על הייצוג הקאנוני. אם הערך של השדה הוא null, החזר 0 (או איזה קבוע אחר, אבל נהוג להשתמש ב-0).

7) אם השדה הוא מערך, התייחס אליו כאילו כל איבר בו הוא שדה נפרד. כלומר, חשב קוד ערבול עבור כל איבר משמעותי על ידי הפעלת כללים אלה באופן רקורסיבי, וצרף את הערכים האלה כמו שמוסבר בשלב 2ב'.

ב. צרף את קוד הערבול c שחושב בשלב א' לתוך result, כך:

```
result = 37 * result + c;
```

3. החזר את result.

4. לאחר שסיימת את כתיבת השיטה hashCode, שאל את עצמך האם למופעים שווים יש קודי ערבול שווים. אם לא, נסה לחשוב מדוע ותקן את הבעיה.

מקובל לסלק שדות יתירים (redundant fields) מהחישוב של קוד הערבול. במילים אחרות, מקובל לסלק כל שדה שהערך שלו יכול להיות מחושב משדות הכלולים בחישוב. **חובה** לסלק את כל השדות שאינם משמשים בהשוואות. כישלון בסילוק שדות אלה יכול להביא לידי הפרת הכלל השני של הסכם hashCode.

ערך התחלתי שאינו אפס משמש בשלב 1, כך שערך הערבול יושפע על ידי השדות הראשוניים שערך הערבול שלהם, כמו שחושב בשלב 2א', הוא אפס. אם הערך אפס משמש כערך ראשוני בשלב 1, ערך הערבול הכולל לא מושפע על ידי שדות ראשוניים כלשהם, אשר יכולים להגביר התנגשויות. הערך 17 הוא שרירותי.

הכפל בשלב 2ב' גורם לערך הערבול להיות תלוי בסדר השדות - מה שמביא לפונקציית ערבול הרבה יותר טובה, אם המחלקה מכילה שדות כפולים דומים. לדוגמה, אם הכפל היה מושמט מפונקציית הערבול String שנבנתה בהתאם למרשם הזה, לכל האנאגרמים (היפוך אותיות היוצר מילה אחרת) יהיו קודי ערבול זהים. הכופל 37 נבחר מכיון שהוא מספר ראשוני מזדמן. אם הוא היה קבוע והכפל היה גורם לגלישה, מידע היה עלול ללכת לאיבוד, מכיון שהכפל ב-2 זהה להזזת סיביות. היתרונות שבשימוש במספרים ראשוניים הם פחות ברורים, אבל מקובל להשתמש בהם למטרה זו.

הבה ניישם את המרשם הזה למחלקת PhoneNumber. יש שלושה שדות משמעותיים, כולם מטיפוס short. יישום ראוי של המרשם מניב את פונקציית הערבול הבאה:

```
public int hashCode() {
    int result = 17;
    result = 37*result + areaCode;
    result = 37*result + exchange;
    result = 37*result + extension;
    return result;
}
```

שיטה זו מחזירה את התוצאה של חישוב דטרמיניסטי פשוט, אשר הקלט שלו הוא רק שלושה שדות משמעותיים במופע של PhoneNumber, ולכן צריך להיות ברור שלמופעי PhoneNumber שווים יש קודי ערבול שווים. למעשה, שיטה זו היא מימוש הגיוני של hashCode עבור PhoneNumber. הפתרון פשוט ומהיר בצורה מתקבלת על הדעת, וגם מבצע עבודה סבירה של פיזור מספרי טלפון שונים לתוך מנות ערבול שונות.

אם המחלקה אינה ניתנת לשינוי והעלות של חישוב קוד הערבול משמעותית, ייתכן שכדאי לשקול הטמנה של קוד ערבול באובייקט, על פני חישובו מחדש בכל פעם שיש בו צורך. אם נראה לך שמרבית האובייקטים מהטיפוס הזה ישמשו כמפתחות ערבול, אז ראוי לחשב את קוד הערבול כאשר המופע נוצר. אם לא, ייתכן שתבחר לאתחל אותו באתחול עצל בפעם הראשונה ש-hashCode נקרא (נושא 48). אין זה ברור אם המחלקה PhoneNumber שלנו ראויה לטיפול זה, אך רק כדי להראות לך איך הדבר נעשה:

```
// Lazily initialized, cached hashCode
private volatile int hashCode = 0;  // (See Item 48)

public int hashCode() {
    if (hashCode == 0) {
        int result = 17;
        result = 37*result + areaCode;
        result = 37*result + exchange;
        result = 37*result + extension;
        hashCode = result;
    }
    return hashCode;
}
```

למרות שהמרשם בנושא הזה מניב פונקציות ערבול טובות ומתקבלות על הדעת, הוא אינו מניב פונקציות ערבול ברמה הטובה ביותר (הענקת קוד ערבול שונה לכל אובייקט שונה במינימום חישובים תוך שמירה על ההסכמים). גם ספריות הפלטפורמה של Java גרסת 1.4 אינן מספקות פונקציות ערבול כאלו. כתיבת פונקציות ערבול כאלו היא נושא למחקר נמרץ בידי מתמטיקאים ואנשי מדעי המחשב התיאורטיים. אולי גירסה מאוחרת יותר של פלטפורמת Java תספק פונקציות ערבול כאלו למחלקות שלה וגם שיטות שירות שתאפשרנה למתכנתים רגילים לבנות פונקציות ערבול כאלו. בינתיים, הטכניקה שתוארה בנושא זה מתאימה ומספיקה לרוב היישומים.

אל תתפתה להשמיט חלקים משמעותיים של אובייקט מחישוב קוד הערבול כדי להשיג שיפור ביצועים. סביר שפונקציית הערבול שתתקבל כתוצאה מכך תפעל מהר יותר. עם זאת, האיכות של קוד הערבול עלולה לרדת לנקודה שבה טבלאות ערבול תיעשנה לא שימושיות מרוב איטיות. לדוגמה, פונקציית הערבול עשויה למעשה להתעמת עם אוסף גדול של מופעים השונים באופן משמעותי באותם חלקים שבחרת להתעלם מהם.

אם זה קורה, פונקציית הערבול תמפה את כל המופעים למספר קודי ערבול קטן מאוד, ואוספים מבוססי ערבול יציגו ביצועים ירודים. אין זו בעיה תיאורטית בלבד. פונקציית הערבול String שמומשה בכל גרסאות Java המוקדמות (לפני 1.2), בחנה לכל היותר 16 תווים המרווחים באופן שווה (evenly spaced) במחרוזת, החל בתו הראשון. מחלקות רבות בספריות הפלטפורמה של Java, כמו String, Integer ו-Date, הגדירו את הערך המדויק המוחזר על ידי השיטה hashCode שלהן כפונקציה של ערך המופע.
בדרך כלל אין זה רעיון טוב, כיון שהדבר מגביל אותך באופן רציני ביכולת לשפר את פונקציות הערבול בגרסאות עתידיות. אם תשאיר את הפרטים של פונקציית ערבול לא מוגדרים ויימצא בה פגם, תוכל לתקן אותה בגירסה הבאה ללא פחד מפני פגיעה בתאימות עם לקוחות התלויים בערכים המדויקים, המוחזרים על ידי פונקציית הערבול.

נושא 9: דרוס תמיד את toString

בעוד ש-java.lang.Object מממשת את השיטה toString, המחרוזת שהיא מחזירה אינה בדרך כלל המחרוזת שהמשתמש של המחלקה שלך רוצה לראות. היא מורכבת משם המחלקה שמלווה בסימן "at" (@) והייצוג ההקסדצימלי הלא מסומן (unsigned) של קוד הערבול, כמו לדוגמה "PhoneNumber@163b91". על פי ההסכם הכללי של toString, המחרוזת המוחזרת צריכה להיות "ייצוג תמציתי אבל אינפורמטיבי קל לקריאה". למרות שאפשר להתווכח לגבי התמציתיות והאינפורמטיביות של המחרוזת "PhoneNumber@163b91", היא אינה אינפורמטיבית בהשוואה ל-"(408) 867-5309". הסכם toString אומר בהמשך ש-"מומלץ שכל תת-המחלקות ידרסו את השיטה הזו". אכן, עצה טובה.

למרות שאין זה חשוב כמו לציית להסכמי equals ו-hashCode (נושאים7 ו-8), **אספקת מימוש toString טוב הופכת את המחלקה שלך לנוחה יותר לשימוש.** השיטה toString נקראת באופן אוטומטי כאשר האובייקט שלך מועבר ל-println, לאופרטור השרשור (+) או assert בגירסה 1.4. אם סיפקת שיטה toString טובה, יצירה של הודעה דיאגנוסטית שימושית היא קלה כמו:

```
System.out.println("Failed to connect: " + phoneNumber);
```

מתכנתים יספקו הודעות דיאגנוסטיות בסגנון זה בין אם תדרוס את toString או לא, אבל ההודעות לא תהיינה מובנות, אלא אם כן תדרוס אותן. היתרונות שבאספקת שיטת toString טובה מגיעים מעבר למופעים של המחלקה לאובייקטים המכילים ייחוסים למופעים אלה, ובמיוחד אוספים. מה תעדיף לראות כאשר תדפיס מפה, {Jenny=PhonerNumber@163b91} או {Jenny=(408) 867-5309} ?

כאשר הדבר מעשי, השיטה toString צריכה להחזיר את כל המידע המעניין שכלול באובייקט, כמו בדוגמה של מספר הטלפון שכעת הוצגה. הדבר אינו מעשי אם האובייקט גדול, או אם הוא מכיל מצב שאינו גורר ייצוג מחרוזת. בנסיבות כאלו, toString צריכה להחזיר קיצור כמו "Manhattan white pages (1487536 listings)" או "Thread[main, 5,main]". במצב האידיאלי, המחרוזת צריכה להסביר את עצמה (הדוגמה Thread נכשלת במבחן הזה).

החלטה חשובה בעת מימוש השיטה toString היא האם לציין את הפורמט של הערך המוחזר בתיעוד. מומלץ שתעשה זאת ל**מחלקות ערך** (value classes), כמו מספרי טלפון או מטריצות. היתרון שבציון הפורמט הוא בכך שהוא מהווה ייצוג של האובייקט שניתן לקריאה אנושית, אינו מעורפל ובתבנית סטנדרטית. ייצוג זה יכול לשמש לקלט ולפלט, ובאובייקטי נתונים קריאים ומתמידים כמו מסמכי XML. אם תציין את הפורמט, יהיה זה בדרך כלל רעיון טוב לספק בנאי String תואם (או בנאי יוצר סטטי, ראה נושא 1), כך שמתכנתים יוכלו בקלות להמיר את האובייקט לייצוג המחרוזתי שלו, ולהיפך.

גישה זו ננקטת על ידי מחלקות ערך רבות שנמצאות בספריות הפלטפורמה של Java, כמו BigInteger, BigDecimal ורבות ממחלקות העטיפה הפשוטות.

החיסרון של הגדרת הפורמט של הערך המוחזר של toString הוא שלאחר שהגדרת אותו, אתה תקוע איתו לכל החיים, בהנחה שהמחלקה שלך נמצאת בשימוש נרחב. מתכנים יכתבו קוד כדי לפצל את הייצוג, כדי לחולל אותו וכדי לשבץ אותו לנתונים מתמידים. אם תשנה את הייצוג בגירסה עתידית, תפגע בנתונים ובקוד שלהם ותגרום להם נזק. אם אינך מגדיר פורמט, אתה שומר את הגמישות להוסיף נתונים או לשפר את הפורמט בגירסה הבאה.

ללא קשר להחלטתך אם להגדיר פורמט או לא, עליך לתעד את כוונותיך באופן ברור. אם תגדיר את הפורמט, עליך לעשות זאת באופן מדויק. לדוגמה, הנה השיטה toString שמצורפת למחלקה PhoneNumber, כפי שראינו בנושא 8:

```
/**
  *Returns the string representation of this phone number.
  *The string consists of fourteen characters whose fo mat
  *is "(XXX)YYY-ZZZZ",where XXX is the a ea code,YYY is
  *the exchange,and ZZZZ is the extension. (Each of the
  *capital lette s represents a single decimal digit.)
  *
  *If any of the three parts of this phone number is too small
  *to fill up its field,the field is padded with leading zeros.
  *For example,if the value of the extension is 123, the last
  *four cha acters of the string representation will be "0123".
  *
  *Note that there is a single space separating the closing
  *parenthesis after the area code from the first digit of the
  *exchange.
  */
public String toString(){
    return "("+toPaddedString(areaCode,3)+")"+
            toPaddedString(exchange,3)+"-"+
            toPaddedString(extension,4);
}
```

```
/**
  *Translates an int to a string of the specified length,
  *padded with leading zeros.Assumes i >=0,
  *1 <=length <=10,and Integer.toString(i)<=length.
  */
private static String toPaddedString(int i,int length){
     String s =Integer.toString(i);
     return ZEROS [length -s.length()]+s;
}
private static String [ ] ZEROS =
     {"","0","00","000","0000","00000",
       "000000","0000000","00000000","000000000"};
```

אם החלטת לא להגדיר פורמט, הערת התיעוד צריכה להציג משהו דומה לזה:

```
/**
  *Returns a brief desc iption of this potion.The exact details
  *of the representation are unspecified and subject to change,
  *but the following may be regarded as typical:
  *
  *"[Potion #9:type=love,smell=turpentine,look=india ink ]"
  */
public String toString() {...}
```

אחרי קריאת הערה זו, מתכנתים המייצרים קוד או נתונים מתמידים אשר תלויים בפרטי הפורמט, לא יוכלו להאשים אף אחד פרט לעצמם כשהפורמט ישתנה.

ללא קשר לשאלה אם אתה מגדיר את הפורמט או לא, **תמיד כדאי לספק גישה תכנותית לכל המידע שכלול בערך שמוחזר על ידי toString**. לדוגמה, המחלקה PhoneNumber צריכה לכלול שיטות לגישה אל קידומת מספר הטלפון (קוד אזור), המרה והרחבה. אם לא תעשה זאת, אתה כופה על מתכנתים שצריכים את המידע הזה, לחלק את המחרוזת לקטעים כדי לפענח אותה. בנוסף לפגיעה בביצועים ויצירת עבודה מיותרת למתכנתים, תהליך זה יכול לגרום לשגיאות, ומתבטא במערכות לא יציבות, הקורסות בעת שינוי הפורמט. העובדה שאינך מספק אמצעי גישה הופכת את המחרוזת למעשה ל-API, אפילו אם הגדרת שהיא ניתנת לשינוי.

נושא 10: דרוס את Clone בתבונה

הממשק Cloneable תוכנן כממשק mixin (נושא 16), כדי שאובייקטים יוכלו לפרסם שהם מתירים יצירת **תואם** או **שיבוט** (cloning) שלהם. למרבה הצער, הוא נכשל במילוי משימה זו. הפגם העיקרי שלו הוא שחסרה לו שיטת clone, והשיטה clone של Object היא מוגנת. אינך יכול לקרוא לשיטה clone עבור אובייקט ללא שימוש בהשתקפות (reflection) (נושא 35), רק בהסתמך על כך כי היא מממשת את Cloneable. אפילו קריאה שקופה תיכשל, מכיון שאין כל הבטחה שיש לאובייקט שיטת clone נגישה. למרות כשל זה ואחרים, נושא זה נמצא בשימוש נרחב, כך שכדאי להבין זאת. בנושא זה מוצג כיצד לממש שיטת clone טובה, מוסבר מתי מתאים לעשות זאת וניתנות חלופות שונות בקצרה.

אז מה עושה הממשק Cloneable, בהתחשב בכך שהוא אינו מכיל שיטות כלשהן? הוא קובע את התנהגות השיטה clone המוגנת של Object: אם מחלקה מממשת את Cloneable, השיטה clone של Object מחזירה העתק שדה-אחר-שדה של האובייקט. אחרת, הוא זורק את CloneNotSupportedException. זהו שימוש מאוד יוצא דופן של ממשקים ואין לחקות אותו. באופן רגיל, מימוש ממשק אומר משהו על מה שהמחלקה יכולה לעשות למען לקוחותיה, ובמקרה זה Cloneable משנה את ההתנהגות של שיטה מוגנת השייכת למחלקת העל.

כדי לממש את הממשק Cloneable כך שתהיה לו השפעה כלשהי על מחלקה, הוא וכל מחלקות-העל שלו חייבים לציית לפרוטוקול גדול, מורכב למדי, בלתי ניתן לאכיפה ולא מתועד ברובו. המנגנון המתקבל הוא יוצא דופן בשפה: הוא יוצר אובייקט בלי לקרוא לבנאי.

ההסכם הכללי של השיטה clone אינו מספיק מפורט במקרה זה. הנה הוא, מתורגם מהמפרט של java.lang.Object:

> השיטה יוצרת ומחזירה עותק של אובייקט זה. המשמעות המדויקת של "עותק" תלויה במחלקה של האובייקט. הכוונה הכללית היא שעבור כל אובייקט x, הביטוי:

```
x.clone() != x
```

> יהיה true, והביטוי:

```
x.clone().getClass() == x.getClass()
```

> יהיה true, אבל אלו אינן דרישות מוחלטות. למרות שבדרך כלל המקרה ש-

```
x.clone().equals(x)
```

> יהיה true, אין זו דרישה מוחלטת. בדרך כלל, העתקת אובייקט תגרור יצירת מופע חדש של המחלקה שלו, אבל ייתכן שהדבר גם ידרוש העתקה של מבני נתונים פנימיים. לא נקראים בנאים כלשהם.

יש מספר בעיות עם הסכם זה. התנאי ש"לא נקראים בנאים כלשהם" הוא חזק מדי. שיטה clone שמתפקדת כראוי יכולה לקרוא לבנאים כדי ליצור אובייקטים פנימיים עבור clone בבנייה. אם המחלקה היא סופית, clone יכולה אפילו להחזיר אובייקט שנוצר על ידי בנאי.

התנאי ש-()x.clone().getClass צריך בדרך כלל להיות זהה ל-x.getClass הוא חלש מדי. למעשה, מתכנתים מניחים שאם הם מרחיבים את המחלקה וקוראים ל-super.clone מתוך תת-המחלקה, האובייקט המוחזר יהיה מופע של תת-המחלקה. הדרך **היחידה** שבה מחלקת-על יכולה לספק יכולת זו, היא שהשיטה clone שלה תחזיר אובייקט שהתקבל על ידי קריאה ל-super.clone. אם השיטה clone תחזיר אובייקט שנוצר על ידי בנאי, הוא יהיה מטיפוס מחלקה שגוי. לכן, **אם אתה דורס את השיטה clone במחלקה לא-סופית, עליך להחזיר אובייקט שהושג על ידי קריאה ל-super.clone**. אם כל מחלקות-העל של מחלקה מצייתות לכלל זה, אז קריאה ל-super.clone תביא לבסוף לקריאה לשיטה clone של Object, תוך יצירת מופע של המחלקה הנכונה. המנגנון הזה דומה בצורה מעורפלת לשרשור בנאי אוטומטי, אלא שהוא לא נאכף.

בגירסה 1.3, הממשק Cloneable אינו מבהיר את האחריות שמחלקה לוקחת כאשר היא מממשת ממשק זה. המפרט אינו אומר דבר, מעבר לדרך שבה מימוש הממשק משפיע על ההתנהגות של מימוש clone של Object. **למעשה, מחלקה שיש בה מימוש של Cloneable צפויה לספק שיטת clone ציבורית מתפקדת היטב**. ברוב המקרים אי אפשר לעשות זאת, אלא אם כן כל מחלקות-העל של המחלקה מספקות מימוש clone שמתפקד היטב, בין אם הוא ציבורי או מוגן.

נניח שאתה רוצה לממש Cloneable במחלקה אשר מחלקות-העל שלה מספקות שיטות clone שמתפקדות היטב. האובייקט שאתה מקבל מ-()super.clone יכול להיות או לא להיות קרוב למה שתחזיר לבסוף, תלוי באופי המחלקה. האובייקט הזה יהיה מבחינתה של כל מחלקת-על של אובייקט, תואם לאובייקט המקורי ומתפקד באופן מלא. לשדות שהוצהרו במחלקה שלך (אם ישנם), יהיו ערכים זהים לאלה של האובייקט התואם. אם כל שדה מכיל ערך פשוט או ייחוס לאובייקט שאינו ניתן לשינוי, האובייקט המוחזר יכול להיות בדיוק מה שאתה צריך, באותו המקרה שבו אין צורך בעיבוד נוסף. זהו המקרה, לדוגמה, עבור מחלקת PhoneNumber שהוצגה בנושא 8. במקרה זה, כל שאתה צריך הוא לספק גישה לשיטה clone המוגנת של Object:

```
public Object clone(){
    try {
        return super.clone();
    } catch(CloneNotSupportedException e) {
        throw new Error ("Assertion failure");//Can't happen
    }
}
```

אם, בכל זאת, האובייקט שלך מכיל שדות המתייחסים לאובייקטים משתנים או ניתנים לשינוי, השימוש במימוש clone זה עלול לגרור תקלה חמורה. לדוגמה, חשוב על המחלקה Stack שראית בנושא 5:

```
public class Stack {
    private Object[] elements;
    private int size = 0;

    public Stack(int initialCapacity) {
        this.elements = new Object[initialCapacity];
    }

    public void push(Object e) {
        ensureCapacity();
        elements[size++] = e;
    }

    public Object pop() {
        if (size == 0)
            throw new EmptyStackException();
        Object result = elements[--size];
        elements[size] = null; // Eliminate obsolete reference
        return result;
    }

    // Ensure space for at least one more element.
    private void ensureCapacity() {
        if (elements.length == size) {
            Object oldElements[] = elements;
            elements = new Object[2 * elements.length + 1];
            System.arraycopy(oldElements, 0, elements, 0, size);
        }
    }
}
```

נניח שאתה רוצה לגרום לכך שניתן יהיה ליצור תואם (עותק) של מחלקה זו. אם השיטה clone שלה רק מחזירה super.clone(), למופע Stack שיתקבל יהיה את הערך הנכון בשדה size שלו, אבל שדה elements שלו יתייחס לאותו מערך כמו מופע Stack המקורי. שינוי של המקור יגרום לפגיעה בערכים הבלתי משתנים במחלקה התואמת, ולהיפך. מהר מאוד תגלה שהתוכנית שלך מייצרת תשובות לא הגיוניות או מעוררת את החריג ArrayIndexOutOfBoundsException.

המצב לא היה מתרחש לעולם כתוצאה מקריאה לבנאי במחלקת Stack. **בפועל, השיטה clone מתפקדת כבנאי נוסף. עליך להבטיח שהיא אינה פוגעת באובייקט המקורי ושהיא מציבה ערכים בלתי משתנים באובייקט התואם**. כדי שהשיטה clone שמופעלת על Stack תפעל כראוי, היא חייבת להעתיק את התכונות הפנימיות של המחסנית. הדרך הכי קלה לעשות זאת היא על ידי קריאה רקורסיבית ל-Clone עבור המערך elements:

```
public Object clone()throws CloneNotSupportedException {
    Stack result =(Stack)super.clone();
    result.elements =(Object []) elements.clone();
    return result;
}
```

שים לב לכך שפתרון זה לא היה פועל אם השדה elements היה סופי, כי השיטה clone היתה מנועה מהקצאת ערך חדש לשדה. זוהי בעיה מהותית: **ארכיטקטורת clone אינה מתאימה לשימוש רגיל בשדות סופיים ביחס לאובייקטים משתנים**, למעט מקרים בהם האובייקטים המשתנים יכולים להיות משותפים באופן בטוח בין אובייקט והתואם (עותק) שלו. כדי שאפשר יהיה ליצור תואם למחלקה, ייתכן שיהיה הכרח להסיר מצייני final מכמה שדות.

לא תמיד מספיק לקרוא ל-clone באופן רקורסיבי. לדוגמה, נניח שאתה כותב שיטת clone לטבלת ערבול אשר התכונות הפנימיות שלה מורכבות ממערך של מנות, שכל אחת מהן מייחסת את הכניסה הראשונה ברשימה מקושרת של זוגות ערכי-מפתח או שהיא null אם המנה ריקה. כדי לשפר ביצועים, המחלקה מממשת את הרשימה המקושרת החד-כיוונית וקלת המשקל שלה, במקום להשתמש ב-java.util.LinkedList מלכתחילה:

```
public class HashTable implements Cloneable {
   private Entry[] buckets =  ...;

   private static class Entry {
      Object key;
      Object value;
      Entry  next;

      Entry(Object key, Object value, Entry next) {
         this.key   = key;
         this.value = value;
         this.next  = next;
      }
   }
   ... // Remainder omitted
}
```

נניח שאתה יוצר עותק של מערך המנות באופן רקורסיבי, כמו שעשינו ל-Stack:

```
//Broken  -  results in shared internal state!
public Object clone()throws CloneNotSupportedException {
    HashTable result =(HashTable)super.clone();
    result.buckets =(Ent y [])buckets.clone();
    return result;
}
```

למרות שלעותק שיצרת יש מערך מנות נפרד, מערך זה מייחס את אותה רשימה מקושרת כמו המקורי, אשר יכול בקלות לגרום להתנהגות בלתי צפויה גם בעותק וגם במקורי. כדי לתקן בעיה זו, עליך להעתיק את הרשימה המקושרת אשר כוללת כל מנה באופן נפרד. הנה גישה אחת מקובלת:

```
public class HashTable implements Cloneable {
   private Entry[] buckets =  ... ;

   private static class Entry {
      Object key;
      Object value;
      Entry  next;

      Entry(Object key, Object value, Entry next) {
         this.key   = key;
         this.value = value;
         this.next  = next;
      }

      // Recursively copy the linked list headed by this Entry
      Entry deepCopy() {
         return new Entry(key, value,
            next == null ? null : next.deepCopy());
      }
   }
   public Object clone()throws CloneNotSupportedException {
      HashTable result =(HashTable)supe .clone();
      result.buckets =new Entry [buckets.length ];
      for (int i =0;i <buckets.length;i++)
           if (buckets [i ] !=null)
              result.buckets [i ] =(Entry)
                   buckets [i ].deepCopy();
      return result;
   }
   ...// Remainder omitted
}
```

המחלקה הפרטית HashTable.Entry הוגדלה כדי לתמוך בנוהל "העתקה עמוקה". השיטה clone הפועלת על HashTable מקצה מערך buckets חדש מהגודל המתאים, וחוזרת שוב ושוב על מערך buckets המקורי ומבצעת העתקה עמוקה לכל מנה שאינה ריקה. שיטת ההעתקה העמוקה על Entry קוראת לעצמה באופן רקורסיבי, כדי להעתיק את כל הרשימה המקושרת החל מ- Entry. למרות שטכניקה זו חמודה למדי ופועלת בסדר כאשר המנות אינן ארוכות מדי, אין זו דרך טובה ליצור עותק תואם של רשימה מקושרת, מכיון שהדבר מחייב מסגרת מחסנית אחת לכל אלמנט ברשימה. אם הרשימה ארוכה, הדבר עלול בקלות לגרום לגלישת מחסנית. כדי למנוע זאת, אתה יכול להחליף את הרקורסיה ב-deepCopy עם איטרציה:

```
// Iteratively copy the linked list headed by this Entry
    Entry deepCopy() {
        Entry result = new Entry(key, value, next);

        for (Entry p = result; p.next != null; p = p.next)
            p.next = new Entry(p.next.key, p.next.value, p.next.next);
        return result;
}
```

גישה סופית ליצירת עותקים של אובייקטים מורכבים היא לקרוא ל-super.clone, להחזיר את כל השדות באובייקט המתקבל למצבם הטהור ואז לקרוא לשיטות מרמה גבוהה יותר כדי לתקן את מצב האובייקט. במקרה של הדוגמה Hashtable, השדה buckets יאותחל למערך מנה חדש, והשיטה put(key, value) (לא הוצגה) תיקרא עבור כל ערך מפתח שממופה בטבלת הערבול שיוצרים ממנה עותק תואם. הגישה הזאת מניבה בדרך כלל שיטת clone אלגנטית, פשוטה והגיונית, שאינה פועלת מהר כמו השיטה המתמרנת ישירות את ה"קרביים" של האובייקט ושל התואם שלו.

כמו בנאי, גם השיטה clone אינה צריכה לקרוא לשיטות לא-סופיות כלשהן על התואם שנמצא בתהליך בנייה (נושא 15). אם clone קוראת לשיטה דרוסה, שיטה זו תתבצע לפני שלתת-המחלקה שבה היא מוגדרת יהיה סיכוי לתקן את מצבה בתואם, וייתכן שהדבר יוביל להשחתה של התואם ושל המקור. לכן, השיטה put(key, value) (שנדונה בפיסקה הקודמת) צריכה להיות או סופית או פרטית (אם היא פרטית, היא ככל הנראה "שיטת העזרה", helper method, לשיטה ציבורית לא-סופית). השיטה clone של Object מוכרזת כעלולה לזרוק את החריג CloneNotSupportedException, אבל שיטות הדורסות את clone יכולות להשמיט הגדרה זו. רצוי ששיטות הדורסות את clone במחלקות **סופיות** ישמיטו את ההכרזה על החריג, כי שיטות אשר אינן מסומנות כעלולות לזרוק חריגים נוחות יותר לשימוש, מכיון שאנו חוסכים מהלקוח את הטיפול המיוחד בחריגה המסומנת (נושא 41). לעומת זאת, במחלקות **שאינן סופיות**, ובמיוחד במחלקות שתוכננו להורשה (נושא 15), רצוי בכל זאת להכריז על הפונקציה הדורסת

של clone כעלולה לזרוק את החריג CloneNotSupportedException. כך נוכל לאפשר יצירת תת-מחלקה שמופעיה (subclasses) יהיו בלתי ניתנים לשכפול על ידי אספקת שיטת ה-clone הבאה בתת-המחלקה:

```
//Clone method to guarantee that instances cannot be cloned
public final Object clone()throws CloneNotSupportedException {
    throw new CloneNotSupportedException();
}
```

אם היינו מנסים לדרוס באופן זה את השיטה clone בתת-המחלקה, כאשר השיטה clone במחלקת-העל אינה מוכרזת כעלולה לזרוק את החריגה, היתה מתקבלת שגיאת קומפילציה, כיון ששיטה דורסת אינה יכולה לזרוק חריגות מסומנות שאינן מוכרזות על ידי השיטה אותה היא דורסת. אמנם השיטה clone של תת-מחלקה שאינה רוצה שיצרו לה תואם, יכולה תמיד לזרוק חריגה לא מסומנת, כמו למשל UnsupportedOperationException, אך הניסיון המעשי אומר שהחריג CloneNotSupportedException הוא החריג הנכון שיש לזרוק בנסיבות אלו.

הבה נסכם את המסקנות שהגענו אליהן עד כה. כל המחלקות שיש בהן מימוש של Cloneable צריכות לדרוס את clone על ידי שיטה ציבורית. שיטה ציבורית זו צריכה לקרוא תחילה ל-super.clone ואז לתקן כל שדה שזקוק לתיקון. בדרך כלל, משמעות הדבר היא העתקת כל תת-האובייקטים הניתנים לשינוי אשר מהווים את המבנה הפנימי של האובייקט המשוכפל, והחלפת הייחוסים לאובייקטים אלה עם הייחוסים לעותקים שיצרנו. למרות שעותקים פנימיים אלה יכולים בדרך כלל להיווצר על ידי קריאה רקורסיבית ל-clone, אין זו תמיד הגישה הטובה ביותר. אם המחלקה מכילה שדות פשוטים בלבד או ייחוסים לאובייקטים בלתי משתנים, אז אף שדה לא מצריך תיקון. יש חריגים לכלל זה. לדוגמה, שדה שמייצג מספר סידורי או ID ייחודי אחר, או שדה שמייצג את זמן היצירה של האובייקט יצריכו תיקון, אפילו אם הוא פשוט או אינו ניתן לשינוי.

האם כל המורכבות הזאת באמת חיונית? לעיתים נדירות. אם אתה מרחיב מחלקה המממשת את Cloneable, אין לך הרבה ברירות, אלא לממש את השיטה clone שמתפקדת היטב. אחרת, **קרוב לוודאי שיותר טוב שתספק כמה אמצעים חילופיים להעתקת אובייקטים, או שלא תספק יכולת זו**. לדוגמה, אין זה הגיוני כל כך שמחלקות שאינן ניתנות לשינוי יתמכו בהעתקת אובייקטים, כי העותקים יהיו בלתי ניתנים להבדלה מהמקור.

גישה טובה להעתקת אובייקטים היא לספק בנאי העתקה (copy constructor). בנאי העתקה הוא למעשה בנאי המקבל ארגומנט יחיד שהטיפוס שלו הוא המחלקה אשר כוללת אותו, לדוגמה:

```
public Yum(Yum yum);
```

גישה מעט שונה היא לספק שיטה יוצרת סטטית (static factory) ולא בנאי:

```
public static Yum newInstance(Yum yum);
```

לגישת בנאי ההעתקה וגרסת השיטה היוצרת הסטטית (static factory) שלו יש הרבה יתרונות על פני Cloneable/clone: הם אינם סומכים על מנגנון יצירת אובייקטים מחוץ לנהלי השפה שיש בהם סיכון רב. הם אינם מחייבים אכיפה למוסכמות שתועדו בצורה גרועה. הם אינם מתנגשים עם השימוש הנכון בשדות סופיים והם אינם דורשים מהלקוח לתפוס חריגים מסומנים שאינם דרושים. למרות שאי אפשר לשלב בנאי העתקה או שיטה יוצרת סטטית בממשק, הממשק Cloneable אינו מצליח לתפקד כממשק כי אינו מכיל שיטה clone ציבורית. לכן, אינך מוותר על יכולת הממשק על ידי שימוש בבנאי העתקה במקום בשיטה clone.

יתרה מזאת, בנאי העתקה (או יצרן סטטי) יכול לקבל ארגומנט שהטיפוס שלו הוא ממשק תואם שמומש על ידי המחלקה. לדוגמה, כמוסכמה, כל המימושים של אוסף כללי מספקים בנאי העתקה אשר הארגומנט שלו הוא מהטיפוס Collection או Map. בנאי העתקה מבוססי ממשק מרשים ללקוח לבחור את המימוש של העותק, ואינם מכריחים אותו לקבל את המימוש של המקור. לדוגמה, נניח שיש לך LinkedList 1 ואתה רוצה להעתיק זאת ולתת שם חדש: ArrayList. השיטה clone אינה מאפשרת יכולת זו, אבל קל לעשות זאת בעזרת בנאי העתקה: new ArrayList(1).

בהתייחס לכל הבעיות שקשורות ל-Cloneable, יהיה זה בטוח לומר שממשקים אחרים אינם צריכים להרחיב אותו ושמחלקות שתוכננו להורשה (נושא 15) אינן צריכות לממש אותו. בגלל הליקויים הרבים שלו, מתכנתים מומחים רבים בוחרים שלא לדרוס את השיטה clone לעולם, ולעולם גם לא לקרוא לה, אולי חוץ ממקרה של העתקת מערכים בצורה פשוטה. שים לב לכך שאם אינך מספק שיטת clone מוגנת מתפקדת היטב עבור מחלקה שתוכננה להורשה, תת-המחלקה שלה לא תוכל לממש את Cloneable (בגלל כל אותן בעיות שנידונו בנושא זה).

נושא 11: שקול לממש את ממשק Comparable

שלא כמו השיטות האחרות שנדונו בפרק זה, השיטה compareTo אינה מוצהרת ב-Object. למעשה, זוהי השיטה היחידה בממשק java.lang.Comparable.
היא דומה במפרט שלה לשיטה equals של Object, פרט לכך שזו מאפשרת השוואות סדר (order comparisons) בנוסף להשוואות רגילות. על ידי מימוש של Comparable, המחלקה מציינת שלמופעים שלה יש **סדר טבעי** (natural ordering). מיון מערך של אובייקטים המממש Comparable הוא פשוט כמו זה:

```
Array.sort(a);
```

בדרך דומה קל לחפש, לחשב ערכים קיצוניים ולשמור על אוספים ממוינים אוטומטית של אובייקטי Comparable. לדוגמה, התוכנית שלהלן, המסתמכת על העובדה ש-String מממשת את Comparable, מדפיסה רשימה אלפביתית של ארגומנטי שורת הפקודה שלה, ללא כפילויות:

```
public class WordList {
   public static void main(String[] args) {
      Set s = new TreeSet();
      s.addAll(Arrays.asList(args));
      System.out.println(s);
   }
}
```

על ידי מימוש Comparable, אתה מאפשר למחלקה לפעול באופן משולב עם כל מימושי האוסף והאלגוריתמים הרבים אשר תלויים בממשק הזה. אתה מקבל עוצמה במאמץ קטן. בפועל, כל מחלקות הערך בספריות הפלטפורמה של java מממשות את Comparable. אם אתה כותב מחלקת ערך עם סדר טבעי פשוט, כמו סדר אלפביתי, סדר מספרי או סדר כרונולוגי, כדאי מאוד שתשקול מימוש של הממשק הזה. כעת תקבל הדרכה כיצד לעשות זאת.

ההסכם הכללי של השיטה compareTo דומה באופיו לזה של השיטה equals. הנה הוא מובא להלן, מתורגם מהמפרט של Comparable:

> השיטה משווה את האובייקט הזה עם אובייקט שצוין כדי לקבוע את הסדר. השיטה מחזירה שלם שלילי, אפס או שלם חיובי אם האובייקט קטן מ, שווה או גדול מהאובייקט שצוין, בהתאמה. השיטה זורקת ClassCastException אם הטיפוס של האובייקט שצוין לא מאפשר את השוואתו לאובייקט הזה.

בתיאור הבא, הציון (ביטוי)sgn מציין את הפונקציה המתמטית signum אשר מוגדרת להחזיר 1-, 0 או 1 אם ערך הביטוי הוא שלילי, אפס או חיובי, בהתאמה.

הממשמ חייב להבטיח sgn(x.compareTo(y)) == -sgn(y.compareTo(x)) לכל x ו-y (הדבר מרמז ש-x.compareTo(y) חייב לעורר חריג, אם ורק אם y.compareTo(x) מעורר חריג).

- הממשמ חייב גם להבטיח שהקשר הוא טרנזיטיבי: (x.compareTo(y) > 0 && y.compareTo(z) > 0) משמעו x.compareTo(z) > 0.
- לבסוף, הממשמ חייב להבטיח ש- x.compareTo(y) == 0 משמעו ש- sgn(x.compareTo(z)) == sgn(y.compareTo(z)) לכל z.
- מאוד מומלץ, אבל לא חובה, ש- (x.compareTo(y)==0) == (x.equals(y)). בדרך כלל, כל מחלקה שיש בה מימוש של הממשק Comparable ומפרה את התנאי הזה, צריכה להעיד על עובדה זו בבירור. הניסוח המומלץ הוא: "שים לב: למחלקה זו יש סידור טבעי אשר אינו עקבי עם equals".

אל תירתע מהנוסח המתמטי של הסכם זה. כמו הסכם equals (נושא 7), גם הסכם compareTo אינו כל כך מסובך כפי שהוא נראה. בתוך אותה מחלקה, כל יחס של סידור הגיוני יענה על דרישות הסכם compareTo. לגבי השוואה עם מחלקות אחרות, הסכם compareTo, שלא כמו הסכם equals, אינו חייב להיות תקף: הוא מורשה לעורר ClassCastException אם שני ייחוסי האובייקטים שמושווים פונים אל אובייקטים ממחלקות שונות. בדרך כלל, זה בדיוק מה ש-compareTo צריך לעשות בנסיבות אלו. למרות שההסכם אינו מוציא מהכלל השוואות בין מחלקתיות, אין מחלקות בספריות הפלטפורמה של Java אשר תומכות בהן, נכון לגירסה 1.4.

כמו שמחלקה המפירה את הסכם hashCode יכולה לפגוע במחלקות אחרות שתלויות בערבול, כך מחלקה המפירה את הסכם compareTo יכולה לפגוע במחלקות אחרות שתלויות בהשוואה. מחלקות שתלויות בהשוואה כוללות את האוספים הממוינים TreeSet ו-TreeMap ואת מחלקות השירות Collections ו-Arrays, אשר יש בהן אלגוריתמים לחיפוש ומיון.

הבה נעבור על תנאי הסכם compareTo. התנאי הראשון (סימטריות) אומר שאם אתה הופך את כיוון ההשוואה בין שני ייחוסי אובייקטים, הדבר הצפוי קורה: אם האובייקט הראשון קטן מהשני, אז השני חייב להיות גדול מהראשון. אם האובייקט הראשון שווה לאובייקט השני, אז האובייקט השני חייב להיות שווה לראשון. אם האובייקט הראשון גדול מהשני, אז האובייקט השני חייב להיות קטן מהאובייקט הראשון. התנאי השני (טרנזיטיביות) אומר שאם אובייקט אחד גדול מאובייקט שני והאובייקט השני גדול מאובייקט שלישי, אז האובייקט הראשון חייב להיות גדול מהאובייקט השלישי. התנאי המסיים אומר שכל האובייקטים אשר פעולת ההשוואה compareTo עליהם מחזירה שהם שווים, חייבים לתת את אותן התוצאות כאשר הם מושווים לכל אובייקט אחר.

תוצאה אחת של שלוש הדרישות האלו היא שמבחן השוויון שמבוצע על ידי השיטה compareTo חייב לציית לאותן ההגבלות שנאכפות על ידי הסכם equals: רפלקסיביות, סימטריות, טרנזיטיביות ואי-איפוס. לכן, חלה כאן אותה אזהרה: אין כל דרך לרשת ממחלקה שניתן ליצור ממנה מופעים בזמן שמקיימים את הסכם compareTo (נושא 7). אותו פתרון עקיף חל גם כאן. אם אתה רוצה להוסיף היבט משמעותי למחלקה אשר מממשת את Comparable, אל תרחיב אותה בהורשה. כתוב מחלקה ללא ייחוס המכילה שדה של המחלקה הראשונה. לאחר מכן, ספק שיטת "view" (״הצגה״) אשר מחזירה שדה זה. הדבר מאפשר חופש מימוש של כל השיטה compareTo עבור המחלקה השנייה, בזמן שהדבר מאפשר ללקוח שלה לראות מופע של המחלקה השנייה בתור מופע של המחלקה הראשונה, בעת הצורך.

הפיסקה האחרונה של הסכם compareTo, היא המלצה חמה, יותר מאשר תנאי אמיתי. היא קובעת שמבחן השוויון אשר נאכף על ידי השיטה compareTo צריך בדרך כלל להחזיר את אותן התוצאות כמו השיטה equals. אם תציית לתנאי זה, הסידור שנאכף על ידי השיטה compareTo **אמור להיות עקבי עם equals**. אם התנאי מופר, הסידור אמור להיות **לא-עקבי עם equals**. מחלקה אשר השיטה compareTo שלה אוכפת סדר שאינו עקבי עם equals עדיין תפעל, אבל ייתכן שאוספים ממוינים הכוללים אלמנטים של המחלקה, לא יצייתו להסכם הכללי של ממשקי האוסף המתאימים (Collection, Set או Map). הדבר נובע מכך שההסכמים הכלליים של ממשקים אלה מוגדרים במונחים של השיטה equals, אבל אוספים ממוינים משתמשים במבחן השוויון אשר נאכף על ידי compareTo ולא על ידי equals. לא תהיה כל תקלה אם הדבר יתבצע, אבל צריך להיות ערניים לכך.

לדוגמה, חשוב על המחלקה Bigdecimal, אשר השיטה compareTo שלה אינה עקבית עם equals. אם אתה יוצר HashSet ומוסיף new Bigdecimal("1.0") ו- new Bigdecimal("1.00"), הקבוצה תכיל שני אלמנטים, כי שני מופעי Bigdecimal שהוספו לקבוצה אינם שווים כאשר הם מושווים על ידי השיטה equals. בכל מקרה, אם תבצע את אותה הפרוצדורה על ידי שימוש ב-TreeSet במקום ב-HashSet, הקבוצה תכיל אלמנט אחד בלבד, כי שני מופעי Bigdecimal שווים כאשר משתמשים בשיטה compareTo (מידע נוסף תמצא בתיעוד של Bigdecimal).

כתיבת השיטה compareTo דומה לכתיבת השיטה equals, אבל יש כמה הבדלים חשובים. אינך צריך לבדוק את טיפוס הארגומנט לפני ההמרה. אם הארגומנט אינו מהטיפוס המתאים, השיטה compareTo **צריכה** לעורר ClassCastException. אם הארגומנט הוא null, השיטה compareTo **צריכה** לעורר NullPointerException. זו בדיוק ההתנהגות שתקבל אם רק תמיר את הארגומנט לטיפוס הנכון, ואז תנסה לגשת לאיברים שלו.

השוואות השדה עצמן הן השוואות סדר יותר מאשר השוואות שוויון. השווה שדות ייחוס אובייקט על ידי קריאה רקורסיבית לשיטה compareTo. אם השדה אינו מממש Comparable או שאתה צריך להשתמש בסידור שאינו מקובל, אתה יכול להשתמש ב-Comparator מפורש. תוכל גם לכתוב אחד משלך, או שתשתמש באחד נתון כמו בשיטה CompareTo הזו עבור המחלקה CaseInsensitiveString שבנושא 7:

```
public int compareTo(Object o) {
      CaseInsensitiveString cis = (CaseInsensitiveString)o;
      return String.CASE_INSENSITIVE_ORDER.compare(s, cis.s);
}
```

השווה שדות פשוטים על ידי שימוש באופרטורי היחס > ו-< ובמערכים, על ידי החלת קווים מנחים אלה לכל אלמנט. אם למחלקה יש שדות משמעותיים כפולים, הסדר שבו אתה משווה אותם הוא קריטי. אתה חייב להתחיל את ההשוואה בשדה המשמעותי ביותר ולהמשיך הלאה בסדר יורד. אם השוואה מסתיימת באפס (המייצג שוויון) - סיימת, וכעת החזר את התוצאה. אם השדות המשמעותיים ביותר שווים, המשך הלאה והשווה את השדות המשמעותיים פחות לפי הסדר, וכן הלאה. אם כל השדות שווים, האובייקטים שווים - החזר אפס. הטכניקה מודגמת על ידי השיטה compareTo הזאת הפועלת על המחלקה PhoneNumber שבנושא 8:

```
public int compareTo(Object o) {
      PhoneNumber pn = (PhoneNumber)o;

      // Compare area codes
      if (areaCode < pn.areaCode)
         return -1;
      if (areaCode > pn.areaCode)
         return  1;

      // Area codes are equal, compare exchanges
      if (exchange < pn.exchange)
         return -1;
      if (exchange > pn.exchange)
         return  1;

      // Area codes and exchanges are equal, compare extensions
      if (extension < pn.extension)
         return -1;
      if (extension > pn.extension)
         return  1;

      return 0;  // All fields are equal
   }
```

למרות ששיטה זו פועלת מצוין, אפשר לשפר אותה. זכור שההסכם של compareTo אינו מציין את סדר הגודל של הערך המוחזר, אלא רק את הסימן. תוכל לנצל זאת כדי לפשט את הקוד, וייתכן שגם מהירות הביצוע תשופר במקצת:

```
public int compareTo(Object o){
    PhoneNumber pn =(PhoneNumber)o;
    //Compare area codes
    int areaCodeDiff =areaCode -pn.areaCode;
    if (areaCodeDiff !=0)
        return areaCodeDiff;
    //Area codes are equal,compare exchanges
    int exchangeDiff =exchange -pn.exchange;
    if (exchangeDiff !=0)
        return exchangeDiff;
    //Area codes and exchanges are equal,compare extensions
    return extension -pn.extension;
}
```

טריק זה עובד בסדר בדוגמה הזו, אבל צריך להשתמש בו בזהירות יתרה. אל תשתמש בו, אלא אם כן אתה בטוח שהשדה שאתה מטפל בו אינו יכול להיות שלילי, או יותר נפוץ, שההבדל בין ערכי השדה הגדולים ביותר האפשריים והערכים הקטנים ביותר האפשריים הוא פחות מ/שווה ל- (INTEGER.MAX_VALUE(231 - 1. הסיבה שטריק זה אינו פועל באופן כללי היא, שערך שלם מסומן בן 32 סיביות אינו גדול מספיק כדי לייצג את ההבדל בין שני שלמים שרירותיים בני 32 סיביות. אם i הוא int גדול חיובי ו-j הוא int גדול שלילי, (i-j) יגלוש ויחזיר ערך שלילי. השיטה compareTo שתתקבל לא תפעל, היא תחזיר תוצאות אבסורדיות עבור כמה ארגומנטים ותפר את התנאים הראשון והשני של הסכם compareTo. אין זו בעיה תיאורטית בלבד, היא גרמה לכשלונות במערכות אמיתיות. קשה לנפות באגים אלה, מכיון שהשיטה compareTo המשובשת הזו עשויה לפעול בצורה תקינה עבור רוב ערכי קלט.

פרק 4

מחלקות וממשקים

מחלקות וממשקים הם ליבה של שפת התכנות Java, ומהווים את יחידות ההפשטה (abstraction) הבסיסיות שלה. השפה מספקת אלמנטים רבי-עוצמה שניתן להשתמש בהם כדי לעצב מחלקות וממשקים. פרק זה מכיל קווים מנחים שיעזרו לך לעשות את השימוש הטוב ביותר באלמנטים אלה, כך שהמחלקות והממשקים שלך יהיו שימושיים, יציבים וגם גמישים.

נושא 12: מזער את הנגישות אל מחלקות ואיברים

הגורם הבודד החשוב ביותר אשר מבדיל בין מודול מתוכנן היטב לבין מודול שהתכנון שלו לקוי, הוא הדרגה שבה המודול מסתיר את נתוניו הפנימיים ושאר פרטי המימוש שלו ממודולים אחרים. מודול מתוכנן היטב מסתיר את כל פרטי המימוש שלו, מפריד בצורה נקייה בין ה-API שלו לבין המימוש. מודולים יכולים לתקשר זה עם זה רק באמצעות ה-API שלהם ומתעלמים מהפעולה הפנימית אחד של השני. תפישת עבודה זו, הידועה כ**הסתרת מידע** (information hiding) או **כימוס** (encapsulation), היא אחת מעקרונות היסוד של תכנון תוכנה [Parnas72].

הסתרת מידע חשובה בגלל סיבות רבות, רובן נובעות מהעובדה שהדבר מגדיל באופן יעיל את **ההפרדה** בין המודולים, אשר מאפשרת פיתוח, בדיקה, שיפור, אופטימיזציה, שימוש והבנה של כל מודול בנפרד, וללא תלות זה בזה. כך ניתן להאיץ את פיתוח המערכת, מכיון שאפשר לפתח מודולים במקביל על ידי מתכנתים וצוותים שונים. כך גם ניתן להקל את נטל התחזוקה כי אפשר להבין מודולים במהירות ולנפות מהם שגיאות עם חשש מועט מפני פגיעה במודולים אחרים. למרות שהסתרת מידע אינה גורמת כשלעצמה לביצועים טובים, היא מאפשרת כיוונון ביצועים יעיל. לאחר שהמערכת הושלמה ובחינת הפרופיל שלה קבעה איזה מודולים גורמים לבעיות ביצועים (נושא 37), אפשר לשפר מודולים אלה מבלי להשפיע על מודולים אחרים או לפגוע בהם. הסתרת מידע מגדילה את האפשרות לעשות בתוכנה שימוש חוזר, כי מודולים יחידים אינם תלויים זה בזה, ולעיתים קרובות מוכיחים שימושיות בהקשר אחר מזה שבשבילו פותחו.

לבסוף, הסתרת מידע מקטינה את הסיכון בעת הקמת מערכות גדולות. מודולים יחידים יכולים לפעול באופן תקין וטוב גם אם המערכת אינה מתפקדת כראוי.

לשפת התכנות Java יש אמצעים רבים לסייע בהסתרת מידע. אמצעי אחד שכזה הוא מנגנון **בקרת ההרשאות** (access control) [JLS, 6.6] הקובע את **הנגישות** (accessibility) של מחלקות, ממשקים ואיברים (members). הנגישות אל ישות נקבעת על פי המיקום שבו היא מוצהרת ועל ידי מגדירי ההרשאה (private, protected ו-public), אם בכלל, שישנם בהצהרת היישות. שימוש נכון במגדירים אלה הינו חיוני להסתרת מידע.

העיקרון המנחה הוא שעליך להפוך כל מחלקה או איבר לבלתי נגישים עד כמה שאפשר. במילים אחרות, עליך להשתמש ברמת ההרשאה האפשרית הנמוכה ביותר שתהיה תואמת או עקבית, לתפקוד הנכון של התוכנה שאתה כותב.

למחלקות וממשקים מהרמה העליונה ביותר (לא מקוננים כמו מחלקות פנימיות למשל), יש שתי רמות גישה אפשריות בלבד: **פרטית-בחבילה** (package-private) **וציבורית** (public). אם אתה מצהיר על מחלקה או ממשק מהרמה העליונה ביותר עם המגדיר public, הם יהיו ציבוריים. אחרת, הם יהיו פרטיים-בחבילה. אם אפשר לגרום למחלקה ולממשק ברמה העליונה להיות פרטיים-בחבילה, אז יש לעשות זאת. על ידי הגדרתם כפרטיים-בחבילה, אתה עושה אותם חלק ממימוש חבילת תוכנה, יותר מאשר API מיובא, ואתה יכול לשנות, להחליף או לסלק אותם בגירסה עוקבת ללא חשש מפגיעה בלקוחות קיימים. אם אתה מגדיר אותם כציבוריים, תיאלץ לתמוך בהם לנצח כדי לשמור על תאימות.

אם תעשה שימוש במחלקה (או בממשק) מהרמה העליונה ביותר המוגדרת "פרטית-בחבילה", מתוך מחלקה בודדת בלבד, כדאי שתשקול הפיכתה למחלקה מקוננת פרטית (או ממשק) של המחלקה שבה נעשה שימוש (נושא 18). הדבר מפחית את הנגישות שלה עוד יותר. עם זאת, לא חשוב כל כך לעשות זאת, כמו שחשוב להפוך מחלקה ציבורית שאינה חיונית למחלקה פרטית-בחבילה, כי מחלקה פרטית-בחבילה היא חלק ממימוש חבילת תוכנה ואינה API.

אם כן, עבור איברים (שדות, שיטות, מחלקות מקוננות וממשקים מקוננים) יש ארבע הרשאות אפשריות, הרשומות להלן בסדר עולה של נגישות:

- **private** (פרטית) - האיבר נגיש רק בתוך המחלקה מהרמה העליונה ביותר שבה הוא מוצהר.
- **package-private** (פרטית-בחבילה) - האיבר נגיש מכל מחלקה בחבילת התוכנה שבה הוא מוצהר. רמה זו ידועה טכנית כהרשאת ברירת מחדל. זו הרמה שתקבל, אם לא תציין רמת גישה כלשהי.
- **protected** (מוגנת) - האיבר נגיש מתת-מחלקה של מחלקה שבה הוא מוצהר (בכפוף לכמה הגבלות [JLS, 6.6.2]) ומכל מחלקה בתוכנה שבה הוא מוצהר.
- **public** (ציבורית) - האיבר נגיש מכל מקום.

לאחר תכנון זהיר של ה-API של המחלקה, עליך להגדיר את כל שאר האיברים כפרטיים. רק אם מחלקה אחרת באותה תוכנה באמת צריכה לגשת לאיבר, הסר את המגדיר private ושנה אותו ל-package-private (פרטי-בחבילה). אם אתה מוצא את עצמך עושה זאת לעיתים קרובות, עליך לבחון מחדש את תכנון המערכת שלך, כדי לראות האם פירוק אחר יניב מחלקות המופרדות זו מזו טוב יותר. הבה נאמר שאיברים פרטיים ואיברים פרטיים-בחבילה הם חלק ממימוש של מחלקה, ובאופן רגיל אינם משפיעים על ה-API המיוצא שלה. עם זאת, שדות אלה יכולים "לדלוף" לתוך ה-API המיוצא אם המחלקה מממשת Serializable (נושאים 54, 55).

עבור איברים של מחלקות ציבוריות, גידול עצום בנגישות מתרחש כאשר רמת הגישה משתנה מפרטית-בחבילה למוגנת. איבר מוגן הוא חלק מה-API המיובא של מחלקה וחייב להיתמך לנצח. יתרה מזאת, איבר מוגן של מחלקה מיובאת מייצג התחייבות ציבורית ליחידת מימוש (נושא 15). הצורך באיברים מוגנים צריך להיות נדיר יחסית.

יש חוק אחד המגביל את יכולתך להפחית את הנגישות של שיטות. אם שיטה דורסת שיטה של מחלקת-על, היא אינה מורשית להיות בעלת רמת נגישות נמוכה יותר בתת-המחלקה מאשר רמת הנגישות שיש לה במחלקת-העל [JLS, 8.4.6.3]. הדבר נחוץ כדי להבטיח שמופע של תת-המחלקה יהיה בר-שימוש בכל מקום שבו מופע של מחלקת-העל הוא בר-שימוש. אם תפר כלל זה, המהדר יפיק הודעת שגיאה כאשר תנסה להדר את תת-המחלקה. מקרה ספציפי של כלל זה הוא שכאשר מחלקה מממשת ממשק, כל שיטות המחלקה שגם נוכחות בממשק חייבות להיות מוגדרות כציבוריות. הדבר נובע מכך שכל שיטה בממשק היא בעצם ציבורית.

במחלקות ציבוריות צריכים להיות שדות ציבוריים לעיתים רחוקות בלבד, אם בכלל (בניגוד לשיטות ציבוריות). אם שדה אינו סופי או ייחוס סופי לאובייקט שניתן לשינוי, אתה מוותר על היכולת להגביל את הערכים שיכולים להיות מאוחסנים בשדה, על ידי הגדרתו כציבורי. אתה גם מוותר על היכולת לנקוט בפעולה כלשהי כאשר השדה עובר שינוי. תוצאה פשוטה היא שמחלקות עם שדות ציבוריים שאינם סופיים אינן בטוחי-מטלות (thread safe). אפילו אם שדה הוא סופי ולא מתייחס לאובייקט שניתן לשינוי, על ידי הגדרת השדה כציבורי אתה מוותר על הגמישות לעבור לייצוג נתונים פנימי חדש, שבו השדה אינו קיים.

יש חריג אחד לכלל האומר שלמחלקות ציבוריות לא צריכים להיות שדות ציבוריים: מחלקות מורשות לחשוף קבועים באמצעות שדות סטטיים ציבוריים סופיים. לפי מוסכמה, שמות שדות אלה מורכבים מאותיות רישיות עם מילים שמופרדות במקף תחתון (נושא 38). שדות אלה **חייבים** להכיל ערכים בסיסיים או ייחוסים לאובייקטים **שאינם ניתנים לשינוי** (נושא 13). לשדה סופי שמכיל ייחוס לאובייקט שניתן לשינוי יש את כל החסרונות של שדה לא סופי.

למרות שהייחוס אינו יכול להשתנות, האובייקט המיוחס כן יכול - והתוצאות לתוכנית הן חמורות.

שים לב שמערך שאינו בעל אורך אפס הוא תמיד ניתן לשינוי, ולכן **זו כמעט תמיד טעות להחזיק בשדה של מערך סופי סטטי ציבורי**. אם למחלקה יש שדה כזה, לקוחות יהיו מסוגלים לשנות את תוכן המערך. זהו מקור נפוץ לפרצות באבטחה:

```
// Potential secrurity hole !
public static final Type[] VALUES = { ... };
```

המערך הציבורי צריך להיות מוחלף במערך פרטי וברשימה ציבורית שאינה ניתנת לשינוי:

```
private static final Type[] PRIVATE_VALUES =  { ... };

public static final List VALUES =
  Collections.unmodifiableList(Array.asList(PRIVATE_VALUES));
```

לחילופין, אם אתה דורש הגנה על הטיפוסים בזמן ההידור (רשימה מכילה ייחוסים מהטיפוס הבסיסי של Object), ומסכים לסבול ירידה בביצועים, תוכל להחליף את שדה המערך הציבורי בשיטה ציבורית המחזירה עותק של מערך פרטי:

```
private static final Type[] PRIVATE_VALUES = { ... };

public static final Type[] values()  {
  return (Type[]) PRIVATE_VALUES.clone();
}
```

לסיכום, תמיד עליך להפחית את מידת הנגישות ככל שתוכל. לאחר שתכנת בזהירות API ציבורי מינימלי, אתה צריך למנוע מכל מחלקה, ממשק או איבר מקריים מלהפוך לחלק מה-API. למעט החריג של שדות סופיים סטטיים ציבוריים, למחלקות ציבוריות לא צריך להיות כל שדה ציבורי. דאג לכך שאובייקטים המיוחסים על ידי שדות סופיים סטטיים ציבוריים אינם יהיו ניתנים לשינוי.

נושא 13: העדף להשתמש במחלקות שאינן ניתנות לשינוי (immulable class) כאשר הדבר אפשרי

מחלקה ש**אינה ניתנת לשינוי** (immulable class) היא מחלקה שלא ניתן לשנות את המופעים שלה. המידע הכלול בכל מופע מסופק כאשר המופע נוצר, ונשאר קבוע לכל אורך חיי האובייקט. ספריות הפלטפורמה של Java מכילות מחלקות רבות שאינן ניתנות לשינוי וביניהן String, מחלקות העטיפה הבסיסיות, BigInteger ו-BigDecimal. יש לכך הרבה סיבות טובות: קל יותר לתכנן, לממש ולהשתמש במחלקות שאינן ניתנות לשינוי מאשר במחלקות הניתנות לשינוי (mutable classes). הן חשופות פחות לטעויות וגם בטוחות יותר.

כדי ליצור מחלקה שאינה ניתנת לשינוי, עקוב אחר חמשת הכללים הבאים:

1. **אל תספק שיטות המשנות את האובייקט** (ידועות כ-**mutators**).
2. **הבטח שאף שיטה לא תידרס**. הדבר מונע מתת-מחלקה לא זהירה או זדונית מלהפר את ההתנהגות שאינה ניתנת לשינוי של מחלקה. מניעת דריסת שיטה נעשית בדרך כלל על ידי הפיכת המחלקה לסופית, אבל יש חלופות שנדון בהן מאוחר יותר.
3. **דאג לכך שכל השדות יהיו סופיים**. הדבר מבהיר את כוונותיך היטב, באופן אשר נאכף על ידי המערכת. כמו כן, ייתכן שיהיה חיוני להבטיח התנהגות נכונה אם ייחוס למופע חדש שנוצר, מועבר ממטלה אחת לחברתה ללא סינכרון - תלוי בתוצאות של מאמצים מתמשכים לעבד את **מודל הזיכרון** (memory model) [Pugh01a].
4. **דאג לכך שכל השדות יהיו פרטיים**. הדבר מונע מלקוחות לשנות באופן ישיר את השדות. למחלקות שאינן ניתנות לשינוי יכולים להיות, באופן טכני, שדות סופיים ציבוריים המכילים ערכים בסיסיים (פרימיטיביים) או ייחוסים לאובייקטים שאינם ניתנים לשינוי. למרות זאת, אין זה מומלץ לנהוג כך, כי הדבר מונע ממך לבצע שינויים בייצוג הפנימי בגירסה מאוחרת יותר שעשויה להידרש (נושא 12).
5. **דאג לחסום גישה לכל רכיב הניתן לשינוי**. אם למחלקה שלך יש שדות כלשהם המתייחסים לאובייקטים הניתנים לשינוי, דאג לכך שלקוחות של המחלקה לא יוכלו להשיג ייחוסים לאובייקטים אלה. צור **עותקים הגנתיים** (defensive copies) (נושא 24) בבנאים, באמצעי גישה ובשיטות readObject.

רבות מהמחלקות לדוגמה שהוצגו בנושאים קודמים הן מחלקות הניתנות לשינוי. מחלקה אחת כזאת היא PhoneNumber שבנושא 8, שיש לה שיטות גישה (accessors) לכל תכונה, אבל אין לה mutator כלשהו. להלן דוגמה מעט יותר מורכבת:

```
public final class Complex {
    private final float re;
    private final float im;

    public Complex(float re, float im) {
        this.re = re;
        this.im = im;
    }

    // Accessors with no corresponding mutators
    public float realPart()      { return re; }
    public float imaginaryPart()  { return im; }

    public Complex add(Complex c)  {
        return new Complex(re + c.re, im + c.im);
    }

    public Complex subtract(Complex c)  {
        return new Complex(re - c.re, im - c.im);
    }

    public Complex multiply(Complex c)  {
        return new Complex(re*c.re - im*c.im, re*c.im + im*c.re);
    }

    public Complex divide(Complex c)  {
        float tmp = c.re*c.re + c.im*c.im;
        return new Complex((re*c.re + im*c.im)/tmp, (im*c.re - re*c.im)/tmp);
    }

    public boolean equals(Object o)  {
      if (o == this)
          return true;
      if (!(o instanceof Complex))
          return false;
      Complex c = (Complex)o;
      return (Float.floatToIntBits(re) ==      // See page 33 to
              Float.floatToIntBits(c.re)) &&   // find out why
             (Float.floatToIntBits(im) ==      // floatToIntBits
              Float.floatToIntBits(c.im));     // is used.
    }
```

```
    public int hashCode()  {
        int result = 17 + Float.floatToIntBits(re);
        result = 37*result + Float.floatToIntBits(im);
        return result; trivial
    }

    public String toString()  {
        return "(" + re + " + " + im + "i)";
    }
}
```

מחלקה זו מייצגת **מספר מורכב** (complex number, **מספר מרוכב**, שיש לו חלק ממשי וחלק מדומה). בנוסף לשיטות Object הסטנדרטיות, המחלקה מספקת אמצעי גישה לשני חלקי המספר ומספקת ארבע פעולות אריתמטיות בסיסיות: חיבור, חיסור, כפל וחילוק. שים לב איך הפעולות האריתמטיות יוצרות ומחזירות מופע Complex חדש, ואינן משנות את המופע הזה. תבנית זו משמשת במרבית המחלקות הלא-טריוויאליות שאינן ניתנות לשינוי. גישה זו ידועה כגישה **הפונקציונלית**, מכיון שהשיטות מחזירות את התוצאה של הפעלת פונקציה על האופרנד שלהן בלי לשנות אותו. בניגוד לגישה **הפרוצדורלית** המקובלת יותר, שבה השיטות מפעילות פרוצדורה על האופרנד שלהן וגורמות לו להשתנות.

