كيف تفكر كعالم كمبيوتر

نسخة جافا

كيف تفكر كعالم كمبيوتر

نسخة جافا – الإصدار 1.0

Author

Allen B. Downey

ترجمة

محمد سعيد

هذا الكتاب هو الإصدار الأول للترجمة العربية لكتاب
"Think Java: How to Think Like a Computer Scientist" الإصدار 5.0.3،
كتبه آلان داوني، أستاذ مساعد في علوم الحاسوب في كلية فرانكلين أولين للهندسة؛ منشور بوساطة GreenTea Press في عام 2011.
الشكل الأساسي للكتاب (باللغة الإنكليزية) مكتوب باستخدام LATEX. والرسومات التوضيحية مرسومة باستخدام xfig. وهي برامج حرة، مفتوحة المصدر. تتوفر شفرة الكتاب الأصلي المصدرية على الموقع http://www.greenteapress.com/thinkapjava.html

Preface

> "As we enjoy great Advantages from the Inventions of others, we should be glad of an Opportunity to serve others by any Invention of ours, and this we should do freely and generously."

—Benjamin Franklin, quoted in *Benjamin Franklin* by Edmund S. Morgan.

Why I wrote this book

This is the fifth edition of a book I started writing in 1999, when I was teaching at Colby College. I had taught an introductory computer science class using the Java programming language, but I had not found a textbook I was happy with. For one thing, they were all too big! There was no way my students would read 800 pages of dense, technical material, even if I wanted them to. And I didn't want them to. Most of the material was too specific—details about Java and its libraries that would be obsolete by the end of the semester, and that obscured the material I really wanted to get to.

The other problem I found was that the introduction to object oriented programming was too abrupt. Many students who were otherwise doing well just hit a wall when we got to objects, whether we did it at the beginning, middle or end.

So I started writing. I wrote a chapter a day for 13 days, and on the 14th day I edited. Then I sent it to be photocopied and bound. When I handed it out on the first day of class, I told the students that they would be expected to read one chapter a week. In other words, they would read it seven times slower than I wrote it.

The philosophy behind it

Here are some of the ideas that make the book the way it is:

- Vocabulary is important. Students need to be able to talk about programs and understand what I am saying. I try to introduce the minimum number of terms, to define them carefully when they are first used, and to organize them in glossaries at the end of each chapter. In my class, I include vocabulary questions on quizzes and exams, and require students to use appropriate terms in short-answer responses.

- To write a program, students have to understand the algorithm, know the programming language, and they have to be able to debug. I think too many books neglect debugging. This book includes an appendix on

debugging and an appendix on program development (which can help avoid debugging).
I recommend that students read this material early and come back to it often.

- Some concepts take time to sink in. Some of the more difficult ideas in the book, like recursion, appear several times. By coming back to these ideas, I am trying to give students a chance to review and reinforce or, if they missed it the first time, a chance to catch up.

- I try to use the minimum amount of Java to get the maximum amount of programming power. The purpose of this book is to teach programming and some introductory ideas from computer science, not Java. I left out some language features, like the switch statement, that are unnecessary, and avoided most of the libraries, especially the ones like the AWT that have been changing quickly or are likely to be replaced.

The minimalism of my approach has some advantages. Each chapter is about ten pages, not including the exercises. In my classes I ask students to read each chapter before we discuss it, and I have found that they are willing to do that and their comprehension is good. Their preparation makes class time available for discussion of the more abstract material, in-class exercises, and additional topics that aren't in the book.

But minimalism has some disadvantages. There is not much here that is intrinsically fun. Most of my examples demonstrate the most basic use of a language feature, and many of the exercises involve string manipulation and mathematical ideas. I think some of them are fun, but many of the things that excite students about computer science, like graphics, sound and network applications, are given short shrift.

The problem is that many of the more exciting features involve lots of details and not much concept. Pedagogically, that means a lot of effort for not much payoff. So there is a tradeoff between the material that students enjoy and the material that is most intellectually rich. I leave it to individual teachers to find the balance that is best for their classes. To help, the book includes appendices that cover graphics, keyboard input and file input.

Object-oriented programming

Some books introduce objects immediately; others warm up with a more procedural style and develop object-oriented style more gradually. This book uses the "objects late" approach.

Many of Java's object-oriented features are motivated by problems with previous languages, and their implementations are influenced by this history.

Some of these features are hard to explain if students aren't familiar with the problems they solve.

It wasn't my intention to postpone object-oriented programming. On the contrary, I got to it as quickly as I could, limited by my intention to introduce concepts one at a time, as clearly as possible, in a way that allows students to practice each idea in isolation before adding the next. But I have to admit that it takes some time to get there.

The Computer Science AP Exam

Naturally, when the College Board announced that the AP Exam would switch to Java, I made plans to update the Java version of the book. Looking at the proposed AP Syllabus, I saw that their subset of Java was all but identical to the subset I had chosen.

During January 2003, I worked on the Fourth Edition of the book, making these changes:

- I added sections to improve coverage of the AP syllabus.
- I improved the appendices on debugging and program development.
- I collected the exercises, quizzes, and exam questions I had used in

my classes and put them at the end of the appropriate chapters. I also made up some problems that are intended to help with AP Exam preparation.

Finally, in August 2011 I wrote the fifth edition, adding coverage of the GridWorld Case Study that is part of the AP Exam.

Free books!

Since the beginning, this book has under a license that allows users to copy, distribute and modify the book. Readers can download the book in a variety of formats and read it on screen or print it. Teachers are free to print as many copies as they need. And anyone is free to customize the book for their needs.

People have translated the book into other computer languages (including Python and Eiffel), and other natural languages (including Spanish, French and German). Many of these derivatives are also available under free licenses.

Motivated by Open Source Software, I adopted the philosophy of releasing the book early and updating it often. I do my best to minimize the number of errors, but I also depend on readers to help out.

The response has been great. I get messages almost every day from people who have read the book and liked it enough to take the trouble to send in a "bug report." Often I can correct an error and post an updated version within

a few minutes. I think of the book as a work in progress, improving a little whenever I have time to make a revision, or when readers send feedback.

Oh, the title

I get a lot of grief about the title of the book. Not everyone understands that it is -mostly- a joke. Reading this book will probably not make you think like a computer scientist. That takes time, experience, and probably a few more classes.

But there is a kernel of truth in the title: this book is not about Java, and it is only partly about programming. If it is successful, this book is about a way of thinking. Computer scientists have an approach to problem-solving, and a way of crafting solutions, that is unique, versatile and powerful. I hope that this book gives you a sense of what that approach is, and that at some point you will find yourself thinking like a computer scientist.

Allen Downey
Needham, Massachusetts
July 13, 2011

Contributors List

When I started writing free books, it didn't occur to me to keep a contributors list. When Jeff Elkner suggested it, it seemed so obvious that I am embarrassed by the omission. This list starts with the 4th Edition, so it omits many people who contributed suggestions and corrections to earlier versions.

- Ellen Hildreth used this book to teach Data Structures at Wellesley College, and she gave me a whole stack of corrections, along with some great suggestions.
- Tania Passfield pointed out that the glossary of Chapter 4 has some leftover terms that no longer appear in the text.
- Elizabeth Wiethoff noticed that my series expansion of e^{-x^2} was wrong. She is also working on a Ruby version of the book!
- Matt Crawford sent in a whole patch file full of corrections!
- Chi-Yu Li pointed out a typo and an error in one of the code examples.
- Doan Thanh Nam corrected an example in Chapter 3.
- Stijn Debrouwere found a math typo.

تُركت هذه الصفحة بيضاء عن عمد

المحتويات

الفصل 1

مسار البرنامج

إن الغرض من هذا الكتاب هو تعليمك كيفية التفكير كعالم كمبيوتر. أنا أحب طريقة تفكير علماء الكمبيوتر لأنهم يجمعون بين أفضل المزايا الموجودة لدى الرياضيين، المهندسين، وعلماء الطبيعة. مثل علماء الرياضيات، يستخدم علماء الكمبيوتر اللغات الاصطلاحية والرموز للتعبير عن الأفكار (وخصوصاً الحسابات). مثل المهندسين، يصممون أشياء و يجمعون المكونات لتؤلف نظماً. مثل علماء الطبيعة، يلاحظون سلوك الأنظمة المعقدة، يبنون فرضيات، ويختبرون التوقعات.

إن المهارة الأكثر أهمية لأي عالم كمبيوتر هي **حل المشكلات.** وبهذا أنا أعني القدرة على صياغة المشاكل، والتفكير الإبداعي في الحلول، ثم التعبير عن الحل بوضوح ودقة. كما اتضح فإن تعلم البرمجة هو فرصة ممتازة للتدرب على مهارات حل المشاكل.

على أحد الأصعدة، ستتعلم كيف تبرمج، وهي مهارة مفيدة بحد ذاتها. وعلى صعيد آخر، ستستخدم البرمجة كوسيلة توصلك إلى نهاية. عندما نتقدم في هذا الكتاب، ستتوضح لك تلك النهاية أكثر فأكثر.

1.1 ما هي لغة البرمجة؟

إن لغة البرمجة التي ستتعلمها هي Java، وهي حديثة نسبياً (أطلقت شركة Sun الإصدار الأول في أيار/مايو، 1995). Java هي مثال عن **اللغات عالية المستوى (High-level languages)**؛ من اللغات عالية المستوى الأخرى التي قد تكون سمعت بها Python، C أو C++، وPerl.

كما هو واضح من الاسم لغات عالية المستوى؛ هناك أيضاً **لغات منخفضة المستوى (low-level languages)**، والتي تدعى أحياناً بلغات التجميع (assembly language) أو لغة الآلة (machine language). يمكن القول أن الحواسيب تستطيع تنفيذ البرامج المكتوبة في اللغات منخفضة المستوى فقط. وبالتالي، يجب ترجمة البرامج المكتوبة باللغات عالية المستوى قبل أن نتمكن من تشغيلها. هذه الترجمة تأخذ بعض الوقت، وهي خسارة بسيطة تسببها اللغات عالية المستوى.

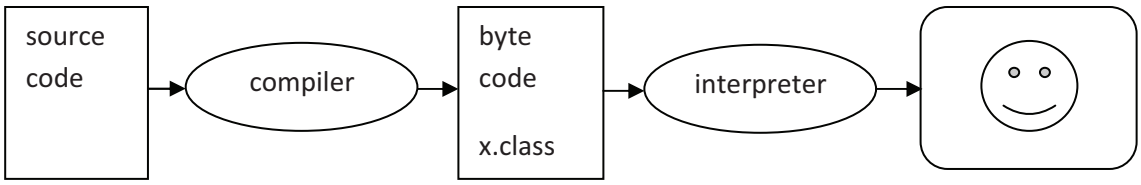
لكن المزايا كثيرة. أولاً، من الأسهل بكثير أن نكتب برنامجاً بلغة عالية المستوى؛ وبكلمة "أسهل" أنا أعني أن البرنامج يأخذ وقتاً أقصر لكتابته، ومن الأسهل والأسرع قراءته، ومن المرجح أكثر أن يكون صحيحاً. ثانياً، البرامج المكتوبة باللغات عالية المستوى **محمولة (portable)**، بمعنى أنها تستطيع أن تعمل على أنواع مختلفة من الحواسيب بمجرد عمل تعديلات بسيطة أو بدون أي تعديلات. البرامج المكتوبة باللغة منخفضة المستوى يمكنها أن تعمل على نوع واحد فقط من الحواسيب، ويجب إعادة كتابتها قبل أن نتمكن من تشغيلها على جهاز آخر.

تبعاً لهذه المزايا، كل البرامج تقريباً مكتوبة بلغات عالية المستوى. أما اللغات المنخفضة المستوى فيتم استخدامها في بعض التطبيقات الخاصة القليلة فقط.

هناك طريقتان لترجمة برنامج؛ **التفسير (Interpreting)** و**التجميع (Compiling).**
المفسر هو برنامج يقرأ البرنامج عالي المستوى وينفذ المكتوب. في الواقع، يترجم المفسر البرنامج سطرا بعد آخر، أي أنه يقرأ السطور وينفذ الأوامر.

المجمِّع (أو المترجم Compiler) هو برنامج يقرأ برنامج عالي المستوى ويترجمه كله دفعة واحدة، قبل تنفيذ أية تعليمة. غالباً ما نترجم البرنامج كخطوة مستقلة، وبعد ذلك ننفذ الشفرة المترجمة (compiled code) لاحقاً. في هذه الحالة، يطلق على البرنامج عالي المستوى اسم **الشفرة المصدرية (source code)**، ويدعى البرنامج المترجم **بالشفرة الهدف (object code)** أو **الملف التنفيذي (executable).**

لغة Java غير تقليدية لأنها لغة مجمّعة ومفسرة بنفس الوقت. بدلاً من ترجمة برامج Java إلى لغة الآلة، يولد مجمع Java شفرة Java byte. شفرة بايت سهلة (وسريعة) عند التفسير، مثل لغة الآلة، لكنها محمولة أيضاً، مثل لغة عالية المستوى. وهكذا، يمكننا تجميع برنامج Java على جهاز واحد، ثم ننقل شفرة بايت إلى جهاز آخر عبر شبكة مثلاً، ثم نفسر شفرة بايت على الجهاز الثاني. هذه القدرة هي إحدى مزايا Java التي تتفوق بها على العديد من اللغات عالية المستوى الأخرى.



يقرأ المجمع الشفرة المصدرية...
... ويولد شفرة Java byte.
يقرأ مفسر Java شفرة byte...
... وتظهر النتيجة على الشاشة.

على الرغم من أن هذه العملية تبدو معقدة، فالخبر الطيب هو أن معظم بيئات البرمجة (أحياناً تدعى بيئات التطوير)، تجري هذه الخطوات تلقائياً بدلاً منك. عادة ما تكتب أنت البرنامج ثم تطلب أمراً واحداً لترجمته وتنفيذه. من جهة أخرى، من المفيد أن تعلم بماهية الخطوات التي تجري وراء الستار، لكي تتمكن من معرفة سبب المشكلة في حال وقوع أي خطأ.

1.2 ما هو البرنامج؟

البرنامج هو مجموعة من التعليمات المتتابعة التي تحدد كيفية تنفيذ عملية حسابية[1]. العملية الحسابية قد تكون رياضية، مثل حل جملة معادلات أو إيجاد جذور كثير حدود، لكن قد تكون أيضاً معالجة رمزية، مثل البحث عن نص واستبداله في مستند أو (غريب بما يكفي) تجميع برنامج آخر.

التعليمات (statements) تبدو مختلفة بين لغات البرمجة المختلفة، لكن توجد بعض الدوال الأساسية التي يمكن لجميع لغات البرمجة تنفيذها تقريباً:

الإدخال (input): الحصول على المعطيات من لوحة المفاتيح، ملف، أو جهاز آخر.

الإخراج (output): عرض البيانات على الشاشة، أو إرسالها إلى ملف أو جهاز آخر.

الحساب (math) : تنفيذ العمليات الحسابية الأساسية مثل الجمع والضرب.

الاختبار (testing): التحقق من شروط معينة وتنفيذ التعليمات المناسبة لكل حالة.

التكرار (repetition): تنفيذ عمل ما بصورة متكررة، عادة مع وجود تغيير.

[1] هذا التعريف لا ينطبق على جميع لغات البرمجة؛ انظر http://en.wikipedia.org/wiki/Declarative_programming.

هذا هو كل شيء تقريباً. أي برنامج استعملته من قبل، مهما كان معقداً، بني من دوال تبدو مثل هذه. وهكذا، يمكن وصف البرمجة بأنها عملية تجزئة للمهام الكبيرة والمعقدة، إلى مهام جزئية أصغر وأصغر حتى نصل إلى مهمة جزئية بسيطة بما يكفي ليتم تنفيذها بواحدة فقط من هذه الدوال البسيطة.

1.3 ما هو التنقيح؟

البرمجة هي عملية معقدة، ولأنها تتم بوساطة البشر، فهي غالباً ما تؤدي إلى أخطاء. لأسباب غريبة، تدعى الأخطاء البرمجية **bugs (حشرات)** وعملية ملاحقتها وتصحيحها تدعى **debugging.**
هناك عدة أنواع مختلفة من الأخطاء التي يمكن أن تحدث في برنامج، ومن المفيد أن نميز بينها في سبيل القبض عليها أسرع.

1.3.1 الأخطاء النحوية

لا يمكن للمجمع أن يترجم البرنامج إلا إذا كان صحيحاً نحوياً؛ وبخلاف ذلك، تفشل عملية الترجمة ولن تكون قادراً على تشغيل برنامجك. **النحو (Syntax)** يشير إلى بنية البرامج و قواعد كتابة تلك البنية.

مثلاً، في اللغة الإنكليزية، يجب أن تبدأ الجملة بحرف كبير وأن تنتهي بنقطة. بالنسبة لمعظم القراء، بعض الأخطاء النحوية ليست بمشكلة ذات اعتبار، لهذا السبب تجدنا نقرأ شعر e e cummings بدون أن نطلق رسائل أخطاء.

المترجمات ليست بذات القدر من التسامح. إذا وجد خطأ نحوي واحد في أي مكان من برنامجك، سيطبع المترجم رسالة خطأ ويتوقف، ولن تتمكن من تشغيل برنامجك.

ولنعقد الموضوع أكثر، فإن القواعد النحوية في Java أكثر منها في اللغة الإنكليزية، ورسائل الخطأ التي تتلقاها من المترجم ليست مفيدة على الأغلب. خلال الأسابيع الأولى من عملك في البرمجة، غالباً ما ستقضي الكثير من الوقت تبحث عن الأخطاء النحوية. على الرغم من ذلك، فإنك سترتكب أخطاءً أقل وستجدها أسرع عندما تكتسب المزيد من الخبرة.

1.3.2 أخطاء وقت التشغيل

النوع الثاني من الأخطاء هي الأخطاء عند التشغيل، وقد تمت تسميتها كذلك لأنها لا تظهر قبل تشغيل البرنامج. في Java، تظهر أخطاء وقت التشغيل عندما يشغل المفسر شفرة byte ويحدث خطأ ما.

الخبر الجيد الآن هو أن Java لغة **آمنة** نوعاً ما، ما يعني أن المجمع يقبض على العديد من الأخطاء. لذلك فإن أخطاء وقت التشغيل نادرة، وخاصة بالنسبة للبرامج البسيطة.

في Java، تدعى أخطاء وقت التشغيل **بالاستثناءات (exceptions)**، وفي معظم بيئات البرمجة تظهر بشكل نوافذ أو مربعات حوار تحتوي على معلومات عما حدث وما الذي كان البرنامج يفعله عند حدوث الخطأ. هذه المعلومات مفيدة عند تنقيح البرنامج من الأخطاء.

1.3.3 الأخطاء المنطقية

النوع الثالث من الأخطاء هو **الخطأ المنطقي (logic error)** أو **الدلالة الخاطئة (wrong semantic)**. إذا وجد خطأ منطقي في برنامجك، فستتم ترجمته وتشغيله بنجاح، بمعنى أن الكمبيوتر لن يعطي أي رسائل خطأ، لكنه لن ينفذ المطلوب كما يجب. بل سينفذ شيئاً آخر. سينفذ البرنامج ما طلبته منه حرفياً.

البرنامج الذي كتبته لم يكن البرنامج الذي ترغب فيه. معنى البرنامج (أو ما يدل عليه) خاطئ. معرفة مكان وقوع الخطأ المنطقي قد يتطلب براعة، ذلك لأنها تتطلب منك العمل بالمقلوب تنظر إلى مخرجات البرنامج وتحاول استنتاج ما يفعله.

1.3.4 تصحيح الأخطاء بالتجريب

أحد أهم المهارات التي يجب عليك اكتسابها أثناء العمل مع هذا الكتاب هو تصحيح الأخطاء (debugging). وبالرغم من أنه محبط، فهو من أكثر أجزاء البرمجة تحدياً، ويتطلب ذكاء ومهارة.

أحياناً يكون اكتشاف الأخطاء مثل عمل التحري. تجد الأدلة ويجب عليك أن تحزر العمليات والأحداث التي أدت إلى النتائج التي تراها.

تصحيح الأخطاء أيضاً يشبه العلوم التجريبية. بمجرد أن تظن أنك عرفت ما هو الخطأ تعدل برنامجك وتجرب ثانية. إذا كان حدسك صحيحاً، عندئذ ستتمكن من التنبؤ بنتيجة التعديل، وتقترب خطوة من البرنامج الصحيح المطلوب. إذا كان حدسك خاطئاً، عليك أن تجد حلاً آخر. وكما يقول (شرلوك هولمز)، "بمجرد أن تستبعد المستحيل، فإن ما يبقى، مهما كان بعيد الاحتمال، لا بد أن يكون الحقيقة". (من *The Sign of Four* للكاتب A. Conan Doyle)

بالنسبة لبعض الناس، البرمجة وتصحيح الأخطاء هما نفس الشيء. وذلك لأن البرمجة هي عملية تصحيح أو تعديل متدرج للبرنامج حتى يفعل المطلوب منه. الفكرة هي أن تبدأ مع برنامج يعمل قادر على تنفيذ شيء ما، ثم عمل تعديلات صغيرة، وتصحيحها أولاً بأول، حتى يبقى برنامجك عاملاً طوال الوقت.

مثلاً، نظام التشغيل لينوكس الذي يحتوي على آلاف الأسطر البرمجية، بدأ كبرنامج بسيط استعمله (لينوز تورفالدز) ليستكشف رقاقة إنتل 80386. وعلى حد قول (لاري غرينفيلد)، "أحد أولى مشاريع لينوز كان برنامجاً يبدل بين طباعة AAAA و BBBB. ثم تطور هذا إلى لينوكس" (من *The Linux Users' Guide* Beta Version 1).

في الفصول اللاحقة سأبدي المزيد من الاقتراحات عن تصحيح الأخطاء وغيرها من الخبرات البرمجية.

1.4 اللغات الطبيعية واللغات الاصطلاحية

اللغات الطبيعية (Natural languages) هي اللغات التي يتكلم بها الناس، مثل الإنكليزية، العربية و الفرنسية. لم يقم البشر بتصميم هذه اللغات (على الرغم من أنهم يحاولون فرض بعض القواعد عليها)؛ بل تطورت هذه اللغات طبيعياً.

اللغات الاصطلاحية (Formal languages) هي لغات صممها الناس لتطبيقات مخصصة. مثلاً، الرموز التي يستخدمها الرياضيون لغة اصطلاحية وهي ملائم بصورة خاصة لتدوين العلاقات بين الأرقام والحروف. الكيميائيين يستعملون لغة اصطلاحية للتعبير عن البنية الكيميائية للجزيئات. وأهم شيء:

لغات البرمجة هي لغات اصطلاحية صممت للتعبير عن الحسابات (computations).

كما ذكرت سابقاً، اللغات الاصطلاحية لها قواعد نحوية صارمة. مثلاً، 3 + 3 = 6 عبارة رياضية صحيحة نحوياً، لكن 3 = +6$ ليست كذلك. أيضاً، H_2O اسم مركب كيميائي صحيح نحوياً، لكن $_2Zz$ ليس كذلك.

القواعد النحوية تأتي في صفتين غالبتين، قواعد الرموز وقواعد البنية. الرموز هي العناصر الرئيسية للغة، مثل الكلمات والأرقام والعناصر الكيميائية. أحد الأخطاء الموجودة في العبارة 3=+6$ هو أن $ ليس رمزاً رياضياً (على الأقل هذا ما أعرفه). بشكل مشابه، $_2Zz$ ليس صحيحاً لعدم وجود عنصر يرمز له بالاختصار *Zz*.

النمط الآخر من الأخطاء النحوية مرتبط ببنية العلاقة؛ وهي الطريقة التي ترتب وفقها الرموز. بنية العبارة 3=+6$ غير صحيحة، لأنك لا تستطيع كتابة علامة الجمع بعد علامة المساواة مباشرة. أيضاً، الأرقام في الصيغ الجزيئية تأتي بعد اسم العنصر، وليس قبله.

عندما تقرأ جملة بالإنكليزية أو عبارة بلغة اصطلاحية، عليك معرفة بنية الجملة (برغم أن ذلك يحدث بدون وعي مع اللغات الطبيعية). هذه العملية تدعى **إعراب (parsing)**.

ومع أن اللغات الطبيعية والاصطلاحية تملك العديد من المقومات المشتركة —رموز، بنية، نحو، ومعاني— إلا أن الاختلافات عديدة.

الغموض: اللغات الطبيعية ملئا بالالتباسات، ويتعامل الناس معها اعتماداً على السياق ومعلومات أخرى. اللغات الاصطلاحية مصممة لتكون خالية من الالتباس تماماً أو تقريباً، ما يعني أن العبارة الواحدة لها معنى واحد، بغض النظر عن مكان وجودها في النص.

الحشو: في سبيل تفسير الغموض في اللغات الطبيعية والتقليل من سوء التفاهم، تحوي اللغات الطبيعية على الكثير من الإسهاب. كنتيجة لذلك فهي غالباً ما تكون محشوة. اللغات الاصطلاحية تكون أقل حشواً وأكثر إيجازاً.

الحرفية: اللغات الطبيعية ملئا بالمصطلحات والمجازات. اللغات الاصطلاحية تعني تماماً ما تقول.

إن الأشخاص الذين كبروا وهم يتحدثون لغة طبيعية (كل الناس) غالباً ما يواجهون صعوبة عن التحول إلى لغة اصطلاحية. بشكل أو بآخر فإن الاختلافات بين اللغات الطبيعية والاصطلاحية تشبه الاختلافات بين الشعر والنثر، ومع ذلك:

الشعر: يستعمل الكلمات لصوتها ورنتها كما لمعناها، والقصيدة بالكامل تعطي أثراً أو رد فعل عاطفي. الغموض ليس شائعاً فحسب بل هو متعمد.

النثر: المعنى الحرفي للكلمات أهم والبنية تشارك في إضافة المعاني. النثر أكثر قابلية للتحليل من الشعر، لكنه يظل غامضاً أحياناً.

البرامج: معنى برنامج الكمبيوتر واضح ولا يقبل الالتباس، ويتم فهمه التام بتحليل رموزه وبنيته.

إليك بعض الاقتراحات لقراءة البرامج (وأية لغات اصطلاحية أخرى). أولاً، تذكر أن اللغات الاصطلاحية كثيفة أكثر من اللغات الطبيعية، لذا فهي تأخذ وقتاً أطول لفهمها عادةً. أيضاً، البنية شديدة الأهمية، لذلك فإن القراءة من اليسار إلى اليمين ومن الأعلى إلى الأسفل ليست فكرة جيدة دائماً. بدلاً من ذلك، تعلم تتبع البرنامج و"إعرابه" في رأسك، متعرفاً الرموز ومفسراً البنية. أخيراً، تذكر أن التفاصيل تحدث فرقاً. الأشياء الصغيرة مثل الأخطاء الإملائية والترقيم السيئ، والتي تستطيع الإفلات بها في اللغات الطبيعية، يمكنها أن تحدث فروقاً هائلة في اللغات الاصطلاحية.

1.5 البرنامج الأول

تقليدياً البرنامج الأول الذي يكتبه الناس بأي لغة برمجة جديدة يسمى "Hello, World" — (مرحباً، بالعالم). لأن كل ما يفعله هو طباعة الكلمات "Hello, World". في لغة Java، سيبدو هذا البرنامج كما يلي:

```
class Hello {

  // main: generate some simple output

  public static void main (String[] args) {
    System.out.println ("Hello, world.");
  }
}
```

هذا البرنامج يحتوي على العديد من المقومات التي يصعب شرحها للمبتدئين، لكنه يظهر "معاينة" لمواضيع سنراها بالتفصيل فيما بعد.

كل البرامج تتألف من تعاريف أصناف (class definitions)، التي تكتب بالشكل:

```
class CLASSNAME {
```

```
  public static void main(String[] args) {
    STATEMENTS
  }
}
```

هنا يشير CLASSNAME إلى اسم كيفي تختاره أنت. اسم الصنف في هذا المثال هو Hello.

main هي عملية (method)، أي مجموعة من العمليات لها اسم. الاسم main اسم خاص؛ يشير إلى المكان في البرنامج حيث يبدأ التنفيذ. عندما نشغل البرنامج، يبدأ بتنفيذ التعليمة الأولى في main ويتابع، بالترتيب، حتى يصل إلى التعليمة الأخيرة، عندها يتم الخروج من البرنامج.

يمكن أن تحوي main أي عدد من التعليمات، لكن المثال السابق يحوي تعليمة واحدة فقط. وهي **تعليمة طباعة (print statement)**، فهي تطبع رسالة على الشاشة. من المحبّر أن "اطبع" تعني أحياناً "اعرض شيئاً على الشاشة"، وأحياناً تعني "أرسل شيئاً إلى الطابعة". لن أتحدث كثيراً عن إرسال الأشياء إلى الطابعة في هذا الكتاب؛ كل طباعتنا ستتم على الشاشة. تنتهي تعليمة الطباعة بفاصلة منقوطة (;).

System.out.println هي عملية توفرها إحدى مكتبات Java. **المكتبة (library)** هي مجموعة من تعاريف الأصناف والعمليات.

تستعمل Java الأقواس المنحنية ({ و }) لتجميع الأشياء مع بعضها. الأقواس الخارجية (في السطرين 1 و8) تحوي تعريف الصنف، والأقواس الداخلية تحوي تعريف main.

السطر الثالث يبدأ بـ //، هذا يعني أن هذا السطر هو **تعليق (comment)**. التعليق هو بعض النص المكتوب باللغة الإنكليزية يمكنك وضعه بين سطور البرنامج، غالباً لشرح ما يفعله البرنامج. عندما يرى المترجم //، فسيتجاهل كل شيء من بعده حتى نهاية السطر.

1.6 المصطلحات

حل المشكلات: عملية وضع المشكلة ضمن صيغة، إيجاد حل، والتعبير عنه.

اللغة عالية المستوى: لغة برمجة مثل Java مصممة لتكون سهلة القراءة والكتابة على البشر.

اللغة منخفضة المستوى: لغة برمجة مصممة لتكون سهلة التنفيذ على الحاسب. أيضاً تدعى "لغة الآلة" أو "لغة التجميع".

اللغة الاصطلاحية: أية لغة صممها الناس لأغراض خاصة، مثل التعبير عن الأفكار الرياضية أو البرامج الحاسوبية. جميع لغات البرمجة هي لغات اصطلاحية.

اللغة الطبيعية: لغة يتحدث بها الناس وتطورت بصورة طبيعية.

قابلية النقل: صفة خاصة ببرنامج يمكن تشغيله على أكثر من نوع واحد من الحواسيب.

التفسير: تنفيذ برنامج مكتوب بلغة عالية المستوى عن طريق ترجمته سطراً تلو الآخر.

التجميع (الترجمة): ترجمة برنامج مكتوب بلغة عالية المستوى إلى لغة منخفضة المستوى، دفعة واحدة، تحضيراً لتنفيذه لاحقاً.

الشفرة المصدرية: برنامج مكتوب بلغة عالية المستوى، قبل أن تتم ترجمته.

الشفرة الهدف: خرج المترجم، بعد ترجمة البرنامج.

الملف التنفيذي: اسم آخر للشفرة الهدف الجاهزة للتنفيذ.

شفرة بايت: نوع خاص من الشفرة الهدف يستعمل في برامج Java. شفرة بايت مشابهة للغات منخفضة المستوى، لكنها محمولة، مثل اللغات العالية المستوى.

التعليمة: جزء من البرنامج يحدد حسبة (computation).

تعليمة الطباعة: تعليمة تسبب عرض خرج على الشاشة.

تعليق: جزء من البرنامج يحوي معلومات عن البرنامج، لكنه لا يملك أي تأثير عند تشغيله.

عملية: مجموعة مسماة من التعليمات.

مكتبة: مجموعة من تعاريف الأصناف والعمليات.

خطأ برمجي: خطأ في البرنامج.

النحو: بنية البرنامج.

الدلالة: معنى البرنامج.

إعراب: فحص البرنامج وتحليل البنية النحوية.

خطأ نحوي: خطأ في البرنامج يجعل إعرابه مستحيلاً (بالتالي تستحيل ترجمته).

استثناء: خطأ في البرنامج يجعله يفشل أثناء تشغيله. يدعى أيضاً بخطأ التشغيل.

خطأ منطقي: خطأ في البرنامج يجعله ينفذ شيئاً يختلف عما يقصده المبرمج.

تصحيح الأخطاء: عملية اكتشاف وإزالة لأي خطأ من الأنواع الثلاثة.

problem-solving: The process of formulating a problem, finding a solution, and expressing the solution.

high-level language: A programming language like Java that is designed to be easy for humans to read and write.

low-level language: A programming language that is designed to be easy for a computer to execute. Also called “machine language” or “assembly language.”

formal language: Any of the languages people have designed for specific purposes, like representing mathematical ideas or computer programs. All programming languages are formal languages.

natural language: Any of the languages people speak that have evolved naturally.

portability: A property of a program that can run on more than one kind of computer.

interpret: To execute a program in a high-level language by translating it one line at a time.

compile: To translate a program in a high-level language into a low-level language, all at once, in preparation for later execution.

source code: A program in a high-level language, before being compiled.

object code: The output of the compiler, after translating the program.

executable: Another name for object code that is ready to be executed.

byte code: A special kind of object code used for Java programs. Byte code is similar to a low-level language, but it is portable, like a high-level language.

statement: A part of a program that specifies a computation.

print statement: A statement that causes output to be displayed on the screen.

comment: A part of a program that contains information about the program, but that has no effect when the program runs.

method: A named collection of statements.

library: A collection of class and method definitions.

bug: An error in a program.

syntax: The structure of a program.

semantics: The meaning of a program.

parse: To examine a program and analyze the syntactic structure.

syntax error: An error in a program that makes it impossible to parse (and therefore impossible to compile).

exception: An error in a program that makes it fail at run-time. Also called run-time error.

logic error: An error in a program that makes it do something other than what the programmer intended.

debugging: The process of finding and removing any of the three kinds of errors.

1.7 تمرينات

تمرين 1.1

عند علماء الكمبيوتر عادة مزعجة وهي استعمال كلمات إنكليزية شائعة للتعبير عن شيء مختلف عن معناها الأصلي. مثلاً، في الإنكليزية، كلمتي statement وcomment هما نفس الشيء تقريباً، لكن عندما نتكلم عن برنامج، فهما مختلفان كلياً.

إن المصطلحات في نهاية كل فصل موجودة للتركيز على الكلمات والعبارات التي تملك معنى خاصاً في علوم الحاسوب. عندما ترى كلمة مألوفة، لا تفترض انك تعرف معناها!

a. في لغة الحواسيب، ما هو الفرق بين statement وcomment؟
b. ماذا يعني أن نقول أن هذا البرنامج محمول؟
c. ما هو الملف التنفيذي؟

تمرين 1.2 قبل أن تفعل أي شيء آخر، اكتشف كيف يتم ترجمة وتشغيل برنامج Java في بيئة البرمجة لديك. بعض بيئات البرمجة توفر أمثلة برمجية مشابهة للمثال في القسم 1.5.

a. اكتب برنامج "Hello, World!"، ثم ترجمه وشغله.
b. أضف تعليمة طباعة ثانية تطبع جملة ثانية بعد "Hello, Word!". شيء ظريف مثل، "How are you?". ترجم وشغل البرنامج مرة ثانية.
c. أضف تعليقاً إلى البرنامج (في أي مكان) وترجمه مرة أخرى. شغل البرنامج ثانية. من المفروض ألا يؤثر التعليق الجديد على تنفيذ البرنامج.

قد يبدو التمرين تافهاً، لكنه نقطة الانطلاق للعديد من البرامج التي سنعمل معها. لتتمكن من تصحيح الأخطاء بثقة، يجب أن تكون واثقاً من بيئة البرمجة التي تستخدمها. في بعض البيئات، من السهل أن تضيع البرنامج الذي تشغله، وقد تجد نفسك تحاول تصحيح أخطاء برنامج ما في حين أنك تشغل برنامج آخر عن طريق الخطأ. إضافة (وتغيير) تعليمات الطباعة هي طريقة بسيطة للتأكد من اتصال البرنامج الذي تنظر إليه مع الخرج الذي يظهر عند تشغيل البرنامج.

تمرين 1.3 من الجيد ارتكاب أكبر عدد ممكن من الأخطاء يمكن أن تفكر فيها، حتى ترى رسائل الخطأ التي يولدها المجمّع. أحياناً يخبرك المجمع ما هو الخطأ بدقة، وكل ما عليك هو إصلاحه. لكن أحياناً تكون رسائل الخطأ مضللة. سيتولد لديك إحساس يخبرك متى تثق بالمجمع ومتى سيتوجب عليك اكتشاف الأخطاء بنفسك.

a. أزل أحد أقواس الفتح المنحنية.
b. أزل أحد أقواس الإغلاق المنحنية.
c. بدلاً من main، اكتب mian.
d. أزل الكلمة static.
e. أزل الكلمة public.
f. أزل الكلمة System.
g. استبدل println بـPrintln.
h. استبدل println بـprint. هذا الخطأ خدعة صغيرة لأنه خطأ منطقي، وليس خطأ نحوياً. التعليمة System.out.print مشروعة، لكنها قد تفعل ما تتوقعه وقد لا تفعل.
i. احذف أحد الأقواس. أضف واحداً زائداً.

تُركت هذه الصفحة بيضاء عن عمد

الفصل 2

المتغيرات وأنواع البيانات

2.1 المزيد من الطباعة

يمكنك أن تضع العدد الذي ترغب فيه من التعليمات في main. مثلاً، لطباعة أكثر من سطر واحد من النص:

```
class Hello {

   // Generates some simple output.

   public static void main(String[] args) {
      System.out.println("Hello, world.");    // print one line
      System.out.println("How are you?");     // print another
   }
}
```

كما هو واضح من هذا المثال، يمكنك وضع التعليقات في نهاية السطر، كما يجوز أن تترك سطراً كاملاً لها.

العبارات التي تظهر بين علامات الاقتباس تدعى **سلاسل محرفية (strings)**، لأنها مكونة من تتابع (تسلسل) للمحارف. يمكن أن تحتوي السلاسل المحرفية على أي توليفة من الحروف، والأرقام، وعلامات الترقيم ومحارف خاصة أخرى.

println هي اختصار ل"print line"، لأنها تضيف بعد كل سطر تطبعه محرفاً خاصاً، يدعى محرف **السطر الجديد (new line)**، يحرك مؤشر الطباعة إلى السطر التالي من شاشة العرض. في المرة التالية التي سيتم استدعاء println فيها، سيظهر النص على السطر التالي.

لإظهار الخرج من عدة تعليمات طباعة على سطر واحد، استخدم print:

```
class Hello {

   // Generates some simple output.

   public static void main(String[] args) {
      System.out.print("Goodbye, ");
      System.out.println("cruel world!");
   }
}
```

في هذه الحالة سيظهر الخرج على سطر واحد كما في Goodbye, cruel world!.

لاحظ وجود الفراغ بين الكلمة "Goodbye," وعلامة الاقتباس الثانية. هذا الفراغ سيظهر في الخرج، فهو يؤثر إذن في سلوك البرنامج.

الفراغات التي تظهر خارج علامات الاقتباس لا تؤثر في عمل البرنامج عادة. مثلاً، كان بإمكاني كتابته كما يلي:

```
class Hello {
public static void main(String[] args) {
System.out.print("Goodbye, ");
```

```
System.out.println("cruel world!");
}
}
```

هذا البرنامج سيترجم وينفذ تماماً كالأصلي. كتابة البرنامج على عدة أسطر لا تؤثر على عمل البرنامج أيضاً، لذا فمن الممكن أن أكتبه كما يلي:

```
class Hello { public static void main(String[] args) {
System.out.print("Goodbye, "); System.out.println("cruel world!");}}
```

هذا البرنامج سيعمل أيضاً، إلا أنك قد لاحظت على الأغلب أن البرنامج أصبح أصعب للقراءة في كل مرة. الأسطر الجديدة والفراغات مفيدة لترتيب البرنامج بصرياً، مما يجعل البرنامج أسهل للقراءة والأخطاء النحوية (syntax errors) أسهل للتصحيح.

2.2 المتغيرات

أحد أقوى المزايا لأي لغة برمجة هي القدرة على معالجة **المتغيرات (variables)**. المتغير هو منطقة ذات اسم تخزّن **قيمة (value)**. القيم هي الأشياء التي يمكن طباعتها، وتخزينها و(كما سنرى لاحقاً) تطبيق العمليات عليها. السلاسل المحرفية التي كنا نطبعها ("Hello, World."، "Goodbye, "، الخ) هي قيم.

لتخزين قيمة، عليك إنشاء متغير. بما أن القيم التي نريد تخزينها هي سلاسل محرفية، سنصرح عن المتغير على أنه سلسلة محرفية (String):

```
String fred;
```

هذا النوع من التعليمات يدعى **تصريح (declaration)**، لأنها تصرح أن نوع المتغير المدعو fred هو String. لكل متغير نوع يحدد نوع القيم التي يمكنه تخزينها. مثلاً، النوع int يمكنه تخزين الأعداد الصحيحة، والنوع String يمكنه تخزين السلاسل المحرفية.

بعض الأنواع تبدأ بحرف كبير وبعضها الآخر يبدأ بحرف صغير. سنعرف معنى هذا التمييز لاحقاً، لكن الآن عليك الانتباه له حتى تفهمه بشكل صحيح. لا يوجد نوع مثل Int أو string، وسيعترض المجمع لو حاولت اختراع مثل هذه الأنواع.

لإنشاء متغير من النوع الصحيح، التعليمة هي

```
int bob;
```

حيث bob هو الاسم الكيفي الذي تختاره للمتغير. بصورة عامة، سترغب باختيار أسماء للمتغيرات بحيث تشير إلى دور المتغير في البرنامج. مثلاً، عندما ترى التصريحات التالية عن المتغيرات:

```
String firstName;
String lastName;
int hour, minute;
```

ستتمكن على الأغلب من تخمين ماهية القيم التي ستخزن فيها. هذا المثال يوضح أيضاً كيفية التصريح عن عدة متغيرات من نفس النوع في تعليمة واحدة: كلاً من hour و minute هو عدد صحيح (من النوع int).

2.3 الإسناد

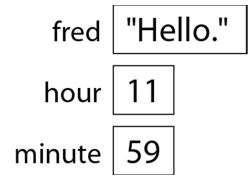
بعد أن أنشأنا بعض المتغيرات، سنقوم بتخزين بعض القيم فيها. يمكننا عمل ذلك باستخدام **تعليمة الإسناد (assignment statement)**.

```
fred = "Hello.";    // give bob the value "Hello."
hour = 11;          // assign the value 11 to hour
minute = 59;        // set minute to 59
```

هذا المثال يبين ثلاث تعليمات إسناد، والتعليقات تظهر ثلاثة أساليب يستخدمها الناس أحياناً عندما يقرؤون تعليمات الإسناد. المفردات قد تكون مربكة هنا، لكن الفكرة واضحة :

- عندما تصرح عن متغير، أنت تنشئ منطقة تخزينية لها اسم.
- عندما تطبق تعليمة الإسناد على متغير، فأنت تعطيه قيمة.

من الطرق الشائعة للتعبير عن المتغيرات على الورق رسم صندوق وكتابة اسم المتغير خارجه وقيمة المتغير بالداخل. هذا المخطط يبين أثر تعليمات الإسناد الثلاث:



كقاعدة عامة، يجب أن يكون نوع المتغير من نفس نوع القيمة التي تسندها إليه. مثلاً، لا يمكنك تخزين سلسلة محرفية في المتغير minute أو عدداً صحيحاً في fred.

من ناحية أخرى، هذه القاعدة قد تكون مصدراً للإرباك أحياناً، بسبب وجود العديد من الطرق التي تسمح لك بتحويل القيم من نوع لآخر، وأحياناً تحول Java الأشياء تلقائياً. لكن الآن عليك فقط أن تتذكر القاعدة العامة بأن المتغيرات والقيم يجب أن تكون من نفس النوع، وسنتحدث عن الحالات الخاصة لاحقاً.

من مصادر الإرباك الأخرى هو أن بعض السلاسل المحرفية تبدو مثل الأرقام، لكنها ليست كذلك. مثلاً، يمكن أن يخزن fred السلسلة المحرفية "123"، المكونة من المحارف 1 و 2 و 3، لكنها ليست مثل العدد 123.

```
fred = "123";       // legal
fred = 123;         // not legal
```

2.4 طباعة المتغيرات

يمكنك طباعة قيمة متغير باستخدام println أو print:

```
class Hello {
  public static void main(String[] args) {
    String firstLine;
    firstLine = "Hello, again!";
    System.out.println(firstLine);
  }
}
```

هذا البرنامج ينشئ متغيراً باسم firstLine، ويسند له القيمة "Hello, Again!" ثم يطبع تلك القيمة.

عندما نتحدث عن "طباعة متغير"، فنحن نعني طباعة قيمة المتغير. أما لطباعة اسم المتغير، فعليك أن تضعه بين علامتي اقتباس. مثلاً:

```
System.out.println("firstLine");
```

مثلا، يمكنك كتابة

```
String firstLine;
firstLine = "Hello, again!";
System.out.print("The value of firstLine is ");
System.out.println(firstLine);
```

سيكون خرج هذا البرنامج هو

```
The value of firstLine is Hello, again!
```

بنية تعليمة طباعة متغير هي نفسها بغض النظر عن نوع المتغير.

```
int hour, minute;
hour = 11;
minute = 59;
System.out.print("The current time is ");
System.out.print(hour);
System.out.print(":");
System.out.print(minute);
System.out.println(".");
```

خرج هذا البرنامج هو `The current time is 11:59.`

تنـويه: لوضع عدة قيم على نفس السطر، من الشائع استخدام عدة تعليمات `print`. لكن عليك أن تتذكر وضع `println` في النهاية. في العديد من بيئات البرمجة، يتم تخزين خرج تعليمات `print` بدون عرضه على الشاشة حتى استدعاء `println`، وعندها يظهر السطر كله دفعة واحدة. إذا أغفلت `println`، فقد ينتهي برنامجك بدون طباعة الخرج المخزن!

2.5 الكلمات المفتاحية

قبل عدة أقسام، أخبرتك أنك تستطيع اختيار أي اسم لمتغيراتك، لكنّ ذلك ليس صحيحاً تماماً. هناك كلمات معينة محجوزة في Java لأن المترجم يستخدمها لإعراب (parse) بنية برنامجك، وإذا استخدمتها كأسماء للمتغيرات، سيرتبك المترجم. هذه الكلمات، تدعى **الكلمات المفتاحية (keywords)**، وهي تتضمن الكلمات public، class ، void ، int ، وغيرها الكثير.

القائمة الكاملة للكلمات المفتاحية موجودة على http://download.oracle.com/javase/tutorial/java/nutsandbolts/_keywords.html، هذا الموقع، المقدم من شركة أوراكل، يحتوي على توثيق Java الذي أعتمد إليه في هذا الكتاب.

بدلاً من حفظ القائمة، سأقترح أن تستفيد من ميزة تقدمها معظم بيئات البرمجة: تلوين الشفرة. بينما تكتب البرنامج، الأجزاء المختلفة منه ستظهر في ألوان مختلفة. مثلاً، الكلمات المفتاحية قد تكون باللون الأزرق، السلاسل المحرفية بالأحمر، وبقية الشفرة بالأسود. إذا كتبت اسماً لمتغير وتغير لونه إلى الأزرق، فانتبه! قد تنتج أفعال غريبة عن المترجم.

2.6 العوامل

العوامل (operators) هي رموز خاصة تستعمل للتعبير عن الحسابات البسيطة مثل الجمع والضرب. معظم العوامل في Java تفعل ما تتوقعه منها تماماً لأنها رموز رياضية شائعة. مثلاً، عامل جمع عددين هو +. عامل الطرح هو - ، وللضرب *، وللقسمة /.

```
1+1          hour-1          hour*60 + minute          minute/60
```

يمكن للعبارات الحسابية أن تحتوي على أسماء متغيرات وأرقام. يتم استبدال المتغيرات بقيمها قبل إجراء الحساب.

يتم تنفيذ الجمع، الطرح والضرب تماماً كما هو متوقع، لكنك قد تفاجأ بعملية القسمة. مثلاً، البرنامج التالي:

```
int hour, minute;
hour = 11;
minute = 59;
System.out.print("Number of minutes since midnight: ");
System.out.println(hour*60 + minute);
System.out.print("Fraction of the hour that has passed: ");
System.out.println(minute/60);
```

سيولد النتيجة التالية:

```
Number of minutes since midnight: 719
Fraction of the hour that has passed: 0
```

السطر الأول يوافق ما توقعناه، لكن السطر الثاني غريب. قيمة المتغير `minute` هي 59، و59 مقسومة على 60 يعطي 0.98333، وليس 0. إن سبب هذا التناقض هو أن Java قامت بتنفيذ **قسمة صحيحة (integer division)**.

عندما يكون كلا **المعاملين (operands)** صحيحين (المعامل هو الشيء الذي تعمل عليه العوامل)، يجب أن تكون النتيجة عدداً صحيحاً أيضاً، وبحسب تعريفها فإن القسمة الصحيحة دائماً تقرب النتائج إلى الأدنى، حتى في الحالات التي يكون الناتج فيها قريباً جداً من العدد الصحيح التالي. من الحلول البديلة في هذه الحالة هو حساب النسبة المئوية بدلاً من العدد الكسري:

```
System.out.println("Percentage of the hour that has passed: ");
System.out.println(minute*100/60);
```

النتيجة هي:

```
Percentage of the hour that has passed: 98
```

مرة أخرى، يتم تقريب النتيجة إلى العدد الأدنى، لكن على الأقل فالإجابة صحيحة تقريباً هذه المرة. للحصول على إجابة أكثر دقة، يمكننا استعمال نوع آخر من المتغيرات، يدعى متغير النقطة العائمة (floating-point)، الذي يستطيع تخزين القيم الكسرية. سنشرح ذلك في الفصل التالي.

2.7 أولوية العمليات الحسابية

عندما يظهر أكثر من عامل في تعبير حسابي فإن ترتيب الحسابات يعتمد على قواعد **الأولوية (precedence)**. الشرح الكامل للأولوية يمكن أن يصبح معقداً، لكن أن نقول كبداية أن:

- يتم تنفيذ الضرب والقسمة قبل الجمع والطرح. لذا فإن 1-3*2 تعطي 5، وليس 4، كما أن 2/3-1 تعطي 1-، وليس 1 (تذكر أن القسمة الصحيحة 2/3 تعطي 0).

- إذا كان للعوامل نفس الأفضلية فيتم تنفيذها بالترتيب، من اليسار إلى اليمين. ففي العبارة الحسابية `minute*100/60`، يتم تنفيذ عملية الضرب أولاً، ويعطي `5900/60`، والذي بدوره يعطي 98. لو أن تنفيذ العملية الحسابية جرى من اليمين لليسار، ستكون النتيجة `1*59` والذي هو 59، وهو جواب خاطئ.
- في أي وقت ترغب فيه بتجاوز قواعد الأولوية (أو أنك لم تكن واثقاً من تلك القواعد) يمكنك استعمال الأقواس. يتم تنفيذ العمليات ضمن الأقواس أولاً، لهذا فإن `(1-3) * 2` تعطي 4. يمكنك استعمال الأقواس أيضاً لجعل العبارات الحسابية أسهل للقراءة، كما في `(minute * 100) / 60`، مع أنها لا تغير النتيجة.

2.8 عوامل السلاسل المحرفية

بصورة عامة، لا يمكنك إجراء عمليات حسابية على السلاسل المحرفية، حتى لو بدت السلسلة كرقم. التعليمات التالية غير مشروعة (إذا علمنا أن `bob` من النوع `String`)

```
bob - 1          "Hello"/123          bob * "Hello"
```

بالمناسبة، هل يمكنك معرفة فيما إذا كان `bob` من نوع `int` أو `String` بالنظر إلى هذه العبارات؟ لا، الطريقة الوحيدة لمعرفة نوع متغير تكون بالنظر إلى المكان الذي تم التصريح عنه فيه.

مما يثير الاهتمام، أن العامل + يعمل على السلاسل المحرفية، لكنه قد لا يقوم بما هو متوقع. بالنسبة للسلاسل المحرفية، يمثل العامل + عملية **ربط السلاسل (concatenation)**، أي دمج المعاملين بربطهما معاً، نهاية الأول إلى بداية الثاني. لذا فإن "Hello, " + "world." ستعطي السلسلة "Hello, world." و"ism" + bob تضيف اللاحقة *ism* إلى آخر الكلمة المخزنة في bob، وهو ما قد يكون مفيداً عند تسمية الأنواع الجديدة من التعصبات (bigotry).

2.9 التركيب

حتى الآن تعرفنا على عناصر لغة البرمجة — المتغيرات، العبارات الحسابية، والتعليمات — بشكل معزول، بدون التحدث عن كيفية تجميعهم.

أحد أفضل مزايا لغات البرمجة هو قدرتها على أخذ قوالب بناء صغيرة و**تركيبها (compose)**. مثلاً، نحن نعرف كيفية ضرب الأعداد ونعرف كيف نخرج القيم؛ هذا يؤدي إلى أننا نستطيع جمع الاثنين في تعليمة واحدة:

```
System.out.println(17 * 3);
```

أي تعبير، يحوي أرقاماً، سلاسل محرفية، ومتغيرات، يمكن استعماله ضمن تعليمة الطباعة. وقد شاهدنا مثالاً على ذلك من قبل:

```
System.out.println(hour*60 + minute);
```

يمكنك أيضاً وضع عبارات كيفية على يمين تعليمة الإسناد:

```
int percentage;
percentage = (minute * 100) / 60;
```

هذه القدرة قد لا تكون مدهشة الآن، لكننا سنرى أمثلة أخرى يمكننا التركيب فيها من التعبير عن الحسابات المعقدة بشكل مختصر ومحكم.

تنويه: يجب أن يكون الطرف الأيسر من تعليمة الإسناد اسم متغير، وليس عبارة رياضية. ذلك لأن الطرف الأيسر يشير إلى المنطقة التخزينية التي ستخزن فيها النتائج. العبارات الرياضية لا تمثل مناطق تخزينية، بل تمثل قيماً فقط. لذا فالتعليمة التالية غير صحيحة: `minute+1 = hour;`.

2.10 المصطلحات

المتغير: منطقة لتخزين القيم ولها اسم. كل متغير له نوع، يتم التصريح عنه عند إنشاء المتغير.

القيمة: سلسلة محرفية، أو رقم، (أو أي شيء آخر مما سنتعرف عليه لاحقاً) يمكن تخزينه في متغير. كل قيمة تنتمي لنوع معين.

النوع: مجموعة من القيم. نوع المتغير يحدد القيم التي يمكن تخزينها فيه. الأنواع التي شاهدناها حتى الآن هي الأعداد الصحيحة (int في Java) والمحارف (char في Java).

الكلمة المفتاحية: كلمة محجوزة يستخدمها المترجم لإعراب البرامج. لا يمكنك استخدام الكلمات المفتاحية مثل `public`، `class`، و`void` كأسماء للمتغيرات.

التصريح: تعليمة تقوم بإنشاء متغير جديد وتحدد نوعه.

الإسناد: تعليمة تعطي قيمة إلى متغير.

العبارة الرياضية: مجموعة من المتغيرات، والعوامل والقيم التي تمثل قيمة ناتجة وحيدة. للعبارات أنواع أيضاً، تحدد بعواملها ومعاملاتها.

العامل: رمز خاص يمثل عملية حسابية، مثل الجمع أو الضرب أو ربط السلاسل.

المعامل: أحد القيم التي يعمل عليها العامل.

الأولوية: الترتيب الذي يتم تنفيذ العمليات الحسابية وفقه.

ربط السلاسل: جمع المعاملين طرف الأول إلى طرف الثاني.