הגישה הפונקציונלית עלולה להיראות לא טבעית אם אינך מכיר אותה, אבל היא עשויה למנוע שינוי, דבר שיש לו יתרונות רבים. **אובייקטים שאינם ניתנים לשינוי הינם פשוטים יותר**. אובייקט שאינו ניתן לשינוי (immutable) יכול להיות במצב אחד בלבד - המצב שבו הוא נוצר. אם אתה מבטיח שכל הבנאים בונים מחלקות שאינן ניתנות לשינוי, מובטח לך שמחלקות אלו תישארנה אינן ניתנות לשינוי כל הזמן, ללא שום מאמץ נוסף מצידך או מצד המתכנת המשתמש במחלקות. לאובייקטים הניתנים לשינוי, לעומת זאת, יכולים להיות מצבים מורכבים ושרירותיים. אם התיעוד אינו מספק תיאור מדויק של שינויי המצב שמתבצעים על ידי שיטות שמשנות אובייקטים אחרים, השימוש במחלקה הניתנת לשינוי באופן מהימן יהיה קשה או אף בלתי אפשרי.

אובייקטים שאינם ניתנים לשינוי הם בטוחי-מטלה מטבעם. הם אינם דורשים כל סינכרון. אובייקטים אלה אינם יכולים להיפגע על ידי מטלות רבות שניגשות אליהם בו-זמנית, ולכן הם נקראים **בטוחי-מטלה** (thread-safe). במידה רבה זו הגישה הקלה ביותר להשגת בטיחות מטלות. למעשה, אף מטלה לעולם אינה יכולה להשפיע על מטלה אחרת כאשר האובייקט אינו ניתן לשינוי. לכן, **ניתן לחלוק אובייקטים שאינם ניתנים לשינוי באופן חופשי** (כלומר, משתמשים רבים יכולים להשתמש בהם באותו זמן). מחלקות שאינן ניתנות לשינוי צריכות לנצל זאת על ידי עידוד לקוחות להשתמש מחדש במופעים קיימים כל זמן שאפשר. דרך קלה אחת לעשות זאת היא לספק קבועים סופיים סטטיים ציבוריים לערכים שנעשה בהם שימוש לעיתים קרובות.

לדוגמה, מחלקת Complex יכולה לספק את הקבועים הבאים:

```
public static final Complex ZERO  =  new Complex(0, 0)
public static final Complex ONE   =  new Complex(1, 0)
public static final Complex I     =  new Complex(0, 1)
```

אפשר להמשיך בגישה זו ולומר שאובייקט שאינו ניתן לשינוי יכול לספק יצרנים סטטיים (static factories) אשר שומרים במאגר מופעים הנדרשים לעיתים תכופות, ולהימנע מיצירת מופעים חדשים בכל פעם שמבוקש מופע שקיים כבר. למחלקות BigInteger ו-Boolean יש יצרנים סטטיים כאלה. שימוש ביצרנים סטטיים גורם ללקוחות לשתף מופעים שקיימים ולא ליצור מופעים חדשים וכך - לחסוך בצריכת משאבי זיכרון ועלויות איסוף אשפה.

כתוצאה מכך שאפשר לשתף אובייקטים שאינם ניתנים לשינוי באופן חופשי, אינך צריך לעולם ליצור **עותקי הגנה** (defensive copies) (נושא 24). למעשה, אינך צריך לעולם ליצור עותקים כלשהם, כיון שהעותקים תמיד יהיו שווים למקור. לכן, אינך צריך וגם אינך חייב לספק את השיטה clone או **בנאי העתקה** (נושא 10) עבור מחלקה שאינה ניתנת לשינוי. הדבר לא היה מובן כל כך בימיה המוקדמים של פלטפורמת Java. לכן, למחלקה String יש בנאי העתקה, אבל צריך להשתמש בו באופן נדיר, אם בכלל (נושא 4).

לא רק שאתה יכול לשתף אובייקטים שאינם ניתנים לשינוי, אלא שאתה גם יכול לשתף את התכונות הפנימיות שלהם. לדוגמה, המחלקה BigInteger משתמשת בסימן סדר גודל (sign-magnitude) באופן פנימי. הסימן מיוצג על ידי int והגודל מיוצג על ידי מערך של ערכי int. השיטה negate מייצרת BigInteger חדש באותו סדר גודל ובסימן הפוך. היא אינה צריכה להעתיק את המערך. ה-BigInteger החדש שנוצר מצביע לאותו מערך פנימי כמו המקורי.

אובייקטים שאינם ניתנים לשינוי מהווים אבני בניין נהדרים עבור יצירת אובייקטים אחרים, בין אם הם אובייקטים הניתנים לשינוי או אובייקטים שאינם ניתנים לשינוי. הרבה יותר קל לשמור את הקבועים של אובייקט מורכב, אם אתה יודע שהאובייקטים המרכיבים אותו לא ישתנו. מקרה מיוחד של עיקרון זה הוא שאובייקטים שאינם ניתנים לשינוי משמשים כמפתחות מפה (map keys) נהדרים. אינך צריך לדאוג שמא ערכיהם ישתנו לאחר שהם במפה או בקבוצה (set) - דבר אשר עלול להרוס את קבועי המפה או הקבוצה.

החיסרון האמיתי היחיד של מחלקות שאינן ניתנות לשינוי הוא בכך שהן מחייבות אובייקט נפרד לכל ערך נפרד. יצירת אובייקטים אלה יכולה להיות יקרה, במיוחד אם הם גדולים. לדוגמה, נניח שיש BigInteger בן מיליון סיביות וברצונך לשנות את מצבה (complement) של הסיבית האחרונה שלו:

```
BigInteger moby = ...;
moby = moby.flipBit(0);
```

השיטה flipBit יוצרת מופע BigInteger חדש, גם הוא בן מיליון סיביות, השונה מהמקור בסיבית אחת בלבד. הפעולה דורשת זמן ומקום באופן יחסי לגודל של BigInteger. השווה זאת ל-java.util.BitSet. כמו BigInteger, גם BigSet מייצג רצף סיביות באורך שרירותי, אבל שלא כמו BigInteger, כאן BitSet הוא משתנה. המחלקה BitSet מספקת שיטה המאפשרת לשנות את המצב של סיבית בודדת במופע בן מיליון סיביות בזמן קבוע.

בעיית הביצועים גדלה אם אתה מבצע פעולה רבת-שלבים, המחוללת אובייקט חדש בכל שלב, עד שלבסוף נפטרים מכל האובייקטים חוץ מהתוצאה הסופית. יש שתי גישות לטפל בבעיה זו. הראשונה היא לנחש איזה פעולות רבות-שלבים תידרשנה באופן הנפוץ ביותר ולספק אותן כאובייקטים בסיסיים. אם פעולה רבת-שלבים מסופקת כערך בסיסי, המחלקה שאינה ניתנת לשינוי אינה צריכה ליצור אובייקט נפרד בכל שלב. באופן פנימי, המחלקה שאינה ניתנת לשינוי יכולה להיות "פיקחית". לדוגמה, ל-BigInteger יש "מחלקה שותפה" ניתנת לשינוי ופרטית-בחבילה, המשמשת להאצת פעולות רבות-שלבים כמו העלאה בחזקה מודולרית. קשה הרבה יותר להשתמש במחלקה השותפה הניתנת לשינוי בגלל כל הסיבות שתוארו לפני כן, אבל למרבה המזל אינך צריך להשתמש בה. מי שכתב את BigInteger עשה את כל העבודה הקשה למענך.

גישה זו פועלת כראוי אם אתה יכול לחזות במדויק איזה פעולות מרובות-שלבים מורכבות ירצו הלקוחות לבצע במחלקה שאינה ניתנת לשינוי שלך. אם אינך יכול לחזות איזה פעולות, אז ההימור הטוב ביותר יהיה לספק מחלקה שותפה ניתנת לשינוי **ציבורית**. הדוגמה העיקרית לגישה זו בספריות הפלטפורמה של Java היא המחלקה String, שהשותפה הניתנת לשינוי שלה היא StringBuffer. ניתן אף לטעון ש-BitSet ממלאת את תפקיד השותפה שאינה ניתנת לשינוי ל-BigInteger, בנסיבות מסוימות.

כעת, לאחר שאתה יודע כיצד ליצור מחלקה שאינה ניתנת לשינוי ומבין את היתרונות והחסרונות שלה, נדון בכמה חלופות תכנון. זכור שכדי להבטיח שהיא תהיה אינה ניתנת לשינוי, אסור שהמחלקה תאפשר לדרוס את השיטות שלה. בנוסף להגדרת המחלקה כמחלקה סופית, יש שתי דרכים נוספות להבטיח זאת. דרך אחת היא להגדיר את כל השיטות של המחלקה כשיטות סופיות בלי להגדיר את המחלקה כסופית. היתרון היחיד של גישה זו הוא בכך שהיא מאפשרת למתכנתים להרחיב את המחלקה בהורשה, על ידי הוספת שיטות חדשות שנבנות בנוסף לשיטות הישנות. באותה מידה של יעילות אפשר לספק את השיטות החדשות כשיטות סטטיות במחלקת עזר נפרדת ללא יכולת ליצור מופע נוסף (נושא 3), אך גישה זו אינה מומלצת.

חלופה שנייה להפיכת מחלקה שאינה ניתנת לשינוי לסופית, היא להפוך את כל הבנאים שלה לפרטיים או לפרטיים-בחבילה, ולהוסיף **יצרנים סטטיים** (static factories) במקום הבנאים הציבוריים (נושא 1).

כדי להבהיר זאת, כך היה נראה הקוד של Complex אם גישה זו היתה מיושמת:

```
// Immutable class with static factories instead of constructors
public class Complex {
   private final float re;
   private final float im;

   private Complex(float re, float im)  {
      this.re = re;
      this.im = im;
   }

   public static Complex valueOf(float re, float im)  {
      return new Complex(re, im);
   }
   ...  // Reminder unchanged
}
```

למרות שגישה זו אינה מיושמת בתכיפות, לעיתים קרובות היא החלופה הטובה ביותר מבין שלוש הגישות. היא הגמישה ביותר, מכיון שהיא מאפשרת שימוש במחלקות רבות שהן פרטיות-בחבילה (package-private). עבור לקוחות שנמצאים מחוץ לחבילת התוכנה שלה, המחלקה שאינה ניתנת לשינוי היא סופית באופן יעיל, מכיון שאין זה אפשרי להרחיב מחלקה שבאה מחבילת תוכנה אחרת כשאין לה בנאי ציבורי או מוגן. פרט למתן הגמישות של מחלקות מרובות, גישה זו מאפשרת לכוון את הביצועים של המחלקה בגרסאות עוקבות על ידי שיפור היכולות של היצרנים הסטטיים לשמור אובייקטים במאגר.

ליצרנים סטטיים יש יתרונות רבים אחרים על פני בנאים, כפי שצוין כבר בנושא 1. לדוגמה, נניח שאתה רוצה לספק אמצעים ליצירת מספר מורכב (complex number), המבוסס על הקואורדינטות הקוטביות (Polarity) שלו (בעזרת רדיוס וזווית) ולא על פי הייצוג הקרטזי שלו (בעזרת ערך ממשי וערך מדומה). יהיה זה מאוד לא נכון להשתמש בבנאים, כי לבנאי הטבעי תהיה אותה החתימה שכבר השתמשנו בה:

```
Complex(float, float).
```

עם יצרנים סטטיים הדבר פשוט וקל. הוסף יצרן סטטי שני בעל שם המבהיר היטב את תפקידו:

```
public static Complex valueOfPolar(float r, float theta)  {
      return new Complex((float) (r * Math.cos(theta)),
                         (float) (r * Math.sin(theta)));
   }
```

כאשר BigInteger ו-BigDecimal נכתבו, לא היה ברור עדיין שמחלקות שאינן ניתנות לשינוי צריכות להיות סופיות לשם היעילות. לכן, ניתן לדרוס את כל השיטות שלהן.

לרוע המזל, אי אפשר היה לתקן זאת לאחר מכן תוך כדי שמירה על תאימות לאחור. אם אתה כותב מחלקה שהבטיחות שלה תלויה בכך שהארגומנט BigInteger או BigDecimal לא יהיו ניתנים לשינוי על ידי לקוח בלתי אמין, אתה חייב לבדוק שהארגומנט הוא BigDecimal או BigInteger "אמיתי", ולא מופע של תת-מחלקה שאי אפשר לבטוח בה. אם הוא מופע של תת-מחלקה כזאת, אתה חייב ליצור עותק הגנה שלו בהנחה שהוא יכול להשתנות (נושא 24):

```
public void foo(BigInteger b )  {
  if (b.getClass() != BigInteger.class)
    b = new BigInteger(b.toByteArray());
  ...
}
```

רשימת הכללים למחלקות שאינן ניתנות לשינוי שבתחילת נושא זה אומרת שאף שיטה לא תשנה את האובייקט ושכל השדות חייבים להיות סופיים. למעשה, כללים אלה יותר קשוחים ממה שהכרחי, ואפשר אפילו להתעלם מהם כדי לשפר את הביצועים. לאמיתו של דבר, אף שיטה לא צריכה לגרום לשינוי **חיצוני נראה** (externally visible) במצב האובייקט. עם זאת, למחלקות רבות שאינן ניתנות לשינוי יש שדה לא סופי מיותר אחד או יותר, שבו הן שומרות במאגר את התוצאות של חישובים "יקרים" שנעשים בפעם הראשונה. אם יידרש אותו החישוב שוב, ישוחזר הערך שנשמר במאגר והדבר יחסוך את העלות של חישוב מחדש. טריק זה פועל היטב, כי האובייקט אינו ניתן לשינוי. העובדה שהוא אינו ניתן לשינוי מבטיחה שהחישוב יניב את אותה התוצאה אם הוא מבוצע שוב.

לדוגמה, השיטה hashCode של PhoneNumber (נושא 8) מחשבת את קוד הערבול בפעם הראשונה שבה היא נקראת, ושומרת אותו במאגר למקרה שבו יידרש שוב. בטכניקה זו, שהיא דוגמה קלאסית ל**אתחול עצל** (נושא 48), נעשה שימוש גם על ידי המחלקה String. אין צורך בסינכרוניזציה, כיון שאין זו בעיה אם ערך הערבול מחושב מחדש פעם אחת או פעמיים.

להלן הקוד הכללי להחזרת פונקציה שנשמרה במאגר ועברה אתחול עצל של אובייקט שאינו ניתן לשינוי:

```
// Cached, lazily initialized function of an immutable object
private volatile Foo cachedFooVal = UNLIKELY_FOO_VALUE;

public Foo foo()  {
  int result = cachedFooVal;
  if (result == UNLIKELY_FOO_VALUE)
    result = cachedFooVal = fooValue();
  return result;
}

// Private helper function to calculate our foo value
private Foo fooVal()  { ... }
```

יש להוסיף אזהרה אחת בנוגע ליכולת ההעברה טורית בזרם (serializability). אם אתה בוחר באפשרות שהמחלקה שלך שאינה ניתנת לשינוי תממש Serializable, והיא מכילה שדה אחד או יותר המתייחסים לאובייקטים משתנים, אתה חייב לספק את השיטה readObject או את readResolve באופן מפורש. ללא דריסה, שיטת ברירת המחדל של readObject תאפשר לפורץ ליצור מופע ניתן לשינוי של המחלקה האחרת שלך שאינה ניתנת לשינוי. תחום זה נדון בפירוט בנושא 56.

לסיכום, דכא את הדחף לכתוב שיטת set לכל שיטת get. **מחלקות צריכות להיות אינן ניתנות לשינוי, אלא אם כן יש סיבה טובה מאוד לעשות אותן ניתנות לשינוי.**
למחלקות שאינן ניתנות לשינוי יש יתרונות רבים וחסרונן היחיד הוא הפוטנציאל לבעיות ביצועים אפשריות בנסיבות מסוימות. אתה צריך תמיד לעשות אובייקטי ערך קטנים, כמו PhoneNumber ו-Complex, שאינם ניתנים לשינוי (יש כמה מחלקות בספריות הפלטפורמה של Java, כמו java.util.Date ו-java.awt.Point אשר היו צריכות להיות אינן ניתנות לשינוי, אבל הן ניתנות לשינוי). אתה צריך לשקול ברצינות יצירת אובייקטי ערך גדולים יותר שאינם ניתנים לשינוי, כמו String ו-BigInteger. עליך לספק מחלקה ציבורית ניתנת לשינוי שתהיה בת-זוג למחלקה שלך שאינה ניתנת לשינוי, **רק** לאחר שנוכחת שהדבר הכרחי להשגת ביצועים יותר טובים (נושא 37).

יש כמה מחלקות שלא מעשי ליצור אותן כבלתי ניתנות לשינוי, ובכלל זה "מחלקות תהליך", כמו Thread ו-TimerTask. **אם אינך יכול ליצור מחלקה שאינה ניתנת לשינוי, עליך בכל זאת להגביל את אי-ההשתנות שלה עד כמה שאפשר**. הפחתת מספר המצבים שבהם אובייקט יכול להתקיים, הופכת את ההתייחסות לאובייקט לקלה יותר ומפחיתה את הסבירות לטעויות. לכן, **בנאים צריכים ליצור אובייקטים מאותחלים באופן מלא עם כל הקבועים שלהם**, ואינם צריכים להעביר מופעים בנויים חלקית אל שיטות אחרות. אל תספק שיטת אתחול ציבורית נפרדת מהבנאי, אלא אם כן יש באמת סיבה מאוד טובה לכך. באופן דומה, אל תספק שיטת "אתחול מחדש" המאפשרת להשתמש באובייקט מחדש כאילו נבנה במצב אתחול נפרד. בדרך כלל, שיטת אתחול מחדש משפרת את הביצועים במעט אם בכלל, במחיר של מורכבות גדולה יותר.

המחלקה Timer מדגימה עקרונות אלה. היא ניתנת לשינוי, אבל טווח אפשרויות השינוי שלה נשמר בכוונה קטן. אתה יוצר מופע, מתזמן אותו לביצוע ומבטל אותו על פי בחירה. לאחר שמשימת timer פעלה עד לסיום או שבוטלה, אינך יכול לתזמן אותה מחדש.

יש להוסיף הערה בקשר למחלקה Complex שבנושא זה. דוגמה זו הובאה רק כדי להדגים מצב של מחלקה שאינה ניתנת לשינוי. אין זה מימוש מספרים מורכבים נבון. הוא משתמש בנוסחאות הסטנדרטיות לכפל וחילוק מורכבים, שאינן מעוגלות באופן נכון ומספקות סמנטיקה חלשה ל-NaNs מורכבים ואינסופיים [Kahan91, Smith62, Thomas94].

נושא 14: העדף הרכבה (composition) על פני הורשה

הורשה (inheritance) היא כלי רב עוצמה לשימוש מחדש בקוד שכבר נכתב, אבל לא תמיד היא האמצעי הטוב ביותר להשגת מטרה זו. שימוש לא מתאים בהורשה מביא ליצירת תוכנה בלתי יציבה. בחבילת תוכנה שבה מימושי תת-המחלקות ומחלקות-העל הם תחת השליטה של אותם המתכנתים, השימוש בהורשה בטוח. גם בהרחבת מחלקות שתוכננו ותועדו במיוחד להרחבה, השימוש בהורשה בטוח (נושא 15). עם זאת, הורשה ממחלקות יציבות רגילות אל מעבר לגבולות חבילת התוכנה, הינה מסוכנת. תזכורת: ספר זה משתמש במילה "הורשה" במובן של "הורשת מימוש" (implementation inheritance, כאשר מחלקה אחת מרחיבה מחלקה אחרת). הבעיות שנדונות בנושא זה אינן מתייחסות ל**הורשת ממשק** (interface inheritance, כאשר מחלקה מממשת ממשק או כאשר ממשק אחד מרחיב ממשק אחר).

שלא כמו בקריאה לשיטות, הורשה מפירה את עיקרון הכימוס (encapsulation) [Snyder86]. במילים אחרות, תת-מחלקה תלויה בפרטי המימוש של מחלקת-העל שלה עבור תפקודה התקין. מימוש מחלקת-העל עשוי להשתנות מגירסה לגירסה ואם הוא ניתן לשינוי, תת-המחלקה עשויה להיפגע, למרות שהקוד שלה כלל לא השתנה. כתוצאה מכך, תת-מחלקה חייבת להתפתח יחד עם מחלקת-העל שלה, אלא אם כן מי שפיתח את מחלקת-העל תכנן ותיעד אותה במפורש להרחבה.

כדי להבהיר זאת, הבה נניח שיש לנו תוכנית המשתמשת ב-HashSet. כדי לכוון את ביצועי התוכנית שלנו, אנו צריכים לחקור את HashSet ולברר כמה אלמנטים נוספו מאז שנוצר (אל תתייחס לגודלו הנוכחי, שהולך ויורד כאשר מסירים אלמנטים). כדי לספק יכולת זו, נכתוב מחלקה שמרחיבה את HashSet ומספקת לו את תכונת הספירה. המחלקה HashSet מכילה שתי שיטות שיכולות להוסיף אלמנטים, add ו-addAll, וכך אנו דורסים את שתיהן:

```
// Broken - Inappropriate use of inheritance!
public class InstrumentedHashSet extends HashSet  {
   // The number of attempted element insertions
   private int addCount = 0;

   public InstrumentedHashSet()  {
   }

   public InstrumentedHashSet(Collection c)  {
      super(c);
   }
```

```
    public InstrumentedHashSet(int initCap, float loadFactor) {
        super(initCap, loadFactor);
    }

    public boolean add(Object o) {
        addCount++;
        return super.add(o);
    }

    public boolean addAll(Collection c) {
        addCount += c.size();
        return super.addAll(c);
    }

    public int getAddCount() {
        return addCount;
    }
}
```

מחלקה זו נראית הגיונית, אבל היא אינה יכולה לפעול. נניח שאנו יוצרים מופע ומוסיפים שלושה אלמנטים תוך שימוש בשיטה addAll:

```
InstrumentedHashSet s = new InstrumentedHashSet();
s.addAll(Arrays.asList(new String[]  {"Snap", "Creckle", "Pop"}));
```

בנקודה זו אנו מצפים מהשיטה getAddCount להחזיר שלוש, אבל היא מחזירה שש. מדוע הקוד אינו פועל כשורה? באופן פנימי, השיטה addAll של HashSet ממומשת בעזרת שימוש בשיטה add שלה, למרות ש-HashSet בהיגיון רב, אינה מתעדת את פרטי המימוש הזה. השיטה addAll שב-InstrumentedHashSet הוסיפה שלוש ל-addCount ואז קראה למימוש addAll של HashSet תוך שימוש ב-super.addAll. זאת בתורה קוראת לשיטה add, כפי שנדרסה ב-InstrumentedHashSet, פעם אחת עבור כל אלמנט. כל אחת משלוש קריאות אלו הוסיפה אחת או יותר ל-addCount, עד להוספה של שש. כל אלמנט שהתווסף על ידי השיטה addAll נספר פעמיים.

יכולנו "לתקן" את תת-המחלקה על ידי סילוק הדריסה של השיטה addAll. למרות שהמחלקה שתתקבל עשויה לפעול, לשם תפקודה התקין היא תהיה תלויה בכך שהשיטה addAll של HashSet תמומש על ידי שימוש בשיטה add שלה. "שימוש עצמי" זה הוא פרט מימוש שאינו מובטח שיתקיים בכל המימושים של פלטפורמת Java והוא עלול להשתנות מגירסה לגירסה. לכן, המחלקה InstrumentedHashSet שתתקבל לא תהיה יציבה.

טוב יותר לדרוס את השיטה addAll כך שתעבור שוב ושוב (באיטרציה) על פני האוסף שמצוין, תוך קריאה לשיטה add פעם אחת עבור כל אלמנט.

הדבר יבטיח את התוצאה הנכונה, בין אם השיטה addAll של HashSet מומשה או לא מומשה לפני השיטה add שלה, כי מימוש addAll של HashSet לא ייקרא עוד. עם זאת, טכניקה זו אינה פותרת את כל בעיותינו. היא מסתכמת במימוש מחדש של שיטות מחלקת-על אשר ייתכן שיסתיימו או לא יסתיימו בשימוש עצמי - דבר שהוא קשה, צורך זמן ונוטה לתקלות. בנוסף, אין הדבר אפשרי תמיד, כיון שאי אפשר לממש שיטות מסוימות ללא גישה לשדות פרטיים אשר אינם נגישים לתת-המחלקה.

פועל יוצא של חוסר היציבות בתת-המחלקות הוא שמחלקת-העל שלהן יכולה לקלוט שיטות חדשות בגרסאות הבאות. נניח שתוכנית תלויה לשם האבטחה שלה בעובדה שכל האלמנטים שהוכנסו לתוך אוסף ממלאים אחר כללים מסוימים. אפשר להבטיח דבר זה על ידי יצירת תת-מחלקות של אוסף ודריסת כל שיטה שיכולה להוסיף אלמנט, כדי להבטיח שהכללים נשמרים לפני הוספת האלמנט. הדבר יפעל כראוי כל עוד שיטה חדשה שיכולה להוסיף אלמנט לא התווספה למחלקת-העל בגירסה החדשה. לאחר שהשיטה התווספה, אפשר להוסיף אלמנט "לא חוקי" למופע של תת-המחלקה על ידי קריאה לשיטה החדשה שלא נדרסה בתת-המחלקה. אין זו בעיה תיאורטית בלבד. כמה פרצות אבטחה מהסוג הזה תוקנו כאשר Hashtable ו-Vector שופרו כך שישתתפו בסביבת העבודה של האוספים (Collections Framework).

שתי הבעיות שנזכרו לעיל נובעות מדריסת שיטות. אתה עשוי לחשוב שיותר בטוח להרחיב מחלקה אם אתה רק מוסיף שיטות חדשות ונמנע מדריסת שיטות קיימות. למרות שסוג זה של הרחבה בטוח יותר, אין הוא נטול סיכונים. אם מחלקת-העל מקבלת שיטה חדשה בגירסה עוקבת ולרוע מזלך נתת לתת-המחלקה שיטה עם אותה החתימה אבל עם טיפוס שונה של ערך מוחזר, תת-המחלקה שלך לא תעבור את ההידור בשלום [JLS, 8.4.6.3]. אם נתת לתת-המחלקה שיטה עם אותה החתימה בדיוק כמו של השיטה של מחלקת-העל, אז תגרום לדריסה שלה וכתוצאה - אתה חשוף לשתי הבעיות שתוארו לעיל. יתר על כן, ספק אם השיטה שלך תמלא את ההסכם של שיטת מחלקת-העל החדשה, כיון שההסכם לא נכתב עדיין כאשר כתבת את השיטה של תת-המחלקה.

למרבה המזל, יש דרך להימנע מכל הבעיות שתוארו לעיל. במקום הרחבת מחלקה קיימת, תן למחלקה החדשה שלך שדה פרטי המתייחס למופע של המחלקה הקיימת. עיצוב זה נקרא **הרכבה** (composition), כי המחלקה הקיימת נעשית מרכיב של המחלקה החדשה. כל שיטת מופע במחלקה החדשה קוראת לשיטה המתאימה של המופע המוכל (שמטיפוס המחלקה המקורית) ומחזירה את התוצאות. הדבר ידוע בשם **העברה** (forwarding), והשיטות שבמחלקה החדשה ידועות כ**שיטות העברה** (forwarding methods). המחלקה שתתקבל תהיה יציבה מאוד, ללא תלות בפרטי המימוש של המחלקה הקיימת.

אפילו הוספת שיטות חדשות למחלקה הקיימת לא תשפיע על המחלקה החדשה. כדי להבהיר זאת, להלן מחלקה חליפית ל-InstrumentedHashSet אשר משתמשת בגישת הרכבה/העברה:

```
// Wrapper class - uses composition in place of inheritance
public class InstrumentedSet implements Set {
    private final Set s;
    private int addCount = 0;

    public InstrumentedSet(Set s) {
        this.s = s;
    }

    public boolean add(Object o) {
        addCount++;
        return s.add(o);
    }

    public boolean addAll(Collection c) {
        addCount += c.size();
        return s.addAll(c);
    }

    public int getAddCount() {
        return addCount;
    }

    // Forwarding methods
    public void clear()                         { s.clear();            }
    public boolean contains(Object o)           { return s.contains(o); }
    public boolean isEmpty()                    { return s.isEmpty();   }
    public int size()                           { return s.size();      }
    public Iterator iterator()                  { return s.iterator();  }
    public boolean remove(Object o)             { return s.remove(o);   }
    public boolean containsAll(Collection c)
                                        { return s.containsAll(c); }
    public boolean removeAll(Collection c)
                                        { return s.removeAll(c);   }
    public boolean retainAll(Collection c)
                                        { return s.retainAll(c);   }
    public Object[] toArray()                   { return s.toArray();  }
    public Object[] toArray(Object[] a)         { return s.toArray(a); }
    public boolean equals(Object o)             { return s.equals(o);  }
    public int hashCode()                       { return s.hashCode(); }
    public String toString()                    { return s.toString(); }
}
```

התכנון של המחלקה InstrumentedSet מתאפשר על ידי הקיום של ממשק Set, אשר לוכד את התפקודיות של המחלקה HashSet. מלבד היותו יציב, התכנון הזה גמיש ביותר. המחלקה InstrumentedSet מממשת את הממשק Set ויש לה בנאי יחיד שטיפוס הארגומנט שלו הוא גם כן Set. בעיקרה, המחלקה משנה Set אחד לאחר, תוך הוספת תפקודיות של הכלים. שלא כמו הגישה המבוססת על הורשה, הפועלת רק עבור מחלקה מסוימת אחת, ודורשת בנאי נפרד עבור כל בנאי נתמך שבמחלקת-העל, מחלקת העטיפה יכולה לשמש להרכבת כל מימוש של Set ותפעל בצירוף עם כל בנאי שהיה קיים קודם.

הנה דוגמה:

```
Set s1 = new InstrumentedSet(new TreeSet(list));
Set s2 = new InstrumentedSet(new HashSet(capacity, loadFactor));
```

המחלקה InstrumentedSet יכולה אפילו לשמש להרכבה זמנית של מופע Set שנעשה בו כבר שימוש ללא הרכבה:

```
static void f(Set s) {
    InstrumentedSet sInst = new InstrumentedSet(s);
    ... // Within this method use sInst instead of s
}
```

המחלקה InstrumentedSet ידועה כמחלקת **עטיפה** (wrapper), מכיון שכל מופע InstrumentedSet עוטף מופע Set אחר. הדבר ידוע גם כתבנית **התפאורן** (Decorator pattern) [Gamma95], מכיון שהמחלקה InstrumentedSet "מקשטת" את הקבוצה (set) על ידי הוספת מרכיבים שונים. לעיתים הצירוף של הרכבה והעברה נקרא בטעות **שילוח** (delegation) או **האצלה**. טכנית, אין זה שילוח, אלא אם כן אובייקט העטיפה מעביר את עצמו אל האובייקט העטוף [Gamma95].

החסרונות של מחלקות עטיפה הם מעטים. אזהרה אחת היא שמחלקות עטיפה אינן מתאימות לשימוש **בסביבות עבודה של התקשרות חוזרת** (callback frameworks), שבה אובייקטים מעבירים ייחוסים עצמיים אל אובייקטים אחרים לשם קריאות מאוחרות יותר (לכך הכוונה במושג "התקשרות חוזרת"). מכיון שהאובייקט העטוף אינו יודע על העוטף שלו, הוא מעביר ייחוס אל עצמו (this) וההתקשרויות החוזרות מתעלמות מהעוטף. הדבר ידוע כ**בעיית SELF** (SELF problem) [Lieberman86]. יש אנשים המודאגים בדבר ההשפעה על הביצועים הנובעת מקריאות לשיטות העברה, או מההשפעה על טביעת האצבע בזיכרון של אובייקטי עטיפה. בפועל, אין השפעה רבה לאף אחד מהדברים האלה. קצת משעמם לכתוב שיטות העברה, אבל השעמום מופג במקצת על ידי העובדה שאתה צריך לכתוב בנאי אחד בלבד.

הורשה מתאימה רק בנסיבות בהן תת-המחלקה היא באמת **תת-טיפוס** (subtype) של מחלקת-העל. במילים אחרות, מחלקה B רשאית להרחיב את המחלקה A רק אם מתקיימים יחסי "is-a" בין שתי המחלקות. אם אתה מתפתה להרחיב את מחלקה A באמצעות מחלקה B, שאל את עצמך את השאלה: "האם כל B היא באמת A?" אם אינך יכול לענות בכנות "כן" לשאלה זו, B אינה רשאית להרחיב את A. אם התשובה היא "לא", אז לעיתים קרובות מדובר במקרה שבו B צריכה להכיל מופע פרטי של A ולחשוף API פשוט וקטן יותר: A אינה חלק חיוני של B, אלא רק פרט מהמימוש שלה.

יש מספר הפרות ברורות של עיקרון זה בספריות הפלטפורמה של Java. לדוגמה, מחסנית אינה Vector, ולכן Stack אינה רשאית להרחיב את Vector. בדומה, רשימת תכונות אינה טבלת ערבול, ולכן Properties אינה רשאית להרחיב את Hashtable. בשני המקרים, היה מתאים יותר להשתמש בהרכבה.

אם אתה משתמש בהורשה במקום שבו מתאים להשתמש בהרכבה, אתה חושף שלא לצורך את פרטי המימוש. ה-API שיתקבל כובל אותך למימוש המקורי, ולעולם מגביל את ביצועי המחלקה שלך. חשוב מכך, על ידי חשיפת התכונות הפנימיות אתה מאפשר ללקוח לגשת אליהן ישירות. במקרה הקל יותר, הדבר יכול להוביל לסמנטיקה מבלבלת. לדוגמה, אם p מתייחס למופע Properties, אז p.getProperty(key) עשויה להניב תוצאות שונות מאשר p.get(key). כלומר, השיטה הראשונה לוקחת ברירות מחדל בחשבון, בעוד שהשיטה השנייה אשר הורשה מ-Hashtable, אינה עושה זאת. החשוב ביותר הוא שהלקוח עשוי להיות מסוגל לקלקל קבועים של תת-המחלקה על ידי שינוי מחלקת-העל באופן ישיר. במקרה של Properties, המתכננים התכוונו שרק מחרוזות תהיינה מותרות כמפתחות וערכים, אבל גישה ישירה ל-Hashtable שמתחתיה, מאפשרת הפרה של כלל זה עבור הקבוע. לאחת שהכלל הופר, אי אפשר יהיה להשתמש יותר בחלקים אחרים של ה-API של Properties (load ו-store). עד שהתגלתה בעיה זו, כבר אי אפשר היה לתקן אותה, כי לקוחות היו תלויים בשימוש במפתחות וערכים שאינם מחרוזות.

יש קבוצה אחרונה של שאלות אשר עליך לשאול את עצמך לפני שאתה מחליט אם להעדיף הורשה על פני הרכבה. האם יש פגמים ב-API של המחלקה שאתה שוקל להרחיב? אם כן, האם אתה מוכן להפיץ פגמים אלה לתוך ה-API של המחלקה שלך? הורשה מפיצה כל פגם ב-API של מחלקת-העל, בעוד שהרכבה מאפשרת לך לתכנן API חדש אשר מסתיר פגמים אלה.

לסיכום, הורשה היא כלי רב-עוצמה, אך היא בעייתית, מכיון שהיא מפירה את עיקרון הכימוס (encapsulation) והסתרת המידע. היא מתאימה רק כאשר יחסי תת-טיפוס אמיתיים מתקיימים בין תת-המחלקה לבין מחלקת-העל. אפילו אז, הורשה עלולה להביא לפריצה או פגיעה, אם תת-המחלקה נמצאת בחבילת תוכנה אחרת מאשר מחלקת-העל, ומחלקת-העל לא תוכננה מראש להרחבה.

כדי להימנע מתקלה זו, השתמש בהרכבה ובהעברה ולא בהורשה, במיוחד אם יש ממשק מתאים למימוש מחלקת עטיפה. לא רק שמחלקות עטיפה יציבות יותר מתת-מחלקות, אלא הן גם רבות עוצמה יותר.

נושא 15: תכנן ותעד ביצוע הורשה או שתאסור שימוש בה

בנושא 14 ניתנה אזהרה מפני הסכנה שביצירת תת-מחלקה ממחלקה "זרה" אשר לא תוכננה ותועדה מראש להורשה. אם כן, למה הכוונה בתכנון ותיעוד של מחלקה להורשה?

לצורך תכנון הורשה, **המחלקה חייבת לתעד במדויק את ההשפעות של דריסת אחת השיטות שלה**. במילים אחרות, המחלקה חייבת לתעד את ה**שימוש העצמי** (self-use) שלה בשיטות שניתן לדרוס: עבור כל שיטה או בנאי מוגנים או ציבוריים, התיעוד שלהם חייב להראות לאיזה שיטות שניתן לדרוס הם קוראים, באיזה סדר וכיצד התוצאות של כל קריאה משפיעות על העיבוד העוקב (במונח **ניתן לדרוס** (overridable), אנו מתכוונים ל"לא סופי" וגם "ציבורי" או "מוגן"). באופן כללי יותר, מחלקה חייבת לתעד את כל הנסיבות שבהן היא עשויה לקרוא לשיטה שניתן לדרוס. לדוגמה, קריאות יכולות לבוא ממטלות רקע או מאתחלים סטטיים.

בהתאם למוסכמות, שיטה שקוראת לשיטות אשר ניתן לדרוס, מכילה תיאור של קריאות אלו בסוף הערת התיעוד (doc) שלה.
התיאור פותח בביטוי This implementation ("מימוש זה"). אין להתייחס לביטוי הזה כמעיד על כך שההתנהגות עשויה להשתנות מגירסה לגירסה. הוא רק רומז על כך שהתיאור מתייחס לפעולה הפנימית של השיטה. הנה דוגמה שהועתקה מהמפרט של java.util.AbstractCollection:

public boolean remove(Object o)

Removes a single instance of the specified element from this collection, if it is present (optional operation). More formally, removes an element e such that (o==null ? e==null :o.equals(e)), if the collection contains one or more such elements. Returns true if the collection contained the specified element (or equivalently, if the collection changed as a result of the call).

This implementation iterates over the collection looking for the specified element. If it finds the element, it removes the element from the collection using the iterator's remove method. Note that this implementation throws an UnsupportedOperationException if the iterator returned by this collection's iterator method does not implement the remove method.

תיעוד זה לא מותיר כל ספק בנוגע לכך שדריסת השיטה iterator תשפיע על ההתנהגות של השיטה remove. יתר על כן, הוא מתאר במדויק כיצד ההתנהגות של Iterator שמוחזר על ידי השיטה iterator תשפיע על ההתנהגות של השיטה remove.

השווה זאת למצב שבנושא 14, שבו יצירת תת-מחלקה של HashSet על ידי המתכנת לא יכולה היתה לומר אם דריסת השיטה add תשפיע על ההתנהגות של השיטה addAll.

אבל האם אין זה מפר את האמירה לפיה תיעוד טוב של API צריך לתאר **מה** שיטה נתונה עושה, ולא **איך** היא עושה זאת?! דרך פעולה זו אכן מפירה! זוהי תוצאה אומללה של העובדה שהורשה מפירה את עיקרון הכימוס. כדי לתעד מחלקה כך שיהיה אפשר ליצור ממנה תת-מחלקות בביטחה, אתה חייב לתאר את פרטי המימוש, אשר בנסיבות אחרות לא היה צורך לתארם.

תכנון להורשה מחייב יותר מאשר תיעוד תכונות של שימוש עצמי. כדי לאפשר למתכנתים לכתוב תת-מחלקות יעילות ללא כאבי ראש מופרזים, **מחלקה אמורה לספק נקודות אחיזה טובות במהלך הפעולה הפנימית שלה בצורת שיטות מוגנות שנבחרו בתבונה**, או במקרים נדירים, שדות מוגנים. לדוגמה, ראה את השיטה removeRange מתוך java.util.AbstractList:

protected void removeRange(int fromIndex,int toIndex)

Removes from this list all of the elements whose index is between fromIndex, inclusive, and toIndex, exclusive. Shifts any succeeding elements to the left (reduces their index). This call shortens the ArrayList by (toIndex - fromIndex) elements. (If toIndex==fromIndex, this operation has no effect.)

This method is called by the clear operation on this list and its sublists. Overriding this method to take advantage of the internals of the list implementation can substantially improve the performance of the clear operation on this list and its subLists.

This implementation gets a list iterator positioned before fromIndex and repeatedly calls ListIterator.next followed by ListIterator.remove, until the entire range has been removed. Note: If ListIterator.remove requires linear time, this implementation requires quadratic time.

Parameters:

fromIndex	index of first element to be removed.
toIndex	index after last element to be removed.

למשתמשי קצה של מימוש List אין כל עניין בשיטה זו. היא מסופקת אך ורק כדי להקל על תת-מחלקות לספק שיטת clear מהירה עבור תת-רשימות. בהיעדר השיטה removeRange, הן תיאלצנה להסתפק בביצועים כפולים ומכופלים כאשר השיטה clear נקראת על פי תת-רשימות, או לכתוב שוב את כל מנגנון sublist מההתחלה - ואין זו משימה קלה!

אם כן, כיצד תחליט אילו שיטות או שדות מוגנים לחשוף, כאשר אתה מתכנן מחלקה להורשה? לרוע המזל, אין שום פתרון קסם לכך. המיטב שבאפשרותך לעשות הוא לחשוב היטב, לנחש את הניחוש הטוב ביותר שלך ואז לבחון אותו על ידי כתיבת מספר תת-מחלקות. אתה צריך לספק כמה שפחות שיטות או שדות מוגנים, כי כל אחד מהם מייצג התחייבות לפרט מימוש כלשהו. עם זאת, אסור לך לספק מעט מדי מידע, כיון ששיטה מוגנת חסרה יכולה לגרום למחלקה להיות בלתי ראויה לחלוטין להורשה.

כאשר אתה מתכנן להורשה מחלקה אשר סביר שייעשה בה שימוש נרחב, עליך לדעת שאתה מחויב **לעולם** לתבניות השימוש העצמי שאתה מתעד, ולהחלטות המימוש המפורשות בשיטות או בשדות המוגנים שלה. התחייבויות אלו יכולות להקשות או אפילו לא לאפשר את שיפור ביצועי המחלקה או התפקודיות שלה בגירסה עוקבת.

כמו כן, שים לב לכך שהתיעוד המיוחד הנדרש להורשה פוגע בתיעוד הרגיל, אשר מיועד עבור מתכנתים שיוצרים מופעים של המחלקה שלך וקוראים לשיטות הפועלות עליהם. נכון לכתיבת ספר זה, יש דרכים מועטות (בצורת כלים או מוסכמות הערה) כדי להפריד בין תיעוד API רגיל לבין מידע שיש בו עניין רק למתכנתים שעשויים לממש תת-מחלקות.

יש עוד מספר הגבלות מועט שמחלקה חייבת לציית להן כדי לאפשר הורשה. **לבנאים אסור לקרוא לשיטות שניתנות לדריסה**, באופן ישיר או באופן עקיף. הפרה של כלל זה כנראה תביא לכישלון תוכנית. הבנאי של מחלקת-העל פועל לפני הבנאי של תת-המחלקה, ואז השיטה הדורסת שבתת-המחלקה תיקרא לפני שהבנאי של תת-המחלקה הופעל. אם השיטה הדורסת תלויה באתחול כלשהו שמתבצע על ידי הבנאי של תת-המחלקה, אז השיטה לא תתנהג כמצופה. כדי להבהיר זאת, הנה מחלקה זעירה המפירה כלל זה:

```
public class Super {
   // Broken - constructor invokes overridable method
   public Super() {
      m();
   }

   public void m() {
   }
}
```

הנה תת-מחלקה הדורסת את m, אשר נקראת בטעות על ידי הבנאי היחיד של Super:

```
final class Sub extends Super {
   private final Date date; // Blank final, set by constructor

   Sub() {
      date = new Date();
   }

   // Overrides Super.m, invoked by the constructor Super()
   public void m() {
      System.out.println(date);
   }

   public static void main(String[] args) {
      Sub s = new Sub();
      s.m();
   }
}
```

אתה עשוי לצפות מתוכנית זו להדפיס את התאריך פעמיים, אבל היא מדפיסה null בפעם הראשונה, כי השיטה m נקראת על ידי הבנאי ()Super לפני שלבנאי ()Sub היה סיכוי לאתחל את השדה date. שים לב לכך שתוכנית זו שומרת שדה סופי בשני מצבים שונים.

ממשקי Cloneable ו-Serializable מציגים קשיים מיוחדים כאשר מתכננים אותם להורשה. בדרך כלל אין זה רעיון טוב למחלקה שתוכננה להורשה לממש אחד מהממשקים האלה, כיון שהם מציבים נטל משמעותי על מתכנתים אשר מרחיבים את המחלקה. עם זאת, יש פעולות מיוחדות שאתה יכול לעשות בהן שימוש, כדי לאפשר לתת-מחלקות לממש ממשקים אלה בלי שתגרום להם לעשות זאת. פעולות אלו מתוארות בנושאים 10 ו- 54.

אם החלטת לבסוף לממש Cloneable ו-Serializable במחלקה שתוכננה להורשה, עליך להיות ער לכך שמכיון שהשיטות clone ו-readObject מתנהגות באופן מאוד דומה להתנהגות בנאים, חלה הגבלה דומה: **לא clone ולא readObject צריכות לקרוא לשיטה שניתנת לדריסה, באופן ישיר או באופן עקיף**. במקרה של השיטה readObject, השיטה הדורסת תפעל לפני שהאובייקט הורכב חזרה מתוך הזרם (deserialized). במקרה של השיטה clone, השיטה הדורסת תרוץ לפני שלשיטות clone של תת-המחלקה יהיה סיכוי לתקן את מצבו של האובייקט המשוכפל. בשני המקרים, סביר שהתוכנית תיכשל בעקבות כך. במקרה של השיטה clone, הכישלון יכול לגרום לנזק לאובייקט המקורי, כמו גם לאובייקט התואם עצמו.

לבסוף, אם אתה מחליט לממש Serializable במחלקה אשר תוכננה להורשה ולמחלקה יש שיטת readResolve או writeReplace, **אתה חייב לקבוע את השיטה readResolve או writeReplace כמוגנת, ולא כפרטית**. אם שיטות אלו הן פרטיות, תת-המחלקה תתעלם מהן באופן סמוי. זהו מקרה אחד נוסף שבו פרט מימוש הופך חלק מ-API של מחלקה כדי להתיר הורשה.

עד כה, צריך להיות ברור לך ש**תכנון מחלקה להורשה מציב עליה הגבלות ניכרות**. אין זו החלטה שיש לקבל בפזיזות. יש כמה מצבים שברור מאליו שזה הדבר הנכון לעשות, כמו במחלקות מופשטות, כולל **מימושי שלד** (skeletal implementations) של ממשקים (נושא 16). יש מצבים אחרים בהם ברור שתהיה זו טעות לעשות זאת, כמו מחלקות שאינן ניתנות לשינוי (נושא 13).

אבל מה בנוגע למחלקות מגובשות רגילות? באופן מסורתי, הן גם לא-סופיות וגם לא-מתוכננות ומתועדות ליצור מהן תת-מחלקות, אבל מצב עניינים זה מסוכן. בכל פעם שמבוצע שינוי במחלקה כזאת, יש סיכוי שמחלקות לקוח המרחיבות את המחלקה תיפגענה. אין זו בעיה תיאורטית בלבד. מקובל לקבל דוח על כל באג שקשור ליצירת תת-מחלקות לאחר שינוי התכונות הפנימיות של מחלקה מגובשת לא-סופית, אשר לא תוכננה ולא תועדה להורשה.

הדרך הטובה ביותר לפתור בעיה זו היא להימנע מיצירת תת-מחלקות מתוך מחלקות אשר לא תוכננו ולא תועדו ליצירה כזו באופן בטוח. יש שתי דרכים למנוע יצירת תת-מחלקות. הקלה מבין השתיים היא להצהיר על המחלקה כסופית. הדרך החילופית היא לעשות את כל הבנאים פרטיים או פרטיים-בחבילה ולהוסיף יצרנים סטטיים ציבוריים במקום הבנאים. חלופה זו, המאפשרת את הגמישות להשתמש בתת-מחלקות באופן פנימי, נדונה בנושא 13. כל גישה אחרת מקובלת גם כן למטרה זו.

עצה זו עשויה להיות מעט שנויה במחלוקת, מכיון שמתכנתים רבים הורגלו ליצור תת-מחלקות מתוך מחלקות מגובשות רגילות, כדי להוסיף אמצעים כמו יכולת פעולה, הודעה וסינכרון, או כדי להגביל תפקודיות. אם במחלקה יש מימוש של ממשק כלשהו הלוכד את עיקריה, כמו Set, List או Map, אז אינך צריך לחוש נקיפות מצפון כלשהן אם תימנע מיצירת תת-מחלקות. **תבנית המחלקה העוטפת** (wrapper class pattern) שתוארה בנושא 14, מספקת תחליף מעולה להורשה כדי לשנות את התפקוד של מחלקה קיימת.

אם במחלקה מגובשת אין מימוש של ממשק סטנדרטי, אתה עשוי לגרום אי נוחות למתכנתים מסוימים על ידי מניעת הורשה. אם אתה מרגיש שאתה חייב לאפשר הורשה למחלקה כזו, גישה הגיונית אחת היא להבטיח שהמחלקה לעולם לא תקרא לאף אחת מהשיטות שלה שניתנות לדריסה, ויש לתעד הנחיה זו.

במילים אחרות, בטל לחלוטין את השימוש העצמי של המחלקה בשיטות שניתנות לדריסה.

אם תעשה זאת, תיצור מחלקה אשר בטוחה באופן מתקבל על הדעת כדי ליצור ממנה תת-מחלקות. דריסת שיטה לעולם לא תשפיע על ההתנהגות של שיטה אחרת כלשהי.

אתה יכול לסלק באופן מכני את השימוש העצמי של מחלקה בשיטות שניתנות לדריסה, ללא שינוי ההתנהגות שלה. הזז את הגוף של כל שיטה שניתנת לדריסה אל "שיטת עזר" פרטית, וגרום לכל שיטה שניתנת לדריסה לקרוא לשיטת העזר הפרטית שלה. לאחר מכן, החלף כל שימוש עצמי בשיטה שניתנת לדריסה, בקריאה ישירה של שיטת העזר הפרטית של השיטה שניתנת לדריסה.

נושא 16: העדף ממשקים על פני מחלקות מופשטות

שפת התכנות Java מספקת שני מנגנונים להגדרת טיפוס (type), המאפשר מימושים שונים: ממשקים ומחלקות מופשטות. ההבדל הברור ביותר בין שני המנגנונים הוא בכך שמחלקות מופשטות מורשות להכיל מימושים של כמה שיטות, בעוד שממשקים אינם מורשים לעשות זאת. הבדל חשוב יותר הוא שכדי לממש את הטיפוס שהוגדר על ידי מחלקה מופשטת, המחלקה חייבת להיות תת-מחלקה של המחלקה המופשטת. לעומת זאת, כל מחלקה המגדירה את כל השיטות הנדרשות ומצייתת להסכם הכללי, רשאית לממש ממשק, ללא תלות במיקום המחלקה בהיררכיית המחלקות. מכיון ש-Java מתירה הורשה יחידה בלבד, הגבלה זו על מחלקות מופשטות מצמצמת את השימוש בהן כהגדרות טיפוס.

אפשר לשפר מחלקות קיימות בקלות, כך שתממשנה ממשק חדש. כל שאתה צריך לעשות הוא להוסיף את השיטות הנדרשות אם הן עדיין לא קיימות ולהוסיף פסוקית implements להצהרת המחלקה. לדוגמה, הרבה מחלקות קיימות שופרו כדי לממש את ממשק Comparable כאשר הוא הוכנס לראשונה לפלטפורמת Java. בדרך כלל, מחלקות קיימות אינן יכולות להיות מותאמות מחדש כדי להרחיב מחלקה מופשטת חדשה.
אם אתה רוצה ששתי מחלקות תרחבנה את אותה מחלקה מופשטת, עליך להציב את המחלקה המופשטת ברמה גבוהה בהיררכיית הטיפוס, במקום שבו היא יוצרת תת-מחלקות מאב של שתי המחלקות. לרוע המזל, הדבר גורם נזק רב להיררכיית הטיפוס ומכריח את כל הצאצאים של האב המשותף להרחיב את המחלקה המופשטת החדשה, בין אם הדבר מתאים או לא.

ממשקים הם אידיאליים להגדרת mixins. המושג **mixin** מציין טיפוס אשר מחלקה יכולה לממש בנוסף ל"טיפוס הראשוני" שלה, כדי להצהיר שהיא מספקת כמה התנהגויות אופציונליות נוספות. לדוגמה, Comparable הוא ממשק mixin המאפשר למחלקה להצהיר שהמופעים שלה בני-סידור והשוואה. ממשק כזה נקרא mixin, כי הוא מאפשר לתפקוד האופציונלי להיות "מעורב" בתפקוד הראשוני של הטיפוס. מחלקות מופשטות אינן יכולות לשמש להגדרת טיפוסי mixin מאותה סיבה שבגללה אי אפשר לבנות אותן ממחלקות קיימות: למחלקה לא יכול להיות יותר מהורה אחד ואין שום מקום הגיוני בהיררכיית המחלקה לשים mixin.

ממשקים מאפשרים בניית סביבות עבודה שאינן היררכיות. היררכיות טובות מאוד עבור ארגון דברים מסוימים, אבל יש דברים אחרים שאינם מתאימים להיררכיה קשוחה. לדוגמה, נניח שיש לנו ממשק שמייצג זמר וממשק אחר שמייצג מלחין:

```
public interface Singer {
    AudioClip Sing(Song s);
}
```

```
public interface Songwrite {
    Song compose(boolean hit);
}
```

בחיים האמיתיים, יש זמרים שהם גם מלחינים. מכיון שהשתמשנו בממשקים ולא במחלקות מופשטות כדי להגדיר את הטיפוסים האלה, מחלקה יחידה רשאית בהחלט לממש גם Singer וגם Songwriter. למעשה, אנו יכולים להגדיר ממשק שלישי שמרחיב גם את Singer וגם את Songwriter ומוסיף שיטות חדשות המתאימות לשילוב הזה:

```
public interface SingerSongwriter extends Singer,Songwriter {
    AudioClip strum();
    void actSensitive();
}
```

אינך זקוק תמיד לרמת גמישות זו, אבל כאשר אתה זקוק לה, ממשקים הם בבחינת מצילי חיים. החלופה היא היררכיית מחלקה מנופחת שמכילה מחלקה נפרדת עבור כל שילוב נתמך של התכונות. אם יש n תכונות במערכת הטיפוס, אז יש 2 בחזקת n צירופים אפשריים שתצטרך לתמוך בהם. הדבר ידוע כ**התפוצצות קומבינטורית** (combinatorial explosion). היררכיות מחלקה מנופחות יכולות להוביל למחלקות מנופחות, המכילות שיטות רבות אשר נבדלות זו מזו רק בטיפוס הארגומנטים שלהן, מכיון שאין טיפוסים בהיררכיית המחלקה כדי ללכוד התנהגויות מקובלות.

ממשקים מאפשרים שיפורי תפקוד חזקים ובטוחים באמצעות סגנון התכנות של מחלקה עוטפת (wrapper class), אשר תוארה בנושא 14. אם אתה משתמש במחלקות מופשטות כדי להגדיר טיפוסים, אתה מותיר את המתכנת אשר רוצה להוסיף תפקודיות ללא שום חלופה, פרט לשימוש בהורשה. המחלקות שתתקבלנה תהיינה פחות חזקות ויותר שבירות מאשר המחלקות העוטפות.

למרות שממשקים אינם רשאים להכיל מימושי שיטות, שימוש בממשקים כדי להגדיר טיפוסים אינו מונע ממך לספק למתכנתים עזרה במימוש. **אתה יכול לשלב את התכונות הטובות של ממשקים ושל מחלקות מופשטות על ידי אספקת מחלקת מימוש שילדית (skeletal) מופשטת, שמותאמת לכל ממשק לא טריוויאלי שאתה מייצא**. הממשק עדיין מגדיר את הטיפוס, אבל המימוש השילדי מבצע את כל העבודה של מימוש הממשק.

לפי המוסכמה, מימושים שילדיים מכונים AbstractInterface, כאשר Interface הוא שם הממשק אשר הם מממשים. לדוגמה, סביבת העבודה של האוספים מספקת מימוש שילדי שיתאים לכל ממשק אוסף עיקרי: AbstractCollection, AbstractSet, AbstractList ו-AbstractMap.

מימושים דמויי שלד שתוכננו כיאות, מקילים **מאוד** על מתכנתים לספק את המימושים שלהם לממשקים שלך. לדוגמה, הנה שיטת יצור סטטית שמכילה מימוש List שלם שמתפקד באופן מלא:

```
// List adapter for int array

public class IntList {
   static List intArrayAsList(final int[] a) {
      if (a == null)
         throw new NullPointerException();

      return new AbstractList() {
         public Object get(int i) {
            return new Integer(a[i]);
         }

         public int size() {
            return a.length;
         }

         public Object set(int i, Object o) {
            int oldVal = a[i];
            a[i] = ((Integer)o).intValue();
            return new Integer(oldVal);
         }
      };
   }
```

כאשר אתה לוקח בחשבון את כל מה שהמימוש של List עושה עבורך, אתה מבין עד כמה דוגמה זו היא המחשה מרשימה לעוצמה שבמימושי שלד. במקרה, דוגמה זו היא **מתאם** (**Adapter**) [Gamma95] אשר מאפשר למערך int להיות מוצג כרשימת מופעי Integer. בגלל כל התרגום הלוך ושוב בין ערכי int למופעי Integer, הביצוע אינו אופטימלי. שים לב לכך שמסופק יצרן סטטי ושהמחלקה היא **מחלקה אנונימית** שאינה נגישה (נושא 18), שחבויה בתוך היצרן הסטטי.

היופי שבמימושים דמויי שלד הוא שהם מספקים את העזרה המימושית של מחלקות מופשטות, בלי לכפות את האילוצים החמורים אשר מחלקות מופשטות כופות כאשר הן משמשות כהגדרות טיפוס. עבור מרבית המיישמים של ממשק, הרחבת המימוש דמוי השלד היא האפשרות הברורה, אך זוהי אפשרות בלבד והיא אינה מחייבת. אם אי אפשר לגרום למחלקה קיימת **להרחיב** את המימוש דמוי השלד (לרוב כיון שהיא כבר מרחיבה מחלקה אחרת), נוכל תמיד לדאוג שהמחלקה **תממש** את הממשק באופן ידני. יתר על כן, המימוש דמוי השלד עדיין יכול לעזור בביצוע המימוש. המחלקה שמממשת את הממשק יכולה להעביר קריאות של שיטות ממשק למופע מוכל של מחלקה פנימית פרטית, אשר מרחיבה את המימוש דמוי השלד.

טכניקה זו, אשר ידועה בכינויה **הורשה כפולה מדומה** (simulated multiple inheritance), די קשורה לרעיון מחלקת העטיפה שנדונה בנושא 14. היא מספקת את מרבית היתרונות של הורשה כפולה תוך הימנעות מהמלכודות.

כתיבת מימוש דמוי שלד היא פשוטה יחסית, אם כי משעממת במקצת. ראשית עליך ללמוד את הממשק ולהחליט אילו שיטות יוכל המשתמש לממש בקלות בשלד. שיטות אלו יהיו השיטות המופשטות במימוש דמוי השלד שלך. לאחר מכן אתה חייב לספק מימושים מוצקים של כל השיטות האחרות בממשק. לדוגמה, הנה מימוש דמוי שלד של ממשק Map.Entry. נכון לזמן כתיבת שורות אלו, מחלקה זו אינה כלולה בספריות הפלטפורמה של Java, אבל ייתכן שהיא תיכלל בעתיד:

```
// Skeletal Implementation
public abstract class AbstractMapEntry implements Map.Entry {
   // Primitives
   public abstract Object getKey();
   public abstract Object getValue();

   // Entries in modifiable maps must override this method
   public Object setValue(Object value) {
      throw new UnsupportedOperationException();
   }

   // Implements the general contract of Map.Entry.equals
   public boolean equals(Object o) {
      if (o == this)
         return true;
      if (! (o instanceof Map.Entry))
         return false;
      Map.Entry arg = (Map.Entry)o;

      return eq(getKey(),   arg.getKey()) &&
            eq(getValue(), arg.getValue());
   }

   private static boolean eq(Object o1, Object o2) {
      return (o1 == null ? o2 == null : o1.equals(o2));
   }

   // Implements the general contract of Map.Entry.hashcode
   public int hashCode() {
      return
         (getKey()   == null ? 0 :   getKey().hashCode()) ^
         (getValue() == null ? 0 : getValue().hashCode());
   }
}
```

בגלל שמימושים דמויי שלד תוכננו להורשה, עליך לעבור על כל הקווים המנחים בנוגע לתכנון ותיעוד להורשה, כפי שהוסבר בנושא 15. מסיבות של קיצור, הערות התיעוד הושמטו מהדוגמה הקודמת, אבל תיעוד טוב הוא חיוני מאוד למימושים דמויי שלד.

לשימוש במחלקות מופשטות כדי להגדיר טיפוסים אשר מאפשרים מימושים כפולים יש יתרון גדול אחד על פני שימוש בממשקים: **הרבה יותר קל לעדכן מחלקה מופשטת מאשר לעדכן ממשק**. אם, בגירסה עוקבת, אתה רוצה להוסיף שיטה חדשה למחלקה מופשטת, אתה תמיד יכול להוסיף שיטה שמכילה מימוש ברירת מחדל סביר. ואז, כל המימושים הקיימים של המחלקה המופשטת יספקו את השיטה החדשה. הדבר אינו עובד עבור ממשקים.

באופן כללי, אין אפשרות להוסיף שיטה לממשק ציבורי בלי לשבור את כל התוכניות הקיימות שמשתמשות בממשק הזה. מחלקות אשר מימשו קודם לכן את הממשק תחסרנה את השיטה החדשה ואי אפשר יהיה להדר אותן. אתה יכול לצמצם מעט את הנזק על ידי הוספת השיטה החדשה למימוש דמוי השלד במקביל להוספת השיטה לממשק, אבל אין זה פותר את הבעיה באמת. כל מימוש אשר לא מרחיב את המימוש דמוי השלד עדיין יישבר.

לכן, יש לתכנן ממשקים ציבוריים בזהירות. לאחר שממשק כבר הופץ ומומש באופן נרחב, כמעט בלתי אפשרי לשנות אותו. אתה באמת צריך לעשות אותו נכון כבר בפעם הראשונה. אם ממשק מכיל פגם שולי, הוא יציק לך ולמשתמשים בו לנצח. אם ממשק לקוי באופן רציני, הדבר יכול לחרוץ את דינו של ה-API. הדבר הטוב ביותר לעשות כאשר מפיצים ממשק חדש הוא לתת לכמה שיותר מתכנתים לממש את הממשק בכמה שיותר דרכים אפשריות, לפני שהממשק נקבע סופית ומופץ. הדבר יאפשר לך לגלות פגמים כל עוד אתה יכול לתקן אותם.

לסיכום, ממשק הוא בדרך כלל הדרך הטובה ביותר להגדיר טיפוס אשר מאפשר מימושים כפולים. חריג לכלל זה הוא המקרה שבו ישנה חשיבות גדולה יותר לאפשר עדכון והרחבה לטיפוס, מאשר גמישות וכוח. בנסיבות אלו, אתה צריך להשתמש במחלקה מופשטת כדי להגדיר את הטיפוס, אבל רק אם אתה מבין ומקבל את ההגבלות. אם אתה מייצא ממשק לא טריוויאלי, כדאי שתשקול ברצינות אספקת מימוש דמוי שלד מתאים. לבסוף, אתה צריך לתכנן את כל הממשקים הציבוריים שלך בתשומת הלב הגדולה ביותר ולבחון אותם ביסודיות על ידי כתיבת מימושים כפולים.

נושא 17: השתמש בממשקים רק כדי להגדיר טיפוסים

כאשר מחלקה מממשת ממשק, הממשק משמש **כטיפוס** אשר יכול לשמש ליצירת ייחוס למופעים של מחלקות הממשות את הממשק. העובדה שמחלקה מממשת ממשק צריכה אם כך לתאר את מה שהלקוח יכול לעשות עם מופעים שלה. יהיה זה בלתי הולם להגדיר ממשק עבור כל מטרה אחרת.

סוג אחד של ממשק אשר אינו עומד במבחן זה הוא הממשק המכונה **ממשק קבוע** (constant interface). ממשק כזה אינו מכיל שיטה כלשהי, הוא מורכב משדות סופיים סטטיים בלבד, שכל אחד מהם מייצא קבוע. מחלקות אשר משתמשות בקבועים אלה מממשות את הממשק כדי להימנע מהצורך להגדיר שמות של קבועים עם שם מחלקה.