التركيب: القدرة على تجميع التعابير البسيطة والتعليمات في تعليمات وتعابير مركبة في سبيل تمثيل الحسابات المعقدة بشكل مختصر.

variable: A named storage location for values. All variables have a type, which is declared when the variable is created.

value: A number or string (or other thing to be named later) that can be stored in a variable. Every value belongs to a type.

type: A set of values. The type of a variable determines which values can be stored there. The types we have seen are integers (int in Java) and strings (String in Java).

keyword: A reserved word used by the compiler to parse programs. You cannot use keywords, like public, class and void as variable names.

declaration: A statement that creates a new variable and determines its type.

assignment: A statement that assigns a value to a variable.

expression: A combination of variables, operators and values that represents a single value. Expressions also have types, as determined by their operators and operands.

operator: A symbol that represents a computation like addition, multiplication or string concatenation.

operand: One of the values on which an operator operates.

precedence: The order in which operations are evaluated.

concatenate: To join two operands end-to-end.

composition: The ability to combine simple expressions and statements into compound statements and expressions to represent complex computations concisely.

2.11 تمرينات

تمرين 2.1 قد تستمتع في هذا التمرين: ابحث عن شريك والعب معه "Stump the Chump" – "اخدع المغفل".

ابدأ مع برنامج تتم ترجمته وتشغيله بشكل صحيح. يبتعد أحد اللاعبين بينما يقوم الآخر بعمل خطأ في البرنامج. بعدها يحاول اللاعب الآخر العثور على ذلك الخطأ وإصلاحه. تحصل على نقطتين إذا عثرت على الخطأ بدون تجميع البرنامج، ونقطة واحدة أغذا عثرت عليه بالاستعانة بالمترجم، ويحصل خصمك على نقطة إذا لم تعثر عليه.

ملاحظة: من فضلك لا تحذف "l" من كلمة "public". فذلك ليس ممتعاً كما تعتقد.

تمرين 2.2

a. أنشئ برنامجاً جديداً باسم Date.java. انسخ أو اكتب برنامجاً يشبه "Hello, world" وتأكد من تجميعه وتشغيله.

b. بالاستعانة بالمثال في القسم 2.4، اكتب برنامجاً ينشئ متغيرات جديدة تدعى day، date، month، وyear. سيحتوي day على اليوم من الأسبوع وdate على اليوم من الشهر. ما هو نوع كل متغير؟ أسند قيماً لهذه المتغيرات بحيث تمثل تاريخ اليوم.

c. اطبع قيمة كل متغير على سطر لوحده. هذه خطوة انتقالية للتأكد من أن كل شيء على ما يرام حتى الآن.

d. عدل البرنامج بحيث يطبع التاريخ في التنسيق الأمريكي القياسي: Saturday, August 13, 2011.

e. عدل البرنامج ثانية حتى يصبح الخرج كما يلي:

```
American format:
Saturday, August 13, 2011
European format:
Saturday 13 August, 2011
```

إن الغرض من هذا التمرين هو استخدام ربط السلاسل لعرض القيم ذات الأنواع المختلفة (int وString)، والتدرب على تطوير البرامج تدريجياً بإضافة عدد قليل من التعليمات في كل مرة.

تمرين 2.3

a. أنشئ برنامجاً جديداً باسم Time.java. من الآن فصاعداً لن أذكرك أن تبدأ مع برنامج صغير صحيح، لكن عليك أن تفعل ذلك.

b. بالاستعانة بالمثال من القسم 2.6، أنشئ متغيرات باسم hour، minute، وsecond، وأسند لها قيماً تمثل الوقت الحالي تقريباً. استعمل نظام 24 ساعة؛ بحيث تكون قيمة hour التي تمثل 2 بعد الظهر هي 14.

c. اصنع برنامجاً يحسب ويطبع عدد الثواني المنقضية منذ منتصف الليل.

d. اصنع برنامجاً يحسب ويطبع عدد الثواني المتبقية للنهار.

e. اصنع برنامجاً يحسب النسبة المئوية التي انقضت من النهار.

f. غير قيم hour، وminute وsecond لتعاكس قيم الوقت الحالي (هذه هي المدة التي أعتقد أنها قد انقضت الآن)، وتحقق من أن البرنامج يعمل مع مختلف القيم.

الغرض من هذا التمرين هو استخدام بعض العمليات الحسابية، والبدء بالتفكير بعناصر مركبة مثل الوقت الذي يمثل بعدة قيم. أيضاً، قد تواجهك مشاكل عند حساب النسبة المئوية باستخدام الأعداد الصحيحة، وهو الدافع لدراسة الأعداد العشرية في الفصل التالي.

مساعدة: قد تحتاج لاستعمال متغيرات إضافية لتخزين بعض القيم مؤقتاً خلال عملية الحساب. إن مثل هذه المتغيرات التي نستخدمها في العمليات الحسابية لكن لا تتم طباعتها أبداً، تدعى أحياناً بالمتغيرات الوسيطة أو المؤقتة.

الفصل 3

العمليات

3.1 النقطة العائمة

في الفصل السابق واجهنا بعض المشاكل عند التعامل مع الأعداد غير الصحيحة. لقد تفادينا المشكلة بحساب النسبة المئوية بدلاً من الكسر، لكن الحل الأعم هو استعمال الأعداد العشرية، التي تستطيع التعبير عن الكسور كما تعبر عن الأعداد الصحيحة. في لغة Java، تدعى الأعداد العشرية double.

يمكنك إنشاء متغيرات عشرية وإسناد القيم لها باستخدام نفس التعليمات التي استخدمناها للأنواع الأخرى. مثلاً:

```
double pi;
pi = 3.14159;
```

كما يمكن أيضاً التصريح عن متغير وإسناد قيمة له في الوقت نفسه:

```
int x = 1;
String empty = "";
double pi = 3.14159;
```

في الواقع، هذا الأسلوب شائع جداً. أحياناً ندعو التصريح والإسناد المدمجين **بالتهيئة (initialization)**.

على الرغم من أن الأعداد العشرية مفيدة، فهي غالباً ما تكون مصدراً للارتباك بسبب ظهور ما يشبه التداخل بين الأعداد الصحيحة والأعداد العشرية. مثلاً، إذا كانت لديك القيمة 1، فهل هي عدد صحيح، أم عدد عشري، أم الاثنين معاً؟

إذا تحدثنا بدقة، فإن Java تفرّق بين القيمة الصحيحة 1 وبين القيمة العشرية 1.0، حتى لو بدا أنهما نفس العدد، فهما يختلفان بالنوع، وعلى وجه الدقة، لا يسمح لك بالقيام بعمليات إسناد بين النوعين. مثلاً، ما يلي ليس مسموحاً:

```
int x = 1.1;
```

لأن المتغير على اليسار int والقيمة على اليمين double. لكنه من السهل نسيان هذه القاعدة، خاصة بوجود الأماكن التي تقوم فيها Java بالتحويل بين الأنواع تلقائياً. مثلاً:

```
double y = 1;
```

من المفترض تقنياً ألا تكون هذه التعليمة مشروعة، لكن Java تسمح بها عن طريق التحويل التلقائي من int إلى double. هذا التساهل مريح، لكن يمكن له أن يسبب المشاكل؛ مثلاً:

```
double y = 1 / 3;
```

قد تتوقع أن يعطى المتغير y القيمة 0.333333، وهي قيمة عشرية مشروعة، لكنه في الواقع سيعطى القيمة 0.0. السبب هو أن العبارة على اليمين هي نسبة بين عددين صحيحين، لذا تقوم Java بإجراء قسمة صحيحة، والتي تنتج القيمة الصحيحة 0. ثم يتم تحويلها إلى قيمة عشرية، الناتج هو 0.0.

إحدى الطرق لحل هذه المشكلة (بعد أن تكتشف أن هذه هي المشكلة) هو جعل الطرف الأيمن عبارة عشرية:

```
double y = 1.0 / 3.0;
```

هذا سيعطي y القيمة 0.333333، كما هو متوقع.

كل العمليات التي شاهدناها حتى الآن – الجمع، الطرح، الضرب، والقسمة – تعمل أيضاً مع القيم العشرية، مع أنك قد تكون مهتماً بمعرفة أن آلية العمل الضمنية مختلفة كلياً. في الواقع، معظم المعالجات تملك عتاداً خاصاً لتنفيذ العمليات العشرية فقط.

3.2 التحويل من double إلى int

كما ذكرت من قبل، Java تحول الأعداد الصحيحة إلى عشرية تلقائياً إذا دعت الحاجة، بسبب عدم فقدان أية معلومات أثناء التحويل. من جهة أخرى، التحويل من double إلى int يتطلب تقريبا. لا تنفذ Java هذه العملية تلقائياً، وذلك لتضمن أنك، بصفتك المبرمج، مدركاً لضياع القسم العشري من العدد.

أبسط طريقة لتحول قيمة عشرية إلى صحيحة هي استعمال **قولبة الأنماط (typecast)**. تم تسمية قولبة الأنماط هكذا لأنها تسمح لك بأخذ قيمة من نوع معين و"قولبتها" في نوع آخر (بمعنى إعادة تشكيلها أو تحويلها).

لسوء الحظ، فإن البنية المستعملة في قولبة الأنماط بشعة: تضع اسم النوع في أقواس وتستعمله كعامل. مثلاً،

```
int x = (int) Math.PI;
```

العامل (int) له أثر تحويل ما يليه إلى int، لذا فإن x سيحصل على القيمة 3.

قولبة الأنماط لها أولوية على العمليات الحسابية، ففي المثال التالي، سيتم تحويل قيمة PI إلى قيمة صحيحة أولاً، وسيكون الناتج 60.0 وليس 62.

```
double x = (int) Math.PI * 20.0;
```

التحويل إلى قيمة صحيحة دائماً يقرب العدد إلى الأدنى، حتى لو كان الجزء العشري 0.99999999. هاتان الخاصيتان (الأولوية والتقريب) قد تجعلان من قولبة الأنماط سبباً للخطأ.

3.3 التوابع الرياضية

في الرياضيات، شاهدت توابعاً مثل sin وlog على الأغلب، وتعلمت حساب قيمة عبارات مثل $\sin(\pi/2)$ و$\log(1/x)$. أولاً، تحسب قيمة العبارة بين قوسين، التي تدعى **متحول (argument)** التابع. مثلاً، $\pi/2$ تقريباً 1.571، و$1/x$ هو 0.1 (بفرض أن x يساوي 10).

بعد ذلك ستتمكن من حساب قيمة التابع نفسه، إما بالبحث عنه في جدول أو بتنفيذ حسابات متنوعة. إن جيب 1.571 هو 1، و لوغاريتم 0.1 هو 1- (على اعتبار أن log تشير إلى اللوغاريتم ذا الأساس 10).

هذه العملية يمكن تطبيقها بشكل متكرر لحساب عبارات أكثر تعقيداً مثل $\log(1/\sin(\pi/2))$. أولاً نحسب قيمة متغير التابع الداخلي، ثم نحسب قيمة ذلك التابع، ونتابع على هذا المنوال.

توفر Java مجموعة من التوابع تنفذ معظم العمليات الرياضية الشائعة. هذه التوابع تدعى **عمليات (methods)**.

تستدعى العمليات الرياضية باستخدام نحو مشابه لأوامر الطباعة print التي سبق أن شاهدناها:

```
double root = Math.sqrt(17.0);
double angle = 1.5;
double height = Math.sin(angle);
```

يعطي المثال الأول المتغير root الجذر التربيعي للعدد 17. يحسب المثال الثاني جيب قيمة المتغير angle، وهي 1.5. تفترض Java أن القيم التي تستخدمها مع sin ومع التوابع المثلثية الأخرى (cos, tan) مقدرة بالراديان. للتحويل من الدرجات إلى الراديان، يمكنك التقسيم على 360 والضرب بπ2. للسهولة، توفر Java قيمة π بصورة جاهزة:

```
double degrees = 90;
double angle = degrees * 2 * Math.PI / 360.0;
```

لاحظ أن PI مكتوبة بأحرف كبيرة بالكامل. لن تتعرف Java على Pi، pi أو pie.

من العمليات المفيدة في صنف Math هي round، التي تدور القيم العشرية إلى أقرب قيمة صحيحة وترجع int.

```
int x = Math.round(Math.PI * 20.0);
```

في هذه الحالة يتم تنفيذ الضرب أولاً، قبل استدعاء العملية round. الناتج هو 63 (قرِّبت القيمة 62.8319 إلى القيمة الأعلى).

3.4 التركيب

كما في التوابع الرياضية -الحقيقية-، يمكن **تركيب (compose)** العمليات في Java، بمعنى أنك تستطيع استعمال عبارة ما كجزء من عبارة أخرى. مثلاً، يمكنك استخدام أية عبارة حسابية كمتحول لعملية ما:

```
double x = Math.cos(angle + Math.PI/2);
```

هذه التعليمة تأخذ قيمة Math.PI، وتقسمها على اثنين ثم تضيف الناتج إلى قيمة المتغير angle. بعد ذلك يمرر المجموع كمتحول لعملية cos. (لاحظ أن PI هو اسم متغير، وليس عملية، لذلك لا نمرر له أية متحولات، ولا حتى المتحول الفارغ ( )).

يمكنك أيضاً أخذ نتيجة إحدى العمليات وتمريرها كمتحول لعملية أخرى:

```
double x = Math.exp(Math.log (10.0.));
```

في Java، تابع log يستعمل الأساس e دائماً، لذلك فإن هذه التعليمة تحسب اللوغاريتم ذا الأساس e للعدد 10 ثم ترفع e إلى تلك القوة. ويتم إسناد النتيجة إلى x؛ أتمنى أن تعرف ما هي.

3.5 إضافة عمليات جديدة

حتى الآن كنا نستعمل العمليات المبنية مسبقاً في مكتبات Java، لكن يمكننا أيضاً أن نضيف عمليات جديدة. في الواقع، لقد رأينا سابقاً تعريف إحدى العمليات: main. العملية المسماة main مميزة لأنها تشير إلى مكان بدء تنفيذ البرنامج، لكن البنية المستخدمة في main هي نفس المستخدمة في العمليات الأخرى:

```
public static void NAME( LIST OF PARAMETERS ) {
STATEMENTS
}
```

يمكنك استخدام أي اسم تريده لعمليتك، عدا أنك لا تستطيع تسميتها main أو أي كلمة أخرى محجوزة في Java.
اصطلاحاً، تبدأ أسماء العمليات في Java بأحرف صغيرة وتستخدم "حروف الجمل – camel caps"، وهو اسم طريف يعني كتابة الحرف الأول من الكلمات المتلاصقة بشكله الكبير، كما نقوم عندما jamWordsTogetherLikeThis.

قائمة المعاملات تحدد ماهية المعلومات، في حال وجودها، التي يتوجب عليك توفيرها حتى تستعمل (أو **تستدعي (invoke))** التابع الجديد.

المعامل الوحيد للعملية main هو String []args، الذي يشير إلى أنه يجب على كل من يستدعي main أن يوفر مصفوفة من السلاسل المحرفية (سنصل إلى المصفوفات في الفصل 10). العمليتان اللتان سنكتبهما في البداية ليس لهما أية معاملات، لذا فإن بنيتها ستبدو مثل هذه:

```
public static void newLine() {
   System.out.println("");
}
```

هذه العملية اسمها newLine، والأقواس الفارغة تبين أنها لا تأخذ أية معاملات. وهي تتضمن تعليمة واحدة فقط، تقوم بطباعة سلسلة فارغة (empty String)، أشرنا إليها بالعلامتين "". إن طباعة سلسلة محرفية لا تحوي أية حروف قد يبدو غير مفيد على الإطلاق، إلا إذا تذكرت أن تعليمة println تنقل مؤشر الطباعة إلى السطر التالي بعد الانتهاء من الطباعة، إذن فهذه التعليمة تؤدي إلى تجاوز سطر واحد ونقل مؤشر الطباعة إلى السطر التالي.

داخل main يمكننا استدعاء هذه العملية الجديدة باستعمال طريقة مشابهة للطريقة المستخدمة عند استدعاء أوامر Java الجاهزة:

```
public static void main(String[] args) {
   System.out.println("First line.");
   newLine();
   System.out.println("Second line.");
}
```

إن خرج البرنامج هو

```
First line.

Second line.
```

لاحظ المسافة الإضافية بين السطرين. ماذا لو أردنا المزيد من المساحة بين السطرين؟ يمكننا استدعاء العملية نفسها عدة مرات:

```
public static void main(String[] args) {
   System.out.println("First line.");
   newLine();
   newLine();
   newLine();
   System.out.println("Second line.");
}
```

أو يمكننا كتابة عملية جديدة، اسمها threeLine، تقوم بطباعة ثلاثة أسطر:

```
public static void threeLine() {
newLine(); newLine(); newLine();
}
public static void main(String[] args) {
```

```
System.out.println("First line.");
threeLine();
System.out.println("Second line.");
}
```

يجب أن تنتبه لعدة أشياء في هذا البرنامج:

- يمكنك استدعاء نفس العملية عدة مرات. في الواقع، هذا الشيء شائع جداً ومفيد.
- يمكنك كتابة عملية تستدعي عملية أخرى. في هذه الحالة، `main` تستدعي `threeLine` و `threeLine` تستدعي `newLine`. أيضاً فإن هذا الشيء شائع ومفيد.
- في `threeLine` كتبت التعليمات الثلاثة كلها على سطر واحد، وذلك صحيح لغوياً (تذكر أن المسافات والأسطر الجديدة لا تغير معنى البرنامج عادة). من جهة أخرى، من الأفضل وضع كل تعليمة في سطر لوحدها في العادة، وذلك لتجعل برنامجك أسهل للقراءة. أنا أحياناً أتجاوز هذه القاعدة.

حتى الآن، قد لا يبدو سبب كل هذا العناء في كتابة هذه العمليات الجديدة واضحاً. في الواقع، هناك العديد من الأسباب، لكن هذا المثال يشرح اثنين فقط:

1. إنشاء عملية جديدة يعطيك القدرة على إعطاء اسم لمجموعة من التعليمات. يمكن للعمليات أن تبسط البرنامج بإخفاء حسابات معقدة وراء أمر واحد، وباستعمال الكلمات الإنكليزية بدلاً من الشفرة الغامضة. أيهما أوضح، `newLine` أو `System.out.println("")` ؟
2. إنشاء عملية جديدة يمكن جعل البرنامج أصغر بالتخلص من الشفرة المكررة. مثلاً، كيف يمكنك طباعة تسعة أسطر جديدة متعاقبة؟ يمكنك استدعاء `threeLine` ثلاثة مرات.

في القسم 7.7 سنعود إلى هذا السؤال وسأسرد المزيد من فوائد تقسيم البرامج إلى عمليات.

3.6 الأصناف والعمليات

بجمع كل أجزاء الشفرة من القسم السابق، سيبدو التعريف الكامل للصنف كالتالي:

```
class NewLine {

   public static void newLine() {
      System.out.println("");
   }

   public static void threeLine() {
      newLine(); newLine(); newLine();
   }

   public static void main (Sttring[] args) {
      System.out.println ("First line.");
      threeLine();
      System.out.println ("Second line.");
   }
}
```

يبين السطر الأول أن هذا تعريف صنف للصنف الجديد المسمى NewLine. الصنف هو مجموعة من العمليات المترابطة. في هذه الحالة، يحتوي الصنف المسمى NewLine على ثلاث عمليات، اسمها newLine، وthreeLine، وmain.

الصنف الآخر الذي شاهدناه كان الصنف Math. وهو يحتوي على العمليات المسماة sqrt، وsin، وغيرها الكثير. عندما نستدعي تابعاً رياضياً، علينا تحديد اسم الصنف (Math) واسم التابع. لهذا السبب فإن بنية التعليمة المستخدمة لاستدعاء العمليات الجاهزة تختلف قليلاً عن البنية المستخدمة لاستدعاء العمليات التي نكتبها:

```
Math.pow(2.0, 10.0);
newLine();
```

التعليمة الأولى تستدعي العملية pow من الصنف Math (التي ترفع المتغير الأول إلى قوة المتغير الثاني). التعليمة الثانية تستدعي العملية newLine، والتي تفترض Java وجودها (وهي موجودة فعلاً) في صنف NewLine، الذي نقوم بكتابته.

إذا حاولنا استدعاء العملية من صنف خاطئ، سيعطي المجمّع خطأ. مثلاً، إذا كتبتَ:

```
pow(2.0, 10.0);
```

سيقول المجمع شيئاً مثل، "Can't find a method named pow in class NewLine."
" لا يمكن العثور على عملية باسم pow في الصنف NewLine". إذا رأيت هذه الرسالة، فلربما تساءلت عن سبب بحثه عن pow في تعريف صنفك. الآن أنت تعرف.

3.7 البرامج ذات العمليات المتعددة

عندما تنظر إلى تعريف صنف يحوي عمليات متعددة، فإن قراءته من الأعلى إلى الأسفل تبدو مغرية، لكن ذلك سيكون مشوشاً على الأغلب، لأنه ليس **ترتيب تنفيذ** البرنامج **(order of execution** of the program).

يبدأ التنفيذ دائماً عند التعليمة الأولى من main، بغض النظر عن موقعها من البرنامج (في حالتنا هذه فقد تعمدت وضعها في النهاية). يتم تنفيذ التعليمات واحدة تلو الأخرى، بالترتيب، حتى تصل إلى استدعاء لعملية. استدعاءات العمليات تشبه الانعطافات في مجرى التنفيذ. بدلاً من الذهاب إلى التعليمة التالية، تذهب إلى السطر الأول من العملية المستدعاة، تنفذ جميع التعليمات الموجودة هناك، ثم تعود وتتابع التنفيذ ثانية من المكان الذي انعطفت منه.

هذا يبدو بسيطاً بما يكفي، عدا أنك يجب أن تتذكر أن تلك العملية قادرة على استدعاء عملية أخرى. بالتالي، بينما نحن في وسط main، قد نضطر إلى الذهاب لتنفيذ التعليمات في threeLine. ولكن بينما نحن ننفذ تعليمات threeLine، ستتم مقاطعتنا ثلاث مرات للذهاب وتنفيذ newLine.

بدورها، تستدعي newLine العملية الجاهزة println، والتي تسبب انعطافاً أخراً أيضاً. لحسن الحظ، فإن Java ماهرة جداً في الحفاظ على مسارٍ يبين مكانها، بحيث تستطيع المتابعة من المكان الذي توقفت عنده في newLine بعد انتهاء تنفيذ println، ثم تعود إلى threeLine، وأخيراً تعود إلى main وهكذا يستطيع البرنامج أن ينتهي.

تقنياً، لن ينتهي البرنامج عند نهاية main. بدلاً من ذلك، سيتابع التنفيذ في المكان الذي تركه في البرنامج الذي قام باستدعاء main، وهو مفسر Java. يهتم مفسر Java بأشياء مثل حذف النوافذ والتنظيف العام، وبعد ذلك ينتهي البرنامج.

ما هو المغزى من هذه الحكاية المملة؟ عندما تقرأ برنامجاً، لا تقرأ من الأعلى إلى الأسفل. بدلاً من ذلك، تتبع مجرى التنفيذ.

3.8 المعاملات والمتحولات

بعض العمليات الجاهزة التي استخدمناها تتطلب **متحولات (arguments)**، وهي قيم تعطيها للعملية لتسمح لها بإتمام عملها. مثلاً، إذا أردت حساب جيب عدد ما، يجب عليك أن تحدد ما هو ذلك العدد. إذن، يأخذ sin قيمة من نوع double كمتحول. لطباعة سلسلة محرفية، عليك أن تعطي تلك السلسة إلى عملية الطباعة، ولهذا فإن println تأخذ String كمتحول.

بعض العمليات تأخذ أكثر من متحول، مثل pow، التي تأخذ متحولين من نوع double، الأساس والأس.

عندما تستخدم عملية، تزودها بالمتحولات. عندما تكتب عملية، تحدد قائمة المعاملات. **المعامل (parameter)** هو متغير يخزن متحول. تحدد قائمة المعاملات المتحولات المطلوبة للعملية.

مثلاً، تحدد printTwice معاملاً واحداً، s، من النوع String. لقد أسميته s ليبدو أنه من نوع String، لكن من الممكن أن أدعوه بأي اسم مسموح آخر.

```
public static void printTwice(String s) {
   System.out.println(s);
   System.out.println(s);
}
```

عندما نستدعي printTwice، علينا تقديم متحول واحد من النوع String.

```
   printTwice("Don't make me say this twice!");
```

عندما تستدعي عملية، يتم إسناد المتحولات التي أعطيتها للمعاملات. في هذا المثال، يسند المتحول "Don't make me say this twice!" إلى المعامل s. هذه العملية تدعى **تمرير المعاملات (parameter passing)** وذلك بسبب تمرير القيم من خارج العملية إلى داخلها.

يمكن أن نمرر أي نوع من العبارات كمتحول، فإذا كنا نملك متغيراً من نوع String، أمكننا تمريره كمتحول:

```
   String argument = "Never say never.";
   printTwice(argument);
```

يجب أن تكون القيمة التي توفرها كمتحول من نفس نوع معامل العملية التي تستدعي. مثلاً، لو جربت هذه التعليمة:

```
printTwice(17);
```

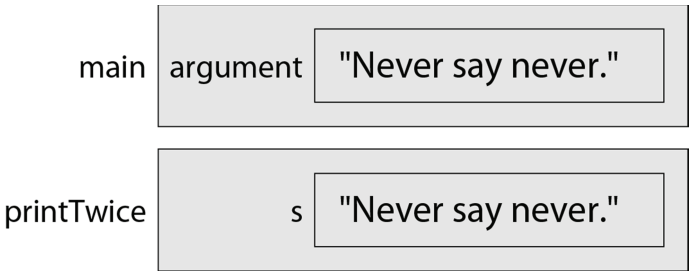
ستحصل على رسالة خطأ مثل "cannot find symbol"، "لا يمكن العثور على الشكل"، وهي غير مفيدة كثيراً. السبب هو أن Java تبحث عن عملية باسم printTwice يمكنها أن تأخذ عدداً صحيحاً كمتحول. نظراً لعدم وجود تلك العملية، فلن تتمكن Java من العثور على ذلك "الشكل".

تستطيع System.out.println أن تقبل أي نوع كمتحول. لكن هذه حالة خاصة؛ معظم العمليات ليست بهذه التوافقية.

3.9 المخططات الهرمية

توجد المعاملات والمتغيرات الأخرى داخل عملياتها فقط. ضمن حدود main، لا يوجد شيء اسمه s. إذا حاولت استخدامه، فسيشتكي المجمّع. أيضاً داخل printTwice لا يوجد شيء اسمه argument.

من إحدى الوسائل المستخدمة لتسجيل المكان الذي عرِّف فيه كل متغير هي **المخططات الهرمية (stack diagrams).** إن المخطط الهرمي للمثال السابق يبدو كما يلي:



لكل عملية يوجد صندوق فضي اللون يدعى **إطاراً (frame)** يحتوي على معاملات العملية ومتغيراتها. يظهر اسم العملية خارج الإطار. كالعادة، قيمة كل متغير ترسم داخل صندوق يَظهر اسم المتغير بجواره.

3.10 العمليات ذات المعاملات المتعددة

إن البنية المستخدمة في التصريح عن واستدعاء العمليات ذات المعاملات المتعددة تشكل مصدراً شائعاً للأخطاء. أولاً، تذكر أنه عليك تحديد نوع كل معامل. مثلاً

```
public static void printTime(int hour, int minute) {
   System.out.print(hour);
   System.out.print(":");
   System.out.println(minute);
}
```

قد يميل البعض لكتابة المعاملات هكذا: `int hour, minute`، لكن ذلك الشكل ممكن فقط في حالة التصريح عن متغيرات، وليس لقوائم المعاملات.

من مصادر التشويش الأخرى هو أنك لا تحتاج للتصريح عن نوع المتحولات عند استدعاء العملية. ما يلي خاطئ!

```
int hour = 11;
int minute = 59;
printTime(int hour, int minute);  //WRONG!
```

في هذه الحالة، تستطيع Java معرفة نوع `hour` و`minute` بالنظر إلى تصريحاتهم. ليس ضرورياً كما أنه ليس مشروعاً تضمين أنواعهم عند تمريرهم كمتحولات. الشكل الصحيح للتعليمة هو
`printTime (hour, minute);`.

تمرين 3.1 ارسم مخططاً هرمياً يبين حالة البرنامج عندما يستدعي `main` عملية `printTime` مع المتحولين 11 و59.

3.11 العمليات ذات النتائج

ربما تكون لاحظت الآن أن بعض العمليات التي نستخدمها، مثل عمليات `Math`، تعطي نتائجاً. بينما تقوم عمليات أخرى، مثل `println` و `newLine`، بتنفيذ أعمال معينة لكنها لا تعيد قيمة. هذا يطرح بعض الأسئلة:

- ماذا يحدث إذا استدعيت عملية ولم تفعل شيئاً بالنتيجة (مثلاً لم تقم بإسنادها إلى متغير أو تستعملها كجزء من عبارة أكبر)؟
- ماذا يحدث إذا استعملت عملية `print` كجزء من عبارة حسابية، مثل `System.out.println("boo!") + 7`؟
- هل نستطيع كتابة عمليات ترجع نتائجاً، أم أننا عالقون مع أشياء مثل `newLine` و`printTwice`؟

الإجابة على السؤال الثالث هي "نعم، يمكنك كتابة عمليات ترجع قيماً،" وسنقوم بذلك بعد فصلين. سأترك مهمة الإجابة على السؤالين الأوليين بالتجريب. في الواقع، في أي وقت يكون لديك سؤال عما هو مشروع أو ممنوع في Java، فإن سؤال المجمع طريقة جيدة لمعرفة ذلك.

3.12 المصطلحات

التهيئة: تعليمة تصرح عن متغير جديد وتسند له قيمة في نفس الوقت.

النقطة العائمة: (العدد العشري) نوع متغير (أو قيمة) يمكن أن يحتوي الأعداد الكسرية كما يحوي الأعداد الصحيحة. إن النوع الذي سنستخدمه لأعداد النقطة العائمة هو `double`.

الصنف: مجموعة مسماة من العمليات. حتى الآن، استخدمنا صنف `Math` وصنف `System`، كما كتبنا أصنافاً اسمها `Hello` و `NewLine`.

العملية: تعليمات متتابعة من العمليات لها اسم، تنفذ تابعاً مفيداً. العمليات قد تأخذ أو لا تأخذ معاملات، وقد تولد أو لا تولد نتيجة.

المعامل: معلومة تحتاجها العملية حتى تعمل. المعاملات هي متغيرات: فهي تحتوي على قيم ولها أنواع.

المتحول: قيمة توفرها عندما تقوم باستدعاء عملية. هذه القيمة يجب أن تكون من نفس نوع المعامل المقابل.

الإطار: بنية (تمثل بصندوق فضي في مخططات الحالة) تحوي معاملات العملية ومتغيراتها.

الاستدعاء: طلب تنفيذ العملية.

initialization: A statement that declares a new variable and assigns a value to it at the same time.

floating-point: A type of variable (or value) that can contain fractions as well as integers. The floating-point type we will use is double.

class: A named collection of methods. So far, we have used the `Math` class and the `System` class, and we have written classes named `Hello` and `NewLine`.

method: A named sequence of statements that performs some useful function. Methods may or may not take parameters, and may or may not produce a result.

parameter: A piece of information you provide in order to invoke a method. Parameters are variables: they contain values and have types.

argument: A value that you provide when you invoke a method. This value must have the same type as the corresponding parameter.

frame: A structure (represented by a gray box in stack diagrams) that contains a method's parameters and variables.

invoke: Cause a method to be executed.

3.13 تمرينات

تمرين 3.2

إن الهدف من هذا التمرين هو التدرب على قراءة الشفرة والتأكد من أنك تفهم مجرى التنفيذ لبرنامج متعدد العمليات.

a. ما هو خرج البرنامج التالي؟ كن دقيقاً بخصوص مواقع المسافات والأسطر الجديدة.
مساعدة: ابدأ بشرح ما تفعله ping و baffle بالكلمات عندما يتم استدعاؤها.

b. ارسم مخططاً هرمياً يبين حالة البرنامج عند استدعاء ping للمرة الأولى.

```
public static void zoop() {
  baffle();
  System.out.print("You wugga ");
  baffle();
}

public static void main(String[] args) {
  System.out.print("No, I ");
  zoop();
  System.out.print("I ");
  baffle();
}

public static void baffle() {
  System.out.print("wug");
  ping();
}

public static void ping() {
  System.out.println(".");
}
```

تمرين 3.3 الهدف من هذا التمرين هو التأكد من أنك تفهم كيفية كتابة واستدعاء عملية تأخذ معاملات.

a. اكتب السطر الأول من عملية باسم `zool` تأخذ ثلاثة معاملات: واحد `int` واثنين `String`.

b. اكتب السطر من الشفرة التي تستدعي `zool`، مرر كمتحولات القيمة 11، اسم أول حيوان أليف امتلكته، واسم الشارع الذي كبرت فيه.

تمرين 3.4 الهدف من هذا التمرين هو أخذ الشفرة من تمرين سابق وتغليفها بعملية تأخذ معاملات. عليك البدء مع حل صحيح للتمرين 2.2.

a. اكتب عملية مسماة `printAmerican` تأخذ اليوم، والشهر والسنة كمعاملات وتطبعهم بالتنسيق الأمريكي.

b. اختبر عمليتك باستدعائها من `main` وتمرير المتحولات المناسبة. يجب أن يكون الخرج كما يلي (عدا أن التاريخ قد يختلف):

```
Wednesday, Septembre 29, 1999
```

c. بعد أن تصحح أية أخطاء في `printAmerican`، اكتب عملية أخرى اسمها `printEuropean` تطبع التاريخ بالتنسيق الأوروبي.

تُركت هذه الصفحة بيضاء عن عمد

الفصل 4

التعليمات الشرطية والتعاود

4.1 عامل باقي القسمة

يعمل عامل باقي القسمة (modulus operator) على الأعداد الصحيحة (والعبارات التي تعيد قيماً صحيحة) ويعطي باقي قسمة المعامل الأول على المعامل الثاني. في Java، عامل باقي القسمة هو علامة النسبة المئوية، %. بنية التعليمة المستخدمة مع هذا العامل مطابقة تماماً لتلك المستخدمة مع العوامل الأخرى:

```
int quotient = 7 / 3;
int remainder = 7 % 3;
```

العامل الأول، عامل القسمة الصحيحة، يعطي الناتج 2. العامل الثاني، باقي القسمة، ينتج 1. بالتالي، فإن 7 تقسيم 3 يساوي 2 والباقي 1.

يتبين لنا أن عامل باقي القسمة مفيد بشكل مدهش. مثلاً، يمكنك استخدامه للتحقق من قواسم عدد ما: إذا كان x % y يساوي الصفر، عندئذ يكون x يقبل القسمة على y، أي أن y أحد قواسم العدد x.

أيضاً، يمكنك استخدام عامل باقي القسمة لاستخراج خانات عدد وفصلها. مثلاً، x % 10 يعطي الخانة اليمنى الأولى من العدد x (في نظام العد العشري). بشكل مشابه، x % 100 سيعطي الخانتين الأخيرتين من العدد x.

4.2 التنفيذ المشروط

حتى نتمكن من كتابة برامج مفيدة، سنحتاج دائماً تقريباً للقدرة على التحقق من شروط معينة وتغيير سلوك البرنامج وفقاً لذلك. تعطينا **التعليمات الشرطية (Conditional Statements)** هذه القدرة. أبسط شكل لهذ التعليمات هي تعليمة if:

```
if (x > 0) {
   System.out.println ("x is positive");
}
```

العبارة بين قوسين تدعى بالشرط. إذا كان محققاً، سيتم تنفيذ التعليمات داخل الأقواس المنحنية. إذا لم يتحقق الشرط، فلا يحدث شيء.

يمكن أن يحتوي الشرط على أي واحد من عوامل المقارنة، أحياناً تدعى **العوامل المنطقية (relational operators):**

```
x == y // x equals y
x != y // x is not equal to y
x > y  // x is greater than y
x < y  // x is less than y
x >= y // x is greater than or equal to y
x <= y // x is less than or equal to y
```

بالرغم من أن هذه العمليات مألوفة إليك على الأغلب، فإن الطريقة التي تكتب بها هذه العمليات مختلفة قليلاً عن الرموز الرياضية مثل =، ≠ و≤. من الأخطاء الشائعة استعمال علامة = مفردة بدلاً من اثنتين ==. تذكر أن = هو عامل الإسناد، و== هو عامل المقارنة. أيضاً، لا يوجد شيء يكتب هكذا =< أو هكذا =>.

يجب أن يكون طرفي العامل الشرطي من نوع واحد. يمكنك المقارنة فقط بين int وint أو بين double وdouble. لسوء الحظ، لا يمكن استخدام هذه العوامل للمقارنة بين السلاسل المحرفية أبداً! توجد طريقة أخرى للمقارنة بين Strings، لكننا لن نصل إليها قبل فصلين تاليين.

4.3 الإجراء البديل

الشكل الآخر للتنفيذ الشرطي هو الإجراء البديل (alternative execution)، الذي يحتوي على احتمالين، ويحدد الشرط أي منهما سيتم تنفيذه. شكل التعليمة يبدو كما يلي:

```
if (x%2 == 0) {
   System.out.println ("x is even");
} else {
   System.out.println ("x is odd");
}
```

إذا كان باقي قسمة x على 2 يساوي الصفر، نعلم عندئذ أن x عدد زوجي، وهذه الشفرة تطبع رسالة نتيجة ذلك الأثر. إذا كان الشرط غير محقق، سيتم تنفيذ تعليمة الطباعة الأخرى. بما أن الشرط لا بد أن يكون إما محققاً وإما غير محقق (true or false)، فسيتم تنفيذ أحد البديلين حتماً.

من جهة أخرى، إذا كنت تفكر بأنك قد تحتاج للتحقق من الأعداد الزوجية والفردية كثيراً، فقد ترغب بجمع هذه الشفرة في عملية، كما يلي:

```
public static void printParity (int x) {
  if (x%2 == 0) {
      System.out.println ("x is even");
  } else {
      System.out.println ("x is odd");
  }
}
```

الآن صرت تملك عملية باسم printParity تطبع رسالة مناسبة لأي عدد صحيح تعطيه لها. في main يمكنك استدعاء هذه العملية كما يلي:

```
printParity (17);
```

تذكر دائماً أنك لا تحتاج للتصريح عن أنواع المتحولات عند استدعاء عملية ما. Java استنتاج أنواعها. عليك مقاومة جاذبية كتابة أشياء مثل هذه:

```
int number = 17;
printParity (int number);        // WRONG!!!
```

4.4 التعليمات الشرطية المترابطة

ستحتاج أحياناً للتحقق من مجموعة من الشروط المترابطة واختيار واحد من عدة أفعال. إحدى الطرق المستعملة لعمل هذا هي **ربط (chain)** سلسلة من عدة ifs and elses:

```
if (x > 0) {
  System.out.println ("x is positive");
} else if (x < 0) {
  System.out.println ("x is negative");
} else {
  System.out.println ("x is zero");
}
```

يمكن أن تكون هذه السلاسل بالطول الذي تريده، إلا أنها سرعان ما تصبح صعبة القراءة إذا خرجت عن السيطرة. إحدى الوسائل المتبعة لجعل هذه السلاسل أسهل للقراءة هي استعمال الترتيب القياسي عند كتابتها، كما هو موضح بهذه الأمثلة. إذا أبقيت جميع التعليمات والأقواس المنحنية مرتبة، فستنخفض احتمالات ارتكابك للأخطاء النحوية وستتمكن من إيجادها أسرع في حال وجودها.

4.5 التعليمات الشرطية المتداخلة

بالإضافة إلى الربط، يمكن أيضاً شبك تعليمة شرطية ضمن تعليمة أخرى. كان بإمكاننا كتابة المثال السابق كما يلي:

```
if (x == 0) {
  System.out.println ("x is zero");
} else {
  if (x > 0) {
    System.out.println ("x is positive");
  } else {
    System.out.println ("x is negative");
  }
}
```

الآن يوجد تعليمة شرطية خارجية لها فرعين. الفرع الأول يحوي تعليمة طباعة بسيطة، لكن الفرع الثاني يحتوي على تعليمة شرطية ثانية، والتي تملك فرعين بدورها هي الأخرى. لحسن الحظ، كلا هذين الفرعين يحتوي على تعليمة طباعة، مع أنه يمكن أن يكونا تعليمات شرطية أخرى أيضاً.

لاحظ أيضاً أن الترتيب جعل البنية واضحة، ومع هذا فإن التعليمات الشرطية المتداخلة تصعب قراءتها سريعاً. بشكل عام، ستكون فكرة جيدة أن تتفادى هذه التعليمات المتداخلة بقدر المستطاع.

من جهة أخرى، هذا النوع من **البنية المتداخلة (nested structure)** شائع، وسنراه ثانية فيما بعد، فمن الأفضل لك إذاً أن تعتاد عليه.

4.6 تعليمة العودة

تسمح لك تعليمة العودة بإنهاء تنفيذ عملية قبل الوصول إلى نهايتها. أحد الأسباب التي نستخدم هذه التعليمة لأجلها هو في حال اكتشاف شرط خاطئ:

```
public static void printLogarithm (double x) {
if (x <= 0.0) {
  System.out.println ("Positive numbers only, please.");
  return;
  }

  double result = Math.log (x);
  System.out.println ("The log of x is " + result);
```

```
}
```

في هذا المثال تعرَّف عملية باسم printLogarithm تأخذ معامل واحد من نوع double اسمه x. أول شيء تفعله هو التحقق ما إذا كان x أصغر من أو يساوي الصفر، حيث تطبع رسالة خطأ في هذه الحالة ثم تستعمل تعليمة العودة return للخروج من العملية. يعود مسار التنفيذ مباشرة إلى مستدعي العملية ولا يتم تنفيذ الأسطر الباقية من العملية.

لقد قمت باستخدام قيمة عشرية في الطرف الأيمن من الشرط بسبب وجود متغير عشري في الطرف الأيسر.

4.7 تحويل الأنواع

قد تتساءل عن الطريقة التي استطعنا من خلالها أن نكتب شيئاً مثل "The log of x is " + result، بما أن أحد الطرفين هو String والآخر double. حسناً، في هذه الحالة Java تتذاكى علينا، وتحول النوع من double إلى String تلقائياً قبل أن تربط السلسلتين معاً.

كلما حاولت "جمع" عبارتين معاً، وكانت إحداهما String، ستحول Java العبارة الأخرى إلى String ثم تجري عملية ربط السلاسل (string concatenation). ماذا تعتقد أنه سيحدث لو أنك أجريت عملية بين قيمة صحيحة وقيمة عشرية؟

4.8 التعاود

لقد ذكرت في الفصل السابق أنه يسمح لعملية ما أن تستدعي عملية أخرى، كما رأينا عدة أمثلة على ذلك. لقد أغفلت ذكر مشروعية استدعاء العملية لنفسها أيضاً. إن السبب الذي يجعل من هذه الإمكانية شيئاً جيداً قد لا يكون واضحاً، لكنه واحد من أكثر الأشياء التي تستطيع البرامج عملها سحراً وإثارة.

مثلاً، انظر إلى العملية التالية:

```
public static void countdown (int n) {
  if (n == 0)
    System.out.println ("Blastoff!");
  } else {
    System.out.println (n);
    countdown (n-1);
  }
}
```

اسم العملية هو countdown وهي تأخذ عدد صحيح واحد كمعامل. إذا كان المعامل صفراً، فستطبع الكلمة "Blastoff!". وإلا، فستطبع العدد ثم تستدعي عملية اسمها countdown – نفسها– وتمرر لها n-1 كمعامل.

ماذا يحدث لو استدعينا هذه العملية في main، هكذا مثلاً:

```
countdown (3);
```

يبدأ تنفيذ countdown مع n=3، وبما أن n ليس صفراً، تطبع العملية القيمة 3، ثم تستدعي نفسها...

يبدأ تنفيذ countdown مع n=2، وبما أن n ليس صفراً، تطبع العملية القيمة 2، ثم تستدعي نفسها...

يبدأ تنفيذ countdown مع n=1، وبما أن n ليس صفراً، تطبع العملية القيمة 1، ثم تستدعي نفسها...

يبدأ تنفيذ countdown مع n=0، وبما أن n صفر، ستطبع العملية الكلمة "Blastoff!" وبعدها ترجع إلى العملية السابقة.

ترجع العملية countdown ذات المتحول n=1.

ترجع العملية countdown ذات المتحول n=2.

ترجع العملية countdown ذات المتحول n=3.

وبعدها تجد نفسك في main ثانية (يا لها من رحلة). ويكون الخرج النهائي للبرنامج كما يلي:

```
3
2
1
Blastoff!
```

كمثال آخر، دعنا ننظر ثانية إلى العمليتين newLine وthreeLine.

```
public static void newLine () {
  System.out.println ("");
}
public static void threeLine () {
  newLine (); newLine (); newLine ();
}
```

بالرغم من أنهما تعملان، إلا أنهما لن تكونا مفيدتين فعلاً إذا أردت طباعة سطرين جديدين، أو 106. إن حلاً أفضل سيكون مثل هذا:

```
public static void nLines (int n) {
  if (n > 0) {
    System.out.println ("");
    nLines (n-1);
  }
}
```

هذا البرنامج شبيه جداً بالسابق؛ طالما أن n أكبر من الصفر، سيطبع سطراً جديداً واحداً، ثم يستدعي نفسه لطباعة n-1 سطر إضافي. بالتالي، فإن العدد الكلي للأسطر المطبوعة سيكون (n-1) + 1، وهو عادة ما يساوي إلى n تقريباً.

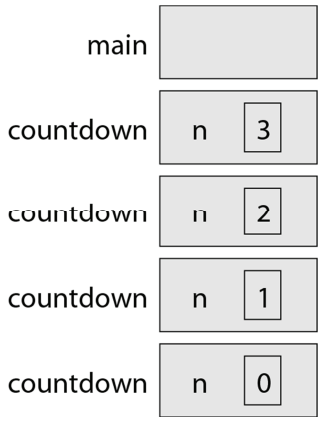
إن استدعاء العملية لنفسها يدعى **بالتعاود (recursion)**، ومثل هذه العمليات تدعى **تعاودية (recursive).**

4.9 المخططات الهرمية للعمليات التعاودية

في الفصل السابق استخدمنا مخططاً هرمياً لتمثيل حالة البرنامج عند استدعاء عملية. نفس النوع من المخططات يمكن أن يسهل تفسير عملية تعاودية.

تذكر أنه في كل مرة يتم فيها استدعاء عملية، تنشئ العملية حالة جديدة لنفسها تحتوي على نسخ جديدة لمتغيرات العملية ومعاملاتها.

الشكل التالي هو مخطط هرمي للعملية countdown، عند استدعائها مع n=3:



يوجد شكل واحد للعملية main وأربع أشكال للعملية countdown، كل واحد منها له قيمة مختلفة للمعامل n. قاع الهرم: العملية countdown ذات n=0 هو الحالة القاعدية. لا تستدعي هذه العملية نفسها مرة أخرى، لذلك فلا يوجد المزيد من الأشكال للعملية countdown.

إن الشكل المقابل للعملية main فارغ لأن main لا تملك أية معاملات أو متغيرات محلية.

تمرين 4.1 ارسم مخططاً هرمياً يظهر حالة البرنامج بعد استدعاء main للعملية nLines مع المعامل n=4، قبيل عودة آخر نسخة مستدعاة من nLines.

4.10 المصطلحات

باقي القسمة: عملية يتم تنفيذها على الأعداد الصحيحة وتعطي باقي قسمة العدد الأول على الثاني. في Java يرمز لهذه العملية برمز النسبة المئوية %.

التعليمة الشرطية: مجموعة من التعليمات التي قد يتم تنفيذها أو لا يتم اعتماداً على شرط ما.

الربط: طريقة لجمع عدة تعليمات شرطية بشكل متعاقب.

التداخل: وضع تعليمة شرطية ضمن فرع واحد أو أكثر من تعليمة شرطية أخرى.

الإحداثي: متغير أو قيمة تحدد موقعاً في نافذة بيانية ثنائية البعد.

البكسل (النقطة الإلكترونية): واحدة قياس الإحداثيات.

إطار مستطيل: طريقة شائعة لتحديد إحداثيات منطقة مستطيلة.

قولبة الأنماط: عملية تحوّل القيم من نوع إلى آخر. في Java تكتب هذه العملية بشكل اسم النوع بين قوسين، مثل (int).

الواجهة: وصف للمعاملات وأنواعها التي تتطلبها عملية ما.

النموذج الأولي: طريقة لوصف واجهة عملية ما باستخدام بنية مشابهة لتعليمات Java.

التعاود: هو استدعاء نفس العملية التي يتم تنفيذها حالياً.

التعاود اللانهائية: هي عملية تستدعي نفسها تعاودياً بدون الوصول إلى حالة أساسية أبداً. النتيجة المعتادة لعملية كهذه هي StackOverflowException.

الحالة القاعدية: شرط يؤدي لعدم استدعاء العملية التعاودية نفسها مرة أخرى.

modulus: An operator that works on integers and yields the remainder when one number is divided by another. In Java it is denoted with a percent sign (%).

conditional: A block of statements that may or may not be executed depending on some condition.

chaining: A way of joining several conditional statements in sequence.

nesting: Putting a conditional statement inside one or both branches of another conditional statement.

coordinate: A variable or value that specifies a location in a two-dimensional graphical window.

pixel: The unit in which coordinates are measured.

bounding box: A common way to specify the coordinates of a rectangular area.

typecast: An operator that converts from one type to another. In Java it appears as a type name in parentheses, like (int).

interface: A description of the parameters required by a method and their types.

prototype: A way of describing the interface to a method using Java-like syntax.

recursion: The process of invoking the same method you are currently executing.

infinite recursion: A method that invokes itself recursively without ever reaching the base case. The usual result is a StackOverflowException.

base case: A condition that causes a recursive method not to make a recursive call.

4.11 تمرينات

تمرين 4.2 هذا التمرين هو مراجعة لمجرى تنفيذ برنامج ذو عمليات متعددة. اقرأ الشفرة التالية وأجب عن الأسئلة أدناها.

```
public class Buzz {
  public static void baffle (String blimp) {
    System.out.println (blimp);
    zippo ("ping", -5);
  }

  public static void zippo (String quince, int flag) {
    if (flag < 0) {
      System.out.println (quince + " zoop");
    } else {
      System.out.println ("ik");
      baffle (quince);
      System.out.println ("boo-wa-ha-ha");
    }
  }

  public static void main (String[] args) {
    zippo ("rattle", 13);
  }

}
```

a. اكتب الرقم 1 بجوار أول تعليمة سيتم تنفيذها من هذا البرنامج. كن حذراً ولا تخلط بين التعليمات والأشياء الأخرى.
b. اكتب الرقم 2 بجوار التعليمة الثانية في التنفيذ، وتابع حتى ينتهي البرنامج. إذا تم تنفيذ تعليمة أكثر من مرة، فيمكن وضع أكثر من رقم بجوارها.
c. ما هي قيمة المعامل `blimp` عندما تم استدعاء `baffle`؟
d. ما هو خرج هذا البرنامج؟

تمرين 4.3 المقطع الأول من أغنية "99 Bottles of Beer" هو:

99 bottles of beer on the wall, 99 bottles of beer, ya' take one down, ya' pass it around, 98 bottles of beer on the wall.

المقاطع التالية مطابقة ما عدا أن عدد الزجاجات يقل بمقدار واحد في كل مقطع، حتى المقطع الأخير:

No bottles of beer on the wall, no bottles of beer, ya' can't take one down, ya' can't pass it around, 'cause there are no more bottles of beer on the wall!

وبعدها تنتهي الأغنية (أخيراً).

اكتب برنامجاً يطبع الكلمات الكاملة لأغنية "99 Bottles of Beer". يجب أن يحتوي برنامجك على عملية تعاودية تنفذ الجزء الصعب، لكنك قد ترغب أيضاً بكتابة عملية إضافية لفصل العمل الكلي للبرنامج.

أثناء تطوير الشفرة، قد ترغب باختبارها مع عدد أقل من المقاطع، مثل "3 Bottles of Beer".

الغرض من هذا التمرين هو أخذ مشكلة وفصلها إلى مشاكل أصغر، وحل المشاكل الصغيرة بكتابة عمليات بسيطة، سهلة التنقيح.

تمرين 4.4 ما هو خرج البرنامج التالي؟

```
public class Narf {
  public static void zoop (String fred, int bob) {
    System.out.println (fred);
    if (bob == 5) {
      ping ("not ");
    } else {
      System.out.println ("!");
    }
  }
  public static void main (String[] args) {
    int bizz = 5;
    int buzz = 2;
    zoop ("just for", bizz);
    clink (2*buzz);
  }
  public static void clink (int fork) {
    System.out.print ("It's ");
    zoop ("breakfast ", fork) ;
  }
  public static void ping (String strangStrung) {
    System.out.println ("any " + strangStrung + "more ");
  }
}
```

تمرين 4.5 آخر نظرية لفيرما (Fermat's Last Theorem) تقول بأنه لا يوجد أعداد صحيحة a، b، وc بحيث تحقق

$$a^n + b^n = c^n$$

ما عدا الحالة التي يكون فيها $n=2$.

اكتب عملية اسمها `checkFermat` تأخذ أربعة أعداد صحيحة كمعاملات —a, b, c and n— وتتحقق ما إذا كانت نظرية فيرما صحيحة. إذا كان n أكبر من 2 وتبين أن $a^n + b^n = c^n$، يجب أن يطبع البرنامج "Holy smokes, Fermat was wrong!" وبخلاف ذلك يجب أن يطبع البرنامج "No, that doesn't work".

عليك أن تفترض وجود عملية اسمها `raiseToPow` تأخذ عددين صحيحين كمتحولات وترفع العدد الأول إلى قوة العدد الثاني. مثلاً

```
int x = raiseToPow (2, 3);
```

ستسند هذه التعليمة القيمة 8 إلى x، لأن $2^3 = 8$.

الفصل 5

GridWorld: الجزء الأول

5.1 البدء بالعمل

الآن هو وقت مناسب لبدء العمل مع GridWorld، وهو برنامج يستخدم للتدريب على اختبار AP لعلوم الحاسوب. لتبدأ العمل، قم بتنصيب GridWorld، الذي تستطيع تنزيله من الموقع:
.http://www.collegeboard.com/student/testing/ap/compsci_a/case.html

عندما تفك الضغط عن الشفرة، يجب أن تحصل على مجلد اسمه GridWorldCode يحتوي على المجلد projects/firstProject، الذي يحتوي على BugRunner.java.

انسخ BugRunner.java إلى مجلد آخر ثم استورده إلى بيئة برمجتك. توجد هنا تعليمات قد تساعدك:
http://www.collegeboard.com/prod_downloads/student/testing/ap/compsci_a/
ap07_gridworld_installation_guide.pdf

(توجد نسخة مترجمة من هذه التعليمات على:
(http://thinklikecs.webs.com/book/resources/gridworld_installation_guide.pdf

بعد تشغيل BugRunner.java، نزِّل دليل الطالب لبرنامج GridWorld من
http://www.collegeboard.com/prod_downloads/student/testing/ap/compsci_a/
ap07_gridworld_studmanual_appends_v3.pdf

يستخدم دليل الطالب مفردات لم أشرحها بعد، لذا سأقدم لك شرحاً سريعاً حتى تستطيع البدء:

- مكونات GridWorld، بما فيها Bugs، Rocks وGrid، هي **كائنات (Objects)**.
- **الباني (constructor)** هو عملية خاصة تنشئ الكائنات الجديدة.
- **الصنف (class)** هو مجموعة من الكائنات؛ كل كائن ينتمي إلى صنف ما.
- يقال عن الكائن أيضاً أنه **حالة (instance)** لأنه عضو، أو حالة، من الصنف.
- **الصفة (attribute)** هي معلومة عن الكائن، مثل لونه أو مكانه.
- **عملية الوصول (accessor method)** هي عملية تعيد إحدى صفات الكائن.
- **عملية التعديل (modifier method)** تغير إحدى صفات الكائن.

الآن يجب أن تكون قادراً على قراءة الجزء الأول من دليل الطالب وأن تحل التمارين.

BugRunner 5.2

يحتوي BugRunner.java على هذه الشفرة:

```
import info.gridworld.actor.ActorWorld;
import info.gridworld.actor.Bug;
import info.gridworld.actor.Rock;

public class BugRunner
{
   public static void main(String[] args)
   {
      ActorWorld world = new ActorWorld();
      world.add(new Bug());
      world.add(new Rock());
      world.show();
   }
}
```

الأسطر الثلاثة الأولى هي تعليمات استيراد (import statements)؛ تستورد هذه الأسطر أصناف GridWorld المستخدمة في هذا البرنامج. يمكنك الحصول على وثائق هذه الأصناف من
http://thinklikecs.webs.com/resources/javadoc/gridworld/

مثل البرامج الأخرى التي شاهدناها، يعرف BugRunner صنفاً يوفر عملية main. ينشئ السطر الأول من main كائن ActorWorld. كلمة new هي كلمة مفتاحية في Java تنشئ الكائنات الجديدة.

ينشئ السطران التاليان Bug (حشرة) وRock (صخرة)، ويضيفانهما إلى world. يظهر السطر الأخير world على الشاشة.

افتح BugRunner.java للتحرير واستبدل هذا السطر:

```
world.add(new Bug());
```

بهذين السطرين:

```
Bug redBug = new Bug();
world.add(redBug);
```

يسند السطر الأول الحشرة إلى متغير اسمه redBug؛ يمكننا استعمال redbug لاستدعاء عمليات الحشرة. جرب هذه:

```
System.out.println(redBug.getLocation());
```

ملاحظة: إذا شغلت هذا قبل إضافة الحشرة إلى العالم (world)، فستكون النتيجة null (لا شيء)، ما يعني أن الحشرة لا تملك موقعاً بعد.

استدعي عمليات الوصول الأخرى واطبع صفات الحشرة. استدعي العمليات canMove، وmove وturn وتأكد من أنك تفهم عملها. الآن جرب هذه التمارين:

تمرين 5.1

a. اكتب عملية باسم moveBug تأخذ حشرة كمعامل وتستدعي move. اختبر عمليتك باستدعائها من main.
b. عدل moveBug بحيث تستدعي canMove وتحرك الحشرة فقط في حال كان ذلك ممكناً.
c. عدل moveBug بحيث تأخذ عدداً صحيحاً، n، كمعامل، وتحرك الحشرة n مرة (إذا كان ذلك ممكناً).
d. عدل moveBug بحيث تستدعي turn بدلاً من move إذا لم تكن الحشرة قادرة على الحركة.

تمرين 5.2

a. يوفر صنف Math عملية باسم random تعيد قيمة من نوع double محصورة بين 0.0 و1.0 (1.0 لا تنتمي لمجال القيم الممكن إعادتها).
b. اكتب عملية باسم randomBug تأخذ حشرة كمعامل وتضبط اتجاه الحشرة على 0، أو 90، أو 180 أو 270 باحتمالات متساوية، ثم تحرك الحشرة إذا كانت قادرة على الحركة.
c. عدل randomBug حتى تأخذ عدداً صحيحاً n وتكرر العملية n مرة.
النتيجة هي حركة عشوائية، التي تستطيع القراءة عنها في
.http://en.wikipedia.org/wiki/Random_walk
d. لكي تشاهد حركة عشوائية أطول، يمكنك إعطاء ActorWorld منصة أكبر. أضف تعليمة الاستيراد هذه إلى بداية BugRunner.java:

```
import info.gridworld.grid.UnboundedGrid;
```

الآن استبدل السطر الذي ينشئ ActorWorld بهذا:

```
ActorWorld world = new ActorWorld(new UnboundedGrid());
```

يجب أن تكون قادراً على تشغيل حركتك العشوائية لعدة آلاف من الخطوات (قد تحتاج لاستعمال أشرطة التمرير للعثور على الحشرة).

تمرين 5.3 يستعمل GridWorld كائنات الألوان (Color objects)، المعرَّفة في إحدى مكتبات Java. يمكنك قراءة الوثائق على .http: // download. oracle.com/ javase/ 6/ docs/ api/ java/ awt/ Color. html.

لعمل حشرات بألوان مختلفة، علينا استيراد Color:

```
import java.awt.Color;
```

ثم نتمكن من الوصول إلى الألوان المعرفة مسبقاً، مثل Color.blue، أو إنشاء لون جديد مثل هذا:

```
Color purple = new Color(148, 0, 211);
```

اصنع بضعة حشرات بألوان مختلفة. ثم اكتب عملية باسم colorBug تأخذ حشرة كمعامل، تقرأ موقعها، وتضبط اللون.

كائن الموقع الذي تحصل عليه من getLocation له عمليات باسم getRow وgetCol تعيد أعداداً صحيحة. بالتالي يمكنك الحصول على إحداثي الفواصل (x-coordinate) لحشرة ما بهذه الطريقة:

```
int x = bug.getLocation().getCol();
```

اكتب عملية باسم makeBugs تأخذ ActorWorld وعدد صحيح n، وتنشئ n حشرة ملونة بحسب مواقعها. استخدم رقم الصف للتحكم بدرجة اللون الأحمر، ورقم العمود للتحكم بدرجة الأزرق.

تُركت هذه الصفحة بيضاء عن عمد

الفصل 6

العمليات المثمرة

6.1 القيم المعادة

كانت بعض العمليات الجاهزة التي استخدمناها، مثل التوابع الرياضية، تعطي نتائجاً. أي أن الأثر الناجم عن استدعاء هذه العمليات هو توليد قيمة جديدة، نسندها عادة إلى متغير أو نستعملها كجزء من عبارة حسابية. مثلاً:

```
double e = Math.exp (1.0);
double height = radius * Math.sin (angle);
```

لكن جميع العمليات التي كتبناها حتى الآن كانت عمليات **فارغة (أو عقيمة void methods)**؛ أي أنها عمليات لا تعيد أية قيمة. عندما تستدعي عملية فارغة، ستكون العملية نموذجياً على سطر لوحدها، بدون أي إسناد:

```
nLines (3);
g.drawOval (0, 0, width, height);
```

في هذا الفصل، سوف نكتب عمليات تعيد أشياء، وسأدعوها بالعمليات **المثمرة (fruitful methods)**. المثال الأول هو area، التي تأخذ عدداً عشرياً (double) كمعامل، وتعيد مساحة الدائرة ذات نصف القطر المعطى:

```
public static double area (double radius) {
  double area = Math.PI * radius * radius;
  return area;
}
```

أول شيء عليك ملاحظته هو أن بداية تعريف العملية قد اختلفت. بدلاً من public static void، التي تشير إلى عملية فارغة، توجد public static double، التي تشير إلى أن القيمة المعادة من هذه العملية ستكون من النوع double. لم أشرح معنى public static حتى الآن، تحلى بالصبر.

أيضاً، لاحظ أن السطر الأخير هو شكل آخر لتعليمة return يتضمن قيمة معادة. هذه التعليمة تعني، "ارجع فوراً من هذه العملية واستعمل العبارة التالية كقيمة معادة." العبارة التي تقدمها لتعليمة return يمكن أن تكون معقدة بقدر ما ترغب، لذا كان من الممكن أن نكتب هذه العملية بشكل أكثر اختصاراً:

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

من جهة أخرى، **المتغيرات المؤقتة (temporary variables)** مثل area غالباً ما تجعل تنقيح الأخطاء أسهل. في كلا الحالتين، يجب أن يوافق نوع العبارة المستخدمة في تعليمة العودة نوع القيمة التي ترجعها العملية. بكلام آخر، عندما تصرح بأن نوع الإرجاع هو double، فإنك تعطي وعداً بأن هذه العملية ستنتج قيمة من نوع double في النهاية. إذا حاولت العودة بدون أي قيمة، أو باستخدام عبارة تنتج قيمة من نوع مخالف، فسيعاقبك المترجم.

أحياناً يكون من المفيد استعمال عدة تعليمات عودة، واحدة في كل فرع من فروع تعليمة شرطية:

```
public static double absoluteValue (double x) {
   if (x < 0) {
      return -x;
```

```
    } else {
       return x;
    }
}
```

بما أن تعليمات العودة هذه موجودة في تعليمة شرطية من نمط التنفيذ البديل، سيتم تنفيذ واحدة منها فقط. على الرغم من أن وجود أكثر من تعليمة return في عملية واحدة مشروع، إلا أنك يجب أن تتذكر دائماً أنه بمجرد تنفيذ واحدة منها، سيتم إنهاء العملية بدون تنفيذ أية تعليمات لاحقة.

الشفرة التي تظهر بعد تعليمة return، أو أي مكان آخر لا يمكن أن يتم تنفيذها فيه أبداً، تدعى **بالشفرة الميتة (dead code).** بعض المجمعات تحذرك في حال وجود جزء ميت في شفرتك.

إذا وضعت تعليمات العودة داخل تعليمات شرطية، فعليك أن تضمن أن كل مسار ممكن للبرنامج ينتهي بتعليمة عودة. مثلاً:

```
public static double absoluteValue (double x) {
  if (x < 0) {
    return -x;
  } else if (x > 0) {
    return x;
  }          // WRONG!!
}
```

هذا البرنامج غير مشروع بسبب وجود الحالة التي يكون فيها x=0، عندها لن يكون أي من الشروط صحيحاً وستنتهي العملية بدون الوصول إلى تعليمة عودة. إن الرسالة الصادرة عن مجمع نموذجي ستكون "مطلوب تعليمة عودة للعملية absoluteValue" - "return statement required in absoluteValue"، وهي رسالة محيرة إذ أنه يوجد تعليمتي عودة فيها.

6.2 تطوير البرامج

عند هذه النقطة يجب أن تكون قادراً على النظر إلى عمليات كاملة في Java ومعرفة ما تفعله. لكن قد لا تكون كيفية كتابة عمليات كهذه واضحة بعد. سأطرح إحدى التقنيات التي أدعوها **التطوير التصاعدي (incremental development).**

كمثال، تخيل أنك تريد حساب المسافة بين نقطتين، إحداثياتهما (x_1, y_1) و(x_2, y_2). باستخدام التعريف المعتاد،

$$distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

الخطوة الأولى هو الأخذ بعين الاعتبار الشكل الذي ستظهر به عملية distance في Java. بكلمات أخرى، ما هو الدخل (المعاملات) وما هو الخرج (القيمة المعادة).

في هذه الحالة، النقطتين هما المعاملات، ومن الطبيعي أن نعبر عنهما باستخدام أربع أعداد عشرية، مع أننا سنرى فيما بعد أن Java تملك كائناً يدعى Point يمكننا استعماله. القيمة المعادة هي المسافة، والتي ستكون من النوع double.

يمكننا الآن كتابة مخططاً تمهيدياً للعملية:

```
public static double distance
            (double x1, double y1, double x2, double y2) {
  return 0.0;
}
```

التعليمة `return 0.0;` هي إشارة علام ضرورية حتى يتم تجميع البرنامج. من الواضح أن البرنامج لا يجري أي شيء مفيد في هذه المرحلة، لكنه يستحق تجربة تجميعه لنتمكن من العثور على أية أخطاء نحوية قبل أن نضيف المزيد من الشفرة.

حتى نختبر العملية الجديدة، علينا استدعاؤها مع قيم اختبار. في مكان ما من `main` يمكن أن أضيف:

```
double dist = distance (1.0, 2.0, 4.0, 6.0);
```

لقد اخترت هذه القيم بحيث تكون المسافة الأفقية 3 والمسافة الشاقولية 4؛ وبهذا ستكون المسافة بين النقطتين 5 (الوتر في مثلث قائم الزاوية أضلاعه 3-4-5). عندما تختبر عملية ما، من المفيد أن تعلم الإجابة الصحيحة.

بعد أن تأكدنا من صحة بنية تعريف العملية، يمكننا البدء بإضافة أسطر من الشفرة بالتدريج. بعد كل تغيير تصاعدي، نجمع البرنامج ونشغله. بهذه الطريقة، سنعرف بالضبط المكان الذي يجب أن يحتوي على الخطأ في أية مرحلة: في السطر الأخير الذي أضفناه.

الخطوة التالية في حسبتنا هي إيجاد الفرق $x_2 - x_1$ و $y_2 - y_1$. سأخزن هذه القيم في متغيرات مؤقتة اسمها `dx` و`dy`.

```
public static double distance
                    (double x1, double y1, double x2, double y2) {
  double dx = x2 - x1;
  double dy = y2 - y1;
  System.out.println ("dx is " + dx);
  System.out.println ("dy is " + dy);
  return 0.0;
}
```

أضفت تعليمات طباعة تمكنني من معرفة القيم الوسيطة الناتجة قبل المتابعة. كما ذكرت من قبل، أنا أعلم مسبقاً أن القيم يجب أن تكون 3.0 و4.0.

عندما أنتهي من كتابة العملية سأزيل تعليمات الطباعة. شفرة كهذه تدعى **سقالات (scaffolding)**، لأنها تساعد في بناء البرنامج، لكنها ليست جزءاً من البرنامج النهائي. أحياناً يكون إبقاء السقالات فكرة جيدة، لكن حوِّلها لتعليقات، لكي تجدها في حال احتجت إليها لاحقاً.

الخطوة التالية في تطوير عمليتنا هو تربيع `dx` و`dy`. يمكننا استخدام عملية `Math.pow`، لكن من الأبسط والأسرع أن نضربهما بنفسيهما.

```
public static double distance
             (double x1, double y1, double x2, double y2) {
  double dx = x2 - x1;
  double dy = y2 - y1;
  double dsquared = dx*dx + dy*dy;
  System.out.println ("dsquared is " + dsquared);
  return 0.0;
}
```

مرة أخرى، سأترجم البرنامج وأشغله للتحقق من القيمة الوسيطة (التي يجب أن تكون 25.0).

أخيراً، يمكننا استخدام عملية `Math.sqrt` لحساب وإرجاع النتيجة.

```
public static double distance
             (double x1, double y1, double x2, double y2) {
  double dx = x2 - x1;
  double dy = y2 - y1;
  double dsquared = dx*dx + dy*dy;
  double result = Math.sqrt (dsquared);
```

```
  return result;
}
```

بعد ذلك في main، يجب أن نطبع القيمة الناتجة وأن نتحقق من صحتها.

عند اكتسابك المزيد من الخبرة البرمجية، قد تجد نفسك تكتب وتنقح أكثر من سطر في كل مرة. بغض النظر عن ذلك، فإن هذه عملية التطوير التصاعدي هذه توفر عليك الكثير من الوقت الضائع في تصحيح الأخطاء.

المقومات الأساسية لهذه العملية هي:

- ابدأ مع برنامج يعمل وقم بتغييرات صغيرة وتصاعدية. إذا حدث أي خطأ في أي مرحلة، فستعرف مكانه بالضبط.
- استعمل متغيرات مؤقتة لتخزين القيم الوسيطة حتى تتمكن من طباعتها والتحقق من صحتها.
- بعد أن يجهز البرنامج، قد ترغب بإزالة بعض السقالات أو دمج عدة تعليمات ضمن عبارات مركّبة، لكن بشرط ألا تجعل قراءة البرنامج أصعب.

6.3 التركيب

كما هو متوقع الآن، بعد أن تعرف عملية جديدة، يمكنك استعمالها كجزء من عبارة حسابية، ويمكنك بناء عمليات جديدة باستخدام عمليات موجودة من قبل. مثلاً، إذا أعطاك شخص نقطتين، مركز الدائرة ونقطة على محيطها، وطلب منك حساب مساحة الدائرة؟

دعنا نقل أن نقطة المركز مخزنة في المتغيرين xc وyc، ونقطة المحيط مخزنة في xp وyp. الخطوة الأولى هي إيجاد نصف قطر الدائرة، وهو البعد بين النقطتين. لحسن الحظ، لدينا عملية، distance قادرة على القيام بذلك.

```
double radius = distance (xc, yc, xp, yp);
```

الخطوة التالية هي حساب مساحة الدائرة، وإعادة قيمته:

```
double area = area (radius);
return area;
```

بجمع كل ذلك في عملية، سنحصل على:

```
public static double circleArea
                    (double xc, double yc, double xp, double yp) {
  double radius = distance (xc, yc, xp, yp);
  double area = area (radius);
  return area;
}
```

المتغيرات المؤقتة radius وarea مفيدة أثناء التطوير والتصحيح، بعد أن يعمل البرنامج بصورة صحيحة يمكننا جعله أكثر اختصاراً بتركيب استدعاءات العمليات:

```
public static double circleArea
                    (double xc, double yc, double xp, double yp) {
  return area (distance (xc, yc, xp, yp));
}
```

6.4 التحميل الزائد

لربما لاحظت في الفقرة السابقة أن circleArea وarea تجريان وظيفة مشابهة — حساب مساحة دائرة — لكنهما تأخذان معاملات مختلفة. بالنسبة إلى area، يجب أن نعطيها نصف القطر؛ أما circleArea، فيجب إعطاؤها نقطتين.

إذا قامت عمليتان بنفس الوظيفة فمن الطبيعي أن نعطيهما نفس الاسم. إن وجود أكثر من عملية بنفس الاسم، وهو ما يدعى **بالتحميل الزائد (Overloading)**، مشروع في Java طالما أن كل نسخة من العملية تأخذ معاملات مختلفة. لذا يمكننا أن نعيد تسمية circleArea:

```
public static double area
                  (double x1, double y1, double x2, double y2) {
  return area (distance (xc, yc, xp, yp));
}
```

عندما تستدعي عملية زائدة التحميل (Overloaded method)، ستعرف Java أي نسخة تريدها أنت بالاعتماد على المتحولات التي تعطيها. إذا كتبت:

```
double x = area (3.0);
```

ستبحث Java عن عملية اسمها area تأخذ متحولاً واحداً من نوع double، وهكذا تستعمل النسخة الأولى، التي تعامل المتحول على أنه نصف القطر. أما إذا كتبت:

```
double x = area (1.0, 2.0, 4.0, 6.0);
```

ستستخدم Java النسخة الثانية من area. لاحظ أيضاً لأن النسخة الثانية من area تستدعي النسخة الأولى عملياً.

العديد من أوامر Java الجاهزة (built-in commands) محمَّلة بشكل زائد، ما يعني وجود نسخ مختلفة تقبل عدداً مختلفاً أو أنواعاً مختلفة من المعاملات. مثلاً، هناك نسخ من print وprintln تقبل معاملاً واحداً من أي نوع. في صنف Math، توجد نسخة من abs تعمل مع الأعداد العشرية، ونسخة أخرى تعمل مع الأعداد الصحيحة.

على الرغم من أن ميزة التحميل الزائد مفيدة، يجب استعمالها بحذر. قد تجد نفسك ضائعاً فعلاً إذا كنت تحاول تنقيح نسخة ما من عملية في حين أنك تستدعي نسخة أخرى بالخطأ.

في الواقع، هذا يذكرني بإحدى القواعد الأساسية في تصحيح الأخطاء: **تأكد أن نسخة البرنامج التي تنظر إليها هي نفس نسخة البرنامج قيد التشغيل!**

ستجد نفسك في أحد الأيام وأنت تعدل في البرنامج مرة بعد أخرى، وترى نفس النتائج كلما شغلته. هذه علامة تحذير إلى أنك لسبب أو لآخر تشغل نسخة من البرنامج غير التي تعتقد أنك تشغلها. للتحقق، ضع تعليمة print (ليس مهماً ما تطبعه) وتأكد من أن سلوك البرنامج تغيّر وفقاً لذلك.

6.5 العبارات البوليانية

معظم العمليات التي رأيناها تعطي نتائجاً من نفس نوع معاملاتها (operands). مثلاً، عملية + تأخذ عددين صحيحين وتعطي عدد صحيح، أو عددين عشريين وتعطي عدد عشري، الخ.

الاستثناء الذي شاهدناه كان مع **العمليات المنطقية (relational operators)**، التي تقارن بين أعداد صحيحة أو بين أعداد عشرية وتعيد إما true أو false. true وfalse قيمتان خاصتان في Java، ومعاً يؤلفان نوعاً للقيم يدعى النوع **البولياني (boolean)**. لربما تتذكر عندما عرفت النوع، وقلت أنه مجموعة من القيم. في حالة الأعداد الصحيحة،

والأعداد العشرية والسلاسل المحرفية، كانت هذه المجموعات كبيرة جداً. بالنسبة إلى booleans، فالمجموعة ليست كبيرة حقاً.

العبارات والمتغيرات البوليانية تعمل تماماً كأنواع العبارات والمتغيرات الأخرى:

```
boolean bob;
bob = true;
boolean testResult = false;
```

المثال الأول هو تصريح بسيط عن متغير؛ المثال الثاني هو عملية إسناد، والثالث هو عمليتي تصريح وإسناد مدمجتين، أحياناً ندعو ذلك **بالتهيئة (initialization)**. القيم true وfalse هي كلمات مفتاحية في Java، لذا قد تظهران بلون مختلف، بحسب بيئة البرمجة التي تستخدمها.

كما ذكرت سابقاً، إن نتيجة العامل الشرطي بوليانية، لذا يمكنك تخزين نتيجة مقارنة في متغير:

```
boolean evenFlag = (n%2 == 0);          // true if n is even
boolean positiveFlag = (x > 0);         // true if x is positive
```

وأن تستعمله كجزء من تعليمة شرطية لاحقاً:

```
if (evenFlag) {
  System.out.println ("n was even when I checked it");
}
```

إن المتغير المستعمل بهذه الطريقة غالباً ما يدعى **علم (flag)**، نظراً لأنه يعلمنا بوجود أو غياب شرط معين.

6.6 العوامل المنطقية

هناك ثلاثة **عوامل منطقية (logical operators)** في Java: (و) AND، (أو) OR، و(النفي) NOT، التي ترمز بالرموز &&، || و!. إن معنى هذه العوامل مشابه لمعناها اللغوي. مثلاً، x > 0 && x < 10 محقق فقط إذا كان x أكبر من صفر **و** كان أصغر من 10.

evenFlag || n%3 == 0 محقق إذا كان أحد الشرطين محققاً، أي إذا كان evenFlag يحمل القيمة true **أو** كان n قابلاً للقسمة على 3.

أخيراً، عامل النفي NOT له أثر نقض أو عكس عبارة بوليانية، لذا فإن !evenFlag له قيمة true إذا كان evenFlag يحمل القيمة false — أي إذا كان العدد فردياً.

غالباً ما توفر العوامل المنطقية طريقة لتبسيط التعليمات الشرطية المتداخلة. مثلاً، كيف يمكننا كتابة الشفرة التالية باستخدام تعليمة شرطية واحدة؟

```
if (x > 0) {
  if (x < 10) {
    System.out.println ("x is a positive single digit.");
  }
}
```

6.7 العمليات البوليانية

يمكن للعمليات (methods) أن ترجع قيماً بوليانية بنفس الطريقة التي ترجع بها قيماً من أنواع أخرى، وهو شيء مفيد غالباً في إخفاء الاختبارات المعقدة داخل عمليات. مثلاً:

```
public static boolean isSingleDigit (int x) {
  if (x >= 0 && x < 10) {
    return true;
  } else {
    return false;
  }
}
```

اسم هذه العملية isSingleDigit. من الشائع إعطاء العمليات البوليانية أسماء تشبه yes/no questions. نوع القيمة المرجعة هو boolean، ما يعني أن تعليمة العودة يجب أن توفر عبارة بوليانية.

الشفرة بحد ذاتها بسيطة، إلا أنها أطول قليلاً مما تحتاج. تذكر أن العبارة x >= 0 && x < 10 ذات نوع بولياني، لذا يمكننا إرجاعها مباشرة، وتجنب عبارة if مرة واحدة:

```
public static boolean isSingleDigit (int x) {
  return (x >= 0 && x < 10);
}
```

في main يمكنك استدعاء عملية كهذه باستخدام الطرق المعتادة:

```
boolean bigFlag = !isSingleDigit (17);
System.out.println (isSingleDigit (2));
```

السطر الأول يسند القيمة true إلى bigFlag فقط في حال لم يكن 17 عدداً مؤلفاً من خانة واحدة. السطر الثاني يطبع true لأن 2 عدد مؤلف من خانة واحدة. نعم، تم التحميل الزائد للعملية println لتستطيع التعامل مع المتحولات البوليانية أيضاً.

الاستعمال الأكثر شيوعاً للعمليات البوليانية هو داخل التعليمات الشرطية

```
if (isSingleDigit (x)) {
  System.out.println ("x is little");
} else {
  System.out.println ("x is big");
}
```

6.8 المزيد من التعاود

الآن بعد أن أصبح لدينا عمليات ترجِع قيماً، أصبح لدينا لغة برمجة **كاملة وفقاً لمعيار تورين** (**Turing complete** programming language)، وذلك يعني أننا قادرين الآن أن نحسب أي شيء قابل للحوسبة، وذلك بالنسبة للتعاريف المنطقية لكلمة "حوسبة".

تم تطوير هذه الفكرة على يد العالمان ألونزو تشيرتش (Alonzo Church) وآلان تورين (Alan Turing)، وهي تعرف بفرضية تشيرتش-تورين. يمكنك قراءة المزيد عنها على http://en.wikipedia.org/wiki/Turing_thesis.

لأعطيك فكرة عما يمكنك فعله باستخدام الأدوات التي تعلمناها حتى الآن، دعنا نلق نظرة على بعض العمليات المستخدمة في حساب توابع رياضية معرفة تعاودياً. التعريف التعاودي مشابه لتعريف دائري، بمعنى أن التعريف يحتوي على مرجع يربطه بالشيء المعرَّف. التعريف الدائري الحقيقي ليس مفيداً جداً بشكل عام:

التعاودية: صفة تستخدم لوصف العمليات التعاودية.

إذا قرأت التعريف السابق في قاموس، فقد تنزعج حقاً. من جهة أخرى، إذا بحثت عن تعريف الرياضي للتابع العاملي (factorial)، فقد تصل إلى شيء مثل:

$$0! = 1$$
$$n! = n \cdot (n-1)!$$

(غالباً ما يتم ترميز التابع العاملي بالرمز !، الذي لا يجب الخلط بينه وبين عامل Java المنطقي ! الذي يعني النفي (NOT)). هذا التعريف يقول أن 0 عاملي يساوي 1، وأن عاملي أي قيمة أخرى، n، يساوي n مضروباً بعاملي n-1. إذاً 3! هو 3 × 2!، و2! هو 2 × 1!، و1! هو 1 × 0!. بوضعها جميعاً معاً، نحصل على أن 3! يساوي 3 × 2 × 1 × 1، يساوي 6.

إذا كنت تستطيع كتابة تعريف تعاودي لشيء ما، يمكنك عادة كتابة برنامج في Java لحسابه. الخطوة الأولى هي تحديد معاملات هذا التابع، ونوع القيمة التي يرجعها. مع قليل من التفكير، يجب أن تستنتج أن التابع العاملي يأخذ عدداً صحيحاً واحداً كمعامل ويرجع قيمة صحيحة:

```
public static int factorial (int n) {
}
```

إذا تصادف أن يكون المتحول صفراً، كل ما علينا فعله هو إرجاع القيمة 1:

```
public static int factorial (int n) {
  if (n == 0) {
  return 1;
  }
}
```

وهذه هي الحالة القاعدية.

وإلا، هذا هو الجزء الممتع، علينا عمل استدعاء تعاودي لحساب عاملي n-1، ثم ضربه بn.

```
public static int factorial (int n) {
  if (n == 0) {
    return 1;
  } else {
    int recurse = factorial (n-1);
    int result = n * recurse;
    return result;
  }
}
```

لو نظرنا إلى مجرى تنفيذ هذا البرنامج، لوجدناه مشابهاً لـ countdown من القسم 4.8. إذا استدعينا factorial مع القيمة 3:

بما أن 3 ليست صفراً، نتبع الفرع الثاني ونحسب العاملي لـ n-1.

بما أن 2 ليس صفراً، نتبع الفرع الثاني ونحسب العاملي لـ n-1.

بما أن 1 ليست صفراً، نتبع الفرع الثاني ونحسب العاملي لـ n-1.

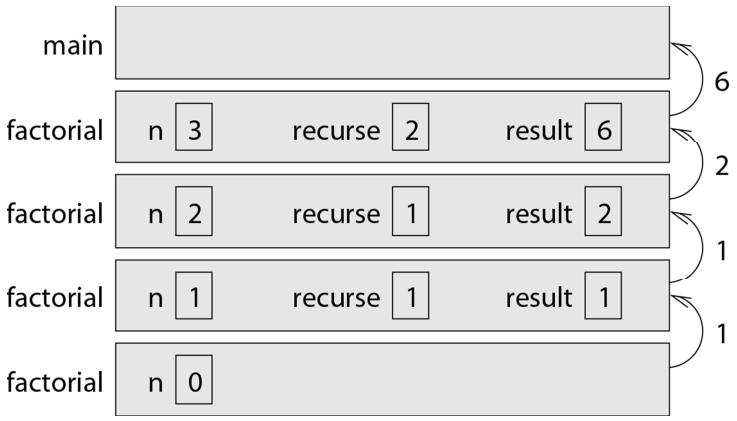
بما أن 0 هو الصفر، نتبع الفرع الأول ونعيد القيمة 1 فوراً بدون القيام بأية استدعاءات تعاودية أخرى.

يتم ضرب القيمة المعادة (1) بـ n، الذي يساوي 1، وتتم إعادة النتيجة.

يتم ضرب القيمة المعادة (1) بـ n، الذي يساوي 2، وتتم إعادة النتيجة.

يتم ضرب القيمة المعادة (2) بـ n، الذي يساوي 3، وتتم إعادة النتيجة، 6، إلى `main`، أو أياً كان من استدعى العملية `factorial` (3).

هنا تجد المخطط الهرمي لهذه السلسلة من استدعاءات العمليات:



يتم إظهار القيم المعادة وهي تمرر عائدة إلى أعلى الهرم.

لاحظ أنه في آخر حالة للعملية factorial، لا يوجد متغيرات محلية باسم recurse أو result لأنه عندما يكون n=0 لا يتم تنفيذ الفرع الذي ينشئهم.

6.9 وثبة الثقة

إن اتباع مجرى التنفيذ هو إحدى الطرق المتبعة عند قراءة البرامج، لكن كما شاهدت في الفصل السابق، يمكن للمسارات في هذه الطريقة أن تتداخل وتتعقد بسرعة. من الطرق البديلة طريقة أدعوها "قفزة الثقة". عندما تصل إلى استدعاء عملية، بدلاً من تتبع مجرى التنفيذ، افرض أن العملية تعمل بشكل صحيح وأنها تعيد القيمة المناسبة.

في الواقع، لقد مارست هذه الطريقة سابقاً عندما كنت تستعمل العمليات المبنية مسبقاً في Java. عندما تستدعي `Math.cos` أو `drawOval`، فلن تفحص مجريات هذه العمليات. بل تفترض أنها تعمل بشكل صحيح، لأن الناس الذي كتبوا هذه العمليات كانوا مبرمجين جيدين.

حسن، نفس الشيء يصح عندما تستدعي واحدة من العمليات التي كتبتها بنفسك. مثلاً، في القسم 6.7 كتبنا عملية اسمها `isSingleDigit` تحدد ما إذا كان العدد بين 0 و9 أو لا. بمجرد إقناع أنفسنا بأن هذه العملية صحيحة — بعد اختبار وفحص الشفرة — يمكننا استخدام تلك العملية بدون النظر إلى الشفرة مرة أخرى أبداً.

نفس الشيء صحيح أيضاً مع البرامج التعاودية. عندما تصل إلى الاستدعاء التعاودي، بدلاً من لحاق مجرى التنفيذ، عليك أن تفترض أن الاستدعاء التعاودي صحيح (وأنه يعطي النتيجة الصحيحة)، وبعدها اسأل نفسك، "بفرض أنني حسبت عاملي n-1، هل أستطيع حساب عاملي n؟" في هذه الحالة، من الواضح انك تستطيع، بضربه بn.

طبعاً، من الغريب قليلاً أن تفترض أن العملية تعمل بشكل صحيح قبل أن تنتهي من كتابتها حتى، لكن هذا هو السبب وراء تسميتها قفزة الثقة!

6.10 مثال إضافي أخير

ثاني أشهر مثال عن التوابع الرياضية المعرفة تعاودياً بعد factorial، هو fibonacci، المعرف بالشكل التالي:

$$fibonacci(0) = 1$$
$$fibonacci(1) = 1$$
$$fibonacci(n) = fibonacci(n - 1) + fibonacci(n - 2)$$

وبترجمته إلى Java يصبح:

```
public static int fibonacci (int n) {
  if (n == 0 || n == 1) {
     return 1;
  } else {
     return fibonacci (n-1) + fibonacci (n-2);
  }
}
```

إذا حاولت تتبع مجرى التنفيذ هنا، حتى من أجل قيم صغيرة للمتحول n، فإن رأسك سينفجر. لكن حسب قفزة الثقة، إذا افترضنا أن الاستدعاءين التعاوديين (نعم، يمكنك عمل استدعاءين تعاوديين) يعملان بشكل صحيح، عندئذ سيكون واضحاً أننا سنحصل على الإجابة الصحيحة بجمعهما معاً.

6.11 المصطلحات

نوع الإرجاع: الجزء من التصريح عن العملية الذي يحدد نوع القيمة التي ستعيدها العملية.

القيمة المعادة: القيمة المقدمة على أنها ناتج استدعاء العملية.

الشفرة الميتة: جزء من البرنامج لا يمكن تنفيذه أبداً، غالباً لأنه يظهر بعد تعليمة return.

السقالات: شفرة تستخدم أثناء تطوير البرنامج لكنها ليست جزءاً من الإصدار النهائي.

void: نوع إرجاع خاص يشير إلى عملية عقيمة؛ وهي عملية لا تعيد أي قيمة.

التحميل الزائد: هو وجود أكثر من عملية بنفس الاسم لكن بمعاملات مختلفة. عندما تستدعي عملية محملة بشكل زائد، ستعرف Java أي واحدة تقصد من خلال المتحولات التي تعطيها إياها.

بولياني: نوع متغير يمكن له تخزين إحدى القيمتين فقط: true أو false.

علم (راية): متغير (بولياني عادة) يسجل شرطاً أو معلومات عن حالة.

عامل شرطي: عامل يقارن بين قيمتين وينتج قيمة بوليانية تشير إلى العلاقة بين المعاملات.

عامل منطقي: عامل يركب قيمتين بوليانيتين وينتج قيمة بوليانية.

return type: The part of a method declaration that indicates what type of value the method returns.

return value: The value provided as the result of a method invocation.

dead code: Part of a program that can never be executed, often because it appears after a return statement.

scaffolding: Code that is used during program development but is not part of the final version.

void: A special return type that indicates a void method; that is, one that does not return a value.

overloading: Having more than one method with the same name but different parameters. When you invoke an overloaded method, Java knows which version to use by looking at the arguments you provide.

boolean: A type of variable that can contain only the two values true and false.

flag: A variable (usually boolean) that records a condition or status information.

conditional operator: An operator that compares two values and produces a boolean that indicates the relationship between the operands.

logical operator: An operator that combines boolean values and produces boolean values.

6.12 تمرينات

تمرين 6.1 اكتب عملية اسمها isDivisible تأخذ عددين صحيحين، n و m وتعيد القيمة true إذا كان n قابلاً للقسمة على m و false فيما عدا ذلك.

تمرين 6.2 يمكن التعبير عن العديد من العمليات المعقدة بشكل مختصر باستخدام عملية "الضرب-جمع"، التي تأخذ ثلاثة معاملات وتحسب a*b + c. حتى أن بعض المعالجات توفر معدات خاصة لتنفيذ هذه العملية للأعداد العشرية.

a. أنشئ برنامجاً جديداً باسم `Multadd.java`.

b. اكتب عملية اسمها `multadd` تأخذ ثلاثة معاملات من نوع `double` وتطبع ناتج ضرب-جمعهم.

c. اكتب عملية `main` تختبر `multadd` باستدعائها باستخدام معاملات بسيطة، مثل `1.0`، `2.0`، `3.0`.

d. أيضاً في `main`، استعمل `multadd` لحساب القيم التالية:

$$\sin\frac{\pi}{4} + \frac{\cos\frac{\pi}{4}}{2}$$

$$\log 10 + \log 20$$

e. اكتب عملية تدعى `yikes` تأخذ عدداً عشرياً كمعامل وتستخدم `multadd` لحساب وطباعة

$$xe^{-x} + \sqrt{1 - e^{-x}}$$

مساعدة: العملية الرياضية التي ترفع e إلى قوة هي Math.exp.

في الجزء الأخير، ستكتب عملية تستدعي عملية أخرى قمتَ بكتابتها. كلما فعلت ذلك، سيكون من الجيد أن تختبر العملية الأولى بتأن قبل البدء بالعمل على الثانية. وإلا، فقد تجد نفسك تدقق عمليتين في نفس الوقت، ما قد يصبح صعباً جداً.

إن أحد أهداف هذا التمرين هو التدرب على "مطابقة النموذج" – "pattern-matching": القدرة على التعرف على مشكلة محددة كحالة خاصة من فئة عامة من المشاكل.

تمرين 6.3 إذا أعطيتَ ثلاثة عيدان، فقد تستطيع ترتيبها بشكل مثلث وقد لا تستطيع. مثلاً، إذا كان طول أحد العيدان 12 بوصة وكان طول كل من العودين الآخرين 1 بوصة، فمن الواضح أنك لن تستطيع جعل العودين القصيرين يلتقيان في المنتصف. بالنسبة لأي ثلاثة أطوال، هناك اختبار بسيط لمعرفة ما إذا كان تشكيل مثلث منها ممكناً:

> "إذا كان أي واحد من الأطوال الثلاثة أكبر من مجموع الطولين الآخرين، لا يمكنك تشكيل مثلث منها، وبخلاف ذلك تستطيع"

اكتب عملية اسمها isTriangle تأخذ ثلاثة أعداد صحيحة كمتحولات، وتعيد قيمة إما true وإما false، اعتماداً على إمكانية أو عدم إمكانية تشكيل مثلث من عيدان أطوالها تساوي الأعداد المعطاة.

الهدف من هذا التمرين هو استعمال التعليمات الشرطية لكتابة عملية مثمرة.

تمرين 6.4 ما هو خرج البرنامج التالي؟ إن الغرض وراء هذا التمرين هو التأكد من أنك تفهم العوامل المنطقية ومجرى التنفيذ في العمليات المثمرة.

```
public static void main (String[] args) {
  boolean flag1 = isHoopy (202);
  boolean flag2 = isFrabjuous (202);
  System.out.println (flag1);
  System.out.println (flag2);
  if (flag1 && flag2) {
    System.out.println ("ping!");
  }
  if (flag1 || flag2) {
    System.out.println ("pong!");
  }
}

public static boolean isHoopy (int x) {
  boolean hoopyFlag;
  if (x%2 == 0) {
    hoopyFlag = true;
  } else {
    hoopyFlag = false;
  }
  return hoopyFlag;
}

public static boolean isFrabjuous (int x) {
  boolean frabjuousFlag;
  if (x > 0) {
    frabjuousFlag = true;
  } else {
    frabjuousFlag = false;
  }
  return frabjuousFlag;
```

```
}
```

تمرين 6.5 المسافة بين نقطتين (x_1, y_1) و(x_2, y_2) هي

$$distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

اكتب عملية اسمها distance تأخذ أربعة معاملات من نوع double– x1, y1, x2, y2- وتطبع المسافة بين النقطتين.

عليك أن تفترض وجود عملية اسمها sumSquares تحسب وتعيد مجموع مربعات معاملاتها، مثلاً:

```
double x = sumSquares (3.0, 4.0);
```

ستسند القيمة 25.0 للمتغير x.

الغرض من هذا التمرين هو كتابة عملية جديدة تستخدم عملية موجودة. عليك كتابة عملية واحدة فقط: distance. لا تكتب العملية sumSquares ولا main كما لا تستدعي distance.

تمرين 6.6 إن الغرض من هذا التمرين هو استخدام المخططات الهرمية لفهم تنفيذ برنامج تعاودي.

```
public class Prod {
  public static void main (String[] args) {
    System.out.println (prod (1, 4));
  }
  public static int prod (int m, int n) {
    if (m == n) {
      return n;
    } else {
      int recurse = prod (m, n-1);
      int result = n * recurse;
      return result;
    }
  }
}
```

a. ارسم مخططاً هرمياً يبين حالة البرنامج قبيل اكتمال حالة prod الأخيرة. ما هو خرج هذا البرنامج؟

b. اشرح باستعمال بضعة كلمات ما تفعله العملية prod.

c. أعد كتابة prod بدون استخدام المتغيرات المؤقتة recurse وresult.

تمرين 6.7 الغرض من هذا التمرين هو ترجمة تعريف تعاودي إلى عملية مكتوبة بلغة Java. يعرف تابع أكرمان (Ackerman function) بالنسبة للأعداد غير السالبة كما يلي:

$$A(m,n) = \begin{cases} n+1 & \textbf{if} \quad m = 0 \\ A(m-1,1) & \textbf{if} \quad m > 0, n = 0 \\ A(m-1, A(m,n-1)) & \textbf{if} \quad m > 0, n > 0 \end{cases}$$

اكتب عملية اسمها ack تأخذ عددين صحيحين كمعاملات وتحسب وإعادة قيمة تابع أكرمان.

اختبر عملية أكرمان باستدعائها من main وطباعة القيمة المعادة.

تحذير: القيمة المعادة تكبر كثيراً بسرعة كبيرة. عليك اختبارها مع قيم صغيرة للمتغيرين n وm (ليس أكبر من 2).

تمرين 6.8

a. أنشئ برنامجاً يدعى Recurse.java واكتب فيه العمليات التالية:

```
// first: returns the first character of the given String
public static char first (String s) {
  return s.charAt (0);
}

// last: returns a new String that contains all but the
// first letter of the given String
public static String rest (String s) {
  return s.substring (1, s.length());
}

// length: returns the length of the given String
public static int length (String s) {
  return s.length();
}
```

b. اكتب بعض الشفرة في main لاختبار هذه العمليات. تأكد من أنها تعمل، وتأكد من أنك قد فهمت وظائفها.

c. اكتب عملية باسم printString تأخذ سلسلة محرفية كمعامل وتطبع أحرف تلك السلسلة، واحد على كل سطر. يجب أن تكون عملية فارغة (void method).

d. اكتب عملية اسمها printBackward تفعل بنفس ما تفعله printString عدا أنها تطبع السلسلة بالمقلوب (الحرف الأخير على السطر الأول).

e. اكتب عملية اسمها reverseString تأخذ سلسلة محرفية كمعامل وتعيد سلسلة جديدة كقيمة معادة. يجب أن تحتوي السلسلة الجديدة على نفس حروف السلسلة المعطاة، لكن بترتيب معكوس. مثلاً، سيكون خرج الشفرة التالية

```
String backwards = reverseString ("Allen Downey");
System.out.println (backwards);
```

كما يلي

```
yenwoD nellA
```

تمرين 6.9 اكتب عملية تعاودية اسمها power تأخذ عدداً عشرياً x وعدداً صحيحاً n وتعيد القيمة x^n.
مساعدة: التعريف التعاودي لهذه العملية هو power (x, n) = x * power (x, n-1). أيضاً، تذكر أن أي شيء مرفوع للقوة صفر يعطي 1.

تحد اختياري: يمكنك جعل هذه العملية أكثر فاعلية، عندما يكون n زوجياً، باستخدام $x^n = (x^{\frac{n}{2}})^2$

تمرين 6.10 (هذا التمرين مبني على الصفحة 44 من كتاب *Structure and Interpretation of Computer Programs* للمؤلفين Ableson and Sussman).

الخوارزمية التالية تعرف باسم خوارزمية إقليدس لأنها مكتوبة في كتاب العناصر لإقليدس (الكتاب 7، 300 ق.م.). قد تكون هذه أقدم خوارزمية معروفة[1].

تبنى الخوارزمية على الملاحظة التالية، إذا كان r باقي قسمة a على b، عندئذ تكون القواسم المشتركة للعدين a وb هي نفس القواسم المشتركة للعدين b وr. وبالتالي يمكننا استخدام المعادلة

[1] لتعرف ما هي "الخوارزمية"، انظر القسم 11.13

$$gcd(a, b) = gcd(b, r)$$

لتحويل معضلة حساب ق.م.أ (القاسم المشترك الأكبر) لعددين صحيحين إلى حساب القاسم المشترك الأكبر لعددين أصغر فأصغر. مثلاً،

$$gcd(36, 20) = gcd(20, 16) = gcd(16, 4) = gcd(4, 0) = 4$$

ينتج أن القاسم المشترك الأكبر (GCD Great Common Divider) للعددين 36 و20 هو 4. يمكن البرهان على أنه من أجل أي عددين ابتدائيين، فإن تكرار هذه العملية سينتهي إلى حساب ق.م.أ لعددين يكون الثاني منهما يساوي الصفر. ويكون ق.م.أ المطلوب هو العدد الأول منهما.

اكتب عملية اسمها gcd تأخذ عددين صحيحين كمعاملين وتستعمل خوارزمية إقليدس لحساب وإرجاع القاسم المشترك الأكبر للعددين.

الفصل 7

التكرار

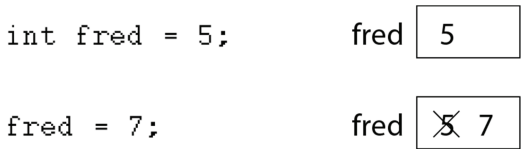
7.1 الإسناد المتعدد

لم أتكلم كثيراً عن ذلك سابقاً، لكن يسمح بعمل أكثر من عملية إسناد للمتغير الواحد في Java. إن ما ستفعله تعليمة الإسناد الثانية هو استبدال قيمة المتغير القديمة بواحدة جديدة.

```
int fred = 5;
System.out.print (fred);
fred = 7;
System.out.println (fred);
```

إن خرج هذا البرنامج هو 57، لأن أول مرة طبعنا فيها fred كانت قيمته 5، وفي المرة الثانية تكون قيمته 7.

هذا النوع من **الإسناد المتعدد (multiple assignment)** هو سبب وصفي للمتغيرات بأنها حاويات للقيم. عندما تسند قيمة ما لتغير، فإنك تغير محتويات الحاوية، كما هو مبين في الشكل:



عند وجود عدة تعليمات إسناد لمتغير واحد، من المهم بصورة خاصة التمييز بين تعليمات الإسناد وتعليمات المساواة. بما أن Java تستعمل الرمز = للإسناد، فقد يؤدي ذلك لتفسير تعليمة مثل a = b على أنها تعليمة مساواة. وهي ليست كذلك!

أولاً، المساواة تبديلية، والإسناد ليس كذلك. مثلاً، في الرياضيات إذا كان a = 7 فإن 7 = a. لكن في Java التعليمة a = 7; هي تعليمة إسناد مشروعة، أما 7 = a; ليست كذلك.

أكثر من ذلك، في الرياضيات، تعليمة المساواة محققة دائماً. إذا كان a = b الآن، فإن a سيظل مساوياً لـ b إلى الأبد. في Java، يمكن لتعليمة الإسناد أن تساوي بين متغيرين، لكن لا شيء يفرض عليهما البقاء هكذا!

```
int a = 5;
int b = a;          // a and b are now equal
a = 3;              // a and b are no longer equal
```

يغير السطر الثالث قيمة a لكنه لا يغير قيمة b، وهكذا لم يعد a وb متساويين. في العديد من لغات البرمجة الأخرى يتم استعمال رمز آخر لعملية الإسناد، مثل -> أو =: ، وذلك لتفادي هذا الإرباك.

على الرغم من أن الإسناد المتعدد مفيد كثيراً، عليك التعامل معه بحذر. إذا استمرت قيم المتغيرات بالتغير في أجزاء البرنامج المختلفة، فستصبح الشفرة أصعب للقراءة والتنقيح.

7.2 التكرار

إن أحد الأشياء التي تستخدم الحواسيب لأجلها غالباً هو أتمتة العمليات التكرارية. تكرار المهام المتشابهة أو المتطابقة بدون عمل أخطاء هو شيء تبرع فيه الحواسيب ويفشل فيه البشر.

شاهدنا سابقاً برامج تنفذ عمليات مكررة تعاودياً، مثل countdown وfactorial، التي تستخدم التعاود لتكرار التعليمات، كما توفر Java عدة مزايا تجعل كتابة الرامج التكرارية أسهل. (**repeate** or **iterate**)

إن التعليمتين اللتين سنتعلمهما هما تعليمة while وتعليمة for.

7.3 تعليمة while

باستعمال تعليمة while، يمكننا إعادة كتابة countdown:

```
public static void countdown (int n) {
  while (n > 0) {
    System.out.println (n);
    n = n-1;
  }
  System.out.println ("Blastoff!");
}
```

يمكنك قراءة تعليمة while كما لو أنها مكتوبة باللغة الطبيعية تقريباً. هذه التعليمة تعني، "طالما n أكبر من الصفر، تابع طباعة قيمة n وبعدها إنقاص قيمة n بقدار 1. عندما تصل للصفر، اطبع الكلمة 'Blastoff!'".

بصورة طبيعية أكثر، يكون مجرى تنفيذ تعليمة while كما يلي:

1. احسب الشرط بين قوسين، يعطي true أو false.
2. إذا كان الشرط غير محقق، اخرج من تعليمة while وتابع التنفيذ عند التعليمة التالية.
3. إذا كان الشرط محققاً، نفذ التعليمات الموجودة بين الأقواس المنحنية، وبعدها ارجع إلى الخطوة 1.

هذا النوع من المسارات يدعى **حلقة (loop)** لأن الخطوة الثالثة تكون حلقة مع الخطوة الأولى. لاحظ أنه في حال عدم تحقق الشرط عند الوصول إلى الحلقة أول مرة، فلن يتم تنفيذ التعليمات الموجودة داخل الحلقة أبداً. تدعى التعليمات داخل الحلقة أحياناً **جسم (body)** الحلقة.

يجب أن يتم تغيير قيمة متغير أو أكثر ضمن جسم الحلقة بحيث، في النهاية، يصبح الشرط غير محقق وتنتهي الحلقة. وإلا فإن تكرار الحلقة سيستمر للأبد، وهو ما يدعى بالحلقة اللانهائية (**infinite** loop). من مصادر التسلية التي لا تنضب بالنسبة لعلماء الكمبيوتر هو ملاحظة أن التعليمات على علبة الشامبو، "ارغي، اشطف، كرر"، هي حلقة لانهائية.

في حالة countdown، يمكننا التأكد أن الحلقة ستنتهي لأننا نعلم أن قيمة n منتهية، ويمكننا أن نرى أن قيمة n تصغر في كل مرة يتم فيها تنفيذ الحلقة (في كل تكرار)، لذا يجب أن نصل إلى الصفر في النهاية. في حالات أخرى لا تكون معرفة ذلك سهلة:

```
public static void sequence (int n) {
  while (n != 1) {
    System.out.println (n);
    if (n%2 == 0) {        // n is even
      n = n / 2;
    } else {               // n is odd
```

```
      n = n*3 + 1;
    }
  }
}
```

إن شرط هذه الحلقة هو 1 =! n، لذا فإن تنفيذ الحلقة سيستمر حتى يصبح n مساوياً 1، ما سيجعل الشرط غير محقق.

في كل تكرار، سيطبع البرنامج قيمة n وبعدها يتحقق ما إذا كانت زوجية أو فردية. إذا كانت زوجية، سيتم تقسيمها على 2. إذا كانت فردية، سيتم استبدالها بالقيمة 1 + 3n. مثلاً، إذا كانت القيمة الابتدائية (المتحول المرر إلى sequence) تساوي 3، سيكون التتابع الناتج هو 3، 10، 5، 16، 8، 4، 2، 1.

بما أن n يزداد أحياناً وينقص أحياناً أخرى، فلا يوجد أي إثبات واضح على أن n سيصل للواحد، أو أن البرنامج سينتهي. بالنسبة لقيم معينة من n، يمكننا إثبات أن البرنامج سينتهي. مثلاً، إذا كانت القيمة الابتدائية من قوى العدد 2، عندئذ ستكون قيمة n زوجية في كل مرة يتم تنفيذ الحلقة فيها، حتى نصل للواحد. المثال السابق ينتهي بسلسلة مثل هذه، تبدأ بالعدد 16.

إذا وضعنا القيم الخاصة جانباً، فإن السؤال المثير هو هل يمكننا إثبات أن هذا البرنامج سينتهي مع أية قيمة لـ n. حتى الآن، لم يتمكن أي شخص من إثبات ذلك أو إثبات العكس! لمزيد من المعلومات، انظر http://en.wikipedia.org/wiki/Collatz_conjecture.

7.4 الجداول

من الأشياء التي تفيدنا الحلقات فيها هو توليد وطباعة البيانات المجدولة. مثلاً، قبل توفر الحواسيب بسهولة، كان البشر مضطرين لحساب الخوارزميات، الجيب والتجيب (جيب التمام)، وتوابع رياضية شائعة أخرى يدوياً.

لتسهيل العملية، وجدت كتب تحوي جداول طويلة يمكنك أن تجد فيها قيم التوابع المختلفة. كان عملية إنشاء هذه الجداول طويلة ومملة، وغالباً ما كانت النتائج مليئة بالأخطاء.

عندما دخلت الحواسيب على الساحة، كان أحد ردود الفعل المبدئية هو، "هذا عظيم! يمكننا استخدام الحواسيب لتوليد الجداول، وبذلك لن توجد فيها أي أخطاء". تبين فيما بعد أن ذلك كان صحيحاً (أغلب الأحيان)، لكنه قليل التبصر. فسرعان ما عم انتشار الحواسيب (والآلات الحاسبة) حتى أصبحت الجداول منتهية الصلاحية.

حسن، ليس تماماً. تبين فيما بعد أنه من أجل بعض العمليات، تستعمل الحواسيب جداول للقيم للحصول على أجوبة تقريبية، ثم تنفذ حسابات لتحسين نسبة التقريب. في بعض الحالات، وجدت أخطاء في الجداول التحتية، ومن أكثر تلك الحالات شهرة الخطأ في الجدول الذي استخدمه معالج إنتل بنتيوم الأول لتنفيذ عملية القسمة على الأعداد العشرية[1].

على الرغم من أن "جدول اللوغاريتم" لم يعد مفيداً كما كان، فلا يزال مثالاً جيداً على التكرار. البرنامج التالي يطبع سلسلة من القيم في العمود الأيسر ولوغاريتماتها في العمود الأيمن:

```
double x = 1.0;
while (x < 10.0) {
  System.out.println (x + "   " + Math.log(x));
  x = x + 1.0;
}
```

إن خرج هذا البرنامج هو:

```
1.0   0.0
2.0   0.6931471805599453
3.0   1.0986122886681098
```

[1] انظر http://en.wikipedia.org/wiki/Pentuim_FDIV_bug.

```
4.0    1.3862943611198906
5.0    1.6094379124341003
6.0    1.791759469228055
7.0    1.9459101490553132
8.0    2.0794415416798357
9.0    2.1972245773362196
```

بالنظر إلى هذه القيم، هل يمكنك معرفة الأساس الذي يستخدمه تابع log افتراضياً؟

نظراً لأن قوى العدد اثنين مهمة جداً في علوم الحاسوب، غالباً ما نرغب بحساب اللوغاريتمات ذات الأساس 2. لمعرفة هذه، يمكننا استخدام المعادلة التالية:

$$log_2 x = log_e x / log_e 2 \qquad (7.1)$$

بتغيير تعليمة الطباعة إلى

```
System.out.println (x + "   " + Math.log(x) / Math.log(2.0));
```

يعطي

```
1.0    0.0
2.0    1.0
3.0    1.5849625007211563
4.0    2.0
5.0    2.321928094887362
6.0    2.584962500721156
7.0    2.807354922057604
8.0    3.0
9.0    3.1699250014423126
```

يمكنك أن ترى أن 1، 2، 4 و8 هي قوى العدد 2، لأن لوغاريتماتها ذات الأساس 2 هي أعداد صحيحة. إذا أردت معرفة لوغاريتمات قوى العدد اثنين الأخرى، يمكننا تعديل البرنامج كما يلي:

```
double x = 1.0;
while (x < 100.0) {
  System.out.println (x + "   " + Math.log(x) / Math.log(2.0));
  x = x * 2.0;
}
```

الآن بدلاً من إضافة شيء ما إلى x في كل مرة تدور فيها الحلقة، ما يعطي متتالية حسابية، نضرب x بشيء ما، سيعطينا هذا متتالية **هندسية**. النتيجة هي:

```
1.0    0.0
2.0    1.0
4.0    2.0
8.0    3.0
16.0    4.0
32.0    5.0
64.0    6.0
```

قد لا تكون جداول اللوغاريتم مفيدة الآن، لكن بالنسبة لعالم كمبيوتر، فإن معرفة قوى 2 مهمة! في وقت ما عندما تملك دقيقة فراغ، عليك حفظ قوى الاثنين حتى 65536 (هذا يساوي 2^{16}).