הנה דוגמה:

```
// Constant interface pattern - do not use!
public interface PhysicalConstants {
   // Avogadro's number (1/mol)
   static final double AVOGADROS_NUMBER   = 6.02214199e23;

   // Boltzmann constant (J/K)
   static final double BOLTZMANN_CONSTANT = 1.3806503e-23;

   // Mass of the electron (kg)
   static final double ELECTRON_MASS      = 9.10938188e-31;
}
```

תבנית הממשק הקבוע היא שימוש עלוב לממשקים. העובדה שמחלקה משתמשת במספר קבועים באופן פנימי היא פרט מימוש. מימוש ממשק קבוע גורם לפרט מימוש זה לדלוף לתוך ה-API המיוצא של המחלקה. אין זה חשוב כלל למשתמשים של מחלקה שהמחלקה מממשת ממשק קבוע. למעשה, הדבר אפילו עלול לבלבל אותם. גרוע מכך, זה מציג התחייבות: אם בגירסה עתידית המחלקה תשונה כך שהיא לא תצטרך יותר להשתמש בקבועים, היא עדיין תצטרך לממש את הממשק כדי להבטיח תאימות. אם מחלקה לא סופית מממשת ממשק קבוע, לכל תת-המחלקות שלה יהיו מרחבי שמות ה"מזוהמים" על ידי הקבועים שבממשק. ישנם מספר ממשקים קבועים בספריות הפלטפורמה של Java, כמו java.io.ObjectStreamConstants. לממשקים אלה יש להתייחס כאל יוצאי דופן שאין לחקות אותם.

אם אתה רוצה לייצא קבועים, יש כמה אפשרויות סבירות. אם הקבועים קשורים בחוזקה למחלקה או ממשק קיימים, אתה צריך להוסיף אותם למחלקה או לממשק. לדוגמה, כל מחלקות העטיפה המספריות שבספריות הפלטפורמה של Java, כמו Integer ו-Float, מפיצות קבועי MIN_VALUE ו-MAX_VALUE.

אם הקבועים נראים הכי טוב כאיברים של טיפוס נספר, אתה צריך להפיץ אותם עם מחלקת typesafe enum (נושא 21). אחרת, אתה צריך להפיץ את הקבועים בעזרת מחלקת עזר שלא ניתן ליצור לה מופע (נושא 3). הנה גרסת מחלקת עזר של הדוגמה PhysicalConstants שלמעלה:

```
// Constant utility class
public class PhysicalConstants {
  private PhysicalConstants() { }  // Prevents instantiation

  public static final double AVOGADROS_NUMBER   = 6.02214199e23;
  public static final double BOLTZMANN_CONSTANT = 1.3806503e-23;
  public static final double ELECTRON_MASS      = 9.10938188e-31;
}
```

למרות שגרסת מחלקת העזר של PhysicalConstants בכל זאת דורשת לציין את שם המחלקה בנוסף לשם הקבוע בכל מקום בו נרצה להשתמש בו, זהו מחיר פעוט שיש לשלם עבור API הגיוני. ייתכן שהשפה תרשה לבסוף לייבא שדות סטטיים. בינתיים, אתה יכול למזער את הצורך ב"הקלדה מיותרת" על ידי אחסון קבועים שמשתמשים בהם בתדירות רבה במשתנים מקומיים או בשדות סטטיים פרטיים, לדוגמה:

```
private static final double PI = Math.PI;
```

לסיכום, ממשקים צריכים לשמש אך ורק כדי להגדיר טיפוסים, הם לא צריכים לשמש להפצת קבועים.

נושא 18: העדף מחלקות מקוננות סטטיות על פני כאלו שאינן סטטיות

מחלקה מקוננת היא מחלקה שמוגדרת בתוך מחלקה אחרת. מחלקות מקוננות צריכות להתקיים אך ורק כדי לשרת את המחלקות שבתוכן הן מקוננות. אם מחלקה מקוננת יכולה להיות שימושית גם בכמה הקשרים אחרים, זה סימן שהיא צריכה להיות מחלקה רגילה (top level class) ולא מחלקה מקוננת. ישנם ארבעה סוגים של מחלקות מקוננות: **מחלקות מקוננות סטטיות**, **מחלקות מקוננות שאינן סטטיות**, **מחלקות אנונימיות** ו**מחלקות מקומיות**. כולן פרט לסוג הראשון ידועות כ**מחלקות פנימיות**. נושא זה יסביר לך מתי ובאיזה סוג של מחלקה מקוננת עליך להשתמש ומדוע.

מחלקה מקוננת סטטית היא הסוג הפשוט ביותר של מחלקה מקוננת. ההגדרה המתאימה לה ביותר היא מחלקה רגילה אשר הוגדרה בתוך מחלקה אחרת ויש לה גישה לכל האיברים הסטטיים של המחלקה שמקננת אותה, אפילו לאלה שהוגדרו כפרטיים. מחלקת איבר סטטית היא איבר סטטי של המחלקה המקננת אותה ומצייתת לאותם חוקי נגישות כמו איברים סטטיים אחרים. אם היא מוגדרת כפרטית, היא נגישה רק בתוך המחלקה המקננת אותה וכן הלאה.

שימוש מקובל אחד של מחלקת איבר סטטית הוא כמחלקת עזר ציבורית, אשר שימושית רק בצירוף עם המחלקה החיצונית שלה. לדוגמה, חשוב על typesafe enum אשר מתאר את הפעולות שנתמכות על ידי מחשבון (נושא 21). מחלקת Operation צריכה להיות מחלקת איבר סטטית ציבורית של מחלקת Calculator. לקוחות של מחלקת Calculator יוכלו אז לפנות אל הפעולות תוך שימוש בשמות כמו Calculator.Operation.PLUS ו- Calculator.Operation.MINUS. שימוש זה יודגם בהמשך הנושא.

באופן **תחבירי**, ההבדל היחיד בין מחלקות מקוננות סטטיות לבין מחלקות מקוננות שאינן סטטיות, הוא שבהגדרת מחלקות מקוננות סטטיות יש את המציין static בהצהרות שלהן. למרות הדמיון התחבירי, שני סוגים אלה של מחלקות מקוננות **מאוד שונים** האחד מהשני. כל מופע של מחלקה מקוננת שאינה סטטית מקושר במרומז ל**מופע** של המחלקה החיצונית שמכילה אותו. מתוך שיטות מופע של מחלקות מקוננות שאינן סטטיות, ניתן לקרוא לשיטות השייכות למופע של המחלקה החיצונית. מתוך ייחוס למופע של מחלקה מקוננת שאינה סטטית, אפשר לקבל ייחוס למופע המקנן שאליו הוא מקושר. אם מופע של מחלקה מקוננת יכול להתקיים בבידוד ממופע של המחלקה המקננת אותו, אז המחלקה המקוננת **לא יכולה** להיות מחלקה מקוננת שאינה סטטית: בלתי אפשרי ליצור מופע של מחלקה מקוננת שאינה סטטית ללא מופע מקנן.

הקישור בין מופע של מחלקה מקוננת שאינה סטטית לבין המופע המקנן, נוצרת כאשר הראשון נוצר. אי אפשר לשנות את הקישור לאחר מכן.
בדרך כלל, הקישור נוצר באופן אוטומטי על ידי קריאה לבנאי של המחלקה המקוננת הלא סטטית מתוך שיטת מופע של מופע מטיפוס המחלקה החיצונית, שבתוכה הוא מקונן. אפשר, אם כי הדבר נדיר, ליצור את ההתחברות באופן ידני על ידי שימוש בביטוי enclosingInstance.newMemberClass(args).

כפי שאתה עשוי לצפות, הקישור גוזל מקום במופע של המחלקה המקוננת הלא סטטית ומוסיף זמן לתהליך הבנייה שלו.

שימוש מקובל אחד במחלקה מקוננת לא סטטית הוא כדי להגדיר **מתאם** [Gamma, עמוד 139], אשר מאפשר למופע של המחלקה החיצונית להיראות כמופע של מחלקה אחרת לא קשורה. לדוגמה, מימושים של הממשק Map משתמשים לרוב במחלקות מקוננות לא סטטיות כדי לממש את **תצוגות האוסף** שלהם, אשר מוחזרות על ידי שיטות keySet, entrySet ו-values של Map. באופן דומה, מימושים של ממשקי האוסף כמו Set ו-List, משתמשים לרוב במחלקות מקוננות לא סטטיות כדי לממש את ה-iterators שלהם:

```
//Typical use of a nonstatic member class
public class MySet extends AbstractSet {
  ...//Bulk of the class omitted

  public Iterator iterator() {
    return new MyIterator();
  }

  private class MyIterator implements Iterator {
    ...
  }
}
```

אם אתה מצהיר על מחלקה מקוננת אשר לא דורשת גישה למופע המקנן, זכור להציב את המגביל static בהצהרה, פעולה אשר תעשה אותה מחלקה מקוננת סטטית במקום מחלקה מקוננת לא סטטית. אם תשמיט את המגביל הסטטי, כל מופע של המחלקה המקוננת יכיל ייחוס מיותר למופע המקנן. שמירת הייחוס הזה גוזלת זמן ומקום ללא שום רווח הולם. אם תנסה להקצות מופע למחלקה מקוננת שאינה סטטית ללא מופע מקנן, לא תצליח לעשות זאת, מכיון שמופעי מחלקות מקוננות לא סטטיות צריכים מופעים מקננים.

שימוש מקובל של מחלקות מקוננות סטטיות פרטיות הוא לייצוג מרכיבים של האובייקט, אשר מיוצג על ידי המחלקה המקננת אותן. לדוגמה, חשוב על מופע של Map, אשר מחבר מפתחות עם ערכים. למופעי Map יש בדרך כלל אובייקט Entry פנימי לכל זוג ערך-מפתח שבמפה. למרות שכל כניסה מחוברת עם מפה, השיטות שעל הכניסה (getKey, getValue ו-setValue) לא צריכות גישה למפה. לכן, יהיה זה בזבוז להשתמש במחלקה מקוננת לא סטטית כדי לייצג כניסות. הכי טוב יהיה להשתמש במחלקה מקוננת סטטית. אם תשמיט בטעות את המגביל static בהצהרה של הכניסה, המפה עדיין תעבוד אבל כל כניסה תכיל ייחוס מיותר למפה וזה בזבוז של מקום וזמן.

חשוב במיוחד לבחור באופן נכון בין מחלקות מקוננות סטטיות לבין מחלקות מקוננות לא סטטיות, במקרה שבו המחלקה הנדונה היא איבר מוגן או ציבורי של מחלקה מיוצאת. במקרה הזה, המחלקת המקוננת היא אלמנט API מיוצא ואינה אמורה להשתנות בגירסה עוקבת מסטטית לכזו שאינה סטטית, ללא הפרה של התאימות.

מחלקות אנונימיות אינן דומות לשום דבר אחר בשפת התכנות Java. כמו שאתה בוודאי מצפה, למחלקה אנונימית אין שם. היא אינה איבר של המחלקה שמקננת אותה. במקום להיות מוצהרת יחד עם איברים אחרים, היא גם מוצהרת וגם נעשית מופע ברגע השימוש. מחלקות אנונימיות מותרות בכל נקודה בקוד שבה ביטוי הוא חוקי. מחלקות אנונימיות מתנהגות כמו מחלקות מקוננות סטטיות או לא סטטיות, תלוי היכן הן מתרחשות: יש להן מופעים שמקננים אותן אם הן מתרחשות בהקשר לא סטטי.

ישנן כמה הגבלות בנוגע לשימוש במחלקות אנונימיות. בגלל שהן מוצהרות וגם נעשות מופע בו-זמנית, אפשר להשתמש במחלקה אנונימית רק אם היא נעשית מופע בנקודה בודדת בקוד. בגלל שלמחלקות אנונימיות אין שם, יש להשתמש בהן רק אם אין שום צורך להתייחס אליהן לאחר שהן נעשות מופע. בדרך כלל, מחלקות אנונימיות מממשות רק שיטות בממשק שלהן או במחלקת-העל שלהן. הן לא מצהירות על שיטה חדשה כלשהי, כיוון שאין שום טיפוס בעל שם שדרכו ניתן יהיה לגשת לשיטות חדשות אלו. בגלל שמחלקות אנונימיות מופיעות בתוך ביטויים, הן צריכות להיות מאוד קצרות, אולי עשרים שורות או פחות. מחלקות אנונימיות ארוכות יותר תפגענה בקריאות התוכנית.

שימוש מקובל אחד של מחלקה אנונימית הוא כדי ליצור **אובייקט כארגומנט לפונקציה**, כמו מופע Comparator. לדוגמה, השיטה הבאה שנקראת, ממיינת מערך מחרוזות בהתאם לאורך שלהן:

```
//Typical use of an anonymous class
Arrays.sort ( args,new Comparator() {
  public int compare(Object o1,Object o2) {
    return ((String)o1).length()-((String)o2).length();
  }
} );
```

שימוש מקובל אחר של מחלקות אנונימיות הוא כדי ליצור **אובייקט תהליך**, כמו Thread, Runnable או מופע TimerTask. שימוש מקובל שלישי הוא מתוך שיטת ייצור סטטית (ראה שיטה intArrayAsList שבנושא 16). שימוש מקובל רביעי הוא במאתחלי השדה הסופי סטטי ציבורי של typesafe enums מתוחכמים אשר דורשים תת-מחלקה נפרדת עבור כל מופע (ראה מחלקה Operation שבנושא 21).

אם מחלקת Operation היא מחלקת איבר סטטית של Calculator, כפי שהומלץ מוקדם יותר, אז קבועי Operation האינדיבידואליים הם מחלקות המקוננות פעמיים :

```
// Typical use of a public static member class
public class Calculator {
  public static abstract class Operation {
    private final String name;

    Operation(String name)   { this.name = name; }

    public String toString() { return this.name; }

    // Perform arithmetic op represented by this constant
    abstract double eval(double x, double y);

    // Doubly nested anonymous classes
    public static final Operation PLUS = new Operation("+") {
      double eval(double x, double y) { return x + y; }
    };
    public static final Operation MINUS = new Operation("-") {
      double eval(double x, double y) { return x - y; }
    };
    public static final Operation TIMES = new Operation("*") {
      double eval(double x, double y) { return x * y; }
    };
    public static final Operation DIVIDE = new Operation("/") {
      double eval(double x, double y) { return x / y; }
    };
  }

  // Return the results of the specified calculation
  public double calculate(double x, Operation op, double y) {
    return op.eval(x, y);
  }
}
```

מחלקות מקומיות הן קרוב לוודאי הכי פחות שימושיות מבין ארבעת הסוגים של מחלקות מקוננות. מחלקה מקומית יכולה להיות מוגדרת בכל מקום שבו אפשר להגדיר משתנה מקומי ומצייתת לאותם חוקי טווח הכרה. למחלקות מקומיות יש כמה תכונות משותפות עם כל אחד משלושת הסוגים האחרים של מחלקות מקוננות. כמו למחלקות מקוננות, יש להן שמות ואפשר להשתמש בהן שוב ושוב. כמו למחלקות אנונימיות, יש להן מופעים שמקננים אותן אם ורק אם נעשה בהן שימוש בהקשר לא סטטי. כמו מחלקות אנונימיות, הן צריכות להיות קצרות כדי לא לפגוע בקריאות השיטה או המאתחל שמקננים אותן.

כדי לסכם ולרענן את הנלמד, יש ארבעה סוגים נפרדים של מחלקות מקוננות ולכל אחד מהם יש את הייחוד שלו. אם מחלקה מקוננת צריכה להיראות מחוץ לשיטה בודדת או שהיא ארוכה מדי מכדי להתאים בצורה מרווחת בתוך שיטה, השתמש במחלקה מקוננת רגילה. אם כל מופע של המחלקה המקוננת צריך ייחוס למופע שמקנן אותו, עשה אותו לא סטטי. אחרת, עשה אותו סטטי. בהנחה שיש צורך במחלקה בתוך שיטה, אם אתה צריך ליצור מופעים ממיקום אחד בלבד ויש טיפוס שהיה קיים קודם אשר מאפיין את המחלקה (ממשק או מחלקת אב), עשה את המחלקה אנונימית. אחרת, עשה אותה מחלקה מקומית.

פרק 5

תחליפים למבני נתונים בשפת C

לשפת התכנות Java יש הרבה במשותף עם שפת התכנות C, אבל כמה **ממבני הנתונים** (constructs) של שפת C הושמטו משפת Java. במרבית המקרים, ברור מדוע הושמטו מבני נתונים אלה וכיצד אפשר להסתדר בלעדיהם. פרק זה גם מציע תחליפים לכמה ממבני הנתונים של שפת C שהושמטו ואשר התחליפים שלהם אינם מובנים מאליהם.

החוט המקשר בין הנושאים שבפרק זה הוא שכל מבני הנתונים שהושמטו הם יותר מוכווני-נתונים (data-oriented) מאשר מוכווני-עצמים (object-oriented). שפת Java מספקת מערכת טיפוסים רבת עוצמה והתחליפים המוצעים מנצלים באופן מרבי מערכת זו, כדי לספק הפשטה באיכות טובה אפילו יותר מאשר מבני שפת C המוחלפים.

אפילו אם החלטת לדלג על פרק זה, מומלץ לקרוא את נושא 21, אשר דן בתבנית (pattern) הנתונים typesafe enum, שהיא תחליף למבנה הנתונים enum של שפת C. נכון לזמן כתיבת שורות אלו, לא רבים מכירים תבנית זו, אך במהרה תראה שיש לה מספר יתרונות על פני השיטות המקובלות הנוכחיות.

נושא 19: החלף מבנים במחלקות

מבנה הנתונים המכונה struct של שפת C הושמט משפת Java מכיון שמחלקה עושה כל מה שעושה המבנה הזה, ואף יותר מכך. מבנה (structure) למעשה מקבץ שדות נתונים מרובים לתוך אובייקט בודד. מחלקה משייכת פעולות לאובייקט המתקבל ומאפשרת לשדות הנתונים להיות מוסתרים (כימוסיות) מהמשתמשים באובייקט. במילים אחרות, מחלקה יכולה **לכמס** (encapsulate) את הנתונים שלה לתוך אובייקט אשר רק לשיטות שלו יש גישה אליו, וכך היא מאפשרת למתכנת את החופש לשנות את הייצוג במהלך הזמן (נושא 12).

מתכנתי שפת C מסוימים אשר נחשפו לראשונה לשפת Java, מאמינים שמחלקות הן יותר מדי מסורבלות מכדי להחליף מבנים בנסיבות מסוימות, אבל אין זה כך.

מחלקות מנוונות אשר מכילות שדות נתונים בלבד שקולות במידה רבה למבנים של שפת C:

```
//Degenerate classes like this should not be public!
class Point {
    public float x;
    public float y;
}
```

מחלקות כאלו נגישות על ידי שדות הנתונים שלהן, הן אינן מציעות את היתרונות של הכימוס (encapsulation). אינך יכול לשנות את הייצוג של מחלקה שכזאת בלי לשנות את ה-API שלה, אינך יכול לאכוף קבוע כלשהו ואינך יכול לנקוט בפעולת עזר כלשהי כאשר שדה עובר שינוי. מתכנתים העוסקים בתכנות מוכוון-עצמים חשים שמחלקות כאלו אינן ראויות ותמיד יש להחליפן במחלקות עם שדות פרטיים **ושיטות גישה** (accessor methods) ציבוריות:

```
//Encapsulated structure class
class Point {
  private float x;
  private float y;

  public Point(float x,float y) {
    this.x =x;
    this.y =y;
  }

  public float getX() {return x;}
  public float getY() {return y;}

  public void setX(float x) {this.x =x;}
  public void setY(float y) {this.y =y;}
}
```

אין ספק שאותם מתכנתים צודקים כאשר הדבר נוגע למחלקות ציבוריות: אם מחלקה נגישה מחוץ לתחום של חבילת התוכנה שלה, מתכנת זהיר יספק שיטות גישה כדי לשמור את הגמישות לשנות את הייצוג הפנימי של המחלקה. אם מחלקה ציבורית היתה צריכה לחשוף את שדות הנתונים, כל תקווה לשינוי הייצוג היתה נגוזה, כיון שקוד לקוח עבור מחלקות ציבוריות יכול להיות מופץ על פני תבל ומלואו.

עם זאת, אם מחלקה שייכת לחבילת תוכנה פרטית או שהיא מחלקה מקוננת פרטית, אין שום פסול בחשיפה ישירה של שדות הנתונים - בהנחה שהם באמת מתארים את ההפשטה אשר מסופקת על ידי המחלקה. תפיסה זו מייצרת פחות ערבוביה חזותית מאשר התפיסה של שיטת גישה, גם בהגדרת המחלקה וגם בקוד הלקוח אשר משתמש במחלקה.

למרות שקוד הלקוח קשור לייצוג הפנימי של המחלקה, קוד זה מוגבל לחבילת התוכנה אשר מכילה את המחלקה. במקרה הנדיר שבו שינוי בייצוג נעשה רצוי, אפשר לבצע את השינוי ללא נגיעה בקוד כלשהו מחוץ לחבילת התוכנה. במקרה של מחלקה מקוננת פרטית, טווח ההכרה של השינוי גם מוגבל למחלקה שמקננת אותה.

כמה מחלקות בספריות הפלטפורמה של Java מפרות את העצה שמחלקות ציבוריות אינן צריכות לחשוף שדות באופן ישיר. דוגמאות בולטות לכך כוללות את מחלקות Point ו-Dimension שבחבילת התוכנה java.awt. יש להתייחס אל מחלקות אלו כאל תמרורי אזהרה, ולא כמודל לחיקוי. כפי שיוסבר בנושא 37, ההחלטה לחשוף את התכונות הפנימיות של המחלקה Dimention מתבטאת בבעיית ביצועים חמורה, אשר אי אפשר לפתור ללא השפעה על תוכניות לקוח.

נושא 20: החלף איחוד (union) בהיררכיות מחלקה

מבנה הנתונים union (איחוד) של שפת C משמש לרוב להגדרת משתנים שיכולים להחזיק יותר מטיפוס אחד של נתונים. מבנה כזה מכיל בדרך כלל לפחות שני שדות: **איחוד** (union) ו**תגית** (tag). התגית היא שדה רגיל אשר משמש לציון סוג הנתונים שנמצאים באיחוד. התגית היא בדרך כלל מטיפוס enum. מבנה שמכיל איחוד ותגית נקרא לעיתים בשם **איחוד מובחן** (discriminated union).

בקוד הדוגמה הבא שכתוב בשפת C, הטיפוס shape_t הוא איגוד מובחן, אשר יכול לשמש לייצוג של עיגול או של מלבן. הפונקציה area לוקחת מצביע אל המבנה shape_t ומחזירה את שטחו, או מחזירה 1- אם המבנה אינו חוקי:

```
/* Discriminated union */
#include "math.h"
typedef enum {RECTANGLE, CIRCLE} shapeType_t;

typedef struct {
   double length;
   double width;
} rectangleDimensions_t;

typedef struct {
   double radius;
} circleDimensions_t;

typedef struct {
   shapeType_t tag;
   union {
      rectangleDimensions_t rectangle;
      circleDimensions_t    circle;
   } dimensions;
} shape_t;

double area(shape_t *shape) {
   switch(shape->tag) {
     case RECTANGLE: {
       double length = shape->dimensions.rectangle.length;
       double width  = shape->dimensions.rectangle.width;
       return length * width;
     }
     case CIRCLE: {
       double r = shape->dimensions.circle.radius;
       return M_PI * (r*r);
     }
     default: return -1.0; /* Invalid tag */
   }
}
```

המתכננים של שפת Java בחרו להשמיט את המושג union מכיון שיש מנגנון טוב יותר בהרבה להגדרת טיפוס נתונים בודד, אשר יכול לייצג אובייקטים מסוגים (טיפוסים) שונים: תת-סוגים. איגוד מובדל הוא למעשה חיקוי חיוור של היררכיית מחלקה.

כדי להפוך איגוד מובדל להיררכיית מחלקה, הגדר מחלקה מופשטת שמכילה שיטה מופשטת עבור כל פעולה אשר ההתנהגות שלה תלויה בערך של התווית. בדוגמה הקודמת, יש רק פעולה אחת כזו, area. מחלקה מופשטת זאת היא השורש של היררכיית מחלקה. אם יש פעולות אשר ההתנהגות שלהן לא תלויה בערך של התווית, הפוך שיטות אלו לשיטות לא מופשטות במחלקת השורש. באותו אופן, אם ישנם שדות נתונים כלשהם באיגוד המובדל נוסף על התווית והאיגוד, שדות אלה מייצגים נתונים אשר משותפים לכל הטיפוסים וצריך להוסיף אותם למחלקת השורש.

לאחר מכן, הגדר תת-מחלקה לא מופשטת של מחלקת השורש עבור כל טיפוס אשר יכול להיות מיוצג על ידי האיגוד המובדל. בדוגמה הקודמת, הטיפוסים הם עיגול ומלבן. כלול בכל תת-מחלקה את שדות הנתונים שמיוחדים לטיפוס שלה. בדוגמה, רדיוס מיוחד לעיגול, ואילו אורך ורוחב מיוחדים למלבן. כלול גם בכל תת-מחלקה את המימוש המתאים של כל שיטה מופשטת במחלקת השורש. הנה היררכיית המחלקה שמתאימה לדוגמת האיגוד המובדל:

```
abstract class Shape {
   abstract double area();
}

class Circle extends Shape {
   final double radius;

   Circle(double radius) { this.radius = radius; }

   double area() { return Math.PI * radius*radius; }
}

class Rectangle extends Shape {
   final double length;
   final double width;

   Rectangle(double length, double width) {
      this.length = length;
      this.width  = width;
   }

   double area() { return length * width; }
}
```

להיררכיית מחלקה יש הרבה יתרונות על פני איגוד מובדל. היתרון הראשי הוא שהיררכיית מחלקה מספקת בטיחות טיפוס. בדוגמה, כל מופע Shape הוא Circle חוקי או Rectangle חוקי. לעומת זאת, אין זה קשה ליצור מבנה shape_t חסר תועלת ומסוכן, מכיון ששפת C אינה דואגת שהחיבור בין התווית לנתוני האיגוד יאכפו כדרוש. אם לדוגמה, התווית מראה שה-shape_t מייצג מלבן אבל האיגוד הוגדר כעיגול, צפה לבעיות חמורות. אפילו אם האיגוד המובדל אותחל כראוי, עדיין ניתן להעביר אותו בטעות לפונקציה אשר מצפה לקבל איגוד עם תווית אחרת, והדבר לא יתגלה בזמן ההידור ויגרום לבעיות בזמן הריצה.

יתרון שני של היררכיית מחלקה הוא שהקוד פשוט וברור. האיגוד המובדל מבולבל ומסורבל: הגדרת טיפוס ה-enum, הגדרת שדה התווית, מיתוג של שדה התווית, טיפול בערכי תווית לא צפויים וכדומה. הקוד של האיגוד המובדל נעשה אפילו עוד פחות קריא בשל העובדה שהפעולות עבור הטיפוסים השונים מעורבבות ולא מופרדות על פי הטיפוס.

יתרון שלישי של היררכיית מחלקה הוא שניתן להרחיב אותה בקלות. כדי להרחיב היררכיית מחלקה, פשוט הוסף תת-מחלקה חדשה. אם אתה שוכח לדרוס אחת מהשיטות המופשטות שבמחלקת-העל, המהדר ידווח לך על כך חד וחלק. כדי להרחיב איגוד מובדל, אתה צריך לגשת לתוך קוד המקור. אתה חייב להוסיף ערך חדש לטיפוס ה-enum וגם מקרה (case) חדש לביטוי Switch, בכל פעולה על האיגוד המובדל. לבסוף, אתה חייב להדר מחדש. אם אתה שוכח להוסיף מקרה (case) חדש עבור שיטה מסוימת, אתה לא תגלה זאת אלא בזמן ריצה ובתנאי שאתה זהיר מספיק כדי לחפש ערכי תווית לא מזוהים ולחולל הודעת שגיאה מתאימה.

יתרון רביעי של היררכיית מחלקה הוא שהיא יכולה לשקף יחסי היררכיה טבעיים בין טיפוסים, דבר שמאפשר גמישות מוגברת ובדיקת טיפוס טובה יותר בזמן ההידור. נניח שהאיגוד המובדל שבדוגמה המקורית מורחב בכדי לקבל גם ריבועים. אפשר להביא את היררכיית המחלקה לידי כך שתשקף את העובדה שריבוע הוא סוג מיוחד של מלבן (בהנחה ששניהם בלתי משתנים):

```
class Square extends Rectangle {
   Square(double side) {
      super(side, side);
   }

   double side() {
      return length; // or equivalently, width
   }
}
```

היררכיית המחלקה שבדוגמה זו אינה היחידה אשר יכולה היתה להיכתב כדי להחליף את האיגוד המובדל. ההיררכיה ממחישה כמה החלטות תכנון שראויות לציון.
המחלקות שבהיררכיה, חוץ מ-Square, נגישות על ידי שדותיהן ולא על ידי שיטות גישה. הדבר נעשה לשם קיצור ולא היה מקובל אם המחלקות היו ציבוריות (נושא 19).
המחלקות הן בלתי משתנות, דבר שלא תמיד מתאים, אבל בדרך כלל זהו דבר טוב (נושא 13).

מכיון ששפת Java לא מספקת את המושג union, אתה עשוי לחשוב שאין שום סכנה במימוש איגוד מובדל, אבל אפשר לכתוב קוד עם הרבה מחסרונות אלה. בכל פעם שאתה מתפתה לכתוב מחלקה עם שדה תווית מפורש, חשוב האם אפשר להסיר את התווית ולהחליף את המחלקה בהיררכיית מחלקה.

שימוש נוסף במושג union של שפת C, לגמרי לא קשור לאיגוד מובדל. שימוש זה מערב צפייה בייצוג הפנימי של פיסת נתונים, תוך הפרה מכוונת של מערכת הטיפוס. שימוש זה מודגם על ידי רסיס הקוד הבא בשפת C, אשר מדפיס את הייצוג ההקסדצימלי כפי שהוא מיוצג על ידי המחשב, של float:

```
union {
   float f;
   int bits;
} sleaze;

sleaze.f =6.699e-41;        /*Put data in one field of union...*/
printf("%x \n",sleaze.bits);          /*...and read it out the other.*/
```

למרות שזה יכול להיות שימושי, במיוחד עבור תכנות מערכת, לשימוש לא נייד זה אין שום מקביל בשפת Java. למעשה, הדבר מנוגד לחלוטין לרוח של שפת Java, אשר מבטיחה בטיחות טיפוס ועושה מאמצים רבים כדי להפריד את המתכנתים מייצוגי נתונים פנימיים של המחשב.

חבילת התוכנה java.lang אכן מכילה שיטות לתרגום מספרי נקודה צפה לייצוגי סיביות, אבל שיטות אלו מוגדרות במונחים של ייצוג סיביות מוגדר במדויק כדי להבטיח ניידות (portability). רסיס הקוד הבא, אשר מקביל במידת מה לרסיס הקוד הקודם בשפת C, מבטיח הדפסה של אותה תוצאה, ללא תלות במחשב שבו הוא רץ:

```
System.out.println(
Integer.toHexString(Float.floatToIntBits(6.699e-41f)));
```

נושא 21: החלף את מושגי enum במחלקות

המושג enum של שפת C הושמט משפת Java. באופן נומינלי, מושג זה מגדיר **טיפוס נספר**: טיפוס אשר ערכיו החוקיים מורכבים מסט קבוע של קבועים. לרוע המזל, המושג enum לא עושה עבודה טובה מאוד בהגדרת טיפוסים נספרים. הוא רק מגדיר סט של קבועים שלמים בעלי שם, לא מספק שום דבר בנוגע לבטיחות טיפוס ומעט בנוגע לנוחיות. לא רק שקטע הקוד הבא בשפת C הוא חוקי,

```
typedef enum {FUJI,PIPPIN,GRANNY_SMITH} apple_t;
typedef enum {NAVEL,TEMPLE,BLOOD} orange_t;
orange_t myFavorite =PIPPIN;     /*Mixing apples and oranges */
```

אלא גם האכזריות הזאת:

```
orange_t  x = (FUJI - PIPPIN)/TEMPLE;              /* Applesauce! */
```

המושג enum לא מייסד מרחב שמות עבור הקבועים שהוא יוצר. לכן, ההגדרה הבאה, אשר משתמשת מחדש באחד מהשמות, מתנגשת עם ההגדרה של orange_t:

```
typedef enum {BLOOD, SWEAT, TEARS} fluid_t;
```

טיפוסים אשר מוגדרים בעזרת המושג enum הם רגישים. הוספת קבועים לטיפוס שכזה בלי להדר את הלקוחות המשתמשים בו גורמת להתנהגות בלתי צפויה, אלא אם כן מוקדשת תשומת לב כדי לשמור את כל ערכי הקבועים שהיו קיימים קודם לכן. המושג enum לא מספק אף דרך קלה לתרגום קבועים נספרים למחרוזות ברות-הדפסה, או לבצע ספירה על פני הקבועים שבטיפוס.

לרוע המזל, התבנית שנעשה בה השימוש השכיח ביותר עבור טיפוסים נספרים בשפת Java, כפי שהיא מוצגת להלן, חולקת עם שפת C את המגרעות של המושג enum:

```
//The int enum pattern - problematic!!
public class PlayingCard {
public static final int SUIT_CLUBS =0;
public static final int SUIT_DIAMONDS =1;
public static final int SUIT_HEARTS =2;
public static final int SUIT_SPADES =3;
...
}
```

אתה עשוי לפגוש וריאציה שונה של תבנית זו במקרה שבו קבועי String משמשים במקום קבועי int. אין להשתמש בווריאציה זו אף פעם. למרות שהיא אכן מספקת מחרוזות ברות-הדפסה עבור הקבועים שלה, היא עלולה להוביל לבעיות ביצועים בגלל שהיא מסתמכת על השוואות מחרוזת. יתרה מזאת, היא עלולה להוביל משתמשים נאיביים לשתול בקוד ישירות את קבועי המחרוזת במקום להשתמש בשמות השדה המתאימים. אם קבוע מחרוזת שכזה מכיל שגיאה טיפוגרפית (כדוגמת שגיאת כתיב), השגיאה לא תתגלה בבדיקה בזמן הידור אבל תגרום לבאגים בזמן ריצה.

למרבה המזל, שפת Java מציגה חלופה אשר נמנעת מכל המגרעות של תבניות String ו-int המקובלות ומספקת יתרונות רבים נוספים. הדבר קרוי תבנית typesafe enum. לרוע המזל, עדיין אין היא נפוצה כל כך. הרעיון הבסיסי הוא פשוט: הגדר מחלקה אשר מייצגת אלמנט בודד של הטיפוס הנספר ואל תספק שום בנאי ציבורי. במקום בנאים ציבוריים, ספק שדות סופיים סטטיים ציבוריים, שדה אחד עבור כל קבוע שבטיפוס הנספר. לפניך התבנית בצורתה הפשוטה ביותר:

```
//The typesafe enum pattern
public class Suit {
  private final String name;

  private Suit(String name) {this.name =name;}

  public String toString() {retu n name;}

  public static final Suit CLUBS = new Suit("clubs");
  public static final Suit DIAMONDS = new Suit("diamonds");
  public static final Suit HEARTS = new Suit("hearts");
  public static final Suit SPADES = new Suit("spades");
}
```

בגלל שאין שום דרך שבה הלקוחות יכולים ליצור אובייקטים של המחלקה או להרחיב את המחלקה, לעולם לא יהיו אף אובייקטים של הטיפוס חוץ מאלה שמיוצאים דרך השדות הסופיים סטטיים ציבוריים. למרות שהמחלקה לא הוגדרה כסופית, אין שום דרך להרחיב אותה (בנאים של תת-מחלקה חייבים לקרוא לבנאי של מחלקת-על ואין בנאי נגיש שכזה).

כמו ששמה מרמז, התבנית typesafe enum מספקת בטיחות טיפוס של זמן הידור. אם אתה מצהיר על שיטה המקבלת פרמטר מטיפוס Suit, מובטח לך שלמעט null היא תוכל לקבל רק אחד מארבע ה-suits החוקיים. כל ניסיון להעביר אובייקט מטיפוס לא נכון יתגלה בזמן הידור, וכך גם לגבי ניסיון לבצע השמה של אוביקט suit בתוך ייחוס מטיפוס אחר. מחלקות typesafe enum מרובות עם קבועי ספירה שווי שם מתקיימות ביחד בלי שום בעיה, מכיון שלכל מחלקה יש את מרחב השמות שלה.

אפשר להוסיף קבועים למחלקת typesafe enum ללא הידור מחדש של הלקוחות שלה בגלל ששדות ייחוסי האובייקט הסטטי ציבורי, אשר מכילים את קבועי הספירה, מספקים שכבה של בידוד בין הלקוח ובין מחלקת ה-enum. הקבועים עצמם לעולם לא מהודרים לתוך לקוחות כפי שהיה בתבנית ה-int המקובלת יותר או בגרסת ה-String שלה.

בגלל שמחלקות typesafe enum הן מחלקות במלוא מובן המילה, אתה יכול לדרוס את שיטת toString כפי שהודגם לפני כן, תוך שאתה מאפשר לערכים להיות מתורגמים למחרוזות ברות-הדפסה. אתה יכול, אם רצונך בכך, ללכת צעד אחד קדימה ולהפוך אותה בקלות ל- typesafe enums בינלאומי. שים לב לכך שבמחרוזות נעשה שימוש רק על ידי שיטת toString. הם לא משמשים להשוואות שוויון מכיון שמימוש equals, אשר הורש מ-Object, מבצע השוואת זהות ייחוס.

באופן יותר כללי, אתה יכול להגדיל מחלקת typesafe enum עם כל שיטה אשר נראית מתאימה. לדוגמה, מחלקת Suit שלנו עשויה להרוויח כתוצאה מהוספת שיטה אשר מחזירה את הצבע של חבילת הקלפים, או שיטה אשר מחזירה תמונה אשר מייצגת את החבילה. מחלקה יכולה להתחיל את חייה כ- typesafe enum פשוט ולהתפתח במשך הזמן.

בגלל ששיטות שרירותיות יכולות להתווסף למחלקות typesafe enum, אפשר להשתמש בהן כדי לממש כל ממשק. לדוגמה, נניח שאתה רוצה ש-Suit תממש את Comparable כך שלקוחות יוכלו למיין ידי ברידג' על ידי suit. הנה גירסה מעט שונה של התבנית המקורית אשר מבצעת את המבצע הזה. משתנה סטטי בשם nextOrdinal משמש להשמת מספר סידורי לכל מופע כאשר הוא נוצר. מספרים סידוריים אלה משמשים את שיטת compareTo כדי לסדר מופעים:

```
// Ordinal-based typesafe enum
public class Suit implements Comparable {
   private final String name;

   // Ordinal of next suit to be created
   private static int nextOrdinal = 0;

   // Assign an ordinal to this suit
   private final int ordinal = nextOrdinal++;

   private Suit(String name) { this.name = name; }

   public String toString()  { return name; }

   public int compareTo(Object o) {
      return ordinal - ((Suit)o).ordinal;
   }
```

```
    public static final Suit CLUBS    = new Suit("clubs");
    public static final Suit DIAMONDS = new Suit("diamonds");
    public static final Suit HEARTS   = new Suit("hearts");
    public static final Suit SPADES   = new Suit("spades");
}
```

בגלל שקבועי typesafe enum הם אובייקטים, אתה יכול להכניס אותם לאוספים. לדוגמה, נניח שאתה רוצה שמחלקת Suit תייצא רשימה שאינה ניתנת לשינוי של חבילות קלפים בסדר סטנדרטי. פשוט הוסף שתי הצהרות שדה אלו למחלקה:

```
private static final Suit[] PRIVATE_VALUES =
        { CLUBS, DIAMONDS, HEARTS, SPADES };
    public static final List VALUES =
        Collections.unmodifiableList(Arrays.asList(PRIVATE_VALUES));
```

שלא כמו הצורה הפשוטה של תבנית typesafe enum, מחלקות מהצורה המבוססת על הדוגמה שלמעלה, דורשות מעט טיפול כדי להפוך ל-serializable. אין זה די מספיק להוסיף implements Serializable להצהרת המחלקה, אתה חייב גם לספק שיטת readResolve:

```
private Object readResolve() throws ObjectStreamException {
    return PRIVATE_VALUES [ordinal ] ;   // Canonicalize
}
```

שיטה זו, אשר נקראת באופן אוטומטי על ידי מערכת הרישום לצורך שליחה בזרם, מונעת קיום של קבועים כפולים כתוצאה מפירוק הרישום בקריאה מהזרם. הדבר שומר על הערבות שרק אובייקט בודד מייצג כל קבוע enum, תוך שהוא נמנע מהצורך לדרוס את Object.equals. ללא ערבות זאת, Object.equals תדווח על חוסר שוויון שקרי כאשר היא מוצגת על ידי שני קבועי ספירה שווים אבל נפרדים. שים לב ששיטת readResolve מתייחסת למערך PRIVATE_VALUES, כך שאתה חייב להצהיר על מערך זה אפילו אם אתה בוחר שלא לייצא את VALUES. שים לב גם שהשיטה readResolve לא משתמשת בשדה name, כך שהוא יכול וצריך להיות מוגדר כארעי (בעזרת המאפיין transient).

המחלקה המתקבלת היא מעט שבירה. בנאים עבור ערכים חדשים כלשהם חייבים להופיע אחרי כל הבנאים של הערכים הקיימים, כדי להבטיח שמופעים אשר נרשמו בזרם לפני כן לא ישנו את ערכיהם לאחר שהם ייקראו מהזרם ויורכבו חזרה. זאת מפני שהצורה הסדרתית (Serializable) (נושא 55) של קבוע ספירה (כפי שהוא נרשם בזרם) מורכבת מהמספר הסידורי שלו בלבד.

ניתן שיהיו התנהגויות המקושרות עם כל קבוע ומשמשות רק מתוך חבילת התוכנה שמחילה את מחלקת ה- typesafe enum.

התנהגויות אלו ממומשות באופן הטוב ביותר כשיטות בעלות הרשאה פרטית-בחבילה (הרשאת ברירת המחדל). כך, כל קבוע enum נושא עמו אוסף חבוי של התנהגויות אשר מאפשר לחבילת התוכנה שמכילה את הטיפוס הנספר להגיב כהלכה, כאשר הוא מוצג עם הקבוע.

אם למחלקת typesafe enum יש שיטות אשר ההתנהגות שלהן ניתנת לשינוי באופן משמעותי מקבוע מחלקה אחד לאחר, אתה צריך להשתמש במחלקה פרטית נפרדת או במחלקה פנימית אנונימית עבור כל קבוע. הדבר מאפשר לכל קבוע להחזיק במימוש שלו של כל שיטה שכזו, והמחלקה קוראת באופן אוטומטי למימוש המתאים. החלופה היא לבנות כל שיטה שכזאת כהסתעפות מרובת דרכים אשר מתנהגת באופן שונה, בהתאם לקבוע שביחס אליו היא נקראת. החלופה הזאת היא מגעילה, נוטה להניב שגיאות ודי סביר שהיא תספק ביצועים ירודים יחסית לאלה של הרצת השיטות האוטומטית של המכונה הווירטואלית.

שתי הטכניקות אשר תוארו בפיסקה הקודמת מודגמות במחלקת ה- typesafe enum הבאה. המחלקה Operation מייצגת פעולה אשר מבוצעת על ידי מחשבון ארבע פעולות בסיסי. מחוץ לחבילת התוכנה שבה המחלקה מוגדרת, כל שאתה יכול לעשות עם קבוע של Operation הוא לקרוא לשיטות של Object (toString, hashCode, equals וכן הלאה). בכל אופן, בתוך חבילת התוכנה, אתה יכול לבצע את הפעולה החשבונית אשר מיוצגת על ידי הקבוע. כפי הנראה, חבילת התוכנה תייצא כמה אובייקטי מחשבון מרמה גבוהה יותר אשר ייצאו שיטה אחת או יותר, אשר לקחה קבוע Operation כפרמטר. שים לב ש-Operation עצמה היא מחלקה מופשטת, שמכילה שיטה פרטית-בחבילה מופשטת יחידה, בשם eval, אשר מבצעת את הפעולה החשבונית המתאימה. מחלקה פנימית אנונימית מוגדרת עבור כל קבוע, כך שכל קבוע יכול להגדיר את הגירסה שלו של שיטת eval:

```
//Typesafe enum with behaviors attached to constants
public abstract class Operation {
  private final String name;

  Operation(String name) {this.name =name;}
  public String toString() {return this.name;}

  //Perform a ithmetic operation represented by this constant
  abstract double eval(double x,double y);
  public static final Operation PLUS =new Operation("+") {
    double eval(double x,double y){return x +y;}
  };

  public static final Operation MINUS =new Operation("-") {
    double eval(double x,double y){return x -y;}
  };
```

```
  public static final Operation TIMES =new Operation("*") {
    double eval(double x,double y){return x *y;}
  };
  public static final Operation DIVIDED_BY = new Operation("/") {
      double eval(double x,double y){return x /y;}
  };
}
```

בדרך כלל ל- typesafe enums יש ביצועים דומים לקבועי ספירה של int. שני מופעים נפרדים של מחלקת typesafe enum לא יכולים לעולם לייצג את אותו הערך, לכן השוואות זהות ייחוס (בעזרת האופרטור ==) שהן מהירות, יכולות לשמש לבדיקת שוויון לוגי (במקום equals).

אם מחלקת typesafe enum מסוימת היא שימושית באופן נרחב, היא צריכה להיות מחלקה מהרמה העליונה. אם השימוש שלה קשור למחלקה ספציפית מהרמה העליונה, היא צריכה להיות מחלקת איבר סטטית של המחלקה מהרמה הזאת מהרמה העליונה (נושא 18). לדוגמה, המחלקה java.math.BigDecimal מכילה אוסף של קבועי ספירת int, אשר מייצגים מצבי עיגול (של מספרים) עבור שברים דצימליים. מצבי עיגול אלה מספקים הפשטה שימושית אשר לא קשורה מיסודה למחלקת BigDecimal. היה עדיף אם הם היו ממומשים כמחלקת java.math.RoundingMode חופשית. דבר זה היה מעודד כל מתכנת אשר היה צריך מצבי עיגול להשתמש שוב במצבי עיגול אלה, דבר שהיה מוביל לעקביות גדולה יותר של APIs.

התבנית הבסיסית של typesafe enum, כפי שהודגמה על ידי שני מימושי Suit שהוצגו לפני כן, היא **בלתי ניתנת להרחבה**: בלתי אפשרי למשתמשים להוסיף אלמנטים חדשים לטיפוס הנספר, מכיון שלמחלקה שלו אין שום בנאים שלמשתמש יכולה להיות גישה אליהם. למעשה, הדבר עושה את המחלקה סופית, בין אם היא הוגדרה או לא הוגדרה עם מגדיר הגישה final. בדרך כלל זה מה שאתה רוצה, אבל לפעמים אתה עשוי לרצות לעשות מחלקת typesafe enum כך שתהיה **בת-הרחבה**. לדוגמה, זה מה שהיית רוצה לעשות אם היית משתמש ב- typesafe enum כדי להציג פורמטים של קידוד תמונה והיית רוצה שצד שלישי יהיה מסוגל להוסיף תמיכה בפורמטים חדשים נוספים.

כדי לעשות typesafe enum בת-הרחבה, פשוט ספק בנאי מוגן. כך, אחרים יוכלו להרחיב את המחלקה ולהוסיף קבועים חדשים לתת-המחלקות שלהם. אתה לא צריך לדאוג בנוגע להתנגשויות קבוע ספירה כפי שהיית צריך לדאוג אם היית משתמש בתבנית enum של int. המשתנה בר-ההרחבה של תבנית typesafe enum מנצל את מרחב השם של חבילת התוכנה כדי ליצור מרחב שם "שסופק באורח פלא" עבור הספירה בת-ההרחבה. ארגונים כפולים יכולים להרחיב את הספירה ללא ידיעה של האחד על השני, וההרחבות שלהם אף פעם לא תתנגשנה.

הוספת אלמנט לטיפוס נספר בר-הרחבה אינה מספיקה בשביל להבטיח שהאלמנט החדש נתמך באופן מלא: שיטות אשר לוקחות אלמנט של הטיפוס הנספר חייבות להתמודד עם האפשרות של קבלת אלמנט שאינו ידוע למתכנת. הסתעפויות מרובות על טיפוסים נספרים בלתי ניתנים להרחבה הן מפוקפקות. לגבי טיפוסים נספרים ברי-הרחבה הן קטלניות, מכיון שלא ייווצר בדרך נס ענף חדש להסתעפות בכל פעם שמתכנת מרחיב את הטיפוס.

דרך אחת להתמודד עם בעיה זו היא לצייד את מחלקת ה- typesafe enum בכל השיטות הדרושות לתאר את ההתנהגות של קבוע של המחלקה. שיטות אשר אינן שימושיות עבור לקוחות של המחלקה צריכות להיות מוגנות כדי להסתיר אותן מלקוחות, למרות שיש לאפשר לתת-מחלקות לדרוס אותן. אם לשיטה שכזאת אין מימוש ברירת מחדל הגיוני, היא צריכה להיות מופשטת וגם מוגנת.

זה רעיון טוב שמחלקות typesafe enum ברות-הרחבה ידרסו את שיטות equals ו-hashCode עם שיטות סופיות אשר קוראות לשיטות Object. הדבר מבטיח שאף תת-מחלקה לא תדרוס בטעות שיטות אלו, תוך שמירה על הערבות שכל האובייקטים השווים של הטיפוס הנספר הם גם זהים (a.equals(b) if and only if a==b):

```
//Override-prevention methods
public final boolean equals(Object that){
   return super.equals(that);
}
public final int hashCode()  {
   return super.hashCode();
}
```

שים לב שהגירסה בת-ההרחבה של תבנית ה- typesafe enum, אינה תואמת לגירסה בת-ההשוואה. אם ניסית לשלב אותם, הסידור בין האלמנטים של תת-המחלקות יהיה תלוי בסדר שבו תת-המחלקות אותחלו (אשר יכול להשתנות מתוכנית לתוכנית ומהרצה להרצה).

הגירסה בת-ההרחבה של תבנית ה- typesafe enum תואמת לגירסה שניתן להעביר בזרם (Serializable), אבל צירוף גרסאות אלו דורש מעט תשומת לב. כל תת-מחלקה חייבת לבצע השמה של המספרים הסידוריים שלה ולספק את שיטת readResolve שלה. בקיצור, כל מחלקה אחראית ל-serialization ול-DeSerialization של המופעים שלה.

כדי להבהיר זאת, הנה גירסה של מחלקת Operation אשר עברה שינוי כך שתהיה גם בת-הרחבה וגם בת-העברה בזרם:

```
// Serializable, extensible typesafe enum
public abstract class Operation implements Serializable {
   private final transient String name;
   protected Operation(String name) { this.name = name; }

   public static Operation PLUS = new Operation("+") {
      protected double eval(double x, double y) { return x+y; }
   };
   public static Operation MINUS = new Operation("-") {
      protected double eval(double x, double y) { return x-y; }
   };
   public static Operation TIMES = new Operation("*") {
      protected double eval(double x, double y) { return x*y; }
   };
   public static Operation DIVIDE = new Operation("/") {
      protected double eval(double x, double y) { return x/y; }
   };

   // Perform arithmetic operation represented by this constant
   protected abstract double eval(double x, double y);

   public String toString() { return this.name; }
   // Prevent subclasses from overriding Object.equals
   public final boolean equals(Object that) {
      return super.equals(that);
   }
   public final int hashCode() {
      return super.hashCode();
   }

   // The 4 declarations below are necessary for serialization
   private static int nextOrdinal = 0;
   private final int ordinal = nextOrdinal++;
   private static final Operation[] VALUES = { PLUS, MINUS, TIMES, DIVIDE };
   Object readResolve() throws ObjectStreamException {
      return VALUES[ordinal];  // Canonicalize
   }
}
```

הנה תת-מחלקה של Operation אשר מוסיפה פעולות לוגריתמיות ומעריכיות.
תת-מחלקה זו יכולה היתה להתקיים מחוץ לחבילת התוכנה שמכילה את מחלקת הפעולה שעברה תיקון.

היא היתה יכולה להיות ציבורית והיתה יכולה להיות בת-הרחבה בעצמה.
תת-מחלקות עצמאיות מרובות יכולות להתקיים זו בצד זו בשלום ובשלווה:

```
// Subclass of extensible, serializable typesafe enum
abstract class ExtendedOperation extends Operation {
   ExtendedOperation(String name) { super(name); }

   public static Operation LOG = new ExtendedOperation("log") {
      protected double eval(double x, double y) {
         return Math.log(y) / Math.log(x);
      }
   };
   public static Operation EXP = new ExtendedOperation("exp") {
      protected double eval(double x, double y) {
         return Math.pow(x, y);
      }
   };

   // The 4 declarations below are necessary for serialization
   private static int nextOrdinal = 0;
   private final int ordinal = nextOrdinal++;
   private static final Operation[] VALUES = { LOG, EXP };
   Object readResolve() throws ObjectStreamException {
      return VALUES[ordinal];  // Canonicalize
   }
}
```

שים לב ששיטות readResolve שבמחלקות שזה עתה הוצגו הן בעלות הרשאה פרטית-בחבילה ולא הרשאה פרטית. הדבר הכרחי מכיון שהמופעים של Operation ו-ExtendedOperation הם, למעשה, מופעים של מחלקות אנונימיות, לכן לשיטות readResolve פרטיות לא תהיה שום השפעה (נושא 57).

לתבנית typesafe enum יש חסרונות מועטים בהשוואה לתבנית ה-int. אולי החיסרון היחיד הרציני הוא הסרבול שבהכנסת קבועי typesafe enum לתוך סטים (sets). בנוגע ל-enums מבוססי int, הדבר נעשה באופן מסורתי על ידי בחירת ערכי קבוע ספירה, כל אחד מהם הוא חזקה חיובית נפרדת של שתיים והייצוג של סט מתבצע בעזרת הפעולה הלוגית OR על הקבועים הרלוונטיים:

```
//Bit-flag variant of int enum pattern
public static final int SUIT_CLUBS =1;
public static final int SUIT_DIAMONDS =2;
public static final int SUIT_HEARTS =4;
public static final int SUIT_SPADES =8;

public static final int SUIT_BLACK =SUIT_CLUBS |SUIT_SPADES;
```

ייצוג סטים של קבועי טיפוס נספרים באופן הזה הוא תמציתי ומהיר ביותר. עבור סטים של קבועי typesafe enum, אתה יכול להשתמש במימוש סט רב-תכליתי מסביבת העבודה של האוספים, אבל הוא לא תמציתי וגם לא מהיר:

```
Set blackSuits = new HashSet();
blackSuits.add(Suit.CLUBS);
blackSuits.add(Suit.SPADES);
```

למרות שקרוב לוודאי אי אפשר לעשות סטים של קבועי typesafe enum תמציתיים ומהירים כמו סטים של קבועי int enum, אפשר להפחית את ההבדל על ידי אספקת מימוש סט מיוחד אשר מקבל רק אלמנטים מטיפוס אחד ומייצג את הסט באופן פנימי כווקטור של סיביות. הדרך הטובה ביותר לממש סט שכזה היא באותה חבילת תוכנה כמו זו של טיפוס האלמנטים שלו כדי לאפשר גישה, באמצעות שדה או שיטה פרטית-בחבילה, לערך סיביות שמקושר באופן פנימי עם כל קבוע typesafe enum. אספקת בנאים ציבוריים אשר לוקחים רצף קצר של אלמנטים כפרמטרים מתקבלת על הדעת, כך ששורות קוד שכאלו הן אפשריות:

```
hand.discard(new SuitSet(Suit.CLUBS, Suit.SPADES);
```

חיסרון קטן של typesafe enums, בהשוואה ל- int enums, הוא ש- typesafe enums לא יכולים לשמש במשפטי switch מכיון שהם אינם קבועים אינטגרליים. במקום זאת, אתה משתמש במשפט if, כמו זה:

```
if (suit ==Suit.CLUBS)  {
   ...
} else if (suit ==Suit.DIAMONDS) {
   ...
} else if (suit ==Suit.HEARTS) {
   ...
} else if (suit ==Suit.SPADES) {
   ...
} else {
   throw new NullPointerException("Null Suit");//suit == null
}
```

משפט ה-if אינו יעיל כמו משפט ה-switch, אבל לעיתים נדירות ההבדל משמעותי. יתר על כן, הצורך בהסתעפויות רבות דרכים על קבועי typesafe enums צריך להיות נדיר, מכיון שהם נוחים להרצת שיטה אוטומטית על ידי ה-JVM, כמו בדוגמת Operator.

חיסרון קטן אחר של typesafe enums הנוגע לביצועים הוא שיש עלות של זמן ומקום בטעינת מחלקות מטיפוס enum ובבניית האובייקטים הקבועים. פרט לסביבות מוגבלות משאבים כמו טלפונים סלולריים ומצנמים, בעיה זו כמעט ולא באה לידי ביטוי בפועל.

לסיכום, היתרונות של typesafe enums על פני int enums הם גדולים ואף אחד מהחסרונות אינו משכנע אלא אם כן טיפוס נספר משמש בעיקר כאלמנט סט או בסביבה מוגבלת משאבים באופן קיצוני. לכן, **תבנית typesafe enum צריכה להיות הדבר שעולה במחשבה כאשר הנסיבות מצביעות על טיפוס נספר**. APIs אשר משתמשים ב- typesafe enums הם הרבה יותר ידידותיים למתכנת מאשר אלה המשתמשים ב- int enums. הסיבה היחידה שבגללה typesafe enums לא נפוצות באופן נרחב יותר ב-APIs של פלטפורמת Java היא שתבנית typesafe enum לא היתה ידועה בזמן שהרבה מ-APIs אלה נכתבו. לבסוף, כדאי להדגיש שוב שהצורך בטיפוסים נספרים מכל סוג שהוא צריך להיות נדיר יחסית, מכיון ששימוש עיקרי של טיפוסים אלה התיישן עקב תת-מחלקתיות (נושא 20).

נושא 22: החלף מצביעי פונקציות במחלקות וממשקים

שפת C תומכת **במצביעי פונקציות**, אשר מאפשרים לתוכנית לאחסן ולהעביר את היכולת לקרוא לפונקציה מסוימת. במצביעי פונקציה נעשה שימוש בדרך כלל כדי לאפשר למי שקורא לפונקציה לייחד את ההתנהגות שלה על ידי העברת מצביע לפונקציה שנייה, אשר לעיתים מיוחסת **כהתקשרות חוזרת**. לדוגמה, פונקציית qsort שבספריה הסטנדרטית של שפת C, מקבלת מצביע לפונקציית **השוואה**, אשר משמשת אותה כדי להשוות את האלמנטים שיש למיין. פונקציית ההשוואה מקבלת שני פרמטרים, כל אחד מהם הוא מצביע לאלמנט. היא מחזירה שלם שלילי אם האלמנט שמוצבע על ידי הפרמטר הראשון הוא קטן מאשר האלמנט שמוצבע על ידי הפרמטר השני. היא מחזירה אפס, אם שני האלמנטים שווים ושלם חיובי אם האלמנט שמוצבע על ידי הפרמטר הראשון גדול מהאלמנט שמוצבע על ידי האלמנט השני. אפשר לקבל סדרי מיון שונים על ידי העברת פונקציות השוואה שונות. זוהי דוגמה לתבנית **האסטרטגיה** [Gamma98, עמוד 315]. פונקצייתהשוואה מייצגת אסטרטגיה למיון אלמנטים.

מצביעי פונקציה הושמטו משפת Java בגלל שייחוסי אובייקטים יכולים לשמש לאותה המטרה. קריאה של שיטה על אובייקט, באופן טיפוסי, מבצעת אי אלו פעולות על **האובייקט הזה**. בכל אופן, אפשר להגדיר אובייקט שהשיטות שלו מבצעות פעולות על **אובייקטים אחרים**, אשר הועברו במפורש לשיטות. מופע של מחלקה אשר מייצאת בדיוק שיטה אחת שכזאת היא מצביע יעיל לשיטה הזאת. מופעים כאלה ידועים **כאובייקטי פונקציה**. לדוגמה, חשוב על המחלקה הבאה:

```
class StringLengthComparator {
   public int compare(String s1,St ing s2) {
      return s1.length() - s2.length();
   }
}
```

מחלקה זו מייצאת שיטה בודדת אשר מקבלת שתי מחרוזות ומחזירה שלם שלילי אם המחרוזת הראשונה היא קצרה יותר מהמחרוזת השנייה, או מחזירה אפס אם המחרוזות הן שוות ערך, או מחזירה שלם חיובי אם המחרוזת הראשונה ארוכה יותר. השיטה היא קומפרטור אשר מסדר מחרוזות בהתבסס על האורך שלהן, במקום הסידור היותר לקסיקוגרפי טיפוסי. ייחוס לאובייקט StringLengthComparator משמש כ"מצביע פונקציה" לקומפרטור הזה, תוך שהוא מאפשר לו להיקרא בנוגע לזוגות שרירותיים של מחרוזות. במילים אחרות, מופע StringLengthComparator הוא אסטרטגיה מוצקה להשוואת מחרוזות.

כפי שנהוג במחלקות אסטרטגיה מוצקה, מחלקת StringLengthComparator היא חסרת נתונים: אין לה שדות כלשהם, לכן כל מופעי המחלקה הם שווי ערך האחד לשני מבחינת התפקודיות. לכן, ניתן להשתמש בתבנית ה-singleton כדי לחסוך בעלות של יצירת אובייקטים לא הכרחיים:

```
class StringLengthComparator {
  private StringLengthComparator() {  }

  public static final StringLengthComparator
    INSTANCE = new StringLengthComparator();

  public int compare(String s1,String s2){
    return s1.length() - s2.length();
  }
}
```

כדי להעביר מופע StringLengthComparator לשיטה, אנו צריכים טיפוס מתאים עבור הפרמטר. לא יהיה טוב להשתמש ב-StringLengthComparator מכיון שלקוחות לא יהיו מסוגלים להעביר אסטרטגיית השוואה אחרת כלשהי. במקום זאת, אנו צריכים להגדיר ממשק Comparator ולהתאים את StringLengthComparator כך שיממש את הממשק הזה. במילים אחרות, אנו צריכים להגדיר ממשק אסטרטגיה שיתאים למחלקת האסטרטגיה המוצקה. הנה הוא:

```
//Strategy interface
public interface Comparator {
   public int compare(Object o1,Object o2);
}
```

ההגדרה של ממשק ה-Comparator באה במקרה מחבילת התוכנה java.util, אבל אין שום קסם בנוגע לכך. באותה מידה, היית יכול להגדיר אותה בעצמך. אם כן, זה מתאים לקומפרטורים של אובייקטים מלבד מחרוזות שיטת compare שלה, המקבלת פרמטרים מטיפוס Object ולא String. לכן, מחלקת StringLengthComparator שהוצגה לפני כן חייבת להשתנות מעט כדי לממש Comparator: הפרמטרים המתקבלים כ-Object חייבים לעבור המרה ל-String לפני קריאה לשיטת האורך.

לעיתים קרובות, מחלקות אסטרטגיה מוצקה מוצהרות תוך שימוש במחלקות אנונימיות (נושא 18). הביטוי הבא ממיין מערך של מחרוזות לפי האורך שלהן:

```
Arrays.sort(stringArray,new Comparator()  {
   public int compare(Object o1,Object o2)  {
      String s1 = (String)o1;
      String s2 = (String)o2;
      return s1.length() - s2.length();
   }
});
```

מכיון שממשק האסטרטגיה משמש כטיפוס עבור כל מופעי האסטרטגיה המוצקה שלו, מחלקת אסטרטגיה מוצקה לא היתה צריכה להיעשות ציבורית כדי לייצא אסטרטגיה מוצקה. במקום זאת, "מחלקה מארחת" יכולה לייצא שדה סטטי ציבורי (או שיטת ייצור סטטית) אשר הטיפוס שלו הוא ממשק אסטרטגיה, ומחלקת האסטרטגיה המוצקה יכולה להיות מחלקת מקוננת פרטית של המחלקה המארחת. בדוגמה הבאה, מחלקה מקוננת סטטית עדיפה לשימוש על פני מחלקה אנונימית כדי לאפשר למחלקת האסטרטגיה המוצקה לממש גם ממשק שני, Serializable:

```
//Exporting a concrete strategy
class Host {
  ...//Bulk of class omitted

  private static class StrLenCmp
            implements Comparator,Serializable {
      public int compare(Object o1,Object o2) {
        String s1 = (String)o1;
        String s2 = (String)o2;
        return s1.length() - s2.length();
      }
  }

  //Returned comparator is serializable
  public static final Comparator
      STRING_LENGTH_COMPARATOR = new StrLenCmp();
}
```

מחלקת String משתמשת בתבנית זו כדי לייצא קומפרטור מחרוזת שאינו תלוי בגודל אות דרך שדה CASE_INSENSITIVE_ORDER שלה.

לסיכום, השימוש הראשוני של מצביעי פונקציה של שפת C הוא לממש את תבנית האסטרטגיה. כדי לממש תבנית זו בשפת Java, הגדר ממשק לייצוג האסטרטגיה ומחלקה אשר מממשת את הממשק הזה עבור כל אסטרטגיה מוצקה. כאשר נעשה שימוש באסטרטגיה מוצקה רק פעם אחת, המחלקה שלה מוצהרת בצורה אופיינית ואינה יכולה לתת-מופע נוסף תוך שימוש במחלקה אנונימית. כאשר אסטרטגיה מוצקה מיוצאת לשימוש חוזר, המחלקה שלה היא בדרך כלל מחלקת איבר סטטית פרטית והיא מיוצאת דרך שדה סופי סטטי ציבורי, אשר הטיפוס שלו הוא הממשק של האסטרטגיה.

פרק 6

שיטות (Methods)

פרק זה דן בכמה היבטים של תכנון שיטות: איך לנהוג בפרמטרים ולהחזיר ערכים, איך לתכנן חתימות לשיטות ואיך לתעד שיטות. רוב החומר שבפרק זה נכון גם לבנאים. כמו פרק 5, פרק זה מתמקד בשימושיות, עמידות וגמישות.

נושא 23: בדוק את תוקפם של פרמטרים

למרבית השיטות והבנאים יש הגבלות בנוגע לאילו ערכים אפשר לשלוח לפרמטרים שלהם. לדוגמה, אין זה בלתי מקובל שערכי אינדקס חייבים להיות לא שליליים וייחוסי אובייקטים חייבים להיות לא ריקים (null). אתה צריך לתעד הגבלות שכאלו ולאכוף אותן על ידי בדיקות בתחילת גוף השיטה. חשוב לנסות לגלות טעויות מהר ככל האפשר לאחר שהן מתרחשות. אם לא תעשה זאת, אתה מקטין את הסיכויים שטעות תתגלה ומקשה על גילוי מקור הבעיה לאחר שהיא התגלתה.

אם ערך פרמטר לא חוקי מועבר לשיטה והשיטה בודקת את הפרמטרים שלה לפני הביצוע, התהליך יסתיים מהר ובאופן נקי עם חריג מתאים. אם השיטה נכשלת בבדיקת הפרמטרים שלה, כמה דברים עלולים לקרות. השיטה עלולה לקרוס עם חריג מבלבל באמצע העיבוד. גרוע מכך, השיטה יכולה להחזיר ערך באופן נורמלי, אבל לחשב באופן סמוי תוצאה שגויה. גרוע מכל, השיטה יכולה להחזיר ערך באופן נורמלי אבל להותיר מספר אובייקטים במצב שגוי, דבר שיגרום לשגיאה בקוד בזמן לא מוגדר בעתיד.

עבור שיטות ציבוריות, השתמש בתווית התיעוד @throws כדי לתעד את החריג אשר יתעורר אם מופרת הגבלה לגבי ערכי פרמטר (נושא 44). בצורה אופיינית, החריג יהיה IllegelArgumentException, IndexOutOfBoundsException או NullPointerException (נושא 42).

לאחר שתיעדת את ההגבלות לגבי פרמטרים של שיטה ותיעדת את החריגים שיתעוררו אם הגבלות אלו תופרנה, זהו עניין פשוט לאכוף הגבלות אלו. הנה דוגמה אופיינית:

```
/ **
  * Returns a BigInteger whose value is (this mod m).  This method
  * differs from the remainder method in that it always returns a
  * nonnegative BigInteger.
  *
  * @param m the modulus, which must be positive.
  * @return this mod m.
  * @throws ArithmeticException if m <=0.
  * /
public BigInteger mod(BigIntege m)  {
    if (m.signum()<=0)
       throw new ArithmeticException("Modulus not positive");

    ...//Do the computation
}
```

עבור שיטה שאינה מיועדת לייצוא ב-API, אתה כמחבר החבילה שולט בנסיבות שבהן השיטה נקראת, כך שאתה יכול וצריך להבטיח שרק ערכי פרמטר חוקיים מועברים לשיטה. לכן, שיטות לא ציבוריות צריכות בדרך כלל לבדוק את הפרמטרים שלהן על ידי שימוש בטענות (assertions) במקום בדיקות רגילות. אם אתה משתמש בגירסה של Java אשר תומכת בטענות (גירסה 1.4 ומעלה), אתה צריך להשתמש במנגנון ה-assert. אחרת, אתה צריך להשתמש במנגנון בדיקה תחליפי.

חשוב במיוחד לבדוק את תוקפם של פרמטרים אשר השיטה לא משתמשת בהם אלא שומרת אותם במקום אחר לשימוש מאוחר יותר. לדוגמה, חשוב על שיטת הייצור הסטטית שבפרק 4, אשר מקבלת מערך int ומחזירה תצוגת List של המערך. אם לקוח של שיטה זו היה מעביר null, השיטה היתה מעוררת NullPointerException בגלל שהשיטה מכילה בדיקה מפורשת. אם הבדיקה היתה מושמטת, השיטה היתה מחזירה ייחוס למופע List חדש אשר יעורר NullPointerException, ברגע שבו לקוח ינסה להשתמש בו. למרבה הצער, נכון לזמן כתיבת שורות אלו, המקור של מופע List עשוי להיות מאוד קשה לקביעה, דבר אשר יכול לסבך מאוד את המשימה של ניפוי השגיאות.

בנאים מייצגים מקרה מיוחד של העיקרון לפיו אתה צריך לבדוק את תוקפם של הפרמטרים אשר נשמרים במקום אחר לשימוש מאוחר יותר. חשוב מאוד לבדוק את תוקפם של פרמטרים שמועברים לבנאים כדי למנוע בנייה של אובייקט אשר מפר קבועי מחלקה.

יש חריגים לכלל לפיו עליך לבדוק פרמטרים של שיטה לפני שמבצעים את החישוב המשתמש בהם. חריג חשוב הוא המקרה שבו בדיקת התוקף תהיה יקרה או לא מעשית ובדיקת התוקף מבוצעת כבר במרומז בתהליך ביצוע החישוב. לדוגמה, חשוב על שיטה שממיינת רשימה של אובייקטים, כמו Collections.sortList. כל האובייקטים ברשימה חייבים להיות בני-השוואה באופן הדדי. בתהליך מיון הרשימה, כל אובייקט ברשימה יושווה לאיזה אובייקט אחר ברשימה. אם האובייקטים לא בני-השוואה באופן הדדי, אחת מההשוואות האלו תעורר ClassCastException, שזה בדיוק מה ששיטת המיון צריכה לעשות. לכן, אין טעם לבדוק לפני הזמן שהאלמנטים ברשימה היו בני-השוואה באופן הדדי. בכל אופן, שים לב שיישום ללא אבחנה של טכניקה זו יכולה להתבטא באובדן של failure atomicity (נושא 46). לפעמים, חישוב מבצע במרומז את בדיקת התוקף הנדרשת לגבי כמה פרמטרים אבל מעורר את החריג הלא נכון אם הבדיקה נכשלת. כלומר, החריג שהחישוב יעורר באופן טבעי כתוצאה מערך פרמטר לא תקף לא תואם לחריג שהשיטה אמורה לעורר לפי התיעוד שערכת. בנסיבות שכאלו, אתה צריך להשתמש בקוד תרגום החריג שמתואר בנושא 43 כדי לתרגם את החריג הטבעי לחריג הנכון.

אל תסיק מנושא זה שהגבלות שרירותיות על פרמטרים הן דבר טוב. ההיפך הוא הנכון, עליך לתכנן שיטות כך שהן תהיינה כלליות עד כמה שניתן. כמה שתציב פחות הגבלות על פרמטרים כך ייטב, בהנחה שהשיטה יכולה לעשות משהו הגיוני עם כל ערכי הפרמטר שהיא מקבלת. אולם, לעיתים יש הגבלות שהן מהותיות להפשטה שממומשת. לסיכום, בכל פעם שאתה כותב שיטה או בנאי, עליך לחשוב על ההגבלות שחלות על הפרמטרים שלהם. עליך לתעד הגבלות אלו ולאכוף אותן בעזרת בדיקות מפורשות בתחילת גוף השיטה. חשוב להתרגל לעשות זאת. העבודה המצומצמת שזה גורר תשתלם לבסוף בתוספת ריבית בפעם הראשונה שבדיקת תוקף הפרמטרים תזהה פרמטר בלתי חוקי.

נושא 24: צור בעת הצורך עותקים הגנתיים

דבר אחד אשר עושה את שפת Java כל כך נוחה לשימוש הוא ששפת Java היא **שפה בטוחה**. הדבר אומר שבהיעדר שיטות native השפה חסינה בפני גלישת חוצץ, גלישת מערכים, מצביעים פרועים ושאר שגיאות נפילת זיכרון אשר אפשר למצוא בשפות לא בטוחות כמו C ו- ++C. בשפה בטוחה אפשר לכתוב מחלקות ולדעת בוודאות שהקבועים שלהן יישארו קבועים, לא משנה מה קורה בחלק אחר של המערכת. דבר זה אינו אפשרי בשפות אשר מתייחסות אל הזיכרון כאל מערך אחד ענק.

כמו כן, אפילו בשפה בטוחה לא תהיה מבודד ממחלקות אחרות ללא מאמץ מסוים מצידך. **אתה חייב לתכנת באופן הגנתי תוך הנחה שלקוחות של המחלקה שלך יעשו ככל יכולתם להרוס את הקבועים של המחלקה**. הדבר יכול להיות גם ממש נכון אם מישהו מנסה לשבור את האבטחה של המערכת שלך, אבל סביר יותר שהמחלקה שלך תיאלץ להתמודד עם התנהגות בלתי צפויה שנובעת מטעויות תמימות מצד המתכנת שמשתמש ב-API שלך. בגלל שתי סיבות אלו, כדאי להקדיש זמן כדי לכתוב מחלקות עמידות בפני לקוחות שלא ינהגו בהן כראוי.

למרות שבלתי אפשרי למחלקה אחרת לשנות את מצבו הפנימי של אובייקט בלי שום עזרה מצד האובייקט, קל באופן מפתיע לספק עזרה שכזו בלי להתכוון לכך. לדוגמה, בחן את המחלקה הבאה אשר מתיימרת לייצג תקופת זמן שאינה ניתנת לשינוי:

```
// Broken "immutable" time period class
public final class Period {
    private final Date start;
    private final Date end;

    /**
     * @param  start the beginning of the period.
     * @param  end the end of the period; must not precede start.
     * @throws IllegalArgumentException if start is after end.
     * @throws NullPointerException if start or end is null.
     *
     * Broken! - Requires defensive copying of parameters.
     */
    public Period(Date start, Date end) {
        if (start.compareTo(end) > 0)
            throw new IllegalArgumentException(start + " after " + end);
        this.start = start;
        this.end   = end;
    }

    public Date start() {
        return start;
    }
    public Date end() {
        return end;
```

```
  }
  ... // Remainder omitted
}
```

במבט ראשון, מחלקה זו עשויה להיראות לא ניתנת לשינוי ואוכפת את קביעת הקבוע, כך שההתחלה של תקופה לא תעבור את הסוף שלה. בכל אופן, קל להפר קבוע זה על ידי ניצול העובדה ש-Date ניתן לשינוי:

```
// Attack the internals of a Period instance.
Date start = new Date();
Date end = new Date();
Period p = new Period(start, end);
end.setYear(78);  // Modifies internals of p!
```

כדי להגן על התכונות הפנימיות של מופע Period מכזה סוג של התקפה, **חיוני ליצור *עותק הגנתי* של כל פרמטר ניתן לשינוי שהבנאי מקבל** ולהשתמש בעותקים כמרכיבים של מופע Period במקום המקוריים:

```
// Repaired constructor - makes defensive copies of parameters
  public Period(Date start, Date end) {
    this.start = new Date(start.getTime());
    this.end  = new Date(end.getTime());

    if (this.start.compareTo(this.end) > 0)
      throw new IllegalArgumentException(start +" after "+ end);
  }
```

כעת, להתקפה הקודמת לא תהיה שום השפעה על מופע Period. שים לב ש**עותקים הגנתיים נעשים *לפני* בדיקת תוקפם של הפרמטרים (נושא 23) ובדיקת התוקף מתבצעת על העותקים ולא על המקוריים**. למרות שזה עשוי להיראות לא טבעי, זה הכרחי. זה מגן על המחלקה מפני שינויים בפרמטרים ממטלה אחרת במהלך "חלון הפגיעות" בין הזמן שבו הפרמטרים נבדקים לבין הזמן שבו הפרמטרים מועתקים.

שים לב גם שלא השתמשנו בשיטת clone של Date כדי ליצור את העותקים ההגנתיים. בגלל ש-Date הוא לא סופי, לא בטוח ששיטת clone תחזיר אובייקט שהמחלקה שלה היא java.util.Date. היא היתה יכולה להחזיר מופע של תת-מחלקה שאי אפשר לבטוח בה ושתוכננה במיוחד לנזק זדוני.

לדוגמה, תת-מחלקה שכזו היתה יכולה לרשום ייחוס לכל מופע ברשימה סטטית פרטית בזמן יצירתו ולאפשר לתוקף גישה לרשימה. דבר זה ייתן לתוקף שלטון חופשי על כל המופעים. כדי למנוע סוג זה של התקפה, **אל תשתמש בשיטת clone כדי ליצור עותק הגנתי של פרמטר מטיפוס שאפשר לבצע בו תת-מחלקתיות על ידי צד שלישי שאי אפשר לסמוך עליו**.

למרות שהבנאי החילופי מגן בהצלחה מפני ההתקפה הקודמת, עדיין אפשר לשנות מופע Period בגלל שאמצעי הגישה שלו מציעים גישה לתכונות הפנימיות המשתנות שלו:

```
//Second attack on the internals of a Period instance
Date start = new Date();
Date end = new Date();
Period p = new Period(start, end);
p.end().setYear(78); //Modifies internals of p!
```

כדי להתגונן מפני ההתקפה השנייה, פשוט הגדר את אמצעי הגישה כך ש**יחזירו עותקים הגנתיים של שדות תכונות פנימיות משתנות**:

```
// Repaired accessors - make defensive copies of internal fields
   public Date start() {
      return (Date) start.clone();
   }

   public Date end() {
      return (Date) end.clone();
   }
```

עם הבנאי החדש ואמצעי הגישה החדשים במקום, Period באמת לא ניתן לשינוי. לא משנה עד כמה מתכנת הוא בעל כוונות זדוניות או בעל ידע מצומצם מאוד בתכנות, פשוט אין שום דרך שבה הוא יכול להפר את הקבוע לפיו התחלת תקופה לא תבוא אחרי סוף התקופה. הדבר נכון בגלל שאין שום דרך שבה מחלקה כלשהי פרט למחלקת Period עצמה תשיג גישה לאחד מהשדות המשתנים שבמופע Period. שדות אלה באמת מכומסים בתוך האובייקט.

שים לב שאמצעי הגישה החדשים, שלא כמו הבנאי החדש, כן משתמשים בשיטת clone כדי ליצור עותקים הגנתיים. הדבר מקובל (למרות שאינו נדרש), כיון שאנו יודעים בוודאות שהמחלקה של אובייקטי Date הפנימיים של Period היא java.util.Date ולא אי אלו תת-מחלקות אפשריות שאי אפשר לסמוך עליהן.

העתקה הגנתית של פרמטרים היא לא רק עבור מחלקות בלתי משתנות. כל פעם שאתה כותב שיטה או בנאי אשר מכניסים אובייקט המסופק על ידי לקוח לתוך מבנה נתונים פנימי, חשוב האם לאובייקט שסופק על ידי הלקוח יש פוטנציאל להיות אובייקט ניתן לשינוי. אם אכן זה כך, חשוב האם המחלקה שלך תתיר שינוי באובייקט לאחר שהוא נכנס לתוך מבנה הנתונים.

אם התשובה היא לא, אתה חייב להעתיק את האובייקט באופן הגנתי ולהכניס את האובייקט לתוך מבנה הנתונים במקום האובייקט המקורי. לדוגמה, אם אתה שוקל שימוש בייחוס אובייקט מסופק לקוח כאלמנט במופע Set פנימי או כמפתח במופע Map פנימי, עליך להיות ער לכך שהקבועים של הסט או המפה ייהרסו אם האובייקט ישתנה לאחר שהוא הוכנס.

אותו הדבר נכון עבור העתקה הגנתית של מרכיבים פנימיים לפני החזרתם ללקוחות. בין אם המחלקה שלך ניתנת לשינוי או לא, עליך לחשוב פעמיים לפני שאתה מחזיר ייחוס למרכיב פנימי ניתן לשינוי. אתה מסתכן בכך שאתה מחזיר עותק הגנתי. כמו כן, קריטי לזכור שמערכים בעלי אורך שאינו אפס הם תמיד בני-שינוי, לכן אתה תמיד צריך ליצור עותק הגנתי של מערך פנימי לפני החזרתו ללקוח. באופן חילופי, אתה יכול להחזיר למשתמש תצוגה שאינה ניתנת לשינוי של המערך. שתי טכניקות אלו הוצגו בנושא 12.

הלקח האמיתי שבכל זה הוא שאתה צריך, כאשר הדבר אפשרי, להשתמש באובייקטים בלתי משתנים כמרכיבים של האובייקטים שלך כך שאינך צריך לדאוג בנוגע להעתקה הגנתית (נושא 13), אבל אפשר להתווכח על כך. במקרה של הדוגמה Period שלנו, ראוי להזכיר שמתכנתים מנוסים משתמשים לעיתים קרובות בייחוס אובייקט long Date הפשוט. הם עושים זאת קודם כל בגלל ש-Date הוא בר-שינוי.

אין זה תמיד מתאים ליצור עותק הגנתי של פרמטר בר-שינוי לפני שילובו באובייקט. יש כמה שיטות ובנאים אשר הקריאה שלהם מראה **handoff** מפורשת של האובייקט שמיוחס על ידי פרמטר. כאשר קוראים לכזו שיטה, הלקוח מבטיח שיותר היא לא תשנה את האובייקט ישירות. שיטה או בנאי אשר מצפים לקחת שליטה על אובייקט ניתן לשינוי המסופק על ידי הלקוח חייבים להבהיר זאת בתיעוד שלהם.

מחלקות שמכילות שיטות או בנאים אשר הקריאה שלהם מראה על העברת שליטה לא יכולות להגן על עצמן מפני לקוחות בעלי כוונות זדון. מחלקות שכאלו הן לגיטימיות רק כאשר יש אמון הדדי בין המחלקה והלקוח שלה, או כאשר גרימת נזק לקבועי המחלקה לא תפגע באף אחד פרט ללקוח. דוגמה למצב מאוחר יותר היא תבנית מחלקת עטיפה (נושא 14). בהסתמך על מה שידוע על מחלקת העטיפה, הלקוח יכול היה להרוס את קבועי המחלקה על ידי גישה ישירה לאובייקט לאחר שהוא נעטף, אבל הדבר יפגע רק בלקוח.

נושא 25: תכנן בזהירות חתימות שיטה

נושא זה הוא צרור עצות לתכנון API אשר לא ראויות כל כך לנושאים עצמאיים משלהן. עצות אלו המקובצות יחדיו תעזורנה לך לעשות את ה-API שלך קל יותר ללימוד ולשימוש ועם נטייה פחותה לטעויות.

בחר בזהירות שמות לשיטות. שמות תמיד צריכים לציית למוסכמות הסטנדרטיות של מתן השמות (נושא 38). מטרתך הראשונית צריכה להיות בחירת שמות מובנים ועקביים ביחס לשמות אחרים באותה חבילת תוכנה. מטרתך המשנית צריכה להיות בחירת שמות עקביים ביחס לקונצנזוס הרחב, במקום שבו ישנו קונצנזוס שכזה. במקרה של ספק, עיין ב-APIs של ספריית Java להנחיה.
למרות שיש הרבה חוסר עקביות - מחויב המציאות, בהתחשב בגודל ובטווח ההכרה של הספריות - ישנו גם קונצנזוס. מקור רב ערך הוא ספרו של Patrick Chan, The Java Developers Almanach [Chan00], אשר כולל את הצהרות השיטות עבור כל שיטה נפרדת בספריות הפלטפורמה של Java, לפי אינדקס אלפביתי. לדוגמה, אם אתה מתלבט האם לקרוא לשיטה remove או delete, מבט חטוף באינדקס של הספר הזה יאמר לך ש-remove היא הבחירה המובנת מאליה. יש מאות שיטות אשר שמותיהן מתחילים ב-remove וחופן קטן של שיטות אשר שמותיהן מתחילים ב-delete.

אל תגזים באספקת "שיטות נוחות". כל שיטה צריכה "למשוך את המשקל שלה". יותר מדי שיטות גורמות למחלקה להיות קשה ללימוד, לשימוש, לתיעוד, לבדיקה ולשמירה. הדבר נכון כפליים עבור ממשקים, שבהם יותר מדי שיטות מסבכות את החיים של מממשים כמו גם של משתמשים. עבור כל פעולה שנתמכת על ידי הטיפוס שלך, ספק שיטה שמתפקדת באופן מלא. שקול אספקת "קצרנות" עבור פעולה רק אם ייעשה בה שימוש בתדירות רבה. **כאשר יש ספק בנוגע לכך, ותר על הקצרנות**.

המנע מרשימות פרמטר ארוכות. כחוק, באופן מעשי, שלושה פרמטרים צריכים להיראות לכל היותר וככל שיפחת מספרם כן ייטב. מרבית המתכנתים לא יכולים לזכור רשימות פרמטרים ארוכות יותר מזה. אם הרבה מהשיטות שלך חוצות גבול זה, ה-API שלך לא יהיה שימושי בלי התייחסות קבועה לתיעוד שלו.
טורים ארוכים של פרמטרים מאותו טיפוס מזיקים במיוחד. לא רק שהמשתמשים ב-API שלך לא יהיו מסוגלים לזכור את הסדר של הפרמטרים, אלא גם כאשר הם ישנו את סדר הפרמטרים בטעות, התוכניות שלהם עדיין תהודרנה ותרוצנה. הן פשוט לא תעשינה את מה שהמחברים שלהן התכוונו.

יש שתי טכניקות לקיצור רשימות פרמטרים ארוכות מדי. האחת היא לשבור את השיטה לשיטות מרובות שכל אחת מהן דורשת רק חלק מהפרמטרים. אם הדבר נעשה בחוסר זהירות, זה יכול להוביל להרבה שיטות, אבל זה גם יכול לעזור **להפחית** את כמות השיטות על ידי הגברת האורתוגונליות (orthogonality, הפיכת הרשימה לאורכית במקום רוחבית).

לדוגמה, חשוב על הממשק של java.util.List. הוא לא מספק שיטות למציאת האינדקס הראשון או האחרון של אלמנט בתת-רשימה, פעולות אשר אמורות לדרוש שלושה פרמטרים. במקום זאת הוא מספק את שיטת subList, אשר מקבלת שני פרמטרים ומחזירה view של תת-רשימה. אפשר לצרף את השיטה עם שיטות indexOf או LastIndexOf, שלכל אחת מהן יש פרמטר בודד, כדי לתת את התפקודיות המבוקשת. יותר מכך, אפשר לצרף את שיטת subList עם כל שיטה אחרת אשר פועלת על מופע של List כדי לבצע חישובים שרירותיים על תת-מחלקות. ל-API המתקבל יש יחס משקל כוח גבוה.

טכניקה שנייה לקיצור רשימות פרמטרים ארוכות מדי היא ליצור **מחלקות עוזר** כדי לשמור צירופים של פרמטרים. בצורה אופיינית, מחלקות עוזר אלו הן מחלקות מקוננות סטטיות (נושא 18). טכניקה זו מומלצת אם סדרת פרמטרים שחוזרת על עצמה לעיתים קרובות נראית כדי לייצג ישויות נפרדות מסוימות. לדוגמה, נניח שאתה כותב מחלקה אשר מייצגת משחק קלפים ואתה מוצא את עצמך מעביר באופן קבוע רצף של שני פרמטרים אשר מייצגים דרגה של קלף ואת הסדרה שלו. ה-API שלך, כמו התכונות הפנימיות של המחלקה שלך, עשויים כפי הנראה להשתפר אם תוסיף מחלקת עוזר כדי לייצג קלף ותחליף כל התרחשות של רצף הפרמטרים בפרמטר בודד של מחלקת העוזר.

עבור טיפוסי פרמטרים, העדף ממשקים על פני מחלקות. בכל פעם שמתקיים ממשק מתאים להגדרת פרמטר, העדף אותו על פני מחלקה אשר מממשת את הממשק. לדוגמה, אף פעם אין שום סיבה לכתוב שיטה אשר מקבלת Hashtable כקלט - השתמש ב-Map במקום. הדבר מאפשר לך להעביר TreeMap, HashMap, Hashtable, תת-מפה של TreeMap או כל מימוש Map שעדיין לא נכתב. על ידי שימוש במחלקה במקום בממשק, אתה מגביל את הלקוח שלך למימוש מיוחד וכופה פעולת העתקה לא נצרכת ובעלת פוטנציאל להיות יקרה, אם נתוני הקלט קיימים באיזו שהיא צורה אחרת.

השתמש באובייקטי פונקציה (נושא 22) בתבונה. ישנן כמה שפות, במיוחד Smalltalk והניבים השונים של Lisp, אשר מעודדות סגנון תכנות עשיר באובייקטים אשר מייצגים פונקציות שהן שימושיות לאובייקטים אחרים. מתכנתים מנוסים של שפות אלו עשויים להתפתות לאמץ סגנון דומה בעת תכנות בשפת Java, אבל אין זו דרך מתאימה מאוד. הדרך הקלה ביותר ליצור אובייקט פונקציה היא בעזרת מחלקה אנונימית (נושא 18), אבל אפילו דרך זו מערבת קצת ערבוביה תחבירית ויש לה הגבלות בכוח ובביצועים בהשוואה למושגי שליטה שהם inline. יתר על כן, סגנון התכנות שבו אתה יוצר באופן קבוע אובייקטי פונקציה ומעביר אותם משיטה לשיטה יצא מהאופנה, לכן למתכנתים אחרים יידרש זמן רב כדי להבין את הקוד שלך אם תאמץ סגנון תכנות שכזה. הכוונה אינה שלאובייקטי פונקציה אין שימושים לגיטימיים. הם חיוניים להרבה תבניות תכנון רבות עוצמה, כמו *Strategy* [Gamma95 עמוד 315], ו-*Visitor* [Gamma95 עמוד 331]. פרט לכך, באובייקטי פונקציה יש להשתמש רק אם יש סיבה טובה לכך.

נושא 26: השתמש בשיקול דעת בהעמסה

הנה ניסיון תמים למיין אוספים על פי סוגם (האם הם סטים, רשימות או סוגים אחרים של אוספים):

```
// Broken - incorrect use of overloading!
public class CollectionClassifier {
    public static String classify(Set s) {
        return "Set";
    }

    public static String classify(List l) {
        return "List";
    }

    public static String classify(Collection c) {
        return "Unknown Collection";
    }

    public static void main(String[] args) {
        Collection[] tests = new Collection[] {
            new HashSet(),          // A Set
            new ArrayList(),        // A List
            new HashMap().values()  // Neither Set nor List
        };

        for (int i = 0; i < tests.length; i++)
            System.out.println(classify(tests[i]));
    }
}
```

אתה עשוי לצפות מתוכנית זו להדפיס "Set" אשר אחריו בא "List" ו- "Unknown Collection" אבל התוכנית לא מבצעת זאת. היא מדפיסה "Unknown Collection" שלוש פעמים. מדוע זה קורה? בגלל ששיטת classify **מועמסת והבחירה לאיזו העמסה לקרוא מתבצעת בזמן הידור**. עבור כל שלושת האיטרציות של הלולאה, טיפוס זמן ההידור של הפרמטר הוא אותו הדבר: Collection. טיפוס זמן הריצה הוא שונה בכל איטרציה, אבל אין זה משפיע על הבחירה של העמסה. בגלל שטיפוס זמן ההידור של הפרמטר הוא Collection, ההעמסה היחידה המתאימה היא השלישית, classify(Collection) והעמסה זאת נקראת בכל איטרציה של הלולאה.

התנהגותה של תוכנית זו היא בניגוד לאינטואיציה בגלל **שבחירה בין שיטות מועמסות היא סטטית, בעוד שבחירה בין שיטות דרוסות היא דינמית**. הגירסה הנכונה של שיטה **דרוסה** נבחרת בזמן ריצה, בהתבסס על טיפוס זמן הריצה של האובייקט שלגביו השיטה נקראת.

תזכורת: שיטה נדרסת כאשר תת-מחלקה אשר מכילה הצהרת שיטה עם בדיוק אותה החתימה כמו הצהרת שיטה במחלקת אב. אם שיטת מופע נדרסת בתת-מחלקה ושיטה זו נקראת על מופע של תת-המחלקה, **השיטה הדורסת** של תת-המחלקה מתבצעת, ללא התחשבות בטיפוס זמן ההידור של מופע תת-המחלקה. כדי להבהיר זאת, התבונן בתוכנית הקטנה הבאה:

```
// Overriding example

class A {
   String name() { return "A"; }
}

class B extends A {
   String name() { return "B"; }
}

class C extends A {
   String name() { return "C"; }
}

public class Overriding {
   public static void main(String[] args) {
      A[] tests = new A[] { new A(), new B(), new C() };

      for (int i = 0; i < tests.length; i++)
         System.out.print(tests[i].name());
   }
}
```

השיטה name מוצהרת במחלקה A ונדרסת במחלקות B ו-C. כפי שאתה עשוי לצפות, תוכנית זו מדפיסה "ABC" למרות שטיפוס זמן ההידור של המופע הוא A בכל חזרה של הלולאה. לטיפוס זמן ההידור של אובייקט אין שום השפעה לגבי איזו שיטה תתבצע כאשר שיטה נדרסת נקראת. השיטה הנדרסת "הכי ספציפית" תמיד תתבצע. השווה זאת להעמסה, שבה לטיפוס זמן הריצה אין שום השפעה לגבי איזו העמסה תתבצע. הבחירה נעשית בזמן הידור, בהתבסס לגמרי על טיפוסי זמן ההידור של הפרמטרים.

בדוגמה CollectionClassifier, מטרת התוכנית היתה להבחין בטיפוס הפרמטר על ידי שליחה אוטומטית להעמסת השיטה המתאימה בהתבסס על טיפוס זמן הריצה של הפרמטר, בדיוק כמו ששיטת name עשתה בדוגמה של "ABC". העמסת שיטות פשוט לא מספקת תפקודיות זו.

הדרך לתקן את התוכנית היא על ידי החלפה של כל שלוש ההעמסות של classify בשיטה בודדת אשר מבצעת מבחן instanceof מפורש:

```
public static String classify(Collection c) {
    return (c instanceof Set ? "Set" :
        (c instanceof List ? "List" : "Unknown Collection"));
  }
```

בגלל שדריסה היא מקובלת והעמסה היא חריגה, קיימת ציפייה להתנהגות של דריסה בקריאה לשיטה. כפי שהודגם על ידי הדוגמה CollectionClassifier, העמסה יכולה בקלות לבלבל ציפיות אלו. זהו נוהג רע לכתוב קוד אשר ההתנהגות שלו לא תהיה ברורה מאליה למתכנת הממוצע ללא חקירה. הדבר נכון במיוחד עבור APIs. אם המשתמש הטיפוסי של API אינו יודע איזו מהעמסת שיטות שונות תיקרא עבור סט נתון של פרמטרים, שימוש ב-API עתיד להתבטא בטעויות. טעויות אלו כנראה תתבטאנה בהתנהגות לא יציבה בזמן ריצה ומתכנתים רבים לא יהיו מסוגלים לאבחן אותן, לכן אתה צריך **להימנע משימושים מבלבלים בהעמסה**.

מה בדיוק גורם לשימוש מבלבל בהעמסה נתון לוויכוח. **מדיניות בטוחה ושמרנית היא לעולם לא לייצא שתי העמסות עם אותו מספר של פרמטרים**. אם אתה דבק בהגבלה זו, למתכנתים לעולם לא יהיה ספק לגבי איזו העמסה מתייחסת לסט כלשהו של פרמטרים. הגבלה זו אינה מפריעה מאוד בגלל שאתה תמיד יכול לתת לשיטות שמות שונים במקום להשתמש בהעמסה.

לדוגמה, חשוב על המחלקה ObjectOutputStream. יש לה גירסה של שיטת write שלה עבור כל טיפוס פשוט ועבור מספר טיפוסי ייחוס. במקום שתתקיים העמסה של שיטת write, לגרסאות אלו יש חתימות נפרדות כמו writeBoolean(boolean), writeInt(int) ו-writeLong(long).

עבור בנאים, אין לך את האפשרות להשתמש בשמות נפרדים. בנאים מרובים של מחלקה הם תמיד מועמסים. יש לך, במקרים מסוימים, את האפשרות לייצא יצרנים סטטיים במקום בנאים (נושא 1), אבל אין זה תמיד מעשי. הצד הטוב הוא שבנוגע לבנאים אתה לא צריך לדאוג לגבי פעולות גומלין בין העמסה לבין דריסה, מכיון שבנאים לא יכולים להידרס. בגלל שסביר שתהיה לך הזדמנות לייצא בנאים מרובים עם מספר זהה של פרמטרים, משתלם לדעת מתי בטוח לעשות זאת.

ייצוא העמסות מרובות עם אותו מספר של פרמטרים לא יבלבל מתכנתים אם תמיד יהיה ברור איזו העמסה מתייחסת לסט נתון כלשהו של פרמטרים אקטואליים. זהו המקרה שבו לפחות לפרמטר צורני מקביל אחד בכל זוג של העמסות יש טיפוס "שונה באופן קיצוני" בשתי ההעמסות. שני טיפוסים הם שונים באופן קיצוני אם בלתי אפשרי בבירור להמיר מופע של טיפוס אחד לטיפוס השני. בנסיבות אלו, ההעמסה מתייחסת לסט נתון של פרמטרים אקטואליים ונקבעת לחלוטין על ידי טיפוסי זמן הריצה של הפרמטרים ולא יכולה להיות מושפעת על ידי טיפוסי זמן ההידור שלהם, כך שהמקור העיקרי לבלבול, מתנדף.

לדוגמה, ל-ArrayList יש בנאי אחד אשר מקבל int ובנאי שני אשר מקבל Collection. קשה לתאר בלבול כלשהו לגבי איזה בנאי ייקרא באלו נסיבות, מכיון שטיפוסים פשוטים וטיפוסי ייחוס שונים אחד מהשני באופן מהותי. באופן דומה, ל-BigInteger יש בנאי אחד אשר מקבל byte ובנאי אחר אשר מקבל String. שום בלבול לא ייתכן במקרה זה. טיפוסי מערכים ומחלקות מלבד Object שונים באופן מהותי. דוגמה נוספת לסיום, ל-Throwable, בגירסה 1.4, יש בנאי אחד אשר מקבל String ובנאי אחר אשר מקבל Throwable. המחלקות String ו-Throwable הן **לא קרובות**, כלומר אף אחת מהן היא לא צאצא של השנייה. בלתי אפשרי לאובייקט כלשהו להיות מופע של שתי מחלקות שאינן קרובות, לכן מחלקות שאינן קרובות שונות באופן מהותי.

יש כמה דוגמאות נוספות של זוגות טיפוסים אשר לא יכולים להיות מוחלפים בשני הכיוונים [JLS, 5.1.7], אבל ברגע שאתה בודק מעבר למקרים פשוטים אלה, המתכנת הממוצע עשוי להתקשות מאוד להבחין איזו העמסה, אם בכלל, מתייחסת לסט של פרמטרים אקטואליים. המפרט שקובע איזו העמסה נבחרת הוא מורכב ומתכנתים מעטים מבינים את כל השינויות שלו [JLS, 15.12.1-3].

לעיתים אתה תהיה חייב להפר קווים מנחים אלה כאשר תשפר מחלקות קיימות, כך שהן תממשנה ממשקים חדשים. לדוגמה, לרבים מטיפוסי הערכים בספריות הפלטפורמה של Java היו שיטות "compareTo" מטיפוס עצמי" לפני ההכנסה של ממשק Comparable. הנה ההצהרה לשיטת compareTo המקורית מטיפוס עצמי של String:

```
public int compareTo(String s);
```

עם ההכנסה של ממשק Comparable, כל המחלקות האלו שופרו כך שתממשנה את הממשק הזה, אשר כלל הוספה של שיטת compareTo יותר כללית עם ההצהרה הזאת:

```
public int compareTo(Object o);
```

למרות שההעמסה המתקבלת היא הפרה ברורה של הקווים המנחים שנזכרו לעיל, אין הדבר גורם לנזק כלשהו כל עוד שתי השיטות המועמסות תמיד מבצעות בדיוק את אותו הדבר כאשר הן נקראות ביחס לאותם הפרמטרים. המתכנת עשוי לא לדעת איזו העמסה תיקרא, אבל אין זה משנה כל עוד שתי השיטות מחזירות את אותה התוצאה. הדרך הסטנדרטית להבטיח התנהגות זו היא להשתמש בהעמסה היותר כללית לפני ההעמסה היותר ספציפית:

```
public int compareTo(Object o) {
    return compareTo((String o);
}
```

ביטוי דומה משמש לעיתים לשיטות equals:

```
public boolean equals(Object o) {
    return o instanceof String && equals((String) o);
}
```

קוד זה אינו מזיק ועשוי להתבטא בביצועים מעט משופרים אם טיפוס זמן ההידור של הפרמטר מתאים לפרמטר של ההעמסה היותר ספציפית. אך למרות זאת אין זה מספיק משמעותי כדי לעשות זאת כעניין שבשגרה (נושא 37).

למרות שספריות הפלטפורמה של Java דבקות באופן נרחב לעצה שבנושא זה, ישנם מספר מקומות שבהם היא מופרת. לדוגמה, מחלקת String מייצאת שתי שיטות ייצור סטטיות מועמסות, valueOf(char[]) ו-valueOf(Object) אשר עושות דברים שונים לחלוטין כאשר מעבירים להן את אותו ייחוס אובייקט. אין שום הצדקה ממשית לכך, ויש להתייחס לכך כאל מקרה לא רגיל בעל פוטנציאל שיכול לגרום בלבול אמיתי.

לסיכום, העובדה שאתה יכול להעמיס שיטות לא אומרת שאתה צריך לעשות זאת. בדרך כלל אתה צריך להימנע מהעמסת שיטות עם חתימות כפולות אשר יש להן את אותו מספר פרמטרים. בכמה מקרים, במיוחד במקרים בהם מעורבים בנאים, ייתכן שאי אפשר יהיה לשמוע לעצה זו. במקרה כזה, עליך לפחות להימנע ממצבים שבהם אותו סט של פרמטרים יכול להיות מועבר להעמסות שונות על ידי התוספת של המרה.

אם אי אפשר להימנע ממצב שכזה, לדוגמה בגלל שאתה משפר מחלקה קיימת כך שתממש ממשק חדש, אתה צריך להבטיח שכל ההעמסות מתנהגות באותו האופן כאשר מועברים להן אותם הפרמטרים. אם אינך מצליח לעשות זאת, המתכנתים לא יהיו מסוגלים לעשות שימוש יעיל בשיטות המועמסות או בבנאים והם לא יבינו מדוע אין זה עובד.

נושא 27: החזר מערכים באורך אפס, לא nulls

אין זה נדיר לראות שיטות אשר נראות כמו הדבר הבא:

```
private List cheesesInStock = ...;

/ **
  * @return an array containing all of the cheeses in the shop,
  *         or null if no cheeses are available for purchase.
  * /
public Cheese[] getCheeses() {
    if (cheesesInStock.size() == 0)
          return null;
    ...
}
```

אין שום סיבה לבנות מקרה מיוחד עבור המצב שבו אף גבינה אינה זמינה לרכישה. מעשה כזה מצריך קוד נוסף בלקוח כדי לטפל בערך מוחזר null, לדוגמה:

```
Cheese[] cheeses = shop.getCheeses();
       if (cheeses != null &&
          Arrays.asList(shop.getCheeses()).contains(Cheese.STILTON))
          System.out.println("Jolly good, just the thing.");
```

במקום:

```
       if (Arrays.asList(shop.getCheeses()).contains(Cheese.STILTON))
          System.out.println("Jolly good, just the thing.");
```

סוג כזה של גיבוב מילים נדרש כמעט בכל שימוש של שיטה אשר מחזירה null במקום מערך באורך אפס. הוא נוטה להפיק טעויות, כיון שהמתכנת אשר כותב את הלקוח עלול לשכוח לכתוב קוד מקרה מיוחד כדי לטפל בהחזרת null. טעות שכזו עלולה להתקיים במשך שנים כיון ששיטות שכאלו בדרך כלל מחזירות אובייקט אחד או יותר. פחות חשובה אבל עדיין ראויה להתייחסות היא העובדה שהחזרת null במקום מערך באורך אפס גם מסבכת את השיטה שמחזירה בעצמה מערך.

לעיתים ישנו ויכוח על כך שערך מוחזר null עדיף על פני החזרת מערך באורך אפס בגלל שזה חוסך את ההוצאה הכרוכה בהקצאת המערך. טיעון זה נופל בגלל שתי סיבות. ראשית, לא רצוי לדאוג בנוגע לביצועים ברמה זו אלא אם כן ניתוח פרופיל הראה שהשיטה הנדונה ממש גורמת לבעיות ביצועים (נושא 37).

שנית, אפשר להחזיר את אותו מערך באורך אפס מכל קריאה אשר לא מחזירה פריט כלשהו, מכיון שמערכים באורך אפס הם בלתי משתנים ואובייקטים בלתי משתנים אמורים להיות משותפים באופן חופשי (נושא 13). למעשה, זה בדיוק מה שקורה כאשר אתה משתמש בקוד הסטנדרטי לריקון פריטים מאוסף לתוך מערך מטיפוס מסוים:

```
private List cheesesInStock = ...;
private final static Cheese[] NULL_CHEESE_ARRAY = new Cheese[0];

/**
  * @return an array containing all of the cheeses in the shop.
  * /
public Cheese[] getCheeses() {
   return (Cheese[]) cheesesInStock.toArray(NULL_CHEESE_ARRAY);
}
```

בקוד הזה, קבוע של מערך באורך אפס מועבר לשיטה toArray כדי להצביע על טיפוס הערך המוחזר המבוקש.
בדרך כלל, שיטת toArray מקצה את המערך המוחזר, אבל אם האוסף הוא ריק, היא מתאימה את מערך הקלט, והמפרט עבור Collection.toArray(Object[]) ערב לכך שמערך הקלט יוחזר אם הוא גדול מספיק להחזיק באוסף. לכן, הקוד לעולם לא מקצה מערך באורך אפס אבל במקום זאת הוא משתמש מחדש ב"קבוע מגדיר טיפוס".

בקיצור, **אין כל סיבה אי פעם להחזיר null משיטה שמחזירה מערך במקום החזרת מערך באורך אפס**. מקורה של הנטייה המוטעית להחזיר null היא לרוב משפת C שבה אורכי מערך מוחזרים בנפרד מהמערך עצמו, ואין שום טעם בהקצאת מערך אם אפס מוחזר כאורכו.

נושא 28: כתוב הערות תיעוד לכל מרכיבי ה- API הגלויים

אם API נועד שיעשו בו שימוש, הוא חייב להיות מתועד. בעבר, התיעוד של ה-API היה נוצר באופן ידני ושמירה על תיאום בין ה-API לבין הקוד היתה עבודה יומיומית רבה. סביבת הפיתוח של Java מקלה על משימה זו על ידי תוכנת עזר שנקראת Javadoc. תוכנת העזר יוצרת תיעוד API באופן אוטומטי מקוד מקור השתולות בו *הערות תיעוד* בפורמט מיוחד, אשר ידועות יותר כ- doc comments. תוכנת העזר Javadoc מספקת דרך קלה ויעילה לתעד את ה-API שלך והשימוש בה נפוץ.

אם אינך בקי עדיין במוסכמות הערות התיעוד, עליך ללמוד אותן. למרות שמוסכמות אלו אינן חלק משפת Java, הן למעשה יוצרות API אשר כל מתכנת צריך לדעת. המוסכמות מוגדרות כ- [Javadoc - b] *The Javadoc Tool Home Page*.

כדי לתעד את ה-API שלך באופן נכון, עליך להקדים כל הצהרת מחלקה מיוצאת, ממשק, בנאי, שיטה ושדה בהערת תיעוד, בכפוף לחריג אחד שנדון בסוף נושא זה. בהיעדר הערת תיעוד, הדבר הטוב ביותר ש-Javadoc יכול לעשות הוא לייצר מחדש את ההצהרה כתיעוד היחיד למרכיב ה-API המושפע. שימוש ב-API שחסר הערות תיעוד הוא מתסכל ונוטה לטעויות. כדי לכתוב קוד בעל "חיים ארוכים", עליך לכתוב גם הערות תיעוד למחלקות, ממשקים, בנאים, שיטות ושדות לא מיוצאים.

הערת תיעוד עבור שיטה צריכה לתאר בתמציתיות את ההסכם בין השיטה והלקוח שלה. להוציא החריג של שיטות במחלקות שתוכננו להורשה (נושא 15), ההסכם צריך לומר **מה** השיטה עושה יותר מאשר **איך** השיטה עושה את עבודתה. הערת התיעוד צריכה למנות את כל **התנאים המוקדמים** של השיטה, אשר חייבים להתקיים כדי שלקוח יוכל לקרוא לשיטה. הערת התיעוד צריכה גם למנות את **התנאים המאוחרים**, שהם הדברים שיתרחשו לאחר שהקריאה לשיטה הושלמה בהצלחה. באופן טיפוסי, תנאים מוקדמים מתוארים במרומז על ידי תוויות @throws עבור חריגים לא מסומנים. כל חריג לא מסומן תואם להפרת תנאי מוקדם. כמו כן, אפשר לציין תנאים מוקדמים ביחד עם הפרמטרים המושפעים בתוויות @param שלהם.

בנוסף לתנאים מוקדמים ומאוחרים, שיטות צריכות לתעד כל **השפעה צדדית** שלהן. השפעה צדדית היא שינוי ניכר במצב המערכת אשר לא נדרש באופן מובן מאליו כדי לבצע את התנאי המאוחר. לדוגמה, אם שיטה מפעילה מטלת רקע, התיעוד צריך לציין זאת. לבסוף, הערות תיעוד צריכות לתאר את **בטיחות המטלה** של מחלקה, כפי שנדון בנושא 52.

כדי לתאר את ההסכם שלה באופן מלא, להערת תיעוד של שיטה צריכה להיות תווית @param לכל פרמטר, תווית @return אלא אם כן טיפוס הערך המוחזר של השיטה הוא void ותווית @throws עבור כל חריג שנזרק על ידי השיטה, בין אם הוא מסומן או לא מסומן (נושא 44). כמוסכמה, הטקסט הנלווה לתווית @throws צריך להיות מורכב מהמילה "if", שאחריה בא ביטוי שם עצם אשר מתאר את התנאים שבהם החריג יתעורר. לפעמים, ביטויים אריתמטיים משמשים במקום ביטויי שם עצם. כל המוסכמות האלו מודגמות בהערת התיעוד הקצרה הבאה, אשר באה מממשק List:

```
/ **
  * Returns the element at the specified position in this list.
  *
  * @param index index of element to return; must be
  *         nonnegative and less than the size of this list.
  * @return the element at the specified position in this list.
  * @throws IndexOutOfBoundsException if the index is out of range
  *         (<tt>index &lt;0 ||index &gt;=this.size()</tt>).
  * /
Object get(int index);
```

שים לב לשימוש בתווי מטא (meta) ובתגיות HTML בהערת תיעוד זו. תוכנת העזר Javadoc מתרגמת הערות תיעוד ל-HTML ומרכיבי HTML שרירותיים בהערת התיעוד מתבטאים במסמך ה-HTML המתקבל. לפעמים, מתכנתים הולכים כל כך רחוק עד לשיבוץ טבלאות HTML בהערות התיעוד שלהם, למרות שאין זה נפוץ. התגיות הכי שימושיות הן <p> כדי להפריד פסקאות, <code> ו-<tt> המשמשות לקטעי קוד ו-<pre> אשר משמשת לקטעי קוד ארוכים יותר.

התגיות <code> ו-<tt> דומות מאוד. התגית <code> יותר שימושית ולפי המפרט של HTML גירסה 4.01, היא בדרך כלל מועדפת בגלל ש-<tt> היא **מרכיב של סגנון גופן**. השימוש במרכיבי סגנון גופן מעודד על ידי לוחות סגנון [HTML401]. כלומר, ישנם מתכנתים אשר מעדיפים <tt> בגלל שהיא קצרה יותר ופחות מפריעה.

אל תשכח שרצפי דליפה נדרשים כדי לחולל תווי מטא של HTML, כמו סימן **הפחות מ-** (<), **סימן הגדול מ-** (<) ו**אמפרסנד** (&). כדי לחולל סימן **פחות מ-**, השתמש ברצף הדליפה "<". כדי לחולל סימן **גדול מ-**, השתמש ברצף הדליפה ">". כדי לחולל **אמפרסנד**, השתמש ברצף הדליפה "&". השימוש ברצפי דליפה מודגם בתווית @throws שבהערת התיעוד לעיל.

לבסוף, שים לב לשימוש במילה "this" בהערת התיעוד. כמוסכמה, כאשר המילה "this" משמשת בהערת התיעוד עבור שיטת מופע, היא תמיד מתייחסת לאובייקט שלגביו השיטה נקראת.

המשפט הראשון של כל הערת תיעוד הופך להיות **תיאור הסיכום** של המרכיב שההערה קשורה אליו. תיאור הסיכום חייב לעמוד בפני עצמו לתיאור התפקודיות של היישות שהוא מסכם. כדי למנוע בלבול, לשני איברים או בנאים במחלקה או בממשק לא יהיה אותו תיאור סיכום. הקדש תשומת לב מיוחדת להעמסות, כיון שלעיתים קרובות טבעי להשתמש באותו משפט ראשון בתיאור פרוזאי.

היזהר לא לכלול נקודה של סוף משפט במשפט הראשון של הערת תיעוד. אם כן תכלול, הדבר יסיים בטרם עת את תיאור הסיכום. לדוגמה, הערת תיעוד אשר פותחת ב- "A college degree, such as B.S, M.S., or Ph.D" תתבטא בתיאור סיכום של "A college degree, such as B.". הדרך הטובה ביותר להימנע מבעיה זו היא להימנע משימוש בקיצורים ובשברים עשרוניים בתיאורי סיכום. בכל אופן, אפשר לכלול נקודה בתיאור סיכום על ידי החלפת הנקודה ב**קידוד הנומרי** שלה, ".". למרות שזה עובד, קוד המקור שנוצר אינו יפה:

```
/ **
  * A college degree, such as B&#46;S&#46;, M&#46;S&#46; or
  * Ph&#46;D.
  * /
public class Degree { ... }
```

יהיה זה מטעה במקצת לומר שתיאור הסיכום הוא ה**משפט** הראשון בהערת תיעוד. מוסכמה מכתיבה שהוא צריך להיות משפט שלם לעיתים נדירות. עבור שיטות ובנאים, תיאור הסיכום צריך להיות ביטוי פועָל אשר מתאר את הפעולה שמבוצעת על ידי השיטה.

לדוגמה,

- ArrayList(int initialCapacity) — Constructs an empty list with the specified initial capacity.
- Collection.size() — Returns the number of elements in this collection.

עבור מחלקות, ממשקים ושדות, תיאור הסיכום צריך להיות ביטוי שם עצם אשר מתאר את הדבר אשר מיוצג על ידי מופע של המחלקה או הממשק או על ידי השדה עצמו.

לדוגמה,

- TimerTask — A task that can be scheduled for one-time or repeated execution by a Time.

- Math.PI — The double value that is closer than any other to pi, the ratio of the circumference of a circle to its diameter.

מוסכמות הערות התיעוד שתוארו בנושא זה הן חיוניות כדי להסתדר אבל יש עוד הרבה אחרות.
יש מספר מדריכי סגנון לכתיבת הערות תיעוד [Javadoc-a, Vermeulen00]. כמו כן, יש תוכנות עזר לבדיקת מידת ההקפדה על כללים אלה [Doclint].

מאז גירסה 1.2.2, ל-Javadoc יש יכולת ל"שימוש חוזר אוטומטי" או "לרשת" הערות של שיטה. אם לשיטה אין הערת תיעוד, Javadoc מחפש אחר הערת התיעוד המתאימה ביותר, תוך מתן עדיפות לממשקים על פני מחלקות על. אפשר למצוא את תוצאות אלגוריתם החיפוש ב- Javadoc Manual.

משמעות הדבר היא שמחלקות יכולות עתה להשתמש שוב בהערות התיעוד מממשקים שהן מממשות, על פני העתקת הערות אלו. לאמצעי זה יש את היכולת להפחית או להסיר את הנטל שבשמירת סטים מרובים של הערות תיעוד די דומות, אבל יש לזה הגבלה. הורשת הערות תיעוד היא הכל או כלום: השיטה היורשת לא יכולה להגדיר את הערת התיעוד המורשת בצורה כלשהי. אין זה בלתי מקובל לשיטה לייחד את ההסכם שהורש מממשק, במקרה שבו השיטה באמת צריכה הערת תיעוד משלה.

דרך פשוטה להפחית את הסבירות לטעויות בהערת תיעוד היא להריץ את קבצי ה-HTML שמיוצרים על ידי Javadoc על ידי **בודק תוקף HTML**. זה יאבחן הרבה טעויות שנובעות משימוש לא תקין בתגיות HTML, כמו גם בתווי מטא (meta) של HTML אשר היו צריכים להיעלם. כמה בודקי תוקף HTML זמינים להורדה, כמו [Weblint] weblint.

יש להוסיף אזהרה אחת בנוגע להערות תיעוד. למרות שהכרחי לספק הערות תיעוד לכל מרכיבי API המיוצאים, אין זה תמיד מספיק. לעיתים קרובות, הכרחי להוסיף את הערת התיעוד יחד עם תיעוד חיצוני אשר מתאר את הארכיטקטורה הכוללת של ה-API עבור APIs מסובכים, אשר מורכבים ממחלקות מרובות בעלות זיקה הדדית. אם קיים תיעוד כזה, הערות התיעוד של חבילת התוכנה או המחלקה הרלוונטיים צריכות לכלול קישור אליו.

לסיכום, הערות תיעוד הן הדרך הטובה והיעילה ביותר לתעד את ה-API שלך. השימוש בהן יכול להיחשב כמחויב המציאות לכל מרכיבי API מיוצאים. אמץ לך סגנון עקבי שדבק במוסכמות סטנדרטיות. זכור ש-HTML שרירותי מותר בהערות תיעוד וחובה לברוח מתווי מטא של HTML.

פרק 7

כללי תכנות כלליים

פרק זה מוקדש במידה רבה לאבני הבניין של שפת Java. הפרק דן בטיפול במשתנים מקומיים, השימוש בספריות, השימוש בטיפוסי נתונים שונים ובשימוש בשני אמצעים יוצאי דופן: השתקפות (**reflection**) ושיטות **native**. לבסוף, הפרק דן באופטימיזציה ובמוסכמות מתן שמות.

נושא 29: מזער את טווח ההכרה של משתנים מקומיים

נושא זה דומה בטבעו לנושא 12. על ידי מזעור טווח ההכרה של משתנים מקומיים, אתה מגדיל את הקריאות ואת יכולת התחזוקה של הקוד שלך ומפחית שכיחות של טעויות.

שפת C קובעת שחובה להצהיר על משתנים מקומיים בראשו של בלוק ומתכנתים ממשיכים לעשות זאת מתוך הרגל. זהו הרגל שכדאי לשבור אותו. לתזכורת, שפת Java מאפשרת לך להצהיר על משתנים בכל מקום שבו משפט הוא חוקי.

הטכניקה הכי רבת עוצמה למזעור טווח ההכרה של משתנים מקומיים היא להצהיר עליהם במקום הראשון שבו נעשה בהם שימוש. אם משתנה מוצהר לפני שנעשה בו שימוש, זה גורם לבלבול מיותר - סתם עוד פרט נוסף שמסיח את דעתו של מי שקורא את הקוד ומנסה להבין מה התוכנית עושה. עד לזמן שבו ייעשה שימוש במשתנה, ייתכן שהקורא לא יזכור כבר את טיפוס המשתנה או את ערכו ההתחלתי. אם התוכנית מתפתחת ולא נעשה יותר שימוש במשתנה, קל לשכוח להסיר את ההצהרה אם היא מוסרת בנקודה רחוקה מהנקודה בה נעשה בו שימוש לראשונה.

לא רק שהצהרת משתנה בטרם עת יכולה לגרום לטווח ההכרה שלו לגדול ולהתחיל יותר מדי מוקדם אלא הוא גם יכול לגדול יותר מדי מאוחר. טווח ההכרה של משתנה מקומי משתרע מהנקודה שבה הוא הוצהר ועד לסוף הבלוק שבו הוא מוגדר. אם משתנה מוגדר מחוץ לבלוק שבתוכו נעשה בו שימוש, הוא עדיין נראה לאחר שהתוכנית יוצאת מהבלוק הזה. אם בטעות נעשה שימוש במשתנה לפני או לאחר התחום שבו התכוונו להשתמש בו, התוצאות עלולות להיות הרות אסון.

כמעט כל הצהרה על משתנה מקומי צריכה להכיל מאתחל. אם עדיין אין לך מספיק מידע כדי לאתחל משתנה באופן הגיוני, עליך לדחות את ההגדרה עד שיהיה לך מידע שכזה. חריג אחד לכלל זה מתייחס למשפטי try-catch. אם משתנה מאותחל על ידי שיטה אשר מעוררת חריג בדוק, הוא חייב להיות מאותחל בתוך בלוק try. אם חייב להיעשות שימוש בערך מחוץ לבלוק try, אז הוא חייב להיות מוגדר לפני הבלוק try, מקום שבו הוא עדיין לא יכול להיות "מאותחל באופן הגיוני". לדוגמה, ראה עמוד 159.

לולאות מייצגות אפשרות מיוחדת למזעור טווח ההכרה של משתנים. לולאת for מאפשרת לך להצהיר על משתני לולאה, דבר אשר מגביל את טווח ההכרה שלהם לתחום המדויק שבו הם נדרשים (תחום זה מורכב מגוף הלולאה כמו גם מחלקי האתחול, הבדיקה והקידום אשר קודמים לגוף הלולאה). לכן, העדף לולאות **for** על פני לולאות **while**, בהנחה שתוכן משתנה/ים של הלולאה אינו נצרך לאחר שהלולאה מסתיימת.

לדוגמה, הנה הקוד המועדף לאיטרציה על פני אוסף:

```
for (Iterator i = c.iterator(); i.hasNext(); ) {
  doSomething(i.next());
}
```

כדי לראות מדוע לולאת for זו עדיפה על פני לולאת while, שלכאורה יותר הגיונית, קרא את פיסת הקוד הבאה, אשר מכילה שתי לולאות while ובאג אחד:

```
Iterator i = c.iterator();
while (i.hasNext()) {
  doSomething(i.next());
}
...
Iterator i2 =c2.iterator();
while (i .hasNext())  {     //BUG!
  doSomethingElse(i2.next());
}
```

הלולאה השנייה מכילה שגיאת גזור והדבק: היא מאתחלת משתנה לולאה חדש, i2, אבל משתמשת במשתנה הישן, i, אשר לרוע המזל עדיין נמצא בטווח ההכרה. הקוד המתקבל עובר הידור ללא שגיאות ורץ ללא עירור של חריג, אבל הוא עושה את הדבר הלא נכון. במקום לחזור שוב ושוב על c2, הלולאה השנייה מסתיימת באופן מיידי ונותנת את הרושם המוטעה ש-c2 ריק. בגלל שהתוכנית שוגה בשקט, הטעות יכולה להישאר לא מאובחנת לזמן ארוך.

אם הטעות האנלוגית גזור והדבק היתה מתרחשת בצירוף עם קוד לולאת for המועדף, הקוד המתקבל אפילו לא היה עובר הידור.

משתנה הלולאה מהלולאה הראשונה לא יהיה בטווח ההכרה בנקודה שבה מתרחשת הלולאה השנייה:

```
for (Iterator i =c.iterator(); i.hasNext(); ) {
  doSomething(i.next());
}
  ...

//Compile-time error -the symbol i cannot be resolved
for (Iterator i2 =c2.ite ator();i .hasNext();){
  doSomething(i2.next());
}
```

יתר על כן, אם תשתמש בקוד של לולאת ה-for, הרבה פחות סביר שתעשה את שגיאת הגזור והדבק, כיון שאין מניע להשתמש בשם משתנה שונה בשתי הלולאות. הלולאות עצמאיות לחלוטין, כך שאין שום נזק בשימוש מחדש בשמו של משתנה הלולאה. למעשה, זה אופנתי לנהוג כך.

לקוד של לולאת ה-for יש עוד יתרון אחד על פני הקוד של לולאת ה-while, אף על פי שהיתרון שולי. קוד לולאת ה-for יותר קצר בשורה אחת, מה שעוזר לשיטה שמכילה אותה להתאים לחלון עורך בגודל קצוב, דבר אשר משפר את הקריאות.

הנה קוד לולאה אחר המשמש למעבר על פני רשימה, אשר ממזער את טווח ההכרה של משתנים מקומיים:

```
//High-performance idiom for iterating over random access lists
for (int i =0, n =list.size();i <n;i++){
  doSomething(list.get(i));
}
```

הקוד שימושי למימושי List מגישה אקראית כמו ArrayList ו-Vector, מכיון שסביר יותר שהוא ירוץ מהר יותר מאשר "הקוד המועדף" שלעיל עבור רשימות כאלו. הדבר החשוב שיש להתייחס אליו בנוגע לקוד הזה הוא שיש לו שני משתני לולאה, i ו-n, שלשניהם יש אותו טווח ההכרה הנכון. השימוש במשתנה השני הוא חיוני לביצועים של הקוד. בלעדיו, הלולאה תצטרך לקרוא לשיטה size פעם אחת עבור כל חזרה, דבר שמנוגד ליתרון הביצועים של הקוד. שימוש בקוד זה מקובל כאשר אתה בטוח שהרשימה באמת מספקת גישה אקראית. אחרת, הוא יציג ביצועים פחותים.

קטעי קוד דומים קיימים עבור משימות מחזוריות אחרות, לדוגמה:

```
for (int i =0, n = expensiveComputation(); i < n; i++) {
  doSomething(i);
}
```

שוב, קטע קוד זה עושה שימוש בשני משתני לולאה והמשתנה השני, n, משמש לחיסכון של העלות של ביצוע חישוב מיותר בכל חזרה. בתור כלל, עליך להשתמש בקטע קוד זה אם הבדיקה שמתבצעת בלולאה מערבת קריאה לשיטה ומובטח שהקריאה לשיטה תחזיר את אותה התוצאה בכל חזרה.

טכניקה אחרונה למזעור טווח ההכרה של משתנים מקומיים היא לשמור שיטות קטנות וממוקדות. אם אתה משלב שתי פעילויות באותה השיטה, משתנים מקומיים שקשורים לפעילות אחת יכולים להיות בטווח ההכרה של הקוד שמבצע את הפעילות האחרת. כדי למנוע זאת, פשוט הפרד את השיטה לשניים: שתי שיטות לכל פעילות.

נושא 30: הכר והשתמש בספריות הקוד של Java

נניח שאתה רוצה לחולל מספרים שלמים אקראיים בין 0 לבין גבול עליון כלשהו. מתכנתים רבים עשויים לכתוב שיטה קטנה לפתרון הבעיה הזאת אשר נראית בערך כמו הדבר הבא:

```
static Random rnd = new Random();

   // Common but flawed!
   static int random(int n) {
      return Math.abs(rnd.nextInt()) % n;
   }
```

השיטה אינה רעה אבל אינה מושלמת. להיפך, יש בה שלושה ליקויים. הליקוי הראשון הוא שאם n הוא חזקה קטנה של שתיים, הרצף של המספרים האקראיים שהשיטה מחוללת יחזור על עצמו לאחר פרק זמן קצר למדי. הליקוי השני הוא שאם n הוא לא חזקה של שתיים, מספרים מסוימים, בממוצע, יחזרו על עצמם בתדירות גבוהה יותר מאשר מספרים אחרים. אם n הוא גדול, ליקוי זה יכול להיות בולט למדי. הדבר מודגם באופן גרפי על ידי התוכנית הבאה, אשר מחוללת מיליון מספרים אקראיים בטווח שנבחר בקפידה ואחר כך מדפיסה כמה מהמספרים נמצאים בחצי התחתון של הטווח:

```
public static void main(String[] args) {
      int n = 2 * (Integer.MAX_VALUE / 3);
      int low = 0;
      for (int i = 0; i < 1000000; i++)
         if (rnd.nextInt(n) < n/2)
            low++;

      System.out.println(low);
   }
```

אם שיטת random עבדה כראוי, התוכנית תדפיס מספר קרוב לחצי מיליון, אבל אם תריץ אותה, תגלה שהיא מדפיסה מספר קרוב ל- 666,666. שני שלישים מהמספרים שחוללה שיטת random נמצאים בחציו התחתון של הטווח!

הליקוי השלישי בשיטת random הוא שהיא יכולה, במקרה נדיר, ליפול בצורה נוראה, כשהיא מחזירה מספר מחוץ לטווח המבוקש. הסיבה היא שהשיטה מנסה למפות את הערך שמוחזר על ידי rnd.nextInt() לתוך מספר שלם לא שלילי בעזרת Math.abs. אם nextInt() מחזירה Integer.MIN_VALUE, Math.abs גם תחזיר Integer.MIN_VALUE, ואופרטור השארית (%) יחזיר מספר שלילי, בהנחה ש-n הוא לא חזקה של שתיים.

הדבר יגרום לתוכנית שלך לקרוס כמעט בוודאות והכישלון עשוי להיות קשה לשחזור.

כדי לכתוב גירסה של random אשר מתקנת שלושה ליקויים אלה, צריך להיות ברשותך ידע רב בנוגע למחוללי מספרים אקראיים מדומים חופפים, תיאוריית המספרים ואריתמטיקה של המשלים לשתיים. למרבה המזל, אינך צריך לדעת כל זאת - הדבר כבר נעשה עבורך. זה נקרא Random.nextInt(int) וזה התווסף לחבילת התוכנה הסטנדרטית java.util בגירסה 1.2.

אינך צריך לדאוג בנוגע לפרטים כיצד nextInt(int) מבצעת את עבודתה (למרות שאתה יכול ללמוד את התיעוד או את קוד המקור אם אתה סקרן באופן חולני). תלמיד השנה האחרונה בפקולטה להנדסה עם רקע באלגוריתמים השקיע זמן לא מועט כדי לתכנן, לממש ולבחון את השיטה הזאת ואז הראה אותה למומחים בתחום כדי להבטיח שהיא נכונה. הספריה עברה מבחן כספריית בטא, הופצה ונוסתה באופן נרחב על ידי אלפי מתכנתים במשך כמה שנים. אף ליקוי לא התגלה עדיין בשיטה, אבל אם היה מתגלה ליקוי, הוא היה אמור להיות מתוקן בגירסה הבאה. **בשימוש בספריה סטנדרטית אתה מנצל את הידע של מומחים אשר כתבו אותה ואת הניסיון של אלה אשר השתמשו בה לפניך**.

יתרון שני של שימוש בספריות הוא שאינך צריך לבזבז את זמנך בכתיבת פתרונות אד הוק לבעיות שקשורות באופן שולי בלבד לעבודתך. אם אתה כמו מרבית המתכנתים, עדיף שתקדיש את זמנך בעבודה על היישום שלך מאשר על הצנרת שמתחת לפני השטח.