7.5 الجداول ثنائية البعد

الجدول ثنائي البعد هو حيث نختار سطراً وعموداً ونقرأ القيمة عند تقاطعهما. جدول الضرب مثال جيد. لنقل أنك تريد طباعة جدول الضرب للأعداد من 1 إلى 6.

إحدى الطرق الجيدة للبدء هي كتابة حلقة بسيطة تطبع جدول ضرب 2، كله على سطر واحد.

```
int i = 1;
while (i <= 6) {
  System.out.print (2*i + " ");
  i = i + 1;
}
System.out.println ("");
```

يهيئ السطر الأول متغيراً اسمه i، وهو سيعمل كعداد، أو **متغير الحلقة (loop variable)**. بينما يتم تنفيذ الحلقة، تزداد قيمة i من 1 إلى 6. وبعدها عندما تصبح قيمة i تساوي 7، تنتهي الحلقة. في كل دورة تكرارية للحلقة، نطبع القيمة 2*i متبوعة بثلاث مسافات. بما أننا نستعمل أمر print بدلاً من println، سيظهر الخرج كله على سطر واحد.

كما ذكرت في القسم 2.4، في بعض بيئات البرمجة يتم تخزين الخرج من تعليمة print بدون عرضها على الشاشة حتى يتم استدعاء println. إذا انتهى البرنامج، ونسيت أن تستدعي println، فقد لا ترى النتائج المخزنة أبداً.

إن خرج هذا البرنامج هو:

```
2     4     6     8     10    12
```

حتى الآن الوضع جيد. الخطوة التالية هي **التغليف (encapsulation)** و**التعميم (generalize)**.

7.6 التغليف والتعميم

التغليف أو الكبسلة يعني عادة أخذ جزء من الشفرة ولفّه في عملية، ما يسمح لك بالاستفادة من مزايا كل الأشياء التي تفيدنا العمليات فيها. قد رأينا مثالين عن التغليف، عندما كتبنا printParity في القسم 4.3 وisSingleDigit في القسم 5.7.

التعميم يعني أخذ شيء مخصص، مثل طباعة جدول ضرب الاثنين، وتعميمه أكثر، مثل طباعة جدول الضرب لأي عدد صحيح.

إليك عملية تغلف الحلقة من القسم السابق وتعممها لتطبع جدول الضرب للعدد n.

```
public static void printMultiples (int n) {
  int i = 1;
  while (i <= 6) {
    System.out.print (n*i + " ");
    i  = i + 1;
  }
  System.out.println ("");
}
```

لتغليفها، كل ما كان علي فعله هو إضافة السطر الأول، الذي يصرح عن الاسم، المعامل، ونوع الإرجاع. للتعميم، كل ما كان علي فعله هو استبدال القيمة 2 بالمعامل n.

إذا استدعيت هذه العملية باستخدام المتحول 2، فسأحصل على نفس الخرج السابق. وباستخدام المتحول 3، سيكون الخرج كما يلي:

```
3      6      9      12     15     18
```

مع المتحول 4، سيكون الخرج كما يلي

```
4      8      12     16     20     24
```

ستتمكن على الأغلب من تخمين الطريقة التي سنطبع بها جدول الضرب الآن: سنستدعي printMultiples بشكل متكرر مع المتحولات المختلفة. في الواقع، سنستخدم حلقة أخرى لتكرر العملية على السطور.

```
int i = 1;
while (i <= 6) {
  printMultiples (i);
  i = i + 1;
}
```

قبل كل شيء، لاحظ الشبه بين هذه الحلقة وبين الحلقة الموجودة داخل printMultiples. كل ما فعلته هو استبدال تعليمة الطباعة باستدعاء لعملية.

خرج البرنامج سيكون كالتالي

```
1      2      3      4      5      6
2      4      6      8      10      12
3      6      9      12      15      18
4      8      12      16      20      24
5      10      15      20      25      30
6      12      18      24      30      36
```

وهو يمثل جدول الضرب (مع أنه مضطرب قليلاً). إذا أزعجك عدم ترتيب الجدول، فإن Java توفر عمليات تعطيك المزيد من التحكم بتنسيق الخرج، لكنني لن أخوض في الحديث عنها هنا.

7.7 العمليات

لقد ذكرت بعض فوائد العمليات في القسم 3.5. إليك بعض الأسباب الإضافية التي تجعل العمليات مفيدة:

- بإعطاء اسم إلى سلسلة من العمليات، تجعل برنامجك أسهل للقراءة والتنقيح.
- تقسيم برنامج طويل إلى عمليات يسمح لك بفصل أجزاء من البرنامج، تنقيحها بشكل منفصل، وبعدها تركيبها في قطعة واحدة ثانية.
- العمليات تسهل التكرار والتعاود.
- العمليات المصممة بشكل جيد غالباً ما تكون مفيدة لعدة برامج. بمجرد كتابة وتنقيح إحدى العمليات، يمكنك إعادة استخدامها.

7.8 المزيد من التغليف

لشرح التغليف مرة ثانية، سآخذ الشفرة من القسم السابق وأضمّها في عملية:

```
public static void printMultTable () {
```

```
  int i = 1;
  while (i <= 6) {
    printMultiples (i);
    i  = i + 1;
  }
}
```

العملية التي أشرحها هي خطة تطوير شائعة تدعى **التغليف والتعميم (encapsulation and generalization)**. تبدأ بإضافة أسطر إلى `main` أو أي عملية أخرى، وبعد ذلك عندما تعمل تلك الشفرة، تستخرجها وتغلفها بعملية. بعدها تعمم تلك العملية بإضافة معاملات.

أحياناً لا تعرف تماماً كيف ستقسم البرنامج إلى عمليات عندما تبدأ الكتابة. هذه الطريقة تسمح لك بتصميم البرنامج بينما تتقدم في طريقك.

7.9 المتغيرات المحلية

في هذا الوقت، قد تتساءل كيف تمكنا من استخدام نفس المتغير `i` في العمليتين `printMultiples` و`printMultTable`. ألم أقل أنك تستطيع التصريح عن المتغير مرة واحدة فقط؟ وألا يسبب ذلك المشاكل عندما تبدّل إحدى العمليتين قيمة المتغير؟

إن الإجابة عن السؤالين هي "لا"، لأن `i` في العملية `printMultiples` و`i` في العملية `printMultTable` *ليسا المتغير نفسه*. صحيح انهما يملكان نفس الاسم، لكنهما لا يشيران إلى نفس المنطقة التخزينية، وإن تغيير قيمة أحدهما لا يؤثر على الآخر.

المتغيرات التي يتم التصريح عنها داخل تعريف عملية تدعى **بالمتغيرات المحلية (local variables)** لأنها محدودة بالعملية التي توجد فيها. لا يمكنك الوصول إلى متغير محلي من خارج عمليته "الأم"، ولك الحرية في امتلاك عدة متغيرات محلية بنفس الاسم، طالما أنها غير موجودة في العملية نفسها.

من الجيد غالباً استعمال أسماء مختلفة في العمليات المختلفة، لتفادي الإرباك، لكن توجد أسباب وجيهة لإعادة استعمال الأسماء. مثلاً، من الشائع استعمال الأسماء `i`، `j`، و`k` كمتغيرات حلقة. إذا تفاديت استعمالها في عملية واحدة فقط لأنها قد استعملت في مكان آخر، فعلى الأغلب أنك ستجعل البرنامج أصعب للقراءة.

7.10 المزيد من التعميم

كمثال آخر عن التعميم، تخيل أنك أردت برنامجاً يطبع جدول الضرب بأي قياس، ليس فقط جدول 6×6. يمكنك إضافة معامل إلى `printMultTable`:

```
public static void printMultTable (int high) {
  int i = 1;
  while (i <= high) {
    printMultiples (i);
    i = i + 1;
 }
}
```

استبدلت القيمة 6 بالمعامل `high`. إذا استدعيت `printMultTable` مع المتحول 7، سأحصل على

```
1      2      3      4      5      6
2      4      6      8      10      12
3      6      9      12      15      18
```

```
4      8      12      16      20      24
5      10      15      20      25      30
6      12      18      24      30      36
7      14      21      28      35      42
```

وهو جدول جيد، عدا أنني أريده أن يكون مربعاً (له نفس العدد من الأسطر والأعمدة)، ما يعني إضافة معامل آخر إلى `printMultiples`، لتحديد عدد الأعمدة التي يجب أن يحتوي عليها الجدول.

فقط لأنني أريد أن أكون مزعجاً، سأدعو هذا المعامل `high` أيضاً، لأوضح أن العمليات المختلفة يمكن أن تملك معاملات لها نفس الاسم (تماماً مثل المتغيرات المحلية):

```
public static void printMultiples (int n, int high) {
  int i = 1;
  while (i <= high) {
    System.out.print (n*i + " ");
    i = i + 1;
  }
  newLine ();
}

public static void printMultTable (int high) {
  int i = 1;
  while (i <= high) {
    printMultiples (i, high);
    i = i + 1;
  }
}
```

لاحظ أنه عند إضافة المعامل الجديد، اضطررت إلى تغيير السطر الأول من العملية، كما اضطررت أيضاً إلى تغيير المكان الذي تم استدعاء العملية `printMultiples` فيه. كما هو متوقع، سيولد هذا البرنامج جدول 7×7 مربع:

```
1      2      3      4      5      6      7
2      4      6      8      10      12      14
3      6      9      12      15      18      21
4      8      12      16      20      24      28
5      10      15      20      25      30      35
6      12      18      24      30      36      42
7      14      21      28      35      42      49
```

عندما تعمم عملية ما بشكل مناسب، غالباً ما ستكتشف أن البرنامج الناتج يملك قدرات لم تكن تقصدها. مثلاً، لربما لاحظت أن جدول الضرب متناظر، لأن $ab = ba$، لذا فإن كل المدخلات في الجدول تظهر مرتين. يمكنك توفير الحبر بطباعة نصف الجدول فقط. لعمل ذلك، عليك تعديل سطر واحد فقط من العملية `printMultTable`. عدل

```
printMultiples (i, high);
```

إلى

```
printMultiples (i, i);
```

وستحصل على

```
1
2      4
3      6      9
4      8      12      16
5      10      15      20      25
6      12      18      24      30      36
7      14      21      28      35      42      49
```

سأترك لك مهمة اكتشاف كيفية حصول ذلك.

7.11 المصطلحات

حلقة: تعليمة يتم تنفيذها طالما أن الشرط محقق أو حتى يتحقق الشرط.

الحلقة اللانهائية: حلقة يكون شرطها محققاً دائماً.

الجسم: التعليمات الموجودة داخل الحلقة.

دورة (تكرارية): مرور واحد على (تنفيذ) جسم الحلقة، متضمناً التحقق من الشرط.

التغليف (الكبسلة): تقسيم برنامج كبير معقد إلى مكونات (مثل العمليات) وعزل المكونات عن بعضها البعض (مثلاً، باستخدام المتغيرات المحلية).

المتغير المحلي: متغير يتم التصريح عنه داخل عملية وهو يوجد ضمن حدود تلك العملية فقط. لا يمكن الوصول إلى المتغيرات المحلية من خارج عمليتها الأم، وهي لا تتعارض مع أي عملية أخرى.

تعميم: استبدال شيء غير مخصص بالضرورة (مثل قيمة ثابتة) بشيء عام بشكل مناسب (مثل متغير أو معامل). التعميم يجعل الشفرة متعددة الأغراض أكثر، وأكثر قابلية لإعادة الاستخدام، وفي بعض الأحيان تصبح الشفرة أسهل في الكتابة.

خطة التطوير: عملية تطوير البرنامج. في هذا الفصل، شرحت أسلوباً في التطوير مبني على أساس تطوير الشفرة للقيام بأعمال بسيطة ومحددة، ثم القيام بالتغليف والتعميم. في القسم 5.2 شرحت تقنية أسميتها التطوير التصاعدي. في الفصل الأخير سأقترح المزيد من أساليب التطوير.

loop: A statement that executes repeatedly while or until some condition is satisfied.

infinite loop: A loop whose condition is always true.

body: The statements inside the loop.

iteration: One pass through (execution of) the body of the loop, including the evaluation of the condition.

encapsulate: To divide a large complex program into components (like methods) and isolate the components from each other (for example, by using local variables).

local variable: A variable that is declared inside a method and that exists only within that method. Local variables cannot be accessed from outside their home method, and do not interfere with any other methods.

generalize: To replace something unnecessarily specific (like a constant value) with something appropriately general (like a variable or parameter). Generalization makes code more versatile, more likely to be reused, and sometimes even easier to write.

development plan: A process for developing a program. In this chapter, I demonstrated a style of development based on developing code to do simple, specific things, and then encapsulating and generalizing. In Section 5.2 I demonstrated a technique I called incremental development. In later chapters I will suggest other styles of development.

7.12 تمرينات

تمرين 7.1

```
public static void main (String[] args) {
  loop (10);
}

public static void loop (int n) {
  int i = n;
  while (i > 1) {
    System.out.println (i);
    if (i%2 == 0) {
      i = i/2;
    } else {
      i = i+1;
    }
  }
}
```

a. ارسم جدولاً يبين قيم المتغيرين i و n خلال تنفيذ الحلقة. يجب أن يحتوي الجدول على عمود واحد لكل متغير وسطر واحد لكل تكرار.

b. ما هو خرج هذا البرنامج؟

تمرين 7.2 لنقل أنك أعطيت رقماً، a، وأنك تريد معرفة جذره التربيعي. إحدى الطرق هي تخمين الناتج، x_0، ثم تحسين ذلك التخمين باستخدام المعادلة التالية:

$$x_1 = (x_0 + a/x_0) /2 \qquad (7.2)$$

مثلاً، إذا أردت معرفة الجذر التربيعي للعدد 9، وبدأت مع العدد $x_0 = 6$، ثم $x_1 = (6+ 9/6)/2 = 15/4 = 3.75$، وهو أقرب إلى الجواب الصحيح.

يمكنك إعادة الإجراء نفسه، باستخدام x_1 لحساب x_2، وهكذا. في هذه الحالة، $x_2 = 3.075$ و $x_3 = 3.00091$. حيث يتقارب الرقم بسرعة إلى الإجابة الصحيحة (3).

اكتب عملية باسم squareRoot تأخذ عدداً من نوع double كمعامل وتعيد الجذر التربيعي التقريبي لذلك المعامل، باستخدام هذه الخوارزمية. لا يمكنك استخدام العملية الجاهزة Math.sqrt.

كما توقعتَ لأول وهلة، عليك استخدام $a/2$، عمليتك يجب أن تتكرر حتى تصل إلى عددين تقريبيين متتاليين يكون الاختلاف بينهما أقل من 0.0001؛ بمعنى آخر، حتى تصبح القيمة المطلقة لـ $x_n - x_{n-1}$ أقل من 0.0001. يمكنك استعمال العملية الجاهزة Math.abs لحساب القيمة المطلقة.

تمرين 7.3 في التمرين 6.9 كتبنا إصداراً تعاودياً من power، يأخذ عدداً عشرياً x وعدداً صحيحاً n ويعيد x^n. الآن اكتب عملية تكرارية للقيام بنفس الحسبة.

تمرين 7.4 يقدم القسم 6.10 عملية تعاودية تحسب العاملي. اكتب إصداراً تكرارياً من factorial.

تمرين 7.5 إحدى طرق حساب e^x هي استعمال السلسلة المنشورة اللانهائية

$$e^x = 1 + x + x^2/2! + x^3/3! + x^4/4! + \ldots \qquad (7.3)$$

إذا كان اسم متغير الحلقة اسمه i، عندئذ يكون العنصر رقم i مساوياً $x^i/i!$.

a. اكتب عملية باسم myexp تجمع n من الحدود الأولى من السلسلة المبينة أعلاه. يمكنك استخدام عملية factorial من القسم 6.10 أو النسخة التكرارية التي كتبتها في التمرين السابق.

b. يمكنك جعل هذه العملية أكثر فاعلية إذا أدركت أنه في كل تكرار يكون بسط الحد هو نفس بسط الحد السابق مضروباً بـ x والمقام هو نفس المقام السابق مضروباً بـ i. استفد من هذه الملاحظة للتخلص من العمليتين Math.pow وfactorial، بعدئذ تأكد من أن النتيجة لاتزال نفسها.

c. اكتب عملية اسمها check تأخذ معامل وحيد، x، وتطبع قيم x، وMath.exp(x) وmyexp(x) لقيم مختلفة للمتغير x. يجب أن يبدو الخرج مشابهاً لما يلي:

```
1.0     2.708333333333333     2.718281828459045
```

مساعدة: يمكنك استخدام المحرف "\t" لطباعة علامة جدولة بين أعمدة الجدول.

d. غير عدد الحدود في السلسلة (المتحول الثاني الذي ترسله check إلى myexp) وانظر إلى تأثير ذلك على دقة النتيجة. اضبط هذه القيمة حتى تتوافق القيمة التقريبية مع الإجابة "الصحيحة" عندما يكون x يساوي 1.

e. اكتب حلقة في main تستدعي check مع القيمة 0.1، 1.0، 10.0 و100.0. كيف تتغير دقة النتيجة مع تغير قيمة x؟ قارن عدد الخانات المتوافقة بين القيم الحقيقية والمقدرة بدلاً من حساب الفرق بين القيمتين.

f. أضف حلقة في main تتحقق من العملية check مع القيم 0.1-، 1.0-، 10.0-، و 100.0-. ضع تعليقاً على الدقة.

تمرين 7.6 من الطرق الممكنة لحساب e^{-x^2} هو استخدام المنشور اللانهائي التالي

$$e^{-x^2} = 1 - x^2 + x^4/6 - x^6/24 + \cdots \qquad (7.4)$$

بكلام آخر، نحتاج لجمع سلسلة من الحدود حيث يعطى الحد رقم i بالصيغة $(-1)^i\ x^{2i}/i!$. اكتب عملية باسم guess تأخذ x و n كمتحولات وتعيد مجموع n حد من الحدود الأولى من السلسلة. يجب عدم استخدام factorial أو pow.

تُركت هذه الصفحة بيضاء عن عمد

الفصل 8

السلاسل المحرفية

8.1 استدعاء العمليات على الكائنات

في Java وغيرها من لغات البرمجة كائنية التوجه، توجد الكائنات، وهي مجموعات من البيانات المرتبطة ببعضها والتي ترفق بمجموعة من العمليات. تعمل هذه العمليات على الكائنات، تجري الحسابات وأحياناً تعدل بيانات الكائن.

من بين الأنواع التي شاهدناها حتى الآن، كانت السلاسل المحرفية (Strings) هي الكائنات الوحيدة. اعتماداً على تعريف الكائن، قد تتساءل "ما هي البيانات الموجودة في الكائن من نوع String؟" و "ما هي العمليات التي يمكننا استدعاؤها على كائن String؟"

البيانات المحتواة في كائن String هي حروف السلسلة المحرفية، أو –بصورة عامة أكثر- محارفها. هناك عدة عمليات قليلة تعمل على السلاسل المحرفية، لكنني لن استخدم سوى بضعة عمليات منها فقط في هذا الكتاب. بقية العمليات موثقة على http://download.oracle.com/javase/6/docs/api/java/lang/String.html.

أول عملية سنتعرف عليها هي charAt، التي تسمح لك باستخراج الحروف من السلسلة المحرفية. ولتخزين النتيجة، نحتاج إلى نوع متغيرات يمكنه تخزين الحروف المفردة. الحروف المفردة تدعى المحارف (characters)، ونوع المتغيرات الذي يخزنهم يدعى char.

تعمل المتغيرات من نوع char مثل بقية المتغيرات التي تعاملنا معها سابقاً:

```
char fred = 'c';
if (fred == 'c') {
  System.out.println (fred);
}
```

تظهر قيم المحارف بين علامتي تنصيص مفردة ('c'). بعكس القيم من نوع string (التي تظهر بين علامتي تنصيص مزدوجة)، القيم المحرفية يمكن أن تحتوي على حرف واحد فقط أو رمز.

هنا شرح طريقة استعمال عملية charAt:

```
String fruit = "banana";
char letter = fruit.charAt(1);
System.out.println (letter);
```

تشير التركيبة fruit.charAt إلى أنني أستدعي العملية charAt على الكائن fruit. مررت المتحول 1 إلى هذه العملية، ما يشير إلى أنني أود معرفة المحرف الأول من السلسلة. النتيجة هي قيمة من نوع char، يتم تخزينها في المتغير letter. عندما أطبع قيمة المتغير letter، أحصل على مفاجأة:

```
a
```

a ليس الحرف الأول من كلمة "banana". إلا إذا كنت عالماً بالكمبيوتر. تطبيقاً لمبدأ "خالف تعرف"، يبدأ علماء الكمبيوتر العد من الصفر دائماً. الحرف الصفري ("0th - "zeroeth letter) من كلمة "banana" هو b. أما الحرف الأول ("1th - "oneth) هو a والحرف الثاني ("2th - "twooth) هو n.

إذا أردت الحرف رقم 0 من سلسلة، عليك تمرير الصفر كمتحول:

```
char letter = fruit.charAt(0);
```

8.2 Length

العملية الثانية من عمليات String التي سنتعرف عليها هي العملية length، التي تعيد عدد المحارف الموجودة في سلسلة محرفية. مثلاً:

```
int length = fruit.length();
```

لا تأخذ length أية متحولات، وهو ما نشير إليه ب ( )، وتعيد عدداً صحيحاً، في هذه الحالة 6. لاحظ أنه من المسموح وجود متغير وعملية لهما نفس اسم (على الرغم من أن هذا قد يشوش القراء البشر).

لمعرفة الحرف الأخير من سلسلة، قد تحاول كتابة شيء مثل هذا

```
int length = fruit.length();
char last = fruit.charAt (length);              // WRONG!!
```

ذلك لن يعمل. السبب هو عدم وجود حرف سادس في كلمة "banana". نظراً لأننا بدأنا العد من الصفر، فإن الحروف الستة مرقمة من 0 إلى 5. للحصول على المحرف الأخير، عليك طرح واحد من length.

```
int length = fruit.length();
char last = fruit.charAt (length-1);
```

8.3 الاجتياز

من الأشياء الشائع فعلها بالسلاسل هو البدء عند المحرف الأول، اختيار كل محرف بدوره، إجراء عملية ما عليه، والمتابعة حتى نهاية السلسلة. هذا النموذج من المعالجة يدعى **الاجتياز (أو العبور traversal)**. إحدى الطرق الطبيعية المستخدمة لترميز الاجتياز (أو تشفيره - بمعنى كتابته بالشفرة البرمجية) تتم باستخدام تعليمة while:

```
int index = 0;
while (index < fruit.length()) {
  char letter = fruit.charAt (index);
  System.out.println (letter);
  index = index + 1;
}
```

هذه الحلقة تعبر السلسلة المحرفية وتطبع كل حرف على سطر لوحده. لاحظ أن الشرط هو `index < fruit.length()`، ما يعني أن الشرط لن يتحقق عندما يكون index مساوياً لطول السلسلة المحرفية، ولن يتم تنفيذ جسم الحلقة. آخر محرف سوف نعالجه هو المحرف ذا الدليل `fruit.length()-1`.

اسم متغير الحلقة هو index. **الدليل (index)** هو متغير أو قيمة تستخدم لتحديد عنصر ما من مجموعة مرتبة (في هذه الحالة مجموعة المحارف في السلسلة المحرفية). الدليل يشير إلى (أو يدل على – بحسب اسمه) العنصر الذي تريده. يجب أن تكون المجموعة مرتبة حتى يكون لكل حرف دليل وكل دليل يشير إلى محرف واحد فقط.

تمرين 8.1 اكتب عملية تأخذ سلسلة محرفية كمتحول وتطبع الأحرف بالمقلوب (من الأخير إلى الأول)، على سطر واحد.

8.4 أخطاء وقت التشغيل

بعيداً جداً في القسم 1.3.1، تحدثت عن أخطاء التشغيل (run-time errors)، وهي أخطاء لا تظهر قبل بدء البرنامج بالعمل. تدعى أخطاء التشغيل في Java **بالاستثناءات (exceptions)**.

الأغلب أنك لم تر العديد من أخطاء التشغيل حتى الآن، لأننا لم نفعل أي أشياء تؤدي إلى ذلك. حسن، الآن نحن نفعل ذلك. إذا استخدمت أمر charAt وأعطيته دليلاً سالباً أو أكبر من length-1، فستحصل على استثناء: بشكل محدد، ستحصل على StringIndexOutOfBoundsExeption. جرب وانظر كيف سيبدو.

إذا تسبب برنامجك باستثناء، سيطبع رسالة خطأ تبين نوع الاستثناء بالإضافة إلى **سجل المكدس (stack trace)**، الذي يبين العملية التي كانت تجري وقت حدوث الاستثناء. إليك مثالاً:

```
public class BadString {
   public static void main(String[] args) {
      processWord("banana");
   }

   public static void processWord(String s) {
      char c = getLastLetter(s);
      System.out.println(c);
   }

   public static char getLastLetter(String s) {
      int index = s.length();          // WRONG!
      char c = s.charAt(index);
      return c;
   }
}
```

لاحظ الخطأ في getLastLetter: يجب أن يكون دليل آخر حرف `s.length()-1`. إليك ما ستحصل عليه:

```
Exception in thread "main" java.lang.StringIndexOutOfBoundsException:
String index out of range: 6
       at java.lang.String.charAt(String.java:694)
       at BadString.getLastLetter(BadString.java:24)
       at BadString.processWord(BadString.java:18)
       at BadString.main(BadString.java:14)
```

بعدها ينتهي البرنامج. قد تكون قراءة سجل المكدس صعبة، لكنه يحتوي على الكثير من المعلومات.

تمرين 8.2 اقرأ سجل المكدس وأجب عن الأسئلة التالية:

- ما هو نوع الاستثناء الحاصل، وما هي الحزمة التي عرِّف فيها؟
- ما هي قيمة الدليل التي سببت الاستثناء؟
- ما هي العملية التي أطلقت الاستثناء (threw the exception)، وأين تم تعريف تلك العملية؟
- ما هي العملية التي استدعت charAt؟
- في BadString.java، ما هو رقم السطر الذي استدعيت فيه العملية charAt؟

8.5 قراءة الوثائق

إذا ذهبت إلى http://download.oracle.com/javase/6/docs/api/java/lang/String.html

ونقرت على charAt، ستحصل على التوثيق التالي (أو شيء يشبهه):

```
public char charAt(int index)
```

Returns the char value at the specified index. An index ranges from 0 to length() - 1. The first char value of the sequence is at index 0, the next at index 1, and so on, as for array indexing.

Parameters: index - the index of the char value.

Returns: the char value at the specified index of this string. The first char value is at index 0.

Throws: IndexOutOfBoundsException - if the index argument is negative or not less than the length of this string.

السطر الأول هو **نموذج** العملية **الأولي (prototype)**، الذي يبين اسم العملية، نوع معاملاتها، ونوع الإرجاع.

السطر التالي يصف وظيفة العملية. السطر الذي يليه يشرح المعاملات والقيم المعادة. كانت الشروحات مطولة في هذه الحالة، لكن الواجب أن يوافق التوثيق صيغة قياسية. السطر الأخير يشرح أية استثناءات، في حال وجود بعضها، يمكن أن تطلقها هذه العملية.

قد تحتاج لبعض الوقتت حتى تعتاد على هذا النوع من الوثائق، لكنها تستحق الجهد المبذول.

8.6 عملية indexOf

عملية indexOf هي نقيض العملية charAt. تأخذ charAt دليلاً وتعيد المحرف الموجود عند ذلك الدليل. أما indexOf فتأخذ محرفاً وتبحث عن الدليل الذي يظهر ذلك المحرف عنده.

تفشل charAt إذا كان الدليل يخرج عن نطاق السلسلة المحرفية، وتتسبب في استثناء. تفشل indexOf إذا لم يظهر المحرف في السلسلة أبداً، وتعيد القيمة -1.

```
String fruit = "banana";
int index = fruit.indexOf('a');
```

هذا المثال يبحث عن دليل الحرف 'a' في السلسلة. في هذه الحالة، يظهر الحرف ثلاثة مرات، لذا لن يكون واضحاً ما يجب أن تفعله indexOf هنا. وفقاً للوثائق، فإنها ستعيد دليل الظهور الأول للحرف.

لإيجاد أدلة التكرارات التالية، توجد نسخة بديلة من indexOf (لشرح هذا النوع من التحميل الزائد، انظر القسم 6.4). تأخذ تلك النسخة متحولاً ثانياً يشير إلى الموقع من السلسلة الذي يجب بدء البحث منه. إذا استدعينا

```
int index = fruit.indexOf('a', 2);
```

فستبدأ البحث عند الحرف الثاني (حرف n الأول) وستجد حرف a الثاني، الموجود عند الدليل 3. إذا تصادف وجود الحرف عند دليل البدء، يكون دليل البدء هو الجواب. وهكذا،

```
int index = fruit.indexOf('a', 5);
```

ستعيد 5.

8.7 الحلقات والعد

يعد البرنامج التالي مرات ظهور الحرف 'a' في السلسلة:

```
String fruit = "banana";
int length = fruit.length();
int count = 0;

int index = 0;
while (index < length) {
  if (fruit.charAt(index) == ’a’) {
    count = count + 1;
  }
  index = index + 1;
}
System.out.println (count);
```

هذا البرنامج يشرح مفهوماً شائعاً، يدعى **العداد (counter)**. تتم تهيئة المتغير count بالقيمة صفر، ثم نزيده بمقدار واحد في كل مرة نجد فيها 'a' (في اللغة الإنكليزية يعبر الفعل **increment** لوحده عن الزيادة بمقدار واحد؛ وهو عكس الفعل **decrement**، ولا علاقة لهما بالاسم excrement). عندما نخرج من الحلقة، يحتوي count على النتيجة: المجموع النهائي لعدد حروف 'a' الموجودة.

تمرين 8.3 قم بتغليف هذه الشفرة في عملية باسم countLetters، وقم بتعميمها بحيث تقبل السلسلة والمحرف كمتحولين.

ثم أعد كتابة العملية بحيث تستخدم indexOf لتحديد مواقع حروف 'a'، بدلاً من التحقق من المحارف واحداً تلو الآخر.

8.8 عوامل الزيادة والإنقاص بمقدار واحد

إن الزيادة أو الطرح بمقدار واحد هي عمليات شائعة جداً لدرجة أن Java توفر عوامل خاصة لها. يضيف العامل ++ واحد إلى القيمة الحالية لمتغير من نوع int أو char. والعامل -- يطرح واحد منها. لا تعمل أي من العمليتين على المتغيرات من نوع double أو boolean أو String.

تقنياً، من المشروع زيادة متغير بمقدار واحد واستعماله في عبارة ما في نفس الوقت. مثلاً، قد ترى شيئاً مثل هذا:

```
System.out.println (i++);
```

بالنظر إلى هذا، لن يكون واضحاً ما إذا كانت الزيادة ستتم قبل طباعة القيمة أو بعدها. وبما أن العبارات المشابهة لهذه تميل للعصيان على الفهم، سأحاول منعك من استخدامها. في الواقع، لأثبط همتك أكثر، لن أقول لك النتيجة. إذا أردت أن تعرف حقاً، يمكنك تجربتها.

يمكننا إعادة كتابة عداد الحروف، باستخدام عوامل الزيادة بمقدار واحد:

```
int index = 0;
while (index < length) {
  if (fruit.charAt(index) == ’a’) {
    count++;
  }
  index++;
}
```

من الأخطاء الشائعة كتابة شيء مثل

```
index = index++;    // WRONG!!
```

لسوء الحظ، هذه العبارة صحيحة لغوياً، لذا فإن المترجم لن يحذرك. إن تأثير هذه التعليمة هو ترك قيمة index بدون تغيير. هذه الغلطة غالباً ما تكون صعبة الاكتشاف.

تذكر، يمكنك كتابة index = index+1;، أو كتابة index++; لكن لا يمكنك خلطهما معاً.

8.9 السلاسل المحرفية غير قابلة للتغيير

عندما تنظر إلى توثيق عمليات السلاسل المحرفية، قد تلاحظ العمليتين toUpperCase وtoLowerCase. غالباً ما تكون هاتان العمليتان مصدراً للحيرة، لأنك قد تشعر أنهما ستغيران السلسلة المعطاة. في الحقيقة، ولا واحدة من هاتين العمليتين أو أي عملية أخرى يمكن لها أن تغير سلسلة محرفية، لأن السلاسل المحرفية **غير قابلة للتحويل (immutable)**.

عندما تستدعي toUpperCase على سلسلة ما، ستحصل على سلسلة جديدة كقيمة معادة. مثلاً:

```
String name = "Alan Turing";
String upperName = name.toUpperCase ();
```

بعد تنفيذ السطر الثاني، ستحتوي السلسلة upperName على القيمة "ALAN TURING"، في حين أن السلسلة name ستحافظ على قيمتها "Alan Turing".

8.10 السلاسل المحرفية غير قابلة للمقارنة

من الضروري في كثير من الأحيان أن نقارن بين سلسلتين لنرى إذا كانتا متطابقتين أو لنرى أي واحدة تسبق الأخرى حسب الترتيب الأبجدي. سيكون لطيفاً لو أمكننا استخدام عوامل المقارنة، مثل == و <، لكن ذلك غير ممكن.

حتى نقارن بين سلسلتين، علينا استخدام عمليتي equals وcompareTo. مثلاً:

```
String name1 = "Alan Turing";
String name2 = "Ada Lovelace";

if (name1.equals (name2)) {
   System.out.println ("The names are the same.");
}

int flag = name1.compareTo (name2);
if (flag == 0) {
   System.out.println ("The names are the same.");
} else if (flag < 0) {
   System.out.println ("name1 comes before name2.");
} else if (flag > 0) {
   System.out.println ("name2 comes before name1.");
}
```

تبدو بنية التعليمات غريبة قليلاً هنا. للمقارنة بين شيئين، عليك استدعاء عملية على أحدهما وتمرير الآخر كمتحول.

القيمة المعادة من equals واضحة بشكل كاف؛ true إذا احتوت السلسلتين على نفس المحارف، وfalse ما عدا ذلك.

القيمة المعادة من compareTo غريبة قليلاً. إن الفرق بين المحارف الأولى في السلسلتين هو الذي يختلف. إذا كانت السلسلتان متساويتين، يكون الفرق 0. إذا كانت السلسلة الأولى (التي تم استدعاء العملية عليها) تسبق الثانية في الأبجدية، يكون الفرق سالباً. وإلا، فسيكون الفرق موجباً. في هذه الحالة سيكون الفرق 8 موجب، لأن الحرف الثاني من "Ada" يأتي قبل الحرف الثاني من "Alan" بثمانية حروف.

ولنبلغ حد الكمال، علينا أن نعترف بأن استخدام عامل == مع السلاسل المحرفية مشروع، إلا أنه نادراً ما يكون صحيحاً. سأشرح السبب في القسم 13.4؛ أما الآن، فلا تستخدمه.

8.11 المصطلحات

كائن: مجموعة من البيانات المترابطة التي ترفق بمجموعة من العمليات التي تشتغل عليها. كانت الكائنات التي استخدمناها حتى الآن هي String، Bug، Rock، وبقية كائنات GridWorld.

دليل: متغير أو قيمة تستخدم لاختيار أحد عناصر مجموعة مرتبة، مثل محرف من سلسلة محرفية.

الاستثناء: خطأ تشغيل.

إطلاق الاستثناء: التسبب بحدوثه.

سجل المكدس: تقرير يظهر حالة البرنامج عند حدوث استثناء.

نموذج أولي: السطر الأول من العملية، الذي يعرف اسمها، ومعاملاتها، ونوع الإرجاع.

الاجتياز: هو المرور على جميع عناصر مجموعة وتنفيذ إجراء مشابه على كل منها.

عداد: متغير يستخدم لعد شيء ما، عادة يتم تهيئته بالصفر ثم تتم زيادته.

الزيادة بمقدار واحد: زيادة قيمة المتغير بقدار واحد. عامل الزيادة في Java هو ++.

الإنقاص بمقدار واحد: طرح واحد من قيمة المتغير. عامل الإنقاص بمقدار واحد في Java هو --.

object: A collection of related data that comes with a set of methods that operate on it. The only objects we have used so far are Strings.

index: A variable or value used to select one of the members of an ordered set, like a character from a string.

exception: A run-time error.

throw: Cause an exception.

stack trace: A report that shows the state of a program when an exception occurs.

prototype: The first line of a method, which specifies the name, parameters and return type.

traverse: To iterate through all the elements of a set performing a similar operation on each.

counter: A variable used to count something, usually initialized to zero and then incremented.

increment: Increase the value of a variable by one. The increment operator in Java is ++.

decrement: Decrease the value of a variable by one. The decrement operator in Java is --.

8.12 تمرينات

تمرين 8.4 الغرض من هذا التمرين هو مراجعة التغليف والتعميم.

a. غلف كسرة الشفرة التالية، لتحويلها إلى عملية تأخذ سلسلة محرفية كمعامل وتعيد قيمة count النهائية.

b. في جملة واحدة أو اثنتين، اشرح ما تفعله العملية الناتجة بصورة مجردة (بدون الدخول في تفاصيل كيفية العمل).

c. بفرض أنك عممت هذه العملية بحيث تعمل على أية سلسلة محرفية، ماذا يمكنك أن تفعل لتعميمها أكثر؟

```
String s = "((3 + 7) * 2)";
int len = s.length ();

int i = 0;
int count = 0;

  while (i < len) {
    char c = s.charAt(i);
    if (c == '(') {
      count = count + 1;
    } else if (c == ')') {
      count = count - 1;
    }
    i = i + 1;
  }

System.out.println (count);
```

تمرين 8.5

الغرض من هذا التمرين هو استكشاف أنواع المتغيرات في Java وتغطية بعض التفاصيل التي لم يغطها هذا الفصل.

a. أنشئ برنامجاً جديداً باسم Test.java واكتب عملية main تحتوي على عدة عبارات تجمع بين أنواع مختلفة باستخدام عامل +. مثلاً، ماذا يحدث إذا "جمعت" String مع char؟ هل يتم تنفيذ عملية الجمع أم ربط السلاسل؟ ما هو نوع النتيجة؟ (كيف يمكنك تحديد نوع النتيجة؟)

b. اصنع نسخة مكبرة من الجدول التالي واملأها. عند تقاطع كل زوج من الأنواع، عليك أن تبين ما إذا كان استخدام عامل + مشروعاً مع هذه الأنواع، ما هي العملية التي سينفذها (جمع أو ربط)، وما هو نوع النتيجة.

	boolean	char	int	String
boolean				
char				
int				
String				

c. فكر ببعض القرارات التي اتخذها مصممو لغة Java عندما ملؤوا هذا الجدول. كم عنصراً يبدو معقولاً، بحيث لا توجد أية خيارات معقولة أخرى؟ كم واحداً يبدو اختياراً عشوائياً من بين عدة احتمالات معقولة؟ كم واحداً من هذه العناصر يبدو غبياً؟

d. إليك حزورة: عادة، تعليمة ++x مطابقة تماماً لتعليمة x = x + 1. إلا إذا كان x من نوع char ! في تلك الحالة، ++x مشروعة، لكن x = x + 1 تسبب خطأ. جربها وانظر إلى رسالة الخطأ التي ستحصل عليها، ثم انظر إذا كنت ستقدر على معرفة ماذا يجري.

تمرين 8.6

ما هو خرج هذا البرنامج؟ صِف، في جملة واحدة ما تفعله العملية bing (ليس كيفية عملها).

```
public static String bing (String s) {
  int i = s.length() - 1;
  String total = "";
  while (i >= 0 ) {
    char ch = s.charAt (i);
    System.out.println (i + " " + ch);
    total = total + ch;
    i--;
  }
    return total;
  }

  public static void main (String[] args) {
    System.out.println (bing ("Allen"));
  }
}
```

تمرين 8.7

أعطاك أحد أصدقاءك العملية التالية ويقول أنه إذا كان number يحتوي على أية عدد مؤلف من خانتين، فإن البرنامج سيطبع الرقم بالمقلوب. هو يدعي أنه في حال كانت قيمة number تساوي 17، ستطبع العملية 71.

هل هو محق؟ إذا لم يكن كذلك، اشرح ما يفعله البرنامج فعلاً وقم بتعديله لينفذ الإجراء الصحيح.

```
int number = 17;
int lastDigit = number%10;
int firstDigit = number/10;
System.out.println (lastDigit + firstDigit);
```

تمرين 8.8 ما هو خرج هذا البرنامج؟

```
public class Enigma {

  public static void enigma(int x) {
    if (x == 0) {
      return;
    } else {
      enigma(x/2);
    }
    System.out.print (x%2);
  }

  public static void main (String[] args) {
    enigma(5);
    System.out.println ("");
```

```
    }
}
```

اشرح في 4-5 كلمات ما تفعله العملية `enigma` فعلاً.

تمرين 8.9

a. أنشئ برنامجاً جديداً باسم Palindrome.java.
b. اكتب عملية باسم first تأخذ سلسلة محرفية وتعيد الحرف الأول، وأخرى باسم last تعيد الحرف الأخير.
c. اكتب عملية باسم middle تأخذ سلسلة محرفية وتعيد سلسلة فرعية تحتوي كل شيء عدا الحرفين الأول والأخير.
مساعدة: اقرأ توثيق العملية substring الموجودة في صنف String. قم ببعض الاختبارات لتتأكد أنك تفهم آلية عمل substring قبل أن تحاول كتابة middle.
ماذا يحدث لو استدعيت middle على سلسلة مؤلفة من حرفين فقط؟ حرف واحد؟ أو سلسلة لا تحوي أية حروف؟
d. إن التعريف المعتاد للكلمة المتناظرة (palindrome) هو كلمة يمكن قراءتها بشكل صحيح بكلا الاتجاهين، مثل "deed" و "level". لتعريف صفة مثل هذه بطريقة مختلفة يجب تحديد طريقة لاختبار وجود الصفة.
مثلاً، يمكننا القول أن "الحرف المفرد هو كلمة متناظرة، وأن أي كلمة مؤلفة من حرفين تكون متناظرة إذا كان الحرفين متطابقين، وأن أي كلمة أخرى تكون متناظرة في حال تطابق الحرفين الأول والأخير وكان وسط الكلمة متناظراً أيضاً".
اكتب عملية تعاودية اسمها isPalindrome تأخذ سلسلة محرفية وتعيد قيمة بوليانية تبين فيما لو الكلمة متناظرة أو لا.
e. بعد أن يعمل فاحص الكلمات المتناظرة الذي كتبته، ابحث عن طرق لتبسيطه من خلال تخفيض عدد الشروط التي يتحقق منها. مساعدة: قد يكون من المفيد تبني فكرة أن السلسلة الفارغة تعتبر كلمة متناظرة.
f. على ورقة، قم بابتكار استراتيجية للتحقق من تناظر الكلمات باستخدام التكرار. توجد عدة أساليب ممكنة، لذا تأكد من امتلاك خطة محكمة قبل البدء بكتابة الشفرة.
g. نفذ استراتيجيتك في عملية باسم `isPalindromeIter`.
h. اختياري: يوجد في الملحق B شفرة تقرأ قائمة كلمات من ملف. اقرأ قائمة كلمات واطبع المتناظرة منها.

تمرين 8.10 يقال عن كلمة أنها "أبجدية - abecedarian" إذا كانت حروف الكلمة تظهر وفق الترتيب الأبجدي. مثلاً، كافة الكلمات التالية هي كلمات إنكليزية أبجدية مؤلفة من 6 حروف.

abdest, acknow, acorsy, adempt, adipsy, agnosy, befist, behint, beknow, bijoux, biopsy, cestuy, chintz, deflux, dehors, dehort, deinos, diluvy, dimpsy

a. صف خوارزمية للتحقق من كلمة معطاة (سلسلة محرفية) فيما إذا كانت أبجدية أو لا، بفرض أن الكلمة مكتوبة بالأحرف الصغيرة فقط. يمكن لخوارزميتك أن تكون تكرارية أوتعاودية.
b. نفذ خوارزميتك في عملية اسمها isAbecedarian.

تمرين 8.11 dupledrome هي كلمة تحتوي على أحرف مضاعفة فقط، مثل "llaammaa" أو "ssaabb". أنا أظن بأنه لا توجد أي dupledromes في اللغة الإنكليزية المعتادة. لاختبار ذلك التخمين، أنا أرغب ببرنامج يقرأ الكلمات من القاموس واحدة تلو لأخرى ويفحصها للتحقق من كونها dupledrome أو لا.

اكتب عملية باسم isDupledrome تأخذ سلسلة محرفية وتعيد قيمة بوليانية تبين فيما إذا كانت الكلمة dupledrome أو لا.

تمرين 8.12

a. تعمل حلقة فك شفرة الكابتن كرنش (Captain Crunch) بأخذ كل حرف في سلسلة محرفية وإضافة 13 إليه. مثلاً، 'a' سيصبح 'n' و'b' سيصبح 'o'. ستلتف الحروف عندما تصل للنهاية، لذا فإن 'z' سيصبح 'm'.
اكتب عملية تأخذ سلسلة محرفية وتعيد سلسلة جديدة تحتوي على النسخة المشفرة. عليك أن تفترض أن السلسلة تحتوي على حروف كبيرة وصغيرة، وفراغات، لكنها لن تحوي أية علامات ترقيم أخرى. يجب تحويل الحروف الصغيرة إلى حروف صغيرة أخرى؛ والكبيرة إلى كبيرة. عليك عدم تشفير المسافات.

b. عمم عملية كابتن كرنش حتى تضيف كمية معطاة إلى الحروف بدلاً من الرقم 13. الآن يجب أن تكون قادراً على تشفير الأشياء بإضافة 13 وفك التشفير بإضافة 13-. جربها.

تمرين 8.13 إذا حللت تمارين GridWorld من الفصل 5، فقد تستمتع بهذا التمرين. الهدف هو استخدام قوانين المثلثات لجعل الحشرات تطرد بعضها.

اصنع نسخة من BugRunner.java باسم ChaseRunner.java واستوردها إلى بيئة البرمجة عندك. قبل عمل أي شيء، تأكد أنك قادر على تجميعه وتشغيله.

- اصنع حشرتين، حمراء وزرقاء.
- اكتب عملية باسم distance تأخذ حشرتين وتحسب المسافة بينهما. تذكر أنك تستطيع الحصول على إحداثي السينات (x-coordinate) للحشرة كما يلي:

```
int x = bug.getLocation().getCol();
```

- اكتب عملية باسم turnToward تأخذ حشرتين وتدوّر إحداهما لتواجه الأخرى. مساعدة: استخدم Math.atan2، لكن تذكر أن النتيجة ستكون بالراديان، وعليك التحويل إلى الدرجات. أيضاً، بالنسبة للحشرات، 0 درجة تشير إلى الشمال، وليس الشرق.
- اكتب عملية باسم moveToward تأخذ حشرتين، تدوّر الأولى لتواجه الثانية، ثم تحرك الأولى، إذا كانت قادرة على الحركة.
- اكتب عملية باسم moveBugs تأخذ حشرتين وعدد صحيح n، وتحرك كل حشرة باتجاه الأخرى n حركة. يمكنك كتابة هذه العملية تعاودياً، أو استخدام حلقة while.
- اختبر كل من عملياتك أثناء تطويرها. عندما تعمل كل العمليات بشكل صحيح، ابحث عن طرق لتحسينها. مثلاً، إذا كنت تملك شفرة فائضة في distance و turnToward، يمكنك تغليف الشفرة المكررة في عملية.

الفصل 9

الكائنات القابلة للتعديل

9.1 Points وRectangles

بالرغم من أن السلاسل المحرفية كائنات، إلا أنها ليست مثيرة للاهتمام حقاً، وذلك لأنها

- غير قابلة للتغيير.
- لا تملك متغيرات حالة.
- لست مضطراً لاستعمال أمر new لإنشاء واحد.

في هذا الفصل، سنستخدم نوعين جديدين من الكائنات الذين يشكلان جزءاً من لغة Java، Point (نقطة) وRectangle (مستطيل). من البداية، أود توضيح أن هذه النقط والمستطيلات ليست كائنات رسومية تظهر على الشاشة. بل هي متغيرات تحتوي على معطيات، مثل المتغيرات من نوع int وdouble تماماً. ومثل المتغيرات الأخرى، يتم استخدام هذه الكائنات داخلياً لإنجاز الحسابات.

9.2 الحزم

إن الأصناف المبنية مسبقاً في Java مقسمة إلى عدد من **الحزم (packages)**، من بينها java.lang، التي تحتوي على جميع الأصناف التي شاهدناها حتى الآن تقريباً، وjava.awt، التي تحتوي على الأصناف الخاصة **بأدوات النوافذ المجردة** (AWT) **(Java Abstract Window Toolkit)**، التي تحتوي على أصناف النوافذ، والأزرار، الرسومات، الخ.

لاستعمال حزمة ما، عليك **استيرادها (import)**. إن الصنفين Point وRectangle موجودين في حزمة java.awt، لذلك سيتوجب علينا استيرادها، كما يلي:

```
import java.awt.Point;
import java.awt.Rectangle;
```

كافة تعليمات الاستيراد تظهر في بداية البرنامج، خارج تعريف الصنف.

يتم استيراد الأصناف الموجودة في java.lang، مثل Math وString، تلقائياً. ولذلك لم تكن هناك حاجة لاستخدام تعليمة الاستيراد import حتى الآن.

9.3 كائنات Point

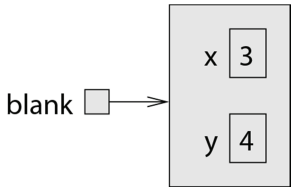
النقطة هي رقمين (إحداثيين) نعاملهما معاً على أنهما كائن واحد. في التدوين الرياضي، تتم كتابة النقط عادة بين قوسين، مع فاصلة واحدة تفصل الإحداثيين. مثلاً، (0,0) تعبر عن مبدأ الإحداثيات، و(x,y) تعبر عن النقطة الموجودة على بعد x وحدة قياس إلى يمين المبدأ وy وحدة قياس فوقه.

في Java، يتم تمثيل نقطة بكائن Point. لإنشاء نقطة جديدة، عليك استعمال أمر new:

```
Point blank;
blank = new Point (3, 4);
```

السطر الأول هو تصريح عادي عن متحول: اسمه blank وهو من نوع Point. أما السطر الثاني فيبدو غريباً نوعاً ما: يستدعي السطر الثاني الأمر new، ويحدد نوع الكائن الجديد، ويعطيه المتحولات. تلك المتحولات هي إحداثيات النقطة الجديدة، (3, 4).

إن ناتج الأمر new هو **مرجع (reference)** إلى النقطة الجديدة، إذن فالمتغير blank يحتوي على مرجع إلى كائن حديث الولادة. توجد طريقة قياسية للتعبير عن تعليمة الإسناد هذه بشكل بياني، مبينة في الشكل.



كالعادة يظهر اسم المتغير blank خارج الصندوق وتظهر قيمته بداخل الصندوق. في هذه الحالة، قيمة المتغير هي مرجع، الممثل بيانياً بسهم. يشير السهم إلى الكائن الذي نشير إليه.

يمثل الصندوق الكبير الكائن المنشأ حديثاً والقيمتين الموجودتين داخله. x وy هما اسمي **متغيرات الحالة (instance variables)**.

إن كافة المتغيرات، القيم، والكائنات معاً تدعى **بالحالة (state)**. المخططات التي تبين حالة البرنامج تدعى **بمخططات الحالة (state diagrams)**. أثناء عمل البرنامج، تتغير حالته، عليك التعامل مع مخطط الحالة على أنه صورة لمرحلة معينة من تنفيذ البرنامج.

9.4 متغيرات الحالة

إن أجزاء البيانات التي تشكل الكائن تدعى بمتغيرات الحالة لأن كل كائن، والذي يشكل **حالة (instance)** من نوعه، يملك نسخته الخاصة من المتغيرات.

هذا يشبه علبة القفازات في سيارة. كل سيارة هي حالة معينة من النوع "سيارة"، ولكل سيارة علبة قفازات خاصة بها. إذا طلبت مني إحضار شيء من علبة قفازات سيارتك، فعليك أن تخبرني أي واحدة هي سيارتك.

بشكل مشابه، إذا أردت قراءة قيمة من متغير حالة، عليك تحديد الكائن الذي ترغب بالحصول على القيمة منه. يتم عمل ذلك في Java باستخدام "النقطة – dot notation".

```
int x = blank.x;
```

تعني العبارة blank.x "اذهب إلى الكائن الذي يشير إليه المتغير blank، واحصل على القيمة المخزنة في x". في هذه الحالة قمنا بإسناد تلك القيمة إلى متغير محلي اسمه x. لاحظ عدم وجود تعارض بين المتغير المحلي x وبين متغير الحالة x. إن الغرض من كتابة النقطة هو تعريف المتغير (متغير الحالة) الذي تشير إليه بشكل واضح.

يمكنك استعمال النقطة في أي جزء من العبارات في Java، لذا فإن ما يلي مشروع.

```
System.out.println (blank.x + ", " + blank.y);
int distance = blank.x * blank.x + blank.y * blank.y;
```

يطبع السطر الأول 3, 4؛ ويحسب السطر الثاني القيمة 25.

9.5 استخدام الكائنات كمعاملات

يمكنك تمرير الكائنات كمعاملات باستخدام الطريقة المعتادة. مثلاً

```
public static void printPoint (Point p) {
   System.out.println ("(" + p.x + ", " + p.y + ")");
}
```

هي عملية تأخذ نقطة كمتحول وتطبعها بالصيغة القياسية. إذا استدعيت printPoint (blank)، فستطبع (3, 4). في الحقيقة، توجد عملية جاهزة في Java لطباعة النقط. إذا استدعيت System.out.println (blank)، فستحصل على

```
java.awt.Point[x=3,y=4]
```

هذا هو التنسيق القياسي الذي تستخدمه Java لطباعة الكائنات. تطبع اسم نوع الكائن، متبوعاً بمحتوياته، بما في ذلك أسماء متغيرات الحالة وقيمها.

كمثال آخر، يمكننا إعادة كتابة العملية distance من القسم 6.2 بحيث تأخذ نقطتين كمعاملات بدلاً من أربع أعداد عشرية.

```
public static double distance (Point p1, Point p2) {
  double dx = (double)(p2.x - p1.x);
  double dy = (double)(p2.y - p1.y);
  return Math.sqrt (dx*dx + dy*dy);
}
```

إن قولبة الأنماط ليست ضرورية هنا؛ لقد وضعتها لأذكرك فقط بأن متغيرات الحالة في Point هي أعداد صحيحة.

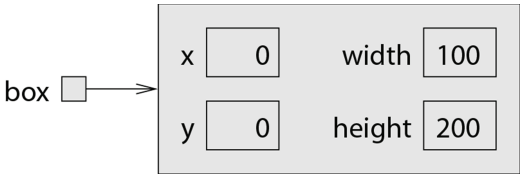
9.6 المستطيلات

تشبه المستطيلات النقط، عدا أنها تملك أربعة متغيرات حالة، أسماؤها x, y, width, height. فيما عدا ذلك، فكل شيء آخر هو نفسه تقريباً.

المثال التالي ينشئ كائناً جديداً من نوع Rectangle ويجعل المتغير box يشير إليه.

```
Rectangle box = new Rectangle (0, 0, 100, 200);
```

يبين هذا الشكل تأثير تعليمة الإسناد هذه.



إذا طبعت box، ستحصل على

```
java.awt.Rectangle[x=0,y=0,width=100,height=200]
```

ثانيةً، هذه هي نتيجة العملية الجاهزة في Java التي تعرف كيفية طباعة كائنات Rectangle.