יתרון שלישי לשימוש בספריות סטנדרטיות הוא שהביצועים שלהן נוטים להשתפר במשך הזמן, ללא שום מאמץ מצידך. בגלל שהרבה אנשים משתמשים בהן ובגלל שהן משמשות כבוחן ביצועים בסטנדרט תעשייתי, לארגונים אשר מספקים ספריות אלו יש תמריץ חזק לגרום להן לפעול מהר יותר. לדוגמה, הספריה המתמטית, java.math, נכתבה מחדש בגירסה 1.3, בעקבות שיפורים דרמטיים בביצועים.

ספריות נוטות גם להשיג תפקודיות חדשה במשך הזמן. אם מחלקת ספריה חסרה תפקודיות חשובה כלשהי, קהילת המתפתחים תפרסם ליקוי זה. פלטפורמת Java תמיד התפתחה תוך משוב איתן מקהילה זו.
בעבר התהליך לא היה רשמי. כעת, יש במקום זה תהליך רשמי שנקרא תהליך הקהילה של Java (Java Community Process - JCP). בכל אופן, קיימת נטייה להוסיף במשך הזמן תכונות שהיו חסרות.

יתרון אחרון של שימוש בספריות הסטנדרטיות הוא שהקוד שלך עושה שימוש בכלים שרוב המתכנתים משתמשים בהם. קוד שכזה הוא יותר קריא, ארוך חיים יותר וניתן לשימוש חוזר על ידי ציבור רחב של מפתחים.

לאור כל היתרונות האלו, נראה הגיוני להשתמש באמצעים של ספריות על פני מימושי אד הוק, אבל עדיין חלק ניכר של מתכנתים לא עושה זאת.

מדוע? אולי הם לא יודעים שאמצעי הספריות קיימים. **תכונות רבות מתווספות לספריות בכל גירסה עיקרית ומשתלם לדעת את ההוספות האלו**. אתה יכול לקרוא בעיון את התיעוד באופן מקוון או לקרוא על הספריות בכל מספר של ספרים [J2SE-APIs, Chan00, Flanagan99, Chan98]. הספריות גדולות מדי מכדי ללמוד את כל התיעוד, אבל כל מתכנת צריך להיות בקי בתוכן של **java.lang**, **java.util** ובהיקף קטן יותר, **java.io**. ידע בספריות אחרות יכול להירכש בהתאם לצורך מסוים.

סיכום של כל היכולות בספריות הוא מעבר לתחומו של נושא זה, אבל כמה מהן ראויות להתייחסות מיוחדת. בגירסה 1.2, Collections Framework התווספה לחבילת התוכנה java.util. היא צריכה להיות חלק מערכת הכלים הבסיסית של כל מתכנת. ה- Collections Framework היא ארכיטקטורה מאוחדת לייצוג ולהפעלת אוספים, אשר מאפשרת להם להיות מופעלים ללא תלות בפרטים של הייצוג שלהם. הדבר מפחית מאמצי תכנות ומשפר את הביצועים. היא מאפשרת פעולה הדדית בין שני API לא קשורים, מפחיתה את המאמצים הדרושים לתכנון וללימוד API חדשים ומעודדת שימוש מחדש בתוכנה.

סביבת העבודה מבוססת על שישה ממשקי אוסף (Collection, Set, List, Map, SortedList ו-SortedMap). היא כוללת מימושים של ממשקים אלה ואלגוריתמים להפעלה שלהם. מחלקות האוסף הנורשות, Vector ו-Hashtable, שופרו כך שתיקחנה חלק בסביבת העבודה, לכן אינך צריך לנטוש אותן כדי לנצל את סביבת העבודה.

סביבת העבודה של האוספים מפחיתה ממש את כמות הקוד אשר דרושה להרבה משימות שכיחות. לדוגמה, נניח שיש לך וקטור של מחרוזות ואתה רוצה למיין אותו באופן אלפביתי. שורה יחידה זו עושה את העבודה:

```
Collections.sort(v);
```

אם אתה רוצה לבצע את אותו המיון אבל ללא התייחסות לגודל אותיות, השתמש בקוד הבא:

```
Collections.sort(v, String.CASE_INSENSITIVE_ORDER);
```

נניח שאתה רוצה להדפיס את כל האלמנטים שבמערך. הרבה מתכנתים משתמשים בלולאת for, אבל אין שום צורך בכך אם תשתמש בשורה הבאה:

```
System.out.println(Arrays.asList(a));
```

לבסוף, נניח שאתה רוצה לדעת את כל המפתחות שעבורם שני מופעי Hashtable, h1 ו-h2 מכילים את אותו המיפוי.

לפני ש- Collections Framework התווספה, היית נדרש לכמות גדולה של קוד, אבל כעת דרושות לכך שלוש שורות:

```
Map tmp =new HashMap(h1);
tmp.entrySet().retainAll(h2.entrySet());
Set result =tmp.keySet();
```

הדוגמאות שלעיל בקושי מגרדות את פני השטח של מה שאתה יכול לעשות עם Collections Framework. אם אתה רוצה לדעת עוד, עיין בתיעוד שבאתר האינטרנט של חברת Sun [Collections] או קרא את המדריך [Bloch99].

ספריית צד שלישי שכדאי לציין היא util.concurrent של Doug Lea [Lea01], אשר מספקת תוכניות שירות שיתופיות ברמה גבוהה כדי לפשט את המשימה של תכנות מרובה מטלות.

יש הרבה הוספות לספריות בגירסה 1.4. תוספות ראויות לציון כוללות את הבאות:

- **java.util.regex** - אמצעי ביטוי רגיל דומה לשפת פרל (Perl) במלוא פריחתה.
- **java.util.prefs** - אמצעי לאחסון מתמיד של העדפות משתמש ונתוני תצורת תוכנית.
- **java.nio** - אמצעי I/O עתיר ביצועים הכולל I/O הניתן לשינוי גודל (scalable) (מקביל לקריאת poll ב-Unix) ו-I/O ממופה זיכרון (מקביל לקריאת mmap ב-Unix).
- **java.util.LinkedHashSet ,LinkedHashMap ,IdentityHashMap** - מימושי אוסף חדשים.

לפעמים, אמצעי של ספריה לא תתאים לצרכיך. ככל שתמקד את צרכיך, כך סביר יותר שהדבר יקרה. למרות שהדחף הראשון שלך צריך להיות שימוש בספריות, אם הסתכלת במה שיש להן להציע בתחום מסוים ואין זה תואם את צרכיך, השתמש במימוש חילופי. תמיד יהיו חורים בתפקודיות שמסופקת על ידי סט מוגבל כלשהו של ספריות. אם התפקודיות שאתה זקוק לה חסרה, ייתכן שאין לך שום ברירה אלא לממש אותה בעצמך.

לסיכום, אל תמציא את הגלגל מחדש. אם אתה צריך לבצע משימה אשר נראית מקובלת, ייתכן שיש כבר מחלקה בספריות אשר מבצעת את מה שאתה צריך. אם יש, השתמש בה. אינך יודע אם יש - בדוק. בדרך כלל, סביר שקוד ספריה יהיה טוב יותר מאשר קוד שכתבת בעצמך וסביר גם שהוא ישתפר עם הזמן. אין זה אומר שיכולותיך כמתכנת הן חסרות. בדיקות מראות שקודי ספריה מקבלים הרבה יותר תשומת לב מאשר זאת שיכול היה המתכנת הממוצע להקדיש לאותה התפקודיות.

נושא 31: הימנע מ- float ומ- double אם נדרשות תשובות מדויקות

טיפוסי float ו-double תוכננו באופן ראשוני לחישובים מדעיים והנדסיים. הם מבצעים אריתמטיקת נקודה צפה בינארית, אשר תוכננה בקפדנות כדי לספק קירוב מדויק במהירות על פני טווח רחב של גדלים. בכל אופן, הם לא מספקים תשובות מדויקות ואין להשתמש בהם היכן שנדרשות תשובות מדויקות. הטיפוסים **float** ו-**double** אינם מתאימים במיוחד עבור חישובים כספיים, מכיון שאי אפשר לייצג 0.1 (או כל חזקה שלילית אחרת של עשר) כ-float או כ-double במדויק.

לדוגמה, נניח שיש לך $1.03 בכיסך והוצאת ¢42. כמה כסף נותר לך? הנה קטע תוכנית נאיבי אשר מנסה לענות על השאלה:

```
System.out.println(1.03 - .42);
```

לרוע המזל, היא מדפיסה 0.06100000000000001. אין זה מקרה בודד. נניח שיש לך דולר אחד בכיסך ואתה קונה תשע סוכריות שמחירן עשרה סנטים האחת. כמה עודף תקבל?

```
System.out.println(1.00 - 9*.10);
```

לפי קטע התוכנית הזו תקבל $0.09999999999999995. אתה עשוי לחשוב שאפשר פשוט לפתור את הבעיה על ידי עיגול התוצאה לפני הדפסתה, אבל לרוע המזל אין זה תמיד עובד. לדוגמה, נניח שיש לך דולר אחד בכיסך ואתה רואה מדף עם שורת ממתקים טעימים במחירים של ¢10, ¢20, ¢30 וכן הלאה, עד לדולר. כמה ממתקים תוכל לקנות וכמה עודף תקבל אם תתחיל בממתק שעולה ¢10 וכן הלאה? הנה תוכנית תמימה אשר תוכננה לפתרון בעיה זו:

```
// Broken - uses floating point for monetary calculation!
public class InappropriateFloat {
    public static void main(String[] args) {
        double funds = 1.00;
        int itemsBought = 0;
        for (double price = .10; funds >= price; price += .10) {
            funds -= price;
            itemsBought++;
        }
        System.out.println(itemsBought + " items bought.");
        System.out.println("Change: $" + funds);
    }
}
```

אם תריץ את התוכנית, תגלה שאתה יכול לקנות שלושה ממתקים ותקבל עודף של $0.3999999999999999. אבל תשובה זו לא נכונה! הדרך הנכונה לפתרון הבעיה היא להשתמש ב-**BigDecimal**, **int** או **long** עבור חישובים כספיים. הנה גירסה משופרת של התוכנית הקודמת שעושה שימוש ב-BigDecimal במקום ב-double:

```
public static void main(String[] args) {
      final BigDecimal TEN_CENTS = new BigDecimal( ".10");

      int itemsBought = 0;
      BigDecimal funds = new BigDecimal("1.00");
      for (BigDecimal price = TEN_CENTS;
           funds.compareTo(price) >= 0;
           price = price.add(TEN_CENTS)) {
         itemsBought++;
         funds = funds.subtract(price);
      }
      System.out.println(itemsBought + " items bought.");
      System.out.println("Money left over: $" + funds);
   }
}
```

אם תריץ את התוכנית המשופרת, תגלה שאתה יכול לקנות ארבע יחידות של ממתקים ותקבל עודף של $0.00. זו התשובה הנכונה. בכל אופן, יש שני חסרונות לשימוש ב-BigDecimal: זה פחות נוח משימוש בטיפוס אריתמטי פשוט וזה איטי יותר. החיסרון השני פחות מורגש כאשר אתה פותר בעיה קטנה יחידה אבל הבעיה השנייה עשויה להטריד אותך.

כחלופה לשימוש ב-BigDecimal אתה יכול להשתמש ב-int או ב-long (תלוי באילו סכומים מדובר) ולשמור מעקב על הנקודה העשרונית בעצמך. בדוגמה זו, הגישה המובנת מאליה היא לבצע את כל החישובים בסנטים במקום בדולרים. הנה גירסה משופרת של התוכנית שזה עתה הוצגה אשר נוקטת בגישה זו:

```
public static void main(String[] args) {
   int itemsBought = 0;
   int funds = 100;
   for (int price = 10; funds >= price; price += 10) {
      itemsBought++;
      funds -= price;
   }
   System.out.println(itemsBought + " items bought.");
   System.out.println("Money left over: "+ funds + " cents");
}
```

לסיכום, אל תשתמש ב-float או ב-double לחישובים כלשהם אשר דורשים תשובה מדויקת. השתמש ב-BigDecimal אם אתה רוצה שהמערכת תשמור מעקב על הנקודה העשרונית ואתה מוכן לקבל את אי הנוחות הכרוכה באי שימוש בטיפוס פשוט.
לשימוש ב-BigDecimal יש את היתרון המוסף של שליטה מלאה על עיגול, תוך מתן אפשרות בחירה מבין שמונה אופני עיגול בכל פעם שמתבצעת פעולה אשר דורשת עיגול. הדבר נוח אם אתה מבצע חישובים עסקיים עם התנהגות עיגול חוקית.
אם הביצועים הם מהותיים, אם לא אכפת לך לשמור מעקב על הנקודה העשרונית בעצמך ואם הגדלים המתמטיים אינם גדולים מדי, השתמש ב-int או ב-long. אם הגדלים אינם עולים על תשע ספרות דצימליות, אתה יכול להשתמש ב-int. אם הם אינם עולים על שמונה עשרה ספרות, אתה יכול להשתמש ב-long. אם הגדלים עולים על שמונה עשרה ספרות אתה חייב להשתמש ב-BigDecimal.

נושא 32: הימנע ממחרוזות כאשר טיפוסים אחרים יותר מתאימים

מחרוזות תוכננו לייצג טקסט והן עושות זאת בצורה טובה. בגלל שמחרוזות הן כל כך מקובלות וכל כך נתמכות טוב על ידי השפה, קיימת נטייה טבעית להשתמש במחרוזות למטרות אחרות מאלו שלהן הן תוכננו. נושא זה דן בכמה דברים שאינך צריך לעשות עם מחרוזות.

מחרוזות הן תחליפים עלובים לטיפוסי ערכים אחרים. כאשר פיסת נתונים מגיעה לתוכנית מקובץ, מרשת תקשורת או מקלט שמקורו במקלדת, היא לעיתים קרובות בצורה של מחרוזת. יש נטייה טבעית להשאיר את הנתונים בצורה זו, אבל נטייה זו מוצדקת רק אם הנתונים הם טקסטואליים מטבעם. אם הנתונים הם נומריים, הם צריכים להיות מתורגמים לטיפוס הנומרי המתאים, כמו int, float או BigInteger. אם הם תשובה לשאלת כן או לא, הם צריכים להיות מתורגמים ל-boolean. באופן יותר כללי, אם יש טיפוס ערך מתאים, בין אם הוא ייחוס אובייקט או ערך פשוט, עליך להשתמש בו. אם אין טיפוס ערך מתאים, אתה צריך לכתוב אחד כזה. למרות שעצה זו עשויה להיראות מובנת מאליה, היא מופרת לעיתים קרובות.

מחרוזות הן תחליפים עלובים לטיפוסים נספרים. כפי שנידון בנושא 21, הן ערכי int והן typesafe enums, מתאימים כקבועי טיפוס נספרים הרבה יותר טובים מאשר מחרוזות.

מחרוזות הן תחליפים עלובים לטיפוסי צירוף. אם ליישות יש מרכיבים מרובים, בדרך כלל זהו רעיון רע לייצג אותם כמחרוזות בודדות. לדוגמה, הנה שורת קוד אשר מגיעה ממערכת אמיתית - השמות המזהים שונו כדי להגן מאשמה:

```
// Inappropriate use of string as aggregate type
String compoundKey = className + "#" + i.next();
```

לגישה זו חסרונות רבים. אם התו אשר משמש להפרדת שדות יופיע במקרה באחד מהשדות, עשויה להיגרם אנדרלמוסיה. כדי לגשת לשדות אינדיבידואליים, אתה צריך לפצל את המחרוזת, מעשה שהוא איטי, מעייף ועם נטייה לטעויות. אתה לא יכול לספק שיטות equals, toString או compareTo אבל אתה נאלץ לקבל את ההתנהגות ש-String מספקת. גישה טובה יותר היא פשוט לכתוב מחלקה לייצוג הצירוף, לעיתים קרובות תהיה זו מחלקת איבר סטטית פרטית (נושא 18).

מחרוזות הן תחליפים עלובים ליכולות (capabilities). לעיתים, מחרוזות משמשות כדי לתת גישה לתפקודיות מסוימת. לדוגמה, חשוב על התכנון של שרות המספק משתנים למטלות כך שלכל מטלה יש את הערך שלה.

לפני מספר שנים, כמה אנשים ניסו לתכנן אמצעי כזה באופן עצמאי על ידי התכנון שבו מפתחות מחרוזת שמסופקות על ידי הלקוח נתנו גישה לתוכן של משתנה מטלה מקומי:

```
//Broken -inappropriate use of String as capability!
public class ThreadLocal {
  private Thread Local()  {  }   //Noninstantiable

  //Sets the current thread's value for the named variable.
  public static void set(String key,Object value);

  //Returns the current thread's value for the named variable.
  public static Object get(String key);
}
```

הבעיה בגישה זו היא שהמפתחות מייצגים מרחב שם שמשותף באופן גלובלי. אם שני לקוחות עצמאיים של חבילת התוכנה מחליטים להשתמש באותו השם עבור משתנה המטלה המקומית שלהם, הם חולקים את המשתנה ביחד בלי להתכוון לכך, דבר שבדרך כלל יגרום לנפילה של שני הלקוחות. כמו כן, האבטחה היא דלה. לקוח בעל כוונות זדון יכול להשתמש באותו המפתח כמו לקוח אחר כדי להשיג גישה לא חוקית לנתוניו של הלקוח האחר.

אפשר לתקן את ה-API הזה על ידי החלפת המחרוזת במפתח בלתי ניתן לזיוף (שלעיתים מכונה **יכולת - capability**):

```
public class Thread Local {
  private Thread Local()   {    }    //Noninstantiable

  public static class Key {
    Key()  {   }
  }

  //Generates a unique, unforgeable key
  public static Key getKey(){
    return new Key();
  }

  public static void set(Key key,Object value);
  public static Object get(Key key);
}
```

למרות שזה פותר את שתי הבעיות של ה-API מבוסס המחרוזת, אתה יכול לעשות יותר מכך. אינך צריך באמת את השיטות הסטטיות יותר. במקום זאת הן יכולות להיהפך לשיטות מופע של המפתח, באותה הנקודה שבה המפתח כבר אינו מפתח: הוא משתנה מטלה מקומי. בנקודה זו, המחלקה ה-noninstantiable מהרמה הגבוהה ביותר לא

עושה יותר שום דבר בשבילך, לכן אתה יכול גם להיפטר ממנה ולשנות את שם המחלקה המקוננת ל-ThreadLocal:

```
public class ThreadLocal {
    public ThreadLocal(){}
    public void set(Object value);
    public Object get();
}
```

אם לנסח זאת באופן מחוספס, זהו ה-API ש-java.util.ThreadLocal מספקת. בנוסף לפתרון הבעיות עם ה-API מבוסס המחרוזת, הוא מהיר יותר ואלגנטי יותר מאשר כל אחד מה-API מבוססי המפתח.

לסיכום, הימנע מהנטייה הטבעית לייצג אובייקטים כמחרוזות כאשר טיפוסי נתונים טובים יותר קיימים או יכולים להיכתב. מחרוזות שנעשה בהן שימוש לא נכון הן מגושמות, פחות גמישות, איטיות יותר ובעלות נטייה לטעויות מאשר טיפוסים אחרים. בדרך כלל, מחרוזות משמשות באופן מוטעה כטיפוסים פשוטים, טיפוסים נספרים וטיפוסי צירוף.

נושא 33: היזהר משרשור מחרוזות

אופרטור שרשור המחרוזות (+) הוא דרך נוחה לצרף כמה מחרוזות לתוך מחרוזת אחת. הדבר טוב כאשר רוצים ליצור שורת פלט יחידה או לבניית הייצוג המחרוזתי של אובייקט קטן בעל גודל קבוע, אבל זה איטי. שימוש באופרטור שרשור המחרוזות באופן חוזר כדי לשרשר **n** מחרוזות מצריך זמן בחזקת **n**. זוהי תוצאה חסרת מזל של העובדה שמחרוזות הן בלתי משתנות (נושא 13). כאשר שתי מחרוזות משורשרות, התוכן של שתיהן מועתק.

לדוגמה, התבונן בשיטה הבאה אשר בונה ייצוג מחרוזת של חשבונית תוך שרשור חוזר של שורה עבור כל פריט:

```
// Inappropriate use of string concatenation - Performs horribly!
  public String statement() {
    String s = "";
    for (int i = 0; i < numItems(); i++)
      s += lineForItem(i);  // String concatenation
    return s;
  }
```

שיטה זו איטית להחריד אם מספר הפריטים הוא גדול. כדי להשיג ביצועים מתקבלים על הדעת, השתמש ב-**StringBuffer** במקום ב-**String** כדי לשמור את המשפט תחת בנייה:

```
public String statement( )  {
    StringBuffer s = new StringBuffer(numItems() * LINE_WIDTH);
    for (int i = 0; i < numItems(); i++)
      s.append(lineForItem(i));
    return s.toString();
  }
```

ההבדל בביצועים הוא דרמטי. אם numItems מחזיר 100 ו-lineForItem מחזיר מחרוזת קבועה בעלת 80 תווים, השיטה השנייה מהירה במחשב שלי פי 90 מהשיטה השנייה. בגלל שהשיטה הראשונה בעלת סיבוכיות ריבועית והשיטה השנייה בעלת סיבוכיות לינארית, הבדל הביצועים הוא אפילו יותר דרמטי עבור מספרים גדולים יותר של פריטים. שים לב שהשיטה השנייה מקצה מראש StringBuffer גדול מספיק כדי להכיל את התוצאה. אפילו אם השיטה מכוונת מחדש כך שתשתמש ב-StringBuffer בגודל ברירת מחדל, עדיין היא מהירה פי 45 מהשיטה הראשונה.

הלקח הוא פשוט: אל תשתמש באופרטור שרשור המחרוזות כדי לצרף יותר מאשר כמה מחרוזות אלא אם כן הביצועים אינם חשובים. השתמש במקום זה בשיטת append של StringBuffer. כחלופה, השתמש במערך תווים או עבד את המחרוזות בנפרד במקום לצרף אותן.

נושא 34: התייחס לאובייקטים בעזרת הממשקים שלהם

נושא 25 מכיל את העצה לפיה אתה צריך להשתמש בממשקים יותר מאשר במחלקות כטיפוסי פרמטרים. באופן יותר כללי, אתה צריך להעדיף את השימוש בממשקים על פני מחלקות כדי להתייחס לאובייקטים. אם קיימים טיפוסי ממשק מתאימים - על פרמטרים, ערכים מוחזרים, משתנים ושדות, צריך להצהיר על ידי שימוש בטיפוסי ממשק. הפעם היחידה שבה אתה באמת צריך להתייחס למחלקה של אובייקט היא כאשר אתה יוצר אותה. כדי להבהיר זאת, חשוב על המקרה של Vector, אשר הוא מימוש של ממשק List. התרגל להקליד את זה:

```
// Good - uses interface as type
List subscribers = new Vector();
```

במקום את זה:

```
// Bad - uses class as type!
Vector subscribers = new Vector();
```

אם תתרגל להשתמש בממשקים בתור טיפוסים, התוכנית שלך תהיה הרבה יותר גמישה. אם תחליט שאתה רוצה להחליף מימושים, כל מה שאתה צריך לעשות הוא לשנות את שם המחלקה בבנאי (או להשתמש ביצרן סטטי שונה). לדוגמה, ההצהרה הראשונה יכולה היתה להשתנות לקריאה

```
List subscribers = new ArrayList();
```

וכל הקוד שמסביב ימשיך לעבוד. הקוד שמסביב לא היה מודע לטיפוס המימוש הישן, כך שהוא לא יחוש בשינוי.

יש אזהרה אחת: אם המימוש הציע איזו תפקודיות מיוחדת אשר לא נדרשת על ידי ההסכם של הממשק והקוד הסתמך על תפקודיות זו, אז זה יהיה קריטי שהמימוש החדש יספק את אותה התפקודיות. לדוגמה, אם הקוד שמסביב להצהרה הראשונה היה תלוי בעובדה ש-Vector היא מסונכרנת, אז יהיה זה לא נכון להחליף את ArrayList ב-Vector בהצהרה.

אם כן, מדוע שתרצה לשנות מימושים? בגלל שהמימוש החדש מציע ביצועים טובים יותר או בגלל שהוא מציע תפקודיות נוספת רצויה. דוגמה מהעולם האמיתי נוגעת למחלקת ThreadLocal. באופן פנימי, מחלקה זו משתמשת בשדה Map עם הרשאה פרטית-בחבילה ב-Thread כדי לקשר ערכים ייחודיים לכל מטלה עם מופעי ThreadLocal. בגירסה 1.3, שדה זה אותחל למופע HashMap.

בגירסה 1.4, מימוש Map חדש למטרה מיוחדת, בשם IdentityHashMap, התווסף לפלטפורמה. על ידי שינוי שורת קוד בודדת כדי לאתחל את השדה ל-IdentityHashMap במקום HashMap, האמצעי ThreadLocal נעשה מהיר יותר.

לאחר שהשדה הוצהר כ-HashMap במקום Map, אין שום הבטחה ששינוי שורה בודדת יספיק. אם הקוד של הלקוח השתמש בפעולות HashMap מחוץ לממשק Map או העביר את המפה לשיטה אשר דורשת HashMap, הקוד לא יוכל לעבור יותר הידור אם השדה השתנה ל-IdentityHashMap. הצהרת השדה עם טיפוס הממשק, "שומרת אותך הוגן".

מתאים לחלוטין להתייחס לאובייקט על ידי מחלקה ולא על ידי ממשק אם לא קיים כל ממשק מתאים. לדוגמה, חשוב על מחלקות ערך, כמו String ו-BigInteger. מחלקות ערך נכתבות לעיתים נדירות עם כוונה למימושים מרובים. לרוב הן סופיות ונדיר שיש להן ממשקים מקבילים. מתאים לגמרי להשתמש במחלקת ערך כפרמטר, משתנה, שדה או ערך מוחזר. באופן יותר כללי, אם למחלקה מוצקה אין שום ממשק מקושר, אז אין לך שום ברירה אלא להתייחס אליה על ידי המחלקה שלה בין אם היא מייצגת ערך או לא. מחלקת Random משתייכת לקטגוריה הזאת.

מקרה שני שבו אין שום טיפוס ממשק מתאים הוא זה שבו אובייקטים משתייכים לסביבת עבודה שהטיפוסים היסודיים שלה הן מחלקות ולא ממשקים. אם אובייקט משתייך לסביבת עבודה מבוססת מחלקה, עדיף להתייחס אליו על ידי מחלקת הבסיס הרלוונטית, שבאופן טיפוסי היא מופשטת, ולא על ידי מחלקת המימוש שלו. מחלקת java.util.TimerTask משתייכת לקטגוריה זו.

מקרה שלישי שבו אין שום טיפוס ממשק מתאים הוא של מחלקות אשר מממשות ממשק אבל מספקות שיטות נוספות אשר לא מצויות בממשק. לדוגמה, LinkedList. במחלקה שכזו צריך לעשות שימוש רק כדי להתייחס למופעים שלה אם התוכנית מסתמכת על השיטות הנוספות; לעולם אין להשתמש בה כטיפוס פרמטר (נושא 25).

מקרים אלה אינם מתיימרים להיות ממצים אלא פשוט לתת מעט מושג על מצבים שבהם מתאים להתייחס לאובייקט על ידי המחלקה שלו. למעשה, צריך להיות גלוי האם לאובייקט נתון יש ממשק מתאים. אם אכן יש לו, התוכנית שלך תהיה יותר גמישה אם תשתמש בממשק כדי להתייחס לאובייקט. אם אין לו, אז פשוט השתמש במחלקה הגבוהה ביותר בהיררכיית המחלקות אשר מספקת את התפקודיות הנדרשת.

נושא 35: העדף ממשקים על פני השתקפות (Reflaction)

אמצעי ההשתקפות java.lang.reflect, מציע גישה תכנותית למידע על מחלקות טעונות. בהינתן מופע Class, אתה יכול לקבל מופעי Constructor, Method ו-Field אשר מייצגים את הבנאים, השיטות והשדות של המחלקה אשר מיוצגת על ידי מופע ה-Class. אובייקטים אלה מספקים גישה תכנותית לשמות איבר המחלקה, טיפוסי שדה, חתימות שיטות וכן הלאה.

יתר על כן, מופעי Constructor, Method ו-Field מאפשרים לך להפעיל את החלקים המקבילים הנסתרים שלהם באופן משקף: אתה יכול לבנות מופעים, לקרוא לשיטות ולגשת לשדות של המחלקה הנסתרת על ידי קריאה לשיטות על מופעי Constructor, Field ו-Method. לדוגמה, Method.invoke מאפשרת לך לקרוא לכל שיטה על כל אובייקט של כל מחלקה (בכפוף לאילוצי אבטחה). השתקפות מאפשרת למחלקה אחת להשתמש בשנייה, אפילו אם המחלקה השנייה לא היתה קיימת כאשר המחלקה הראשונה עברה הידור. בכל אופן, לעוצמה זו יש מחיר:

- **אתה מפסיד את כל הרווח שבבדיקות טיפוס זמן הידור, כולל בדיקות חריג**. אם תוכנית מנסה לקרוא לשיטה בלתי נגישה או בלתי קיימת באופן משקף, היא תיפול בזמן ריצה אלא אם כן נקטת באמצעי זהירות מיוחדים.
- **הקוד שנדרש לביצוע גישת השתקפות הוא מגושם ומרובה מילים**. מעייף לכתוב אותו וקשה לקרוא אותו.
- **הביצועים ירודים**. בגירסה 1.3, קריאת שיטה באופן משקף היתה איטית פי ארבעים במחשב שלי מאשר קריאת שיטה באופן רגיל. בגירסה 1.4, השתקפות עוצבה מחדש לביצועים משופרים לאין שיעור, אבל היא עדיין איטית פי שניים מגישה רגילה ולא סביר שהפער יצטמצם.

אמצעי ההשתקפות תוכנן במקורו עבור כלי פיתוח יישום מבוססי רכיבים. כלים כאלה בדרך כלל טוענים מחלקות לפי דרישה ומשתמשים בהשתקפות כדי לגלות באילו שיטות ובנאים הן תומכות. הכלים מאפשרים למשתמשים בהם לבנות באופן אינטראקטיבי יישומים אשר ניגשים למחלקות אלו, אבל היישומים שנוצרים ניגשים למחלקות בדרך הרגילה, לא באופן משקף. השתקפות משמשת רק בזמן תכנון. **ככלל, אובייקטים לא צריכים להיות נגישים באופן משקף ביישומים רגילים בזמן ריצה**.

יש מספר יישומים מתוחכמים אשר דורשים את השימוש בהשתקפות (reflection). לדוגמה, דפדפני מחלקה, מפקחי אובייקט, כלי ניתוח קוד ומערכות משובצות. השתקפות מתאימה גם לשימוש במערכות RPC כדי להימנע מהצורך במהדרי stub. אם יש לך ספקות בנוגע ל-האם היישום שלך משתייך גם הוא לאחת מהקטגוריות האלו, אז ככל הנראה שהוא לא.

אתה יכול להשיג הרבה מהיתרונות שבהשתקפות ולסבול רק מעט מהחסרונות שלה על ידי שימוש בהשתקפות רק בצורה מאוד מוגבלת. להרבה תוכניות אשר חייבות להשתמש במחלקה אשר אינה זמינה בזמן הידור, קיים בזמן הידור ממשק או מחלקת-על מתאימים שבאמצעותם ניתן להתייחס למחלקה (נושא 34). אם זהו המקרה, אתה יכול ליצור מופעים באופן משקף ולגשת אליהם בדרך רגילה באמצעות הממשק או מחלקת-העל שלהם. אם לבנאי המתאים אין כל פרמטרים, כמו שקורה בדרך כלל, אז אתה אפילו לא צריך להשתמש בחבילת התוכנה java.lang.reflect. שיטת Class.newInstance מספקת את התפקודיות הדרושה.

לדוגמה, הנה תוכנית אשר יוצרת מופע Set אשר המחלקה שלו מצוינת על ידי ארגומנט שורת הפקודה הראשון. התוכנית מכניסה את ארגומנטי שורת הפקודה הנותרים לתוך הסט ומדפיסה אותו. בלי להתחשב בארגומנט הראשון, התוכנית מדפיסה את הארגומנטים הנותרים תוך סילוק העותקים שלהם. הסדר שבו הארגומנטים האלה מודפסים תלוי במחלקה שמצוינת בארגומנט הראשון. אם תציין "java.util.HashSet", הם יודפסו לכאורה בסדר אקראי. אם תציין "java.util.TreeSet", הם יודפסו בסדר אלפביתי, כפי שהאלמנטים ב-TreeSet ממוינים:

```
// Reflective instantiation with interface access
public class LimitedReflection {
   public static void main(String[] args) {
      // Translate the class name into a class object
      Class cl = null;
      try {
         cl = Class.forName(args[0]);
      } catch(ClassNotFoundException e) {
         System.err.println("Class not found.");
         System.exit(1);
      }

      // Instantiate the class
      Set s = null;
      try {
         s = (Set) cl.newInstance();
      } catch(IllegalAccessException e) {
         System.err.println("Class not accessible.");
         System.exit(1);
      } catch(InstantiationException e) {
         System.err.println("Class not instantiable.");
         System.exit(1);
      }

      // Exercise the set
      s.addAll(Arrays.asList(args).subList(1, args.length-1));
      System.out.println(s);
   }
}
```

למרות שתוכנית זו היא בסך הכל שעשוע, הטכניקה שהיא מדגימה היא מאוד רבת עוצמה. אפשר להפוך בקלות את תוכנית השעשוע לבוחן כללי אשר נותן תוקף למימוש Set שצוין על ידי הפעלה אגרסיבית של מופע אחד או יותר ובדיקה שהם מצייתים להסכם של Set. באופן דומה, אפשר להפוך אותה לכלי ניתוח ביצועים. למעשה, הטכניקה שהיא מדגימה מספיקה כדי לממש סביבת עבודה (פרק 2, נושא 1). ברוב הזמן, אתה רק זקוק לטכניקה זו במובן של השתקפות.

אתה יכול לראות שני חסרונות של השתקפות בדוגמה. הראשון הוא שהתוכנית יכולה לעורר שלוש טעויות זמן ריצה שכולן היו יכולות להיות טעויות זמן הידור אם לא היה נעשה שימוש בהשתקפות. השני הוא שהתוכנית דורשת עשרים שורות של קוד משעמם כדי לייצר מופע של המחלקה מהשם שלו בעוד שקריאה לבנאי תתאים בצורה נקייה לשורה יחידה. בכל אופן, חסרונות אלה מוגבלים לחלק של התוכנית אשר יוצר מופע של האובייקט. לאחר שנוצר מופע שלו, הוא בלתי ניתן להבדלה מכל מופע Set אחר. לכן, בתוכנית אמיתית החלק הארי של הקוד לא מושפע על ידי השימוש המוגבל הזה בהשתקפות.

שימוש לגיטימי, אם כי נדיר, של השתקפות הוא לשבור תלות של מחלקה במחלקות אחרות, שיטות או שדות אשר עשויים להיעדר בזמן ריצה. הדבר יכול להיות שימושי אם אתה כותב חבילת תוכנה אשר חייבת לרוץ כנגד גרסאות מרובות של חבילת תוכנה אחרת. הטכניקה היא להדר את חבילת התוכנה שלך כנגד הסביבה המינימלית הדרושה כדי לתמוך בה, באופן טיפוסי הגירסה הכי ישנה, ולגשת למחלקות או שיטות חדשות יותר באופן משקף. כדי לגרום לזה לעבוד, אתה צריך לנקוט בפעולה מתאימה אם שיטה או מחלקה חדשה יותר אשר אתה מנסה לגשת אליה לא קיימת בזמן ריצה. פעולה מתאימה עשויה להיות מורכבת משימוש באמצעים חילופיים מסוימים כדי להשלים את אותה המטרה, או כדי לפעול עם תפקודיות נמוכה יותר.

בקיצור, השתקפות היא אמצעי רב עוצמה אשר נדרש למשימות מתוחכמות מסוימות של תכנות מערכת, אבל יש לה הרבה חסרונות. אם אתה כותב תוכנית אשר צריכה לעבוד עם מחלקות שאינן ידועות בזמן הידור, אתה צריך, אם הדבר אפשרי בכלל, להשתמש בהשתקפות רק כדי ליצור מופעים של אובייקטים ולגשת לאובייקטים תוך שימוש בממשק או מחלקת-על מסוימים, אשר ידועים בזמן הידור.

נושא 36: השתמש בשיטות native בתבונה

הממשק המקומי של **JNI** (Java Native Interface) מאפשר ליישומי Java לקרוא לשיטות מקומיות (native), שהן שיטות מיוחדות אשר כתובות בשפות תכנות מקומיות כמו C או C++. שיטות מקומיות יכולות לבצע פעולות בשפות מקומיות מחוץ לשפת Java.

מבחינה היסטורית, לשיטות מקומיות היו שלושה שימושים עיקריים. הן סיפקו גישה לאמצעים של פלטפורמה ספציפית כמו לרישום המערכת (registry) ונעילות קובץ. הן סיפקו גישה לספריות של קוד legacy, אשר יכול בתורו לספק גישה לנתוני legacy. לבסוף, שיטות מקומיות שימשו כדי לכתוב חלקים בעלי חשיבות קריטית לביצועים של יישומים בשפות מקומיות כדי לשפר את הביצועים.

לגיטימי להשתמש בשיטות מקומיות כדי לגשת לאמצעים של פלטפורמה ספציפית, אבל ככל ששפת Java מתפתחת, היא מספקת יותר ויותר תכונות שלפני כן נמצאו רק בפלטפורמות מארחות. לדוגמה, חבילת התוכנה java.util.prefs, שהתווספה בגירסה 1.4, מציעה את האפשרות של רישום המערכת. לגיטימי גם להשתמש בשיטות מקומיות כדי לגשת לקוד legacy, אבל יש דרכים טובות יותר כדי לגשת לקוד legacy מסוים. לדוגמה, ה- API JDBC מספק גישה לבסיסי נתונים מסוג legacy.

נכון לגירסה 1.3, לעיתים נדירות מומלץ להשתמש בשיטות מקומיות כדי לשפר ביצועים. בגרסאות מוקדמות יותר, הדבר היה הכרחי לעיתים קרובות, אבל מימושי JVM הפכו להיות מהירים הרבה יותר. עבור מרבית המשימות, אפשרי כעת לשמור על ביצועים דומים ללא שימוש בשיטות מקומיות. לדוגמה, כאשר java.math התווספה לפלטפורמה בגירסה 1.1, המחלקה BigInteger מומשה על גבי ספריה אריתמטית רבת דיוק ומהירה שנכתבה בשפת C. באותו הזמן, הדבר היה הכרחי כדי לספק ביצועים מספקים. בגירסה 1.3, BigInteger נכתבה מחדש לגמרי בשפת Java והותאמה בקפידה. הגירסה החדשה מהירה יותר מהגירסה המקורית בכל מימושי JVM של גירסה 1.3 של Sun עבור מרבית הפעולות והגדלים של האופרנדים.

לשימוש בשיטות מקומיות יש חסרונות רציניים. בגלל ששפות מקומיות אינן **בטוחות** (נושא 24), יישומים אשר משתמשים בשיטות מקומיות אינם עמידים יותר בפני שגיאות קריסת זיכרון. בגלל ששפות מקומיות תלויות בפלטפורמה, יישומים אשר משתמשים בשיטות מקומיות אינם ניידים יותר באופן חופשי. קוד מקומי חייב לעבור הידור מחדש עבור כל פלטפורמת מטרה ועשוי לדרוש גם שינוי. יש עלות קבועה גבוהה בהקשר לכניסה ויציאה מקוד מקומי, כך ששיטות מקומיות יכולות אפילו **להפחית** ביצועים אם הן משמשות רק לביצוע עבודה מצטברת קטנה. לבסוף, מעייף לכתוב שיטות מקומיות וקשה לקרוא אותן.

לסיכום, חשוב פעמיים לפני שימוש בשיטות מקומיות. לעיתים נדירות, אם בכלל, עשה בהן שימוש כדי להשיג ביצועים משופרים. אם אתה חייב להשתמש בשיטות מקומיות כדי לגשת למשאבים מהרמה הנמוכה או לספריות שהורשו, השתמש בקוד מקומי קטן ככל האפשר ובחן אותו ביסודיות. באג בודד בקוד מקומי יכול להביא לקריסת כל היישום שלך.

נושא 37: בצע אופטימיזציה בתבונה

יש שלושה פתגמים בנוגע לאופטימיזציה אשר כל אחד צריך לדעת. אולי הם מתחילים לסבול מחשיפת יתר, אבל במקרה שאינך מכיר אותם, הנה הם:

- יותר שגיאות תכנות מתרחשות בשם היעילות (בלי שהיא בהכרח מושגת) מאשר כל סיבה בודדת אחרת - כולל טיפשות עיוורת [William A.Wulf - Wulf72].
- אנו צריכים לשכוח מהתייעלויות קטנות, בערך ב- 97% מהזמן. אופטימיזציה במקום שאינה דרושה היא השורש של כל הרע [Donald E.Knuth[Knuth74].
- בנוגע לאופטימיזציה אנו נשמעים לשני חוקים:

  חוק מספר 1: אל תבצע אופטימיזציה.

  חוק מספר 2 (למומחים בלבד): עדיין אל תבצע אופטימיזציה, כלומר, כל עוד אין לך פתרון מושלם וברור שעדיין לא עבר אופטימיזציה [M.A. Jackson[Jackson75].

כל הפתגמים האלה קדמו לשפת Java בשני עשורים. הם מוסרים אמת חשובה לגבי אופטימיזציה: קל לעשות נזק יותר מאשר תועלת, במיוחד אם אתה מבצע אופטימיזציה בטרם עת. תהליך האופטימיזציה יכול להביא ליצירת תוכנה שהיא גם לא מהירה וגם לא נכונה ויהיה קשה לתקן אותה.

אל תקריב עקרונות ארכיטקטוניים מבוססים כדי להשיג ביצועים. **שאף לכתוב תוכניות טובות יותר מאשר מהירות**. אם תוכנית טובה אינה מספיק מהירה, הארכיטקטורה שלה תרשה לה לעבור אופטימיזציה. תוכניות טובות ממחישות את העיקרון של **הסתרת מידע**: כאשר הדבר אפשרי, הן ממקמות החלטות תכנון בתוך מודולים אינדיבידואליים ולכן אפשר לשנות החלטות אינדיבידואליות בלי להשפיע על שאר המערכת (נושא 12).

אין הדבר אומר שאתה יכול להתעלם משיקולי ביצועים כל עוד תוכניתך לא הושלמה. בעיות מימוש אפשר לתקן על ידי אופטימיזציה מאוחרת יותר, אבל פגמים ארכיטקטואלים עמוקים אשר מגבילים ביצועים כמעט ואי אפשר לתקן ללא כתיבה מחדש של המערכת. שינוי היבט יסודי של התכנון שלך לאחר מעשה יכול להתבטא במערכת עם מבנה חלש שקשה לתחזק ולפתח. לכן, אתה צריך לחשוב על הביצועים במהלך תהליך התכנון.

שאף להימנע מהחלטות תכנון אשר מגבילות את הביצועים. מרכיבי התכנון אשר הכי קשה לשנות לאחר מעשה הם אלה שמציינים אינטראקציות בין מודולים ועם העולם החיצון.

העיקריים שבין מרכיבים אלה הם APIs, פרוטוקולי רמת כבל ותבניות נתונים קבועות. לא רק שקשה או אפילו בלתי אפשרי לשנות מרכיבים אלה לאחר מעשה, אלא גם שכל אחד מהם יכול להציב הגבלות משמעותיות על הביצועים שמערכת תוכל להשיג אי פעם.

קח בחשבון את נסיבות הביצועים של החלטות תכנון ה-API שלך. עשיית טיפוס ציבורי משתנה יכולה לדרוש המון העתקה הגנתית חסרת צורך (נושא 24). באופן דומה, שימוש בהורשה במחלקה ציבורית במקום שבו היה אולי יותר מתאים להשתמש בהכלה, קושר את המחלקה לעולם למחלקת-העל שלה, דבר אשר יכול לגרום להגבלות מלאכותיות על ביצועי תת-המחלקה (נושא 14). ואחרון חביב, שימוש בטיפוס מימוש על פני ממשק ב-API כובל אותך למימוש ספציפי, למרות שייתכן שמימוש מהיר יותר ייכתב בעתיד (נושא 34).

ההשפעות של תכנון API על הביצועים הן מאוד ממשיות. חשוב על שיטת getSize במחלקת java.awt.Component. ההחלטה ששיטה קריטית לביצועים זו היתה אמורה להחזיר מופע Dimension, יחד עם ההחלטה שמופעי Dimension הם משתנים, כופה על כל מימוש של השיטה הזו להקצות מופע Dimension חדש בכל קריאה. למרות זאת, בגירסה 1.3, הקצאת אובייקטים קטנים היא בלתי יקרה יחסית והקצאת מיליוני אובייקטים ללא צורך ממש יכולה לפגוע בביצועים.

במקרה זה, היו קיימות מספר חלופות. באופן האידיאלי, Dimension היתה צריכה להיות לא ניתנת לשינוי (נושא 13). באופן חילופי, שיטת getSize היתה יכולה להיות מוחלפת בשתי שיטות שמחזירות את המרכיבים הפשוטים האינדיבידואליים של אובייקט Dimension. למעשה, שתי שיטות כאלו הוספו ל-API של Component בגירסה 1.2 בגלל סיבות ביצועים. בכל אופן, קוד לקוח שהיה קיים קודם לכן, עדיין משתמש בשיטת getSize ועדיין סובל מתוצאות הביצועים של החלטות תכנון ה-API המקורי.

למרבה המזל, בדרך כלל תכנון API טוב קשור לביצועים טובים. **עטיפת API כדי להשיג ביצועים טובים היא רעיון רע מאוד**. סוגיית הביצועים אשר גרמה לך לעטוף את ה-API יכולה להיעלם בגירסה עתידית של הפלטפורמה או בתוכנת רקע אחרת, אבל ה-API העטוף וכאבי הראש שתגרום לך התמיכה בו, עשויים ללוות אותך לצמיתות.

לאחר שתכננת בקפדנות את התוכנית שלך ויצרת מימוש ברור, תמציתי ובנוי היטב, **אז** אולי זה הזמן לשקול אופטימיזציה, בהנחה שעדיין אינך מרוצה מביצועיה של התוכנית שלך. זכור ששני החוקים של ג'קסון הם "אל תעשה זאת" ו-"עדיין אל תעשה זאת" (למומחים בלבד). הוא היה יכול להוסיף עוד חוק : **אמוד את הביצועים לפני ואחרי כל ניסיון אופטימיזציה**.

אתה עשוי להיות מופתע ממה שתגלה. לעיתים קרובות, לניסיונות אופטימיזציה אין השפעה משמעותית על הביצועים. לעיתים הם אפילו פוגעים בביצועים. הסיבה העיקרית היא שקשה לנחש איזו נקודה גורמת לתוכנית שלך לבזבז את הזמן שלה. יכול להיות שאותו חלק של התוכנית שאתה חושב שהוא איטי, אינו אשם וסתם תבזבז את זמנך בניסיון לבצע בו אופטימיזציה. רבים הגיעו למסקנה שתוכניות מוציאות 80% מזמנן על 20% מהקוד שלהן.

כלי חתך פרופיל יכולים לעזור לך להחליט היכן למקד את מאמצי האופטימיזציה שלך. כלים כאלה נותנים לך מידע זמן ריצה כמו בערך כמה זמן כל שיטה צורכת וכמה פעמים היא נקראת. בנוסף למיקוד מאמצי התיאום שלך, הם יכולים להתריע על הצורך בשינויים אלגוריתמיים. אם אלגוריתם שזמן ביצועו ריבועי (או יותר מכך) אורב בתוכנית שלך, אף ניסיון להתאמה לא יתקן את הבעיה. אתה חייב להחליף את האלגוריתם באלגוריתם יותר יעיל. ככל שיש יותר קוד במערכת שלך, כך יותר חשוב להשתמש בכלי חתך פרופיל. זה כמו לחפש אחר מחט בערמת שחת: ככל שהערימה גדולה יותר, כך יהיה יותר שימושי להשתמש במגלה מתכות. ה-SDK של Java 2 בא עם profiler פשוט וכמה כלי חתך פרופיל מתוחכמים זמינים באופן מסחרי.

הצורך באומדן של השפעות אופטימיזציה הוא אפילו גדול יותר בפלטפורמת Java מאשר פלטפורמות מסורתיות יותר, מכיון שלשפת Java אין **מודל ביצועים** חזק. העלויות היחסיות של הפעולות הפשוטות אינן מוגדרות היטב. ה"פער הסמנטי" בין מה שהמתכנת כותב לבין מה שהמעבד מבצע הוא גדול הרבה יותר מאשר בשפות מסורתיות העוברות הידור, דבר שמקשה מאוד על חיזוי אמין של תוצאות אופטימיזציה כלשהי על הביצועים. יש הרבה מיתוסים נפוצים בנוגע לביצועים אשר מתגלים כחצאי אמיתות או כשקרים גמורים.

לא רק שמודל הביצועים מוגדר באופן לא מוצלח, אלא הוא ניתן לשינוי ממימוש JVM אחד למשנהו ומגירסה לגירסה. אם תריץ את התוכנית שלך על מימושי JVM מרובים, חשוב שתאמוד את ההשפעות של האופטימיזציה שלך על כל אחד מהם. לעיתים, תיאלץ לעשות החלפות בין ביצוע על מימושי JVM שונים.

לסיכום, אל תשאף לכתוב תוכניות מהירות - שאף לכתוב תוכניות טובות. שיקולי מהירות ניצבים אחר כך. אל תחשוב על סוגיות ביצועים כאשר אתה מתכנן מערכות ובמיוחד כאשר אתה מתכנן APIs, פרוטוקולים ותבניות נתונים קבועות. לאחר שסיימת לבנות את המערכת, אמוד את ביצועיה. אם היא מהירה מספיק, אז סיימת את מלאכתך. אם לא, אתר את מקור הבעיה ובצע אופטימיזציה לחלקים הרלוונטיים של המערכת. הצעד הראשון הוא לבחון את האלגוריתמים שבחרת: שום כמות של אופטימיזציית רמה נמוכה לא יכולה לשפר בחירת אלגוריתמים גרועה. חזור על התהליך כמה שצריך, תוך אומדן של הביצועים לאחר כל שינוי, עד שתהיה מרוצה.

נושא 38: דבוק במוסכמות מתן שמות מקובלות

לפלטפורמת Java יש סט מבוסס היטב של **מוסכמות מתן שמות**, הרבה מהן כלולות במפרט של שפת Java [JLS, 6.8 - Java Language Specification]. בשפה פשוטה, מוסכמות מתן שמות משתייכות לשתי קטגוריות: טיפוגרפיות ודקדוקיות.

יש רק מעט מוסכמות מתן שמות טיפוגרפיות שמכסות חבילות תוכנה, מחלקות, ממשקים, שיטות ושדות. אתה צריך להפר אותן **רק** במקרים נדירים וכאשר יש לך סיבה ממש טובה. אם API מפר מוסכמות אלו, ייתכן שיהיה קשה להשתמש בו. אם מימוש מפר אותן, ייתכן שיהיה קשה לתחזק אותו. בשני המקרים, להפרות יש את הפוטנציאל לבלבל ולהרגיז מתכנתים אחרים אשר עובדים עם הקוד והן גם יכולות לגרום לתפיסות שגויות אשר מובילות לטעויות. המוסכמות מתומצתות בנושא זה.

שמות חבילות צריכים להיות היררכיים כאשר חלקיהם מופרדים על ידי נקודות. החלקים צריכים להיות מורכבים מתווים אלפביתיים של אותיות **קטנות** ולעיתים נדירות - ספרות. השם של כל חבילת תוכנה שתשמש מחוץ לארגון שלך צריך להתחיל עם שם תחום האינטרנט של הארגון שלך, תחילה עם שם התחום הגבוה ביותר. לדוגמה: edu.cmu, com.sun, gov.nsa. חבילות התוכנה של Java, אשר שמותיהן מתחילים ב-java וב-javax, הן חריגות לכלל הזה. למשתמשים אסור ליצור חבילות תוכנה אשר שמותיהן מתחילים ב-java או ב-javax. כללים מפורטים להמרת שמות תחום אינטרנט לתחיליות של שמות חבילות תוכנה אפשר למצוא במפרט של שפת Java [JLS, 7.7].

השארית של שם חבילת תוכנה צריכה להיות מורכבת מחלק אחד או יותר שמתארים את חבילת התוכנה. החלקים צריכים להיות קצרים, בדרך כלל שמונה תווים או פחות. קיצורים משמעותיים הם רצויים, לדוגמה util במקום utilities. ראשי תיבות מותרים, לדוגמה awt. החלקים צריכים להיות מורכבים בדרך כלל ממילה יחידה או מקיצור.

להרבה חבילות תוכנה יש שמות עם בסך הכל חלק אחד בנוסף לשם תחום האינטרנט. חלקים נוספים מתאימים לאמצעים גדולים אשר גודלם דורש שהם יישברו להיררכיה לא פורמלית. לדוגמה, לחבילת התוכנה java.swing יש היררכיה עשירה של חבילות תוכנה עם שמות כמו java.swing.plaf.metal. חבילות תוכנה כאלו נחשבות לעיתים כתת-חבילות תוכנה, למרות שהן תת-חבילות תוכנה על פי מוסכמה בלבד. אין שום תמיכה לשונית להיררכיות חבילת תוכנה.

שמות מחלקה וממשק צריכים להיות מורכבים ממילה אחת או יותר, כאשר האות הראשונה של כל מילה היא אות גדולה, לדוגמה Timer או TimerTask. יש להימנע מקיצורים, פרט לראשי תיבות וקיצורים מקובלים מסוימים כמו max ו-min. אין כמעט הסכמה לגבי האם ראשי תיבות צריכים להיות כתובים באותיות גדולות או שרק האות הראשונה שלהם צריכה להיות אות גדולה. למרות שאותיות גדולות יותר מקובלות, להגדלת האותיות הראשונות בלבד יש טיעון חזק. אפילו אם ראשי תיבות מרובים ממוקמים גב אל גב, אתה עדיין יכול לומר היכן מילה אחת מתחילה והמילה הבאה מסתיימת. איזה שם מחלקה אתה מעדיף לראות, HTTPURL או HttpUrl?

לשמות שדה ושיטה יש אותן המוסכמות הטיפוגרפיות כמו לשמות ממשק ומחלקה, חוץ מהעובדה שהאות הראשונה בשם שיטה או שדה צריכה להיות אות קטנה, לדוגמה remove, ensureCapacity. אם המילה הראשונה של שם שדה או שיטה היא ראשי תיבות, היא צריכה להיות מורכבת מאותיות קטנות.

החריג היחיד לחוק הקודם נוגע ל"שדות קבועים", אשר שמותיהם צריכים להיות מורכבים ממילה אחת או יותר של אותיות גדולות, כשקו תחתון מפריד בין המילים, לדוגמה VALUES או NEGATIVE_INFINITY. שדה קבוע הוא שדה סופי סטטי שערכו לא ניתן לשינוי. אם לשדה סופי סטטי יש טיפוס פשוט או טיפוס ייחוס לא ניתן לשינוי (נושא 13), אז הוא שדה קבוע. אם לטיפוס יש פוטנציאל להשתנות, הוא עדיין יכול להיות שדה קבוע אם האובייקט המיוחס לא ניתן לשינוי. לדוגמה, typesafe enum יכול לייצא את מיגוון הקבועים הנספרים שלו בקבוע List לא ניתן לשינוי (עמוד 107). שים לב ששדות קבועים הם **היחידים** שמומלץ להשתמש בתוכם בקו תחתון.

לשמות משתנים מקומיים יש מוסכמות מתן שמות טיפוגרפיות דומות לאלו של שמות איברים, חוץ מהעובדה שמותרים קיצורים, כמו גם תווים אינדיבידואליים ורצפים קצרים של תווים אשר משמעותם תלויה בהקשר שבו נעשה שימוש במשתנה המקומי, לדוגמה i, xref, houseNumber.

לעיון חטוף, טבלה 7.1 מראה דוגמאות של מוסכמות טיפוגרפיות:

טבלה 7.1: דוגמאות של מוסכמות טיפוגרפיות

טיפוס מזהה	דוגמאות
חבילת תוכנה	com.sun.medialib, com.sun.jdi.event
מחלקה/ממשק	Timer, TimerTaks, KeyFactorySpi, HttpServlet
שיטה או שדה	remove, ensureCapacity, getCrc
שדה קבוע	VALUES, NEGATIVE_INFINITY
משתנה מקומי	i, xref, houseNumber

המוסכמות הדקדוקיות הן יותר גמישות ויותר שנויות במחלקות מאשר המוסכמות הטיפוגרפיות. אין שום מוסכמות מתן שמות דקדוקיות שאפשר לדבר עליהן בנוגע לחבילות תוכנה. מחלקות בדרך כלל מקבלות שם עצם או ביטוי שם עצם, לדוגמה Timer או BufferedWriter.

ממשקים מקבלים שם כמו מחלקות (לדוגמה, Collection או comparator), או שם תואר שמסתיים ב-"able-" או "ible-" (לדוגמה, Runnable או Accessible).

שיטות אשר מבצעות פעולות מסוימות בדרך כלל מקבלות שמות עם פועל או ביטוי פועל, לדוגמה append או drawImage. לשיטות אשר מחזירות ערך בוליאני יש בדרך כלל שמות אשר מתחילים במילה "is", כאשר אחריה מופיע שם עצם, ביטוי שם עצם או כל מילה או ביטוי אשר מתפקדים כתואר השם. לדוגמה: isDigit, isProbablePrime, isEmpty, isEnabled, isRunning.

לשיטות אשר מחזירות ערך לא בוליאני או תכונה של האובייקט שלגביו הן נקראו, בדרך כלל יש שמות עם שם עצם, ביטוי שם עצם או ביטוי פועל שמתחילים בפועל "get". לדוגמה: hashCode, size או getTime. יש הטוענים שרק הצורה השלישית (שפותחת ב-get) היא מקובלת, אבל אין שום בסיס לטענה זו. שתי הצורות הראשונות מובילות בדרך כלל לקוד קריא יותר, לדוגמה:

```
if (car.speed() > 2 * SPEED_LIMIT)
  generateAudibleAlert("Watch out for cops!");
```

הצורה שמתחילה ב-"get" היא הכרחית אם המחלקה שמכילה את השיטה היא Bean [JavaBeans] והיא מומלצת אם אתה שוקל לשנות את המחלקה ל-Bean בזמן מאוחר יותר. כמו כן, אם המחלקה מכילה שיטה שקובעת את ערכי אותה התכונה, שתי השיטות צריכות להיקרא get*Attribute* ו-set*Attribute*.

כמה שמות שיטות ראויים להתייחסות מיוחדת. שיטות אשר ממירות את הטיפוס של אובייקט ומחזירות אובייקט בלתי תלוי מטיפוס אחר, נקראות לעיתים קרובות to*Type*. לדוגמה: toString, toArray. שיטות אשר מחזירות view (נושא 4) אשר הטיפוס שלו שונה מזה של האובייקט המתקבל, נקראות לעיתים קרובות as*Type*. לדוגמה, asList. שיטות אשר מחזירות פשוט עם אותו הערך כמו של האובייקט שלגביו הן נקראו, נקראות לעיתים קרובות *type*Value. לדוגמה: intValue. שמות מקובלים ליצרנים סטטיים הם ValueOf ו-getInstance (נושאים 1, 9).

מוסכמות דקדוקיות לשמות שדות הן פחות מבוססות ופחות חשובות מאלו של מחלקה, ממשק ושמות שיטה, מכיון ש-API מתוכנן היטב מכיל מעט שדות חשופים, אם בכלל. באופן טיפוסי, לשדות מטיפוס boolean שמות כמו שיטות אמצעי גישה מטיפוס boolean כשהתחילית "is" מושמטת. לדוגמה: initialized, composite. שדות או טיפוסים אחרים בדרך כלל מקבלים שמות עם שם עצם או ביטויי שם עצם, כמו height, digits או bodyStyle. מוסכמות דקדוקיות למשתנים מקומיים דומות לאלו של שדות אבל הן אפילו חלשות יותר.

לסיכום, הפנם את המוסכמות הסטנדרטיות למתן שמות ולמד להשתמש בהן כטבע שני. המוסכמות הטיפוגרפיות הן ברורות ובמידה רבה אינן משתמעות לשתי הפנים. המוסכמות הדקדוקיות הן יותר מורכבות ויותר חופשיות. אם לצטט מהמפרט של שפת Java [6.8, JLS], "לא צריך לציית למוסכמות אלו כעבד נרצע אם שימוש מקובל שהתרגלת אליו במשך זמן רב מכתיב אחרת". השתמש בשכל הישר.

פרק 8

חריגים (Exceptions)

כאשר משתמשים בהם באופן נכון, חריגים יכולים לשפר את הקריאות, אמינות ויכולת התחזוקה של תוכנית. כאשר משתמשים בהם באופן לא נכון, יכולה להיות להם השפעה הפוכה. פרק זה מספק קווים מנחים לשימוש יעיל בחריגים.

נושא 39: השתמש בחריגים רק במצבים יוצאי דופן

יום אחד, אם תהיה חסר מזל, אתה עלול למעוד בגלל קטע קוד אשר נראה דומה לקטע הבא:

```
//Horrible abuse of exceptions. Don't ever do this!
try {
    int i =0;
    while(true)
        a [i++].f();
} catch(ArrayIndexOutOfBoundsException e) {
}
```

מה הקוד הזה עושה? לאחר סקירה מהירה לא הכל ברור וסיבה זו מספיקה כדי לא להשתמש בו. מתברר שזהו קטע קוד נורא ואיום למעבר על פני אלמנטים של מערך בלולאה. הלולאה האינסופית מסתיימת על ידי זריקה, תפיסה והתעלמות מ-ArrayIndexOutOfBoundsException כאשר היא מנסה לגשת לאלמנט המערך הראשון מחוץ לתחומי המערך. הדבר אמור להיות מקביל לקטע הקוד הסטנדרטי עבור מעבר על פני מערך בלולאה, קטע קוד שמייד נראה מוכר לכל מתכנת Java:

```
for (int i = 0; i < a.length; i++)
  a[i].f();
```

אם כן, אז מדוע לא מומלץ להשתמש בקוד מבוסס-החריג על פני הקוד הסטנדרטי? זהו ניסיון מוטעה לשפר ביצועים בהתבסס על המסקנה המוטעית לפיה, מאחר שה-VM בודק את תחומי כל הגישות של המערך, מבחן הסיום הרגיל של הלולאה (i < a.length) מיותר ויש להימנע ממנו. ישנן שלוש טעויות בנוגע למסקנה זו:

- בגלל שחריגים מתוכננים לשימוש בנסיבות חריגות, מימושי JVM מעטים, אם בכלל, מנסים לבצע אופטימיזציה לביצועים שלהם. בדרך כלל יצירה, זריקה ותפיסה של חריג הן פעולות יקרות.
- מיקום קוד בתוך בלוק try-catch מונע אופטימיזציות מסוימות אשר מימושי JVM מודרניים עשויים לבצע.
- הקוד הסטנדרטי למעבר בלולאה על פני מערך לא מסתיים בהכרח בבדיקות מיותרות. מימושי JVM מודרניים מסוימים מבצעים להם אופטימיזציה.

למעשה, הקוד מבוסס החריג איטי בהרבה מאשר הקוד הסטנדרטי בכל מימוש JVM נוכחי. במחשב שלי, הקוד מבוסס החריג איטי פי שבעים מאשר הקוד הסטנדרטי כאשר הלולאה רצה מ- 0 עד 99.

לא רק שקוד הלולאה מבוסס החריג מערפל את המטרה של הקוד ומפחית את הביצועים, אלא גם שלא מובטח שהוא אכן יעבוד. כתוצאה מנוכחות של באג שלא קשור, הקוד יכול ליפול באופן חרישי ולהסתיר את הבאג, דבר שמסבך מאוד את תהליך ניפוי השגיאות. נניח שהחישוב בגוף הלולאה מכיל באג אשר מתבטא בגישה מחוץ לתחום לאיזה מערך שאינו קשור. אם היית משתמש בקוד לולאה הגיוני, הבאג היה יוצר חריג שלא נתפס, דבר שמביא לסיום מטלה מיידי עם הודעת שגיאה מתאימה. לעומת זאת, אם היית משתמש בקוד הלולאה הגרוע מבוסס החריג, החריג שקשור לבאג היה מתעורר ומפורש בטעות כסיום לולאה רגיל.

הלקח מתוך כך הוא פשוט: **כפי ששמם מרמז, יש להשתמש בחריגים רק במקרים חריגים. אין להשתמש בהם לזרימת שליטה רגילה**. באופן יותר כללי, אתה צריך להעדיף קודים סטנדרטיים שמזוהים בקלות על פני קודים יותר מדי חכמים אשר מתיימרים להציע ביצועים משופרים. אפילו אם השיפור בביצועים הוא ממשי, הוא לא אמור לשרוד במובן של שיפור מתמיד של מימושי JVM. בכל אופן, הבאגים השנונים וכאבי הראש אשר נובעים מקודים יותר מדי חכמים, בטוח יישארו.

לעיקרון זה יש גם משמעות עבור תכנון API. **אסור ל-API מתוכנן היטב לכפות על הלקוחות שלו להשתמש בחריגים לזרימת שליטה רגילה**. למחלקה עם שיטה "תלוית מצב", אשר יכולה להיקרא רק במצבים בלתי צפויים מסוימים, צריכה בדרך כלל להיות שיטת "בחינת מצב" נפרדת אשר אומרת האם מתאים לקרוא לשיטה הראשונה.

לדוגמה, למחלקת ה-Iterator יש את שיטת next תלוית המצב, אשר מחזירה את האלמנט הבא באיטרציה ואת שיטת בחינת המצב המתאימה, hasNext. הדבר מאפשר לקוד הסטנדרטי לחזור שוב ושוב על פני אוסף:

```
for (Iterator i =collection.iterator(); i.hasNext();  ) {
    Foo foo =(Foo)i.next();
    ...
}
```

אם Iterator היה חסר את שיטת hasNext, הלקוח היה נאלץ לעשות את הדבר הבא במקום:

```
//Do not use this hideous idiom for iteration over a collection!
try {
    Iterator i =collection.iterator();
    while(true){
        Foo foo =(Foo)i.next();
        ...
     }
} catch (NoSuchElementException e) {
}
```

זה צריך להיראות מאוד מוכר לאחר הדוגמה של המעבר על פני המערך אשר פתחה את הנושא הזה. מלבד היותו מרבה במילים ומטעה, סביר שהקוד מבוסס החריג יתבצע באופן גרוע במידה משמעותית מאשר הקוד הסטנדרטי ויכול להסתיר באגים בחלקים לא קשורים של המערכת.

חלופה לאספקת שיטת בחינה מצב נפרדת היא לגרום לשיטה תלוית המצב להחזיר ערך מיוחד, כמו null, אם היא נקראת כשהאובייקט במצב לא מתאים. טכניקה זו לא תהיה מתאימה ל-Iterator, מכיון ש-null היא ערך מוחזר לגיטימי של שיטת next.

הנה כמה קווים מנחים שיעזרו לך לבחור בין שיטת בחינת מצב לבין ערך מוחזר מיוחד. אם אובייקט נגיש בו-זמנית ללא סינכרוניזציה חיצונית או שהוא כפוף לשינויי מצב מבחוץ, ייתכן שיהיה חיוני להשתמש בערך מוחזר מיוחד, מכיון שמצב האובייקט יכול להשתנות במרווח שבין הקריאה לשיטת בחינת המצב והשיטה תלוית המצב המקבילה שלה. עניינים הנוגעים לביצועים עשויים להכתיב שערך מוחזר מיוחד ישמש אם שיטת בחינת מצב נפרדת תשכפל, מתוך צורך, את העבודה של השיטה תלוית המצב. בכל אופן, שיטת בחינת מצב היא מעט עדיפה יותר על פני ערך מוחזר מיוחד, היא מציעה קריאות מעט יותר טובה וסביר ששימוש לא נכון יתגלה ויתוקן יותר בקלות.

נושא 40: השתמש בחריגים מסומנים למצבים בני-התאוששות ובחריגי זמן ריצה לשגיאות תכנות

שפת Java מספקת שלושה סוגי חריגים שניתן לזרוק (throwable):

- **חריגים מסומנים** - Checked Exceptions
- **חריגי זמן ריצה** - Runtime Exceptions
- **שגיאות** - Errors

יש בלבול מסוים אצל מתכנתים בנוגע לנסיבות שבהן מתאים כל סוג חריג. למרות שההחלטה היא לא תמיד מהוקצעת, יש כמה כללים כלליים אשר מקלים מאוד על הבחירה.

החוק הראשי בנוגע להחלטה האם להשתמש בחריג מסומן או בחריג לא מסומן קובע: **השתמש בחריגים מסומנים למצבים שמהם אפשר לצפות באופן הגיוני שמפעיל השיטה יוכל להתאושש**. על ידי זריקה של חריג מסומן, אתה מכריח את מפעיל השיטה לטפל בחריג בפיסקה catch או להכריז עליה ולהעביר אותה החוצה. לכן, כל חריג מסומן ששיטה מוצהרת עשויה לזרוק הוא אינדיקציה חזקה למשתמש ב-API, כאשר המצב המקושר לזריקת החריג הוא תוצאה אפשרית של קריאה לשיטה.

על ידי עימות המשתמש ב-API עם חריג מסומן, מתכנן ה-API מאפשר להתאושש מהמצב. המשתמש יכול להתעלם מאפשרות זו על ידי תפיסת החריג והתעלמות ממנו, אבל זהו בדרך כלל רעיון רע (נושא 47).

יש שני סוגים של חריגים לא מסומנים שאפשר לזרוק: חריגי זמן ריצה ושגיאות. הם זהים בהתנהגות שלהם: שניהם בני-זריקה אשר אין צורך, ובדרך כלל לא צריך, לתפוס. אם תוכנית מעוררת חריג לא מסומן או שגיאה, זהו בדרך כלל המקרה שבו התאוששות היא בלתי אפשרית והמשך ביצוע עלול להזיק יותר מלהועיל. אם תוכנית לא תופסת דבר בר-זריקה שכזה, הדבר יגרום למטלה הנוכחית להפסיק עם הודעת שגיאה מתאימה.

השתמש בחריגי זמן ריצה כדי להצביע על טעויות תכנות. רוב חריגי זמן הריצה מצביעים על **הפרות תנאי מוקדם**. הפרת תנאי מוקדם היא פשוט כישלון של לקוח API לדבוק בהסכם שבמפרט ה-API. לדוגמה, ההסכם לגישה למערך מציין שאינדקס המערך חייב להיות בין אפס לבין אורך המערך פחות אחד.
ArrayIndexOutOfBoundsException מצביע על כך שתנאי מוקדם זה הופר.

למרות שה-JLS לא דורש זאת, ישנה מוסכמה חזקה ששגיאות שמורות לשימוש על ידי ה-JVM כדי להצביע על ליקויי משאב, כשלי קבועים או תנאים אחרים אשר לחלוטין לא מאפשרים המשך ביצוע [Chan98, Horstman00]. בהתחשב בתפוצה הכמעט אוניברסלית של מוסכמה זו, הכי טוב לא לממש מחלקות Error חדשות כלשהן.
כל החריגים הלא מסומנים בני-הזריקה שאתה מממש צריכים להיות תתי-מחלקות של-RuntimeException (באופן ישיר או בלתי ישיר).

אפשר להגדיר חריג בר-זריקה שאינו תת-מחלקה של Exception, RuntimeException או Error. ה-JLS לא מגדיר חריגים בני-זריקה כאלה באופן ישיר, אבל מציין במרומז שהם זהים בהתנהגותם לחריגים מסומנים רגילים (שהם תת-מחלקות של Exception אבל לא RuntimeException). אם כן, מתי עלינו להשתמש בחיה שכזאת? בשתי מילים: לעולם לא. אין לזה שום יתרונות על פני מסומנים רגילים חוץ מכך שזה פשוט יבלבל את משתמש ה-API שלך.

לסיכום, השתמש בחריגים מסומנים למצבים בני-התאוששות ובחריגי זמן ריצה לטעויות תכנות. כמובן שהמצב אינו תמיד שחור ולבן. לדוגמה, חשוב על המקרה של קריסת משאבים, אשר יכולה להתרחש עקב טעות תכנות כמו הקצאת מערך גדול באופן בלתי הגיוני או בגלל מחסור אמיתי במשאבים. אם קריסת משאבים מתרחשת עקב מחסור זמני או בגלל דרישה מוגברת זמנית, ייתכן שאפשר יהיה להתאושש ממצב זה. זהו עניין שנתון לשיקול דעתו של מתכנן ה-API - האם סביר שמופע נתון של קריסת משאבים יאפשר התאוששות. אם אתה מאמין שסביר שמצב יאפשר התאוששות, השתמש בחריג מסומן. אם לא, השתמש בחריג זמן ריצה. אם לא ברור האם התאוששות היא אפשרית, כנראה עדיף שתשתמש בחריג לא מסומן, מהסיבות שנדונות בנושא 41.

מתכנני API שוכחים לעיתים קרובות שחריגים הם אובייקטים בשלים לגמרי שלגביהם אפשר להגדיר שיטות שרירותיות. השימוש הראשוני בשיטות כאלו הוא לספק את הקוד אשר תופס את החריג עם מידע נוסף שנוגע למצב אשר גרם לחריג להתעורר. בהיעדר שיטות כאלו, מתכנתים ידועים בכך שהם מפצלים את מחרוזת ייצוג החריג כדי לחשוף מידע נוסף. זהו נוהג רע באופן קיצוני. לעיתים רחוקות, מחלקות מציינות את הפרטים של ייצוגי המחרוזת שלהם. לכן, ייצוגי מחרוזת עשויים להשתנות ממימוש למימוש ומגירסה לגירסה. לכן, סביר שקוד אשר מפצל את הייצוג המחרוזתי של חריג יהיה שביר ולא נייד.

בגלל שחריגים מסומנים בדרך כלל מצביעים על מצבים בני-התאוששות, חשוב במיוחד לחריגים כאלה לספק שיטות אשר מספקות מידע שיכול לסייע למתקשר להתאושש. לדוגמה, נניח שחריג מסומן מתעורר כאשר ניסיון לבצע שיחה בטלפון ציבורי נכשל בגלל שהמתקשר לא שם סכום מספיק של כסף. החריג צריך לספק שיטת עזר כדי לבדוק את הסכום החסר כך שיהיה אפשר לשדר את הסכום למשתמש בטלפון.

נושא 41: הימנע משימוש מיותר בחריגים מסומנים

חריגים מסומנים הם תכונה נהדרת של שפת Java. שלא כמו קודי שגיאה המוחזרים משיטה, הם **מכריחים** את המתכנת לטפל במצבים חריגים, מה שמגדיל מאוד את האמינות. אבל, שימוש יתר בחריגים מסומנים יכול לגרום ל-API להיות הרבה פחות נוח לשימוש. אם שיטה מעוררת חריג מסומן אחד או יותר, הקוד שקורא לשיטה חייב לטפל בחריגים בבלוק catch אחד או יותר, או שהוא חייב להצהיר שהוא מעורר את החריגים ונותן להם להתפשט החוצה. שתי הדרכים מציבות נטל לא פשוט על המתכנת.

הנטל מוצדק אם המצב החריג לא יכול להימנע על ידי שימוש נכון ב-API, והמתכנת שמשתמש ב-API יכול לנקוט בפעולה שימושית כלשהי ברגע שהוא מתעמת עם החריג. חריג לא מסומן הוא יותר מתאים אלא אם כן שני התנאים האלה מתקיימים. כמבחן אישי, שאל את עצמך: כיצד המתכנת יטפל בחריג? האם זה הטוב ביותר שניתן לעשות?

```
}catch(TheCheckedException e){
    throw new Error("Assertion error");  //Should never happen!
}
```

מה בנוגע לזה?

```
} catch(TheCheckedException e) {
    e.printStackTrace(); //Oh well, we lose.
    System.exit(1);
}
```

אם המתכנת שמשתמש ב-API לא יכול לעשות דבר טוב יותר, חריג לא מסומן יהיה יותר מתאים.
דוגמה אחת לחריג אשר נכשל במבחן זה היא CloneNotSupportedException. מעוררת אותו Object.clone, אשר צריכה להיקרא רק על אובייקטים אשר מממשים Cloneable (נושא 10). למעשה, לבלוק catch כמעט תמיד יש את האופי של כישלון עמידה בתוקף (assertion failure). החריג לא מספק כל רווח למתכנת, אבל הוא דורש מאמץ ומסבך תוכניות.

הנטל הנוסף על המתכנת שנגרם בגלל חריג מסומן גבוה יותר באופן ממשי אם זה החריג המסומן היחיד ששיטה עוררה. אם ישנם חריגים אחרים, השיטה כבר חייבת להופיע בבלוק try וחריג זה פשוט דורש בלוק catch אחר. אם שיטה מעוררת חריג מסומן יחיד, חריג זה לבדו אחראי לעובדה שהשיטה חייבת להופיע בבלוק try.
בנסיבות אלו, משתלם לשאול את עצמך האם אין איזו שהיא דרך להימנע מהחריג המסומן.

טכניקה אחת להפיכת חריג מסומן לחריג לא מסומן היא לשבור את השיטה אשר מעוררת את החריג לשתי שיטות, כאשר הראשונה שבהן מחזירה boolean שמציין האם החריג יתעורר. API זה ששינה צורתו משנה את זה:

```
//Invocation with checked exception
try {
    obj.action(args);
} catch(TheCheckedException e) {
    //Handle exceptional condition
    ...
}
```

לזה:

```
//Invocation with state-testing method and unchecked exception
if (obj.actionPermitted(args)) {
    obj.action(args);
} else {
    //Handle exceptional condition
    ...
}
```

שינוי צורה זה אינו מתאים תמיד, אבל במקומות שבהם הוא מתאים הוא יכול ליצור API יותר נוח לשימוש. למרות שהרצף הקורא המאוחר יותר אינו נחמד יותר מהקודם, ה-API המתקבל הוא גמיש יותר. במקרים שבהם המתכנת יודע שהקריאה תצליח או שהוא מסתפק בלתת למטלה להסתיים אם הקריאה נכשלת, שינוי הצורה מאפשר גם רצף קורא פשוט זה:

```
obj.action(args);
```

אם אתה חושד שהרצף הקורא הפשוט יהיה הנורמה, אז טרנספורמציית API זו עשויה להתאים. ה-API שמתקבל משינוי צורה זו זהה מיסודו ל-API של "שיטת בחינת מצב" בנושא 39 ואותן האזהרות חלות: אם אובייקט יהיה נגיש בו-זמנית ללא סינכרוניזציה חיצונית או שהוא כפוף למעברי מצב הנכפם עליו באופן חיצוני, שינוי צורה זה אינו מתאים, מכיון שמצב האובייקט עשוי להשתנות בין הקריאות של actionPermitted ו-action. אם שיטת actionPermitted נפרדת תשכפל, מתוך צורך, את העבודה של שיטת action, שינוי הצורה עשוי להיות משובץ על ידי שיקולי ביצועים.

נושא 42: העדף שימוש בחריגים סטנדרטיים

אחת התכונות שמבדילה הכי הרבה ממתכנתים מנוסים ממתכנתים פחות מנוסים היא שמתכנתים מנוסים שואפים ובדרך כלל מצליחים להשיג דרגה גבוהה של שימוש חוזר בקוד. חריגים אינם חריגים למוסכמה הכללית ששימוש חוזר בקוד הוא טוב. ספריות הפלטפורמה של Java מספקות סט בסיסי של חריגים לא מסומנים אשר מכסים חלק גדול מצרכי זריקת חריגים של מרבית ה-APIs. בנושא זה, נדון בחריגי שימוש חוזר מקובלים אלה.

לשימוש חוזר בחריגים שקיימים לפני כן יש מספר יתרונות. העיקרי שבהם הוא שה-API שלך יהיה קל יותר ללימוד ולשימוש, מכיון שהוא מתאים למוסכמות מבוססות שמתכנתים הספיקו כבר להכיר. יתרון קרוב שני הוא שתוכניות שמשתמשות ב-API שלך הן יותר קריאות מכיון שהן אינן מעורבבות בחריגים לא מוכרים. לבסוף, פחות מחלקות חריגים משמעותן פחות צריכת זיכרון ופחות זמן שמבוזבז על טעינת מחלקות.

החריג הכי מקובל שנעשה בו שימוש חוזר הוא IllegelArgumentException. זהו בדרך כלל החריג שצריך לעורר כאשר המתקשר מעביר ארגומנט שהערך שלו אינו מתאים. לדוגמה, זה עשוי להיות החריג שיש לזרוק אם מפעיל השיטה העביר מספר שלילי בפרמטר אשר מייצג את מספר הפעמים שפעולה מסוימת צריכה לחזור על עצמה.

חריג מקובל אחר שנעשה בו שימוש חוזר הוא IllegalStateException. זהו בדרך כלל החריג שיש לעורר אם הקריאה היא לא חוקית, בהתחשב במצב של האובייקט שהתקבל. לדוגמה, זה עשוי להיות החריג שיש לעורר אם מפעיל השיטה ניסה להשתמש באובייקט מסוים לפני שהוא אותחל כראוי.

באופן שניתן להתווכח לגביו, כל הקריאות המוטעות לשיטות מצטמצמות לארגומנט לא חוקי או מצב לא חוקי, אבל חריגים אחרים משמשים באופן סטנדרטי לסוגים מסוימים של ארגומנטים או מצבים לא חוקיים.
אם מתקשר מעביר null כפרמטר במקרה שבו ערכי null הם אסורים, המוסכמה מכתיבה זריקה של NullPointerException על פני IllegalArgumentException. באופן דומה, אם מתקשר מעביר ערך מחוץ לטווח בפרמטר שמייצג אינדקס לתוך רצף, עדיף לעורר IndexOutOfBoundsException על פני IllegalArgumentExcption.

עוד חריג כלל שימושי שכדאי לדעת הוא ConcurrentModificationException. יש לעורר חריג זה אם אובייקט שתוכנן לשימוש על ידי מטלה בודדת או עם סינכרוניזציה חיצונית מגלה שהוא עובר (או עבר) שינוי בו-זמנית.

חריג רב-שימושי אחרון שכדאי לדעת הוא UnsupportedOperationException. זה החריג שיש לעורר אם אובייקט לא תומך בניסיון לבצע פעולה מסוימת. השימוש בו נדיר יחסית לחריגים האחרים שנדונו בנושא זה, מכיון שרוב האובייקטים תומכים בכל השיטות שהם מממשים.

חריג זה משמש מימושים של ממשקים אשר נכשלים לממש פעולה אפשרית אחת או יותר, אשר מוגדרת על ידי הממשק. לדוגמה, מימוש List לצירוף בלבד עשוי לעורר את החריג הזה אם מישהו ניסה למחוק אלמנט.

טבלה 8.1: מסכמת את החריגים הכי נפוצים שנעשה בהם שימוש חוזר

חריג	נסיבות השימוש
IllegalArgumentException	ערך הפרמטר אינו מתאים
IllegalStateException	מצב האובייקט אינו מתאים לקריאה לשיטה
NullPointerException	ערך הפרמטר הוא null במקום שערך זה אסור
IndexOutOfBoundsException	אינדקס הפרמטר נמצא מחוץ לטווח
ConcurrentModificationException	אובחן שינוי בו-זמנית של אובייקט כאשר הדבר אסור
UnsupportedOperationException	האובייקט אינו תומך בשיטה

למרות שבמידה רבה חריגים אלה בולטים בכך שהם משמשים באופן שכיח בספריות הפלטפורמה של Java, חריגים אחרים יכולים גם הם לשמש כאשר הנסיבות מצדיקות זאת. לדוגמה, אם אתה מממש אובייקטים אריתמטיים כמו מספרים מורכבים או מטריצות, שימוש חוזר ב-ArithmeticException וב-NumberFormatException עשוי להיות מתאים. אם חריג מתאים לצרכיך, אז לך קדימה והשתמש בו אבל רק אם הנסיבות שבהן אתה מעורר אותו מתאימות לתיעוד של החריג. שימוש חוזר חייב להתבסס על סמנטיקה, לא רק על השם. כמו כן, הרגש חופשי לבצע תת-מחלקתיות לחריג קיים אם אתה רוצה להוסיף עוד קצת מידע על לכידת כישלון (נושא 45).

לבסוף, היה מודע לכך שבחירת חריג לשימוש חוזר אינה מדע מדויק, מכיון ש"נסיבות השימוש" שבטבלה 8.1 אינן בלעדיות באופן הדדי. לדוגמה, שקול את המקרה של אובייקט אשר מייצג חפיסת קלפים. נניח שהיתה שיטה לחילוק של יד מחפיסת הקלפים אשר מקבלת כארגומנט את גודל החפיסה. נניח שהמתקשר העביר לפרמטר הזה ערך שגדול ממספר הקלפים אשר נשארו בחפיסה. הדבר היה עשוי להיות מובן כ-IllegalArgumentException (ערך הפרמטר handSize גבוה מדי) או כ-IllegalStateException (אובייקט חפיסת הקלפים מכיל פחות מדי קלפים בשביל הבקשה). במקרה זה, IllegalArgumentException נראה הנכון, אבל אין פה חוקים נוקשים ומהירים.

נושא 43: עורר חריגים שמתאימים להפשטה

מביך להיתקל במצב שבו שיטה מעוררת חריג אשר אין לו שום קשר גלוי למשימה שהוא מבצע. דבר זה קורה לעיתים קרובות כאשר שיטה מפיצה חריג שנזרק על ידי הפשטה מרמה נמוכה. מלבד כך שהדבר מביך, זה גם מזהם את ה-API של השכבה הגבוהה יותר בפרטי מימוש. אם המימוש של השכבה הגבוהה יותר ניתן לשינוי בגירסה עוקבת, החריגים שהיא מעוררת עשויים להשתנות גם הם, דבר שיכול לשבור תוכניות קיימות של לקוחות.

כדי להימנע מבעיה זו, **שכבות גבוהות יותר צריכות לתפוס חריגי רמה נמוכה יותר, ואז, במקומן, לזרוק חריגים שניתנים להסבר במונחים של ההפשטה של הרמה הגבוהה יותר**. קוד זה, אשר אנו קוראים לו **תרגום חריג**, נראה כמו זה:

```
//Exception Translation
try {
    //Use lower-level abstraction to do our bidding
    ...
} catch(LowerLevelException e) {
    throw new HigherLevelException(...);
}
```

הנה דוגמה של דוח חריג שנלקחה ממחלקת AbstractSequentialList, שהיא **מימוש דמוי שלד** (נושא 16) של הממשק List. בדוגמה זו, תרגום חריג מתאפשר על ידי המפרט של שיטת get בממשק List:

```
/ **
  * Returns the element at the specified position in this list.
  * @throws IndexOutOfBoundsException if the index is out of range
  *(index <0 ||index >=size()).
  * /
public Object get(int index){
    ListIterator i =listIterator(index);
    try {
        return i.next();
    } catch(NoSuchElementException e) {
        throw new IndexOutOfBoundsException("Index:"+index);
    }
}
```

צורה מיוחדת של תרגום חריג אשר קרויה **שרשור חריג** מתאימה במקרים שבהם חריג הרמה הנמוכה עשוי להיות לעזר למי שמנפה שגיאות מהמצב שגרם לחריג.

בגישה זו, חריג הרמה הנמוכה נשמר על ידי חריג הרמה הגבוהה, אשר מספק שיטת אמצעי גישה כדי לאחזר את חריג הרמה הנמוכה:

```
//Exception Chaining
try {
    //Use lower-level abstraction to do our bidding
    ...
} catch (LowerLevelException e) {
    throw new HigherLevelException(e);
}
```

נכון לגירסה 1.4, שרשור חריג נתמך על ידי Throwable. אם אתה מתמקד בגירסה 1.4 (או גירסה מאוחרת יותר), אתה יכול לנצל תמיכה זו על ידי שרשור הבנאי של חריג הרמה הגבוהה יותר ל- Throwable(Throwable):

```
//Exception chaining in release 1.4
HigherLevelException(Throwable t){
     super(t);
}
```

אם אתה מתמקד בגירסה מוקדמת יותר, החריג שלך חייב לאחסן את חריג הרמה הנמוכה יותר ולספק אמצעי גישה:

```
//Exception chaining prior to release 1.4
private Throwable cause;

HigherLevelException(Throwable t) {
    cause = t;
}

public Throwable getCause(){
    return cause;
}
```

על ידי קריאה לאמצעי הגישה בשם getCause ושימוש בהצהרה המוצגת, אתה מבטיח שהחריג שלך יפעל באופן אינטראקטיבי עם אמצעי השרשור של הפלטפורמה כאשר תשתמש בחריג כמו של גירסה 1.4. יש לכך את היתרון של צירוף מעקב המחסנית של חריג הרמה הנמוכה יותר עם זו של חריג הרמה הגבוהה יותר באופן סטנדרטי. כמו כן, הדבר מאפשר לכלי ניפוי שגיאות סטנדרטיים לגשת אל חריג הרמה הנמוכה יותר.

למרות שתרגום חריג נעלה על פני הפצה לא זהירה של חריגים משכבות נמוכות יותר, אין להשתמש בזה יתר על המידה. היכן שהדבר אפשרי, הדרך הטובה ביותר לטפל בחריגים משכבות נמוכות יותר היא להימנע מהם לחלוטין על ידי הבטחה ששיטות מרמה נמוכה יותר תצלחנה לפני קריאה אליהן. לפעמים אתה יכול לעשות זאת על ידי בדיקה במפורש של תוקף הארגומנטים של השיטה מהרמה הגבוהה יותר לפני העברתם הלאה לשכבות נמוכות יותר.

בלתי אפשרי למנוע חריגים משכבות נמוכות יותר, הדבר הכי טוב הבא שאפשר לעשות הוא לתת לשכבה הגבוהה יותר לעבוד בשקט על חריגים אלה, תוך בידוד המתקשר של השיטה מהרמה הגבוהה יותר מהבעיה של הרמה הנמוכה יותר.
בנסיבות אלו, עשוי להיות מתאים לתעד את החריג תוך שימוש באמצעי תיעוד מתאימים, כמו java.util.logging, אשר הוצג לראשונה בגירסה 1.4. הדבר מאפשר ל-administrator לחקור את הבעיה תוך בידוד קוד הלקוח ואת משתמש הקצה ממנה.

במצבים בהם לא סביר למנוע חריגים משכבות נמוכות יותר או לבודד שכבות גבוהות יותר מהם, צריך בדרך כלל להשתמש בתרגום חריג. רק אם מיפרט השיטה מהרמה הנמוכה יותר מבטיח במקרה שכל החריגים שהוא מעורר מתאימים לרמה הגבוהה יותר, רק אז יהיה מותר לחריגים להתפשט מהשכבה הנמוכה יותר לשכבה הגבוהה יותר.

נושא 44: תעד את כל החריגים שכל שיטה מעוררת

תיאור של החריגים שמעוררת שיטה כולל חלק חשוב של התיעוד שנדרש כדי להשתמש בשיטה באופן נכון. לכן, חשוב ביותר שתקדיש את הזמן הדרוש לתיעוד זהיר של כל החריגים שמעוררת כל שיטה.

תמיד הצהר על חריגים מסומנים באופן אינדיבידואלי ותעד במדויק את התנאים שבהם כל אחד מהם נזרק על ידי שימוש בתגית התיעוד @throws. אל תנסה לקצר על ידי הצהרה ששיטה מעוררת מחלקת-על של מחלקות חריג מרובות שהיא עשויה לזרוק. כדוגמה קיצונית, לעולם אל תצהיר ששיטה "throws Exception" או אפילו גרוע מכך, "throws Throwable". בנוסף להתנערות מכל הדרכה כלשהי למתכנת שמתעניין בחריגים שהשיטה מסוגלת לעורר, הצהרה כזאת מחביאה במידה רבה מאוד את השימוש של השיטה, מכיון שהיא מטשטשת ביעילות כל חריג אחר אשר עשוי להתעורר באותו הקשר.

למרות שהשפה לא דורשת ממתכנתים להצהיר על החריגים הלא מסומנים ששיטה מסוגלת לעורר, יהיה זה נבון לתעד אותם בזהירות כמו החריגים המסומנים. חריגים לא מסומנים בדרך כלל מייצגים שגיאות תכנות (נושא 40), והעלאת כל השגיאות שמתכנתים עלולים לעשות, עוזרת להם להימנע מהן.
רשימה מתועדת היטב של החריגים הלא מסומנים ששיטה יכולה לעורר, מתארת ביעילות את ה**תנאים המוקדמים** לביצועה המוצלח. חיוני שכל תיעוד של שיטה יתאר את תנאיה המוקדמים ותיעוד חריגיה הלא מסומנים היא הדרך הטובה ביותר לספק דרישה זו.

חשוב במיוחד ששיטות בממשקים תתעדנה את החריגים הלא מסומנים שהן עשויות לעורר. תיעוד זה יוצר חלק של **ההסכם הכללי** של הממשק ומאפשר התנהגות מקובלת בין מימושים מרובים של הממשק.

השתמש בתגית @throws כדי לתעד כל חריג לא מסומן אשר שיטה יכולה לעורר, אבל אל תשתמש במילת המפתח throws כדי לכלול חריגים לא מסומנים בהצהרת השיטה. חשוב שהמתכנת שמשתמש ב-API שלך ידע אילו חריגים הם מסומנים ואילו חריגים אינם מסומנים, מכיון שהאחריות שלו ניתנת לשינוי בהתאם לכל אחד מהמקרים. התיעוד שנוצר על ידי התגית @throws, בהיעדר כותרת השיטה שנוצרת על ידי ההצהרה של throws, מספק דוגמה ויזואלית חזקה כדי לעזור למתכנת להבדיל חריגים מסומנים מחריגים לא מסומנים.

יש לציין שתיעוד כל החריגים הלא מסומנים אשר כל שיטה יכולה לעורר היא שאיפה, לא תמיד מושגת, בעולם האמיתי. כאשר מחלקה עוברת שיפור, אין זו הפרה של מקור או תאימות בינארית אם שיטה מיוצאת מוגדרת לעורר חריגים לא מסומנים נוספים. נניח שמחלקה קוראת לשיטה ממחלקה אחרת שכתובה באופן עצמאי. המחבר של המחלקה הקודמת יכול לתעד בקפדנות את כל החריגים הלא מסומנים אשר כל שיטה זורקת, אבל אם המחלקה השנייה עברה שיפור כך שהיא זורקת חריגים לא מסומנים נוספים, די ייתכן שהמחלקה הקודמת (אשר לא עברה שיפור) תפיץ את החריגים הלא מסומנים החדשים למרות שהיא לא מצהירה עליהם.

אם חריג נזרק על ידי הרבה שיטות במחלקה מאותה הסיבה, יהיה זה מתקבל על הדעת לתעד את החריג בהערת התיעוד של המחלקה, יותר מאשר תיעוד אינדיבידואלי שלה עבור כל שיטה. דוגמה מקובלת היא NullPointerException. יהיה זה תקין אם הערת תיעוד של מחלקה תאמר "כל השיטות במחלקה זו מעוררות NullPointerException אם ייחוס אובייקט null מועבר בכל פרמטר שהוא", או ניסוח אחר בעל אותו משמעות.

נושא 45: כלול מידע לכידת כישלון בהודעות המייצגות את החריג

כאשר תוכנית נכשלת בגלל חריג שלא נתפס, המערכת מדפיסה אוטומטית את מעקב המחסנית של החריג. מעקב המחסנית מכיל את ה**ייצוג המחרוזתי** של החריג, שהוא תוצאה של השיטה toString שלו. הייצוג המחרוזתי מורכב באופן טיפוסי משם המחלקה של החריג שאחריו הודעת הפירוט שלו. לעיתים קרובות, זהו כל המידע שיקבל מתכנת שחוקר כישלון בתוכנה כדי להמשיך הלאה. אם קשה ליצור שנית את הכישלון, ייתכן שיהיה קשה או אף בלתי אפשרי להשיג מידע נוסף. לכן, חשוב ביותר ששיטת toString של החריג תחזיר כמה שאפשר יותר מידע בנוגע לגורם הכישלון. במילים אחרות, הייצוג המחרוזתי של חריג צריך לכלול את המידע על הכישלון בשביל אנליזה מאוחרת.

כדי לתפוס את הכישלון, הייצוג המחרוזתי של חריג צריך להכיל את הערכים של כל הפרמטרים והשדות אשר "גרמו לחריג". לדוגמה, הודעת הפרטים של חריג IndexOutOfBounds צריכה להכיל את התחום התחתון, התחום העליון ואת האינדקס האקטואלי שחרג מתחומים אלה. מידע זה אומר הרבה על הכישלון. כל אחד מהערכים או כולם יכולים להיות מוטעים. האינדקס האקטואלי יכול להיות נמוך באחד מהתחום התחתון או שווה לתחום העליון ("שגיאת חוץ לתחום") או שזה יכול להיות ערך פרוע, נמוך או גבוה בהרבה מהמותר. התחום התחתון יכול להיות גדול מהתחום העליון (כישלון קבוע פנימי רציני). כל אחד ממצבים אלה מצביע על בעיה שונה ומאוד עוזר למתכנת שמאבחן את התקלה לדעת איזה סוג בעיה עליו לחפש.

למרות שקריטי לכלול את כל "הנתונים הקשיחים" הרלוונטיים בייצוג המחרוזתי של חריג, בדרך כלל אין צורך לכלול סיפורים ארוכים. מעקב המחסנית מיועד לעבור ניתוח בצירוף עם קבצי המקור ובדרך כלל כולל את הקובץ המדויק ואת מספר השורה שבהם התעורר החריג, כמו גם את הקבצים ומספרי השורה של כל קריאות שיטה אחרות על המחסנית. סיפורים ארוכים אשר מתארים את הכישלון הם בדרך כלל מיותרים. אפשר לאסוף את המידע על ידי קריאת קוד המקור.

אין לבלבל בין הייצוג המחרוזתי של חריג לבין הודעת שגיאה של רמת המשתמש, אשר חייבת להיות ניתנת להבנה אצל משתמשי הקצה. שלא כמו הודעת שגיאה ברמת המשתמש, הייצוג המחרוזתי נועד בעיקר להביא תועלת למתכנתים כאשר הם חוקרים כישלון. לכן, תוכן המידע הרבה יותר חשוב מיכולתו להיות מובן.

דרך אחת להבטיח שחריגים יכילו מספיק מידע לכידת כישלון בייצוג המחרוזתי שלהם היא לדרוש מידע זה בבנאים שלהם במקום הודעת פירוט מחרוזתית. אז, יהיה אפשר לייצר את הודעת הפירוט באופן אוטומטי כדי לכלול את המידע.

לדוגמה, **במקום בנאי** של String, ל-IndexOutOfBoundsException יכול להיות בנאי אשר נראה כמו הדבר הבא:

```
/ **
  * Construct an IndexOutOfBoundsException.
  *
  * @param lowerBound          the lowest legal index value.
  * @param upperBound     the highest legal index value plus one.
  * @param index               the actual index value.
  * /
public IndexOutOfBoundsException(int lowerBound, int upperBound,
                                 int index) {
  //Generate a detail message that captures the failure
  super("Lower bound: " + lowerBound +
      ", Upper bound: "  + upperBound +
      ", Index: "        + index);
}
```

לרוע המזל, ספריות הפלטפורמה של Java לא עושות שימוש נרחב בקוד זה, אבל הוא מאוד מומלץ. הדבר מקל על המתכנת אשר זורק חריג ללכוד את הכישלון. למעשה, זה מקשה על המתכנת לא ללכוד את המידע הקשור לכישלון בחריג! התוצאה היא שהייצוג המחרוזתי הינו איכותי במובן שלא כל מי שעושה שימוש במחלקה צריך לבנות את הייצוג המחרוזתי לכישלון, פעולה מיותרת לכל הדעות.

כפי שמוצע בנושא 40, עשוי להיות מתאים לחריג לספק שיטות אמצעי גישה עבור מידע לכידת הכישלון שלו (lowerBound, upperBound ו-index בדוגמה שלעיל). חשוב יותר לספק שיטות אמצעי גישה שכאלו לגבי חריגים מסומנים מאשר לגבי חריגים לא מסומנים, מכיון שמידע לכידת הכישלון יכול להיות שימושי כדי להתאושש מהכישלון. נדיר (אם כי לא בלתי סביר) שמתכנת ירצה גישה תכנותית לפרטים של חריג לא מסומן. בכל אופן, אפילו עבור חריגים לא מסומנים, נראה שמומלץ לספק אמצעי גישה אלה כעיקרון כללי (נושא 9, עמוד 44).

נושא 46: שאף ל- failure atomicity

לאחר שאובייקט מעורר חריג, בדרך כלל רצוי שהאובייקט עדיין יהיה במצב שימושי, אפילו אם הכישלון התרחש באמצע ביצוע פעולה. הדבר נכון במיוחד עבור חריגים מסומנים, שמהם צפוי שמזמן השיטה יתאושש. **באופן כללי, קריאה שנכשלה לשיטה צריכה להשאיר את האובייקט במצב שבו הוא היה לפני הקריאה**. אפשר לומר על שיטה שיש לה תכונה זו שהיא **failure atomic**.

יש כמה דרכים להשיג זאת. הדרך הפשוטה ביותר היא לתכנן אובייקטים בלתי משתנים (נושא 13). אם אובייקט לא ניתן לשינוי, failure atomicity מושגת באופן אוטומטי. אם פעולה נכשלת, הדבר יכול למנוע יצירת אובייקט חדש אבל הוא לעולם לא יותיר אובייקט קיים במצב לא עקבי, מכיון שמצבו של כל אובייקט הוא עקבי כאשר הוא נוצר ולעולם אינו יכול להשתנות לאחר מכן.

הדרך הכי מקובלת להשיג failure atomicity עבור שיטות אשר פועלות על אובייקטים משתנים, היא לבדוק את תוקפם של פרמטרים לפני ביצוע הפעולה (נושא 23). הדבר גורם לכל חריג להתעורר לפני שינוי מצב האובייקט. לדוגמה, בחן את השיטה Stack.pop שבנושא 5:

```
public Object pop() {
    if (size ==0)
        throw new EmptyStackException();
    Object result =elements [--size ];
    elements [size ] =null;//Eliminate obsolete reference
    return result;
}
```

אם בדיקת הגודל ההתחלתי היתה מוסרת, השיטה עדיין תעורר חריג כאשר היא תנסה לשלוף אלמנט ממחסנית ריקה. בכל אופן זה ישאיר את שדה הגודל במצב לא עקבי (שלילי), דבר שיגרום לכישלון כל קריאת שיטה עתידית על האובייקט. בנוסף, החריג שמעוררת שיטת pop עשוי לא להיות מתאים להפשטה (נושא 43).

גישה כמעט קרובה להשגת failure atomicity היא לסדר את הקוד, כך שכל חלק שעלול ליפול מתרחש לפני כל חלק אשר משנה את האובייקט. גישה זו היא הרחבה טבעית של הגישה הקודמת כאשר אי אפשר לבדוק ארגומנטים ללא ביצוע חלק מהחישוב. לדוגמה, חשוב על המקרה של TreeMap, אשר האלמנטים שלו מסודרים לפי הוראה מסוימת. כדי להוסיף אלמנט ל-TreeMap, האלמנט חייב להיות מטיפוס אשר אפשר להשוות אותו תוך שימוש בהוראה של TreeMap. ניסיון להוסיף אלמנט מטיפוס לא נכון יפול באופן מובן מאליו עם ClassCastException, כתוצאה מחיפוש אחר האלמנט בעץ, לפני שהעץ עבר התאמה באופן כלשהו.

גישה שלישית והרבה פחות מקובלת להשגת failure atomicity היא לכתוב **קוד התאוששות**, אשר בולם כישלון אשר קורה באמצע פעולה וגורם לאובייקט להחזיר את מצבו לנקודה שבה היה לפני שהפעולה התחילה. גישה זו משמשת בעיקר למבני נתונים שנעשה בהם שימוש ממושך.

גישה נוספת ואחרונה להשגת failure atomicity היא לבצע את הפעולה על עותק זמני של האובייקט ולהחליף את התוכן של האובייקט בתוכן של האובייקט הזמני לאחר שהפעולה הסתיימה. פעולה זו מתרחשת בטבעיות כאשר החישוב יכול להתבצע יותר מהר, ברגע שהנתונים מאוחסנים במבנה נתונים זמני. לדוגמה, Collection.sort מעתיק את רשימת הקלט שלו לתוך מערך לפני מיון כדי להפחית את עלות הגישה לנתונים בלולאה הפנימית של המיון. הדבר מתבצע כדי לשפר את הביצועים, אבל, כערך מוסף, זה מבטיח שרשימת הקלט לא תיפגע אם המיון נכשל.

למרות ש- failure atomicity היא בדרך כלל רצויה, לא תמיד אפשר להשיג אותה. לדוגמה, אם שתי מטלות מנסות לשנות את אותו האובייקט בו-זמנית ללא סינכרוניזציה מתאימה, האובייקט עלול להישאר במצב לא עקבי.
לכן, יהיה זה מוטעה להניח שעדיין אפשר להשתמש באובייקט לאחר תפיסת ConcurrentModificationException. בדרך כלל, אי אפשר להתאושש משגיאות - Error (בניגוד לחריגים - Exceptions), ושיטות אפילו לא צריכות לנסות לשמור failure atomicity כאשר הן מעוררות שגיאות.

אפילו במקומות שבהם failure atomicity היא אפשרית, היא אינה תמיד רצויה. עבור כמה פעולות, הדבר עלול להגדיל עלות או מורכבות באופן משמעותי. בכל אופן, לעיתים קרובות פשוט וקל להשיג failure atomicity ברגע שאתה מודע לנושא זה. ככלל, כל חריג שהוא חלק ממפרט של שיטה צריך להשאיר את האובייקט באותו המצב שבו היה לפני הקריאה לשיטה. במקום שבו כלל זה מופר, התיעוד של ה-API צריך לציין בבירור באיזה מצב צריך האובייקט להישאר. לרוע המזל, הרבה תיעודים של API קיימים אינם ממלאים אידיאל זה.

נושא 47: אל תתעלם מחריגים

למרות שעצה זו עשויה להיראות מובנת מאליה, היא מופרת לעיתים קרובות עד כדי כך שראוי לחזור עליה. כאשר המתכננים של API מצהירים על שיטה שמעוררת חריג, הם מנסים לומר לך משהו. אל תתעלם מזה! קל להתעלם מחריגים על ידי הקפת הקריאה לשיטה במשפט try עם בלוק catch ריק:

```
//Empty catch block ignores exception -Highly suspect!
try {
    ...
} catch (SomeException e) {
}
```

בלוק catch ריק מחמיץ את המטרה של חריגים, שהיא להכריח אותך לטפל במצבים יוצאי דופן. התעלמות מחריג דומה להתעלמות מאזעקת שריפה - וכיבוי שלה כך שלאף אחד אחר לא יהיה סיכוי לראות האם באמת יש אש. יכול להיות שתינצל מזה אבל גם יכול להיות שהתוצאות תהיינה הרסניות. בכל פעם שאתה רואה בלוק catch ריק, פעמון אזעקה צריך לצלצל בראשך. **לכל הפחות, בלוק ה-catch צריך להכיל הערה שמסבירה מדוע מתאים להתעלם מהחריג**.

מקרה לדוגמה אשר בו יכול להיות שמתאים להתעלם מחריג הוא ציור תמונות לאנימציה. אם המסך מעודכן בפרקי זמן רגילים, הדרך הטובה ביותר לטפל בשגיאה ארעית היא פשוט להתעלם ממנה ולחכות לעדכון הבא.

העצה שבנושא זה חלה הן על חריגים מסומנים והן על חריגים לא מסומנים. בין אם חריג מייצג מצב חריג שניתן לחיזוי ובין אם החריג מייצג שגיאת תכנות, התעלמות ממנו על ידי בלוק catch ריק תתבטא בתוכנית אשר ממשיכה לרוץ בשקט תוך הסתרה של טעות. התוכנית עשויה לקרוס בעתיד בנקודת זמן שרירותית ובנקודה בקוד אשר ייתכן שלא תספק שום מידע על הקשר בין הקוד לטעות. טיפול נכון בחריג יכול למנוע לחלוטין כישלונות.

פרק 9

מטלות (Threads)

מטלות מאפשרות לפעולות מרובות להתנהל בו-זמנית באותה התוכנית. תכנות מרובה מטלות הוא יותר קשה מתכנות בעל מטלה יחידה, לכן העצה של נושא 30 מתאימה לכאן במיוחד: אם יש ספריית מחלקה אשר יכולה לחסוך לך טיפול ברמה הנמוכה של תכנות מרובה מטלות, בוודאי שאתה צריך להשתמש בה. מחלקת java.util.Timer היא דוגמה אחת וחבילת התוכנה util.concurrent [Lea01] של Doug Lea, היא אוסף שלם של עזרי מטלות ברמה גבוהה. אפילו אם אתה משתמש בכאלו ספריות כאשר הדבר מתאים, עדיין תצטרך לכתוב או לתחזק קוד מרובה מטלות מעת לעת. פרק זה מכיל עצות שתסייענה לך לכתוב תוכניות מרובות מטלות ברורות, נכונות ומתועדות היטב.

נושא 48: סנכרן גישה לנתונים משתנים משותפים

מילת המפתח synchronized מבטיחה שרק מטלה בודדת תבצע משפט או בלוק בזמן נתון. מתכנתים רבים חושבים על סינכרוניזציה **אך ורק** כעל אמצעי של מניעה הדדית, כדי למנוע שמירת אובייקט במצב לא עקבי בעוד מטלה אחרת משנה אותו. מנקודת מבט זו, אובייקט נוצר במצב עקבי (נושא 13) וננעל על ידי השיטות שניגשות אליו. שיטות אלו מאבחנות את המצב וגורמות באופן אופציונלי ל**שינוי מצב**, והכוונה לשינוי מצב האובייקט ממצב עקבי אחד למצב עקבי אחר. שימוש נכון בסינכרוניזציה מבטיח שאף שיטה לא תשאיר אף פעם את האובייקט במצב לא עקבי.

נקודת מבט זו היא נכונה, אבל אין היא מספרת את כל הסיפור. לא רק שסינכרוניזציה מונעת ממטלה לשנות אובייקט שנמצא במצב לא עקבי, אלא היא גם מבטיחה שאובייקטים יתקדמו ממצב עקבי למשנהו על ידי רצף מסודר של מעברי מצב, אשר נראים כמתבצעים ברציפות. כל מטלה אשר נכנסת לשיטה או בלוק מסונכרנים רואה את ההשפעות של כל מעברי המצב הקודמים שנשלטים על ידי אותה הנעילה. לאחר שמטלה יוצאת מהאזור המסונכרן, כל מטלה אשר נכנסת לאזור אשר מסונכרן על ידי אותה הנעילה רואה את מעברי המצב שנגרמו על ידי אותה המטלה, אם ישנם כאלה.

השפה מבטיחה שקריאה או כתיבת משתנה יחיד היא אטומית אלא אם כן המשתנה הוא מטיפוס long או double. במילים אחרות, קריאת משתנה אחר מאשר long או double מבטיחה להחזיר ערך אשר אוחסן בתוך המשתנה הזה על ידי איזו שהיא מטלה, אפילו אם מטלות מרובות משנות את המשתנה בו-זמנית ללא סינכרוניזציה.

אתה עשוי לשמוע שכדי לשפר ביצועים, אתה צריך להימנע משימוש בסינכרוניזציה בעת קריאה או כתיבת נתונים אטומיים. עצה זו מוטעית באופן מסוכן. למרות שאחריות ה-atomicity מבטיחה שמטלה לא תראה ערך אקראי בעת קריאת נתונים אטומיים, היא לא מבטיחה שערך אשר נכתב על ידי מטלה אחת ייראה על ידי אחרת: **סינכרוניזציה נדרשת עבור תקשורת אמינה בין מטלות כמו גם עבור הפעלה יחידנית הדדית**. זוהי תוצאה של היבט טכני למדי של שפת Java אשר ידוע כ**מודל הזיכרון** [JLS ,17]. למרות שסביר שמודל הזיכרון יעבור שינוי ניכר בגירסה קרובה [Pugh01a], כמעט ודאי שעובדה זו לא תשתנה. guarantee

התוצאות של אי הצלחה לסנכרן גישה למשתנה משותף יכולות להיות נוראיות אפילו אם המשתנה ניתן לכתיבה ולקריאה באופן atomically. התבונן באמצעי יצירת מספר סידורי הבא:

```
//Broken - requires synchronization!
private static int nextSerialNumber = 0;

public static int generateSerialNumber() {
    return nextSerialNumber++;
}
```

התכלית של אמצעי זה היא להבטיח שכל קריאה של generateSerialNumber תחזיר מספר סידורי שונה, כל עוד אין יותר מ- 2 בחזקת 32 קריאות. סינכרוניזציה אינה נדרשת כדי להגן על הקבועים של מחולל המספרים הסידוריים בגלל שאין כאלה. המצב שלו מורכב משדה atomically writable (nextSerialNumber) וכל ערך אפשרי של שדה זה הוא חוקי. בכל אופן, השיטה לא יכולה לעבוד ללא סינכרוניזציה. אופרטור הקידום (++) גם כותב וגם קורא את השדה nextSerialNumber, ולכן הוא לא אטומי. הקריאה והכתיבה הן פעולות עצמאיות שמתבצעות ברצף. לכן, מטלות בו-זמניות מרובות יכולות לשמר את השדה nextSerialNumber עם אותו הערך ולהחזיר את אותו מספר סידורי.

באופן יותר מפתיע, מטלה אחת יכולה לקרוא ל-generateSerialNumber שוב ושוב ולקבל רצף של מספרים סידוריים מאפס עד ל- n, אחרי שמטלה אחרת קוראת ל-generateSerialNumber ומקבלת מספר סידורי אפס. ללא סינכרוניזציה, המטלה השנייה לא תראה אף אחד מהעדכונים שנעשו בידי המטלה הראשונה. זוהי תוצאה של נושא מודל הזיכרון שהוזכר לעיל.

תיקון שיטת generateSerialNumber הוא קל כמו הוספת המגדיר synchronized להצהרה שלה. הדבר מבטיח שקריאות מרובות לא יתנגשו ושכל קריאה תיראה את ההשפעות של כל הקריאות הקודמות. כדי לחסן את השיטה, יהיה זה חכם גם להשתמש ב-long במקום int או לעורר חריג אם nextSerialNumber עומד לגלוש מ-int.

לאחר מכן, חשוב על התהליך של עצירת מטלה. למרות שהפלטפורמה מספקת שיטות לעצירת מטלה שלא באופן רצוני, שיטות אלו יצאו משימוש (deprecated) מכיון שהן **אינן בטוחות** מטבען - השימוש בהן יכול להביא להשחתת אובייקטים. השיטה המומלצת לעצירת מטלה היא פשוט לגרום לשיטה לקרוא שדה כלשהו שהערך שלו יכול להשתנות, כדי לציין שהשיטה צריכה לעצור בעצמה. באופן טיפוסי השדה הוא boolean או ייחוס אובייקט. מכיון שקריאה וכתיבת שדה כזה היא אטומית, מתכנתים מסוימים מתפתים לוותר על סינכרוניזציה כאשר ניגשים לשדה. לכן, אין זה בלתי מקובל לראות קוד אשר נראה כמו זה:

```
//Broken -requires synchronization!
public class StoppableThread extends Thread {
  private boolean stopRequested =false;

  public void run(){
      boolean done =false;

      while (!stopRequested &&!done){
         ...//do what needs to be done.
      }
  }

  public void requestStop(){
    stopRequested =true;
  }
}
```

הבעיה עם הקוד הזה היא שבהיעדר סינכרוניזציה, אין שום הבטחה שהמטלה בת-העצירה "תראה", אם בכלל, שינוי בערך של stopRequested אשר נעשה על ידי מטלה אחרת. כתוצאה, השיטה requestStop עשויה להיות לגמרי לא אפקטיבית. לא סביר שתבחין בהתנהגות הבעייתית בפועל (אלא אם כן אתה מריץ במערכת עם יותר ממעבד אחד), אבל אין שום הבטחה לכך.

הדרך הברורה לתקן את הבעיה היא פשוט לסנכרן כל גישה לשדה stopRequested:

```
//Properly synchronized cooperative thread termination
public class StoppableThread extends Thread {
  private boolean stopRequested = false;

  public void run() {
      boolean done = false;

      while (!stopRequested() && !done) {
       ...//do what needs to be done.
      }
  }

  public synchronized void requestStop(){
    stopRequested =true;
  }

  private synchronized boolean stopRequested(){
    return stopRequested;
  }
}
```

שים לב שהפעולות של כל אחת מהשיטות המסונכרנות הן אטומיות: הסינכרוניזציה משמשת רק בגלל השפעותיה על התקשורת, לא בגלל מניעת הפעלה הדדית.
ברור שהקוד המתוקן עובד ושהעלות של הסינכרון על כל חזרה של הלולאה כמעט שאינה מובחנת. למרות זאת, יש חלופה נכונה נוספת שקצת פחות מרבה במילים ושביצועיה יכולים להיות מעט טובים יותר. אפשר להשמיט את הסינכרוניזציה אם stopRequested מוצהרת כ-volatile. המגדיר volatile מבטיח שכל מטלה אשר תקרא שדה תראה את הערך שנכתב הכי לאחרונה.

הנזק שיכול להיגרם בגלל כישלון בסינכרון גישה ל-stopRequested בדוגמה הקודמת הוא שולי יחסית. ההשפעה של השיטה requestStop יכולה להתעכב עד אין סוף. הנזק שנגרם בגלל כישלון לסנכרן גישה לנתונים משותפים משתנים יכול להיות חמור בהרבה. התבונן בקוד בדיקה כפולה עבור אתחול עצל:

```
//The double-check idiom for lazy initialization - broken!
private static Foo foo = null;

public static Foo getFoo() {
    if (foo ==null){
        synchronized (Foo.class) {
            if (foo ==null)
                foo =new Foo();
        }
    }
    return foo;
}
```

הרעיון שעומד מאחורי קוד זה הוא שאתה יכול להימנע מהעלות של סינכרוניזציה במקרה המקובל של גישה לשדה (foo), לאחר שהוא אותחל. סינכרוניזציה משמשת רק כדי למנוע ממטלות מרובות לאתחל את השדה. הקוד אכן מבטיח שהשדה יאותחל לכל היותר פעם אחת ושכל המטלות שקוראות ל-getFoo תקבלנה את הערך הנכון עבור ייחוס האובייקט. לרוע המזל, לא מובטח שייחוס האובייקט יעבוד כיאות. קטעים שאינם מוגדרים כמסונכרנים יכולים להתבצע במקביל לבלוק המסונכרן. שיטה שתנסה לקרוא את foo אחרי שמטלה אחרת בנתה (או התחילה לבנות) אותו, לא תכנס לקטע המסונכרן ותוכל להתבצע גם אם מטלה מסונכרנת אחרת מתבצעת. אם מטלה קוראת את הייחוס ללא סינכרוניזציה ואז קוראת לשיטה על האובייקט המיוחס, השיטה עשויה לשמור את האובייקט במצב מאותחל באופן חלקי ולקרוס באופן הרסני.

העובדה שמטלה יכולה לשמור את האובייקט שנבנה בעצלתיים במצב מאותחל באופן חלקי היא מאוד לא אינטואיטיבית. האובייקט בנוי במלואו לפני שהייחוס "יוצא לאור" בשדה שממנו הוא נקרא על ידי מטלות אחרות (foo). אבל בהיעדר סינכרוניזציה, קריאת ייחוס אובייקט ש"יצא לאור" לא מבטיחה שמטלה תראה את כל הנתונים שאוחסנו בזיכרון, לפני ההוצאה לאור של ייחוס האובייקט. במיוחד, קריאת ייחוס אובייקט שיצא לאור לא מבטיחה שהמטלה הקוראת תראה את הערכים האחרונים ביותר של הנתונים, אשר מייסדים את התכונות הפנימיות של האובייקט המיוחס. באופן כללי, **קוד הבדיקה הכפולה לא עובד**, למרות שהוא כן עובד אם המשתנה המשותף מכיל ערך פשוט במקום ייחוס אובייקט [Pugh1b].

יש כמה דרכים לתקן את הבעיה. הקלה שבהן היא לוותר על האתחול העצל לחלוטין:

```
//Normal static initialization (not lazy)
private static final Foo foo = new Foo();

public static Foo getFoo() {
    return foo;
}
```

הדבר בטוח עובד, ושיטת getFoo היא מהירה עד כמה שאפשר. היא לא מבצעת סינכרוניזציה וגם לא חישוב. כפי שנדון בנושא 37, אתה צריך לכתוב תוכניות פשוטות, ברורות ונכונות ולהשאיר את האופטימיזציה לסוף ולבצע אופטימיזציה רק אם מדידה מראה שהדבר הכרחי. לכן, ויתור על אתחול עצל הוא בדרך כלל הגישה הטובה ביותר. אם אתה מוותר על אתחול עצל, מודד את הביצועים ומגלה שהם אינם מספיק טובים, הדבר הבא הטוב ביותר הוא להשתמש בשיטה מסונכרנת כיאות כדי לבצע אתחול עצל:

```
//Properly synchronized lazy initialization
private static Foo foo = null;

public static synchronized Foo getFoo() {
    if (foo == null)
        foo = new Foo();
    return foo;
}
```

מובטח ששיטה זו תעבוד, אבל היא תובעת את העלות של סינכרוניזציה בכל קריאה. במימושי JVM מודרניים, עלות זו יחסית קטנה.
בכל אופן, אם החלטת לאחר אומדן של ביצועי המערכת שלך שאינך יכול להרשות את העלות של אתחול רגיל ולא את העלות של סינכרון כל גישה, אז יש אפשרות אחרת. קוד מחלקת initialize-on-demand holder מתאים לשימוש כאשר יקר לאתחל שדה סטטי וייתכן שלא יהיה בו צורך, אבל ייעשה בו שימוש אינטנסיבי אם יהיה בו צורך. קוד זה מוצג להלן:

```
//The initialize-on-demand holder class idiom
private static class FooHolder {
    static final Foo foo =new Foo();
}

public static Foo getFoo() { return FooHolder.foo; }
```

הקוד מנצל את ההבטחה שמחלקה לא תאותחל עד שנעשה בה שימוש [JLS, 12.4.1]. כאשר השיטה getFoo נקראת בפעם הראשונה, היא קוראת את השדה FooHolder.foo, בזמן שהיא גורמת למחלקה FooHolder להיות מאותחלת. היופי שבקוד זה הוא שהשיטה getFoo אינה מסונכרנת ומבצעת רק גישה לשדה, ולכן אתחול עצל למעשה לא מוסיף דבר לעלות של גישה. החיסרון היחיד של הקוד הוא שהוא אינו עובד עבור שדות מופע, רק על שדות סטטיים.

לסיכום, **בכל פעם שמטלות מרובות חולקות נתונים משתנים, כל מטלה אשר קוראת או כותבת את הנתונים חייבת לבצע נעילה**. אל תתן לערבות של קריאות או כתיבות אטומיות להרתיע אותך מפני ביצוע סינכרוניזציה מתאימה. ללא סינכרוניזציה, אין שום ערובה ששינויים מסוימים של מטלה אחת יישמרו על ידי מטלה אחרת, אם בכלל. liveness וכשלי אבטחה יכולים להיגרם עקב גישת נתונים לא מסונכרנת. שחזור כשלים כאלה יהיה קשה במידה קיצונית. הם עשויים להיות תלויים בתזמון ויהיו מאוד תלויים בפרטים של מימוש ה-JVM והחומרה שעליה הם רצים.

השימוש במגדיר volatile יוצר חלופה בת-קיום לסינכרוניזציה רגילה בנסיבות מסוימות, אבל זוהי טכניקה מתקדמת. כמו כן, היקף אפשרות היישום שלו לא ייוודע עד שתושלם העבודה המתמשכת על מודל הזיכרון.

נושא 49: הימנע מסינכרוניזציה מוגזמת

נושא 48 מזהיר מפני הסכנות של סינכרוניזציה לא מספקת. נושא זה נוגע לבעיה ההפוכה. תלוי במצב, סינכרוניזציה מוגזמת יכולה לגרום לביצועים מופחתים, קיפאון או אפילו התנהגות לא צפויה.

כדי להימנע מסכנה של קיפאון, לעולם אל תוותר על שליטה לטובת הלקוח מתוך בלוק או שיטה מסונכרנים. במילים אחרות, בתוך אזור מסונכרן, אל תקרא לשיטה מוגנת או ציבורית אשר מתוכננת להידרס (באופן טיפוסי שיטות כאלו הן מופשטות, אבל לפעמים יש להן מימוש ברירת מחדל). מנקודת המבט של המחלקה שמכילה את האזור המסונכרן, שיטה כזו היא **זרה (alien)**. למחלקה אין שום ידע לגבי מה שהשיטה עושה ואין לה אף שליטה עליה. לקוח יכול לספק מימוש של שיטה זרה אשר יוצר מטלה אחרת אשר קוראת בחזרה לתוך המחלקה. אז, המטלה שנוצרה יכולה להשיג את אותה הנעילה שמוחזקת על ידי המטלה המקורית, דבר שיגרום למטלה שנוצרה לאחרונה להיחסם. קיפאון יתרחש אם השיטה שיצרה את המטלה מחכה למטלה שתסתיים.

כדי להמחיש זאת, חשוב על המחלקה הבאה אשר מממשת **work queue**. מחלקה זו מאפשרת ללקוחות למקם בתור פריטי work לעיבוד אסינכרוני. השיטה enqueue תיקרא ככל שהדבר יידרש. הבנאי מתחיל במטלת רקע אשר מסירה פריטים מהתור, בסדר שבו הם מוקמו בתור ומעבדת אותם על ידי קריאה לשיטת processItem. כאשר תור העבודה אינו נצרך יותר, הלקוח קורא לשיטה stop כדי לבקש מהמטלה להסתיים בצורה יפה, לאחר השלמת פריט עבודה כלשהו בעיצומו.

```
public abstract class WorkQueue {
   private final List queue = new LinkedList();
   private boolean stopped = false;

   protected WorkQueue() { new WorkerThread().start(); }

   public final void enqueue(Object workItem) {
      synchronized (queue) {
         queue.add(workItem);
         queue.notify();
      }
   }

   public final void stop()  {
      synchronized (queue) {
         stopped = true;
         queue.notify();
      }
   }
```

```
    protected abstract void processItem(Object workItem)
        throws InterruptedException;

    // Broken - invokes alien method from synchronized block!
    private class WorkerThread extends Thread {
        public void run() {
            while (true) {  // Main loop
                synchronized (queue) {
                    try {
                        while (queue.isEmpty() && !stopped)
                            queue.wait();
                    } catch (InterruptedException e) {
                        return;
                    }

                    if (stopped)
                        return;

                    Object workItem = queue.remove(0);
                    try {
                        processItem(workItem); // Lock held!
                    } catch (InterruptedException e) {
                        return;
                    }
                }
            }
        }
    }
}
```

כדי להשתמש במחלקה זו, אתה חייב ליצור לה תת-מחלקה שתספק את המימוש של השיטה המופשטת processItem. לדוגמה, תת-המחלקה הבאה מדפיסה כל פריט של work, תוך שהיא מדפיסה לא יותר מפריט אחד לשני, לא משנה עד כמה הפריטים ממוקמים בתור לעיתים תכופות:

```
class DisplayQueue extends WorkQueue {
    protected void processItem(Object workItem)
            throws InterruptedException {
        System.out.println(workItem);
        Thread.sleep(1000);
    }
}
```

מכיון שמחלקת WorkQueue קוראת לשיטה המופשטת processItem מתוך בלוק מסונכרן, היא נוטה לקיפאון. תת-המחלקה הבאה תגרום לה לקיפאון על ידי האמצעים שתוארו לעיל:

```
class DeadlockQueue extends WorkQueue {
   protected void processItem(final Object workItem)
         throws InterruptedException {
      // Create a new thread that returns workItem to queue
      Thread child = new Thread() {
         public void run() { enqueue(workItem); }
      };
      child.start();
      child.join(); // Deadlock!
   }
}
```

דוגמה זו מוצלחת מכיון שאין שום סיבה לשיטת processItem ליצור מטלת רקע, אבל הבעיה היא ממשית. קריאה לשיטות שמסופקות באופן חיצוני מתוך בלוקים מסונכרנים גרמה להרבה קיפאונות במערכות ממשיות, כמו ערכת כלי GUI. למרבה המזל, קל לתקן את הבעיה. פשוט העבר את קריאת השיטה מחוץ לבלוק המסונכרן, כפי שמוצג להלן:

```
// Alien method outside synchronized block - "Open call"
   private class WorkerThread extends Thread {
      public void run() {
         while (true) {  // Main loop
            Object workItem = null;
            synchronized (queue) {
               try {
                  while (queue.isEmpty() && !stopped)
                     queue.wait();
               } catch (InterruptedException e) {
                  return;
               }
               if (stopped)
                  return;
               workItem = queue.remove(0);
            }
            try {
               processItem(workItem); // No lock held
            } catch (InterruptedException e) {
               return;
            }
         }
      }
   }
```

שיטה זרה אשר נקראת מחוץ לאזור מסונכרן ידועה כ**קריאה פתוחה** [Lea00, 2.4.1.3]. מלבד היותה מונעת קיפאונות, היא יכולה גם להגדיל שיתוף פעולה. שיטה זרה יכולה לרוץ במשך פרק זמן שרירותי ארוך, במשך אותו הזמן שבו תמנע שלא לצורך את הגישה ממטלות אחרות לאובייקט המשותף, אם השיטה הזרה היתה נקראת בתוך האזור המסונכרן.

ככלל, אתה צריך לבצע כמה שפחות עבודה בתוך אזורים מסונכרנים. קבל את הנעילה, בחן את הנתונים המשותפים, שנה את הנתונים כפי הצורך והשלך את הנעילה. אם אתה חייב לבצע פעילות כלשהי שצורכת זמן, מצא דרך להעביר את הפעילות אל מחוץ לאזור המסונכרן.

קריאה לשיטה זרה מתוך אזור מסונכרן יכולה לגרום לכישלונות יותר חמורים מאשר קיפאונות, אם השיטה הזרה נקראת בזמן שהקבועים שמוגנים על ידי האזור המסונכרן אינם תקפים באופן זמני (דבר זה אינו יכול לקרות בדוגמה work queue השבורה, מכיון שהתור הוא במצב עקבי כאשר processItem נקראת). כישלונות כאלו לא מערבים יצירת מטלה חדשה מתוך השיטה הזרה. הם מתרחשים כאשר השיטה הזרה עצמה קוראת בחזרה למחלקה הפגומה. בגלל שנעילות בשפת Java הן **רקורסיביות**, קריאות כאלו לא קופאות כמו שהיה קורה אילו היו נוצרות על ידי מטלה אחרת. המטלה הקוראת כבר מחזיקה בנעילה, לכן המטלה תצליח כאשר היא מנסה להשיג את הנעילה פעם שנייה, למרות שיש פעולה אחרת לא קרובה מבחינה מושגית בהתקדמות על הנתונים שמוגנים על ידי הנעילה. התוצאות של כישלון כזה יכולות להיות הרסניות. בעיקרן, הנעילה נכשלה בביצוע משימתה. נעילות רקורסיביות מפשטות את הבנייה של תוכניות מוכוונות עצמים מרובות מטלות, אבל הן יכולות להפוך כישלונות שהם חיוניים לכישלונות אבטחה.