9.7 استخدام الكائنات كنوع إرجاع

يمكنك كتابة عملية ترجع كائناً. مثلاً، findCenter تأخذ مستطيلاً كمتحول وترجع نقطة تحتوي على إحداثيات مركز المستطيل:

```
public static Point findCenter (Rectangle box) {
  int x = box.x + box.width/2;
  int y = box.y + box.height/2;
  return new Point (x, y);
}
```

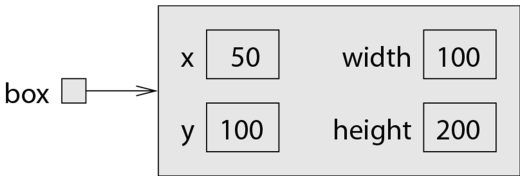
لاحظ أنك تستطيع استعمال new لإنشاء كائن جديد، واستخدام النتيجة مباشرة كقيمة معادة.

9.8 الكائنات قابلة للتغيير

يمكنك تغيير محتويات كائن بعمل إسناد لأحد متغيرات الحالة الخاصة به. مثلاً، "لتحريك" مستطيل بدون تغيير حجمه، يمكنك تغيير قيم x وy:

```
box.x = box.x + 50;
box.y = box.y + 100;
```

تظهر النتيجة في الشكل:



يمكننا أخذ هذه الشفرة وتغليفها في عملية، وتعميمها لتحريك المستطيل بأية قيمة:

```
public static void moveRect (Rectangle box, int dx, int dy) {
  box.x = box.x + dx;
  box.y = box.y + dy;
}
```

يشير المتغيران dx وdy إلى المسافة المطلوب تحريك المستطيل إليها في كل اتجاه. إن استدعاء هذه العملية يغير المستطيل المرر إليها كمتحول.

```
Rectangle box = new Rectangle (0, 0, 100, 200);
moveRect (box, 50, 100);
System.out.println (box);
```

تطبع java.awt.Rectangle[x=50,y=100,width=100,height=200].

يمكن أن يكون تعديل الكائنات بتمريرها إلى عملية مفيداً، لكنه يجعل من تنقيح البرنامج عملية صعبة لأنه ليس من الواضح دائماً متى تعدل استدعاءات للعمليات على متحولاتها أو لا تعدل. فيما بعد، سأناقش بعض مزايا ومساوئ أسلوب البرمجة هذا.

توفر Java عمليات تشتغل على النقاط (كائنات Points) والمستطيلات (كائنات Rectangle). يمكنك قراءة الوثائق على http://download.oracle.com/javase/6/docs/api/java/awt/Rectangle.html

مثلاً، translate، التي تنفذ نفس وظيفة moveRect تماماً، لكن بدلاً من تمرير المستطيل كمتحول، نستخدم النقطة:

```
box.translate (50, 100);
```

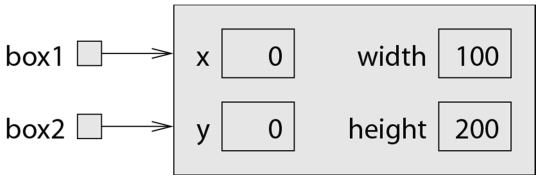
النتيجة هي نفسها تماماً.

9.9 تعدد الأسماء

تذكر أنك عندما تسند شيئاً إلى متغير كائني، فأنت تسند مرجعاً إلى كائن. من الممكن وجود عدة متغيرات تشير إلى الكائن نفسه. مثلاً، هذه الشفرة:

```
Rectangle box1 = new Rectangle (0, 0, 100, 200);
Rectangle box2 = box1;
```

تولد مخطط حالة يبدو كهذا:

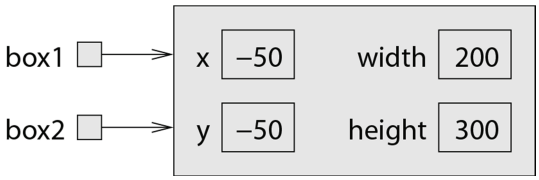


كلا المتغيرين box1 وbox2 يشيران إلى الكائن نفسه. بكلمات أخرى، هذا الكائن له اسمين، box1 وbox2. عندما يستخدم شخص ما اسمين، يدعى ذلك **aliasing**. نفس الشيء مع الكائنات.

عندما يشير متغيران إلى كائن واحد، فإن أي تغييرات تؤثر على أحدهما ستؤثر على الآخر أيضاً. مثلاً:

```
System.out.println (box2.width);
box1.grow (50, 50);
System.out.println (box2.width);
```

يطبع السطر الأول القيمة 100، وهي عرض المستطيل المشار إليه بالمتغير box2. يستدعي السطر الثاني عملية grow على box1، التي توسع المستطيل بمقدار 50 بكسل في كل جهة (انظر في الوثائق للمزيد من التفاصيل). تم تمثيل أثر ذلك في الشكل:



يجب أن يتضح من الشكل، أن أية تعديلات تجري على box1 ستطبق على box2 أيضاً. بالتالي، ستكون القيمة المطبوعة في السطر الثالث 200، عرض المستطيل بعد التوسيع. (من جهة أخرى، من المشروع تماماً أن تكون إحداثيات المستطيل سالبة).

كما يتبين من هذا المثال البسيط، يمكن للشفرة التي تشتمل على تعدد الأسماء أن تصبح مربكة بسرعة، ويمكن لها أن تصبح صعبة التنقيح للغاية. بشكل عام، يجب تفادي الأسماء المتعددة أو استعمالها بحذر.

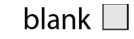
9.10 العدم

عندما تنشئ متغير كائني، تذكر أنك تنشئ مرجعاً لكائن. قبل أن تجعل المتغير يشير إلى كائن، تكون قيمة المتغير null. null هي قيمة خاصة في Java (وهي كلمة مفتاحية أيضاً) تستعمل لتعبر عن "عدم وجود كائن".

إن التصريح `Point blank;` مماثل لهذه التهيئة

```
Point blank = null;
```

يوضح مخطط الحالة التالي أثر هذه التعليمة:



تمثل القيمة null بمربع صغير بدون سهم.

إذا حاولت استعمال كائن معدوم (null object)، سواء بمحاولة الوصول إلى متغير حالة أو استدعاء عملية، فستحصل على NullPointerException. سيطبع النظام رسالة خطأ وينهي البرنامج.

```
Point blank = null;
int x = blank.x;                // NullPointerException
blank.translate (50, 50);       // NullPointerException
```

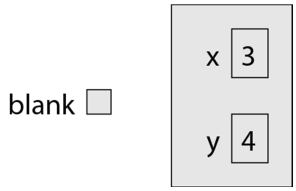
من جهة أخرى، من المسموح تمرير كائن معدوم كمتحول أو استقباله كقيمة معادة. في الواقع، من الشائع عمل مثل ذلك، مثلاً للتعبير عن مجموعة خالية أو للإشارة إلى حالة خطأ.

9.11 جمع القمامة

في القسم 9.9 تحدثنا عما يحدث عندما يشير أكثر من متغير إلى نفس الكائن. ماذا يحدث عندما لا يشير أي متغير إلى الكائن؟ مثلاً:

```
Point blank = new Point (3, 4);
blank = null;
```

ينشئ السطر الأول كائناً جديداً من الصنف Point ويجعل المتغير blank يشير إليه. يعدل السطر الثاني المتغير blank فبدلاً من الإشارة إلى الكائن، لا يشير إلى شيء (الكائن المعدوم).



إذا لم يشر أحد إلى الكائن، فلن يستطيع أحد قراءة أو كتابة أي من قيمه، أو استدعاء عملية عليه. نتيجة لذلك، سيبقى الكائن محاصراً حتى الخروج من البرنامج. يمكننا ترك الكائن في الذاكرة لكنه سيهدر المساحة وحسب، لذلك يبحث نظام Java عن الكائنات المعزولة بشكل دوري أثناء عمل البرنامج، ويتخلص منهم لاستعادة المساحة التخزينية، في عملية تدعى **جمع القمامة (garbage collection)**. بعد ذلك، تصبح المساحة التخزينية التي شغلها الكائن متوفرة للاستعمال كجزء من كائن جديد.

لا توجد حاجة لعمل أي شيء حتى تجعل عملية جمع القمامة تبدأ، وبشكل عام لن تشعر بتلك العملية أبداً.

9.12 الأنواع الكائنية والأنواع البسيطة

يوجد نمطين للأنواع في Java، الأنواع البسيطة والأنواع الكائنية. الأنواع البسيطة (أو البدائية - primitive)، مثل int وboolean تبدأ بحروف صغيرة؛ الأنواع الكائنية تبدأ بحروف كبيرة. هذا التمييز مفيد لأنه يذكرنا ببعض الاختلافات بين هذين النمطين:

- عند التصريح عن متغير بسيط، تحصل على منطقة تخزينية لقيمة بسيطة. عندما تصرح عن متغير كائني، تحصل على مساحة لمرجع يشير إلى كائن. للحصول على مساحة للكائن نفسه، يجب استعمال الأمر new.
- إذا لم تهيئ متغيراً من نوع بسيط، فسيعطى قيمة افتراضية تختلف حسب النوع. مثلاً، 0 للأعداد الصحيحة وtrue للمتغيرات البوليانية. القيمة الافتراضية للأنواع الكائنية هي null، التي تشير لعدم وجود كائن.
- المتغيرات البسيطة معزولة بشكل تام ما يعني عدم وجود أي شيء يمكنك عمله في عملية ما ثم يؤثر على متغير في عملية أخرى. قد يكون العمل مع المتغيرات الكائنية صعباً قليلاً لأنها غير معزولة بشكل كامل. إذا مررت مرجعاً يشير إلى كائن كمتحول، فقد تعدّل العملية التي استدعيتها على الكائن، وفي تلك الحالة سترى أثر ذلك. نفس الشيء يصح عندما تستدعي عملية على كائن. طبعاً، يمكن لذلك أن يكون شيئاً جيداً، لكن يجب أن تكون واعياً له.

يوجد اختلاف آخر بين الأنواع البسيطة والكائنية. لا يمكنك إضافة أنواع بسيطة جديدة إلى لغة Java (إلا إذا كنت عضواً في لجنة معايير اللغة القياسية)، لكن يمكنك إنشاء أنواع كائنية جديدة! سنرى كيفية عمل ذلك في الفصل التالي.

9.13 المصطلحات

حزمة: مجموعة من الأصناف. يتم تنظيم الأصناف الجاهزة في Java ضمن حزم.

AWT: أدوات النوافذ المجردة، أحد أكبر حزم Java وأكثرها استخداماً.

الحالة (instance): مثال عن فئة ما. قطتي هي حالة من فئة "السنوريات". كل كائن يمثل حالة عن صنف معين.

متغير الحالة: أحد عناصر البيانات المسماة التي تكون الكائن. كل كائن (حالة) له نسخة خاصة من متغيرات الحالة المعرفة في صنفه.

مرجع: قيمة تشير إلى كائن. في مخططات الحالة، يمثل المرجع بسهم.

تعدد الأسماء: الحالة التي يشير فيها متغيرين أو أكثر إلى نفس الكائن.

جمع القمامة: عملية البحث عن كائنات لا تملك أي مرجعيات واستعادة المساحة التخزينية التي تشغلها.

الحالة (state): وصف كامل لكافة المتغيرات والكائنات وقيمها، عند نقطة معينة من تنفيذ البرنامج.

مخطط الحالة: صورة لحالة البرنامج موضحة بيانياً.

package: A collection of classes. The built-in Java classes are organized in packages.

AWT: The Abstract Window Toolkit, one of the biggest and most commonly-used Java packages.

instance: An example from a category. My cat is an instance of the category “feline things.” Every object is an instance of some class.

instance variable: One of the named data items that make up an object. Each object (instance) has its own copy of the instance variables for its class.

reference: A value that indicates an object. In a state diagram, a reference appears as an arrow.

aliasing: The condition when two or more variables refer to the same object.

garbage collection: The process of finding objects that have no references and reclaiming their storage space.

state: A complete description of all the variables and objects and their values, at a given point during the execution of a program.

state diagram: A snapshot of the state of a program, shown graphically.

9.14 تمرينات

تمرين 9.1

a. ارسم مخططاً هرمياً للبرنامج التالي، يبين المتغيرات المحلية ومعاملات العمليتين main وriddle، وبين أية كائنات تشير إليها هذه المتغيرات.
b. ما هو خرج هذا البرنامج؟

```
public static void main (String[] args)
{
  int x = 5;
  Point blank = new Point (1, 2);

  System.out.println (riddle(x, blank));
  System.out.println (x);
  System.out.println (blank.x);
  System.out.println (blank.y);
}

public static int riddle(int x, Point p)
{
  x = x + 7;
  return x + p.x + p.y;
}
```

إن الغرض من هذا التمرين هو التأكد من فهمك لآلية تمرير الكائنات كمعاملات.

تمرين 9.2

a. ارسم مخططاً هرمياً للبرنامج التالي، يبين حالة البرنامج قبيل عودة العملية distance. قم بتضمين كافة المتغيرات والمعاملات والكائنات التي تشير إليها هذه المتغيرات.
b. ما هو خرج هذا البرنامج؟

```
public static double distance (Point p1, Point p2) {
  int dx = p1.x - p2.x;
  int dy = p1.y - p2.y;
  return Math.sqrt (dx*dx + dy*dy);
}

public static Point findCenter (Rectangle box) {
  int x = box.x + box.width/2;
  int y = box.y + box.height/2;
```

```
  return new Point (x, y);
}

public static void main (String[] args) {
  Point blank = new Point (5, 8);

  Rectangle rect = new Rectangle (0, 2, 4, 4);
  Point center = findCenter (rect);

  double dist = distance (center, blank);

  System.out.println (dist);
}
```

تمرين 9.3 العملية grow هي جزء من الصنف الجاهز Rectangle. اقرأ وثائقها على
http://download. oracle. com/javase/6/docs/api/java/awt/
Rectangle. html# grow(int,int)

ثم أجب عن الأسئلة

a. ما هو خرج البرنامج التالي؟
b. ارسم مخطط حالة يبين حالة البرنامج قبيل انتهاء main. قم بتضمين كافة المتغيرات المحلية والكائنات التي تشير إليها.
c. عند نهاية main، هل يشير المتغيرين p1 وp2 إلى الكائن نفسه؟ لماذا أو لم لا؟

```
public static void printPoint (Point p) {
   System.out.println ("(" + p.x + ", " + p.y + ")");
}

public static Point findCenter (Rectangle box) {
   int x = box.x + box.width/2;
   int y = box.y + box.height/2;
   return new Point (x, y);
}

public static void main (String[] args) {

   Rectangle box1 = new Rectangle (2, 4, 7, 9);
   Point p1 = findCenter (box1);
   printPoint (p1);

   box1.grow (1, 1);
   Point p2 = findCenter (box1);
   printPoint (p2);
}
```

تمرين 9.4 على الأغلب أنك مرضت من عملية العاملي الآن، لكننا سنعمل نسخة أخرى منها أيضاً.

a. أنشئ برنامجاً باسم Big.java وابدأ بكتابة نسخة تكرارية من factorial.
b. اطبع جدولاً للأعداد الصحيحة من 0 إلى 30 مع عاملي كل منها. عند مكان ما بالقرب من 15، ستلاحظ على الأغلب أن الإجابات لم تعد صحيحة. لم حدث ذلك؟
c. BigIntegers هي كائنات جاهزة في Java يمكنها أن تمثل أعداداً صحيحة ذات حجم اختياري. لا سقف لها سوى محدودية الذاكرة وسرعة المعالجة. اقرأ وثائق صنف BigIntegers من
http://download.oracle.com/javase/6/docs/api/java/math/BigInteger.html
d. توجد طرق عدة لإنشاء BigInteger جديد، لكنني أنصح بالطريقة التي تستخدم valueOf. الشفرة التالية تحول عدداً صحيحاً إلى BigInteger:

```
int x = 17;
BigInteger big = BigInteger.valueOf (x);
```

اكتب هذه الشفرة وجرب بعض الحالات البسيطة مثل إنشاء BigInteger وطباعته. لاحظ أن println تعرف كيف تطبع الBigIntegers! لا تنسى إضافة import java.math.BigInteger; إلى بداية برنامجك.

e. لسوء الحظ، لا يمكننا استخدام العوامل الرياضية المعتادة على الBigIntegers، لأنها ليست نوعاً بسيطاً. بل يجب علينا استخدام عمليات الكائن مثل add. لجمع كائنين BigInteger معاً، عليك استدعاء add على أحدهما وتمرير الآخر كمتحول. مثلاً:

```
BigInteger small = BigInteger.valueOf (17);
BigInteger big = BigInteger.valueOf (1700000000);
BigInteger total = small.add (big);
```

جرب بعض العمليات الأخرى، مثل multiply وpow.

f. حول factorial حتى تستخدم BigIntegers في حساباتها، ثم تعيد BigInteger كنتيجة. يمكنك ترك المعامل كما هو – سيبقى عدداً صحيحاً.

g. جرب طباعة الجدول ثانية باستخدام تابع العاملي المعدل. هل هو صحيح حتى 30؟ لأي رقم يمكنك حساب العاملي؟ أنا حسبت العاملي لكل الأعداد من 0 حتى 999، لكن جهازي بطيء فعلاً، وقد استغرقت العملية بعض الوقت. آخر رقم، !999، يتألف من 2565 خانة.

تمرين 9.5 العديد من خوارزميات التشفير تعتمد على القدرة على رفع الأعداد الصحيحة الكبيرة إلى قوة صحيحة. إليك عملية تجري خوارزمية سريعة (بشكل معقول) لحساب القوى الصحيحة:

```
public static int pow (int x, int n) {
    if (n==0) return 1;

    // find x to the n/2 recursively
    int t = pow (x, n/2);

    // if n is even, the result is t squared
    // if n is odd, the result is t squared times x

    if (n%2 == 0) {
        return t*t;
    } else {
        return t*t*x;
    }
}
```

مشكلة هذه الخوارزمية أنها فعالة فقط طالما أن النتيجة أصغر من 2 مليار. أعد كتابة العملية بحيث تكون النتيجة BigInteger. لكن يجب أن يبقى المعامل integer مع ذلك.

يمكنك استخدام عمليتي add وmultiply الخاصتين بالBigInteger، لكن لا تستعمل عملية pow الجاهزة، حتى لا تفسد المرح.

ملاحظة: إذا كنت مهتماً بالرسوميات، فيمكنك أن تقرأ الملحق A الآن، وأن تقوم بالتمرينات الموجودة فيه.

الفصل 10

GridWorld: الجزء الثاني

الجزء الثاني من حقيبة GridWorld الدراسية تستخدم مقومات لم نرها حتى الآن، لذا سنلقي نظرة سريعة الآن وستحصل على المزيد من التفاصيل لاحقاً. للتذكير فقط، يمكنك الحصول على وثائق أصناف GridWorld من http://thinklikecs.webs.com/resources/javadoc/gridworld.

عندما تنصب GridWorld، يجب أن تحصل على مجلد باسم boxBug داخل المجلد projects، يحتوي على BoxBug.java، BoxBugRunner.java، وBoxBug.gif.

انسخ هذه الملفات إلى مجلد عملك واستوردهم إلى بيئة البرمجة لديك. توجد تعليمات هنا قد تساعدك:
http://www.collegeboard.com/prod_downloads/student/testing/ap/compsci_a/ap07_gridworld_installation_guide.pdf.

(توجد نسخة مترجمة من هذه التعليمات على:
http://thinklikecs.webs.com/book/resources/gridworld_installation_guide.pdf)

هذه هي الشفرة من BoxBugRunner.java:

```
import info.gridworld.actor.ActorWorld;
import info.gridworld.grid.Location;
import java.awt.Color;

public class BoxBugRunner {
   public static void main(String[] args) {
      ActorWorld world = new ActorWorld();
      BoxBug alice = new BoxBug(6);
      alice.setColor(Color.ORANGE);
      BoxBug bob = new BoxBug(3);
      world.add(new Location(7, 8), alice);
      world.add(new Location(5, 5), bob);
      world.show();
   }
}
```

يجب أن يكون كل شيء هنا مألوفاً، ما عدا Location ربما، وهو جزء من GridWorld، يشبه java.awt.Point.

يحتوي BoxBug.java على تعريف الصنف BoxBug.

```
public class BoxBug extends Bug {
   private int steps;
   private int sideLength;

   public BoxBug(int length) {
      steps = 0;
      sideLength = length;
   }
}
```

يقول السطر الأول أنه يوسّع (extends) Bug، ما يعني أن BoxBug هو من نوع Bug.

السطرين التاليين هما متغيري حالة. لكل حشرة (كائن Bug) متغيرين باسم sideLength، يحدد حجم الصندوق الذي ترسمه، وsteps، الذي يحتفظ بعدد الخطوات التي تحركتها الحشرة.

يعرف السطر التالي **عملية بناء (أو باني constructor)**، وهي عملية خاصة تهيئ متغيرات الحالة. عندما تصنع حشرة باستدعاء new، تستدعي Java هذا الباني.

يستخدم معامل الباني لتهيئة sideLength (المسافة الجانبية).

تتحكم العملية act بسلوك الحشرة. ها هي العملية act للصنف BoxBug:

```
public void act() {
   if (steps < sideLength && canMove()) {
      move();
      steps++;
   }
   else {
      turn();
      turn();
      steps = 0;
   }
}
```

إذا كانت BoxBug (حشرة الصندوق) قادرة على الحركة، ولم تصل بعد للعدد المطلوب من الخطوات، ستتحرك وتزيد steps.

إذا ارتطمت بجدار أو أنهت أحد جوانب الصندوق، ستستدير بزاوية 90 درجة على اليمين وتعيد ضبط steps إلى 0.

شغل البرنامج وشاهد ماذا يفعل. هل حصلت على السلوك المتوقع منه؟

تمرين 10.1 الآن أصبحت تعرف ما يكفي لحل التمارين الموجودة في دليل الطالب، الجزء 2. اذهب إليها، ثم عد إلى هنا لمزيد من المرح.

10.1 النمل الأبيض

كتبتُ صنفاً باسم Termite يوسع الصنف Bug ويضيف له القدرة على التفاعل مع الأزهار.

لتشغيله، نزل هذه الملفات واستوردها إلى بيئة البرمجة لديك:

http://thinklikecs.webs.com/resources/code/Termite.java
http://thinklikecs.webs.com/resources/code/Termite.gif
http://thinklikecs.webs.com/resources/code/TermiteRunner.java

بما أن Termite يوسع Bug، كل عمليات Bug ستعمل على Termite. لكن النمل الأبيض (Termites) يملك عمليات إضافية لا تملكها الحشرات (Bugs).

```
/**
 * Returns true if the termite has a flower.
 */
public boolean hasFlower();
```

```
/**
 * Returns true if the termite is facing a flower.
 */
public boolean seeFlower();

/**
 * Creates a flower unless the termite already has one.
 */
public void createFlower();

/**
 * Drops the flower in the termites current location.
 *
 * Note: only one Actor can occupy a grid cell, so the effect of
 * dropping a flower is delayed until the termite moves.
 */
public void dropFlower();

/**
 * Throws the flower into the location the termite is facing.
 *
 */
public void throwFlower();

/**
 * Picks up the flower the termite is facing, if there is one,
 * and if the termite doesn't already have a flower.
 */
public void pickUpFlower();
```

بالنسبة لبعض العمليات، يوفر الصنف Bug تعريفاً لها ويوفر Termite لها تعريفاً آخر. في تلك الحالة، يجب أن **تهيمن (override)** عملية Termite على عملية Bug.

مثلاً، تعيد العملية Bug.canMove القيمة true إذا وجدت زهرة في الموقع التالي، لذا يمكن للحشرات أن تدوس الأزهار. أما Termite.canMove فتعيد false إذا وجد أي كائن في الموقع التالي، لذا فإن سلوك النمل الأبيض مختلف.

كمثال آخر، يملك النمل الأبيض نسخة من turn تأخذ عدداً صحيحاً من الدرجات كمعامل. أخيراً، يملك النمل الأبيض randomTurn، التي تدوره 45 درجة على اليمين أو اليسار عشوائياً.

ها هي الشفرة من TermiteRunner.java:

```
public class TermiteRunner
{

   public static void main(String[] args)
   {

   ActorWorld world = new ActorWorld();
   makeFlowers(world, 20);

   Termite alice = new Termite();
   world.add(alice);

   Termite bob = new Termite();
   bob.setColor(Color.blue);
   world.add(bob);
```

```
   world.show();
   }

   public static void makeFlowers(ActorWorld world, int n) {
      for (int i=0; i<n; i++) {
         world.add(new EternalFlower());
      }
   }
}
```

يجب أن يبدو كل شيء هنا مألوفاً. يقوم TermiteRunner بإنشاء ActorWorld فيه عشرين زهرة خارجية (ExternalFlower) ونملتين بيضاوين (two Termites).

الزهرة الخارجية هي زهرة تتجاهل العملية act حتى لا تصبح الأزهار أغمق.

```
class EternalFlower extends Flower {
   public void act() {}
}
```

إذا شغلت TermiteRunner.java يجب أن ترى نملتين تتحركان عشوائياً بين الأزهار.

يوضح MyTermite.java العمليات التي تتفاعل مع الأزهار. ها هو تعريف الصنف:

```
public class MyTermite extends Termite {

   public void act() {

      if (getGrid() == null)
         return;

      if (seeFlower()) {
         pickUpFlower();
      }

      if (hasFlower()) {
         dropFlower();
      }

      if (canMove()) {
         move();
      }

      randomTurn();
   }
}
```

MyTermite يوسع Termite، ويتجاوز العملية act. إذا رأت MyTermite زهرة، ستلتقطها، وإذا كانت تملك زهرة، ستتركها.

تمرين 10.2 الغرض من هذا التمرين هو اسكتشاف سلوك النمل الأبيض الذي يتفاعل مع الأزهار.

عدل TermiteRunner.java لإنشاء MyTermites بدلاً من Termites. بعدها شغله ثانية. يجب أن تتحرك MyTermites عشوائياً، محركة الأزهار. يجب ألا يتغير عدد الأزهار (بما في ذلك الأزهار التمي تمسكها MyTermites).

في كتاب Termites, Turtles and Traffic Jams يصف الكاتب Mitchell Resnick نمطاً بسيطاً من سلوك النمل الأبيض:

- إذا شاهدت زهرة، التقطها. إلا إذا كنت تملك زهرة من قبل؛ في تلك الحالة، ارم الزهرة التي معك.
- تحرك للأمام إذا استطعت.
- استدر يميناً أو يساراً بشكل عشوائي.

عدل MyTermite.java لتنفيذ هذا النمط. ما أثر هذا التعديل على سلوك MyTermites برأيك؟

جربه. يجب ألا يتغير عدد الأزهار أيضاً، لكن مع الوقت ستتكدس الأزهار في عدد قليل من الكوم، على الأغلب كومة واحدة فقط.

هذا السلوك هو **صفة متطورة (emergent property)**، التي تستطيع أن تقرأ عنها في http://en.wikipedia.org/wiki/Emergence. تتبع MyTermites قوعد بسيطة معتمدة على معلومات قصيرة المدى، لكن النتيجة هي منظمة كبيرة الحجم.

10.2 نمل لانغتون الأبيض

نملة لانغتون (Langton's Ant) تعكس نمطاً بسيطاً من سلوك النمل الذي يظهر سلوكاً معقداً بشكل مدهش. تعيش النملة في شبكة تشبه شبكة GridWorld حيث تكون الخلية إما بيضاء أو سوداء. تتحرك النملة وفق هذه القواعد:

- إذا كانت النملة على خلية بيضاء، تستدير إلى اليمين، وتجعل الخلية سوداء، ثم تتقدم إلى الأمام.
- إذا كانت النملة على خلية سوداء، تستدير إلى اليسار، وتجعل الخلية بيضاء، ثم تتحرك للأمام.

بما أن هذه القواعد بسيطة، فقد تتوقع أن النملة ستفعل شيئاً بسيطاً مثل رسم مربع أو تكرار نقش معين. لكن إذا كانت على شبكة ذات خلايا بيضاء، ستقوم النملة بأكثر من 10,000 حركة فيما يبدو أنه نقشة عشوائية قبل أن تستقر في حلقة دورانية متكررة تتألف من 104 خطوات.

يمكنك قراءة المزيد عن نملة لانغتون على http://en.wikipedia.org/wiki/Langton_ant.

إن صنع نملة لانغتون في GridWorld ليس سهلاً لأننا لا نستطيع التحكم بلون الخلايا. يمكننا استخدام الأزهار كبديل، لوضع علامة على الخلايا، لكننا لا نستطيع وضع زهرة ونملة على نفس الخلية، لذا لا نقدر على تطبيق قواعد النملة تماماً.

بدلاً من ذلك سننشئ ما سأسميه LangtonTermite، التي تستعمل seeFlower للتحقق من وجود زهرة في الخلية التالية، وpickUpFlower لالتقاطها، وthrowFlower لوضع زهرة في الخلية التالية. قد ترغب بقراءة شفرة هذه العمليات حتى تتأكد أنك تفهم عملها.

تمرين 10.3

a. اصنع نسخة من Termite.java سمها LangtonTermite.java ونسخة من TermiteRunner.java باسم LangtonRunner.java. عدلهم بحيث تكون أسماء الأصناف بنفس أسماء الملفات، وبحيث يقوم LangtonRunner بإنشاء LangtonTermite.
b. إذا أنشأت ملفاً باسم LangtonTermite.gif، سيستخدمه GridWorld لتمثيل نملتك. يمكنك تنزيل صور حشرات ممتازة من http://www.cksinfo.com/animals/insects/realisticdrawings/index.html. لتحويلها إلى صيغة GIF، يمكنك استخدام برنامج مثل ImageMagick.
c. عدل act لإجراء قواعد تشبه قواعد نملة لانغتون. قم بتجارب مستخدماً قواعد مختلفة، وبانعطافات 45 و90 درجة. أوجد القواعد التي تستغرق أكبر عدد من الخطوات قبل أن تبدأ النملة بالدوران في حلقة.
d. لتعطي نملتك مساحة كافية، يمكنك صنع شبكة أكبر أو التحويل إلى UnboundedGrid.
e. أنشئ أكثر من نملة واحدة وانظر كيف ستتفاعل فيما بينها.

تُركت هذه الصفحة بيضاء عن عمد

الفصل 11

اصنع كائناتك الخاصة

11.1 تعاريف الأصناف وأنواع الكائنات

في كل مرة تكتب فيها تعريف صنف جديد، يتم إنشاء نوع كائني جديد له نفس اسم الصنف. بعيداً جداً في القسم 1.5، عندما عرفنا الصنف Hello، تم إنشاء نوع كائني جديد اسمه Hello. لم نقم بإنشاء أية متغيرات من نوع Hello، ولم نستعمل الأمر new لإنشاء أية متغيرات من صنف Hello، لكن ذلك كان ممكناً!

ذلك المثال ليس منطقياً أبداً، وذلك لعدم وجود سبب يدعونا لإنشاء كائن Hello، وحتى لو قمنا بذلك فهو لن ينفعنا كثيراً. في هذا الفصل، سنلقي نظرة على تعاريف الأصناف التي تعطينا أنواعاً كائنية مفيدة.

إليك أهم أفكار هذا الفصل:

- تعريف صنف جديد ينشئ نوعاً كائنياً جديداً له نفس الاسم.
- تعريف الصنف يشبه قالباً للكائنات: يحدد التعريف متغيرات الحالة التي تملكها الكائنات والعمليات التي تشتغل عليها.
- كل كائن ينتمي إلى نوع كائني ما؛ أي أنه حالة لصنف معين.
- عندما تستدعي new لصنع كائن، تستدعي Java عملية خاصة تدعى **الباني (constructor)**، يهيّئ متغيرات الحالة. يمكنك كتابة بانٍ واحد أو أكثر في تعريف الصنف.
- العمليات التي تشتغل على نوع ما تكون معرفة في تعريف صنف ذلك النوع.

إليك بعض مشاكل البنية اللغوية المتعلقة بتعاريف الأصناف:

- يجب أن تبدأ أسماء الأصناف (وبالتالي أسماء الأنواع الكائنية) بحرف كبير، ما يساعد على التمييز بينها وبين الأنواع البسيطة وأسماء المتغيرات.
- عادة ما نضع تعريف صنف واحد في كل ملف، ويجب أن يكون اسم الملف نفس اسم الصنف، وأن ينتهي باللاحقة `.java`. مثلاً، يعرف الصنف Time في ملف باسم Time.java.
- في أي برنامج، يتم التصريح عن صنف واحد على أنه **الصنف البادئ (startup class)**. يجب أن يحتوي صنف البداية على عملية باسم main، وهو المكان الذي يبدأ عنده تنفيذ البرنامج. يمكن للأصناف الأخرى أن تحوي عملية باسم main، لكن تلك العملية لن تنفذ.

وبإزاحة هذه المشاكل عن طريقنا، دعنا ننظر على مثال لصنف معرف بوساطة المستخدم، Time.

11.2 الوقت

من الدوافع الشائعة لإنشاء نوع كائني جديد هو تغليف البيانات ذات الصلة معاً في كائن يمكن التعامل معه كعنصر واحد. لقد رأينا نوعين مشابهين لذلك، Point وRectangle.

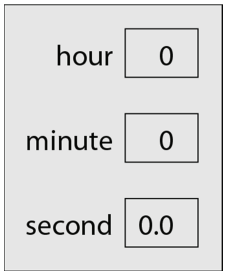
مثال آخر، سننفذه بأنفسنا هو Time، الذي يمثل التوقيت. البيانات المغلفة في كائن الوقت هي ساعة، دقيقة، وعدد من الثواني. بما أن كل كائن Time يملك نفس البيانات هذه، نحتاج إلى متغيرات حالة لتخزينها.

الخطوة الأولى هي تحديد نوع كل متغير. من الواضح أن hour وminute يجب أن يكونا integers. ولإضافة بعض التشويق إلى العمل، دعنا نجعل متغير الثواني second من النوع double.

يتم التصريح عن متغيرات الحالة في بداية تعريف الصنف، خارج أي تعريف لأية عملية، هكذا:

```
class Time {
   int hour, minute;
   double second;
}
```

إن قطعة الشفرة هذه لوحدها، تشكل تعريفاً مشروعاً للصنف. مخطط الحالة (state diagram) لكائن Time يبدو كهذا:



بعد التصريح عن متغيرات الحالة، الخطوة التالية هي تعريف بانِ للصنف الجديد.

11.3 العمليات البانية

تهيئ العمليات البانية متغيراتَ الحالة. بنية الباني مشابهة لبنية أي عملية أخرى، ما عدا ثلاثة اختلافات:

- اسم الباني يكون نفس اسم الصنف.
- لا يملك الباني نوع إرجاع ولا قيمة معادة.
- تُغفَل الكلمة المفتاحية static.

إليك مثالاً عن عملية بانية للصنف Time:

```
public Time() {
   this.hour = 0;
   this.minute = 0;
   this.second = 0.0;
}
```

في الموقع الذي تتوقع فيه رؤية نوع الإرجاع، بين Public وTime، لا يوجد شيء. وهذا هو السبب الذي يمكننا (ويمكن المجمع) من معرفة أن هذه العملية هي عملية بانية.

هذا الباني لا يأخذ أية متحولات. كل سطر من الباني يهيئ متغير حالة مختلف بقيمة افتراضية (في هذه الحالة، منتصف الليل). الاسم this هو كلمة خاصة تشير إلى الكائن الذي سنصنعه. يمكنك استخدام this بنفس طريقة استخدام اسم أي كائن آخر. مثلاً، يمكنك قراءة وكتابة قيم متغيرات الحالة للكائن this، ويمكنك تمرير this كمتحول إلى عمليات أخرى.

لكن لا يمكنك التصريح عن this ولا يمكنك إسناد أي شيء له. يتم إنشاء this بوساطة النظام؛ كل ما عليك فعله هو تهيئة متغيرات الحالة الخاصة به.

من الأخطاء الشائعة عند كتابة العمليات البانية وضع تعليمة return عند النهاية. قاوم العادة.

11.4 المزيد من البناء

يمكن عمل تحميل زائد للعمليات البانية، مثل العمليات الأخرى تماماً، ما يعني أنك تستطيع كتابة عدة عمليات بانية لكل منها معاملات مختلفة. ستعرف Java أي عملية يجب استدعاؤها بمطابقة متحولات new مع معاملات العمليات البانية.

من الشائع وجود بانٍ لا يأخذ أية متحولات (المبين أعلاه)، وآخر يأخذ عدد من معاملات تطابق (بالعدد والنوع) متغيرات الحالة. مثلاً:

```
public Time(int hour, int minute, double second) {
   this.hour = hour;
   this.minute = minute;
   this.second = second;
}
```

تطابق أسماء وأنواع المعاملات أسماء وأنواع متغيرات الحالة. كل ما يفعله الباني هو نسخ المعلومات من المعاملات إلى متغيرات الحالة.

إذا نظرت إلى وثائق Point وRectangle، ستجد أن الاثنين يملكان عمليات بانية تشبه هذه. إن التحميل الزائد للعمليات البانية يوفر المرونة اللازمة لإنشاء كائن أولاً ثم تعبئة الفراغات، أو جمع كافة المعلومات أولاً قبل إنشاء الكائن.

قد يبدو هذا مشوقاً جداً، لكنه ليس كذلك في الحقيقة. كتابة العمليات البانية عملية مملة وميكانيكية. بعد أن تكتب اثنتين منها، ستكتشف أنك تستطيع كتابتها بسرعة بمجرد النظر إلى قائمة متغيرات الحالة.

11.5 صناعة كائن جديد

بالرغم من أن العمليات البانية تبدو مثل العمليات العادية، إلا أنك لا تستدعيها بشكل مباشر أبداً. بدلاً من ذلك، تستدعي new، ويحجز النظام مساحة للكائن الجديد وبعدها يستدعي الباني.

يوضح البرنامج التالي طريقتين لصنع وتهيئة كائن Time:

```
class Time {
   int hour, minute;
   double second;

   public Time() {
      this.hour = 0;
      this.minute = 0;
      this.second = 0.0;
   }

   public Time(int hour, int minute, double second) {
      this.hour = hour;
      this.minute = minute;
      this.second = second;
   }

   public static void main(String[] args) {

      // one way to create and initialize a Time object
      Time t1 = new Time();
      t1.hour = 11;
      t1.minute = 8;
      t1.second = 3.14159;
```

```
        System.out.println(t1);

        // another way to do the same thing
        Time t2 = new Time(11, 8, 3.14159);
        System.out.println(t2);
    }
}
```

في main، أول مرة نستدعي new، لا نعطيه أية متحولات، لذا تستدعي Java الباني الأول. تسند الأسطر القليلة التالية قيماً لمتغيرات الحالة.

في المرة الثانية التي نستدعي new فيها، نعطيه متحولات تطابق معاملات الباني الثاني. هذه الطريقة في تهيئة متغيرات الحالة أكثر اختصاراً وأكثر فاعلية بقليل، لكن يمكن لقراءتها أن تكون أصعب، نظراً لعدم وضوح أية قيمة سيتم إسنادها لأي متغير حالة.

11.6 طباعة الكائنات

إن خرج البرنامج السابق هو:

```
Time@80cc7c0
Time@80cc807
```

عندما تطبع Java قيمة نوع كائني معرف بوساطة المستخدم، تطبع اسم النوع ورمز ست عشري خاص (عدد أساسه 16) فريد ولا يمكن أن يكون مكرراً لكائنين. لا معنى لهذا الرمز في حد ذاته؛ في الحقيقة، يمكن أن يتغير هذا الرمز بين جهاز وآخر بل حتى بين تشغيل وآخر. لكن يمكن أن يكون مفيداً عند تصحيح الأخطاء، في حال كنت ترغب بمتابعة مسار الكائنات المنفردة.

لطباعة الكائنات بطريقة ذات معنى للمستخدمين (ما يقابل المبرمجين)، يمكنك كتابة عملية تدعى شيئاً ما مثل printTime:

```
public static void printTime(Time t) {
    System.out.println(t.hour + ":" + t.minute + ":" + t.second);
}
```

قارن هذه العملية بنسخة printTime الموجودة في القسم 3.10.

إن خرج هذه العملية هو 11:8:3.14159، سواء مررنا t1 أو t2 كمتحول لها. ومع أننا نستطيع معرفة أن هذا الشكل يمثل الوقت، إلا أنه ليس مكتوباً بصيغة صحيحة. مثلاً، إذا كان عدد الدقائق أو الثواني أقل من 10، فيفترض وجود 0 على يسار الرقم لتملأ الفراغ. قد نرغب أيضاً بإهمال الجزء العشري من الثواني. أي أننا نريد شيئاً مثل 11:08:03.

في معظم لغات البرمجة، توجد أساليب بسيطة للتحكم بتنسيق طباعة الأرقام. أما في Java فلا توجد طرق بسيطة.

توفر Java أدوات قوية لطباعة أشياء بتنسيق معين مثل التاريخ والوقت، كما توفر أيضاً أدوات لتفسير المدخلات المنسقة. لسوء الحظ، فإن استعمال هذه الأدوات ليس بسيطاً، لذلك لن أشرحها في هذا الكتاب. يمكنك إلقاء نظرة على وثائق الصنف class الموجود في حزمة java.util، إذا كنت ترغب بذلك.

11.7 العمليات على الكائنات

في الأقسام القليلة المقبلة سأشرح ثلاثة أنواع من العمليات التي تشتغل على الكائنات:

التابع المجرد (pure function): يأخذ كائنات كمتحولات لكنه لا يعدلها. قيمته المعادة إما أن تكون قيمة بسيطة أو كائن جديد تم إنشاؤه داخل العملية.

عملية التعديل (modifier): يأخذ كائنات كمتحولات ويعدل على بعض منها أو كلها. غالباً ما يكون نوع إرجاعه void.

عملية التعبئة (fill-in method): يكون أحد متحولاتها كائن "فارغ" تعبئه العملية. تقنياً، هذه العمليات هي نوع من عمليات التعديل.

من الممكن أغلب الأحيان كتابة نفس العملية بشكل تابع مجرد أو عملية تعديل أو عملية تعبئة. سأناقش مزايا ومساوئ كل منها.

11.8 التوابع المجردة

يتم اعتبار العملية تابعاً مجرداً إذا اعتمدت نتيجتها على المتحولات فقط، ولم تكن للعملية أية تأثيرات جانبية مثل تعديل أحد المتحولات، أو طباعة شيء ما. إن النتيجة الوحيدة المترتبة على استدعاء تابع مجرد هي القيمة المعادة.

من الأمثلة على التوابع المجردة isAfter، التي تقارن بين زمنين (كائنين من الصنف Time) وتعيد قيمة بوليانية تبين فيما إذا كان المعامل الأول يتأخر عن الثاني أو لا:

```
public static boolean isAfter(Time time1, Time time2) {
   if (time1.hour > time2.hour) return true;
   if (time1.hour < time2.hour) return false;

   if (time1.minute > time2.minute) return true;
   if (time1.minute < time2.minute) return false;

   if (time1.second > time2.second) return true;
   return false;
}
```

ما هي نتيجة هذه العملية لو كان الوقتان متساويين؟ هل تبدو تلك النتيجة ملائمة لهذه العملية؟ لو كنت تكتب وثائق هذه العملية، فهل كنت لتذكر تلك الحالة بشكل محدد؟

ومن الأمثلة الأخرى addTime، التي تحسب مجموع زمنين. مثلاً، إذا كانت الساعة 9:14:30، وكانت آلة تحميص الخبز عندك تستغرق 3 ساعات و35 دقيقة، يمكنك استخدام addTime لمعرفة الوقت الذي سيجهز فيه الخبز.

إليك مسودة أولية لهذه العملية وهي ليست مضبوطة تماماً:

```
public static Time addTime(Time t1, Time t2) {
   Time sum = new Time();
   sum.hour = t1.hour + t2.hour;
   sum.minute = t1.minute + t2.minute;
   sum.second = t1.second + t2.second;
   return sum;
}
```

بما أن هذه العملية تعيد كائن Time، فقد تظن أنها عملية بناء، إلا أنها ليست كذلك. عليك العودة والتحقق من بنية العمليات العادية (مثل هذه) والعمليات البانية، لأنك قد تخلط بينهما بسهولة.

إليك مثالاً عن كيفية استعمال هذه العملية. إذا احتوى currentTime على الوقت الحالي وbreadTime على الزمن اللازم لتحميص الخبز، فعندئذ يمكنك استعمال addTime لمعرفة الوقت الذي سيكون الخبز فيه جاهزاً.

```
Time currentTime = new Time(9, 14, 30.0);
```

```
Time breadTime = new Time(3, 35, 0.0);
Time doneTime = addTime(currentTime, breadTime);
printTime(doneTime);
```

إن خرج هذا البرنامج هو 12:49:30ز0، وهو صحيح. من ناحية أخرى، توجد حالات تكون فيها الإجابة خاطئة. هل يمكنك التفكير في واحدة منها؟

المشكلة هي أن هذه العملية لا تعالج الحالات التي يصبح فيها عدد الثواني أو الدقائق أكبر من 60 بعد عملية الجمع. في تلك الحالة، علينا "حمل" الثواني الزائدة إلى عمود الدقائق، أو الدقائق الزائدة إلى عمود الساعات.

ها هي النسخة الصحيحة من العملية.

```
public static Time addTime(Time t1, Time t2) {
  Time sum = new Time();
  sum.hour = t1.hour + t2.hour;
  sum.minute = t1.minute + t2.minute;
  sum.second = t1.second + t2.second;

  if (sum.second >= 60.0) {
    sum.second -= 60.0;
    sum.minute += 1;
  }
  if (sum.minute >= 60) {
    sum.minute -= 60;
    sum.hour += 1;
  }
  return sum;
}
```

ومع أنها صحيحة، إلا أنها بدأت تكبر. سأقترح بدائل أقصر لها لاحقاً.

تشرح هذه الشفرة عمليتين لم نرهما من قبل، وهما =+ و=-. هذان العاملان يوفران طريقة مختصرة لزيادة أو إنقاص المتغيرات. وهما يشبهان ++ و-- في عملهما، عدا (1) يمكن أن يعملا على الأعداد العشرية بالإضافة إلى الأعداد الصحيحة، و(2) ليس بالضرورة أن تكون الزيادة أو النقصان بمقدار 1. التعليمة sum.second -= 60.0; مكافئة للتعليمة sum.second = sum.second – 60.0;.

11.9 عمليات التعديل

لنأخذ increment، كمثال عن عملية تعديل، التي تضيف عدد معطى من الثواني إلى كائن Time. المسودة الأولية لهذه العملية ستبدو كهذه:

```
public static void increment(Time time, double secs) {
  time.second += secs;

  if (time.second >= 60.0) {
    time.second -= 60.0;
    time.minute += 1;
  }
  if (time.minute >= 60) {
    time.minute -= 60;
    time.hour += 1;
  }
}
```

يجري السطر الأول العملية الأساسية؛ وتتم معالجة الباقي كما فعلنا في الحالات التي مرت معنا سابقاً.

هل هذه العملية صحيحة؟ ماذا يحدث لو أن المتحول كان أكبر بكثير من 60؟ في تلك الحالة لن يكون طرح 60 مرة واحدة كافياً؛ علينا الاستمرار في الطرح حتى تصبح قيمة second أقل من 60. يمكننا عمل ذلك باستبدال تعليمة if بتعليمة while:

```
public static void increment(Time time, double secs) {
   time.second += secs;

   while (time.second >= 60.0) {
     time.second -= 60.0;
     time.minute += 1;
   }
   while (time.minute >= 60) {
     time.minute -= 60;
     time.hour += 1;
   }
}
```

هذا الحل صحيح، لكنه غير فعال جداً. هل يمكنك التفكير بحل لا يتطلب التكرار؟

11.10 عمليات التعبئة

بدلاً من إنشاء كائن جديد في كل مرة يستدعى فيها addTime، يمكننا أن نطلب من المستدعي أن يوفر كائن تخزن addTime النتيجة فيه. قارن هذه النسخة مع النسخة السابقة:

```
public static void addTimeFill(Time t1, Time t2, Time sum) {
   sum.hour = t1.hour + t2.hour;
   sum.minute = t1.minute + t2.minute;
   sum.second = t1.second + t2.second;

   if (sum.second >= 60.0) {
     sum.second -= 60.0;
     sum.minute += 1;
   }
   if (sum.minute >= 60) {
     sum.minute -= 60;
     sum.hour += 1;
   }
}
```

يتم تخزين النتيجة في sum، لذلك يكون نوع الإرجاع void.

عمليات التعديل وعمليات التعبئة فعالة بسبب عدم الحاجة لصنع كائنات جديدة. لكنها تصعب فصل أجزاء البرنامج؛ في المشاريع الكبيرة يمكن أن تتسبب هذه العمليات بأخطاء يصعب العثور عليها.

التوابع المجردة تساعد في إدارة المشاريع الكبيرة، وذلك يعود جزئياً لأنها تجعل أنواعاً معينة من الأخطاء مستحيلة الوقوع. كما أنها مناسبة أيضاً لأنواع معينة من التركيب والتداخل. وبما أن نتيجة التابع المجرد تعتمد على المعاملات فقط، فيمكن تسريعها بتخزين النتائج المحسوبة مسبقاً.

أنا أنصحك بأن تكتب توابع مجردة طالما أن ذلك ممكن، وأن تلجأ إلى عمليات التعديل في حال وجود مزايا ضرورية فقط.

11.11 التطوير والتخطيط التصاعدي

لقد اعتمدت في هذا الفصل على أسلوب في تطوير البرامج يدعى **rapid prototyping**[1]. في كل مرة، كتبت مسودة أولية للعملية تجري الحسابات الأساسية، بعد ذلك اختبرتها على عدة حالات، وصححت الأخطاء التي وجدتها.

يمكن لهذا الأسلوب أن يكون فعالاً، لكنه قد يقودنا إلى شفرة معقدة أكثر من اللازم – نظراً لأنه يعالج العديد من الحالات الخاصة – وغير موثوقة – وذلك لأنك لأن إقناع نفسك بأنك قد وجدت جميع الأخطاء صعب.

من الأساليب البديلة هو البحث عن فكرة في تلك المشكلة تجعل البرمجة أسهل. في هذه الحالة الفكرة هي أن الوقت في الحقيقة هو عدد مؤلف من ثلاث خانات في الأساس الستيني! الثواني هي خانة "الآحاد"، والدقائق هي "الستينات"، والساعات هي خانة "3600".

عندما كتبنا addTime وincrement، كنا نجري عملية الجمع في النظام الستيني في الحقيقة، ولذلك اضطررنا إلى "الحمل" من عمود إلى تاليه.

من الأساليب الأخرى للتعامل مع المشكلة ككل هو تحويل الأوقات إلى أعداد عشرية والاستفادة من قدرة الحواسيب على تنفيذ العمليات الرياضية على الأعداد العشرية. ها هي العملية التي تحول كائن Time إلى عدد من نوع double:

```
public static double convertToSeconds(Time t) {
   int minutes = t.hour * 60 + t.minute;
   double seconds = minutes * 60 + t.second;
   return seconds;
}
```

كل ما نحتاجه الآن هو التحويل من double إلى كائن Time. يمكننا كتابة عملية تجري بذلك، لكن كتابتها كعملية بانية ثالثة تبدو منطقية أكثر:

```
public Time(double secs) {
   this.hour =(int)(secs / 3600.0);
   secs -= this.hour * 3600.0;
   this.minute =(int)(secs / 60.0);
   secs -= this.minute * 60;
   this.second = secs;
}
```

هذا الباني يختلف قليلاً عن سابقيه؛ فهو يشتمل على حسابات بالإضافة إلى تعليمات الإسناد إلى متغيرات الحالة.

قد تحتاج للتفكير قبل أن تقنع نفسك بأن التقنية التي أستخدمها للتحويل من أساس إلى آخر صحيحة. وبمجرد أن تقتنع، سنكون جاهزين لاستعمال هذه العمليات لكتابة addTime من جديد:

```
public static Time addTime(Time t1, Time t2) {
   double seconds = convertToSeconds(t1) + convertToSeconds(t2);
   return new Time(seconds);
}
```

هذه النسخة أقصر من الأصلية، ومن الأسهل بكثير معرفة أنها ستعمل بشكل صحيح (بفرض – كالعادة – أن العمليات التي تستدعيها صحيحة).

كتمرين، أعد كتابة increment بنفس الطريقة.

---

[1] ما أدعوه بالنمذجة الأولية السريعة rapid prototyping مشابه للتطوير الموجه بالاختبار test-driven development (TDD)؛ الفرق بينهما هو أن TDD يقوم عادة على أساس الاختبار المؤتمت. انظر http://en.wikipedia.org/wiki/Test-driven_development.

11.12 التعميم

في بعض الأحيان يكون التحويل من الأساس 60 إلى الأساس 10 وبالعكس أصعب من التعامل مع الوقت وحسب. التحويل بين الأسس أكثر تجريداً؛ عند التعامل مع الوقت فإن إدراكنا البديهي يكون أفضل.

لكن لو عرفنا إمكانية التعامل مع الأوقات على أنها أعداد في النظام الستيني، وقمنا باختراع عمليات التحويل (convertToSeconds والباني الثالث)، سنحصل على برنامج أقصر، أسهل عند القراءة وتنقيح الأخطاء، وأكثر موثوقية.

ستصبح إضافة مزايا أخرى فيما بعد أسهل أيضاً. تخيل طرح زمنين ومعرفة المدة بينهما. سيكون الحل الغر هو إجراء عملية الطرح كاملة مع "الاستعارة". أما استعمال عمليات التحويل فسيكون أسهل بكثير.

ومن السخرية أن جعل المشكلة أصعب (جعلها مشكلة عامة) يجعلها أسهل للحل أحياناً (عدد أقل من الحالات الخاصة، احتمالات أقل للخطأ).

11.13 الخوارزميات

عندما تكتب حلاً عاماً لفئة من المشاكل، بدلاً من حل خاص لمشكلة واحدة، فأنت تكتب **خوارزمية (algorithm)**. إن تعريف هذه الكلمة ليس سهلاً، لذا سأجرب أسلوبين للتعريف.

أولاً، خذ بعين الاعتبار بعض الأشياء غير الخوارزميات. عندما تعلمت ضرب الأعداد ذات الخانة الواحدة، قمت بحفظ جدول الضرب على الأغلب. أي أنك حفظت 100 حل خاص، لذا فإن المعرفة ليست خوارزمية فعلاً.

لكن لو كنت "كسولاً"، فعلى الأغلب أنك تعلمت بعض الحيل. مثلاً، لإيجاد ناتج ضرب n ب9، يمكنك كتابة n-1 في الخانة الأولى و10-n في الخانة الثانية. هذه الحيلة هي حل عام لضرب أي عدد مؤلف من خانة واحدة بالعدد 9. هذه الحيلة هي خوارزمية!

برأيي، من المخجل أن البشر يقضون الكثير من الوقت في المدرسة يتعلمون إجراء خوارزميات لا تتطلب أي نوع من الذكاء.

من جهة أخرى، فإن تصميم الخوارزميات عملية ممتعة، فيها تحد للعقل، وهي جزء مركزي لما ندعوه بالبرمجة.

بعض الأشياء التي يعملها الناس بصورة طبيعية، بدون صعوبة أو تفكير واعي، تكون أصعب الأشياء عند التعبير عنها بشكل خوارزمية. فهم اللغات الطبيعية هو مثال جيد. كلنا نفهم اللغة، لكن حتى الآن لم يتمكن أحد من شرح كيفية قيامنا بذلك، على الأقل ليس بشكل خوارزمية.

قريباً ستملك القدرة على تصميم خوارزميات بسيطة لمجموعة متنوعة من المشاكل.

11.14 المصطلحات

صنف: سابقاً، عرّفتُ الصنف على أنه مجموعة من العمليات المترابطة. في هذا الفصل تعلمنا أن تعريف الصنف هو قالب لنوع كائنات جديد أيضاً.

حالة: عضو ينتمي لصنف ما. كل كائن هو حالة من صنف ما.