החלק הראשון של נושא זה היה על בעיות שיתוף פעולה. כעת נפנה את תשומת לבנו לביצועים. למרות שהעלות של סינכרוניזציה ירדה מאוד מאז ימיה הראשונים של פלטפורמת Java, היא לעולם לא תיעלם לחלוטין. אם פעולה שנעשה בה שימוש תכוף מסונכרנת שלא לצורך, לדבר יכולה להיות השפעה ניכרת על הביצועים. לדוגמה, חשוב על המחלקות StringBuffer ו-BufferedInputStream. מחלקות אלו הן בטוחות-מטלה (נושא 52) אבל כמעט תמיד משתמשת בהן מטלה יחידה, לכן הנעילה שהן מבצעות היא בדרך כלל מיותרת. הן תומכות בשיטות מפוררות היטב, שפועלות ברמת התו או הסיבית הבודדת, לכן לא רק שמחלקות אלו נוטות לבצע נעילות מיותרות, אלא הן גם נוטות לעשות הרבה נעילות מיותרות. הדבר יכול להתבטא באובדן ביצועים משמעותי. מאמר אחד דיווח על אובדן קרוב ל- 20% ביישום ממשי [Heydon 99]. לא סביר שתראה אובדן ביצועים כזה דרמטי שנגרם בגלל סינכרוניזציה מיותרת, אבל אובדן של בין 5 ל- 10 אחוזים הוא בגדר האפשר.

באופן שניתן להתווכח לגביו, הדבר שייך ל"התייעלויות הקטנות" שאנו צריכים לשכוח מהן (פרק 7, נושא 37).

בכל אופן, אם אתה כותב הפשטה של רמה נמוכה אשר תשמש בדרך כלל מטלה בודדת או כמרכיב באובייקט מסונכרן גדול יותר, אז אתה צריך לשקול הימנעות מסינכרון פנימי של המחלקה. בין אם החלטת לסנכרן מחלקה או לא, קריטי שתתעד את תכונות בטיחות-המטלה שלה (נושא 52).

לא תמיד ברור האם מחלקה נתונה צריכה לבצע סינכרוניזציה פנימית. אם להשתמש במינוח של נושא 52, אז לא תמיד ברור האם מחלקה צריכה להיעשות **בטוחת-מטלה** או **תואמת-מטלה**. הנה כמה קווים מנחים אשר יעזרו לך לבחור באופן הנכון.

אם אתה כותב מחלקה שייעשה בה שימוש רב בנסיבות בהן נדרשת סינכרוניזציה וגם בנסיבות בהן לא נדרשת סינכרוניזציה, גישה הגיונית תהיה לספק הן משתנים מסונכרנים (בטוחי מטלה) והן משתנים לא מסונכרנים (תואמי מטלה). דרך אחת לעשות זאת היא לספק **מחלקת עטיפה** (נושא 14), אשר מממשת ממשק אשר מתאר את המחלקה ומבצעת סינכרוניזציה מתאימה לפני העברת קריאות לשיטה אל השיטה המקבילה של האובייקט העטוף. זו הגישה אשר ננקטת על ידי סביבת העבודה של האוספים. באופן שניתן להתווכח לגביו, גישה זו היתה צריכה להינקט גם על ידי java.util.Random. גישה שנייה שמתאימה למחלקות שלא תוכננו להיות מורחבות או ממומשות מחדש, היא לספק מחלקה לא מסונכרנת ותת-מחלקה שמורכבת משיטות מסונכרנות בלבד, אשר קוראות לשיטות המקבילות שלהן במחלקת-העל.

סיבה אחת טובה לסנכרן מחלקה באופן פנימי היא בגלל שהמחלקה מיועדת לשימוש בו-זמנית רב, ואתה יכול להשיג בו-זמניות גבוהה יותר במידה ניכרת על ידי ביצוע סינכרוניזציה פנימית מפוררת היטב. לדוגמה, אפשר לממש טבלת ערבול שאינה ניתנת להרחבה אשר באופן עצמאי מסנכרנת גישה לכל מנה. הדבר מספק בו-זמניות גדולה בהרבה מאשר נעילה של כל הטבלה כדי לגשת לכניסה יחידה.

אם מחלקה או שיטה סטטית מסתמכת על שדה **סטטי** ניתן לשינוי, היא חייבת לעבור סינכרון באופן פנימי, אפילו אם באופן טיפוסי משתמשת בה מטלה יחידה. שלא כמו מופע משותף, הלקוח לא יכול לבצע סינכרון חיצוני בגלל שאין שום ערובה לכך שלקוחות אחרים יעשו את אותו הדבר. השיטה הסטטית Math.random מדגימה מצב זה.

לסיכום, כדי להימנע מקיפאון והשחתת נתונים, לעולם אל תקרא לשיטה זרה מתוך אזור מסונכרן. באופן יותר כללי, נסה להגביל את כמות העבודה שאתה מבצע מתוך אזורים מסונכרנים. כאשר אתה מתכנן מחלקה ניתנת לשינוי, חשוב האם היא צריכה לעשות את הסינכרוניזציה העצמית שלה. החיסכון בעלויות שאתה יכול לקוות להשיג על ידי ויתור על סינכרוניזציה הן כבר לא עצומות כבעבר, אבל הן בהחלט ניתנות למדידה. בסס את ההחלטה שלך על האם השימוש הראשוני של ההפשטה יהיה מרובה מטלות ותעד את החלטתך בבירור.

נושא 50: לעולם אל תקרא ל- wait מחוץ ללולאה

השיטה Object.wait משמשת ליצירת מטלת wait עבור מצבים מסוימים. היא חייבת להיקרא בתוך אזור מסונכרן אשר נועל את האובייקט שלגביו הוא נקרא. **זהו הקוד הסטנדרטי לשימוש בשיטה wait**:

```
synchronized (obj) {
    while (<condition does not hold>)
        obj.wait();

    ...//Perform action appropriate to condition
}
```

תמיד השתמש בקוד לולאה wait כדי לקרוא לשיטה wait. לעולם אל תקרא לה מחוץ ללולאה. הלולאה משמשת כדי לבחון את המצב לפני ואחרי המתנה.

בחינת המצב לפני המתנה ופסיחה על ההמתנה אם המצב אינו דורש המתנה הן חיוניות. אם המצב כבר אינו דורש המתנה ושיטת notify (או notifyAll) כבר נקראה לפני שמטלה ממתינה, אין שום הבטחה שהמטלה אי פעם תתעורר מההמתנה.

בחינת המצב לפני המתנה והמתנה שוב אם המצב דורש המתנה הן חיוניות כדי להבטיח **בטיחות**. אם המטלה ממשיכה בפעולה כאשר המצב דורש המתנה, זה יכול להרוס את הקבועים שמוגנים על ידי הנעילה. יש כמה סיבות לכך שמטלה עשויה להתעורר כאשר המצב דורש המתנה:

- מטלה אחרת קיבלה את הנעילה ושינתה את המצב המוגן בין הזמן שבו מטלה קראה ל-notify לבין הזמן שבו מטלת ההמתנה התעוררה.
- מטלה אחרת קראה ל-notify בטעות או בזדון כאשר המצב דורש המתנה. מחלקות חושפות את עצמן לסוג זה של נזק כאשר הן ממתינות על אובייקטים נגישים ציבורית. כל wait שכלולה בתוך שיטה מסונכרנת של אובייקט שנגיש באופן ציבורי רגישה לבעיה זו.
- המטלה המודיעה היא "נדיבה" יתר על המידה בלהעיר (waking) מטלות ממתינות. לדוגמה, המטלה המודיעה חייבת לקרוא ל-notifyAll אפילו אם רק כמה מהמטלות הממתינות מרוצות מהמצב שלהן.
- המטלה הממתינה התעוררה בהיעדר הודעה. הדבר ידוע כ**התעוררות מזויפת**. למרות שהמפרט של שפת Java [JLS] לא מזכיר אפשרות זו, הרבה מימושי JVM משתמשים באמצעי threading שבהם ידוע שמתרחשות התעוררויות מזויפות, אף על פי שבאופן נדיר [Posix, 11.4.3.6.1].

נושא שקשור לענייננו הוא האם אתה צריך להשתמש ב-notify או ב-notifyAll כדי לעורר מטלות ממתינות (זכור ש-notify מעוררת מטלה ממתינה יחידה, בהנחה שמטלה זו קיימת ו-notifyAll מעוררת את כל המטלות הממתינות). לעיתים קרובות נאמר שאתה צריך להשתמש **תמיד** ב-notifyAll. עצה זו היא הגיונית ושמרנית, בהנחה שכל קריאות wait הן בתוך לולאות while. היא תמיד תניב תוצאות נכונות מכיון שהיא מבטיחה שאתה תעורר את המטלות שצריך לעורר. אתה עלול לעורר גם כמה מטלות אחרות, אבל הדבר לא ישפיע על נכונות התוכנית שלך. מטלות אלו תבדוקנה את המצב שהן ממתינות לו/בגללו ואם יימצא שהוא שקר, הן תמשכנה להמתין.

כאופטימיזציה, אתה יכול לבחור לקרוא ל-notify במקום ל-notifyAll אם כל המטלות שיכולות להיות במצב ההמתנה ממתינות לאותו המצב ורק מטלה אחת בכל רגע נתון יכולה להרוויח מכך שהמצב נהיה אמיתי. שני תנאים אלה מתמלאים באופן טריוויאלי אם רק מטלה יחידה ממתינה על אובייקט מיוחד (כדוגמת WorkQueue, נושא 50).

אפילו אם תנאים אלה מתמלאים, יכולה להיות סיבה להשתמש ב-notifyAll במקום ב-notify. בדיוק כמו שהצבת קריאת wait בתוך לולאה מגינה מפני הודעות מקריות או זדוניות על אובייקט נגיש באופן ציבורי, כך שימוש ב-notifyAll במקום ב-notify מגן מפני המתנות מקריות או זדוניות על ידי מטלה שאינה קשורה. אחרת, מטלות כאלו עלולות "לבלוע" הודעה קריטית ולגרום למקבל המיועד שלה להמתין עד אין סוף. הסיבה ש**לא** נעשה שימוש ב-notifyAll בדוגמה WorkQueue היא שמטלת העובד ממתינה על **אובייקט פרטי** (queue), ולכן אין שום סכנה של המתנות מקריות או זדוניות.

ישנה אזהרה אחת בנוגע לעצה להשתמש ב-notifyAll בעדיפות על פני notify. למרות שהשימוש ב-notifyAll לא יכול לפגוע בנכונות, הוא יכול לפגוע בביצועים. למעשה, הוא מבזה באופן שיטתי את הביצועים של מבני נתונים מסוימים ממספר לינארי של מטלות ממתינות למספר ריבועי. המחלקות של מבני נתונים שכל כך מושפעות מכך הן אלו שבהן רק מספר מסוים של מטלות מציע מעמד מיוחד מסוים בכל זמן נתון, ומטלות אחרות חייבות להמתין. דוגמאות כוללות מאותתים (semaphores), חוצצים מוגדרים (bounded buffers) ונעילות קריאה-כתיבה (read-write locks).

אם אתה מממש סוג זה של מבנה נתונים ואתה מעורר כל מטלה ברגע שהיא נעשית כשירה ל"מעמד מיוחד", אתה מעורר כל מטלה פעם אחת ל-n התעוררויות. אם אתה מעורר את כל n המטלות כאשר רק אחת יכולה לקבל מעמד מיוחד ושאר n-1 המטלות חוזרות להמתין, אתה תסיים עם n+(n-1)+(n-2)...+1 התעוררויות בזמן שבו כל המטלות הממתינות הסכימו למעמד מיוחד. סכום הסדרה הזו הוא $O(n^2)$. אם אתה יודע שמספר המטלות תמיד יהיה קטן, לא אמורה להיות בעיה בפועל, אבל אם אין לך ערובה לכך שמספר המטלות תמיד יהיה קטן אז חשוב שתשתמש באסטרטגיית עירור (wakeup strategy) יותר סלקטיבית.

אם כל המטלות שמתחרות על מעמד מיוחד הן שקולות מבחינה לוגית, אז עליך רק להשתמש בזהירות ב-notify במקום ב-notifyAll. בכל אופן, אם רק כמה מהמטלות הממתינות מתאימות למעמד מיוחד בכל זמן נתון, אז אתה חייב להשתמש בתבנית שידועה כ**הודעה ספציפית** [Cargill96, Lea99]. תבנית זו היא מעבר לתחומו של ספר זה.

לסיכום, תמיד קרא ל-wait מתוך לולאת while תוך שימוש בקוד הסטנדרטי. פשוט אין שום סיבה לנהוג אחרת. בדרך כלל, עליך להעדיף שימוש ב-notifyAll על פני notify. בכל אופן, ישנם מצבים בהם זה יגרום לפגיעה ניכרת בביצועים. אם נעשה שימוש ב-notify, חובה להקדיש תשומת לב רבה כדי להבטיח שלא יוכל להיווצר קיפאון.

נושא 51: אל תהיה תלוי במתזמן המטלות

כאשר מטלות מרובות הן בנות-ריצה, מתזמן המטלות קובע איזו מטלה תרוץ וכמה זמן. כל מימוש JVM הגיוני ינסה לקבוע את זה במידה מסוימת של הגינות, אבל המדיניות המדויקת שונה מאוד בין המימושים. לכן, תוכניות מרובות מטלות טובות לא צריכות להיות תלויות בפרטיה של מדיניות זו. **סביר שכל תוכנית אשר מסתמכת על מתזמן המטלות בשביל הביצועים שלה או הנכונות שלה, לא תהיה ניידת**.

הדרך הטובה ביותר לכתוב יישום מרובה מטלות מוצק, נייד ומגיב בחיוב היא להבטיח שבכל זמן נתון תהיינה מעט מטלות בנות-ריצה. הדבר לא מותיר למתזמן המטלות הרבה ברירות: הוא פשוט מריץ את המטלות בנות-הריצה עד שהן כבר לא בנות-ריצה. כתוצאה, התנהגות התוכנית לא ניתנת לשינוי כל כך אפילו תחת אלגוריתמים קיצוניים של תזמון מטלות.

הטכניקה העיקרית לשמור מספר קטן של מטלות בנות-ריצה היא לגרום לכל מטלה לבצע עבודה קטנה בלבד ואז לחכות למצב מסוים תוך שימוש ב-Object.wait, או לחכות לזמן מסוים שיחלוף תוך שימוש ב-Thread.sleep. מטלות לא צריכות להיות במצב **busy-wait**, תוך שהן חוזרות ובודקות שוב ושוב מבנה נתונים כשהן מחכות למשהו שיקרה. מלבד הפיכת התוכנית לפגיעה לגחמותיו של מתזמן המשימות, busy waiting יכולה להגדיל מאוד את העומס על המעבד, דבר שמפחית את סך כל העבודה השימושית שתהליכים אחרים יכולים להשלים באותו המחשב.

הדוגמה work queue בנושא 49 ממלאת אחר המלצות אלו: בהנחה ששיטת processItem שסיפק הלקוח מתנהגת כראוי, המטלה העובדת שסיפק הלקוח מוציאה את רוב זמנה בהמתנה על מוניטור (monitor) לתור שייהפך ללא-ריק. כדוגמה קיצונית לדבר שאסור לעשות, התבונן במימוש מחדש של WorkQueue, אשר עושה שימוש ב- busy waits במקום להשתמש ב-monitor:

```
// HORRIBLE PROGRAM - uses busy-wait instead of Object.wait!
public abstract class WorkQueue {
   private final List queue = new LinkedList();
   private boolean stopped = false;

   protected WorkQueue() { new WorkerThread().start(); }

   public final void enqueue(Object workItem) {
      synchronized (queue) { queue.add(workItem); }
   }
   public final void stop()  {
      synchronized (queue) { stopped = true; }
   }
   protected abstract void processItem(Object workItem)
      throws InterruptedException;
```

```
    private class WorkerThread extends Thread {
        public void run() {
            final Object QUEUE_IS_EMPTY = new Object();
            while (true) {  // Main loop
                Object workItem = QUEUE_IS_EMPTY;
                synchronized (queue) {
                    if (stopped)
                        return;
                    if (!queue.isEmpty())
                        workItem = queue.remove(0);
                }

                if (workItem != QUEUE_IS_EMPTY) {
                    try {
                        processItem(workItem);
                    } catch (InterruptedException e) {
                        return;
                    }
                }
            }
        }
    }
}
```

כדי לתת לך מושג על המחיר שתשלם עבור מימוש כזה, התבונן בבוחן הביצועים הזעיר שלהלן, אשר יוצר שני תורי work ומעביר פריט work הלוך ושוב ביניהם (פריט ה-work שעובר מתור אחד לשני הוא ייחוס לתור הקודם, אשר משמש כאיזה סוג של כתובת מוחזרת). התוכנית רצה במשך עשר שניות לפני התחלת המדידה כדי לאפשר למערכת "להתחמם", ואז היא סופרת את מספר הנסיעות הלוך ושוב מתור לתור במשך עשר השניות הבאות. במחשב שלי, הגירסה הסופית של WorkQueue שבנושא 49 מפגינה 23,000 נסיעות הלוך ושוב לשנייה, בעוד שהמימוש ההפוך לעיל מפגין 17 נסיעות הלוך ושוב לשנייה:

```
class PingPongQueue extends WorkQueue {
    volatile int count = 0;

    protected void processItem(final Object sender) {
        count++;
        WorkQueue recipient = (WorkQueue) sender;
        recipient.enqueue(this);
    }
}
```

```
public class WaitQueuePerf {
   public static void main(String[] args) {
      PingPongQueue q1 = new PingPongQueue();
      PingPongQueue q2 = new PingPongQueue();
      q1.enqueue(q2); // Kick-start the system

      // Give the system 10 seconds to warm up
      try {
         Thread.sleep(10000);
      } catch (InterruptedException e) {
      }

      // Measure the number of round trips in 10 seconds
      int count = q1.count;
      try {
         Thread.sleep(10000);
      } catch (InterruptedException e) {
      }
      System.out.println(q1.count - count);

      q1.stop();
      q2.stop();
   }
}
```

למרות שמימוש WorkQueue לעיל עשוי להיראות קצת מאולץ, אין זה בלתי מקובל לראות מערכות מרובות מטלות עם מטלה אחת או יותר שהן בנות-ריצה שלא לצורך. ייתכן שהתוצאות לא תהיינה קיצוניות כפי שהודגם לעיל, אבל סביר שהביצועים והניידות ייפגעו.

כאשר אתה נתקל בתוכנית שבקושי עובדת מכיון שכמה מטלות לא מקבלות מספיק זמן מעבד, יחסית למטלות אחרות, **התגבר על הפיתוי "לתקן" את התוכנית על ידי הצבת קריאות ל-Thread.yield**. אתה עשוי להצליח לגרום לתוכנית לעבוד, אבל התוכנית שתתקבל תהיה לא ניידת מההיבט של הביצועים. אותן קריאות yield אשר משפרות את הביצועים במימוש JVM אחד עלולות לפגוע בביצועים במימוש שני ולא להשפיע עליהם בכלל במימוש שלישי. ל-Thread.yield אין שום סמנטיקה בת-בחינה. כיוון פעולה יותר טוב הוא לבנות מחדש את היישום כדי להפחית את המספר של מטלות בנות-ריצה בו-זמנית.

טכניקה קרובה, אליה מתייחסות אזהרות דומות, היא להסדיר **קדימויות מטלה**. **קדימויות מטלה היא בין התכונות הכי פחות ניידות של פלטפורמת Java**. אין זה בלתי הגיוני לכוון את ההיענות של יישום על ידי קביעת כמה קדימויות מטלה, אבל זה נחוץ לעיתים נדירות והתוצאות ישתנו ממימוש JVM אחד למשנהו.

לא הגיוני לפתור בעיית חיוניות רצינית על ידי הסדרת קדימויות מטלה. סביר שהבעיה תחזור על עצמה כל עוד לא מצאת ותיקנת את הגורם היסודי.

השימוש היחיד שרוב המתכנתים אי פעם יעשו ב-Thread.yield הוא להגדיל באופן מלאכותי את בו-הזמניות של תוכנית במהלך בדיקה שלה. הדבר חושף באגים על ידי סקירת חלק גדול יותר של מרווח המצב של תוכנית, ולכן מגביר את הביטחון בנכונותה של המערכת. טכניקה זו הוכחה כיעילה מאוד בחשיפת באגים דקים של בו-זמניות.

לסיכום, אל תהיה תלוי במתזמן המטלות בנוגע לנכונות של היישום שלך. היישום שיתקבל לא יהיה איתן ולא יהיה נייד. כמסקנה, אל תסמוך על Thread.yield או על קדימויות מטלה. אמצעים אלה הם פשוט עצות למתזמן. אפשר לעשות בהם שימוש מצומצם כדי לשפר את איכות השירות של מימוש שכבר עובד, אבל אין להשתמש בהם אף פעם כדי "לתקן" תוכנית שבקושי עובדת.

נושא 52: תעד בטיחות מטלה

כיצד מחלקה מתנהגת כאשר המופעים שלה או השיטות הסטטיות שלה חשופים לשימוש בו-זמנית, היא חלק חשוב של ההסכם אשר המחלקה מייסדת עם הלקוח שלה. אם לא תתעד מרכיב זה של התנהגות מחלקה, המתכנתים שישתמשו בה ייאלצו לשער השערות. אם השערות אלו הן מוטעות, התוכנית שתתקבל עלולה לבצע סינכרוניזציה לא מספקת (נושא 48) או סינכרוניזציה מוגזמת (נושא 49). בשני המקרים, עלולות להתרחש שגיאות רציניות.

יש הטוענים שמשתמשים יכולים לקבוע את בטיחות המטלה של שיטה על ידי חיפוש אחרי הנוכחות של המגדיר synchronized בתיעוד שיוצר Javadoc. הדבר מוטעה בגלל כמה סיבות. למרות שתוכנת העזר Javadoc אכן כללה את המגדיר synchronized בפלט שלה בגרסאות שקדמו לגירסה 1.2, זה היה באג והוא תוקן. **הנוכחות של המגדיר synchronized בהצהרה של שיטה היא פרט מימוש, לא חלק מה-API המיוצא**. הנוכחות שלו לא מצביעה באופן שניתן לסמוך עליו שהשיטה היא בטוחת מטלה, היא כפופה לשינויים מגירסה לגירסה.

יתר על כן, הטענה לפיה הנוכחות של מילת המפתח synchronized מספיקה כדי לתעד בטיחות מטלה ממחישה את התפיסה המוטעית המקובלת, לפיה בטיחות מטלה היא תכונת הכל או כלום. **כדי לאפשר שימוש מרובה מטלות בטוח, מחלקה חייבת לתעד בבירור ובאריכות את הרמה של בטיחות מטלה שהיא תומכת בה**.

הרשימה הבאה מסכמת את הרמות של בטיחות מטלה אשר מחלקה יכולה לתמוך. רשימה זו אינה מתיימרת להיות ממצה, אבל היא מכסה את המקרים המצויים. השמות שמשמשים ברשימה אינם סטנדרטיים מכיון שאין מוסכמות מקובלות רחבות בנוגע לתחום זה:

- **אי השתנות** - מופעים של מחלקה זו נראים קבועים ללקוחות שלהם. אין שום צורך בשום סינכרוניזציה חיצונית. דוגמאות כוללות את String, Integer ואת BigInteger (נושא 13).
- **בטיחות מטלה** - מופעים של מחלקה זו הם משתנים, אבל כל השיטות מכילות סינכרוניזציה פנימית מספקת אשר מופעים עשויים להשתמש בה בו-זמנית בלי הצורך בסינכרוניזציה חיצונית. קריאות בו-זמניות עשויות להתבצע באופן סדרתי בסדר עקבי גלובלי מסוים. דוגמאות כוללות את Random ואת java.util.Timer.
- **בטיחות מטלה על תנאי** - כמו בטיחות מטלה, פרט לכך שהמחלקה (או מחלקה קשורה) מכילה שיטות אשר חייבות להיקרא ברצף ללא הפרעה מצד מטלות אחרות. כדי למנוע אפשרות של הפרעה, הלקוח חייב לקבל נעילה מתאימה למשך זמן הרצף. דוגמאות כמו Hashtable ו- Vector, שדורשות סינכרוניזציה חיצונית.

- **תואם-מטלה** - אפשר להשתמש במופעים של מחלקה זו בו-זמנית על ידי הקפת כל קריאת שיטה (ובמקרים מסוימים, כל רצף של קריאות שיטה) על ידי סינכרוניזציה חיצונית. דוגמאות כוללות את מימושי האוסף הרב-שימושי, כמו ArrayList ו-HashMap.
- **עוין-מטלה** - מחלקה זו אינה בטוחה לשימוש בו-זמנית על ידי מטלות מרובות, אפילו אם כל הקריאות לשיטה מוקפות בסינכרוניזציה חיצונית. באופן טיפוסי, עוינות מטלה נובעת מהעובדה ששיטות משנות נתונים סטטיים אשר משפיעים על מטלות אחרות. למרבה המזל, יש מעט מאוד שיטות או מחלקות עוינות מטלה בספריות הפלטפורמה. השיטה System.runFinalizersOnExit היא עוינת מטלה וגונתה בשל כך.

תיעוד מחלקה בטוחת מטלה על תנאי מצריך תשומת לב. אתה חייב לציין אילו רצפי קריאה דורשים סינכרוניזציה חיצונית ואיזו נעילה (או במקרים נדירים, אילו נעילות) חייבים להשיג כדי למנוע גישה בו-זמנית. באופן טיפוסי זוהי הנעילה על המופע עצמו, אבל יש לכך חריגים. אם אובייקט מציג view חילופי על איזה אובייקט אחר, הלקוח חייב להשיג נעילה על אובייקט הגיבוי כדי למנוע שינוי ישיר מאובייקט הגיבוי.
לדוגמה, התיעוד של Hashtable.keys צריך לומר משהו כמו זה:

If there is any danger of another thread modifying this hash table, safely enumerating over its keys requires that you lock the Hashtable instance prior to calling this method, and retain the lock until you are finished using the returned Enumeration, as demonstrated in the following code fragment:

```
Hashtable h =...;

synchronized (h){
    for (Enumeration e = h.keys() ; e.hasMoreElements(); )
            f(e.nextElement());
}
```

נכון לגירסה 1.3, התיעוד של Hashtable לא כולל את הסיפור הזה, אבל אני מקווה שמצב זה יתוקן במהרה. באופן יותר כללי, ספריות הפלטפורמה של Java היו יכולות לעשות עבודה טובה יותר בתיעוד בטיחות המטלה שלהן.

למרות שהשימוש בנעילת אובייקט שנגיש באופן ציבורי מאפשר ללקוחות לבצע רצף של קריאות לשיטה באופן אינדיבידואלי בלעדי, לגמישות זו יש מחיר.

לקוח בעל כוונות זדון יכול לשים התקפת שלילת-שירות פשוט על ידי החזקת הנעילה על האובייקט:

```
//Denial-of-service attack
synchronized (importantObject) {
    Thread.sleep(Integer.MAX_VALUE);  // Disable importantObject
}
```

אם אתה מודאג מהתקפת שלילת-שירות זו, אתה צריך להשתמש באובייקט נעילה פרטי כדי לסנכרן פעולות:

```
//Private lock object idiom - thwarts denial-of-service attack
private Object lock = new Object();

public void foo() {
    synchronized(lock) {
        ...
    }
}
```

בגלל שהנעילה מתבצעת על אובייקט שאינו נגיש ללקוחות, האובייקט המכיל חסין מפני התקפת שלילת-השירות שהוצגה לעיל. שים לב שמחלקות בטוחות מטלה על תנאי תמיד נוטות להתקפה זו, מכיון שהן חייבות לתעד את הנעילה שצריך להחזיק בעת ביצוע רצפי פעולות באופן יחידני. בכל אופן, מחלקות בטוחות מטלה תהיינה בטוחות מפני התקפה זו על ידי השימוש בקוד אובייקט הנעילה הפרטי.

שימוש באובייקטים פנימיים לנעילה מתאים במיוחד למחלקות שתוכננו להורשה (נושא 15) כמו המחלקה WorkQueue שבנושא 49. אם מחלקת-העל היתה משתמשת במופעים שלה לנעילה, תת-מחלקה היתה יכולה מבלי להתכוון להפריע עם הפעולה שלה. על ידי שימוש באותה הנעילה למטרות שונות, מחלקת-העל ותת-המחלקה היו יכולות לסיים ב"דריכה על האצבעות של כל אחת מהן על השנייה".

לסיכום, כל מחלקה צריכה לתעד בבירור את מאפייני בטיחות המטלה שלה. הדרך היחידה לעשות זאת היא לספק תיאורים פרוזאיים שהמילים שלהם נבחרו בקפידה. למגדיר synchronized אין שום חלק בתיעוד בטיחות המטלה של מחלקה. בכל אופן, חשוב שמחלקות בטוחות מטלה על תנאי תתעדנה איזה אובייקט חייב להינעל כדי לאפשר לרצפים של קריאות לשיטה להתבצע באופן יחידני. התיאור של בטיחות המטלה של מחלקה בדרך כלל משתייך להערת התיעוד של המחלקה, אבל שיטות עם מאפייני בטיחות מטלה מיוחדים צריכות לתאר מאפיינים אלה בהערות התיעוד שלהן עצמן.

נושא 53: הימנע מקבוצות מטלה

ביחד עם מטלות, נעילות ומוניטורים, הפשטה בסיסית שמציעה מערכת המטלות היא **קבוצות מטלה**. קבוצות מטלה נחשבו במקור למנגנון שמבודד יישומונים מטעמי בטיחות. הם לעולם לא מילאו הבטחה זו וחשיבות האבטחה שלהם התמעטה עד להיקף כזה שהם אפילו לא מוזכרים בעבודה המקורית של מודל האבטחה של פלטפורמת Java2 [Gong99].

לאחר שהתוודענו לכך שקבוצות מטלה לא מספקות כל תפקודיות אבטחה שאפשר לדון בה, נשאלת השאלה: אם כך, אז איזו תפקודיות הם כן מספקים? בהערכה ראשונית, הם מאפשרים לך לשייך פשוטי Thread לקבוצה של מטלות בפעם אחת. כמה מהפשוטים האלה יצאו מכלל שימוש (deprecated) והשאר לא משמשים לעיתים תכופות. בהשוואה, קבוצות מטלה לא מספקות הרבה במובן של תפקודיות שימושית.

עיוות אירוני הוא שה-API של ThreadGroup חלש מבחינת בטיחות מטלה. כדי לקבל רשימה של המטלות האקטיביות בקבוצת מטלה, אתה חייב לקרוא לשיטה enumerate, אשר מקבלת כפרמטר מערך מספיק גדול כדי להחזיק את כל המטלות האקטיביות. השיטה ActiveCount מחזירה את מספר המטלות האקטיביות בקבוצת מטלה, אבל אין שום ביטחון שמספר זה עדיין יהיה מדויק לאחר שמערך הוקצה והועבר לשיטה enumerate. אם המערך קטן מדי, השיטה enumerate מתעלמת בשקט מכל מטלות נוספות.

ה-API שמשמש לקבלת רשימת תת-קבוצות של קבוצת מטלה, פגום באופן דומה. למרות שאפשר היה לתקן בעיות אלו על ידי הוספת שיטות חדשות, הן לא תוקנו בגלל שאין שום צורך אמיתי בכך: **קבוצות מטלה הן מיושנות במידה רבה**.

לסיכום, קבוצות מטלה לא מספקות הרבה במובן של תפקודיות שימושית והרבה מהתפקודיות שהן כן מספקות היא פגומה. האופן הטוב ביותר להתייחס אל קבוצות מטלה הוא כאל ניסוי לא מוצלח, ואתה יכול פשוט להתעלם מקיומן. אם אתה מתכנן מחלקה אשר מטפלת בקבוצות לוגיות של מטלות, פשוט אחסן את ייחוסי Thread אשר כוללים כל קבוצה לוגית במערך או באוסף. קורא ערני עשוי להבחין שעצה זו נראית כסותרת את זו של נושא 30 שאומרת "דע והשתמש בספריות". במקרה הזה, נושא 30 טועה.

יש חריג קטן לעצה לפיה אתה צריך פשוט להתעלם מקבוצות מטלה. אפשרות קטנה של תפקודיות זמינה **רק** ב-API של ThreadGroup.
השיטה ThreadGroup.uncaughtException נקראת באופן אוטומטי כאשר מטלה בקבוצה מעוררת חריג שלא נתפס. "סביבת הביצוע" משתמשת בשיטה זו כדי להגיב באופן מתאים לחריגים שלא נתפסים. מימוש ברירת המחדל מדפיס מעקב מחסנית לערוץ השגיאה הסטנדרטי. לעיתים אתה תרצה לדרוס מימוש זה, לדוגמה, כדי להפנות את מעקב המחסנית לרישום ספציפי של יישום.

פרק 10

Serialization

פרק זה עוסק ב-API של העברה בטור של אובייקט, אשר מספקת סביבת עבודה לקידוד אובייקטים כזרם של בתים ובניית אובייקטים מחדש מקידודי הבתים שלהם. קידוד אובייקט כזרם של בתים ידוע כ**העברה טורית** (serialization) של האובייקט. התהליך ההפוך ידוע כ**שחזור ההעברה הטורית** (deserialization) של האובייקט. ברגע שאובייקט הועבר בטור, הקידוד שלו יכול להימסר ממכונה וירטואלית רצה אחת למכונה אחרת או להישמר בדיסק לשחזור מאוחר יותר. העברה טורית מספקת את הייצוג הסטנדרטי של האובייקט ברמת הכבל לתקשורת רחוקה ואת תבנית הנתונים המתמידים (persistant) הסטנדרטית עבור ארכיטקטורת רכיב JavaBeans™.

נושא 54: ממש העברה טורית בתבונה

מתן אפשרות למופע של מחלקה להיות מועבר בטור יכולה להיות פשוטה כמו הוספת המילים **implements Serializable** להצהרה שלה. מכיון שכל כך קל לעשות זאת, יש תפיסה מוטעית מקובלת לפיה העברה טורית דורשת מאמץ פעוט מצד המתכנת. האמת היא הרבה יותר מורכבת. בעוד שהעלות המיידית ליצירת מחלקה שיכולה לממש העברה טורית יכולה להיות אפסית, העלויות לטווח הרחוק הן לעיתים קרובות ניכרות.

עלות עיקרית של מימוש העברה טורית היא שהעברה טורית מקטינה את הגמישות לשנות מימוש של מחלקה לאחר שהיא הופצה. כאשר מחלקה מממשת את ה-Serializable, קידוד זרם הבתים שלה (או צורת ההעברה הטורית) נהיה חלק מה-API המיוצא שלה. לאחר שהפצת מחלקה באופן נרחב, בדרך כלל תידרש תמיד לתמוך בצורה המועברת בטור, בדיוק כמו שאתה נדרש לתמוך בכל החלקים האחרים של ה-API המיוצא. אם אינך מוכן להשקיע מאמץ כדי לתכנן צורת **העברה טורית מותאמת אישית**, אלא פשוט מסתפק בצורת ברירת המחדל, הצורה המועברת בטור תהיה קשורה תמיד לייצוג הפנימי המקורי של המחלקה. במילים אחרות, אם אתה מקבל את צורת ברירת המחדל של ההעברה הטורית, שדות המופע הפרטיים והחבילה הפרטית של המחלקה הופכים להיות חלק מה-API המיוצא שלה והנוהג למזער גישה לשדות (נושא 12) מאבד את יעילותו בתור אמצעי להסתרת מידע.

אם אתה מקבל את צורת ברירת המחדל של ההעברה הטורית ומאוחר יותר משנה את ייצוגה הפנימי של המחלקה, אתה עלול לגרום לשינוי לא תואם בצורה המועברת בטור. לקוחות אשר ינסו להעביר בטור מופע תוך שימוש בגירסה ישנה של המחלקה וישחזרו אותו תוך שימוש בגירסה החדשה של המחלקה, ייתקלו בתקלות בתוכנית. אפשרי לשנות את הייצוג הפנימי תוך שמירה על צורת ההעברה הטורית המקורית (על ידי שימוש ב- ObjectOutputStream.putFields ו- ObjectInputStream.readFields), אבל הדבר יכול להיות קשה וגם להשאיר עקבות נראות בקוד המקור. לכן, עליך לתכנן בתשומת לב צורת העברה טורית איכותית אשר אתה יכול להישאר איתה לפרק זמן ארוך (נושא 55). הדבר יגדיל את עלות הפיתוח, אבל הוא שווה את המאמץ. אפילו צורת העברה טורית מתוכננת היטב מציבה אילוצים על ההתפתחות של מחלקה. צורת העברה טורית שאינה מתוכננת היטב עלולה ממש לפגוע.

דוגמה פשוטה לאילוצים על התפתחותה של מחלקה אשר מתלווים ליכולת לבצע העברה טורית נוגעת ל**מזהי זרם ייחודים** (stream unique identifiers), אשר ידועים יותר כ-UDIs של גירסה טורית. לכל מחלקה שהיא serializable יש מספר זיהוי ייחודי שמקושר לה. אם אינך מציין את מספר הזיהוי במפורש על ידי הצהרת שדה long סופי סטטי פרטי בשם serialVersionUID, אז המערכת תיצור אותו באופן אוטומטי על ידי הוספת פרוצדורה דטרמיניסטית מורכבת למחלקה. הערך שנוצר באופן אוטומטי מושפע על ידי שם המחלקה, שמות הממשקים שהיא מממשת וכל האיברים המוגנים והציבוריים שלה. אם תשנה אחד מהדברים האלה בדרך כלשהי, לדוגמה על ידי הוספת שיטה, ה-UID של הגירסה הטורית שנוצר באופן אוטומטי ישתנה. אם תיכשל בהגדרת UID של גירסה סידורית מפורשת, לא תהיה תאימות.

עלות שנייה של מימוש Serializable היא שהדבר מגדיל את האפשרות לבאגים וחורי אבטחה. בדרך כלל, אובייקטים נוצרים על ידי בנאים. העברה טורית היא מנגנון חיצוני לשפת Java אשר יוצר אובייקטים. בין אם אתה מקבל את התנהגות ברירת המחדל או דורס אותה, שחזור העברה טורית הוא "בנאי מוסתר" עם כל אותן התוצאות כמו בנאים אחרים. מכיון שאין שום בנאי מפורש, קל לשכוח שעליך להבטיח שהשחזור של ההעברה הטורית יספק את כל הקבועים שנבנים על ידי בנאים אמיתיים ושהוא לא יאפשר לתוקף להשיג גישה לתכונות הפנימיות של אובייקט שנמצא בתהליך בנייה. הסתמכות על מנגנון ברירת המחדל של שחזור העברה טורית יכולה בקלות להשאיר אובייקטים פרוצים להשחתת קבועים ולגישה לא חוקית (נושא 56).

עלות שלישית של מימוש Serializable היא שהדבר מגדיל את נטל המבחן הכרוך בהפצת גירסה חדשה של מחלקה. כאשר מחלקה בת-יכולת העברה טורית עוברת שיפור, חשוב לבדוק שאפשר בגירסה החדשה להעביר מופע בטור ולשחזר אותו בגרסאות הישנות ולהיפך. סך כל הבדיקות הנדרשות הוא פרופורציונאלי למספר המחלקות שמממשות העברה טורית ולמספר הגרסאות, ולכן הוא יכול להיות גדול מאוד.

אי אפשר לבנות את הבדיקות האלו באופן אוטומטי מכיון שבנוסף ל**תאימות בינארית** אתה חייב לבדוק גם **תאימות סמנטית**. במילים אחרות, עליך להבטיח שתהליכי העברה טורית ושחזור העברה טורית מצליחים, ושהאובייקט שהם יוצרים הוא העתק מדויק ואמין של האובייקט המקורי. ככל שהשינוי במחלקה שמממשת העברה טורית גדול יותר, כך גדל הצורך בבדיקות. הצורך מופחת אם צורה טורית מותאמת אישית מתוכננת בקפדנות כאשר המחלקה נכתבת לראשונה (נושא 55), אבל הוא לא נעלם לגמרי.

מימוש הממשק Serializable אינו החלטה שיש לקבל מתוך פזיזות. הוא מציע יתרונות ממשיים: הוא חיוני אם מחלקה צריכה לקחת חלק באיזו סביבת עבודה אשר מסתמכת על העברה טורית כדי למסור אובייקט או כדי להיות מתמשכת. יתר על כן, הוא מקל מאוד על השימוש במחלקה בתור רכיב במחלקה אחרת אשר חייבת לממש Serializable. בכל אופן, ישנן הרבה עלויות הנוגעות למימוש Serializable. בכל פעם שאתה מממש מחלקה, השווה את העלויות מול הרווחים. כעיקרון מנחה, מחלקות ערך כמו Date ו-BigInteger צריכות לממש Serializable, כמו שצריכות רוב מחלקות האוסף. מחלקות אשר מייצגות ישויות אקטיביות, כמו מצרפי מטלות, צריכות לממש Serializable באופן נדיר. החל מגירסה 1.4, יש מנגנון JavaBeans מתמיד מבוסס XML, כך שאין יותר צורך ל-Beans לממש Serializable.

מחלקות שתוכננו להורשה (נושא 15) צריכות לממש Serializable באופן נדיר וממשקים צריכים להרחיב את זה באופן נדיר. הפרת כלל זה גורמת לנטל משמעותי על כל מי שמגדיל את המחלקה או מממש את הממשק. ישנן נסיבות בהן מתאים להפר כלל זה. לדוגמה, אם מחלקה או ממשק קיימים באופן ראשוני כדי להשתתף באיזו סביבת עבודה אשר דורשת מכל המשתתפים לממש Serializable, אז מאוד הגיוני שהמחלקה או הממשק יממשו או ירחיבו Serializable.

ישנה אזהרה אחת בנוגע להחלטה **לא** לממש Serializable. אם מחלקה אשר מתוכננת להורשה היא לא serializable, ייתכן שיהיה בלתי אפשרי לכתוב תת-מחלקה שהיא serializable. במיוחד זה יהיה בלתי אפשרי אם מחלקת-העל לא מספקת בנאי חסר פרמטרים נגיש. לכן, **אתה צריך לשקול הספקת בנאי חסר פרמטרים על מחלקות שהן nonserializable שתוכננו להורשה**. לעיתים קרובות הדבר אינו דורש כל מאמץ מכיון שלהרבה מחלקות שמתוכננות להורשה אין שום שדות, אבל לא תמיד זה המקרה.

יצירת אובייקטים שכל הקבועים שלהם כבר מבוססים היא הטובה ביותר (נושא 13). אם מידע מסופק לקוח דרוש כדי לבסס קבועים אלה, זה ימנע שימוש בבנאי חסר פרמטרים. הוספה נאיבית של בנאי חסר פרמטרים ושיטת אתחול למחלקה אשר הבנאים הנותרים שלה מבסיסים קבועים אלה, עשויה לסבך את מרווח המצב של מחלקה תוך הגדלת הסבירות לטעויות.

הנה דרך להוסיף בנאי חסר פרמטרים למחלקה בת-הגדלה שאינה serializable, שנמנעת מהליקויים האלה. נניח שלמחלקה יש בנאי אחד:

```
public AbstractFoo(int x, int y) {  ...   }
```

הקוד הבא מוסיף בנאי חסר פרמטרים מוגן ושיטת אתחול. לשיטת האתחול יש את אותם פרמטרים כמו הבנאי הרגיל והיא מבססת את אותם הקבועים:

```
//Nonrerializable stateful class allowing serializable subclass
public abstract class AbstractFoo {
  private int x, y; //The state
  private boolean initialized = false;

  public AbstractFoo(int x, int y) { initialize(x, y); }

  / **
    * This constructor and the following method allow subclass's
    * readObject method to initialize our internal state.
    * /
  protected AbstractFoo() { }

  protected final void initialize(int x, int y) {
    if (initialized)
      throw new IllegalStateException(
          "Already initialized");
    this.x =x;
    this.y =y;
    ...//Do anything else the original constructor did
    initialized = true;
  }

  / **
    * These methods provide access to internal state so it can
    * be manually serialized by subclass's writeObject method.
    * /
  protected final int getX() { return x; }
  protected final int getY() { return y; }

  //Must be called by all public instance methods
  private void checkInit() throws IllegalStateException {
      if (!initialized)
          throw new IllegalStateException("Uninitialized");
  }
  ...//Remainder omitted
}
```

כל שיטות המופע ב-AbstractFoo חייבות לקרוא ל-checkInit לפני שהן פונות לעסקיהן. הדבר מבטיח שקריאות שיטה תיכשלנה במהירות ובאופן נקי, אם תת-מחלקה שנכתבה באופן עלוב נכשלת באתחול מופע. עם המנגנון הזה במקום, הגיוני וברור לממש תת-מחלקה שהיא serializable:

```
//Serializable subclass of nonserializable stateful class
public class Foo extends AbstractFoo implements Serializable {
  private void readObject(ObjectInputStream s)
      throws IOException, ClassNotFoundException {
    s.defaultReadObject();

    //Manually deserialize and initialize superclass state
    int x = s.readInt();
    int y = s.readInt();
    initialize(x,y);
  }

  private void writeObject(ObjectOutputStream s)
        throws IOException {
    s.defaultWriteObject();

    //Manually serialize superclass state
    s.writeInt(getX());
    s.writeInt(getY());
  }

  //Constructor does not use any of the fancy mechanism
  public Foo(int x, int y) { super(x, y); }
}
```

מחלקות פנימיות (נושא 18) צריכות באופן נדיר, אם בכלל, לממש Serializable. הן משתמשות ב**שדות סינתטיים** שהמהדר מחולל כדי לאחסן ייחוסים למופעים שכוללים אותם ולאחסן ערכים של משתנים מקומיים מטווחי הכרה שכוללים אותם. לא מצוין כיצד שדות אלה תואמים להגדרת המחלקה, כמו שגם השמות של מחלקות מקומיות ואנונימיות לא מצוינים. לכן, **צורת ברירת המחדל של העברה טורית של מחלקה פנימית מוגדרת רע**. בכל אופן, **מחלקת איבר סטטית** יכולה לממש Serializable.

לסיכום, הקלות של מימוש Serializable היא מטעה. פרט למקרה שבו מחלקה תושלך לאחר תקופת שימוש קצרה, מימוש Serializable הוא התחייבות רצינית שמצריכה תשומת לב. זהירות נוספת נדרשת אם מחלקה מתוכננת להורשה. עבור מחלקות כאלו, נקודת תכנון ביניים בין מימוש Serializable לבין הימנעות מכך בתת-מחלקות, היא לספק בנאי חסר פרמטרים נגיש. נקודת תכנון זו מרשה, אבל לא דורשת מתת-מחלקות לממש Serializable.

נושא 55: שקול שימוש בהעברה טורית מותאמת אישית

כאשר אתה מספק מחלקה תחת לחץ זמן, בדרך כלל ראוי לרכז את מאמציך בתכנון ה-API הטוב ביותר. לעיתים משמעות הדבר היא הפצת מימוש שהוא "עלון פרסומת" ואתה יודע שתחליף אותו בגירסה עתידית. בדרך כלל אין זו בעיה, אבל אם המחלקה מממשת Serializable ומשתמשת בצורת ברירת המחדל של העברה טורית, לעולם לא תוכל להתעלם לחלוטין מהמימוש הזמני הראשון ("עלון הפרסומת"). הוא יכתיב את צורת ההעברה הטורית לתמיד. אין זו בעיה תיאורטית. הדבר קרה לכמה מחלקות בספריות הפלטפורמה של Java, כמו BigInteger.

אל תקבל את צורת ברירת המחדל של ההעברה הטורית בלי ששקלת קודם האם היא מתאימה. בהחלטתך לקבל את צורת ברירת המחדל של ההעברה הטורית אתה מודע לכך שקידוד זה הוא סביר מנקודת המבט של גמישות, ביצועים ונכונות. בדרך כלל, אתה צריך לקבל את צורת ברירת המחדל של ההעברה הטורית רק אם היא דומה מאוד לקידוד שהיית בוחר אם היית מתכנן צורת העברה טורית מותאמת.

צורת ברירת המחדל של העברה טורית של אובייקט היא קידוד סביר יעיל של הייצוג **הפיסי** של גרף האובייקט שמושרש באובייקט. במילים אחרות, הוא מתאר את הנתונים שמכיל האובייקט וכל אובייקט שנגיש מאובייקט זה. הוא גם מתאר את הטופולוגיה לפיה כל האובייקטים האלה קשורים יחדיו. צורת ההעברה האידיאלית של אובייקט בטור מכילה רק את הנתונים ה**לוגיים** שמיוצגים על ידי האובייקט. היא לא תלויה בייצוג הפיסי.

סביר שצורת ברירת המחדל של ההעברה הטורית תתאים אם ייצוגו הפיסי של אובייקט זהה לתוכנו הלוגי. לדוגמה, צורת ברירת המחדל של ההעברה הטורית תהיה הגיונית עבור המחלקה הבאה, אשר מייצגת שם של אדם:

```
//Good candidate for default serialized form
public class Name implements Serializable {
  / **
   * Last name. Must be non-null.
   * @serial
   * /
  private String lastName;

  / **
   * First name. Must be non-null.
   * @serial
   * /
  private String firstName;
```

```
  / **
   * Middle initial, or '\u0000' if name lacks middle initial.
   * @serial
   * /
  private char      middleInitial;

  ...//Remainder omitted
}
```

באופן הגיוני, שם מורכב משתי מחרוזות אשר מייצגות שם משפחה ושם פרטי ותו אשר מייצג תחילית שם אמצעי. שדות המופע של Name משקפים במדויק את התוכן הלוגי הזה.

אפילו אם החלטת שצורת ברירת המחדל של ההעברה הטורית היא מתאימה, לעיתים קרובות אתה חייב לספק שיטת readObject כדי להבטיח קבועים ואבטחה. במקרה של Name, שיטת readObject יכולה להבטיח ש-lastName ו-firstName לא יהיו null. נושא זה נדון באריכות בנושא 56.

שים לב שיש הערות בשדות lastName, firstName ו-middleInitial, למרות שהם פרטיים. זאת מכיון ששדות פרטיים אלה מגדירים API ציבורי, וצורת ההעברה הטורית של המחלקה וה-API הציבורי הזה חייבים להיות מתועדים. הנוכחות של התגית @serial אומרת לתוכנת העזר Javadoc למקם תיעוד זה בעמוד מיוחד אשר מתעד צורות העברה טורית.

מהצד ההפוך של נקודת המבט של Name, התבונן במחלקה הבאה, אשר מייצגת רשימת מחרוזות (התעלם לרגע מכך שיותר טוב להשתמש באחד ממימושי List הסטנדרטיים שבספריה):

```
//Awful candidate for default serialized form
public class StringList implements Serializable {
  private int size = 0;
  private Entry head = null;

  private static class Entry implements Serializable {
    String data;
    Entry next;
    Entry previous;
  }

  ...//Remainder omitted
}
```

באופן הגיוני, מחלקה זו מייצגת רצף של מחרוזות. באופן פיסי, היא מייצגת את הרצף כרשימה מקושרת כפולה. אם תקבל את צורת ברירת המחדל של ההעברה הטורית, צורת ההעברה הטורית תשקף בדקדקנות כל כניסה ברשימה המקושרת ואת כל הקשרים בין הכניסות בשני הכיוונים.

לשימוש בצורת ברירת המחדל של העברה טורית כאשר ייצוגו הפיסי של אובייקט שונה מאוד מתוכן נתוניו הלוגיים יש ארבעה חסרונות:

- **הדבר קושר לצמיתות את ה-API המיוצא לייצוג הפנימי**. בדוגמה שלעיל, המחלקה הפרטית StringList.Entry הופכת להיות חלק מה-API הציבורי. אם הייצוג ישתנה בגירסה עתידית, מחלקת StringList עדיין תצטרך לקבל את ייצוג הרשימה המקושרת כקלט וליצור אותו כפלט. המחלקה לעולם לא תשתחרר מהקוד ליצירת רשימות מקושרות, אפילו אם היא לא משתמשת בהן יותר.
- **זה יכול לצרוך מקום מוגזם**. בדוגמה לעיל, צורת ההעברה הטורית מייצגת שלא לצורך כל כניסה ברשימה המקושרת ואת כל הקשרים. הכניסות והקשרים האלה הם בסך הכל פרטי מימוש אשר לא ראויים להכללה בצורת ההעברה הטורית. מכיון שה-serialized form גדול מדי, כתיבתו בדיסק או שליחתו ברשת תקשורת תהיה איטית מדי.
- **זה יכול לגרום לגלישות מחסנית**. הליך ברירת המחדל של העברה בטור מבצע מעבר רקורסיבי על גרף האובייקט, דבר שיכול לגרום לגלישות מחסנית אפילו עבור גרפי אובייקט בגודל מתון. העברה בטור של מופע StringList עם 1200 אלמנטים גורמת למחסנית במחשב שלי לגלוש. מספר האלמנטים שדרוש כדי לגרום לבעיה זו יכול להשתנות כתלות במימוש ה- JVM. ייתכן שמימושים מסוימים לא יסבלו מבעיה זו בכלל.

serialized form הגיוני ל-StringList הוא פשוט מספר המחרוזות ברשימה כשאחריו המחרוזות עצמן. זה מייסד את הנתונים הלוגיים שמיוצגים על ידי StringList ללא פרטי ייצוגו הפיסי. הנה גירסה מתוקנת של StringList שמכילה שיטות writeObject ו-readObject שמממשות את ה- serialized form הזה. כתזכורת, המגדיר transient מצביע על כך ששדה מופע יושמט מה- default serialized form של מחלקה:

הערה!

בתוכנית הבאה יש שימוש במשתנה size בתוך השיטה readObject. אני מציע שתתן למשתנה זה שם אחר כדי שלא תתבלבל, וכדי שגם אחרים לא יתבלבלו.

```
//StringList with a reasonable custom serialized form
public class StringList implements Serializable {
  private transient int size = 0;
  private transient Entry head = null;
```

```
  //No longer Serializable!
  private static class Entry {
    String data;
    Entry next;
    Entry previous;
  }

  //Appends the specified string to the list
  public void add(String s) {  ...  }

  / **
    * Serialize this <tt>StringList</tt>instance.
    *
    * @serialData The size of the list (the number of strings
    * it contains) is emitted (<tt>int</tt>), followed by all of
    * its elements (each a <tt>String</tt>), in the proper
    * sequence.
    * /
  private void writeObject(ObjectOutputStream s)
            throws IOException {
      s.defaultWriteObject();
      s.writeInt(size);

      //Write out all elements in the proper order.
      for (Entry e = head; e != null; e = e.next)
            s.writeObject(e.data);
  }

  private void readObject(ObjectInputStream s)
            throws IOException, ClassNotFoundException {
      s.defaultReadObject();
      int size = s.readInt();

      //Read in all elements and insert them in list
      for (int i = 0; i < size; i++)
            add((String)s.readObject());
  }

  ...//Remainder omitted
}
```

שים לב ששיטת writeObject קוראת ל-defaultWriteObject ושיטת readObject קוראת ל-defaultReadObject, אפילו אם כל השדות של StringList הם ארעיים (transient). **אם כל השדות של מופע הם ארעיים, אז אפשר טכנית לוותר על קריאה ל-defaultWriteObject ול-defaultReadObject, אבל הדבר אינו מומלץ**. אפילו אם כל השדות של מופע הם ארעיים, קריאה ל-defaultWriteObject משפיעה על ה- serialized form, דבר שמתבטא בגמישות גדולה בהרבה.

ה- serialized form המתקבל מאפשר להוסיף שדות מופע לא ארעיים בגירסה מאוחרת יותר, תוך שמירה על תאימות עם גרסאות קודמות ועתידיות.
אם מופע מועבר בטור בגירסה מאוחרת יותר ומשוחזר מההעברה בטור בגירסה מוקדמת יותר, תהיה התעלמות מהשדות הנוספים. אם השיטה readObject של הגירסה המוקדמת יותר נכשלה לקרוא ל-defaultReadObject, ה-deserialization ייכשל עם StreamCorruptedException.

שים לב שישנה הערת תיעוד בשיטה writeObject, למרות שהיא פרטית. זה אנלוגי להערת התיעוד בשדות הפרטיים של המחלקה Name. שיטה פרטית זו מגדירה API ציבורי, את ה- serialized form, ואומרת ש-API ציבורי צריך להיות מתועד. כמו התגית @serial עבור שדות, התגית @serialDate עבור שיטות אומרת לתוכנת העזר Javadoc למקם תיעוד זה בעמוד של ה- serialized forms.

כדי לתת קנה מידה מסוים לביצועים בדיון הקודם, אם אורך המחרוזת הממוצעת הוא עשרה תווים, ה- serialized form של הגירסה המשופרת של StringList תופסת בערך מחצית מהמקום של ה- serialized form של הגירסה המקורית. במחשב שלי, העברה טורית של הגירסה המשופרת של StringList היא בערך פי שניים וחצי יותר מהירה מאשר העברה טורית של הגירסה המקורית. שוב, עם מחרוזת באורך עשרה תווים. לבסוף, אין שום בעיית גלישת מחסנית בצורה המשופרת, לכן אין שום גבול עליון מעשי לגודל של StringList אשר יכול לעבור בטור.

בעוד ש- default serialized form תהיה גרועה עבור StringList, יש מחלקות עבורן היא תהיה גרועה הרבה יותר. עבור StringList, ה- default serialized form לא גמישה וביצועיה גרועים, אבל היא **נכונה** מהבחינה שהעברה טורית ושחזור מההעברה טורית של מופע StringList מניב העתק אמין של האובייקט המקורי בלי לפגוע בקבועים שלו. זה לא מה שקורה לאובייקט אשר הקבועים שלו קשורים לפרטי מימוש ספציפיים.

לדוגמה, חשוב על המקרה של טבלת ערבול. הייצוג הפיסי הוא רצף של מנות ערבול אשר כוללות כניסות ערך-מפתח. איזו מנה תמוקם בכל כניסה היא פונקציה של קוד הערבול של המפתח, אשר בדרך כלל לא בטוח שהוא יהיה דומה בכל מימוש JVM. למעשה, אפילו לא בטוח שהוא יהיה דומה בכל ריצה של אותו מימוש JVM. לכן, קבלת ה- default serialized form של טבלת ערבול יכולה לגרום לבאג רציני. העברה טורית ושחזור מהעברה טורית של טבלת הערבול יכולה להניב אובייקט שהקבועים שלו הושחתו באופן רציני.

בין אם אתה משתמש או לא משתמש ב- default serialized form, כל שדה מופע אשר לא יוגדר כ-transient יועבר בטור כאשר שיטת defaultWriteObject נקראת. לכן, כל שדה מופע אשר יכול להיות ארעי צריך להפוך לכזה. הדבר כולל שדות מיותרים, אשר ערכיהם יכולים להיות מחושבים מתוך "שדות נתונים ראשוניים", כמו ערכי ערבול ששמורים במאגר.

הדבר כולל גם שדות שערכיהם קשורים לריצה מיוחדת של JVM, כמו שדה long שמייצג מצביע למבנה נתונים מקומי. **לפני שאתה מחליט לעשות שדה לא ארעי, שכנע את עצמך שהערך שלו הוא חלק מהמצב הלוגי של האובייקט**. אם אתה משתמש ב- custom serialized form, רוב או כל שדות המופע צריכים להיות מתויגים כ-transient, כמו בדוגמה StringList שהוצגה לעיל.

אם אתה משתמש ב- default serialized form ותייגת שדה אחד או יותר כ-transient, זכור ששדות אלה יאותחלו ל**ערכי ברירת המחדל שלהם** כאשר המופע משוחזר מהזרם: null עבור שדות ייחוס אובייקט, אפס עבור שדות פשוטים נומריים ו-false עבור שדות boolean [JLS, 4.5.5].
אם ערכים אלה אינם קבילים עבור שדות ארעיים כלשהם, אתה חייב לספק את השיטה readObject אשר קוראת לשיטה defaultReadObject ואז משחזרת שדות ארעיים לערכים קבילים (נושא 56). באופן חילופי, שדות אלה יכולים לעבור אתחול עצל בפעם הראשונה בה הם משמשים.

ללא התחשבות באיזו serialized form אתה בוחר, הצהר serial version UID מפורש בכל מחלקה serializable שאתה כותב. הדבר מונע מה- serial version UID להיות מקור פוטנציאלי של אי תאימות (נושא 54). כמו כן, יש רווח קטן של ביצועים. אם לא מסופק אף serial version UID, נדרש חישוב יקר כדי ליצור אחד כזה בזמן ריצה.

הגדרת serial version UID היא פשוטה. פשוט הוסף שורה זו למחלקה שלך:

```
private static final long serialVersionUID = randomLongValue ;
```

אין זה משנה איזה ערך אתה בוחר עבור **randomLongValue**. מנהג מקובל מכתיב שאתה תיצור את הערך על ידי הרצת ה- serial utility על המחלקה, אבל זה בסדר גם לבחור מספר משום מקום. אם אי פעם תרצה ליצור גירסה חדשה של המחלקה שהיא incompatible עם גרסאות קיימות, פשוט שנה את הערך בהצהרה.
זה יגרום לניסיונות של שחזור מופעים של גרסאות קודמות שעברו בטור להיכשל עם InvalidClassException.

לסיכום, כאשר אתה מחליט שמחלקה צריכה להיות serializable (נושא 54), חשוב היטב על מה צריכה להיות ה- serialized form. השתמש ב- default serialized form רק אם היא תיאור סביר של מצבו הלוגי של האובייקט.
אחרת, תכנן cutom serialized form אשר מתארת את האובייקט כהלכה. אתה צריך להקצות לתכנון ה- serialized form של מחלקה את אותו הזמן שאתה מקצה לתכנון השיטות המיוצאות שלה. כמו שאתה לא יכול להוציא שיטות מיוצאות מגרסאות עתידיות, אתה לא יכול להוציא שדות מה- serialized form. הם חייבים להישמר לתמיד כדי להבטיח תאימות serialization. לבחירת ה- serialized form הלא נכונה יכולה להיות השפעה שלילית קבועה על המורכבות והביצועים של מחלקה.

נושא 56: כתוב שיטות readObject הגנתיות

נושא 24 כולל מחלקת Period לא ניתנת לשינוי, אשר מכילה שדות תאריך פרטיים ניתנים לשינוי. המחלקה עושה רבות כדי לשמר את הקבועים שלה ואת תכונת אי ההשתנות שלה על ידי העתקה הגנתית של אובייקטי Date בבנאי שלה ובאמצעי הגישה שלה. הנה המחלקה:

```
//Immutable class that uses defensive copying
public final class Period {
  private final Date start;
  private final Date end;

  / **
    * @param start the beginning of the period.
    * @param end the end of the period; must not precede start.
    * @throws IllegalArgument if start is after end.
    * @throws NullPointerException if start or end is null.
    * /
  public Period(Date start, Date end) {
     this.start = new Date(start.getTime());
     this.end = new Date(end.getTime());

     if (this.start.compareTo(this.end)>0)
       throw new IllegalArgumentException(start +" > "+ end);
  }

  public Date start () {   return (Date) start.clone();  }

  public Date end ()  {   return (Date) end.clone();   }

  public String toString() {   return start + " - " + end;   }

  ...//Remainder omitted
}
```

נניח שאתה רוצה שמחלקה זו תהיה serializable. מכיון שהייצוג הפיסי של אובייקט Period משקף במדויק את תוכן נתוניו הלוגי, אין זה בלתי סביר להשתמש ב- default serialized form (נושא 55).
לכן, עשוי להיראות שכל מה שאתה צריך לעשות כדי לעשות את המחלקה serializable הוא להוסיף את המילים "implements serializable" להצהרת המחלקה. בכל אופן, אם תעשה כך המחלקה לא תערוב יותר לקבועים הקריטיים שלה.

הבעיה היא שהשיטה readObject היא למעשה עוד בנאי ציבורי והיא דורשת את אותה תשומת הלב כמו כל בנאי אחר. כמו שבנאי צריך לבדוק את תוקפם של הפרמטרים שלו (נושא 23) וליצור עותקים הגנתיים של פרמטרים כאשר הדבר מתאים (נושא 24),

כך חייבת לעשות גם השיטה readObject. אם השיטה readObject לא מצליחה לעשות את אחד מהדברים האלה, אז למתקיף יהיה יחסית קל לפגוע בקבועים של המחלקה.