الباني: عملية خاصة تهيئ متغيرات الحالة للكائنات المنشأة حديثاً.

الصنف البادئ: الصنف الذي يحتوي على عملية main التي يبدأ تنفيذ البرنامج عندها.

تابع مجرد: عملية تعتمد نتيجتها على معاملاتها فقط، وليس لها أي تأثيرات سوى إرجاع قيمة.

عملية تعديل: عملية تغير كائناً واحداً أو أكثر من الكائنات التي تأخذها كمعاملات، وعادة تعيد void.

عمليات التعبئة: نوع من العمليات التي تأخذ كائناً "فارغاً" كمعامل وتملأ متغيرات الحالة الخاصة به بدلاً من توليد قيمة معادة.

خوارزمية: مجموعة من التعليمات المستخدمة لحل فئة من المشاكل بعملية ميكانيكية.

class: Previously, I defined a class as a collection of related methods. In this chapter we learned that a class definition is also a template for a new type of object.

instance: A member of a class. Every object is an instance of some class. constructor: A special method that initializes the instance variables of a newly-constructed object.

startup class: The class that contains the main method where execution of the program begins.

pure function: A method whose result depends only on its parameters, and that has no side-effects other than returning a value.

modifier: A method that changes one or more of the objects it receives as parameters, and usually returns void.

fill-in method: A type of method that takes an “empty” object as a parameter and fills in its instance variables instead of generating a return value.

algorithm: A set of instructions for solving a class of problems by a mechanical process.

11.15 تمرينات

تمرين 11.1 في لعبة الكلمات سكرابل (Scrabble)[2]، كل قطعة تحتوي على حرف، يستعمل لتركيب الكلمات، ونقاط، تستخدم لتحديد قيمة الكلمة.

a. اكتب تعريفاً لصنف اسمه Tile يعبر عن قطع سكرابل. متغيرات الحالة يجب أن يكونا محرفاً يدعى letter وعدداً صحيحاً يدعى value.
b. اكتب عملية بناء تأخذ معاملات اسمها letter وvalue وتهيئ متغيرات الحالة.
c. اكتب عملية اسمها printTile تأخذ كائن Tile كمعامل وتطبع متغيرات الحالة بصيغة سهلة القراءة.
d. اكتب عملية اسمها testTile تنشئ كائن Tile له الحرف Z والقيمة 10، ثم تستخدم printTile لطباعة حالة الكائن.

إن الغرض من هذا التمرين هو التدرب على الجزء الميكانيكي من إنشاء تعريف لصنف جديد وشفرة تختبره.

[2] Scrabble هي علامة تجارية مسجلة تملكها Hasbro, Inc. في الولايات المتحدة الأمريكية وكندا، وفي بقية العالم J.W. Spear & Sons Limited of Maidenhead, Berkshire, England، فرع من شركة Mattel, Inc.

تمرين 11.2 اكتب تعريفاً للصنف Date، وهو نوع كائني يحوي ثلاثة أعداد صحيحة، year، month، وday. يجب أن يحوي هذا الصنف عمليتي بناء. الأولى لا تأخذ أية معاملات. أما الثانية فيجب أن تأخذ ثلاثة معاملات year، month، وday وتستخدمها لتهيئة متغيرات الحالة.

اكتب عملية main تنشئ كائن Date جديد باسم birthday. يجب أن يحتوي الكائن الجديد على تاريخ ميلادك. يمكنك استخدام أي من عمليتي البناء.

تمرين 11.3

العدد الكسري هو عدد يمكن تمثيله بكسر يكون حديه عددين صحيحين. مثلاً، 2/3 عدد كسري، ويمكنك اعتبار 7 عدد كسري بما أن مقامه 1. في هذه الوظيفة، سوف تكتب تعريف صنف للأعداد الكسرية.

a. أنشئ برنامجاً جديداً يدعى Rational.java يعرف صنفاً باسم Rational. يجب أن يحتوي كائن Rational على متغيري حالة من النوع int لتخزين البسط والمقام.

b. اكتب عملية بانية لا تأخذ أية متحولات وتعطي المتغيرين قيمة الصفر.

c. اكتب عملية تدعى printRational تأخذ كائن Rational كمتحول وتطبعه في صيغة مناسبة.

d. اكتب عملية main تنشئ كائناً جديداً من النوع Rational، وتعطي قيماً ما لمتغيري الحالة الخاصين به، وتطبع ذلك الكائن.

e. في هذه المرحلة، ستملك برنامجاً مصغراً قابلاً للاختبار. اختبره و –في حال دعت الحاجة- صحح الأخطاء.

f. اكتب عملية بانية أخرى للصنف تأخذ متحولين وتستخدمهم لتهيئة متغيرات الحالة.

g. اكتب عملية باسم negate تعكس إشارة العدد الكسري. يجب أن تكون هذه العملية معدّلة، لذلك يجب أن يكون نوع إرجاعها void. أضف بعض السطور إلى main لاختبار العملية الجديدة.

h. اكتب عملية باسم invert تقلب العدد بالتبديل بين بسطه ومقامه. أضف سطوراً إلى main لاختبار العملية الجديدة.

i. اكتب عملية تدعى toDouble تحول العدد الكسري إلى عدد عشري (عدد ذو فاصلة) وتعيد النتيجة. هذه العملية هي تابع مجرد؛ فهي لا تعدل على الكائن. كما هو الحال دائماً، اختبر العملية الجديدة.

j. اكتب عملية معدّلة باسم reduce تختزل العدد الكسري إلى أبسط شكل له وذلك بحساب القاسم المشترك الأكبر (GCD) للبسط والمقام وتقسيمهما عليه. يجب أن تكون هذه العملية تابعاً مجرداً؛ يجب ألا تعدل متغيرات الحالة للكائن الذي استدعيت العملية عليه. لحساب GCD، انظر التمرين 6.10.

k. اكتب عملية باسم add تأخذ عدد كسريين كمتحولات وتعيد كائناً جديداً من نوع Relational. يجب أن يحتوي الكائن المعاد على مجموع المتحولين.
توجد عدة طرق لجمع الكسور. يمكنك استخدام أي منها، لكن عليك التأكد من اختزال نتيجة العملية بحيث لا يوجد للبسط والمقام قواسم مشتركة (ما عدا 1).

إن الغرض من هذا التمرين هو كتابة تعريف صنف يتضمن مجموعة من العمليات المتنوعة. بما في ذلك عمليات بناء ومعدلات وتوابع مجردة.

الفصل 12

المصفوفات

المصفوفة (array) هي مجموعة من القيم حيث تكون كل قيمة معرفة بوساطة دليل. يمكنك عمل مصفوفات من الأعداد الصحيحة أو العشرية أو أي نوع آخر، لكن يجب أن تكون جميع العناصر من نوع واحد في المصفوفة الواحدة.

نحوياً، تكتب الأنواع المصفوفية في Java كما تكتب الأنواع الأخرى عدا أنها تتبع بالقوسين المربعين []. مثلاً، []int هو نوع "مصفوفة أعداد صحيحة" و[]double هو النوع "مصفوفة أعداد عشرية".

يمكنك التصريح عن متغيرات من هذين النوعين بالطرق المعتادة:

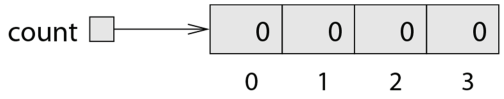
```
int[] count;
double[] values;
```

يتم إعطاء القيمة null لهذين المتغيرين، حتى تهيئهما. لإنشاء المصفوفة الفعلية، استعمل new.

```
count = new int[4];
values = new double[size];
```

تجعل تعليمة الإسناد الأولى count يشير إلى مصفوفة من أربعة أعداد صحيحة؛ تعليمة الإسناد الثنية تجعل values يشير إلى مصفوفة أعداد عشرية. إن عدد العناصر في values يعتمد على size. يمكنك استعمال أي تعبير حسابي صحيح (يكون ناتجه عدداً صحيحاً) كحجم لمصفوفة.

يبين الشكل التالي كيف تمثل المصفوفات في مخططات الحالة:



الأرقام الكبيرة داخل الصناديق هي **عناصر (elements)** المصفوفة. تستعمل الأرقام الصغيرة خارج الصناديق لتعريف كل صندوق. عندما تحجز مصفوفة أعداد صحيحة، تعطى عناصرها القيمة الافتراضية 0.

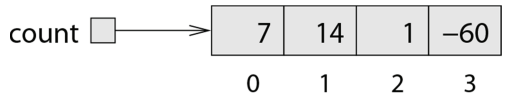
12.1 الوصول إلى العناصر

لتخزين القيم في المصفوفة، استخدم عامل []. مثلاً، [0]count تشير إلى العنصر الصفري (رقم صفر – zeroeth) في المصفوفة، و[1]count تشير إلى العنصر الأول.

يمكنك استخدام عامل [] في أي مكان من عبارة ما:

```
count[0] = 7;
count[1] = count[0] * 2;
count[2]++;
count[3] -= 60;
```

هذه التعليمات هي تعليمات إسناد مشروعة. ها هي نتيجة قطعة الشفرة هذه:



أن عناصر المصفوفة مرقمة من 0 إلى 3، ما يعني عدم وجود عنصر دليله 4. يجب أن يكون هذا مألوفاً، نظراً لأننا رأينا الشيء نفسه عندما تعاملنا مع أدلة السلاسل المحرفية. ومع ذلك، فمن الأخطاء الشائعة تجاوز حدود المصفوفة، ما سيسبب الاستثناء ArrayOutOfBoundsException. كما هو الحال مع باقي الاستثناءات، سيعطيك البرنامج رسالة خطأ ثم يخرج.

يمكنك استعمال أي تعبير كدليل، طالما أنه من النوع int. من أكثر الطرق شيوعاً في تحديد أدلة المصفوفات هو استخدام متغير حلقة. مثلاً:

```
int i = 0;
while (i < 4) {
    System.out.println(count[i]);
    i++;
}
```

هذه حلقة while قياسية تعد من 0 إلى 4، وعندما يصبح متغير الحلقة i يساوي 4، لا يتحقق الشرط وتنتهي الحلقة. بالتالي، فإن جسم الحلقة ينفذ فقط عندما يكون i يساوي 0، 1، 2 و3.

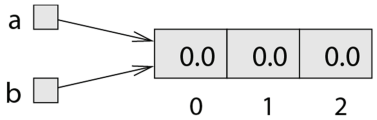
في كل مرة تدور فيها الحلقة نستعمل i كدليل للمصفوفة، لطباعة العنصر رقم i. هذا النوع من عبور المصفوفات شائع جداً.

12.2 نسخ المصفوفات

عندما تنسخ متغير مصفوفي، تذكر أنك تنسخ مرجعاً لمصفوفة. مثلاً:

```
double[] a = new double [3];
double[] b = a;
```

هذه الشفرة تنشئ مصفوفة مؤلفة من ثلاثة أعداد عشرية، وتجعل متغيرين مختلفين يشيران إليها. هذه الحالة هي شكل من الأسماء المستعارة.



أي تغيير في أي من المصفوفتين سيؤثر على الأخرى. ليس هذا ما تريده على الأغلب؛ بل إنك ستحتاج غالباً إلى حجز مصفوفة جديدة ونسخ العناصر من واحدة إلى أخرى.

```
double[] b = new double [3];
int i = 0;
while (i < 4) {
b[i] = a[i];
i++;
}
```

12.3 حلقات for

إن جميع الحلقات التي كتبناها حتى الآن لها عدد من العناصر المشتركة. كلها تبدأ بتهيئة متغير؛ وتختبر شرط يعتمد على ذلك المتغير؛ وفي كل حلقة يوجد شيء يؤثر على ذلك المتغير، مثل زيادة قيمته.

هذا النمط من الحلقات شائع جداً لدرجة أنهم اخترعوا تعليمة شرطية أخرى لأجله، تدعى for، والتي تعبر عن الخطوات السابقة بشكل أكثر اختصاراً. البنية العامة تبدو كهذه:

```
for (INITIALIZER; CONDITION; INCREMENTOR) {
   BODY
}
```

هذه التعليمة مكافئة لما يلي:

```
INITIALIZER;
while (CONDITION) {
   BODY
   INCREMENTOR
}
```

ما عدا أنها أكثر اختصاراً و، نظراً لأنها تضع جميع التعليمات المتعلقة بالحلقة في مكان واحد، أسهل للقراءة. مثلاً:

```
for (int i = 0; i < 4; i++) {
   System.out.println(count[i]);
}
```

تكافئ

```
int i = 0;
while (i < 4) {
   System.out.println(count[i]);
   i++;
}
```

تمرين 12.1 اكتب حلقة for لنسخ عناصر مصفوفة.

12.4 المصفوفات والكائنات

تتصرف المصفوفات بشكل مشابه للكائنات في العديد من النواحي:

- عندما تصرح عن متغير مصفوفي، تحصل على مرجع لمصفوفة.
- عليك استعمال new لإنشاء المصفوفة الفعلية.
- عندما تمرر مصفوفة كمتحول، فأنت تمرر مرجعاً، ما يعني أن العملية المستدعاة تقدر على تغيير محتويات المصفوفة.

بعض الكائنات التي تعاملنا معها، مثل المستطيلات (Rectangle)، تشبه المصفوفات، بمعنى أنها عبارة عن مجموعة من القيم. وهنا يطرح السؤال نفسه، "بم تختلف المصفوفة المؤلفة من 4 أعداد صحيحة عن كائن Rectangle؟"

إذا عدت إلى تعريف "المصفوفة" في بداية هذا الفصل، فستجد اختلافاً واحداً، وهو أن عناصر المصفوفة تعرف باستخدام أدلة، في حين تملك عناصر الكائن (متغيرات الحالة) أسماء.

يوجد اختلاف آخر وهو أن عناصر المصفوفة يجب أن تكون من نفس النوع. أما الكائنات فيمكن أن تملك متغيرات حالة من أنواع مختلفة.

12.5 طول المصفوفة

في الواقع، تملك المصفوفات متغير حالة وحيد له اسم: length. ومن غير المفاجئ، أنه يحتوي على طول المصفوفة (عدد عناصرها). من الجيد استخدام هذه القيمة كحد أعظمي لحلقة، بدلاً من استخدام قيمة ثابتة. وبذلك، لن تضطر إلى المرور على جميع الحلقات في البرنامج لتعديل الشرط، عندما يتغير حجم المصفوفة؛ لأنها ستعمل بشكل صحيح من أجل أي حجم للمصفوفة.

```
for (int i = 0; i < a.length; i++) {
   b[i] = a[i];
}
```

في آخر مرة يتم تنفيذ جسم الحلقة فيها، i يساوي a.length – 1، وهو دليل العنصر الأخير. عندما يكون i مساوياً لـa.length، لا يتحقق الشرط ولا يتم تنفيذ جسم الحلقة، وهو شيء جيد، لأن تنفيذه كان سيتسبب في استثناء. هذه الشفرة تفترض أن المصفوفة b تحتوي عدداً من العناصر يساوي عدد عناصر a على الأقل.

تمرين 12.2 اكتب عملية تدعى cloneArray تأخذ مصفوفة أعداد صحيحة كمعامل، وتنشئ مصفوفة جديدة بنفس الحجم، وتنسخ العناصر من المصفوفة الأولى إلى المصفوفة الجديدة، ثم تعيد مرجعاً للمصفوفة الجديدة.

12.6 الأرقام العشوائية

معظم برامج الحاسوب تجري نفس الأفعال في كل مرة يتم تشغيلها، ولذلك يقال أنها **حتمية (deterministic)**. في العادة، تكون الحتمية شيئاً جيداً، لأننا نتوقع أن تعطي العملية الحسابية نفس النتيجة كل مرة. لكن في بعض التطبيقات نحتاج من الحاسب أن يتصرف بشكل غير محسوب. الألعاب هي مثال واضح على هكذا تطبيقات، لكن يوجد غيرها أيضاً.

إن جعل البرنامج غير متوقع بالمرة (truly **nondeterministic**) ليس سهلاً، لكن توجد طرق لجعله يبدو غير محتم على الأقل. إحدى تلك الطرق توليد الأرقام العشوائية واستعمالها لتحديد خرج البرنامج. توفر Java، عملية تولد أرقاماً **شبه عشوائية (pseudorandom)**، التي قد لا تكون عشوائية تماماً، لكن مقارنة مع احتياجاتنا، فهي عشوائية بما يكفي.

انظر إلى وثائق العملية random في صنف Math. القيمة المعادة هي عدد عشري محصور بين 0.0 و1.0. لنتوخى الدقة، العدد أكبر من أو يساوي 0.0 وأصغر تماماً من 1.0. في كل مرة تستدعي random تحصل على الرقم التالي في سلسلة شبه عشوائية. لتشاهد مثالاً على ذلك، شغل هذه الحلقة:

```
for (int i = 0; i < 10; i++) {
   double x = Math.random();
   System.out.println(x);
}
```

لتوليد عدد عشري عشوائي محصور بين 0.0 وحد أعظمي متغير مثل high، يمكنك ضرب x بـhigh. كيف يمكنك توليد عدد عشوائي بين حدين low وhigh؟ كيف تستطيع توليد عدد صحيح عشوائي؟

تمرين 12.3 اكتب عملية باسم randomDouble تأخذ عددين عشريين، low وhigh، وتعيد قيمة عشوائية عشرية x بحيث يكون $low \leq x < high$.

تمرين 12.4 اكتب عملية باسم randomInt تأخذ متحولين، low وhigh، وتعيد قيمة عشوائية صحيحة أكبر أو تساوي low وأصغر تماماً من high.

12.7 مصفوفة الأعداد العشوائية

إذا كانت عملية randomInt التي كتبتها صحيحة، عندئذ فإن كل قيمة تنتمي إلى المجال من low إلى high-1 يجب أن تملك نفس الاحتمال. إذا ولدّت سلسلة طويلة من الأرقام، فيجب أن تظهر كل قيمة، تقريباً على الأقل، بنفس العدد من المرات.

إحدى الطرق لاختبار عمليتك هي توليد عدد كبير من القيم العشوائية، وتخزينهم في مصفوفة، وحساب عدد مرات تكرار كل قيمة.

تأخذ العملية التالية متحولاً وحيداً، حجم المصفوفة. وتحجز مصفوفة أعداد صحيحة جديدة، وتملأها بقيم صحيحة عشوائية، وتعيد مرجعاً للمصفوفة الجديدة.

```
public static int[] randomArray(int n) {
   int[] a = new int[n];
   for (int i = 0; i<a.length; i++) {
      a[i] = randomInt(0, 100);
   }
   return a;
}
```

نوع الإرجاع هو []int، ما يعني أن هذه العملية تعيد مصفوفة أعداد صحيحة. لاختبار هذه العملية، سنحتاج إلى عملية تطبع محتويات مصفوفة.

```
public static void printArray(int[] a) {
   for (int i = 0; i<a.length; i++) {
      System.out.println(a[i]);
   }
}
```

الشفرة التالية تولد مصفوفة ما وتطبعها:

```
int numValues = 8;
int[] array = randomArray(numValues);
printArray(array);
```

كان الخرج على جهازي كالتالي

```
27
6
54
62
54
2
44
81
```

وهو يبدو عشوائي قليلاً. قد تختلف النتائج على جهازك.

لو كانت هذه علامات امتحان (لكانت علامات امتحان سيئة فعلاً) لربما قام المدرس بتمثيل هذه النتائج وتقديمها إلى الفصل بشكل **مخطط أعمدة (histogram)**، وهو مجموعة من العدادات التي تتابع عدد تكرار كل قيمة من القيم الموجودة.

بالنسبة لعلامات الامتحان، لربما جعلنا العدادات تحسب عدد الطلاب الذين كانت علاماتهم في التسعينات، الثمانينات، الخ. سنطوّر في الأقسام القليلة القادمة شفرة لتوليد مخطط أعمدة.

12.8 العد

من الأساليب الجيدة لحل مشاكل مثل هذه، التفكير بعمليات بسيطة سهلة الكتابة، ثم جمعهم معاً لتشكيل الحل. هذه العملية تدعى **التطوير من الأسفل إلى الأعلى (bottom-up development)**. انظر
.http://en.wikipedia.org/wiki/Top-down_and_bottom-up_design

ليس واضحاً دائماً أين يجب أن تبدأ، لكن من الجيد أن تبحث عن مشاكل متفرعة تطابق نمطاً قد شاهدته من قبل.

في القسم 8.7 رأينا حلقة اجتازت سلسلة محرفية وعدّت مرات تكرار حرف معطى في السلسلة. يمكنك اعتبار هذا البرنامج مثالاً عن نمط يدعى "اعبر وعد – traverse and count". عناصر هذا النمط هي:

- مجموعة أو حاوية يمكن عبورها، مثل مصفوفة أو سلسلة محرفية.
- اختبار يمكنك إجراؤه على كل عنصر في الحاوية.
- عداد يتابع عدد تكرار العناصر التي نجحت في الاختبار.

في هذه الحالة، يكون الوعاء هو مصفوفة أعداد صحيحة. الاختبار هو هل تنتمي درجة معطاة إلى مجال معين من القيم أم لا.

هنا عملية باسم inRange تحسب عدد العناصر في مصفوفة، التي تنتمي لمجال قيم معطى. المعاملات هي المصفوفة وعددين صحيحين يبينان حدي المجال الأدنى والأعلى.

```
public static int inRange(int[] a, int low, int high) {
  int count = 0;
  for (int i=0; i<a.length; i++) {
    if (a[i] >= low && a[i] < high) count++;
  }
  return count;
}
```

لم أحدد فيما إذا كان شيء ما مساوياً للحد الأدنى أو الأعلى سينتمي إلى المجال أم لا، لكن يمكنك أن ترى من الشفرة أن low ينتم إلى المجال في حين high لا ينتمي إليه. هذا سيمنعنا من عد أية عنصر مرتين.

الآن يمكننا حساب عدد العلامات الموجودة ضمن المجالات التي تهمنا:

```
int[] scores = randomArray(30);
int a = inRange(scores, 90, 100);
int b = inRange(scores, 80, 90);
int c = inRange(scores, 70, 80);
int d = inRange(scores, 60, 70);
int f = inRange(scores, 0, 60);
```

12.9 مخطط الأعمدة

هذه الشفرة تكرارية، لكنها مقبولة فقط طالما أن عدد المجالات صغير. لكن تخيل أننا نريد متابعة عدد تكرار كل علامة تظهر، كل القيم الممكنة من 0 إلى 100. أتريد كتابة هذا؟

```
int count0 = inRange(scores, 0, 1);
int count1 = inRange(scores, 1, 2);
int count2 = inRange(scores, 2, 3);
...
int count3 = inRange(scores, 99, 100);
```

لا أعتقد ذلك. ما تريده حقاً هو طريقة لتخزين 100 عدد صحيح، يفضل أن نستطيع استخدام دليل للوصول إلى كل واحد منها. مساعدة: مصفوفة.

أن مضمون عملية العد هو نفسه سواء استخدمنا عداداً مفرداً أو مصفوفة من العدادات. في هذه الحالة، سنهيئ المصفوفة خارج الحلقة؛ ثم، داخل الحلقة، نستدعي inRange ونخزن النتيجة:

```
int[] counts = new int[100];

for (int i = 0; i < counts.length; i++) {
   counts[i] = inRange(scores, i, i+1);
}
```

الشيء الجديد الوحيد هنا هو أننا أعطينا متغير الحلقة مهمتين: دليل للمصفوفة، ومعامل للعملية inRange.

12.10 حل بدورة واحدة

هذه الشفرة تعمل، لكنها ليست فعالة كما ينبغي. في كل مرة تستدعى فيها inRange، يتم المرور على كامل المصفوفة. ومع ازدياد عدد المجالات، سيكون لدينا الكثير من عمليات الاجتياز.

سيكون من الأفضل المرور على المصفوفة مرة واحدة، ومن أجل كل قيمة، نحسب أي مجال تنتمي إليه. بعدها يمكننا زيادة العداد المناسب. في هذا المثال، ستكون تلك الحسبة تافهة، لأننا نستطيع استعمال القيمة نفسها كدليل لمصفوفة العدادات.

ها هي الشفرة التي تجتاز مصفوفة العلامات، مرة واحدة، وتولد مخطط الأعمدة الخاص بها.

```
  int[] counts = new int[100];

  for (int i = 0; i < scores.length; i++) {
    int index = scores[i];
    counts[index]++;
}
```

تمرين 12.5. غلف هذه الشفرة في عملية باسم makeHist تأخذ مصفوفة علامات وتعيد مخطط الأعمدة (histogram) قيم المصفوفة.

12.11 المصطلحات

مصفوفة: مجموعة من القيم، حيث تكون جميع القيم من نفس النوع، وكل قيمة تعرف بدليل.

عنصر: واحد القيم في مصفوفة. يختار العامل [] العناصر.

دليل: متغير من النوع الصحيح أو قيمة تستعمل للإشارة إلى عنصر من مصفوفة.

حتمي: برنامج ينفذ نفس الشيء في كل مرة يستدعى فيها.

شبه عشوائي: سلسلة من الأرقام تبدو عشوائية، لكنها أساساً ناتجة عن عملية حسابية محتمة.

مخطط الأعمدة: مصفوفة من الأعداد الصحيحة حيث يمثل كل عدد صحيح عدد القيم التي تنتمي إلى نطاق معين.

array: A collection of values, where all the values have the same type, and each value is identified by an index.

element: One of the values in an array. The [] operator selects elements.

index: An integer variable or value used to indicate an element of an array.

deterministic: A program that does the same thing every time it is invoked.

pseudorandom: A sequence of numbers that appear to be random, but which are actually the product of a deterministic computation.

histogram: An array of integers where each integer counts the number of values that fall into a certain range.

12.12 تمرينات

تمرين 12.6 اكتب عملية باسم areFactors تأخذ عدداً صحيحاً n ومصفوفة أعداد صحيحة، وتعيد القيمة true إذا كانت جميع الأرقام في المصفوفة عواملاً للعدد n (أي أن n يقبل القسمة عليهم جميعاً). مساعدة: انظر التمرين 6.1.

تمرين 12.7 اكتب عملية تأخذ مصفوفة أعداد صحيحة وعدد صحيح اسمه target كمتحولات، وتعيد الدليل الأول الذي يظهر target عنده في المصفوفة، وذلك في حال ظهوره، و1- فيما عدا ذلك.

تمرين 12.8 بعض المبرمجين يخالفون القاعدة العامة التي تقول بأن أسماء المتغيرات والعمليات يجب أن تكون ذات معنى يعبر عن وظيفتها. بدلاً من ذلك، يعتقد هؤلاء أن المتغيرات والعمليات يجب أن تسمى تيمناً بالفاكهة.

لكل واحدة من العمليات التالية، اكتب جملة واحدة تصف بشكل مجرد ما تفعله تلك العملية. ولكل متغير، اكتب جملة تعرف الدور الذي يؤديه.

```
public static int banana(int[] a) {
   int grape = 0;
   int i = 0;
   while (i < a.length) {
      grape = grape + a[i];
      i++;
   }
   return grape;
}

public static int apple(int[] a, int p) {
   int i = 0;
```

```
   int pear = 0;
   while (i < a.length) {
      if (a[i] == p) pear++;
      i++;
   }
   return pear;
}

public static int grapefruit(int[] a, int p) {
   for (int i = 0; i<a.length; i++) {
      if (a[i] == p) return i;
   }
   return -1;
}
```

الغرض من هذا التمرين هو التدرب على قراءة الشفرة والتعرف على أماط الحسابات التي شاهدناها من قبل.

تمرين 12.9

a. ما هو خرج البرنامج التالي؟
b. ارسم مخططاً هرمياً يبين حالة البرنامج قبيل عودة mus.
c. صف ما تفعله mus في بضعة كلمات.

```
public static int[] make(int n) {
  int[] a = new int[n];

  for (int i=0; i<n; i++) {
    a[i] = i+1;
  }
  return a;
}

public static void dub(int[] jub) {
  for (int i=0; i<jub.length; i++) {
    jub[i] *= 2;
  }
}

public static int mus(int[] zoo) {
  int fus = 0;
  for (int i=0; i<zoo.length; i++) {
    fus = fus + zoo[i];
  }
  return fus;
}

public static void main(String[] args) {
  int[] bob = make(5);
  dub(bob);
  System.out.println(mus(bob));
}
```

تمرين 12.10 معظم أساليب اجتياز المصفوفات التي رأيناها حتى الآن يمكن كتابتها بشكل تعاودي أيضاً. إن عمل ذلك ليس شائعاً، لكنه تمرين مفيد.

a. اكتب عملية باسم maxInRange تأخذ مصفوفة أعداد صحيحة ومجال من الأدلة (lowIndex وhighIndex)، وتوجد القيمة العظمى في المصفوفة، وذلك بمعالجة العناصر بين lowIndex وhighIndex فقط، بما في ذلك نهايتي المجال.
يجب أن تكون هذه العملية تعاودية. إذا كان طول المجال 1، أي إذا كان lowIndex == highIndex، سنعرف مباشرة أن العنصر الوحيد الوجود في المجال لا بد أن يكون أكبر قيمة موجودة. إذن تلك هي الحالة الأساسية (base case).
في حال وجود أكثر من عنصر واحد في المجال، يمكننا أن نجزئ المصفوفة إلى أجزاء، نوجد القيمة العظمى في كل قطعة، وبعدها نوجد القيمة العظمى للقيم العظمى.
b. العمليات مثل maxInRange قد تكون صعبة الاستخدام. لإيجاد العنصر الأكبر في مصفوفة، علينا تزويدها بمجال يحتوي المصفوفة بالكامل.

```
double max = maxInRange(array, 0, a.length-1);
```

اكتب عملية باسم max تأخذ مصفوفة كمعامل وتستعمل maxInRange لإيجاد وإعادة القيمة العظمى. أحياناً تدعى العمليات التي تشبه max **بالعمليات المغلفة (wrapper methods)**، لأنها تشكل طبقة عازلة حول العملية الغريبة وتجعل استخدامها أسهل. تدعى العملية التي تجري الحساب الفعلي **بالعملية المساعدة (helper mthod)**.

اكتب نسخة تعاودية من العملية find باستخدام أسلوب العمليات المغلفة والمساعدة (wrapper-helper pattern). يجب أن تأخذ find مصفوفة أعداد صحيحة وعدد صحيح كهدف. عليها أن تعيد دليل الموقع الأول الذي يظهر فيه الهدف في المصفوفة، أما في حال عدم ظهوره تعيد القيمة 1-.

تمرين 12.11 من الطرق الغير فعالة كثيراً في ترتيب عناصر مصفوفة هي البحث عن العنصر الأكبر وتبديله مع العنصر الأول، بعدها إيجاد العنصر الأصغر منه مباشرة وتبديله مع العنصر الثاني، وهكذا. هذه الطريقة تدعى **الترتيب الانتقائي (selection sort)** – انظر http://en.wikipedia.com/wiki/Selection_sort.

a. اكتب عملية باسم indexOfMaxInRange تأخذ مصفوفة أعداد صحيحة وتوجد العنصر الأكبر في المجال المعطى، وتعيد دليله. يمكنك تعديل نسختك التعاودية من العملية maxInRange أو يمكنك كتابة نسخة تكرارية من الصفر.
b. اكتب عملية باسم swapElement تأخذ مصفوفة أعداد صحيحة ودليلين، وتبدل بين العنصرين الموجودين عند الدليلين المعطيين.
c. اكتب عملية باسم selectionSort تأخذ مصفوفة أعداد صحيحة وتستخدم indexOfMaxInRange وswapElement لترتيب المصفوفة من الأكبر إلى الأصغر.

تمرين 12.12 اكتب عملية باسم letterHist تأخذ سلسلة محرفية كمعامل وتعيد مخطط الأعمدة لأحرف السلسلة المحرفية. العنصر الصفري من المخطط يجب أن يحتوي على عدد أحرف a في السلسلة (كبير أو صغير)؛ العنصر الخامس والعشرون يجب أن يحتوي على عدد أحرف z. يجب على برنامجك أن يجتاز السلسلة المحرفية مرة واحدة.

تمرين 12.13 يقال عن الكلمة أنها doubloon إذا ظهر كل حرف من حروف الكلمة مرتين فيها. مثلاً، الكلمات التالية هي جميع الdoubloons التي وجدتها في قاموسي.

Abba, Anna, appall, appearer, appeases, arraigning, beriberi, bilabial, boob, Caucasus, coco, Dada, deed, Emmett, Hannah, horseshoer, intestines, Isis, mama, Mimi, murmur, noon, Otto, papa, peep, reappear, redder, sees, Shanghaiings, Toto

اكتب عملية باسم isDoubloon تعيد قيمة true إذا كانت الكلمة المعطاة doubloon وfalse فيما عدا ذلك.

تمرين 12.14 يقال عن كلمتين أنهما anagram (جناس تصحيفي) إذا كان لهما نفس الحروف (ونفس عدد التكرار لكل حرف). مثلاً، إذا أعدنا ترتيب "stop" ستصبح "pots"، و"allen downey" ستصبح "well annoyed".

اكتب عملية تأخذ سلسلتين محرفيتين وتعيد true إذا كان للسلسلتين نفس الحروف (لكن الترتيب مختلف).

تحد اختياري: اجعل البرنامج يقرأ محارف السلسلتين مرة واحدة فقط.

تمرين 12.15 في لعبة سكرابل، لكل لاعب مجموعة من القطع المكتوب عليها حروف، ويكون الهدف من اللعبة هو استعمال هذه الأحرف لتكوين كلمات. نظام حساب النقاط معقد، لكن الكلمات الأطول تساوي أكثر من الكلمات الأقصر في العادة.

تخيل أنك أعطيت مجموعة من القطع بشكل سلسلة محرفية، مثل "quijibo" وأنك أعطيت سلسلة أخرى لتختبرها، مثل "jib". اكتب عملية باسم canSpell تأخذ سلسلتين محرفيتين وتعيد true إذا كان تكوين الكلمة باستخدام مجموعة القطع ممكناً. يمكن أن تملك أكثر من قطعة بنفس الحرف، لكن لا يمكنك استعمال القطعة الواحدة أكثر من مرة.

تحد اختياري: اقرأ حروف السلسلتين مرة واحدة فقط.

تمرين 12.16 في لعبة سكرابل الحقيقية، يوجد قطع فارغة يمكن استعمالها لتمثل أي حرف مطلوب (أو ما يدعى wild card).

فكر بخوارزمية للعملية canSpell يمكن لها أن تعالج المحارف العامة. لا تشغل نفسك بتفاصيل المجريات مثل كيفية تمثيل الأحرف العامة. قم بوصف الخوارزمية وحسب، سواء باستعمال اللغة الطبيعية، أو باستعمال طريقة "الشفرة الزائفة" — pseudocode، أو Java.

تُركت هذه الصفحة بيضاء عن عمد

الفصل 13

مصفوفات الكائنات

13.1 الطريق القادم

في الفصول الثلاثة القادمة سنطور برامجاً لتعمل مع لعب الورق ومجموعات ورق اللعب. قبل أن نبدأ، سأقدم لك فكرة عن الخطوات:

1. في هذا الفصل سنعرف صنف Card ونكتب عمليات تعمل مع أوراق اللعب ومصفوفات الأوراق.
2. في الفصل 14 سننشئ صنف Deck ونكتب عمليات تشتغل على مجموعات الورق.
3. في القسم 14.9 سأقدم البرمجة كائنية التوجه (OOP) وسنحول صنفي Card وDeck إلى أسلوب أقرب إلى OOP.

أعتقد أن أسلوب العمل هذا يجعل الطريق أسهل؛ العقبة الوحيدة هي أننا سنرى العديد من النسخ المعدلة لنفس الشفرة، ما قد يسبب بعض الحيرة. إذا ذلك كان سيساعدك، فيمكنك تنزيل شفرة كل فصل لتسهيل العمل. شفرة هذا الفصل متوفرة على: http://thinklikecs.webs.com/resources/code/Card1.java

13.2 كائنات Card

إذا لم تكن تعرف كيف تلعب الورق، فسيكون الآن وقتاً مناسباً لتشتري مجموعة ورق، وإلا فلن تفهم شيئاً من هذا الفصل. أو اقرأ http://en.wikipedia.org/wiki/Playing_card.

يوجد 52 ورقة لعب في المجموعة؛ كل منها تنتمي لأحد المنظومات الأربعة وواحدة من الرتب الثلاثة عشر. المنظومات هي: الاسباتي، كوبة، دياري، والزهر (مرتبة تنازلياً بحسب قوتها في لعبة بريدج). الرتب هي آص، 2، 3، 4، 5، 6، 7، 8، 9، 10، صبي، بنت وشيخ. وبحسب اللعبة التي تلعبها، إما أن يعتبر الآص أعلى من الشيخ أو أدنى من 2.

إذا أردت تعريف صنف جديد لتمثيل أوراق اللعب، فمن الواضح أن متغيرات الحالة يجب أن تكون: rank (للرتبة)، وsuite (للمنظومة). لا يبدو نوع هذه المتغيرات واضحاً. من الاحتمالات الممكنة هو نوع String، ويمكن أن يحتوي على شيء مثل "Spade" لمنظومة الاسباتي و"Queen" لورقة البنت. من علل هذا الأسلوب هو صعوبة المقارنة بين الأوراق لنعرف أي منها أعلى رتبة أو منظومة.

من الحلول البديلة استخدام الأعداد الصحيحة **لترميز (encode)** الرتب والمنظومات. أنا لا أعني بكلمة "ترميز" التشفير أو التحويل إلى شفرة سرية، وهو ما قد يعتقده البعض. ما يعنيه عالم الكمبيوتر بكلمة "ترميز" هو شيء مثل "تحديد خريطة تربط بين سلسلة من الأرقام وبين أشياء أخرى أريد تمثيلها." مثلاً،

اسباتي ← 3
كوبة ← 2
دياري ← 1
زهر ← 0

الميزة الواضحة لهذا التخطيط هي استبدال المنظومات بالأرقام الصحيحة بالترتيب، حتى نتمكن من مقارنة المنظومات بالمقارنة بين الأعداد الصحيحة. إن خريطة الرتب واضحة فعلاً؛ كل واحدة من رتب الأرقام ترتبط بالعدد الصحيح المقابل، وبالنسبة لأوراق الصور:

صبي ← 11
بنت ← 12
شيخ ← 13

إن سبب استخدامي للتدوين الرياضي لهذه الخرائط هو أنها ليست جزءاً من البرنامج. هي جزء من تصميم البرنامج، لكنها لن تظهر صراحة في الشفرة أبداً. تعريف الصنف للنوع Card يبدو مثل هذا:

```
class Card
{
int suit, rank;

public Card() {
this.suit = 0; this.rank = 0;
}

public Card(int suit, int rank) {
this.suit = suit; this.rank = rank;
}
}
```

كالعادة، أكتب عمليتين بانيتين: واحدة تأخذ معامل لكل متغير حالة؛ والثانية لا تأخذ أية معاملات.

لإنشاء كائن يمثل 3 الزهر، نستدعي new:

```
Card threeOfClubs = new Card(0, 3);
```

المتحول الأول، 0 يعبر عن منظومة الزهر.

13.3 عملية printCard

عندما تنشئ صنفاً جديداً، فإن الخطوة الأولى هي التصريح عن متغيرات الحالة وكتابة العمليات البانية. الخطوة الثانية هي كتابة العمليات القياسية التي يجب أن يملكها كل كائن، بما فيها عملية لطباعة الكائن، وواحدة أخرى أو اثنتين للمقارنة بين الكائنات. دعنا نبدأ مع العملية printCard.

لطباعة كائنات Card بطريقة يقدر البشر على قراءتها بسهولة، علينا عمل خريطة تربط بين الرموز العددية وما يقابلها من كلمات. إحدى الطرق الطبيعية لعمل ذلك هي استعمال مصفوفة سلاسل محرفية. يمكنك إنشاء مصفوفة سلاسل محرفية بنفس الطريقة التي تنشئ فيها مصفوفة لأحد الأنواع البسيطة:

```
String[] suits = new String [4];
```

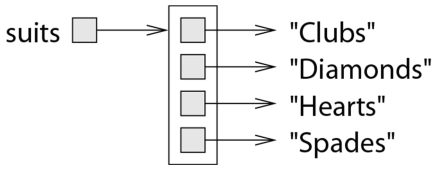
ثم نصبح قادرين على ضبط قيم عناصر المصفوفة.

```
suits[0] = "Clubs";
suits[1] = "Diamonds";
suits[2] = "Hearts";
suits[3] = "Spades";
```

إن إنشاء مصفوفة وتهيئة عناصرها بالقيم هي عملية شائعة لدرجة أن Java توفر تعليمة خاصة لها:

```
String[] suits = { "Clubs", "Diamonds", "Hearts", "Spades" };
```

تكافئ هذه التعليمة إلى التصريح عن المصفوفة بشكل منفصل، وحجز مكان لها في الذاكرة، وتعليمات الإسناد. يبدو مخطط الحالة لهذ المصفوفة كما يلي:



عناصر هذه المصفوفة ليست سلاسلاً محرفية، بل هي مرجعيات (references) للسلاسل المحرفية الفعلية.

سنحتاج الآن لمصفوفة سلاسل محرفية أخرى لفك تشفير الرتب:

```
String[] ranks = { "narf", "Ace", "2", "3", "4", "5", "6",
             "7", "8", "9", "10", "Jack", "Queen", "King" };
```

إن سبب وجود "narf" هو حفظ مكان العنصر الصفري من المصفوفة، الذي لن نستعمله أبداً (أو لا يفترض بنا أن نستعمله). الرتب الصالحة هي من 1 حتى 13. كان من الممكن أن نبدأ من الصفر لتجنب هذا العنصر المهدور، لكن الخريطة ستبدو أكثر منطقية لو رمزنا 2 ب2، و3 ب3، الخ.

باستخدام هذه المصفوفات، يمكننا الوصول إلى السلاسل المحرفية المناسبة باستخدام suit وrank كأدلة. في العملية printCard

```
public static void printCard(Card c) {
String[] suits = { "Clubs", "Diamonds", "Hearts", "Spades" };
String[] ranks = { "narf", "Ace", "2", "3", "4", "5", "6",
             "7", "8", "9", "10", "Jack", "Queen", "King" };
System.out.println(ranks[c.rank] + " of " + suits[c.suit]);
}
```

تعني العبارة suits[c.suit] "استخدم متغير الحالة suit من الكائن c كدليل للمصفوفة المدعوة باسم suits، واختر السلسلة المحرفية المناسبة". خرج هذه الشفرة

```
Card card = new Card(1, 11);
printCard(card);
```

هو Jack of Diamonds.

13.4 عملية sameCard

كلمة "same" أو "نفس" هي أحد الأشياء التي تطرأ في اللغة الطبيعية وتبدو واضحة تماماً حتى تفكر فيها قليلاً، وعندها ستستنتج أنها تملك أكثر مما تتوقعه منها.

مثلاً، لو قلت "أنا وكريس نملك نفس السيارة"، فأنا أعني أن سيارتينا من نفس الماركة والموديل، لكنهما سيارتين مختلفتين. لو قلت "أنا وكريس لنا نفس الأم"، فأنا أعني أن أمنا هي امرأة واحدة. لذا فإن فكرة "التطابق" تختلف اعتماداً على السياق.

عندما تتكلم عن الكائنات، يوجد غموض مشابه. مثلاً، إذا كانت ورقتين (two Cards) متطابقتين، فهل يعني هذا أنهما يحويان نفس البيانات (الرتبة والمنظومة)، أو أنهما فعلاً كائن Card وحيد؟

لنتحقق فيما إذا كان مرجعين يشيران إلى نفس الكائن، نستخدم عامل ==. مثلاً:

```
Card card1 = new Card(1, 11);
Card card2 = card1;
```

```
if (card1 == card2) {
System.out.println("card1 and card2 are identical.");
}
```

المرجعيات التي تشير إلى نفس الكائن **متطابقة (identical)**. أما المرجعيات التي تشير إلى كائنات تحوي نفس البيانات فهي **متساوية (equivalent)**.

من الشائع كتابة عملية للتحقق من المساواة، وتدعى باسم مثل sameCard.

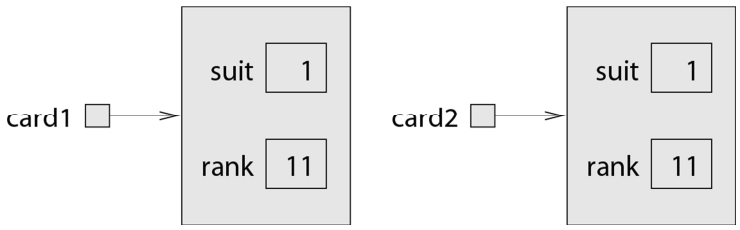
```
public static boolean sameCard(Card c1, Card c2) {
return(c1.suit == c2.suit && c1.rank == c2.rank);
}
```

هذا مثال ينشئ كائنين لهما نفس البيانات، ويستعمل sameCard ليرى إذا كانا متساويين:

```
Card card1 = new Card(1, 11);
Card card2 = new Card(1, 11);
if (sameCard(card1, card2)) {
System.out.println("card1 and card2 are equivalent.");
}
```

إذا كان المرجعين متطابقين، فهما متساويين أيضاً، لكن إذا كانا متساويين، فليس بالضرورة أن يكونا متطابقين.

في هذه الحالة، card1 وcard2 متساويين لكنهما غير متطابقين، لذا سيبدو مخطط الحالة كهذا:



كيف سيدو مخطط الحالة لو كان card1 وcard2 متطابقين؟

في القسم 8.10 أخبرتك ألا تستعمل == مع السلاسل المحرفية لأنه لن يعطيك ما تتوقعه منه. بدلاً من مقارنة محتويات السلسلتين (المساواة)، سيتحقق فيما إذا كانت السلستين نفس الكائن (المطابقة).

13.5 عملية compareCard

بالنسبة للأنواع البسيطة، تقارن العوامل الشرطية القيم وتحدد أيها أكبر أو أصغر من الآخر. هذه العوامل (> و < وغيرها) لا تعمل مع الأنواع الكائنية. بالنسبة للسلاسل المحرفية توفر Java عملية compareTo. بالنسبة لأوراق اللعب علينا كتابة واحدة بنفسنا، والتي سنسميها compareCard. لاحقاً، سنستخدم هذه العملية لترتيب مجموعة أوراق اللعب.

بعض المجموعات مرتبة كلياً، بمعنى أنك تستطيع المقارنة بين أي عنصرين وأن تعرف أيهما أكبر. الأعداد الصحيحة والعشرية هي مجموعات مرتبة كلياً. بعض المجموعات غير مرتبة، ما يعني عدم وجود طريقة منطقية لنقول أن عنصراً ما أكبر من عنصر آخر. الفاكهة هي مجموعة غير مرتبة، ولذلك لا نستطيع المقارنة بين التفاح والبرتقال. في Java، النوع البولياني غير مرتب؛ لا يمكننا القول أن true أكبر من flase.

مجموعة ورق اللعب مرتبة جزئياً، أي أننا نستطيع المقارنة بين الأوراق أحياناً وأحياناً لا نستطيع. مثلاً، أنا أعلم أن 3 الزهر أعلى من 2 زهر، وأن 3 ديناري أعلى من 3 زهر. لكن أيهما أعلى، 3 زهر أم 2 ديناري؟ أحدهما من رتبة أعلى، لكن الآخر منظومته أعلى.

لنجعل الأوراق قابلة للمقارنة، علينا أن نقرر أيهما ذا قيمة أكثر، الرتبة أم المنظومة. الخيار عشوائي، لكن عندما تشتري مجموعة ورق جديدة، تأتي مرتبة وكل أوراق الزهر معاً، تليها أوراق الديناري وهكذا. لذا سنقول أن المنظومة ذات قية أكثر.

بعد أن اعتمدنا ذلك، يمكننا كتابة compareCard. تأخذ ورقتين (كائنين من نوع Card) كمعاملين وتعيد 1 إذا ربحت الورقة أولى، 1- إذا ربحت الورقة الثانية، و0 إذا كانتا متساويتين.

أولاً نقارن المنظومات:

```
if (c1.suit > c2.suit) return 1;
if (c1.suit < c2.suit) return -1;
```

إذا لم تتحقق أي من التعليمتين، فلا بد أن المنظومتين متساويتين، وعلينا مقارنة الرتب:

```
if (c1.rank > c2.rank) return 1;
if (c1.rank < c2.rank) return -1;
```

إذا لم تتحقق أي من هذه أيضاً، فلا بد أن الرتب متساوية، لذا نعيد القيمة 0.

تمرين 13.1 غلف هذه الشفرة في عملية. ثم عدلها بحيث تكون ورقة الآص أعلى من الشيخ.

13.6 مصفوفات الأوراق

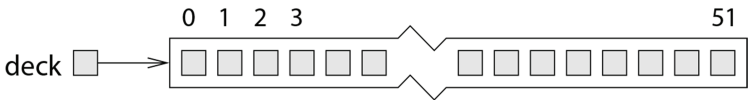
لقد شاهدنا عدة أمثلة عن التركيب (القدرة على جمع عدة مقومات من اللغة في عدة ترتيبات متنوعة) حتى الآن. من الأمثلة الأولى التي شاهدناها كان استدعاء عملية ضمن عبارة حسابية. مثال آخر كان البنية المتداخلة للتعليمات: يمكنك وضع تعليمة if ضمن حلقة while، أو ضمن تعليمة if أخرى، الخ.

بعد أن شاهدت هذه الأشكال، ودرست المصفوفات والكائنات، لا يجب أن تفاجأ عندما تعلم أنك تستطيع عمل مصفوفات من الكائنات. وأنك تستطيع تعريف كائنات تحوي مصفوفات كمتغيرات حالة؛ يمكنك عمل مصفوفات تحتوي على مصفوفات؛ يمكنك تعريف كائنات تحتوي على كائنات، وهكذا. سنرى في الفصلين القادمين أمثلة عن هذه التراكيب باستخدام كائنات Card.

هذا المثال ينشئ مصفوفة من 52 ورقة:

```
Card[] cards = new Card [52];
```

هذا هو مخطط الحالة لهذا الكائن:



تحتوي المصفوفة على مرجعيات للكائنات؛ ولا تحتوي على كائنات Card ذاتها. يتم تهيئة العناصر بالقيمة null. يمكنك الوصول إلى العناصر في المصفوفة بالطريقة المعتادة:

```
if (cards[0] == null) {
System.out.println("No cards yet!");
}
```

لكن إذا حاولت الولوج إلى متغير حالة لكائن Card غير موجود، ستحصل على NullPointerException.

```
cards[0].rank;            // NullPointerException
```

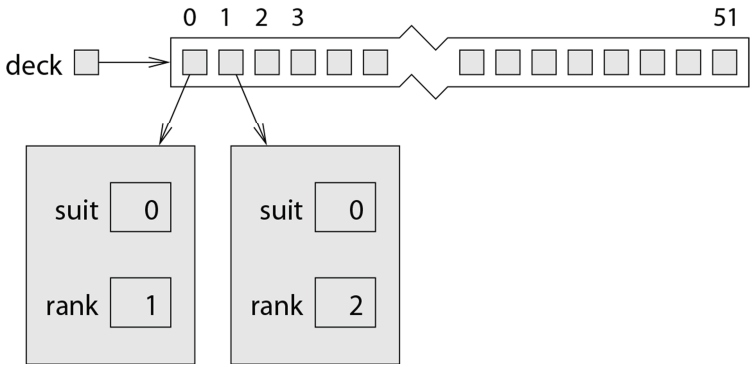
لكن هذه هي البنية الصحيحة للوصول إلى رتبة الورقة "الصفرية" من المجموعة. كما أن هذا مثال عن التركيب، يجمع البنية النحوية المستخدمة في الوصول إلى عنصر من مصفوفة ومتغير حالة في كائن.

أسهل طريقة لتعبئة مجموعة اللعب بكائنات Card هي كتابة حلقة متداخلة:

```
int index = 0;
for (int suit = 0; suit <= 3; suit++) {
for (int rank = 1; rank <= 13; rank++) {
cards[index] = new Card(suit, rank);
index++;
}
}
```

تعد الحلقة الخارجية الرتب من 0 إلى 3. ومن أجل لكل منظومة، تعد الحلقة الداخلية الرتب من 1 إلى 13. وبما أن الحلقة الخارجية تشتغل 4 مرات، والداخلية 13 مرة، فسيتم تنفيذ جسم الحلقة 52 مرة.

استخدمت index لمعرفة الموقع الذي يجب أن تذهب إليه الورقة التالية في مجموعة اللعب. يبين مخطط الحالة التالي شكل المجموعة بعد حجز أول ورقتين:



تمرين 13.2 غلف شفرة بناء مجموعة ورق اللعب هذه في عملية باسم makeDeck لا تأخذ أية معاملات وتعيد مصفوفة أوراق ممتلئة بالكامل.

13.7 عملية printDeck

عندما تعمل مع المصفوفات، من المناسب أن تملك عملية لطباعة المحتويات. لقد رأينا نموذج عبور المصفوفة عدة مرات، لذا يجب أن تكون الشفرة التالية مألوفة:

```
public static void printDeck(Card[] cards) {
for (int i=0; i < cards.length; i++) {
printCard(cards[i]);
}
}
```

بما أن cards من نوع []Card، فإن العنصر من cards سيكون من النوع Card. لذا فإن cards[i] سيكون متحولاً مشروعاً للعملية printCard.

13.8 البحث

ستكون العملية التالية التي سأكتبها هي findCard، التي ستبحث في مصفوفة أوراق لترى إذا كانت تحتوي على ورقة معينة. ستعطيني هذه العملية الفرصة لأشرح خوارزميتين: **البحث الخطي (linear search)** و**البحث المجزّأ (bisection search)**.

البحث الخطي واضح للغاية؛ نجتاز مجموعة الورق ونقارن كل ورقة بالورقة التي نبحث عنها. إذا عثرنا عليها نعيد الدليل الذي تظهر عنده الورقة. إذا لم تكن موجودة في مجموعة الورق، نعيد القيمة 1-.

```
public static int findCard(Card[] cards, Card card) {
for (int i = 0; i< cards.length; i++) {
if (sameCard(cards[i], card)) {
return i;
}
}
return -1;
}
```

متحولات العملية findCard هي card وcards. قد يبدو وجود متحول بنفس اسم النوع غريباً (المتغير card من النوع Card). يمكننا أن نميز بينهما لأن المتغير يبدأ بحرف صغير.

ترجع العملية بمجرد اكتشاف الورقة، ما يعني أننا لا نحتاج للمرور على مجموعة الورق بالكامل إذا وجدنا الورقة التي كنا نبحث عنها. إذا وصلنا إلى آخر الحلقة، سنعرف أن الورقة غير موجودة في المجموعة.

إذا لم تكن الأوراق في المجموعة مرتبة، لا توجد أي طريقة أسرع للبحث من هذه. علينا فحص كل ورقة لأننا لا نملك طريقة أخرى تمكننا من التأكد أن الورقة التي نبحث عنها ليست هذه الورقة.

لكن عندما تبحث عن كلمة في قاموس، فأنت لا تبحث عنها خطياً، لأن الكلمات مرتبة أبجدياً. ولذلك ستستخدم خوارزمية مشابهة لخوارزمية البحث المجزأ:

1. ابدأ في مكان ما في الوسط.
2. اختر كلمة في الصفحة وقارنها مع الكلمة التي تبحث عنها.
3. إذا وجدت الكلمة التي تبحث عنها، توقف.
4. إذا كانت الكلمة التي تبحث عنها تأتي بعد الكلمة الموجودة في هذه الصفحة، قلّب الصفحات إلى مكان لاحق في القاموس وانتقل إلى الخطوة 2.
5. إذا كانت الكلمة التي تبحث عنها تأتي قبل الكلمة الموجودة في هذه الصفحة، قلّب الصفحات إلى مكان سابق في القاموس وانتقل إلى الخطوة 2.

إذا وصلت إلى نقطة حيث توجد كلمتان متجاورتان في الصفحة وكان من المفروض أن توجد كلمتك بينهما، ستستنتج أن كلمتك غير موجودة في القاموس.

لنعد إلى مجموعة ورق الشدة، إذا علمنا أن الأوراق مرتبة، يمكننا كتابة نسخة أسرع من findCard. أفضل طريقة لكتابة البحث المجزأ هي باستخدام عملية تعاودية، لأن التجزئة تعاودية بطبيعتها.

الحيلة هي كتابة عملية باسم findBisect تأخذ دليلين كمعاملات، low وhigh، يشيران إلى القطعة من المصفوفة التي سيتم البحث فيها (تتضمن العنصر ذا الدليل low وكذا الدليل high).

1. للبحث في المصفوفة، اختر دليلاً بين low وhigh (سمّه mid) وقارن العنصر الموافق له مع الورقة التي تبحث عنها.
2. إذا عثرت عليها توقف.
3. إذا كانت الورقة mid أعلى من الورقة التي تبحث عنها، ابحث في النطاق من low إلى mid-1.

4. إذا كانت الورقة mid أدنى من ورقتك، ابحث في النطاق من mid+1 إلى high.

تبدو الخطوتين 3 و4 كاستدعاءات تعاودية مريبة. هذا ما ستبدو عليه هذه الخوارزمية بعد ترجمتها إلى شفرة Java:

```
public static int findBisect(Card[] cards, Card card, int low, int high) {
// TODO: need a base case
int mid = (high + low) / 2;
int comp = compareCard(cards[mid], card);

if (comp == 0) {
return mid;
} else if (comp > 0) {
return findBisect(cards, card, low, mid-1);
} else {
return findBisect(cards, card, mid+1, high);
}
}
```

هذه الشفرة تحتوي على لب البحث المجزأ، لكن تنقصها قطعة، ولذلك أضفت تعليق TODO (يلزم عمله).

كما هو مكتوب، ستظل العملية تستدعي نفسها للأبد إذا لم تكن الورقة موجودة في المجموعة. نحتاج إلى حالة قاعدية لمعالجة هذه الحالة.

إذا كان high أصغر من low، فلا توجد أوراق بينهما، كما ترى فقد استنتجنا أن تلك الورقة غير موجودة في المجموعة. إذا عالجنا هذه الحالة، ستعمل العملية بشكل صحيح:

```
public static int findBisect(Card[] cards, Card card, int low, int high) {
System.out.println(low + ", " + high);

if (high < low) return -1;

int mid = (high + low) / 2;
int comp = cards[mid].compareCard(card);

if (comp == 0) {
return mid;
} else if (comp > 0) {
return findBisect(cards, card, low, mid-1);
} else {
return findBisect(cards, card, mid+1, high);
}
```

{أضفت تعليمة طباعة حتى أتمكن من متابعة تتابع الاستدعاءات التعاودية. لقد جربت الشفرة التالية:

```
Card card1 = new Card(1, 11);
System.out.println(findBisect(cards, card1, 0, 51));
```

وحصلت على الخرج التالي:

```
0, 51
0, 24
13, 24
19, 24
22, 24
23
```

ثم اخترعت ورقة غير موجودة في المجموعة (15 دينارى)، وحاولت العثور عليها. حصلت على ما يلي:

```
0, 51
0, 24
13, 24
13, 17
13, 14
13, 12
-1
```

هذه الاختبارات لا تثبت أن هذا البرنامج صحيح. في الواقع، لا توجد أية كمية من الاختبارات يمكن أن تبرهن أن برنامجاً ما يعمل بشكل صحيح. من جهة أخرى، بتجربة عدة حالات واختبار الشفرة، قد تقدر على إقناع نفسك بذلك.

عدد الاستدعاءات التعاودية نموذجياً 6 أو 7، لذا فنحن نستدعي compareCard 6 أو 7 مرات، مقارنة بما قد يصل إلى 52 مرة إذا اعتمدنا البحث الخطي. بشكل عام، البحث المجزأ أسرع بكثير من البحث الخطي، ويزداد فرق السرعة كلما ازداد حجم المصفوفة.

خطأين شائعي الحدوث في البرامج التعاودية هما نسيان تضمين حالة قاعدية أو كتابة الاستدعاء التعاودي بحيث لا يصل البرنامج للحالة القاعدية أبداً. كلا الخطأين سيسبب تعاوداً لانهائياً، الذي سيولد StackOverflowEception.

13.9 مجموعات ورق الشدة والمجموعات الفرعية

هذا هو النموذج الأولي للعملية findBisect:

```
public static int findBisect(Card[] deck, Card card, int low, int high)
```

يمكننا اعتبار cards، وlow وhigh كمعامل وحيد يمثل **مجموعة فرعية (subdeck)**. هذه الطرقة في التفكير شائعة جداً، وأحياناً أعتبرها **كمعامل مجرد (abstract parameter)**. ما أعنيه بكلمة "مجرد"، هو شيء ليس جزءاً من نص البرنامج بشكل فعلي، لكنه يصف عمل البرنامج على مستوى أعلى.

مثلاً، عندما تستدعي عملية وتمرر لها مصفوفة وlow وhigh اللذان يمثلان الحدود، لا يوجد شيء يمنع العملية المستدعاة من الوصول إلى أجزاء المصفوفة الواقعة خارج النطاق المعطى. لذا فأنت فعلياً لا ترسل مجموعة فرعية من مجموعة الورق؛ بل ترسل المجموعة كاملة. لكن طالما أن المستقبِل يلعب وفق القواعد، فمن المعقول أن تعتبرها مجموعة فرعية، بشكل مجرد.

هذا النمط من التفكير، الذي يعطي البرنامج معنى أبعد مما هو مكتوب فعلياً، جزء مهم من أسلوب التفكير كعالم كمبيوتر.

لقد استخدمت الكلمة "مجرد" في العديد من الأماكن حتى كادت تفقد معناها. مع ذلك، فالتجريد فكرة مركزية في علوم الحاسوب (والعديد من المجالات الأخرى).

لتعريف "التجريد" بشكل أكثر عمومية نقول "هو عملية قولبة لنظام معقد باستخدام وصف مبسط للتخلص من التفاصيل غير الضرورية والحصول على سلوك مناسب".

13.10 المصطلحات

ترميز: تمثيل مجموعة قيم باستخدام مجموعة قيم أخرى، وذلك بصنع خريطة تربط بينهما.

التطابق: تساوي المرجعيات. مرجعين يشيران إلى كائن واحد.

التساوي: تساوي القيم. مرجعين يشيران إلى كائنين يحويان نفس البيانات.

معامل مجرد: مجموعة من المعاملات التي تعمل معاً عمل معامل واحد.

التجريد: عملية تفسير برنامج (أو أي شيء آخر) في مستوى أعلى مما هو ممثل بالشفرة فعلياً.

encode: To represent one set of values using another set of values, by constructing a mapping between them.

identity: Equality of references. Two references that point to the same object.

equivalence: Equality of values. Two references that point to objects that contain the same data.

abstract parameter: A set of parameters that act together as a single parameter.

abstraction: The process of interpreting a program (or anything else) at a higher level than what is literally represented by the code.

13.11 تمرينات

تمرين 13.3 في لعبة بلاك جاك يكون الهدف هو الحصول على ورق تكون نقاطه 21. تحسب نقاط اليد بجمع نقاط كافة الأوراق. الآص نقطة واحدة، أوراق الصور تساوي 10 نقاط، ونقاط بقية الأوراق تكون نفس رتبة الورقة. مثلاً: اليد (آص، 10، صبي، 3) تكون نقاطها 1 + 10 + 10 + 3 = 24.

اكتب عملية باسم handScore تأخذ مصفوفة أوراق كمتحول وتعيد مجموع النقاط.

تمرين 13.4 في البوكر يكون "flush" هو يد تحوي خمس أوراق أو أكثر من نفس المنظومة. يمكن أن تحوي اليد أي عدد من الأوراق.

a. اكتب عملية باسم suiHist تأخذ مصفوفة أوراق كمعامل وتعيد مخطط أعمدة لمنظومات الأوراق في اليد. يجب على الحل أن يجتاز المصفوفة مرة واحدة.
b. اكتب عملية باسم hasFlush تأخذ مصفوفة أوراق كمعامل وتعيد true إذا احتوت اليد على flush، وfalse فيما عدا ذلك.

تمرين 13.5 سيكون العمل مع الأوراق ممتعاً أكثر لو كنت تستطيع عرضها على الشاشة. إذا لم تكن قد عبثت مع الأمثلة الرسومية في الملحق A، فقد ترغب بعمل ذلك الآن.

بعدها نزل http://thinkapjava.com/code/CardTable.java
وhttp://thinkapjava.com/code/cardset.zip.

فك الضغط عن cardset.zip ثم شغل CardTable.java. يجب أن ترى حزمة من الأوراق تجلس على "طاولة" خضراء.

يمكنك استعمال هذا الصنف كنقطة انطلاق لصنع ألعاب الورق التي تريدها.

تُركت هذه الصفحة بيضاء عن عمد

الفصل 14

كائنات المصفوفات

تنْويه: في هذا الفصل، سنخطو خطوة أخرى نحو البرمجة كائنية التوجه، لكننا لن نصل إليها بعد. لذا فإن العديد من الأمثلة هنا ليست رسمية؛ أي أنها ليست برامج جيدة بلغة Java. هذه المرحلة الانتقالية ستساعدك على التعلم (آمل ذلك)، لكنني لا أكتب شفرة كهذه في الحالة العادية.

يمكنك تنزيل شفرة هذا الفصل من http://thinklikecs.webs.com/resources/code/Card2.java.

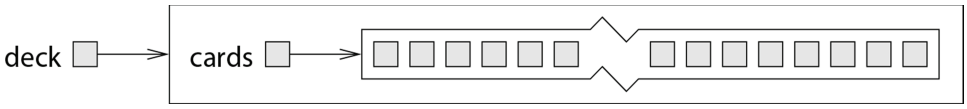
14.1 الصنف Deck

في الفصل السابق، اشتغلنا مع مصفوفة كائنات، لكنني ذكرت أيضاً أنه من الممكن وجود كائن يحتوي مصفوفة كمتغير حالة. في هذا الفصل سننشئ كائن Deck الذي يحتوي على مصفوفة Cards.

سيبدو تعريف الصنف كما يلي:

```
class Deck {
   Card[] cards;

   public Deck(int n) {
      this.cards = new Card[n];
   }
}
```

يهيئ الباني متغير الحالة بمصفوفة من الأوراق، لكنه لا ينشئ أية أوراق. هنا نجد مخطط حالة يبين مظهر Deck بدون أوراق:



هنا يوجد باني بدون متحولات يأخذ مجموعة ورق (deck) من 52 ورقة ويملأها بالأوراق (Cards):

```
public Deck() {
   this.cards = new Card[52];
   int index = 0;
   for (int suit = 0; suit <= 3; suit++) {
      for (int rank = 1; rank <= 13; rank++) {
         cards[index] = new Card(suit, rank);
         index++;
      }
   }
}
```

هذه العملية مشابهة للعملية makeDeck؛ لقد غيرنا البنية قليلاً لنجعلها عملية بناء. لاستدعائها، نستخدم new:

```
Deck deck = new Deck();
```

الآن سيكون من المعقول أن نضع العمليات المتعلقة بمجموعات الورق في تعريف الصنف Deck. بالنظر إلى العمليات التي كتبناها حتى الآن، أحد المرشحين الواضحين هي printDeck (القسم 13.7). ها هي هنا، بعد التعديل لتعمل مع Deck:

```
public static void printDeck(Deck deck) {
   for (int i=0; i < deck.cards.length; i++) {
      Card.printCard(deck.cards[i]);
   }
}
```

أحد التغييرات هو نوع المعامل، من []Card إلى Deck.

التغيير الثاني هو عدم قدرتنا على استخدام deck.length بعد ذلك للحصول على طول المصفوفة، لأن deck الآن هو كائن من نوع Deck، وليس مصفوفة. هو يحتوي على مصفوفة، لكنه ليس مصفوفة. لذا علينا كتابة deck.cards.length لاستخراج المصفوفة من كائن Deck والحصول على طولها.

وللسبب نفسه، علينا استخدام deck.cards[i] للولوج إلى عنصر من المصفوفة، بدلاً من deck[i] فقط.

التغيير الأخير هو أن استدعاء printCard يجب أن يبين صراحة أن printCard معرفة في الصنف Card.

14.2 خلط الأوراق

في معظم ألعاب الورق ستحتاج إلى القدرة على خلط الأوراق؛ وجعلها في ترتيب عشوائي. في القسم 12.6 رأينا كيفية توليد أرقام عشوائية، لكن استخدامها لخلط أوراق الشدة ليس واضحاً.

أحد الاحتمالات هو محاكاة الطريقة التي يخلط البشر بها الأوراق، وهي تكون عادة بتقسيم المجموعة إلى قسمين ثم اختيار الورق بالتبادل منهما. بما أن البشر لا يخلطون جيداً، سيكون الورق عشوائياً بشكل جيد بعد تكرار تلك العملية حوالي 7 مرات. لكن برنامج الحاسوب سيخلط الورق بشكل مثالي في كل مرة، وبعد 8 مرات من خلط الورق مثالياً، ستجد مجموعة الورق قد عادت إلى الترتيب الذي بدأت منه. لمزيد من المعلومات، انظر http://en.wikipedia.org/wiki/Faro_shuffle.

توجد خوارزمية أفضل لخلط أوراق اللعب وهي المرور على مجموعة الورق ورقة واحدة في كل مرة، ثم اختيار ورقتين عند كل تكرار والتبديل بينهما.

هنا تجد مخططاً لكيفية عمل هذه الخوارزمية. لتخطيط البرنامج، سأستعمل مزيجاً من تعليمات Java والكلمات الإنكليزية وهو ما يدعى أحياناً **بالشفرة الزائفة (pseudocode)**:

```
for (int i=0; i<deck.length; i++) {
   // choose a random number between i and deck.cards.length
   // swap the ith card and the randomly-chosen card
}
```

الشيء الجميل في الشفرة الزائفة هو أنها غالباً ما تجعل العمليات التي ستحتاج إليها واضحة. في هذه الحالة، سنحتاج شيئاً ما مثل randomInt، تختار عدداً صحيحاً عشوائياً بين low وhigh، وswapCards التي تأخذ دليلين وتبدل بين الأوراق الموجودة في الأماكن المشار إليها.

هذه العملية – كتابة الشفرة الزائفة ثم كتابة العمليات التي تعمل – تدعى **التطوير من الأعلى للأسفل (top-down development)** (انظر
.(http://en.wikipedia.org/wiki/Top-down_and_bottom-up_design

14.3 الترتيب

الآن وبعد أن أفسدنا مجموعة الورق، نحتاج طريقة إعادتها مرتبة كما كانت. توجد خوارزمية ترتيب تشابه خوارزمية الخلط بشكل معاكس. تدعى باسم **الترتيب الانتقائي (selection sort)** لأنها تعمل بالمرور على المصفوفة بشكل متكرر واختيار الورقة الأدنى المتبقية كل مرة.

خلال الدورة الأولى نبحث عن الرقة الأدنى ونستبدلها بالورقة الموجودة في الموقع 0. خلال الدورة رقم i، نبحث عن الورقة الأدنى على يمين i ونستبدلها مع الورقة الموجودة في الموقع i.

ها هي الشفرة الزائفة لخوارزمية الترتيب الانتقائي:

```
for (int i=0; i<deck.length; i++) {
   // find the lowest card at or to the right of i
   // swap the ith card and the lowest card
}
```

ثانية، تساعدنا الشفرة الزائفة على تصميم **العمليات المساعدة (helper methods)**. في هذه الحالة يمكننا استخدام swapCards مرة أخرى، لذا سنحتاج عملية جديدة واحدة فقط، تدعى indexLowestCard، تأخذ مصفوفة أوراق ودليل يجب أن تبدأ بالبحث عنده.

14.4 مجموعات الورق الفرعية

كيف يفترض بنا تمثيل يد أو مجموعة فرعية أخرى من مجموعة الورق الكاملة؟ أحد الاحتمالات هو إنشاء صنف جديد باسم Hand، والذي قد يوسع Deck (في Java يقال عن الصنف أنه يوسع –extends- صنفاً آخر إذا كان يرثه). يوجد احتمال آخر، وهو الذي سأشرحه، يكون باستخدام كائن Deck عدد أوراقه أقل من 52.

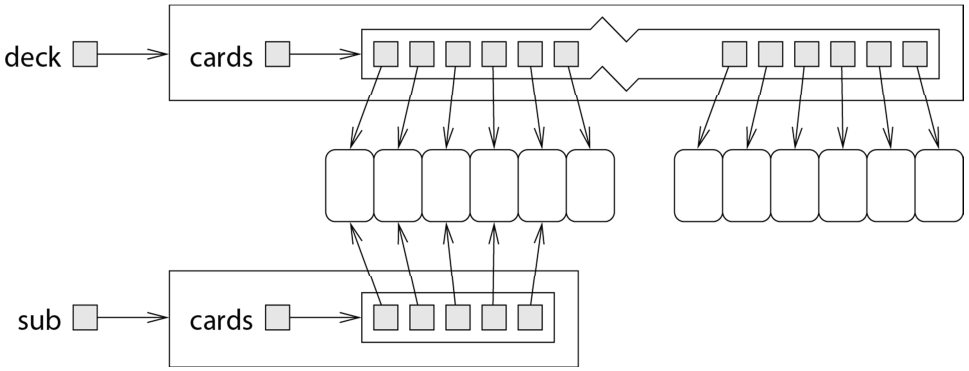
قد نرغب بعملية، subdeck، تأخذ Deck مع مجال من الأدلة، وتعيد مجموعة ورق جديدة تحتوي على المجموعة الفرعية المحددة من الأوراق:

```
public static Deck subdeck(Deck deck, int low, int high) {
   Deck sub = new Deck(high-low+1);

   for (int i = 0; i<sub.cards.length; i++) {
      sub.cards[i] = deck.cards[low+i];
   }
   return sub;
}
```

إن طول المجموعة الفرعية يساوي high-low+1 وذلك بسبب تضمين الورقتين الموجودتين عند الدليلين high وlow. هذا النوع من الحسابات قد يكون مضللاً، ويؤدي إلى خطأ "بفارق واحد". رسم شكل هو أفضل وسيلة في العادة لتفاديهم.

لأننا نعطي new متحولاً، فسيتم استدعاء الباني الأول، الذي يحجز المصفوفة فقط ولا يحجز أية أوراق. بداخل حلقة for، يتم تعبئة المجموعة الفرعية بنسخ عن مرجعيات مجموعة الورق.

فيما يلي مخطط الحالة لمجموعة فرعية تم إنشاؤها باستخدام المعاملات low=3 وhigh=7. النتيجة هي يد فيها 5 أوراق يتم مشاركتها مع المجموعة الأصلية؛ أي تم استخدام أسماء مستعارة.



التسمية المستعارة ليست فكرة جيدة عادة، لأن التغييرات في إحدى المجموعات الفرعية تؤثر على البقية، وهو ما يخالف السلوك المتوقع من أوراق الشدة ومجموعات ورق اللعب الحقيقة. لكن إذا كانت الأوراق غير قابلة للتعديل، ستكون التسمية المستعارة أقل خطورة. في هذه الحالة، لن يوجد أي سبب على الأغلب يدعونا لتغيير رتبة أو منظومة ورقة ما. بدلاً من ذلك يمكننا إنشاء كل ورقة مرة واحدة ثم معاملتها على أنها كائن غير قابل للتحوير. حتى الآن تبدو التسمية المستعارة مع كائنات Card خياراً معقولاً.

14.5 خلط الأوراق والتوزيع

في القسم 14.2 كتبت شفرة زائفة لخوارزمية خلط الأوراق. بفرض أننا نملك عملية باسم shuffleDeck تأخذ مجموعة ورق لعب كمعامل وتخلطها، يمكننا استعمالها لتوزيع الأوراق على اللاعبين:

```
Deck deck = new Deck();
shuffleDeck(deck);
Deck hand1 = subdeck(deck, 0, 4);
Deck hand2 = subdeck(deck, 5, 9);
Deck pack = subdeck(deck, 10, 51);
```

هذه الشفرة تضع الأوراق الخمسة الأولى في يد، والأوراق الخمسة التالية في يد أخرى، والباقي يعود إلى العلبة.

عندما فكرت بتوزيع الورق، هل تخيلت أننا سنعطي ورقة واحدة لكل لاعب بالتبادل كما هو شائع في ألعاب الورق الحقيقية؟ لقد فكرت بالموضوع، ثم تبين لي ن ذلك ليس ضرورياً بالنسبة لبرنامج حاسوب. اصطلاح توزيع الورق بالتبادل يقصد منه تقليل أثر خلط الورق غير المثالي وتصعيب الغش على من يوزع الورق. كلا السببين ليسا مشكلة بالنسبة للحاسوب.

هذا المثال سيذكرك بأحد أخطر مجازات الهندسة: أحياناً نفرض قيوداً غير ضرورية على الحواسيب، أو نتوقع منها إمكانيات لا تملكها، لأننا نوسع مجازاً ما أبعد من حدوده بدون وعي.

14.6 الترتيب بالدمج

في القسم 14.3، شاهدنا خوارزمية ترتيب بسيطة تبين فيما بعد أنها غير فعالة حقاً. لترتيب n عنصر، عليها عبور المصفوفة n مرة، وكل عبور يستهلك كمية من الزمن تتناسب مع n. وهو ما يؤدي إلى أن الزمن الكلي المستغرق يتناسب مع n^2.

في هذا القسم سأشرح خوارزمية أكثر فعالية تدعى **الترتيب بالدمج (mergesort)**. لترتيب n عنصر، تستهلك خوارزمية الترتيب بالدمج زمناً يتناسب مع n.log(n). قد لا يبدو هذا مبهراً، لكن عندما يكبر n، فإن الفرق بين n^2 و n.log(n) قد يكون شاسعاً. جرب بضعة قيم لn وانظر بنفسك.

إن الفكرة الأساسية للترتيب بالدمج هي: إذا كنت تملك مجموعتين فرعيتين، وكل منهما قد تم ترتيبها، فسيكون سهلاً (وسريعاً) أن تدمجهما في مجموعة واحدة مرتبة. جرب هذه مع مجموعة ورق شدة:

1. شكل مجموعتين فرعيتين من 10 أوراق تقريباً لكل منهما ورتبها بحيث تكون الورقة الأدنى في البداية عندما يكون وجه الأوراق إلى الأعلى. ضع المجموعتين أمامك ووجههما إلى الأعلى.
2. قارن بين الأوراق الأولى من كل مجموعة واختر الأدنى بينهما. اقلبها وأضفها إلى المجموعة المدمجة.
3. أعد الخطوة 2 حتى تنتهي إحدى المجموعتين. ثم خذ الأوراق الباقية وأضفها إلى المجموعة المدمجة.

يجب أن تكون النتيجة مجموعة واحدة مرتبة. هذا هو ما ستبدو عليه الخوارزمية باستخدام الشفرة الزائفة:

```
public static Deck merge(Deck d1, Deck d2) {
   // create a new deck big enough for all the cards
   Deck result = new Deck(d1.cards.length + d2.cards.length);

   // use the index i to keep track of where we are in
   // the first deck, and the index j for the second deck
   int i = 0;
   int j = 0;

   // the index k traverses the result deck
   for (int k = 0; k<result.cards.length; k++) {

      // if d1 is empty, d2 wins; if d2 is empty, d1 wins;
      // otherwise, compare the two cards

      // add the winner to the new deck
   }
   return result;
}
```

أفضل طريقة لاختبار merge هي إنشاء مجموعة ورق وخلطها، واستعمال subdeck لتشكيل مجموعتين فرعيتين (صغيرتين)، ثم استعمال عملية الترتيب من الفصل السابق لترتيب القسمين. ثم يمكنك تمرير هذين القسمين إلى merge لترى إذا كانت ستعمل.

إذا استطعت جعلها تعمل، جرب كتابة شكل بسيط للعملية mergeSort:

```
public static Deck mergeSort(Deck deck) {
   // find the midpoint of the deck
   // divide the deck into two subdecks
   // sort the subdecks using sortDeck
   // merge the two halves and return the result
}
```

بعد ذلك، إذا تمكنت من جعل تلك النسخة تعمل، يبدأ المرح الحقيقي! الشيء السحري في الترتيب بالدمج هو أنه تعاودي. عند ترتيب المجموعتين الجزئيتين، لم علينا استدعاء النسخة القديمة والبطيئة من sort؟ لم لا نستدعي العملية الجديدة الأنيقة mergeSort التي تكتبها الآن؟

هذه ليست مجرد فكرة جيدة، بل هي ضرورية للحصول على التحسن في الأداء الذي وعدتك به. لكن حتى تعمل عليك أن تملك حالة قاعدية؛ وإلا فسوف تستمر العملية بالتعاود للأبد. يمكن أن نعتبر مجموعة فرعية من ورقة واحدة أو مجموعة

لا تحوي أوراق كحالة قاعدية بسيطة. إذا استقبلت mergeSort مجموعة صغيرة كهذه، يمكن لها أن تعيدها مباشرة، لأنها مرتبة أصلاً.

ستبدو النسخة التعاودية من mergeSort مثل هذه:

```
public static Deck mergeSort(Deck deck) {
   // if the deck is 0 or 1 cards, return it

   // find the midpoint of the deck
   // divide the deck into two subdecks
   // sort the subdecks using mergesort
   // merge the two halves and return the result
}
```

كالعادة، توجد طريقتين للتفكير بالبرامج التعاودية: يمكنك تتبع مجرى التنفيذ كله، أو يمكنك القيام "بوثبة الثقة". لقد بنيت هذا المثال لأشجعك على "القفز بثقة".

عند استعمال sortDeck لترتيب المجموعات الفرعية، لن تشعر أنك مضطر لتتبع مسار التنفيذ، صحيح؟ ستفترض أنها تعمل بشكل صحيح لأنك نقحتها من الأخطاء مسبقاً. حسناً، كل ما فعلته لجعل mergeSort تتعاود هو استبدال خوارزمية الترتيب بواحدة أخرى. لا يوجد أي سبب يدعوك لقراءة البرنامج بشكل مختلف.

في الواقع، عليك التفكير قليلاً بالحالة القاعدية والتأكد من أنك ستصل إليها في النهاية، لكن فيما عدا ذلك، يجب ألا تكون كتابة النسخة التعاودية مشكلة على الإطلاق. حظاً طيباً!

14.7 متغيرات الصنف

حتى الآن، شاهدنا المتغيرات المحلية التي يتم تصريح عنها داخل عملية، ومتغيرات الحالة، التي يتم التصريح عنها في تعريف الصنف، غالباً قبل تعاريف العمليات.

يتم إنشاء المتغيرات المحلية عند استدعاء العملية ويتم تدميرها عند انتهائها. يتم إنشاء متغيرات الحالة عندما تنشئ كائناً جديداً ويتم تدميرها عندما يتم جمع الكائن مع القمامة.

الآن حان وقت معرفة **متغيرات الصنف (class variables)**. مثل متغيرات الحالة، يتم تعريف متغيرات الصنف في تعريف الصنف قبل تعاريف العمليات، لكننا نعرفها باستخدام الكلمة المفتاحية static. يتم إنشاء هذه المتغيرات عند تشغيل البرنامج، وتظل على قيد الحياة حتى إنهاء البرنامج.

يمكنك الوصول إلى متغير الصنف من أي مكان داخل تعريف الصنف. يتم استخدام متغيرات الصنف غالباً لتخزين القيم الثابتة التي نحتاجها في عدة أماكن.

كمثال، هذه نسخة من Card حيث يكون suits وranks متغيري صنف:

```
class Card {
   int suit, rank;

   static String[] suits = { "Clubs", "Diamonds", "Hearts", "Spades" };
   static String[] ranks = { "narf", "Ace", "2", "3", "4", "5", "6",
   "7", "8", "9", "10", "Jack", "Queen", "King" };

   public static void printCard(Card c) {
   System.out.println(ranks[c.rank] + " of " + suits[c.suit]);
   }
}
```

يمكننا الوصول إلى suits وranks من داخل printCard كما لو كانا متغيرين محليين.

14.8 المصطلحات

الشفرة الزائفة: طريقة لتصميم البرامج بكتابة مسودة سريعة باستخدام مزيج من اللغة الإنكليزية ولغة Java.

العملية المساعدة: غالباً ما تكون عملية صغيرة لا تفعل أي عمل مفيد فعلاً لوحدها، لكنها تساعد عملية أخرى، أكثر فائدة.

pseudocode: A way of designing programs by writing rough drafts in a combination of English and Java.
helper method: Often a small method that does not do anything enormously useful by itself, but which helps another, more useful, method.

14.9 تمرينات

تمرين 14.1 إن الغرض من هذا التمرين هو تطبيق خوارزميات الخلط والترتيب الموجودة في هذا الفصل.

a. نزل شفرة هذا الفصل من http://thinkapjava.com/code/Card2.java. واستوردها إلى بيئة البرمجة عندك. لقد كتبت مخططات أولية للعمليات، لذلك يفترض أن تقدر على تجميع البرنامج. لكن عند تشغيله سيطبع رسائل تبين أن العمليات الفارغة لا تعمل بكل صحيح. عندما تملأ الفراغات في تلك العمليات بشكل صحيح، يجب أن تختفي تلك الرسائل.

b. إذا قمت بحل التمرين 12.4، فلا بد أنك كتبت العملية randomInt. إذا لم يكن ذلك ما حصل، اكتبها الآن وأضف بعض الشفرة لاختبارها.

c. اكتب عم لية باسم swapCards تأخذ مجموعة ورق (مصفوفة أوراق) ودليلين، وتبدل بين الورقتين الموجودتين في هذين الموقعين.

مساعدة: يجب أن تبدل المرجعيات وليس محتويات الكائنين. هذا أسرع؛ كما أنها تعالج القضية بشكل صحيح عندما تملك الأوراق أسماء مستعارة.

d. اكتب عملية باسم shuffleDeck تستخدم الخوارزمية من القسم 14.2. قد ترغب باستخدام عملية randomInt من التمرين 12.4.

e. اكتب عملية باسم indexLowestCard تستخدم العملية compareCard لإيجاد الورقة الأدنى في مجال معطى من مجموعة الورق (من lowIndex حتى highIndex، حيث ينتمي كل منها إلى المجال).

f. اكتب عملية باسم sortDeck ترتب مجموعة أوراق من الأدنى إلى الأعلى.

g. بالاستفادة من الشفرة الزائفة في القسم 14.6، اكتب العملية merge. تأكد من اختبارها قبل محاولة استخدامها كجزء من mergeSort.

h. اكتب النسخة المبسطة من mergeSort، تلك التي تقسم مجموعة الورق إلى نصفين، وتستخدم sortDeck لترتيب النصفين، وتستخدم merge لإنشاء مجموعة جديدة، مرتبة بالكامل.

i. اكتب نسخة تعاودية بالكامل من العملية mergeSort. تذكر أن sortDeck عملية تعديل وأن mergeSort تابع، ما يعني أن استدعاءهما يتم بشكل مختلف:

```
SortDeck(deck);           // modifies existing deck
deck = mergeSort(deck);   // replaces old deck with new
```

الفصل 15

البرمجة كائنية التوجّه

15.1 لغات البرمجة وأساليبها

يوجد العديد من لغات البرمجة وهي تقريباً بعدد أساليب البرمجة (أحياناً تدعى paradigms). كانت البرامج التي كتبناها حتى الآن **إجرائية (procedural)**، لأن تركيزنا كان منصباً على تحديد مجريات الحسابات.

معظم البرامج المكتوبة بلغة Java تكون **كائنية التوجه (Object-oriented)**، ما يعني أن التركيز يكون على الكائنات وتفاعلاتها (interactions). ها هي بعض خصائص البرمجة كائنية التوجه:

- غالباً ما تمثل الكائنات البرمجية (Objects) عناصر من العالم الحقيقي. في الفصل السابق، كان إنشاء الصنف Deck خطوة باتجاه البرمجة كائنية التوجّه.
- غالبية العمليات تكون عمليات كائنية (object methods) (مثل العمليات التي تستدعيها على السلاسل المحرفية) بدلاً من أن تكون عمليات أصناف (مثل عمليات Math). كانت العمليات التي كتبناها حتى الآن عمليات أصناف. في هذا الفصل سنكتب بعض عمليات الكائنات.
- تعزل الكائنات عن بعضها بتحديد الطرق التي تتفاعل بها الكائنات فيما بينها، خاصة بمنعها من الوصول إلى متغيرات الحالة بدون استدعاء عمليات وسيطة.
- تنظم الأصناف في أشجار عائلة (family trees) حيث تمتد الأصناف الجديدة من الأصناف الموجودة قبلها، مضيفةً عمليات جديدة ومستبدلةً عمليات أخرى.

في هذا الفصل سأترجم برنامج Card من الفصل السابق من الأسلوب الإجرائي إلى الأسلوب كائني التوجه. يمكنك تنزيل شفرة هذا الفصل من http://thinklikecs.webs.com/resources/code/CardSoln3.java.

15.2 عمليات الكائنات وعمليات الأصناف

يوجد نوعين من العمليات في Java، يدعيان **عمليات الأصناف (class methods)** و**عمليات الكائنات** أو **العمليات الكائنية (object methods)**. تميَّز عمليات الأصناف بالكلمة المفتاحية static في السطر الأول. أي عملية لا تملك الكلمة المفتاحية static تكون عملية كائنية.

بالرغم من أننا لم نكتب أية عمليات كائنية حتى الآن، إلا أننا قد استدعينا بعضاً منها. كلما تستدعي عملية "على" كائن، تكون هذه العملية عملية كائنية. مثلاً، charAt وغيرها من العمليات التي استدعيناها على كائنات String كانت كلها عمليات كائنية.

أي شيء يمكن كتابته كعملية صنف يمكن كتابته أيضاً كعملية كائن، والعكس صحيح. لكن في بعض الأحيان سيكون استعمال أحد النوعين طبيعياً أكثر من استعمال النوع الآخر.

مثلاً، هذه هي العملية printCard مكتوبة كعملية صنف:

```
public static void printCard(Card c) {
   System.out.println(ranks[c.rank] + " of " + suits[c.suit]);
}
```

وها هي كعملية كائنية:

```
public void print() {
   System.out.println(ranks[rank] + " of " + suits[suit]);
}
```

ها هي الاختلافات بين النسختين:

1. أزلت الكلمة static.
2. غيرت اسم العملية ليكون معبراً أكثر.
3. أزلت المعامل.
4. يمكنك الإشارة إلى متغيرات الحالة كما لو كانت متغيرات محلية داخل عمليات الكائنات، لذا غيرت c.rank إلى rank، كما قمت بنفس التغيير مع suit.

سيتم استدعاء هذه العملية بالشكل التالي:

```
Card card = new Card(1, 1);
card.print();
```

عندما تستدعي عملية على كائن، يصبح ذلك الكائن **الكائن الحالي (current object)**، ويعرف أيضاً بthis. داخل print، تشير this إلى كائن Card الذي تم استدعاء العملية عليه.

15.3 عملية toString

لكل نوع كائني عملية تدعى toString تعيد الكائن ممثلاً بسلسلة من المحارف (String). عندما تطبع كائناً باستخدام العملية print أو println، تستدعي Java العملية الكائنية toString.

النسخة الافتراضية من toString تعيد سلسلة محرفية تحتوي على نوع الكائن ومعرّف فريد (انظر القسم 11.6). عندما تعرف نوعاً كائنياً جديداً، يمكنك **تجاهل (override)** السلوك الافتراضي بكتابة عملية جديدة تنفذ السلوك الذي ترغب.

مثلاً، ها هي عملية toString للكائن Card:

```
public String toString() {
   return ranks[rank] + " of " + suits[suit];
}
```

من الطبيعي أن يكون نوع الإرجاع String، ولا تأخذ هذه العملية أية معاملات. يمكنك استدعاء toString بالطريقة المعتادة:

```
Card card = new Card(1, 1);
String s = card.toString();
```

أو يمكنك استدعاءها مباشرة من خلال println:

```
System.out.println(card);
```

15.4 عملية equals

في القسم 13.4 تحدثنا عن مفهومين للتكافؤ: التطابق، وهو يعني أن المتغيرين يشيران إلى نفس الكائن، والمساواة، التي تعني أن للمتغيرين القيمة نفسها.

يختبر عامل == التطابق، ولا يوجد عامل لاختبار المساواة، لأن تعريف "المساواة" يعتمد على نوع الكائن. بدلاً من ذلك، تملك الكائنات عملية تدعى equals تعرّف مساواة الكائنات.

توفر أصناف Java عمليات equals التي تنفذ عمليات المقارنة بالشكل الصحيح، لكن بالنسبة للأنواع التي يعرفها المستخدم يكون تعريف العملية الافتراضي نفس تعريف التطابق، وهو ما لا تريده عادة.

بالنسبة لكائنات Card فقد كتبنا مسبقاً عملية تتحقق من المساواة:

```
public static boolean sameCard(Card c1, Card c2) {
   return (c1.suit == c2.suit && c1.rank == c2.rank);
}
```

لذا كل ما علينا فعله هو إعادة كتابتها بشكل عملية كائنية:

```
public boolean equals(Card c2) {
   return (suit == c2.suit && rank == c2.rank);
}
```

لقد أزلت الكلمة static والمعامل الأول c1. هذه هي كيفية استدعاء هذه العملية:

```
Card card = new Card(1, 1);
Card card2 = new Card(1, 1);
System.out.println(card.equals(card2));
```

داخل equals، يكون card هو الكائن الحالي وcard2 يكون المعامل c2. بالنسبة للعمليات التي تعمل على كائنين من نفس النوع، أحياناً أستخدم الكلمة this صراحة وأدعو المعامل الثاني that:

```
public boolean equals(Card that) {
   return (this.suit == that.suit && this.rank == that.rank);
}
```

أعتقد أن هذا يجعل فهم العملية أسهل.

تمرين 15.1 نزل http://thinklikecs.webs.com/resources/code/CardSoln2.java و http://thinklikecs.webs.com/resources/code/CardSoln3.java.

يحتوي CardSoln2 على حلول تمارين الفصل السابق. وهو يستخدم عمليات الأصناف فقط (ما عدا العمليات البانية).

يحتوي CardSoln3 على نفس البرامج، لكن معظم العمليات أصبحت كائنية. لقد تركت merge كما هي لأنني أعتقد أنها أسهل للفهم كعملية صنف.

حول merge إلى عملية كائنية، وغير mergeSort وفقاً لذلك. أي نسخة من merge أفضل؟

15.5 الشذوذ والأخطاء

إذا كانت لديك عمليات كائنية وعمليات أصناف في نفس الصنف، فمن السهل أن تضيع. الطريقة الشائعة لتنظيم تعريف الصنف هي وضع كافة عمليات البناء في البداية، متبوعة بكل العمليات الكائنية ثم عمليات الصنف.

يمكن أن توجد عملية كائنية وعملية صنف بنفس الاسم، طالما أنهما تختلفان في عدد المعاملات أو نوعها. وكما يحصل مع الأنواع الأخرى من التحميل الزائد (overloading)، تقرر Java أي نسخة من العملية ستستدعيها بالنظر إلى المتحولات التي تعطيها.

الآن وقد بتنا نعرف معنى الكلمة المفتاحية static، ستكون قد استنتجت على الأغلب أن main هي عملية صنف، ما يعني عدم وجود "كائن حالي" عند استدعائها.

نظراً لعدم وجود كائن حالي في عمليات الأصناف، من الخطأ استعمال الكلمة المفتاحية this. إذا حاولت عمل ذلك، ستحصل على رسالة خطأ مثل: "Undefined variable: this" – "متغير غير معرّف: this".

كما لا يمكنك الوصول إلى متغيرات الحالة بدون استخدام النقطة وتوفير اسم كائن. إذا حاولت عمل ذلك، ستحصل على رسالة مثل "non-static variable… cannot be referenced from a static context". يقصد المجمع بكلمة "non-static variable" أن يقول: "متغير حالة" — "instance variable".

15.6 الوراثة

إن **الوراثة (inheritance)** هي أكثر مقومات اللغة المرتبطة بمفهوم البرمجة الكائنية على الأغلب. الوراثة هي القدرة على تعريف صنف جديد يكون نسخة معدلة من صنف موجود.

لتوسيع المصطلح، الصنف الموجود يدعى أحياناً الصنف **الأب (parent)** والصنف الجديد يكون **الابن (child)**.

الميزة الأساسية للوراثة هي القدرة على إضافة عمليات جديدة ومتغيرات حالة بدون تعديل الأب. هذه القدرة مفيدة بشكل خاص لأصناف Java الجاهزة، نظراً لأنك لا تستطيع تعديلها حتى لو أردت ذلك.

إذا قمت بحل تمارين GridWorld (الفصلين 5 و10) فقد شاهدت أمثلة عن الوراثة:

```
public class BoxBug extends Bug {
   private int steps;
   private int sideLength;

   public BoxBug(int length) {
      steps = 0;
      sideLength = length;
   }
}
```

BoxBug extends Bug تعني أن BoxBug هو نوع جديد من Bug يرث العمليات ومتغيرات الحالة الخاصة بـBug. يضاف إلى ذلك:

- يمكن أن يملك الصنف الابن متغيرات حالة إضافية؛ في هذا المثال، BoxBug يملك steps وsideLength.
- يمكن للصنف الابن أن يملك عمليات إضافية؛ في هذا المثال، BoxBug يملك عملية بناء إضافية تأخذ عدداً صحيحاً كمعامل.
- يمكن للابن أن **يتجاهل (override)** عملية من عمليات الصنف الأب؛ في هذا المثال، يوفر الابن عملية act (ليست مبينة هنا)، تغطي على العملية act من الصنف الأب.

وإذا قمت بحل تمارين الرسوميات في الملحق A، فقد رأيت مثالاً آخر:

```
public class MyCanvas extends Canvas {
   public void paint(Graphics g) {
      g.fillOval(100, 100, 200, 200);
   }
}
```

MyCanvas هو نوع جديد من Canvas ليس له أي متغيرات حالة أو عمليات جديدة، لكنه يتجاهل العملية paint.

إذا لم تكن قد حللت أياً من هذه التمارين، سيكون الآن وقتاً مناسباً!

15.7 سلسلة الأصناف الهرمية

في Java، كل الأصناف توسع أصنافاً أخرى. الصنف الأساسي الأول يدعى Object. لا يحتوي ذاك الصنف على أية متغيرات حالة، لكنه يوفر العمليتين toString وequals، من بين عمليات أخرى يوفرها أيضاً.

العديد من الأصناف توسع Object، بما فيها جميع الأصناف التي كتبناها تقريباً والعديد من أصناف Java، مثل java.awt.Rectangle. أي صنف لا يصرح عن اسم والده صراحة سيرث من Object افتراضياً.

بعض السلاسل الوراثية تكون أطول. مثلاً، java.swing.JFrame يوسّع java.awt.Frame، الذي يوسّع Window، الذي يوسع Container، الذي يوسّع Object. مهما كان طول السلسلة، سيكون Object الجد المشترك لجميع الأصناف.

"شجرة العائلة" الخاصة بالأصناف تدعى بسلسلة الأصناف الهرمية (class hierarchy). عادة ما يظهر Object في قمة السلسلة، وجميع "أبناءه" الأصناف في الأسفل. إذا اطلعت على وثائق JFrame مثلاً، سترى جزءاً من السلسلة الهرمية التي تشكل أصل JFrame.

15.8 التصميم كائني التوجه

الوراثة هي عنصر قوي. بعض البرامج التي كانت لتتعقد بدونها يمكن كتابتها باختصار وبساطة معها. أيضاً، يمكن للوراثة أن تسهل إعادة استخدام الشفرة، بما أنك تستطيع تغيير سلوك الأصناف الموجودة بدون الحاجة إلى تعديلها.

من جهة أخرى، قد تجعل الوراثة البرامج صعبة القراءة. عندما ترى استدعاءً لعملية، من الصعب معرفة أي عملية ستستدعى.

أيضاً، العدي من الأشياء التي يمكن تنفيذها بالاستعانة بالوراثة يمكن إجراؤها بنفس الجودة أو أفضل بدونها. من البدائل الشائعة **التركيب (composition)**، حيث تتركب الكائنات الجديدة من الكائنات الموجودة، مضيفة المزيد من القدرات بدون الوراثة.

تصميم الكائنات والعلاقات فيما بينها هو محور **التصميم كائني التوجّه (object-oriented design)**، وهو يتخطى مدى هذا الكتاب. لكن إذا كنت مهتماً، أنصحك بكتاب *Head First Design Patterns*، الذي نشرته O'Reilly Media.

15.9 المصطلحات

عملية كائنية: عملية يتم استدعاؤها على كائن، وتشتغل على ذلك الكائن، الذي يشار له بالكلمة المفتاحية this في Java أو "الكائن الحالي" في اللغة العربية. العمليات الكائنية لا تملك الكلمة المفتاحية static.

عملية صنف: عملية لها الكلمة المفتاحية static. لا تستدعى عمليات الأصناف على الكائنات وهي لا تملك كائناً حالياً.

الكائن الحالي: الكائن الذي تم استدعاء العملية الكائنية عليه. داخل العملية، يشار للكائن الحالي بالكلمة this.

this: الكلمة المفتاحية التي تشير إلى الكائن الحالي.

ضمني: شيء يترك بدون أن نقوله أو يكون غير صريح. داخل عملية كائنية، يمكنك الإشارة إلى متغيرات الحالة ضمنياً (بدون أن تسمي الكائن).

صريح: أي شيء يقال بشكل كامل. داخل عملية صنف، يجب أن تكون كافة المرجعيات إلى متغيرات الحالة صريحة.

object method: A method that is invoked on an object, and that operates on that object, which is referred to by the keyword this in Java or "the current object" in English. Object methods do not have the keyword static.

class method: A method with the keyword static. Class methods are not invoked on objects and they do not have a current object.

current object: The object on which an object method is invoked. Inside the method, the current object is referred to by this.

this: The keyword that refers to the current object.

implicit: Anything that is left unsaid or implied. Within an object method, you can refer to the instance variables implicitly (without naming the object).

explicit: Anything that is spelled out completely. Within a class method, all references to the instance variables have to be explicit.

15.10 تمرينات

تمرين 15.2 حول عملية الصنف التالية إلى عملية كائنية.

```
public static double abs(Complex c) {
   return Math.sqrt(c.real * c.real + c.imag * c.imag);
}
```

تمرين 15.3 حول العملية الكائنية التالية إلى عملية صنف.

```
public boolean equals(Complex b) {
   return(real == b.real && imag == b.imag);
}
```

تمرين 15.4 هذا التمرين تابع للتمرين 11.3. الغرض هو التدرب على بنية عمليات الكائنات والتعود على رسائل الخطأ المتعلقة بها.

a. حول العمليات في صنف Relational من عمليات صنف إلى عمليات كائنات، وقم بالتغييرات اللازمة في main.

b. اصنع بعض الأخطاء. جرب استدعاء عمليات صنف على أنها عمليات كائنية وبالعكس. حاول أن تعرف ما هو المشروع وما هو الممنوع، وحاول أن تفهم رسائل الخطأ التي تحصل عليها عندما تخلط الأمور.

c. فكر بمحاسن ومساوئ عمليات الأصناف وعمليات الكائنات. أيهما أكثر اختصاراً (عادة)؟ أيهما تبدو طبيعية أكثر للتعبير عن الحسابات (computations)؟ (أو ربما، أي نوع من الحسابات يمكن التعبير عنه بصورة طبيعية أكثر باستخدام كل من النمطين؟)

تمرين 15.5

The goal of this exercise is to write a program that generates random poker hands and classifies them, so that we can estimate the probability of the various poker hands. If you don't play poker, you can read about it here http://en.wikipedia.org/wiki/List_of_poker_hands.

a. Start with http://thinklikecs.webs.com/resources/code/CardSoln3.java. and make sure you can compile and run it.

b. Write a definition for a class named `PokerHand` *that extends* `Deck`.

c. Write a Deck method named `deal` *that creates a* `PokerHand`*, transfers cards from the deck to the hand, and returns the hand.*

d. In `main` *use* `shuffle` *and* `deal` *to generate and print four* `PokerHands` *with five cards each. Did you get anything good?*

e. Write a `PokerHand` *method called* `hasFlush` *returns a boolean indicating whether the hand contains a flush.*

f. Write a method called `hasThreeKind` *that indicates whether the hand contains Three of a Kind.*

g. Write a loop that generates a few thousand hands and checks whether they contain a flush or three of a kind. Estimate the probability of getting one of those hands. Compare your results to the probabilities at http://en.wikipedia.org/wiki/List_of_poker_hands .

h. Write methods that test for the other poker hands. Some are easier than others. You might find it useful to write some general-purpose helper methods that can be used for more than one test.

i. In some poker games, players get seven cards each, and they form a hand with the best five of the seven. Modify your program to generate seven-card hands and recompute the probabilities.

تُركت هذه الصفحة بيضاء عن عمد

الفصل 16

GridWorld: الجزء الثالث

إذا لم تقم بحل التمارين في الفصلين 5 و10، عليك حلهم قبل قراءة هذا الفصل. للتذكير فقط، يمكنك العثور على وثائق أصناف GridWorld على http://thinklikecs.webs.com/resources/javadoc/gridworld/.

يعرض الجزء الثالث من دليل الطالب لبرنامج GridWorld الأصناف المكونة للبرنامج والتفاعلات فيما بينها. هذه الأصناف هي مثال عن التصميم كائني التوجه وستمنحنا الفرصة للتحدث عن مشاكل التصميم كائني التوجه.

قبل قراءة دليل الطالب، توجد بعض الأشياء التي يجب أن تعرفها أولاً.

16.1 ArrayList

يستخدم البرنامج java.util.ArrayList، وهو كائن يشبه المصفوفات. بل هو **مجموعة (collection)**، ما يعني أنه كائن يحتوي على كائنات أخرى. توفر Java مجموعات أخرى بقدرات متفاوتة، لكن لاستخدام GridWorld سنحتاج إلى ArrayList فقط.

لترى مثالاً، نزل http://thinklikecs.webs.com/resources/code/BlueBug.java و http://thinklikecs.webs.com/resources/code/BlueBugRunner.java. الحشرة الزرقاء (BlueBug) هي حشرة تتحرك عشوائياً وتبحث عن الصخور. إذا حثرت على صخرة، ستلونها بالأزرق.

إليك كيفية عملها. عند استدعاء act، تعطي الحشرة الزرقاء موقعها ومرجعاً للشبكة:

```
Location loc = getLocation();
Grid<Actor> grid = getGrid();
```

النوع بين قوسي الزاوية (<>) هو **معامل نوع (type parameter)** يحدد محتويات الشبكة (grid). بكلمات أخرى، grid ليست مجرد كائن من نوع Grid، بل هي Grid تحتوي على Actors.

الخطوة التالية هي معرفة جيران الموقع الحالي. توفر Grid عملية تفعل ذلك:

```
ArrayList<Actor> neighbors = grid.getNeighbors(loc);
```

القيمة المعادة من getNeighbors هي ArrayList من Actors. تعيد عملية size طول ArrayList، وتختار get عنصراً. لذا يمكننا طباعة الجيران كما يلي:

```
for (int i=0; i<neighbors.size(); i++) {
   Actor actor = neighbors.get(i);
   System.out.println(actor);
}
```

إن المرور على ArrayList هو عملية شائعة لدرجة وجود تعليمة خاصة بها. لذا يمكننا أن نكتب:

```
for (Actor actor: neighbors) {
   System.out.println(actor);
```

```
}
```

نعلم أن الجيران هم Actors، لكننا لا نعرف نوعهم: قد يكونوا من النوع Bug، أو Rock، الخ. حتى نميز الصخور، سنستعمل عامل instanceof، الذي يتحقق من انتماء كائن ما إلى صنف معين.

```
for (Actor actor: neighbors) {
   if (actor instanceof Rock) {
      actor.setColor(Color.blue);
   }
}
```

حتى نتمكن من تشغيل كل هذا علينا استيراد الأصناف التي استخدمناها:

```
import info.gridworld.actor.Actor;
import info.gridworld.actor.Bug;
import info.gridworld.actor.Rock;
import info.gridworld.grid.Grid;
import info.gridworld.grid.Location;
import java.awt.Color;
import java.util.ArrayList;
```

تمرين 16.1 ابدأ مع نسخة من BlueBug.java، اكتب تعريف صنف لنوع جديد من الحشرات التي تبحث عن الأزهار وتأكلها. يمكنك "أكل" الأزهار باستدعاء removeSelfFromGrid عليها.

16.2 الواجهات

يستخدم GridWorld واجهات Java (Java interfaces)، لذلك أريد أن أشرح ما هي هذه الواجهات. تعني كلمة "interface" أشياء مختلفة بحسب موقعها من الكلام، لكنها في Java تشير إلى ميزة للغة البرمجة: الواجهة هي تعريف صنف لا تملك عملياته أجسام (bodies).

في تعريف الصنف العادي، يوجد لكل عملية نموذج أولي (prototype) وجسم (body) (انظر القسم 8.5). يدعى النموذج الأولي أيضاً **بالتوصيف (specification)** لأنه يصف اسم العملية، ومعاملاتها ونوع إرجاعها؛ يدعى الجسم **بالتطبيق (implementation)** لأنه يطبق (implement) التوصيف.

في واجهات Java لا تملك العمليات أجسام، لذا فهي تصف العمليات بدون أن تطبّقها.

مثلاً، java.awt.Shape هي واجهة فيها نماذج أولية للعمليات contains، وintersects، والعديد من العمليات الأخرى. يوفر الصنف java.awt.Rectangle تطبيقات لهذه العمليات، لذلك نقول "Rectangle implements Shape". هذا هو السطر الأول من تعريف الصنف Rectangle:

```
public class Rectangle extends Rectangle2D implements Shape, Serializable
```

يرث الصنف Rectangle العمليات من الصنف Rectangle2D ويوفر تطبيقات للعمليات الموجودة في Shape وSerializable. في برنامج GridWorld يطبق صنف Location واجهة java.lang.Comparable بتوفيره العملية compareTo، المشابهة لعملية compareCards في القسم 13.5.

يعرّف GridWorld أيضاً واجهة جديدة، اسمها Grid، التي توصّف العمليات التي يجب على الشبكات (grids) توفيرها. كما يحتوي البرنامج على تطبيقين لها، BoundedGrid وUnboundedGrid.

بقي أن نعقب على الاختصار API الذي يمثل الكلمات "Application programming Interface" – "واجهة برمجة التطبيقات". API هي مجموعة العمليات المتوفرة لمبرمجي التطبيقات حتى يستخدموها. انظر http://en.wikipedia.org/wiki/Application_programming_interface.

16.3 public وprivate

أتذكر عندما قلت في الفصل 1 أنني سأشرح سبب وجود كلمة public قبل العملية main؟ أخيراً، آن الأوان لذلك.

تعني كلمة public إمكانية استدعاء العملية من الأصناف الأخرى. الكلمة البديلة لهذه هي private، التي تعني أن استدعاء العملية ممكن فقط داخل الصنف الذي عرّفت فيه.

يمكن أن تكون متغيرات الحالة عامة (public) أو خاصة (private) أيضاً: متغير الحالة الخاص لا يمكن الوصول إليه إلا داخل الصنف الذي عرّف فيه.

السبب الرئيسي لجعل العمليات ومتغيرات الحالة خاصة هو الحد من التعاملات بين الأصناف المختلفة في سبيل التحكم بالتعقيد.

مثلاً، يبقي صنف Location متغيرات حالاته خاصة. لديه عمليات وصول (accessor methods) مثل getRow وgetCol، لكنه لا يوفر أية عمليات للتعديل على متغيرات حالاته (his instance variables). بالتالي، كائنات Location غير قابلة للتحوير، أي أننا نستطيع مشاركتها بدون أن نقلق بخصوص أشياء غير متوقعة تنتج عن تعدد الأسماء (aliasing).

إن تخصيص العمليات يساعد على تبسيط واجهة برمجة التطبيقات API. غالباً ما تتضمن الأصناف عمليات مساعِدة تُستخدَم لتنفيذ عمليات أخرى، لكن جعل هذه الأخيرة جزءاً من الAPI سيكون غير ضروري ومسبباً للأخطاء.