בשפה פשוטה, readObject היא בנאי אשר מקבל זרם של בתים בתור הפרמטר היחיד שלו. בשימוש תקין, זרם הבתים נוצר על ידי העברה טורית של מופע שנבנה באופן תקין. הבעיה מתעוררת כאשר readObject מיוצגת על ידי זרם בתים אשר נבנה באופן מלאכותי לייצר אובייקט, אשר פוגע בקבועים של המחלקה שלו. נניח שפשוט הוספנו "implements Serializable" להצהרת המחלקה של Period. תוכנית מגעילה זו יוצרת מופע Period אשר הסוף שלו בא לפני ההתחלה שלו:

```
public class BogusPeriod {
    //Byte stream could not have come from real Period instance
    private static final byte[ ] serializedForm = new byte[ ] {
    (byte)0xac, (byte)0xed, 0x00, 0x05, 0x73, 0x72, 0x00, 0x06,
    0x50, 0x65, 0x72, 0x69, 0x6f, 0x64, 0x40, 0x7e, (byte)0xf8,
    0x2b, 0x4f, 0x46, (byte)0xc0, (byte)0xf4, 0x02, 0x00, 0x02,
    0x4c, 0x00, 0x03, 0x65, 0x6e, 0x64, 0x74, 0x00, 0x10, 0x4c,
    0x6a, 0x61, 0x76, 0x61, 0x2f, 0x75, 0x74, 0x69, 0x6c, 0x2f,
    0x44, 0x61, 0x74, 0x65, 0x3b, 0x4c, 0x00, 0x05, 0x73, 0x74,
    0x61, 0x72, 0x74, 0x71, 0x00, 0x7e, 0x00, 0x01, 0x78, 0x70,
    0x73, 0x72, 0x00, 0x0e, 0x6a, 0x61, 0x76, 0x61, 0x2e, 0x75,
    0x74, 0x69, 0x6c, 0x2e, 0x44, 0x61, 0x74, 0x65, 0x68, 0x6a,
    (byte)0x81, 0x01, 0x4b, 0x59, 0x74, 0x19, 0x03, 0x00, 0x00,
    0x78, 0x70, 0x77, 0x08, 0x00, 0x00, 0x00, 0x66, (byte)0xdf,
    0x6e, 0x1e, 0x00, 0x78, 0x73, 0x71, 0x00, 0x7e, 0x00, 0x03,
    0x77, 0x08, 0x00, 0x00, 0x00, (byte)0xd5, 0x17, 0x69, 0x22,
    0x00, 0x78 };

    public static void main(String[] args)) {
      Period p = (Period) deserialize(serializedForm);
      System.out.println(p);
    }

    //Returns the object with the specified serialized form
    public static Object deserialize(byte[] sf)) {
      try {
          InputStream is = new ByteArrayInputStream(sf);
          ObjectInputStream ois = new ObjectInputStream(is);
          return ois.readObject();
      } catch (Exception e) {
          throw new IllegalArgumentException(e.toString());
      }
    }
}
```

המערך byte שימש לאתחול serializedForm, אשר נוצר על ידי העברה טורית של מופע Period רגיל ועריכה ידנית של זרם הבתים המתקבל. פרטיו של הזרם אינם חשובים לדוגמה, אבל אם אתה סקרן, תבנית זרם הבתים המועבר בטור מתוארת ב- Java™ Object Serialization Specification [Serialization, 6]. אם תריץ תוכנית זו, היא תדפיס "Fri Jan 01 12:00:00 PST 1999 - Sun Jan 01 12:00:00 PST 1984".
יצירת Period כ-serializable איפשרה לנו ליצור אובייקט אשר פוגע בקבועים של המחלקה שלו. כדי לתקן בעיה זו, ספק שיטת readObject עבור Period אשר קוראת ל-defaultReadObject, ואז בודקת את תוקפו של האובייקט deserialized. אם בדיקת התוקף נכשלת, השיטה readObject מעוררת InvalidObjectException ומונעת את השלמת ה-deserialization:

```
private void readObject(ObjectInputStream s)
      throws IOException, ClassNotFoundException {
  s.defaultReadObject();

  //Check that our invariants are satisfied
  if (start.compareTo(end)>0)
      throw new InvalidObjectException( start +"after "+ end );
}
```

למרות שתיקון זה מונע מהתוקף ליצור מופע Period לא תקף, יש בעיה יותר עדינה שעדיין מסתתרת. אפשר ליצור מופע Period ניתן לשינוי על ידי זיוף זרם בתים אשר מתחיל עם זרם בתים אשר מייצג מופע Period תקף, ואז מצרף ייחוסים נוספים לשדות Date הפרטיים שפנימיים למופע Period. התוקף קורא את המופע של Period מה-ObjectInputStream ואז קורא את "ייחוסי האובייקט הנוכל" אשר צורפו לזרם. ייחוסים אלה נותנים לתוקף גישה לאובייקטים אשר מיוחסים על ידי שדות Date הפרטיים מתוך האובייקט מטיפוס Period. על ידי שינוי מופעי Date אלה, התוקף יכול לשנות את המופע של Period. המחלקה הבאה מדגימה את ההתקפה הזאת:

```
public class MutablePeriod {
  //A period instance
  public final Period period;

  //period's start field, to which we shouldn't have access
  public final Date start;

  //period's end field, to which we shouldn't have access
  public final Date end;
```

```
  public MutablePeriod(){
    try {
      ByteArrayOutputStream bos =  new ByteArrayOutputStream();
      ObjectOutputStream out = new ObjectOutputStream(bos);

      //Serialize a valid Period instance
      out.writeObject(new Period(new Date(), new Date()));

      /*
       *Append rogue "previous object refs" for internal
       *Date fields in Period. For details, see "Java
       *Object Serialization Specification," Section 6.4.
       */
      byte [ ] ref = { 0x71, 0, 0x7e, 0, 5 }; //Ref #5
      bos.write(ref);   //The start field
      ref [4 ] = 4; //Ref # 4
      bos.write(ref);  //The end field

      //Deserialize Period and "stolen" Date references
      ObjectInputStream in = new ObjectInputStream(
      new ByteArrayInputStream(bos.toByteArray()));
      period =(Period)  in.readObject();
      start = (Date)       in.readObject();
      end = (Date)        in.readObject();
    } catch (Exception e)  {
        throw new RuntimeException(e.toString());
    }
  }
}
```

כדי לראות את ההתקפה בפעולה, הרץ את התוכנית הבאה:

```
public static void main(String [] args) {
  MutablePeriod mp = new MutablePeriod();
  Period p = mp.period;
  Date pEnd = mp.end;

  //Let's turn back the clock
  pEnd.setYear(78);
  System.out.println(p);

  //Bring back the 60's!
  pEnd.setYear(69);
  System.out.println(p);
}
```

הרצת תוכנית זו מייצרת את הפלט הבא:

```
Wed Ma 07 23:30:01 PST 2001 -Tue Mar 07 23:30:01 PST 1978
Wed Ma 07 23:30:01 PST 2001 -Fri Mar 07 23:30:01 PST 1969
```

למרות שהקבועים של המופע Period שנוצרו הם שלמים וללא פגע, אפשר לשנות את תכונותיו הפנימיות של המופע לפי שיקול דעת. ברגע שמופע של Period שניתן לשינוי נמצא בחזקתו של תוקף, התוקף יכול לגרום נזק גדול על ידי העברת המופע למחלקה אשר הבטיחות שלה תלויה בתכונת אי ההשתנות של Period. זה לא כל כך מאולץ: יש מחלקות אשר תלויות באי ההשתנות של String לבטיחות שלהן.

המקור של הבעיה הוא שהשיטה readObject של Period לא מבצעת מספיק העתקה הגנתית. **כאשר האובייקט הוא deserialized, זה קריטי להעתיק באופן הגנתי כל שדה אשר מכיל ייחוס אובייקט אשר אסור שלקוח יחזיק בו**. לכן, כל מחלקה serializable שאינה ניתנת לשינוי ואשר מכילה רכיבים משתנים פרטיים חייבת להעתיק רכיבים אלה באופן הגנתי בשיטת readObjcet שלה. השיטה readObject הבאה מספיקה להבטיח את הקבועים של Period ולשמור על אי ההשתנות שלה:

```
private void readObject(ObjectInputStream s) throws IOException,
  ClassNotFoundException {
  s.defaultReadObject();

  //Defensively copy our mutable components
  start = new Date(start.getTime());
  end = new Date(end.getTime());

  //Check that our invariants are satisfied
  if (start.compareTo(end) > 0)
      throw new InvalidObjectException(start +" after " + end);
}
```

שים לב שההעתקה ההגנתית מתבצעת לפני בדיקת התוקף ושלא השתמשנו בשיטה clone של Date כדי לבצע את ההעתקה ההגנתית. שני פרטים אלה דרושים כדי להגן על Period מפני התקפה (נושא 24). שים לב גם שההעתקה הגנתית אינה אפשרית עבור שדות סופיים. כדי להשתמש בשיטה readObject, אנו חייבים לעשות את שדות start ו-end לא סופיים. זהו חוסר מזל אבל ברור שזו הרעה הקטנה יותר מבין שתי האפשרויות. עם השיטה readObject החדשה במקום והסרת המגדיר final מהשדות start ו-end, מחלקת MutablePeriod הופכת להיות בלתי אפקטיבית. תוכנית ההתקפה שלעיל מייצרת כעת את הפלט הבא:

```
Thu Ma 08 00:03:45 PST 2001 -Thu Mar 08 00:03:45 PST 2001
Thu Ma 08 00:03:45 PST 2001 -Thu Mar 08 00:03:45 PST 2001
```

יש מבחן שניתן לבצע כדי להחליט האם ברירת המחדל של השיטה readObject מתקבלת על הדעת. האם תרגיש נוח להוסיף בנאי ציבורי אשר מקבל כפרמטרים את הערכים עבור כל שדה לא ארעי באובייקט שלך ומאחסן את הערכים בשדות ללא בדיקת תוקף כלשהי? אם אינך יכול לענות בחיוב על שאלה זו, אז אתה חייב לספק שיטת readObject מפורשת והיא תהיה חייבת לבצע את כל בדיקות התוקף וההעתקה ההגנתית אשר עשויה להידרש מבנאי.

יש דמיון אחד נוסף בין שיטות readObject לבין בנאים, הנוגע למחלקות serializable לא סופיות. לשיטה readObject אסור לקרוא לשיטה בת-דריסה, באופן ישיר או באופן לא ישיר (נושא 15). אם חוק זה מופר והשיטה נדרסת, השיטה הדורסת תרוץ לפני שהמצב של תת-המחלקה עבר deserialized. סביר שהדבר יגרום לכישלון התוכנית.

לסיכום, בכל פעם שאתה כותב את השיטה readObject, אמץ לך את העיקרון המנחה לפיו אתה כותב בנאי ציבורי אשר חייב לייצר מופע בר-תוקף ללא קשר לאיזה זרם בתים הוא מקבל. אל תניח שזרם הבתים מייצג מופע serialized אקטואלי. למרות שהדוגמאות בנושא זה נוגעות למחלקה אשר משתמשת ב- default serialized form, כל הנושאים אשר הוצגו נוגעים באופן דומה למחלקות עם custom serialized forms.
להלן, באופן מקוצר, הקווים המנחים לכתיבת שיטת readObject חסינה:

- למחלקות עם שדות ייחוס אובייקט אשר חייבים להישאר פרטיים, העתק באופן הגנתי כל אובייקט אשר צריך להיות מאוחסן בשדה כזה. רכיבים משתנים של מחלקות בלתי משתנות שייכים לקטגוריה זו.
- למחלקות עם קבועים, בדוק קבועים וזרוק InvalidObjectException אם הבדיקה נכשלת. הבדיקות צריכות לבוא לאחר כל העתקה הגנתית כלשהי.
- אם גרף אובייקט מלא חייב לקבל תוקף לאחר שהוא deserialized, צריך להשתמש בממשק ObjectInputValidation.
  השימוש בממשק זה הוא מעבר לתחום העיסוק של ספר זה. שימוש לדוגמה אפשר למצוא ב- The Java Class Libraries, Second Edition, Volume 1 [Chan98, p.1256].
- אל תקרא לשיטות בנות-דריסה כלשהן במחלקה, באופן ישיר או באופן לא ישיר.

אפשר להשתמש בשיטה readResolve כחלופה לשיטת readObject הגנתית. חלופה זו תידון בנושא 57.

נושא 57: ספק את השיטה readResolve בעת הצורך

נושא 2 מתאר את התבנית Singleton ונותן את הדוגמה הבאה של מחלקת singleton. מחלקה זו מגבילה גישה לבנאי שלה כדי להבטיח שרק מופע יחיד ייווצר אי פעם:

```
public class Elvis {
    public static final Elvis INSTANCE = new Elvis();

    private Elvis(){
        ...
    }

    ...//Remainde omitted
}
```

כפי שצוין בנושא 2, מחלקה זו לא תהיה יותר singleton אם המילים "implements Serializable" הוספו להצהרה שלה. אין זה משנה האם המחלקה משתמשת ב- default serialized form או ב- custom serialized form (נושא 55) וזה גם לא משנה האם המחלקה מספקת שיטת readObject מפורשת (נושא 56). כל שיטת readObject, בין אם היא מפורשת או ברירת מחדל, מחזירה מופע שנוצר מחדש, אשר לא יהיה אותו המופע אשר נוצר בזמן אתחול המחלקה. לפני גירסה 1.2, אי אפשר היה לכתוב מחלקה serializable singleton.

בגירסה 1.2, התכונה readResolve התווספה לאמצעי ה-serialization [Serialization, 3.6]. אם המחלקה של אובייקט שעובר deserialized מגדירה שיטת readResolve עם ההצהרה המתאימה, שיטה זו נקראת על האובייקט שנוצר מחדש לאחר שהוא deserialized. אז, ייחוס האובייקט שמוחזר על ידי שיטה זו מוחזר במקום האובייקט שנוצר מחדש. ברוב השימושים של תכונה זו, אף ייחוס לאובייקט שנוצר מחדש לא יישמר. דינו של אובייקט זה שאוסף האשפה (garbage collection) יטפל בו.

אם נרצה שמחלקת Elvis תממש Serializable, שיטת readResolve הבאה מספיקה כדי להבטיח את תכונת ה-singleton:

```
private Object readResolve() throws ObjectStreamException {
  //Return the one true Elvis and let the garbage collector
  //take care of the Elvis impersonator.
  return INSTANCE;
}
```

שיטה זו מתעלמת מהאובייקט deserialized, ופשוט מחזירה את מופע Elvis המפורסם שנוצר כאשר המחלקה אותחלה. לכן, ה- serialized form של מופע Elvis לא צריך להכיל נתונים ממשיים כלשהם. כל שדות המופע צריכים להיות מצוינים כארעיים. הדבר אמור לא רק לגבי Elvis אלא לכל singletons.

השיטה readResolve היא חיונית לא רק ל-singletons אלא לכל המחלקות *נשלטות-המופע* האחרות. במילים אחרות, לכל המחלקות אשר שולטות ביצירת אובייקט בקפדנות כדי לשמור קבוע מסוים. דוגמה אחרת למחלקה נשלטת מופע היא a typesafe enum (נושא 21), אשר שיטת readResolve שלה חייבת להחזיר את המופע הקאנוני אשר מייצג את קבוע הספירה שמצוין. כעיקרון מנחה, אם אתה כותב מחלקה שהיא serializable אשר לא מכילה בנאים ציבוריים או מוגנים, שקול האם היא דורשת שיטת readResolve.

שימוש שני לשיטה readResolve הוא בתור חלופה שמרנית לשיטה readObject ההגנתית שהומלצה בנושא 56. בגישה זו, כל בדיקות התוקף והעתקה הגנתית מסולקות משיטת readObject ובמקומן באות בדיקות התוקף והעתקה הגנתית שמסופקות על ידי בנאי רגיל. אם משתמשים ב- default serialized form, אפשר לסלק את השיטה readObject לגמרי. כפי שהוסבר בנושא 56, זה מאפשר ללקוח בעל כוונות זדון ליצור מופע עם compromized invariants.

בכל אופן, המופע compromized deserialized בפוטנציה לעולם לא נכנס לשירות פעיל, הוא פשוט משויך לקלט עבור בנאי ציבורי או יצרן סטטי ומסולק.

היופי שבגישה זו הוא שהיא מסלקת באופן וירטואלי את הרכיב החוץ לשוני serialization, דבר שהופך את הפגיעה בקבועים של המחלקה אשר היו קיימים לפני שהמחלקה נעשתה serializable לבלתי אפשרית. כדי להמחיש טכניקה זאת, השיטה readResolve הבאה יכולה לשמש במקום שיטת readObject ההגנתית בדוגמה Period שבנושא 56:

```
//The defensive readResolve idiom
private Object readResolve()throws ObjectStreamException {
    return new Period(start, end);
}
```

שיטת readResolve זו מונעת לגמרי את שתי ההתקפות שתוארו בנושא 56.
ל-readResolve ההגנתית יש כמה יתרונות על פני readObject ההגנתית. זוהי טכניקה **מכנית** להפוך מחלקה ל-serializable ללא סיכון של הקבועים שלה. היא דורשת קוד קטן ועבודה מעטה ומובטח שהיא תעבוד. לבסוף, היא מסלקת את ההגבלות המלאכותיות אשר serialization מציבה על השימוש בשדות סופיים.

למרות שקוד readResolve ההגנתי לא משמש באופן נרחב, הוא ראוי להתייחסות רצינית. חסרונו העיקרי הוא שהוא אינו מתאים למחלקות אשר מתירות הורשה מחוץ לחבילת התוכנה שלהן. אין זה נושא למחלקות בלתי משתנות מכיון שהן בדרך כלל סופיות (נושא 13). חיסרון שולי של הקוד הוא שהקוד מפחית קצת את הביצועים של serialization, מכיון שהוא גורם ליצירת אובייקט נוסף. במחשב שלי, הקוד האט את ה-serialization של מופעי Period בערך באחוז אחד בהשוואה לשיטת readObject ההגנתית.

הנגישות של שיטת readResolve ההגנתית היא משמעותית. אם אתה ממקם את שיטת readResolve במחלקה סופית, כמו singleton, היא תהיה פרטית. אם אתה ממקם שיטת readResolve במחלקה לא סופית, אתה חייב לשקול בזהירות את הנגישות שלה. אם היא פרטית, היא לא תתייחס לתת-מחלקה כלשהי. אם היא פרטית-בחבילה, היא תתייחס רק לתת-מחלקות שבאותה חבילת התוכנה. אם היא מוגנת או ציבורית, היא תתייחס לכל תת-המחלקות אשר לא דורסות אותה. אם השיטה readResolve היא מוגנת או ציבורית ותת-מחלקה לא דורסת אותה, ביצוע deserializing למופע תת-מחלקה שהוא serialized יצור מופע של מחלקת-העל שזה ככל הנראה לא מה שאתה רוצה.

הפיסקה הקודמת רומזת לסיבה שבגללה אין להחליף את השיטה readResolve בשיטה readObject ההגנתית במחלקות אשר מתירות הורשה. אם שיטת readResolve של מחלקת-העל היתה סופית, הדבר ימנע ממופעי תת-מחלקה לעבור deserialized כהלכה. אם היא בת-דריסה, תת-מחלקה זדונית יכולה לדרוס אותה עם שיטה אשר מחזירה compromized instance.

לסיכום, אתה חייב להשתמש בשיטה readResolve כדי להגן על "קבועי שליטת מופע" של singletons ושל מחלקות נשלטות מופע אחרות. בעיקרה, השיטה readResolve הופכת את השיטה readObject מבנאי ציבורי דה פקטו לשיטת ייצור סטטית דה פקטו. כמו כן, השיטה readResolve שימושית כחלופה פשוטה לשיטת readObject הגנתית למחלקות אשר אוסרות על הורשה מחוץ לחבילת התוכנה שלהן.

התקליטור המצורף

תקליטור זה מופעל באופן אוטומטי עם הכנסתו לכונן.

אם התקליטור אינו מופעל באופן אוטומטי:

1. ודא שהתקליטור בכונן.
2. בשולחן העבודה לחץ לחיצה כפולה על סמל **המחשב שלי**.
3. לחץ לחיצה כפולה על סמל **כונן התקליטורים**.
4. לחץ לחיצה כפולה על הקובץ **hodami.exe**.

התקליטור כולל:

- קטלוג ספרי מחשבים בהוצאת **הוד-עמי**
- קוד מקור של הדוגמאות שבספר (במידה וקיים) או קבצים שעליהם מדברים בספר או תוכניות שאליהן מתייחסים בספר.
- תוכנות/תוכניות שירות מועילות
- שאלות נפוצות - לגבי רכישת ספרים בהוצאה
- אודות החברה
- דרכי התקשרות להוצאת **הוד-עמי**

הערה!

אם מנהל התקן כונן התקליטורים המותקן הוא 16 סיביות - ייתכן ותראה רק 8 תווים ראשונים של שם הקובץ (במקרה ובמקור הוא ארוך יותר).

הסיבה: כונני תקליטורים במהירות x4 עובדים עם מנהל התקן שעבד בסביבת DOS ו-Windows 3.11 ויכול לעבוד גם עם Windows 9x, למעט היכולת לזהות קבצים עם שמות ארוכים.

הפתרון: להתקין מנהל התקן 32 סיביות (אם קיים), או לקנות כונן תקליטורים חדש ולוודא שמצורף אליו מנהל התקן 32 סיביות.

קרא קובץ ONCD שנמצא בתקליטור לגבי תכולת התקליטור ולהוראות התקנה של חלק מהתוכנות המצורפות.

הפעלת התקליטור

במשך כל זמן ההפעלה חייב התקליטור להיות בכונן.

הוראות התקנה והפעלה מפורטות נוספות נמצאות בקובץ **ONCD.PDF** בתקליטור.

כדי לקרוא את תוכן הקובץ ONCD.PDF (הכולל פירוט מעוצב של תוכן התקליטור), עליך להתקין את תוכנת Adobe Acrobat Reader מהתפריט **התקנת תוכנות**.

בנוסף, תשמש אותך תוכנה זו לקריאה והדפסה של הפרקים לדוגמה, אותם תמצא בקטלוג ספרי המחשבים בהוצאת **הוד-עמי**.

התוכנה מאפשרת עיון והדפסת המסמך המוצג, כמו גם חיפוש בעברית ובאנגלית.

בעזרת תוכנה זו תוכל גם לקרוא את המסמכים אותם מפרסמת ההוצאה באתר האינטרנט.

קטלוג ספרים

בחלק זה תוכל לעיין בקטלוג הספרים בהוצאת **הוד-עמי**.

עיון בקטלוג ספרי המחשבים של הוצאת **הוד-עמי** כולל:

- תוכן עניינים מלא של הספר
- פרק לדוגמה
- אינדקס לחלק מהספרים

את כל המידע ניתן לראות וגם להדפיס. לצורך זה יש להתקין את תוכנת Acrobat Reader מהתפריט **התקנת תוכנות**.

כדי לצפות בקטלוג העדכני ביותר ולצפות במבצעים המיוחדים, מומלץ לבקר באתר האינטרנט של ההוצאה בכתובת www.hod-ami.co.il.

הקטלוג מומלץ לצפייה ברזולוציית מסך של 600x800.

הקטלוג מומלץ לצפייה באמצעות Internet Explorer 5 ומעלה. התוכנה מצורפת בתקליטור. כדי להתקין אותו בחר בתפריט **התקנת תוכנות** ובחר באפשרות **תוכנות נוספות**. ראה הוראות נוספות בהמשך.

התקנת תוכנות

התקליטור כולל מיגוון תוכנות שירות, חלקן תוכנות חופשיות (freeware) וחלקן תוכנות שיתופיות (shareware). התוכנות מסופקות כמו שהן (as is). הוצאת הוד-עמי אינה אחראית כלפי יחיד או ארגון עבור כל נזק, אשר ייגרם, אם ייגרם, משימוש בתקליטור. השימוש בתקליטור הוא על אחריותו הבלעדית של המשתמש.

הוצאת הוד-עמי אינה תומכת בתוכנות המסופקות בתקליטור זה.

קרא על התוכנות המצורפות בתקליטור בקובץ ONCD.PDF.

המוצרים המותקנים בתקליטור זה מסופקים באחריות החברות המייצרות אותם.

הוצאת הוד-עמי אינה אחראית, בשום צורה שהיא, לאופן ולטיב התוכנות המותקנות.

התוכנות המומלצות להתקנה

1. תוכנת Adobe Acrobat Reader, כדי שניתן יהיה לעיין וגם להדפיס את הפרקים לדוגמה ואת תוכן העניינים של כל ספר בהוצאת הוד-עמי.
2. תוכנת TCLite (מהדר שפת C). התקן אותה רק אם אתה זקוק לה.

להתקנת שאר התוכנות יש ללחוץ על **התקנת תוכנות**, לאשר את תנאי השימוש בתקליטור וללחוץ על **תוכנות נוספות**. פירוט רחב יותר לגבי חלק מהתוכנות המצורפות בתקליטור תוכל לקרוא בקובץ ONCD.PDF אשר בתקליטור.

Adobe Acrobat Reader

התקן תוכנה זו כדי שתוכל לעיין ולהדפיס את הפרקים לדוגמה ואת תוכן העניינים של כל ספר בהוצאת הוד-עמי.

אם יצאת מממשק התקליטור:

1. ודא שהתקליטור נמצא בכונן התקליטורים.
2. בשולחן העבודה, לחץ לחיצה כפולה על הסמל **המחשב שלי**.
3. לחץ לחיצה כפולה על סמל **כונן התקליטורים**.
4. לחץ לחיצה כפולה על הקובץ **hodami.exe**.
5. לחץ על **התקנת תוכנות**.
6. לחץ על לחצן האפשרויות ליד **אני מאשר את תנאי השימוש בתקליטור**.
7. לחץ **המשך**.
8. לחץ על **Acrobat Reader**.
9. אשף ההתקנה מתקין את הרכיבים הנדרשים.

במהלך ההתקנה, עליך ללחוץ על **Next**, **Accept** ו-**Next** פעם נוספת כדי לבצע את ההתקנה.

10. בסיום ההתקנה, עשויה להופיע על המסך תיבת דו-שיח **התנגשות בין גירסאות** ומייד אחר כך להיעלם. על המסך תופיע תיבת הודעה של תוכנית ההתקנה. לחץ על **אישור**. בתיבת הדו-שיח **התנגשות בין גירסאות** לחץ על **כן**, כדי לשמור את גרסת הקובץ שלך.

קוד המקור שבספר

התיקיה **Books\59345** מכילה את קוד המקור המופיע בספר. התיקיה מחולקת לתת-תיקיות לפי מספרי הפרקים.

אופן העתקת קוד המקור וקבצים קשורים אחרים הוא כלהלן:

1. בממשק התקליטור לחץ על **קוד מקור**.
2. בתיבת הדו-שיח הנפתחת, בתיבה **מצא תיק מספר**, הקלד את מספר הפריט (מספר הפריט של ספר זה הוא **59345**) ולחץ **חפש**.
3. בתיבת הדו-שיח הבאה המופיעה לחץ על **העתק**.
4. שים לב: רצוי שלא לשנות את מיקום ברירת המחדל של העתקת הקבצים.
5. אם מופיעה תיבת אזהרה המציינת כי התיקיה שנבחרה אינה קיימת, השב **כן** כדי לאפשר את יצירתה.
6. בסיום תהליך ההעתקה תופיע תיבת הודעה. לחץ **סיום** כדי לסגור אותה ולשוב לממשק התקליטור.

Microsoft Internet Explorer 6 - התקנת הגירסה המצורפת

גירסה זו של Internet Explorer המצורפת בתקליטור, מיועדת להתקנה אך ורק במערכות בהן מערכת ההפעלה היא Windows 98 ומעלה ובעברית מלאה (לחצן "התחל" בפינה הימנית התחתונה של המסך).

אין אפשרות להתקין גירסה זו במחשב בו מותקנת מערכת ההפעלה Windows 95.

אם תנסה להתקין גירסה זו במחשב בו מערכת ההפעלה אינה עברית אתה עלול לגרום נזק למערכת ההפעלה, דבר שעשוי לדרוש ממך התקנה מחדש של מערכת ההפעלה והתוכנות הנלוות.

אם יצאת מממשק התקליטור:

1. ודא שהתקליטור נמצא בכונן התקליטורים.
2. בשולחן העבודה, לחץ לחיצה כפולה על הסמל **המחשב שלי**.
3. לחץ לחיצה כפולה על סמל **כונן התקליטורים**.
4. לחץ לחיצה כפולה על הקובץ **hodami.exe**.
5. לחץ על **התקנת תוכנות**.
6. לחץ על לחצן האפשרויות ליד **אני מאשר את תנאי השימוש בתקליטור**.
7. לחץ **המשך**.
8. לחץ על **תוכנות נוספות**.
9. מצא את התיקיה **Software\IE6\i386** ובחר בקובץ בשם **Setup.exe**.
10. לחץ עליו לחיצה כפולה.
11. לחץ על לחצן **אישור**.
12. פעל על פי ההוראות שעל המסך.

קרא עוד על התוכנות המצורפות בקובץ ONCD שבתקליטור

שאלות נפוצות

בחלק זה של התקליטור תמצא אוסף של שאלות נפוצות ותשובות במיגוון נושאים: איך מחפשים בספר? כיצד מזמינים ספר דרך האתר באינטרנט? למי פונים בבקשת תמיכה? האם ניתן לבצע הזמנה טלפונית? וכדומה.

הערה חשובה!

הוצאת **הוד-עמי** אינה תומכת בתוכנות המסופקות בתקליטור זה. התוכנות ניתנות ללקוח כבונוס, לשימושו האישי, והן מסופקות AS-IS כפי שסופקו להוצאת **הוד-עמי** על ידי יצרן התוכנה.

התמיכה ניתנת ללקוחות ההוצאה בנוגע לספרים הרואים אור בהוצאה.

ניתן לשאול שאלות, להציע הצעות ולבקש עזרה בהפעלת קוד או קובץ המוזכר בספר.

מחלקת התמיכה של הוצאת הוד-עמי אינה תומכת ביישומים ובתוכנות המסופקות בתקליטור.

צור קשר

תוכל ליצור עמנו קשר באמצעי התקשורת הבאים:

טלפון: 09-9564716

פקס: 09-9571582

דואר אלקטרוני בנושא כללי: info@hod-ami.co.il

דואר אלקטרוני בנושאי תמיכה: support@hod-ami.co.il

אנשי התמיכה שלנו ערוכים לתת לך את העזרה הדרושה לך בשימוש בספר.

אין אנו נותנים תמיכה הקשורה בתוכנות המצורפות בתקליטור.

בפנייתך אלינו לקבלת תמיכה, אנא הצטייד בנתונים הבאים:

1. שמו המלא של הספר.
2. מספר דאנאקוד, כפי שמופיע על הכריכה האחורית של הספר (למשל, בספר זה 259-10345).
3. פרטים מדויקים של הבעיה (הכוללים מספר פרק, מספר עמוד, שם קובץ).
4. אמצעי קשר בו נוכל להשיג אותך: טלפון נייד, פקס, E-Mail.

אם יש לך שאלה או הערה בנוגע לספר מספרי הוצאת **הוד-עמי**, נשמח לשמוע ממך.

יש מספר דרכים בהן תוכל להעביר לנו את הערותיך:

בדואר אלקטרוני: support@hod-ami.co.il

בפקס: 09-9571582

בדואר: ת.ד. 6108 הרצליה 46160

בפנייתך אנא ציין את הפרטים הבאים: שמו המדויק של הספר, מספר עמוד, מספר שורה מלמעלה/מלמטה, מה לא בסדר, מה לדעתך צריך להיות.

תודה רבה שבחרתם **הוד-עמי**

טלפון: 09-9564716 פקס: 09-9571582

אתר באינטרנט: http://www.hod-ami.co.il

דואר אלקטרוני: info@hod-ami.co.il

קרא קובץ ONCD שנמצא בתקליטור לפרטים נוספים לגבי תכולת התקליטור ולהוראות התקנה של חלק מהתוכנות המצורפות.

אינדקס ועוד

שאר הספר הינו באנגלית. כיוון הקריאה משמאל, מסוף הספר.

כלומר, פתח את הספר בסופו ושם תמצא את ה- References בעמוד 1.

References..... 1

Index of Patterns and Idioms 6

Index 10

util.concurrent, 148, 189
utility classes, 12
as alternative to constant interfaces, 90
in Collections Framework, 54

V

variables
atomic operations on, 190
declaring with interface types, 156
local, 141
See also local variable
loop, 142
views
as use of nonstatic member classes, 92
locking for, 209
naming conventions for, 167
object reuse and, 15
to avoid subclassing, 31, 55
to maintain immutability, 125
volatile modifier, 192
examples, 40, 69, 205
vs. subclassing, 82, 85

W

wait loop, 201-203
while loop, prefer for loop to, 142
wrapper classes, 74-75
as alternative to constant interfaces, 89
for synchronization, 200
for synchronization of unsynchronized classes, 200

X

XML, for JavaBean persistence, 215

Z

zero-length arrays
immutability of, 134
vs. null as return value, 134-135

deserialization and, 223
examples, 111, 220
logical state of an object and, 223
readResolve and, 107
singletons and, 231
transient modifier, 220
transitivity
compareTo and, 55
equals and, 28
translation
of exceptions, 121, 178
types
radically different, 131
typesafe enums, 105
adding behaviors to, 106-109
anonymous classes and, 108
as replacement for C enum, 104-114
constructors for, 105
disadvantages of, 112
equals and, 26, 110
hashCode and, 110
implementing with top-level class vs. static member class, 109
readResolve and, 231
variants
comparable, 106
extensible, 109
extensible serializable, 110
serializable, 107

U

unchecked exceptions, 172, 174
documenting, 136, 181
idiom for highlighting, 181
ignoring, 187
standard, 176
transforming checked exceptions into, 175
vs. checked exceptions, 172
unintentional object retentions, *See* memory leaks
unintentionally instantiable classes, 12
unions, class hierarchies as replacement for, 100-103
user, 3

performance and, 196, 199
providing in a subclass of an unsynchronized class, 200
synchronized modifier
as an implementation detail, 208
documentation and, 208
purpose of, 189
synthetic fields, 217

T

thread groups, 211
thread priorities, 206
thread safety, 210
and immutability, 65
documenting, 208-210
levels of, 208
ThreadGroup API and, 211
thread scheduler, 204-207
thread termination, cooperative, 192
Thread.yield, 206
testing and, 207
thread-compatible, 209
vs. thread-safe, 200
thread-hostile, 209
threads, 189-211
thread-safe, 208
vs. thread-compatible, 200
thread-safety
classes with public mutable fields and, 61
immutable objects and, 65
@throws tag, 119, 136, 137, 181
toString, 25
documenting return value of, 42-44
exceptions and, 183
general contract for, 42
overriding, 42-44, 106
providing programmatic access to data returned by, 44
return value as a de facto API, 44
transient fields, 220
custom serialized form and, 223
defaultReadObject and, 221, 223
defaultWriteObject and, 221, 222

for immutable objects, 67
for instance-controlled classes, 6
naming conventions for, 9
strategy pattern and, 117
static fields
immutable objects and, 11
initialize-on-demand holder class and, 194
strategy pattern and, 117
synchronization of mutable, 200
static initializer, 14
static member classes, 91
and serialization, 217
common uses of, 91, 92, 94
for shortening parameter lists, 127
prefer to nonstatic member classes, 91-95
prefer to strings for representing an aggregate, 152
to implement strategy pattern, 117
to implement typesafe enums, 109
strategy interface, 116
stream unique identifiers, 214, 223
string concatenation, 155
string representation, 42-44
strings, as inappropriate substitutes for other types, 152-154
structure, class as replacement for, 97-99
subclassing, 3
access levels of methods and, 61
equals and, 28, 31
of RuntimeException vs. Error, 172
prohibiting, 8, 12, 82
See inheritance
summary description, 138
symmetry
compareTo and, 55
equals and, 27
synchronization
fine-grained, 200
for mutual exclusion, 189
for shared mutable data, 189-195
for thread communication, 190-195
internal, 200
of atomic data, 190, 192

documenting for, 219, 222
effect on exported APIs, 213
extralinguistic mechanism, 214
immutability and, 70
inheritance and, 215
inner classes and, 217
interface extension and, 215
JavaBeans and, 213
serialized form
as part of exported API, 213
custom, *See* custom serialized form
default, *See* default serialized form
defaultWriteObject and, 221
documenting, 219
of inner class, 217
of singletons, 231
of typesafe enum, 107
serialver utility, 223
service provider framework, 7
reflection and, 160
signature, 3, 126-127
simulated multiple inheritance, 87
singletons, 10
deserialization and, 11
enforcing with a private constructor, 10-11
incompatibility with readObject, 230
readResolve and, 11, 230
serialized form of, 231
skeletal implementation, 85, 178
specific notification, 203
state transition, 65, 189
state-testing method, 170-171, 175
static factory methods, 5, 11, 13
advantages over constructors, 5
and immutable objects, 66
anonymous classes within, 93
as basis of service provider frameworks, 7
as replacement for cloning, 51
as replacement for constructors, 5-9
disadvantages, compared to constructors, 8
flexibility of, 6

recovery code, 186
recursive lock, 199
redundant fields, 34, 39
reference types, 3
    compareTo and, 56
    equals and, 32
    hashCode and, 38
reflection
    clone method and, 45
    drawbacks of, 158, 160
    for object creation, 159
    original purpose of, 158
    to break run-time dependencies, 160
    vs. interfaces, 158-160
reflexivity
    compareTo and, 55
    equals and, 27
@return tag, 137
run-time exceptions, 172
    documenting, 136
    vs. checked exceptions, 172-173

S

safety
    ensuring, 189-195
    failures, 195, 199
    wait and, 201
scope of variables
    local, 141-144
    loop, 142
    obsolete references and, 18
self-use
    documenting, for inheritance, 78
    eliminating, for inheritance, 82
semaphores, 202
@serial tag, 219
serial version UIDs, 214
    declaring in serializable classes, 223
@serialData tag, 222
serialization, 213-232
    costs of, 213-215

R

radically different types, 131
readObject, 224-229
    acceptability of default, 229
    as another constructor, 224
    default serialized form and, 219, 223
    defensive, 224-229
    defensive copying in, 228
    final fields and, 228
    for immutable objects, 70
    incompatibility with singletons, 230
    overridable methods and, 81, 229
    readResolve as alternative to, 231
    transient fields and, 223
readResolve
    access levels of, 82, 112, 232
    as alternative to readObject, 231
    examples, 107, 111, 112, 230, 231
    for immutable objects, 70
    for instance-controlled classes, 231
    for singletons, 11, 230
    for typesafe enums, 107, 110
    when to provide, 230-232
read-write locks, 202
recipes
    clone, 51
    compareTo, 55-57
    composition, 73
    equals, 32
    finalizer guardian, 23
    hashCode, 38
    invoking wait, 201
    noninstantiable classes, 12
    readObject, 229
    singletons, 10
    static factory method, 5
    typesafe enums
        extensible, 109
        extensible serializable, 110
        serializable, 107
        simple, 105

eliminating obsolete references to, 17-19
function, 93
immutable, *See* immutability
methods common to all, 25-57
process, 93
reuse, 13-16
serializing, 213-232
using base classes to refer to, 157
using interfaces to refer to, 156-157
viewing in partially initialized state, 81, 193
obsolete object references, 17-19, 47, 185
optimizations, 162-164
==instead of equals, 6, 33
initialization and, 194
notify vs. notifyAll, 202
object reuse, 13-16
static initialization, 14-15
StringBuffer and, 155
try-catch blocks and, 170
overloading, *See* methods, overloadings
overriding, *See* methods, overridings

P

package-private
access level, 3, 60
constructors, 67, 82
packages, naming conventions for, 165, 166
@param tag, 136, 137
parameter lists, 126-127
of constructors, 6
parameterless constructor, 12, 215
parameters, checking for validity, 119-121
performance, *See* optimizations
performance model, 164
postconditions, 136
preconditions, 136, 181
primitives, 3
compareTo and, 56
equals and, 33
hashCode and, 38
process objects, 93

native, 22, 161
ordering computation in, 185
overloadings, 128-133
same number of parameters in, 130-132
static selection among, 128
overridings, 128-129
dynamic selection among, 128
parameter lists for, 126
small and focused, 144
state-testing, vs. distinguished return value, 171
static factory, *See* static factory methods
mixin interfaces, 45, 84
module, 2
Monty Python reference, subtle, 134
mutable companion classes, 67
mutual exclusion, 189

N

naming conventions, 9, 165, 165-168
native methods, 22, 161
native peers, 22
nested classes, 60, 91-95
access levels of, 60
as concrete strategy classes, 117
nonhierarchical type frameworks, 84
noninstantiability, 12
non-nullity
in general contract of compareTo, 55
in general contract of equals, 32
nonstatic member classes, 91
example, 92
for defining adapters, 92
prefer static member classes to, 91-95
notify vs. notifyAll, 202-203

O

object pool, 16
objects, 3
avoiding reflective access, 158-160
creating and destroying, 5-24
deserializing, 213-232

using private objects for, 210
logging, 180
long, use for monetary calculations, 149-151
loop variables, 142
loops
for invoking wait, 201-203
minimizing scope of variables and, 142
prefer for to while, 142

M

member classes, 3
nonstatic, *See* nonstatic member classes
static, *See* static member classes
static vs. nonstatic, 91-95
members, 3
accessibility of, 60
minimizing accessibility of, 59-62
memory leaks, 18
causes of, 18-19
example, 17
memory model, 63, 190
methods, 3, 119-139
access levels of, 60, 61
accessor, vs. public fields, 98-99
adding to exception classes, 173
alien, *See* alien methods
checking parameters for validity, 119-121
common to all objects, 25-57
defensive copying before parameter checking, 123
designing signatures of, 126-127
documenting, 136-137, 138
exceptions thrown by, 181-182
overridable, 78
thread safety of, 208-210
explicit termination, 21-22
failure atomicity and, 185-186
forwarding, *See* forwarding methods
immutability and overriding, 67
inheriting doc comments, 139
naming conventions for, 9, 166, 167
naming of, 126

evolving, 88
extending Serializable, 215
for defining mixins, 84
for nonhierarchical type frameworks, 84
for type definition, 89-90
member, 3
mixin, 45, 84
naming conventions for, 165-166
purpose of, 45, 89-90
Serializable, 213-217
skeletal implementations and, 85-88
strategy, 116
to enable functionality enhancements, 85
to refer to objects, 156-157
vs. abstract classes, 84-88
vs. reflection, 158-160

J

JavaBeans
serialization and, 213
XML and, 215
Javadoc, 136

L

lazy initialization, 7, 15, 40, 69
double-check idiom for, 193
initialize-on-demand holder class for, 194
libraries, 145-148
for multithreaded programming, 189
liveness, ensuring, 196-200, 201
thread priorities and, 206
local classes, 91, 94
local variables, 141
declaring, 141
initializing, 142
minimizing scope of, 141-144
naming conventions for, 166, 167
locks
for classes at various levels of thread safety, 208
in multithreaded programs, 189-195
recursive, 199

prohibiting, 82
See also extension
serialization and, 215
uses of, 76
using internal objects for locking and, 210
initialization
at object creation, 70
defensive copying and, 63
examples, 194
of double-check idiom, 193
of lazy, 40, 69
of loop variables, 142
of static, 14, 194
to allow serializable subclasses, 216
initialize-on-demand holder class and, 194
lazy, 7, 15, 40, 69, 193
of fields on deserialization, 223
of local variables, 142
static, 14
to allow serializable subclasses, 216
viewing objects before complete, 81, 193
initialize-on-demand holder class, 194
inner classes, 91
and serialization, 217
anonymous, *See* anonymous classes
extending skeletal implementations with, 86
instance-controlled classes, 231
readResolve and, 231
singleton, 10
typesafe enum, 105
utility classes, 12
int, use for monetary calculations, 149-151
interface inheritance, 71
interface-based frameworks, 7, 84-88
interfaces, 3, 59-95
access levels of, 60
as parameter types, 127
Cloneable, 45-52
Comparable, 53-57
constant, 89-90
documenting, 136, 138, 181

I

immutability, 63-70
advantages of, 65
and hashCode, 40
canonical forms and, 34
Cloneable and, 51
constants and, 61, 166
defensive copies and, 122, 124
disadvantage of, 66
examples, 64
broken, 122
serialization and, 224
static factories and, 68
failure atomicity and, 185
functional approach and, 65
object reuse and, 13
readObject and, 224-228
readResolve and, 232
rules for, 63
serialization and, 70, 224-229
static factory methods and, 67
thread safety and, 208
zero length arrays and, 134
immutable, as a level of thread safety, 208
implementation inheritance, 71
implements, 3
inconsistent with equals, 55
information hiding, 59, 162
inheritance, 3
and encapsulation, 71
causes of fragility in, 73
designing for, 79-83
documenting for, 78-79
eliminating self-use of overridable methods for, 83
example, 71
favor composition over, 71-77
hooks to facilitate, 79
implementation vs. interface, 71
incompatibility with eadResolve, 232
of doc-comments, 139
overridable methods and, 80

forwarding, 73
to private extension of skeletal implementation, 86
forwarding methods, 73
ensuring consistent behavior in overloaded methods, 132
example, 74
in wrapper classes, 75, 200
object composition and, 73
frameworks
callback, 75
class-based, 157
Collections Framework, 147
interface-based, 7
nonhierarchical type, 84
object serialization, 213
service provider, 7
function objects, 93, 115-117, 127
function pointers, 115-117
functional approach, 65

G

general contract, 25, 181
clone, 45
compareTo, 53
equals, 26
hashCode, 36
implementing interfaces and, 84
toString, 42

H

handoff, 125
hashCode, 25
general contract for, 36
how to write, 38
immutable objects and, 40
lazy initialization and, 40, 69
overriding equals and, 34, 36-41
typesafe enums and, 110
heap profiler, 19
helper classes, 127

default values of, 223
defensive copies of, 124
documenting, 136, 138, 219
encapsulation and, 98
equals and, 33
exposing, 61, 98
final, *See* final fields
hashCode method and, 38
immutability and, 63
interface types and, 156
naming conventions for, 166, 167
protected, 79
public, 61
redundant, 34, 39
reflection and, 158
serialization and, 223
stateless classes and, 115
synthetic, 217
thread safety and, 61
transient, *See* transient fields
final fields
constant interface pattern and, 89
constants and, 61, 166
incompatibility with clone, 48
incompatibility with defensive copy in readObject, 228
readResolve and, 231
references to mutable objects and, 61
to implement singletons, 10
typesafe enums and, 105
finalizer chaining, 22
finalizer guardian, 23
finalizers, 20-24
chaining of, 22
critical persistent state and, 21
execution time, 20
uses for, 22
vs. explicit termination method, 21-22
fine-grained synchronization, 200
Float, compareTo inconsistent with equals, 55
float, when to avoid, 149-151
for loop, prefer to while loop, 142

transforming checked into unchecked, 175
translation, 121
translation of, 178
uncaught, during finalization, 21
explicit termination method, 21-22
exported API, 3
access levels and, 60
constant interface pattern and, 89
documentation comments and, 139
member classes and, 93
serialization and, 213
synchronized modifier and, 208
extension, 3
Cloneable interface and, 81
of class hierarchies, 102
of classes, 71, 82
appropriateness, 76
clone and, 46
compareTo and, 55
equals and, 30
immutability and, 67
private constructors and, 68
Serializable interface and, 81
of interfaces, 85
of skeletal implementations, 86
of typesafe enums, 109
Serializable interface and, 215
extralinguistic mechanisms
cloning, 45
native methods, 161
reflection, 158
serialization, 214

F

failure atomicity, 121, 185-186
fields, 3
access levels of, 60
clone and, 47
compareTo and, 56
constant, 61
constant interface pattern and, 89

Rectangle, 101
Shape, 101
Singe, 84
SingerSongwriter, 85
SongWriter, 85
Square, 102
Stack, 17, 47
StoppableThread, 191, 192
StringLengthComparator, 115, 116
StringList, 219, 220
StrLenCmp, 117
Sub, 81
Suit, 105, 106
Super, 80
ThreadLocal, 153, 154
UtilityClass, 12
WaitQueuePerf, 206
WordList, 53
WorkerThread, 197, 198, 205
WorkQueue, 196, 204
exception chaining, 178
exception translation, 121, 178
exceptions, 169-187
avoidance of, 174-175, 180
chaining of, 178
checked vs. run-time, 172-173
commonly used, 119, 177
control flow and, 170
defining methods on, 173
detail messages for, 183-184
documentation of, 181-182
as part of method documentation, 136, 137
failure-capture information and, 183
favor standard, 176-177
for exceptional conditions only, 169-171
handling with ThreadGroup.uncaughtException, 211
ignoring, 187
performance and, 170
purpose of
checked, 172
run-time, 172

examples
A, B, C, 129
AbstractFoo, 216
AbstractMapEntry, 87
anonymous inner class, 93
BogusPeriod, 225
Calculator, 94
CaseInsensitiveString, 27
Circle, 101
CollectionClassifier, 128
ColorPoint, 29, 31
Comparator, 116
Complex, 64, 68
DeadlockQueue, 198
Degree, 138
DisplayQueue, 197
Elvis, 10, 230
Entry, 48, 49, 219, 221
ExtendedOperation, 112
Foo
finalizer guardian, 23
provider framework, 8
serializable subclass of nonserializable class, 217
FooHolder, 194
HashTable, 48, 49
Host, 117
InstrumentedHashSet, 71
InstrumentedSet, 74
Key, 153
MutablePeriod, 226
MyIterator, 92
MySet, 92
Name, 218
Operation, 94, 108, 111
Overriding, 129
Period, 122, 224
Person, 13, 14
PhoneNumber, 36
PhysicalConstants, 89, 90
PlayingCard, 104
Point, 29, 98

of writeObject for serialization, 222
summary description in, 138
synchronized modifier and, 208
documentation comments, 136
double, when to avoid, 149-151
double-check idiom, 193, 193-195

E

encapsulated structure class, 98
encapsulation, 59, 97
broken by inheritance, 71
data fields and, 98
enclosing instance, 91
anonymous classes and, 93
finalizer guardian and, 23
local class and, 94
nonstatic member class and, 91
serialization and, 217
enum, classes as replacement for, 104-114
equals, 25-35
accidental overloading of, 35
canonical forms and, 34
examples
accidental overloading, 35
as unsupported operation, 26
for typesafe enums, 110
forwarding, 132
general contract and, 32
general contract of, 28, 31, 32, 87
preventing overriding, 111
violation of general contract, 27, 29, 30
extending an abstract class and, 31
extending an instantiable class and, 30
general contract for, 26-32
how to write, 32
method-forwarding and, 132
overriding hashCode and, 34, 36-41
typesafe enums and, 26, 110
unreliable resources and, 34
when to override, 25-26
errors, 172

readResolve as alternative to, 231
vs. object reuse, 16
degenerate class, 98
delegation, 75
denial-of-service attack, 210
deserialization, 213-232
as a constructor, 214
preventing completion of, 226
singletons and, 11
typesafe enums and, 107
detail message, 183
discriminated unions, 100
class hierarchies as replacement for, 100-103
distinguished return value, vs. state-testing method, 171
doc comments, 136
documentation, 136-139
@param tag, 136, 137
@return tag, 137
@serial tag, 219
@serialData tag, 222
@throws tag, 119, 136, 137, 181
for inheritance, 78-79
for serialized fields, 219
HTML in Javadoc, 137
inheritance of doc comments, 139
Javadoc, 136-139
links to architecture documents from Javadoc output, 139
moving descriptions of common exceptions to class comment, 182
of an object's state after an exception, 186
of conditional thread safety, 209
of exceptions, 181-182
of methods, 136
of parameter restrictions, 119
of postconditions, 136
of preconditions, 136, 181
of required locks, 209, 210
of return value of toString, 42
of self-use of overridable methods, 78, 82
of side effects, 136
of thread safety, 136, 208-210
of transfer of control, 125

in singletons, 10
use of overridable methods in, 80
for typesafe enums, 105, 109
overloading, 130
parameterless, 12, 215
readObject as a, 224
replacing with static factory, 5-9
signature of, 6
cooperative thread termination, 192
copy constructor, 51, 66
custom serialized form, 213, 218-223
example, 220
transient fields and, 223

D

data consistency
maintaining in face of failure, 185-186
synchronization, 189-195
unreliable resources and, 34
deadlock, avoiding, 196, 199
default access, 3, 60
default constructor, 12
default serialized form
criteria for accepting, 218
disadvantages of, 220
initial values of transient fields and, 223
transient modifier and, 220
defaultReadObject
examples, 217, 221, 226, 228
transient fields and, 221, 223
defaultWriteObject
examples, 217, 221
transient fields and, 221, 222
defensive copy, 122-125
clone and, 123, 124, 228
deserialization and, 228
immutable objects and, 66
of array, 125
of mutable input parameters, 123-124
of mutable internal fields, 124
readObject and, 228

composition, 8, 73
    favor over inheritance, 71-77
concrete strategy, 115
concurrency
    documenting method behavior for, 208-210
    fine-grained synchronization and, 200
    increasing for testing, 207
    open calls and, 199
    utilities to simplify multithreaded programming, 148, 189
conditionally thread-safe, 208
    denial-of-service attack and, 210
    documenting, 209
    documenting lock object for, 210
consistency requirement
    in equals contract, 26, 32
    in hashCode contract, 36
consistency, data, *See* data consistency
consistent with equals, 55
constant interface, 89
constant utility class, 90
constants
    accessibility of, 61
    data types of, 61
    in interfaces, 89-90
    in typesafe enums, 104-114
    naming convention for, 166
constructor, 3
    calling overridable methods in, 80
    clone as a, 48
    copy, 51, 66
    default, 12
    defensive copying and, 123
    deserialization as a, 214
    documenting self-use, 78
    enforcing noninstantiability with, 12
    enforcing singleton property with, 10-11
    establishing invariants, 65, 70
    examples
        enforcing noninstantiability, 12
        for typesafe enums, 105
        in immutable classes, 122, 123

utility, *See* utility classes
clients, 3
clone, 25, 45-52
as a constructor, 48
as another constructor, 81
copy constructor as alternative to, 51-52
defensive copies and, 62, 123-124, 228
examples
defensive copies, 62, 124, 224
implementing, 46, 48, 49
general contract of, 45-46
immutable objects and, 66
incompatibility with final fields, 48
nonfinal methods and, 50, 81
references to mutable objects and, 47-50
Cloneable interface, 45
alternatives to, 51-52
behavior of, 45
designing for inheritance and, 81
examples, 48, 49
instructions for implementing, 51
purpose of, 45
combinatorial explosion, 85
companion classes, mutable, 67
Comparable interface, 53-57
Comparator
anonymous classes and, 93
class, 56, 115
examples, 93, 115, 116, 117
instance, 93
interface, 116
compareTo, 25, 53
consistent or inconsistent with equals, 55
differences from equals, 54
examples, 56, 57, 106, 226
overloading in, 132
use of, 14
using, 122, 123, 150, 224, 228
general contract for, 53-55
instructions for writing, 55-57

C

callback frameworks, 75
callbacks, 75, 115
canonical form, 34
Chapter, 1
checked exceptions, 172
    accessor methods in, 173, 184
    avoiding unnecessary use of, 174-175
    documenting, 181
    failure atomicity and, 185
    ignoring, 187
    purpose of, 172
    transforming into unchecked exception, 175
class hierarchies, 84
    as replacement for discriminated unions, 100-103
    combinatorial explosion of, 85
class-based frameworks, 157
classes, 3, 59-95
    access levels of, 60
    anonymous, *See* anonymous classes
    as replacement for C enum s, 104-114
    as replacement for C struct s, 97-99
    as replacement for function pointers, 115-117
    base, 157
    designing for inheritance, 79-83
    documenting, 136, 138
        for inheritance, 78-79
        thread safety of, 208-210
    helpers, for shortening parameter lists, 127
    hierarchies as replacement for discriminated unions, 100-103
    immutable, *See* immutability
    instance, 3
    levels of thread safety for, 208-209
    member classes, 3
    members, 3
    minimizing accessibility of, 59-62
    naming conventions for, 165-166
    singletons, *See* singletons
    stateful, 216
    stateless, 115
    unrelated, 55, 92, 131

alien methods, 196
deadlock and, 196-199
example, 197
safety failures and, 199
anonymous classes, 91, 93-94
as concrete strategy classes, 116
as function objects, 127
examples
finalizer guardian, 23
in adapters, 86
in typesafe enums, 94, 108
finalizer guardian and, 23
in adapters, 86
in typesafe enums, 108
limitations of, 93
uses of, 93
API
See exported API
API design
exceptions and, 170, 172
performance consequences of, 163
API elements, 4
documenting, 136-139
arrays, 3
defensive copying of, 62, 125
in public fields as security holes, 62
nonzero-length and mutability, 62, 125
zero-length and immutability, 134
zero-length vs. null as return value, 134-135
assertions, and parameter checking, 120
atomic data, 190
synchronization and, 190-192
auxiliary class, 91

B

base classes, 157
BigDecimal, for monetary calculations, 149-151
binary compatibility, 215
bounded buffers, 202
busy-wait, 204

Index

A

abstract classes
adding an aspect to, 31
as replacement for discriminated union, 101
designing for inheritance, 82
evolution of, vs. interfaces, 88
examples
adding behaviors to typesafe enum, 108
replacement for discriminated union, 101
skeletal implementation, 87
static member class, 94
for adding behaviors to typesafe enum, 108
for service provider framework, 8
for skeletal implementations, 82, 85
noninstantiability and, 12
vs. interfaces, 84-88
access control, 60
access levels, 3
method overriding and, 61
of classes and interfaces, 60
of constants, 61
of members, 60
of static member classes, 91
readResolve and, 232
accessor methods, 98
defensive copies and, 63, 124
examples
defensive copies, 124
immutability, 64
for failure-capture information, 173, 184
for information returned by toString, 44
for lower-level exception, 179
immutability and, 63
naming conventions for, 167
vs. public fields, 98-99, 103

skeletal implementation, 85
state-testing method, 170, 175
static factory method, 5-9
strategy, 115
string concatenation, excessive *(antipattern)*, 155
string overuse *(antipattern)*, 152

T

toString, 42-44
typesafe enum, 105
    behaviors attached to constants of, 108
    ordinal-based, 106
    serializable, extensible, 111

U

unsynchronized concurrent access *(antipattern)*, 189-195
utility class, 12

W

wait loop, 201
wrapper class, 74

iteration, high-performance list, 143

L

list iteration, high-performance, 143

M

mixin interface, 84

N

nonhierarchical type framework, 84
noninstantiability enforcement, 12
nonserializable stateful class allowing
serializable subclass, 216
null as substitute for zero-length array *(antipattern)*, 134-135

O

open call, 198
override-prevention method, 110

P

parameter check, 119-121
private lock object, 210
provider framework, 8

R

readObject, defensive, 224-229
readResolve, defensive, 231
reflective instantiation with interface access, 159

S

safe array access, 62
self-use elimination, 83
serializable class designed for extension *(antipattern)*, 215
serializable inner class *(antipattern)*, 217
simulated multiple inheritance, 87
singleton, 10-11

D

decorator, 75
defensive copy, 122-125
defensive readObject, 224-229
defensive readResolve, 231
discriminated union *(antipattern)*, 100
doc comment, 136-139
double-check idiom for lazy initialization *(antipattern)*, 193

E

eliminating self-use, 83
empty catch block *(antipattern)*, 187
encapsulated structure class, 98
equals, 32
exception chaining, 179
exception for loop control *(antipattern)*, 169
exception translation, 178
excessive string concatenation *(antipattern)*, 155
explicit serial version UID, 223
explicit termination method, 21-22

F

failure-capture, 183-184
finalizer chaining, 23
finalizer guardian, 23
floating point for monetary calculation *(antipattern)*, 149
function object, 115, 127

H

hashCode, 38

I

immutable class, 63
initialize-on-demand holder class, 194
int enum *(antipattern)*, 104
int enum, bit-flag variant of, 112
interface as type, 156
iteration, collection, 142

Index of Patterns and Idioms

A

adapter, 86, 92
adding an aspect, 31
antipattern
    busy-wait, 204
    constant interface, 89
    discriminated union, 100
    double-check idiom for lazy initialization, 193
    empty catch block, 187
    exception for loop control, 169
    excessive string concatenation, 155
    floating point for monetary calculation, 149
    int enum, 104
    null as substitute for zero-length array, 134-135
    serializable class designed for extension, 215
    serializable inner class, 217
    string overuse, 152
    unsynchronized concurrent access, 189-195
array printing, 147

B

busy-wait *(antipattern)*, 204

C

capability, 153
class designed for inheritance, 78-83
class hierarchy, 101-103
clone, 45-52
collection iteration, 142
compareTo, 55-57
constant interface *(antipattern)*, 89
constant utility class, 90
cooperative thread termination, 192
copy constructor, 51

[Vermeulen00] Vermeulen, Allan, Scott W. Ambler, Greg Bumgardener, Eldon Metz, Trevor Mesfeldt, Jim Shur, Patrick Thompson. *The Elements of Java™ Style,* Cambridge University Press, Cambridge, United Kingdom, 2001. ISBN: 0521777682.

[Weblint] *The Weblint Home Page.* Weblint.org. March 2001. <http://www.weblint.org/>.

[Wulf72] Wulf, W. A Case Against the GOTO. *Proceedings of the 25th ACM National Conference* 2 (1972): 791–797.

[Meyers98] Meyers, Scott. *Effective C++, Second Edition: 50 Specific Ways to Improve Your Programs and Designs.* Addison-Wesley, Reading, MA, 1998. ISBN: 0201924889.

[Parnas72] Parnas, D.L. On the Criteria to Be Used in Decomposing Systems into Modules. *Communications of the ACM* 15 (1972): 1053–1058.

[Posix] 9945-1:1996 (ISO/IEC) [IEEE/ANSI Std. 1003.1 1995 Edition] Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application: Program Interface (API) [C Language] (ANSI), IEEE Standards Press, ISBN: 1559375736.

[Pugh01a] *The Java Memory Model.* Ed. William Pugh. University of Maryland. March 2001. <http://www.cs.umd.edu/~pugh/java/memoryModel/>.

[Pugh01b] *The "Double-Checked Locking is Broken" Declaration.* Ed. William Pugh. University of Maryland. March 2001. <http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html>.

[Serialization] *Java™ Object Serialization Specification.* Sun Microsystems. March 2001. <http://java.sun.com/j2se/1.3/docs/guide/serialization/spec/serialTOC.doc.html>.

[Smith62] Smith, Robert. Algorithm 116 Complex Division. In *Communications of the ACM,* 5.8 (August 1962): 435.

[Snyder86] Synder, Alan. Encapsulation and Inheritance in Object-Oriented Programming Languages. In *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings,* 38–45, 1986. ACM Press.

[Thomas94] Thomas, Jim, and Jerome T. Coonen. Issues Regarding Imaginary Types for C and C++. In *The Journal of C Language Translation,* 5.3 (March 1994): 134–138.

[JavaBeans] *JavaBeans™ Spec.* Sun Microsystems. March 2001. <http://java.sun.com/products/javabeans/docs/spec.html>.

[Javadoc-a] *How to Write Doc Comments for Javadoc.* Sun Microsystems. January 2001. <http://java.sun.com/j2se/javadoc/writingdoccomments/>.

[Javadoc-b] *Javadoc Tool Home Page.* Sun Microsystems. January, 2001. <http://java.sun.com/j2se/javadoc/index.html>.

[JLS] Gosling, James, Bill Joy, Guy Steele, Gilad Bracha. *The Java™ Language Specification, Second Edition,* Addison-Wesley, Boston, 2000. ISBN: 0201310082.

[Kahan91] Kahan, William, and J. W. Thomas. *Augmenting a Programming Language with Complex Arithmetic,* UCB/CSD-91-667, University of California, Berkeley, 1991.

[Knuth74] Knuth, Donald. Structured Programming with go to Statements. *Computing Surveys* 6 (1974): 261–301.

[Lea01] *Overview of Package util.concurrent Release 1.3.0.* State University of New York, Oswego. January 12, 2001. <http://g.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurren
t /intro.html>.

[Lea00] Lea, Doug. *Concurrent Programming in Java™ : Design Principles and Patterns, Second Edition,* Addison-Wesley, Boston, 2000. ISBN: 0201310090.

[Lieberman86] Lieberman, Henry. Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems. *Proceedings of the First ACM Conference on Object-Oriented Programming Systems, Languages, and Applications,* pages 214–223, Portland, September 1986. ACM Press.

[Collections] *The Collections Framework.* Sun Microsystems. March 2001. <http://java.sun.com/j2se/1.3/docs/guide/collections/index.html>.

[Doclint] *Doclint.* Ernst de Haan. March, 2001. <http://www.znerd.demon.nl/doclint/>.

[Flanagan99] Flanagan, David. *Java™ in a Nutshell, Third Edition*, O'Reilly and Associates, Sebastopol, CA, 1999. ISBN: 1565924878.

[Gamma95] Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software,* Addison-Wesley, Reading, MA, 1995. ISBN: 0201633612.

[Gong99] Gong, Li. *Inside Java™ 2 Platform Security,* Addison-Wesley, Reading, MA, 1999. ISBN: 0201310007.

[Heydon99] Allan Heydon and Marc A. Najork. Performance Limitations of the Java Core Libraries. In *ACM 1999 Java Grande Conference,* pages 35–41. ACM Press, June 1999. Also available as <http://research.compaq.com/SRC/mercator/papers/Java99/final.pdf>

[Horstman00] Horstmann, Cay, and Gary Cornell. *Core Java™ 2: Volume II-Advanced Features,* Prentice Hall, Palo Alto, CA, 2000. ISBN: 0130819344.

[HTML401] *HTML 4.01 Specification.* World Wide Web Consortium. December 1999. <http://www.w3.org/TR/1999/REC-html401-19991224/>.

[J2SE-APIs] *Java™ 2 Platform, Standard Edition, v 1.3 API Specification.* Sun Microsystems. March 2001. <http://java.sun.com/j2se/1.3/docs/api/overview-summary.html>.

[Jackson75] Jackson, M.A. *Principles of Program Design,* Academic Press, London, 1975. ISBN: 0123790506.

References

[Arnold00] Arnold, Ken, James Gosling, David Holmes. *The Java™ Programming Language, Third Edition.* Addison-Wesley, Boston, 2000. ISBN: 0201704331.

[Beck99] Beck, Kent. *Extreme Programming Explained: Embrace Change.* Addison-Wesley, Reading, MA, 1999. ISBN: 0201616416.

[Bloch99] Bloch, Joshua. Collections. In *The Java™ Tutorial Continued: The Rest of the JDK™.* Mary Campione, Kathy Walrath, Alison Huml, and the Tutorial Team. Addison-Wesley, Reading, MA, 1999. ISBN: 0201485583. Pages 17–93. Also available as <http://java.sun.com/docs/books/tutorial/collections/index.html>.

[Campione00] Campione, Mary, Kathy Walrath, Alison Huml. *The Java™ Tutorial Continued: A Short Course on the Basics.* Addison-Wesley, Boston, MA, 2000. ISBN: 0201703939. Also available as <http://java.sun.com/docs/books/tutorial/index.html>.

[Cargill96] Cargill, Thomas. Specific Notification for Java Thread Synchronization. *Proceedings of the Pattern Languages of Programming Conference,* 1996.

[Chan00] Chan, Patrick. *The Java™ Developers Almanac 2000,* Addison-Wesley, Boston, MA, 2000. ISBN: 0201432994.

[Chan98] Chan, Patrick, Rosanna Lee, and Douglas Kramer. *The Java™ Class Libraries Second Edition, Volume 1,* Addison-Wesley, Reading, MA, 1998. ISBN: 0201310023.