العمليات ومتغيرات الحالة الخاصة هما ميزتان لغويتان تساعد المبرمجين على ضمان **تغليف البيانات (data encapsulation)**، الذي يعني عزل الكائنات الموجودة في أحد الأصناف عن الأصناف الأخرى.

تمرين 16.2 أصبحت تعرف الآن ما يكفي لقراءة الجزء 3 من دليل الطالب لبرنامج GridWorld وحل التمرينات الموجودة فيه.

16.4 لعبة الحياة

اخترع الرياضي (عالم رياضيات) جون كُنواي "لعبة الحياة" – "Game of Life"، التي يصفها بأنها "لعبة بدون لاعبين" - "zero-player game" بسبب عدم الحاجة إلى لاعبين ليختاروا استراتيجيات أو يصنعوا قرارات. بعد تجهيز الشروط الابتدائية، تجلس وتشاهد اللعبة تلعب لوحدها. لكن تبين أن ذلك أكثر تشويقاً مما يبدو؛ يمكنك القراءة عنها على http://en.wikipedia.org/wiki/Conways_Game_of_Life.

إن الغرض من هذا التمرين هو تنفيذ لعبة الحياة باستخدام GridWorld. ستكون الشبكة رقعة اللعبة (game board)، وستمثل الصخور القطع.

تجرى اللعبة في أدوار، أو **خطوات زمنية (time steps)**. في بداية الخطوة الزمنية، كل صخرة إما أن تكون "على قيد الحياة" أو "ميتة". على الشاشة، سيمثِل لون الصخرة حالتها.

تعتمد حالة كل صخرة على حالة **جيرانها**. لكل صخرة 8 جيران، عدا الصخور الموجودة على حافة الشبكة. ها هي القواعد:

- إذا كان للصخرة الميتة ثلاثة جيران بالضبط، تعود إلى الحياة! وإلا ستظل ميتة.
- إذا كان للصخرة الحيّة جارين أو ثلاثة، فستنجو. وإلا ستموت.

بعض تبعات هذه القواعد: إذا كانت جميع الصخور ميتة، لا تعود أي منها إلى الحياة. إذا بدأت اللعبة مع صخرة حية وحيدة، تموت. لكن إذا وجدت 4 صخور في مربع، فستبقى كلها على قيد الحياة، لذلك نقول أن هذا الوضع مستقر.

معظم الأوضاع الابتدائية البسيطة إما أن تنقرض فيها الصخور سريعاً أو تصل إلى وضع مستقر. لكن توجد بعض الشروط الابتدائية التي تظهِر تعقيداً مميزاً. أحدها هو r-pentomino: يبدأ مع 5 صخور فقط، يعمل 1103 خطوة زمنية وينتهي في وضع مستقر مع 116 صخرة حية (انظر http://www.conwaylife.com/wiki/R-pentomino).

الأقسام التالية هي اقتراحاتي لتنفيذ لعبة الحياة (GoF) في GridWorld. يمكنك تنزيل الحل من
http://thinklikecs.webs.com/resources/code/LifeRunner.java و
.http://thinklikecs.webs.com/resources/code/LifeRock.java

16.5 LifeRunner

اصنع نسخة من BugRunner.java باسم LifeRunner.java وأضف العمليات ذات النماذج الأولية التالية:

```
/**
 * Makes a Game of Life grid with an r-pentomino.
 */
public static void makeLifeWorld(int rows, int cols)

/**
 * Fills the grid with LifeRocks.
 */
public static void makeRocks(ActorWorld world)
```

يجب أن تصنع makeLifeWorld شبكة عناصر (Actors) بالإضافة إلى ActorWorld، ثم تستدعي makeRocks، التي ستضع LifeRock في كل موقع من الشبكة.

16.6 LifeRock

اصنع نسخة من BoxBug.java باسم LifeRock.java. يجب أن يوسع LifeRock الصنف Rock. أضف عملية act لا تنفذ أي شيء. يجب أن تكون قادراً الآن على تشغيل البرنامج وأن ترى شبكة ممتلئة بالصخور.

لتسجيل حالة الصخرة، يمكنك إضافة متغير حالة جديد، أو يمكنك استخدام لون الصخرة ليدل على حالتها. في كلا الطريقتين، اكتب العمليات ذات النماذج الأولية التالية:

```
/**
 * Returns true if the Rock is alive.
 */
public boolean isAlive()

/**
 * Makes the Rock alive.
 */
public void setAlive()

/**
 * Makes the Rock dead.
 */
public void setDead()
```

اكتب عملية بناء تستدعي setDead وتأكد أن جميع الصخور ميتة.

16.7 التحديثات المتزامنة

في لعبة الحياة، يتم تحديث حالات جميع الصخور في وقت واحد؛ أي أن جميع الصخور ستتحقق من حالة جيرانها قبل أن تغير أي صخرة حالتها. وإلا فإن سلوك النظام سيعتمد على ترتيب التحديثات.

في سبيل إجراء التحديثات في وقت متزامن، أقترح أن تكتب عملية act بطورين: خلال الطور الأول، تحسب كافة الصخور جيرانها وتسجل النتائج؛ وفي الطور الثاني، تحدّث كافة الصخور حالتها.

هكذا بدت عملية act التي كتبتها أنا:

```
/**
 * Check what phase we're in and calls the appropriate method.
 * Moves to the next phase.
 */
   public void act() {
   if (phase == 1) {
      numNeighbors = countLiveNeighbors();
      phase = 2;
   } else {
      updateStatus();
      phase = 1;
   }
}
```

phase وnumNeighbors متغيرات حالة. وها هي النماذج الأولية للعمليات countLiveNeighbors وupdateStatus:

```
/**
 * Counts the number of live neighbors.
 */
public int countLiveNeighbors()

/**
 * Updates the status of the Rock (live or dead) based on the number
 * of neighbors.
 */
public void updateStatus()
```

ابدأ بكتابة نسخة بسيطة من updateStatus تغير الصخور الحية إلى ميتة وبالعكس. الآن شغل البرنامج وتأكد أن الصخور تغير ألوانها. كل خطوتين في World تكافئ خطوة زمنية واحدة في لعبة الحياة.

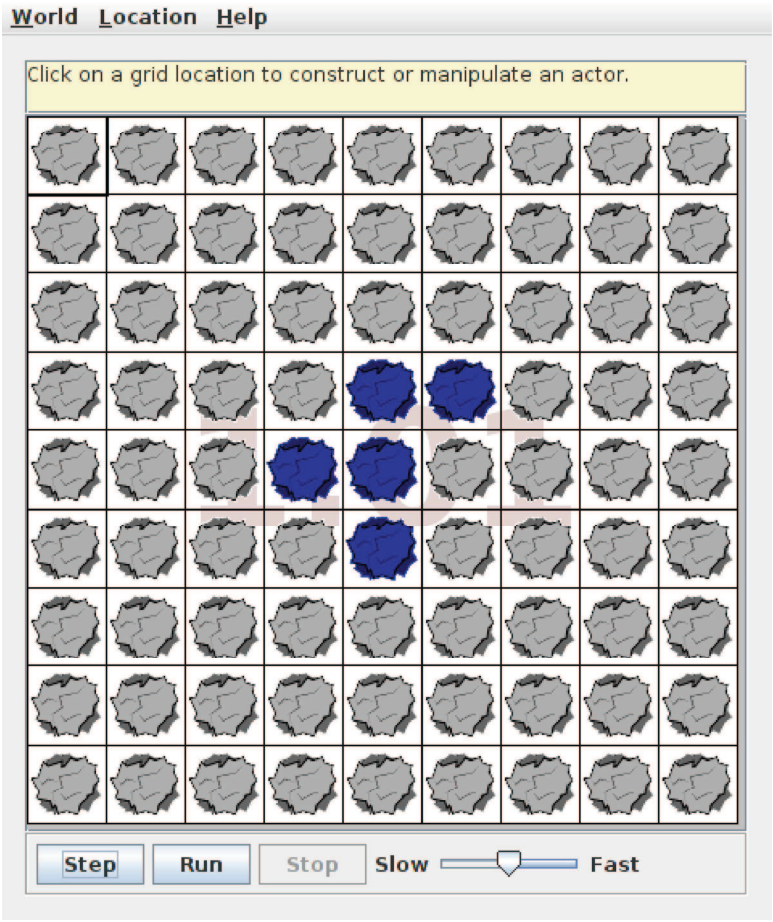
املأ الآن أجسام countLiveNeighbors وupdateStatus وفقاً للقواعد وانظر لترى إذا كان النظام سيعمل كما هو متوقع.

16.8 الشروط الابتدائية

لتغيير الشروط الابتدائية، يمكننا استخدام قائمة GridWorld المنبثقة لتغيير حالة الصخور بباستدعاء setAlive. أو يمكنك كتابة عمليات تؤتمت هذه العملية.

في LifeRunner، أضف عملية باسم makeRow تنشئ وضعاً أولياً فيه n صخرة على قيد الحياة في صف في منتصف الشبكة. ماذا يحدث عند استخدام قيم مختلفة للمتغير n؟

أضف عملية باسم makePentomino تصنع شكل r-pentomino في منتصف الشبكة. يجب أن يبدو الوضع الابتدائي كهذا:



إذا شغلت هذا البرنامج لعدد كبير من الخطوات، سيصل إلى نهاية الشبكة. حدود الشبكة تؤثر على سلوك النظام؛ حتى ترى التطور الكامل للبنتومينو، يجب أن تكون الشبكة كبيرة بما يكفي. قد تضطر للتجربة حتى تعرف القياس الصحيح، واعتماداً على سرعة حاسوبك، فقد تستغرق تلك العملية وقتاً.

موقع لعبة الحياة على الإنترنت يصف أوضاعاً ابتدائية أخرى تعطي نتائج مثيرة (http://www.conwaylife.com). اختر واحداً يعجبك ونفذه.

يوجد أيضاً أنواع أخرى للعبة الحياة مبنية على أساس قواعد مختلفة. جرب أحد تلك الأنواع وانظر لعلك تحصل على شيء مثير للاهتمام.

تمرين 16.3 إذا نفذت لعبة الحياة، فأنت جاهز للجزء 4 من دليل الطالب لبرنامج GridWorld. اقرأه وحل تمارينه.

تهانينا، لقد انتهيت!

تُركت هذه الصفحة بيضاء عن عمد

الملحق A

الرسوميات في Java

A.1 الرسوميات ثنائية الأبعاد

يقدم هذا الملحق أمثلة وتمارين تشرح الرسم في Java. توجد عدة طرق لإنشاء الرسوم في Java؛ أبسطها استخدام java.awt.Graphics. إليك مثالاً كاملاً:

```
import java.awt.Canvas;
import java.awt.Graphics;
import javax.swing.JFrame;

public class MyCanvas extends Canvas {

   public static void main(String[] args) {
      // make the frame
      JFrame frame = new JFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      // add the canvas
      Canvas canvas = new MyCanvas();
      canvas.setSize(400, 400);
      frame.getContentPane().add(canvas);

      // show the frame
      frame.pack();
      frame.setVisible(true);
   }

   public void paint(Graphics g) {
      // draw a circle
      g.fillOval(100, 100, 200, 200);
   }
}
```

يمكنك تنزيل هذه الشفرة من http://thinklikecs.webs.com/resources/code/MyCanvas.java.

تستورد الأسطر الأولى الأصناف التي نحتاجها من `java.awt` و`java.swing`.

`MyCanvas extends Canvas`، تعني أن كائن `MyCanvas` هو من نوع `Canvas` ويحتوي على عمليات لرسم كائنات رسومية.

في `main`:

1. أنشأنا `JFrame`، وهي نافذة تقدر على احتواء الصورة، والأزرار، والقوائم ومكونات النوافذ الأخرى؛
2. أنشأنا `MyCanvas`، وضبطنا العرض والارتفاع، وأضفناها إلى الإطار؛ و

3. عرضنا الإطار على الشاشة.

`paint` هي عملية خاصة يتم استدعاؤها عندما تلزم الحاجة لرسم `MyCanvas`. إذا شغلت هذه الشفرة، يجب أن ترى دائرة سوداء على أرضية رمادية.

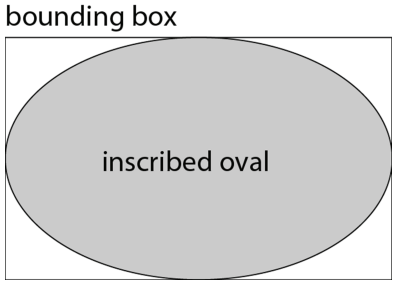
A.2 عمليات Graphics

حتى ترسم فوق حقل الصورة، عليك استدعاء عمليات على كائن الرسوميات. المثال السابق يستعمل fillOval. من العمليات الأخرى `drawLine`، و`drawRect` وغيرها. يمكنك قراءة وثائق هذه العمليات على http://download.oracle.com/javase/6/docs/api/java/awt/Graphics.html.

ها هو نموذج fillOval الأولي:

```
public void fillOval(int x, int y, int width, int height)
```

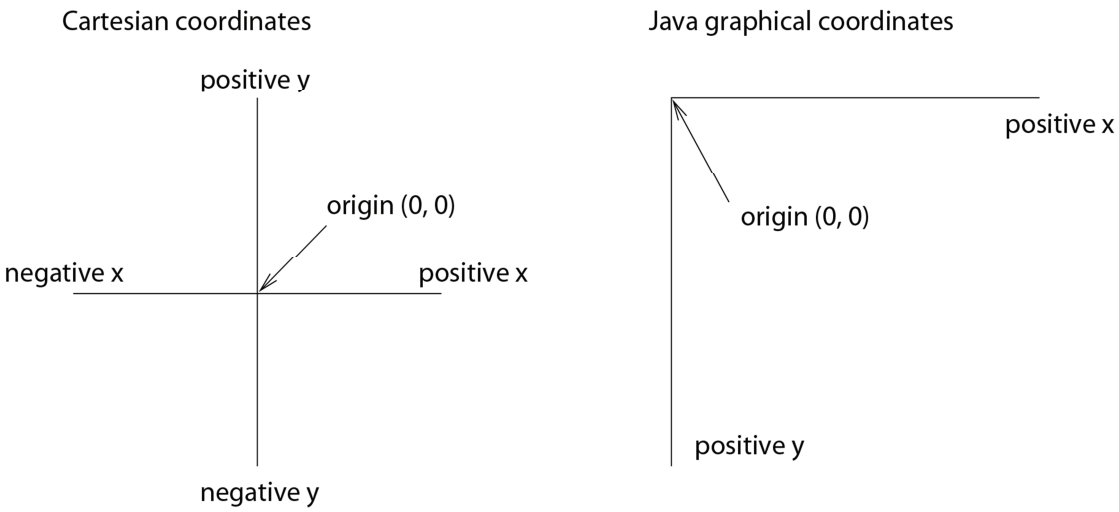
تحدد المعاملات **صندوقاً مؤطراً (bounding box)**، وهو مستطيل يتم رسم الشكل البيضوي داخله (مبين في الشكل). لن يتم رسم الصندوق المؤطر نفسه.



يحدد x وy موقع زاوية الصندوق المؤطر اليسرى العليا في نظام الإحداثيات الرسومية (Graphics coordinates system).

A.3 الإحداثيات

على الأغلب أنك تعرف الإحداثيات الديكارتية ثنائية البعد، حيث يعرف كل موقع بإحداثي x (البعد على طول المحور x عن المبدأ) وإحداثي y. اصطلاحاً، تتزايد الإحداثيات الديكارتية إلى اليمين والأعلى، كما هو مبين في الشكل.



وقد تم الاصطلاح على أن يستخدم نظام رسوميات الحواسيب نظام إحداثيات يكون فيه المبدأ في الزاوية اليسرى العليا، ويكون الاتجاه الموجب لمحور التراتيب (محور y) نحو الأسفل. تتبع Java هذا الاصطلاح.

واحدة قياس الإحداثيات هي البكسل (pixel)؛ كل بكسل يمثل نقطة على الشاشة. الشاشة النموذجية تكون بعرض 1000 بكسل تقريباً. تكون الإحداثيات أعداداً صحيحة دوماً. إذا أردت استعمال قيمة عشرية كإحداثي، فعليك تقريبها للتخلص من الفاصلة (انظر القسم 3.2).

A.4 الألوان

لاختيار لون شكل، استدعي العملية setColor على الكائن الرسومي:

```
g.setColor(Color.red);
```

تغير setColor اللون الحالي؛ كل شيء يتم رسمه يكون باللون الحالي.

Color.red هي قيمة يقدمها صنف Color؛ لاستخدامها عليك استيراد java.awt.Color. من الألوان الأخرى:

```
black       blue        cyan        darkGray    gray        lightGray
magenta     orange      pink        red         white       yellow
```

يمكنك إنشاء ألوان أخرى بتحديد قيم المزج للألوان الثلاثة الأساسية، الأحمر والأخضر والأزرق (RGB). انظر http://download.oracle.com/javase/6/docs/api/java/awt/Color.html.

يمكنك التحكم بلون أرضية حقل الصورة (canvas) باستدعاء Canvas.setBackground.

تمرين A.1 ارسم علم اليابان، دائرة حمراء على أرضية بيضاء عرضها أكبر من ارتفاعها.

A.5 ميكي ماوس

لنقل أننا نريد رسم صورة ميكي ماوس. يمكننا استعمال الشكل البيضوي الذي رسمناه منذ قليل كوجه، ثم نضيف الأذنان. لجعل قراءة الشفرة أسهل، دعنا نستخدم المستطيلات كصناديق تأطير.

ها هي العملية التي تأخذ مستطيلاً وتستدعي fillOval.

```
public void boxOval(Graphics g, Rectangle bb) {
   g.fillOval(bb.x, bb.y, bb.width, bb.height);
}
```

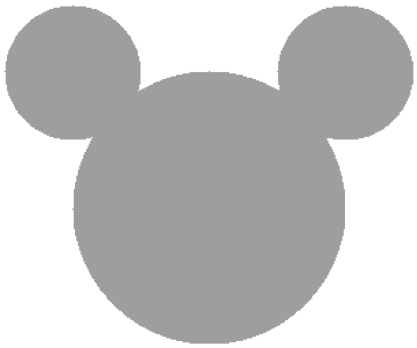
وها هي العملية التي ترسم ميكي:

```
public void mickey(Graphics g, Rectangle bb) {
   boxOval(g, bb);

   int dx = bb.width/2;
   int dy = bb.height/2;

   Rectangle half = new Rectangle(bb.x, bb.y, dx, dy);
   half.translate(-dx/2, -dx/2);
   boxOval(g, half);
```

```
    half.translate(dx*2, 0);
    boxOval(g, half);
}
```

يرسم السطر الأول الوجه. تنشئ الأسطر الثلاثة التالية مستطيلاً أصغر للأذنان. نقلنا المستطيل إلى الأعلى واليسار للأذن الأولى، ثم إلى اليمين للأذن الثانية.

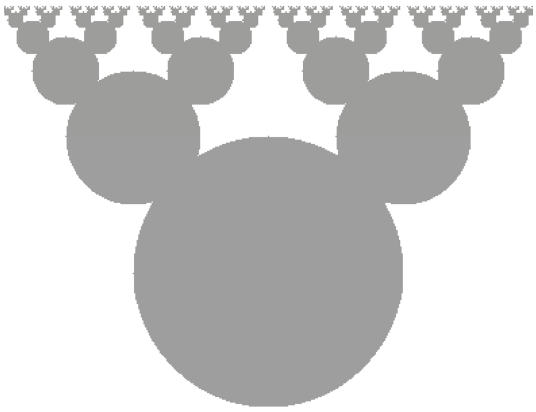
تبدو النتيجة كما يلي:



يمكنك تنزيل هذه الشفرة من http://thinklikecs.webs.com/resources/code/Mickey.java.

تمرين A.2 عدل Mickey.java لترسم أذنان على الأذنان، وأذنان على تلك الأذنان، ومزيد من الآذان حتى تصبح أصغر أذن بقياس 3 بكسل عرضاً فقط.

يجب أن تبدو النتيجة مثل ميكي موظ:



مساعدة: ستحتاج لإضافة أو لتعديل بضعة أسطر فقط من الشفرة.

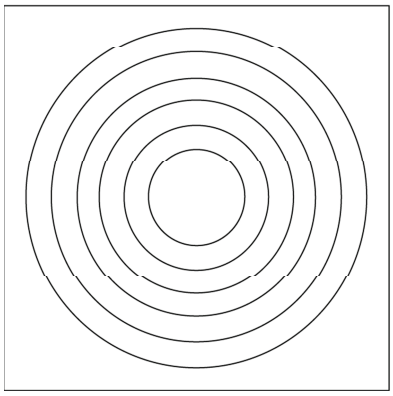
يمكنك تنزيل الحل من http://thinklikecs.webs.com/code/MickeySoln.java.

تمرين A.3

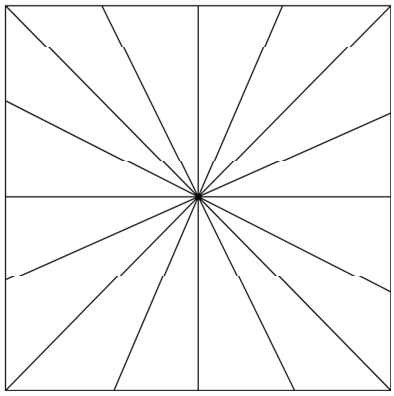
1. نزل http://thinklikecs.webs.com/code/Moire.java واستوردها إلى بيئة البرمجة عندك.

2. اقرأ عملية paint وارسم على ورقة ما تتوقع منها أن تفعل. الآن شغلها. هل حصلت على ما توقعت؟ لشرح ما يحصل، انظر http://en.wikipedia.org/wiki/Moire_pattern.
3. عدل البرنامج بحيث تكون المسافة بين الدوائر أكبر أو أصغر. انظر ماذا سيحدث للصورة.
4. عدل البرنامج بحيث ترسم الدوائر متمركزة في مركز الشاشة، كما في الشكل التالي (على اليسار). يجب أن تكون المسافة بين الدوائر صغيرة بما يكفي حتى يتضح التداخل المتموج.

concentric circles



radial Moire pattern



5. اكتب عملية باسم radial ترسم مجموعة من القطع المستقيمة شعاعياً كما هو مبين في الشكل (على اليمين)، لكن يجب أن تكون قريبة من بعضها بما يكفي لإنشاء نموذج متموج.
6. يمكن لأي نموذج رسومي تقريباً أن يولد نموذج تداخل مشابه لنموذج التموج. تلاعب بالبرنامج وانظر إلى ما يمكنك عمله.

الملحق B

الدخل والخرج في Java

B.1 كائنات System

يوفر صنف System عمليات وكائنات تحصل على المدخلات من لوحة المفاتيح، تطبع النص على الشاشة، وتقوم بالإدخال والإخراج من وإلى الملفات.

System.out هو الكائن الذي يطبع على الشاشة. عندما تستدعي العمليتين print وprintln، فأنت تستدعيهما على الكائن System.out.

يمكنك استخدام System.out لطباعة System.out:

```
System.out.println(System.out);
```

النتيجة هي:

```
java.io.PrintStream@80cc0e5
```

عندما تطبع Java كائناً، فهي تطبع نوع الكائن –PrintStream– والحزمة التي عرّف فيها النوع –java.io– ومعرف فريد للكائن. كان المعرف على جهازي 80cc0e5، لكن لو شغلت نفس الشفرة على جهاز آخر فستحصل على نتيجة مختلفة.

يوجد أيضاً كائن اسمه System.in يجعل الإدخال من لوحة المفاتيح ممكناً. لسوء الحظ، هذا الكائن لا يجعل الإدخال من لوحة المفاتيح سهلاً.

B.2 الإدخال من لوحة المفاتيح

أولاً، علينا استخدام System.in لإنشاء InputStreamReader جديد.

```
InputStreamReader in = new InputStreamReader(System.in);
```

بعد ذلك استخدم in لإنشاء BufferedReader جديد:

```
BufferedReader keyboard = new BufferedReader(in);
```

أخيراً، يمكنك استدعاء readLine على keyboard، حتى تحصل على الدخل من لوحة المفاتيح وتحوله إلى سلسلة محرفية.

```
String s = keyboard.readLine();
System.out.println(s);
```

هذه مشكلة واحدة فقط. هناك أشياء يمكن أن تسير بشكل خاطئ عند استدعائك readLine، وقد تسبب IOException. العملية التي تسبب استثناء عليها تضمينه في نموذجها الأولي (prototype)، مثل هذه:

```
public static void main(String[] args) throws IOException {
    // body of main
}
```

B.3 الإدخال من ملف

هذا برنامج يقرأ سطوراً من ملف ويطبعها:

```
import java.io.*;

public class Words {

   public static void main(String[] args)
            throws FileNotFoundException, IOException {

      processFile("words.txt");
   }

   public static void processFile(String filename)
            throws FileNotFoundException, IOException {

      FileReader fileReader = new FileReader(filename);
      BufferedReader in = new BufferedReader(fileReader);

      while (true) {
         String s = in.readLine();
         if (s == null) break;
         System.out.println(s);
      }
   }
}
```

يستورد السطر الأول java.io، الحزمة التي تحتوي FileReader، BufferedReader وبقية سلسلة الأصناف الهرمية المعقدة التي تستخدمها Java لعمل أشياء شائعة وبسيطة. تعني * استيراد كافة الأصناف الموجودة في الحزمة.

إليك نفس البرنامج مكتوباً بلغة Python:

```
for word in open('words.txt'):
    print word
```

أنا لا أمزح. هذا هو البرنامج بالكامل، وهو يقوم بنفس الوظيفة.

B.4 القبض على الاستثناءات

في المثال السابق، يمكن للعملية processFile أن تسبب FileNotFoundException وIOException. ونظراً لأن main تستدعي processFile، فعليها التصريح عن نفس الاستثناءات. في برنامج أكبر من هذا، قد تصرح main عن كل الاستثناءات الموجودة في اللغة.

إن البديل لهذا هو **القبض (catch)** على الاستثناء بتعليمة try. هذا مثال عن كيفية عمل ذلك:

```
public static void main(String[] args) {

    try {
        processFile("words.txt");
    } catch(Exception ex) {
        System.out.println("That didn't work. Here's why:");
        ex.printStackTrace();
    }
}
```

البنية مشابهة لبنية تعليمة if. إذا اشتغل "الفرع" الأول بدون التسبب باستثناء، يتم تجاوز الفرع الثاني.

إذا تسبب الفرع الأول باستثناء، يقفز مسار التنفيذ إلى الفرع الثاني، الذي يحاول معالجة حالة الاستثناء (طريقة مهذبة لنقول "الخطأ"). في هذه الحالة سيطبع البرنامج رسالة خطأ بالإضافة إلى دليل المكدس (stack trace).

يمكنك تنزيل هذه الشفرة من http://thinklikecs.webs.com/resources/code/Words.java وقائمة الكلمات من http://thinklikecs.webs.com/resources/code/words.txt.

الآن اذهب وحل التمارين 8.9، 8.10، و8.11.

تُركت هذه الصفحة بيضاء عن عمد

الملحق C

تطوير البرامج

C.1 الاستراتيجيات

لقد قدمت أساليب مختلفة لتطوير البرامج في الكتاب، لذا أردت جمعها معاً هنا.

أساس كل الاستراتيجيات هي **التطوير التصاعدي (incremental development)**، التي تتم كما يلي

1. ابدأ مع برنامج يعمل ينفذ شيء مرئي، مثل طباعة شيء ما.
2. أضف عدداً قليلاً من أسطر الشفرة كل مرة، واختبر البرنامج عند كل تغيير.
3. كرر الخطوات حتى يعمل البرنامج ما يفترض به أن يعمل.

بعد كل تغيير، يجب أن ينتج البرنامج بعض التأثيرات المرئية الناتجة عن اختبار الشفرة الجديدة. هذا الأسلوب في البرمجة يمكن أن يوفر الكثير من الوقت.

لأنك أضفت بضعة أسطر من الشفرة فقط في كل مرة، فسيكون العثور الأخطاء النحوية سهلاً.

وبما أن كل نسخة من البرنامج تولد نتيجة مرئية، فأنت تختبر بشكل مستمر نموذج كيفية عمل البرنامج في عقلك. إذا كان نموذجك العقلي خاطئاً، سيواجهك التعارض (وستملك الفرصة لتصحيحه) قبل كتابة الكثير من الشفرة الفاسدة.

صعوبة التطوير التصاعدي تكمن في صعوبة اكتشاف مسار مناسب من نقطة البداية وحتى الوصول إلى برنامج كامل وصحيح.

توجد عدة أساليب ممكنة لتساعدك بذلك:

التغليف والتعميم (Encapsulation and generalization): إذا لم تكن تعلم بعد كيفية تقسيم الحسابات إلى عمليات، ابدأ بكتابة الشفرة في main، ثم ابحث عن القطع المتماسكة لتغلفها في عملية، وتعميمها بشكل مناسب.

النمذجة السريعة (Rapid prototyping): إذا كنت تعلم ما هي العملية التي تريد كتابتها، لكنك لا تعرف كيفية كتابتها، ابدأ بمسودة سريعة تعالج أبسط حالة، ثم اختبرها مع الحالات الأخرى، وسّعها وصححها على طول الخط.

الأدنى أولاً (Bottom-up): ابدأ بكتابة العمليات البسيطة، بعد ذلك اجمعها في حل متكامل.

الأعلى للأسفل (Top-down): استعمل الشفرة الزائفة (pseudocode) لتصميم بنية الحسابات وتعريف العمليات التي ستحتاج. ثم اكتب العمليات واستبدل الشفرة الزائفة بالشفرة الحقيقية.

على طول الطريق، قد تحتاج بعض السقالات للدعم؛ مثلاً، يجب أن يكون لكل صنف عملية toString تسمح لك بطباعة حالة الكائن بصيغة يقدر البشر على قراءتها. هذه العملية تفيد في تنقيح البرنامج، لكنها في العادة ليست جزءاً من البرنامج النهائي.

C.2 أساليب الفشل

إذا كنت تقضي الكثير من الوقت في تصحيح الأخطاء، فذلك غالباً لأنك تستخدم استراتيجية تطوير غير فعالة. هذه هي أكثر الأساليب الفاشلة التي أراها غالباً (وأقع ضحية لها أحياناً):

التطوير غير التصاعدي (Non-incremental development): إذا كتبت أكثر من بضعة أسطر من الشفرة بدون ترجمتها واختبارها، فأنت تبحث عن المتاعب. سألت أحد الطلاب ذات مرة عن وظيفته، فأجاب، "تمام! لقد كتبتها بالكامل. بقي أن أصحح الأخطاء الموجودة فيها فقط."

الارتباط بالشفرة السيئة (Attachment to bad code): إذا كتبت أكثر من بضعة أسطر من الشفرة بدون أن تجمعها وتختبرها، فقد لا تتمكن من تنقيحها. أبداً. أحياناً يكون الحل الوحيد هو (التقط أنفاسك!) حذف الشفرة السيئة والبدء من جديد (باستخدام طريقة تصاعدية). لكن المبتدئين غالباً ما يرتبطون عاطفياً بالشفرة التي كتبوها، حتى لو لم تكن تعمل. الطريق الوحيدة للخروج من هذا الفخ هي أن تجعل قلبك قاسياً.

البرمجة عشوائية المجرى (Random-walk programming): أحياناً أعمل مع طلاب يبدو أنهم يبرمجون عشوائياً. يغيرون شيئاً ما، يشغلون البرنامج، يحصلون على خطأ، يغيرون شيئاً ما، يشغلون البرنامج، الخ. المشكلة هنا هي عدم وجود علاقة ظاهرة بين خرج البرنامج وبين التغييرات التي يجرونها.

إذا حصلت على رسالة خطأ، خذ الوقت لقراءتها. عموماً، عليك أخذ الوقت الكافي للتفكير.

الخضوع للمجمع (Compiler submission): رسائل الخطأ مفيدة، لكنها ليست صحيحة دائماً. مثلاً، إذا قالت الرسالة، "أتوقع فاصلة منقوطة في السطر 13 - Semi-colon expected on line 13"، فهذا يعني وجود خطأ نحوي بالقرب من السطر 13، لكن وضع فاصلة منقوطة على السطر 13 ليس الحل بالضرورة. لا ترضخ لرغبات المجمّع.

يقدم القسم التالي المزيد من المقترحات لتصحيح الأخطاء بأساليب فعالة.

تُركت هذه الصفحة بيضاء عن عمد

الملحق D

التنقيح

يعتمد الأسلوب الأفضل في التنقيح (debugging) على نوع الخطأ الذي تواجهه:

- تولّد الأخطاء النحوية (syntax errors) بوساطة المجمّع (compiler) وتشير إلى وجود خطأ في بنية البرنامج. مثال: نسيان كتابة الفاصلة المنقوطة في نهاية تعليمة.
- الاستثناءات (أو أخطاء التشغيل) تنتج في حال حصول خطأ أثناء تشغيل البرنامج. مثال: تعاود لا نهائي يسبب StackOverflowException في النهاية.
- تسبب الأخطاء المنطقية تنفيذ شيء خاطئ. مثال: قد لا يتم تنفيذ إحدى العبارات الحسابية بالترتيب الذي تتوقعه، ما يعطي نتائج غير متوقعة.

تم ترتيب الأقسام التالية وفق أنواع الأخطاء، توجد بعض التقنيات المفيدة لأكثر من نوع واحد.

D.1 الأخطاء النحوية

أفضل نوع من التنقيح هو الذي لا تحتاج إلى القيام به لأنك تتفادى عمل الأخطاء من البداية. في القسم السابق، اقترحت أساليب تطوير تقلل من الأخطاء وتجعل العثور عليها أسهل في حال وجودها.

الفكرة هي أن تبدأ مع برنامج يعمل وتضيف كميات صغيرة من الشفرة في كل مرة. عند وجود خطأ، ستكون عندك كرة جيدة عن مكان وقوعه.

ومع ذلك، فقد تجد نفسك في أحد المواقف التالية. في كل موقف، سأقدم بعض المقترحات عن كيفية التعامل معه.

المجمّع يمطر رسائل الأخطاء

إذا أعطى المجمع 100 رسالة خطأ، فهذا لا يعني وجود 100 خطأ في برنامجك. عندما يواجه المجمع خطأ، فسيخرج عن المسار لفترة. سيحاول استرجاع المسار والمتابعة بعد الخطأ الأول، لكنه أحياناً يعطي أخطاء زائفة.

يمكن الاعتماد على رسالة الخطأ الأولى فقط. أنا أقترح أن تصحح خطأ واحداً فقط في كل مرة، ثم أعد تجميع البرنامج. قد تجد أن فاصلة منقوطة واحدة "تصحح" 100 خطأ.

أتلقى رسالة غريبة من المجمع لا ترضى بأن تذهب عني

أولاً، اقرأ رسالة الخطأ بدقة. إنها مكتوبة بكلمات مكثفة، لكن غالباً ما توجد فيها معلومات مفيدة مخبأة بعناية.

إذا لم تفهم منها شيئاً، فستعرف على الأقل مكان حدوث المشكلة في البرنامج. في الواقع، ستخبرك الرسالة بالمكان الذي لاحظ فيه المجمع وجود مشكلة، وليس بالضرورة أن يكون المكان الذي يوجد فيه الخطأ. استخدم المعلومات التي يعطيك المجمع كدليل، لكن إذا لم تعثر على الخطأ في المكان الذي يشير إليه المجمع، وسِّع نطاق البحث.

بشكل عام، سيكون الخطأ في مكان سابق للموقع المشار إليه في رسالة الخطأ، لكن توجد حالات يكون الخطأ فيها بمكان آخر مختلف تماماً. مثلاً، إذا حصلت على خطأ عند استدعاء عملية، قد يكون الخطأ الحقيقي في تعريف العملية.

إذا لم تعثر على الخطأ بسرعة، خذ نفساً وانظر على البرنامج ككل. تأكد من أن البرنامج مرتب (indented) بشكل مناسب؛ ما يجعل اكتشاف الأخطاء النحوية أسهل.

الآن ابدأ بالبحث عن الأخطاء الشائعة:

1. تحقق من تناظر جميع الأقواس وأنها متداخلة بشكل صحيح. يجب أن تكون تعاريف العمليات كلها داخل تعريف صنف. كافة تعليمات البرنامج يجب أن تكون داخل تعريف عملية.
2. تذكر أن الأحرف الكبيرة لا تكافئ الأحرف الصغيرة.
3. تحقق من وجود الواصل المنقوطة عند نهاية كل تعليمة (ومن عدم وجودها بعد الأقواس المنحنية).
4. تأكد أن جميع السلاسل المحرفية في الشفرة لها علامتي اقتباس عند البداية والنهاية. تأكد من استعمال علامات الاقتباس المزدوجة للسلاسل المحرفية وعلامات الاقتباس المفردة للمحارف.
5. بالنسبة لتعليمات الإسناد، تأكد أن نوع الطرف الأيسر من نفس نوع الطرف الأيمن. تأكد أن الطرف الأيسر هو اسم متغير أو شيء آخر يمكنك إسناد قيمة له (مثل عنصر من مصفوفة).
6. بالنسبة لكل استدعاء لعملية، تأكد أن المتحولات التي تعطيها لها مكتوبة بالترتيب الصحيح، وأنها من نوع مناسب، وأن يكون الكائن الذي تستدعي العملية عليه من النوع المناسب.
7. إذا كنت تستدعي عملية مثمرة (عملية تعيد قيمة)، تأكد من عمل شيء ما بالنتيجة. إذا كنت تستدعي عملية عقيمة (void method)، تأكد من أنك لا تحاول استخدام نتيجتها لعمل شيء ما.
8. إذا كنت تستدعي عملية كائنية (object method)، تأكد من أنك تستدعيها على كائن من النوع الملائم. إذا كنت تستدعي عملية صنف من مكان خارج الصنف الذي عرِّفت فيه، تأكد من ذكر اسم الصنف.
9. يمكنك الولوج إلى متغيرات الحالة بدون تحديد الكائن داخل عمليات الكائنات. إذا حاولت عمل ذلك في عملية صنف، ستحصل على رسالة مثل، "Static reference to non-static variable".

إذا لم ينفع أي من هذه الأشياء، انتقل إلى القسم التالي...

لا أستطيع تجميع برنامجي مهما فعلت

إذا قال المجمع أنه يوجد خطأ في البرنامج ولم تره، فقد يعود ذلك لأنك تنظر إلى شفرة غير التي ينظر إليها المجمع. تحقق من بيئة برمجتك لتتأكد أن البرنامج الذي تعدله هو نفس البرنامج الذي يترجمه المجمع. إذا لم تكن متأكداً، جرب وضع أخطاء نحوية واضحة وصريحة في بداية البرنامج. الآن جمّعه ثانية. إذا لم يعثر المجمع على الخطأ الجديد، فذلك غالباً ما يكون بسبب خطأ بأسلوب إعدادك لبيئة البرمجة.

إذا فحصت الشفرة بصورة شاملة، وكنت متأكداً أن المجمع يترجم الشفرة الصحيحة، فقد آن الأوان لاتخاذ تدابير اليأس: **التنقيح المجزأ (debugging by bisection)**.

- انسخ الملف الذي تعمل عليه. إذا كنت تعمل على Bob.java، اصنع نسخة منه باسم Bob.java.old.

- احذف ما يعادل نصف الشفرة تقريباً من Bob.java. جرب تجميعه ثانيةً.
  - إذا تمكنت من تجميع البرنامج الآن، ستعرف أن الخطأ في النصف الثاني. أرجع ذاك النصف الذي حذفته وأعد الكرة.
  - إذا لم تتمكن من تجميع البرنامج، فلا بد أن الخطأ في هذا النصف. احذف نصف هذا النصف وأعد الكرة.
- بمجرد العثور على الخطأ وتصحيحه، ابدأ بإعادة الشفرة التي حذفتها، قطعة صغيرة في كل مرة.

هذه العملية بشعة، لكنها تتم أسرع مما تتصور، وهي موثوقة تماماً.

لقد قمت بما طلبه المجمع مني، ولا يزال البرنامج لا يعمل

تحتوي بعض رسائل المجمع على نصائح تبشر بالخير، مثل

"class Golfer must be declared abstract. It does not define int compareTo(java.lang.Object) from interface java.lang.Comparable"

يبدو أن المجمع يخبرك بوجوب التصريح عن الصنف Golfer كصنف مجرد، وإذا كنت تقرأ هذا الكتاب، فعلى الأغلب أنك لا تعرف ما هذا أو كيفية عمله.

لحسن الحظ، المجمع على خطأ. الحل في هذه الحالة هو التأكد أن الصنف Golfer يملك عملية باسم compareTo تأخذ Object كمعامل.

لا تدع المجمّع يسحبك من أنفك. رسائل الخطأ التي يعطيك إياها دليل على حدوث خطأ، لكن لا يمكنك الاعتماد على الحلول التي يقترحها عليك.

D.2 أخطاء التشغيل

برنامجي يعلق

إذا توقف البرنامج وبدا انه لا يفعل أي شيء، نقول أنه يعلق (أو يجمد hang). ذلك يكون معناه غالباً أن البرنامج دخل في حلقة لا نهائية أو عملية عودية لا نهائية.

- في حال وجود حلقة معينة تشك أنها سبب المشكلة، أضف تعليمة طباعة قبل الحلقة مباشرة تقول "entering the loop". وواحدة أخرى بعد الحلقة مباشرة تقول "exiting the loop".

شغل البرنامج. إذا حصلت على الجملة الأولى دون الثانية، فمشكلتك هي حلقة لا نهائية. اذهب إلى السم بعنوان "الحلقات اللانهائية".

- في معظم الأوقات ستسبب التعاود اللانهائي بتشغيل البرنامج لفترة من الزمن ثم يولد StackOverflowException. إذا حدث ذلك، اذهب إلى القسم بعنوان "التعاود اللانهائي".

إذا لم تكن تحصل على StackOverflowException، لكنك تشك بوجود مشكلة في عملية عودية، فلا يزال استعمال الأساليب الموجودة في قسم التعاود اللانهائي ممكناً.

- إذا لم تنفع أياً من هذه المقترحات، فقد لا تكون مستوعباً لمجرى التنفيذ في برنامجك. اذهب إلى القسم بعنوان "مجرى التنفيذ".

الحلقات اللانهائية

إذا كنت تعتقد أنك تملك حلقة لا نهائية وكنت تعرف أية حلقة هي، أضف تعليمة طباعة في نهاية الحلقة تطبع قيم المتغيرات الموجودة في الجملة الشرطية، وقيمة الشرط.

مثلاً،

```
while (x > 0 && y < 0) {
   // do something to x
   // do something to y

   System.out.println("x: " + x);
   System.out.println("y: " + y);
   System.out.println("condition: " + (x > 0 && y < 0));
}
```

الآن عندما تشغل البرنامج سترى ثلاثة أسطر من الخرج في كل مرة تدور فيها الحلقة. في آخر مرة يتم تنفيذ الحلقة فيها، يجب أن تكون قيمة الشرط false. إذا استمرت الحلقة بالعمل، انظر إلى قيم x وy فقد تعرف سبب عدم تحديثهما بشكل صحيح.

التعاود اللانهائي

في أغلب الأحيان يسبب التعاود اللانهائي توليد استثناء StackOverflowException. لكن إذا كان البرنامج بطيئاً فقد يستغرق وقتاً طويلاً قبل أن يملأ المكدس.

إذا كنت تعرف أية عملية هي سبب المشكلة، تحقق من وجود حالة قاعدية فيها. يجب وجود شرط يجعل العملية تنتهي بدون عمل استدعاء تعاودي آخر. إذا لم توجد حالة قاعدية للعملية، عليك إعادة التفكير بالخوارزمية وتعريف حالة قاعدية لها.

في حال وجود حالة قاعدية، لكن لا يبدو أن البرنامج يصل إليها، أضف تعليمة طباعة إلى بداية العملية تطبع معاملاتها. يجب أن ترى الآن بضعة أسطر من الخرج عند كل استدعاء للعملية، ترى فيه قيم معاملاتها. إذا لم تتحرك المعاملات نحو الحالة القاعدية، فقد تتمكن من رؤية السبب.

مجرى التنفيذ

إذا لم تكن متأكداً من حركة مجرى التنفيذ في البرنامج، أضف تعليمات طباعة إلى بداية كل عملية تطبع رسالة مثل "entering method foo"، حيث foo اسم العملية.

عندما تشغل البرنامج الآن فسوف يترك أثراً لكل عملية يتم استدعائها.

يمكن طباعة المتحولات التي تستقبلها كل عملية أيضاً. عندما تشغل البرنامج، تأكد أن القيم منطقية، وتحقق من عدم وقوعك في أكثر الأخطاء شيوعاً – توفير المتحولات بترتيب خاطئ.

عندما أشغل البرنامج يولد استثناء

عند حدوث استثناء، تطبع Java رسالة تتضمن اسم الاستثناء، السطر الذي حدثت فيه المشكلة، وسجل المكدس.

يحتوي سجل المكدس (stack trace) على العملية التي كانت تعمل أثناء حدوث الخطأ، العملية التي استدعتها، والعملية التي استدعت العملية التي سبقتها، وهكذا.

الخطوة الأولى هي فحص المكان الذي حدث فيه الخطأ ومحاولة اكتشاف ما الذي جرى.

NullPointerException: لقد حاولت الولوج إلى متغير حالة أو استدعاء عملية على كائن معدوم (null object). عليك معرفة أي متغير هو المعدوم ثم معرفة السبب وراء وصوله إلى تلك الحالة.

تذكر أن التصريح عن متغيرات من نوع كائني (object type)، يكون معدوماً في الحالة الابتدائية، ويظل هكذا حتى تسند له قيمة. مثلاً، هذه الشفرة تسبب NullPointerException:

ArrayIndexOutOfBoundsException: الدليل الذي تستخدمه للوصول إلى عناصر مصفوفة إما أن يكون سالباً أو أكبر من array.length-1. إذا عثرت على الموقع الذي حدثت تلك المشكلة فيه، أضف تعليمة طباعة قبله تماماً تطبع قيمة الدليل (index) وطول المصفوفة. هل حجم المصفوفة صحيح؟ هل قيمة الدليل صحيحة؟

الآن أعمِل طريقك رجوعاً في البرنامج وابحث عن المكان الذي جاءت منه المصفوفة وجاء منه الدليل. اعثر على أقرب تعليمة إسناد وتحقق من أنها تعمل بشكل الصحيح.

إذا كان أحدهما معاملاً، اذهب إلى مكان استدعاء المصفوفة وانظر إلى المكان الذي أتت منه القيم.

StackOverflowException: انظر "التعاود اللانهائي".

FileNotFoundException: هذا يعني أن Java لم تعثر على الملف الذي كانت تبحث عنه. إذا كنت تستخدم بيئة برمجة تعتمد المشاريع كأساس للعمل، مثل Eclipse، فقد تحتاج إلى استيراد الملف إلى المشروع. وإلا عليك التأكد أن الملف موجود فعلاً وأن مساره صحيح. هذه المشكلة تعتمد على نظام الملفات لديك.

ArithmeticException: يحدث عند حدوث خطأ أثناء عملية رياضية، أغلب الأحيان بسبب القسمة على صفر.

لقد أضفت تعليمات طباعة كثيرة لدرجة أنني غرقت بمخرجات البرنامج

إحدى المشاكل التي تحدث عند استعمال تعليمات الطباعة لتنقيح البرامج هي احتمال أن تنتهي مدفوناً بالمخرجات. يوجد طريقتين للمتابعة: إما أن تبسط الخرج، أو أن تبسط البرنامج.

حتى تبسط المخرجات يمكنك حذف تعليمات الطباعة الغير مفيدة أو تحويلها إلى تعليقات، أو جمعها معاً، أو تنسيق المخرجات حتى تصبح أسهل للفهم. أثناء تطوير أي برنامج، عليك كتابة شفرة لتوليد معلومات بصرية مختصرة عما يفعله البرنامج.

لتبسيط البرنامج، صغر حجم المشكلة التي يعالجها البرنامج. مثلاً، إذا كنت تعمل على ترتيب مصفوفة، رتب مصفوفة صغيرة. إذا كان البرنامج يستقبل مدخلات من المستخدم، أعطه أبسط مقدار من المعطيات التي تسبب الخطأ.

أيضاً، قم بتنظيف الشفرة. أزل الشفرة الميتة وأعد ترتيب البرنامج حتى يصبح أسهل للقراءة. مثلاً، إذا شككت بأن الخطأ يحدث في مكان عميق التداخل من البرنامج، أعد كتابة ذاك الجزء باستخدام بنية أبسط. إذا شككت بأمر عملية ضخمة، قسّمها إلى عمليات أصغر واختبر كل واحدة لوحدها.

عملية البحث عن أبسط حالة اختبار غالباً ما تقودك إلى الخطأ. مثلاً، إذا وجدت أن البرنامج يعمل عندما يكون عدد عناصر المصفوفة زوجياً، ولا يعمل عندما يكون عدد عناصرها فردياً، فذلك دليل يفيدك في معرفة ما يجري.

يمكن أن تساعدك إعادة ترتيب البرنامج للعثور على الأخطاء الخبيثة. مثلاً، لو غيرت شيئاً في البرنامج وكنت تعتقد أن هذا التغيير لن يؤثر على سلوك البرنامج، ثم أثر عليه، فذلك قد يدلك على المشكلة.

D.3 الأخطاء المنطقية

برنامجي لا يعمل

يصعب العثور على الأخطاء المنطقية أكثر لأن المجمع ونظام التشغيل (run-time system) لا يوفران أي معلومات عن الخطأ. أنت فقط تعرف ما يفترض بالبرنامج أن يفعله، وتعرف أنه لا يفعل ذلك.

الخطوة الأولى هي إيجاد صلة بين الشفرة وبين السلوك الذي تراه. تحتاج إلى فرضية عما يحدث في البرنامج فعلاً.

إليك بعض الأسئلة لتطرحها على نفسك:

- أيوجد شيء يفترض من البرنامج عمله، لكن لا يبدو أنه يحدث؟ ابحث عن الجزء من الشفرة الذي ينفذ تلك الوظيفة وتأكد أنه يشتغل بصورة مناسبة. انظر "مجرى التنفيذ" ، أعلاه.
- هل يحدث شيء لا يفترض أن يحدث؟ ابحث عن أية شفرة في البرنامج تجري وظيفة ما في الوقت الذي لا يجب حدوث ذلك فيه.
- هل يولد أحد أجزاء الشفرة أثراً غير المتوقع؟ تأكد من فهمك للشفرة، خصوصاً إذا كانت تستدعي عمليات جاهزة في Java. اقرأ وثائق هذه العمليات، وجربها باستخدام حالات اختبار بسيطة. قد لا تنفذ هذه العمليات ما تعتقد أنها تنفذه.

حتى تبرمج، تحتاج إلى نموذج عقلي (mental model) لما تجريه الشفرة التي تكتبها. إذا لم تعمل الشفرة كما هو متوقع، فقد لا تكون المشكلة في البرنامج؛ يمكن أن تكون المشكلة عندك.

أفضل طريقة لتقويم نموذجك العقلي تكون بتقسيم البرنامج إلى مكونات (عادة الأصناف والعمليات) واختبارها على انفراد.

إليك أكثر الأخطاء المنطقية شيوعاً لتتحقق منها:

- تذكر أن القسمة الصحيحة تقرب النتائج إلى الأدنى دائماً. إذا أردت الحصول على كسور، استخدم الأعداد العشرية doubles.
- الأعداد العشرية تقريبية دائماً، لذا لا تعتمد عليها للحصول على الدقة الكاملة.
- بصورة أكثر شمولية، استخدم الأعداد الصحيحة للأشياء المعدودة والأعداد العشرية للأشياء القبلة للقياس.
- إذا استخدمت عامل الإسناد، =، بدلاً من عامل المقارنة، ==، في إحدى الجمل الشرطية لتعليمات if، while، أو for، فقد تحصل على عبارة صحيحة لغوياً لكنها تعطي دلالة خاطئة.
- عندما تطبق عامل المقارنة، ==، على كائن، فسوف يتحقق من المطابقة. إذا قصدت التحقق من المساواة، عليك استخدام عملية equals.
- بالنسبة للأنواع المعرّفة بوساطة المستخدم، تتحقق equals من المطابقة. إذا أردت استخدام مفهوم آخر للتكافؤ، عليك كتابة عملية أخرى بنفس الاسم تهيمن على القديمة (override it).
- يمكن أن تؤدي الوراثة إلى أخطاء منطقية خبيثة، بسبب إمكانية تشغيل شفرة موروثة دون الانتباه لذلك. انظر "مجرى التنفيذ"، أعلاه.

لدي عبارة كبيرة غليظة ولا تنفذ ما أريده

كتابة العبارات المعقدة أمر محمود طالما أنها مقروءة، لكنها قد تكون صعبة التنقيح. من الجيد عادة تقسيم العبارات المعقدة إلى سلسلة من الجمل بالاستعانة بمتغيرات مؤقتة.

مثلاً:

```
rect.setLocation(rect.getLocation().translate(
-rect.getWidth(), -rect.getHeight()));
```

يمكن إعادة كتابتها كما يلي

```
int dx = -rect.getWidth();
int dy = -rect.getHeight();
Point location = rect.getLocation();
Point newLocation = location.translate(dx, dy);
rect.setLocation(newLocation);
```

النسخة المفصلة أسهل للقراءة، لأن أسماء المتغيرات توفر معلومات إضافية، وأسهل عند تصحيح الأخطاء، بسبب القدرة على التحقق من أنواع المتغيرات المؤقتة وطباعة قيمها.

من مشاكل العبارات الكبيرة الأخرى هي أن ترتيب الحساب قد لا يكون موافقاً لما تتوقعه. مثلاً، لحساب $\frac{x}{2\pi}$، قد تكتب

```
double y = x / 2 * Math.PI;
```

هذا ليس صحيحاً، لأن أولوية الضرب والقسمة متساوية، وسيتم تنفيذ العمليتين بحسب ترتيب ورودها من اليسار إلى اليمين. هذه العبارة تحسب $\frac{x\pi}{2}$.

إذا لم تكن واثقاً من ترتيب العمليات، استعمل الأقواس لتوضيحه أكثر.

```
double y = x / (2 * Math.PI);
```

هذه النسخة صحيحة، وأسهل للقراءة لمن لم يحفظ ترتيب العمليات.

عمليتي لا تعيد النتيجة المطلوبة

إذا كانت تعليمة العودة ذات عبارة معقدة، فلن تجد فرصة لطباعة قيمة ما للتحقق منها قبل إنهاء العملية. يمكنك استخدام المتغيرات المؤقتة هنا أيضاً. مثلاً، بدلاً من

```
public Rectangle intersection(Rectangle a, Rectangle b) {
return new Rectangle(
Math.min(a.x, b.x),
Math.min(a.y, b.y),
Math.max(a.x+a.width, b.x+b.width)-Math.min(a.x, b.x)
Math.max(a.y+a.height, b.y+b.height)-Math.min(a.y, b.y) );
}
```

يمكنك كتابة

```
public Rectangle intersection(Rectangle a, Rectangle b) {
int x1 = Math.min(a.x, b.x);
int y2 = Math.min(a.y, b.y);
int x2 = Math.max(a.x+a.width, b.x+b.width);
```

```
int y2 = Math.max(a.y+a.height, b.y+b.height);
Rectangle rect = new Rectangle(x1, y1, x2-x1, y2-y1);
return rect;
}
```

الآن لديك فرصة لعرض أي من المتغيرات الوسيطة قبل الخروج من العملية. وبإعادة استخدام x1 وy1، أصبحت الشفرة أصغر أيضاً.

تعليمة الطباعة لا تفعل شيئاً

إذا استخدمت عملية println، سيتم عرض المخرجات فوراً، لكن إذا استخدمت print (على الأقل في بعض بيئات البرمجة) تخزن المخرجات بدون عرضها حتى الوصول إلى السطر الجديد التالي. إذا انتهى البرنامج بدون طباعة سطر جديد، فقد لا ترى الخرج المخزن أبداً.

إذا كنت تشك بأن هذا ما يحصل، غير بعض تعليمات print أو كلها إلى println.

أنا عالق فعلاً، وأحتاج إلى المساعدة حقاً

أولاً ابتعد عن الحاسوب لعدة دقائق. تبث الحواسيب موجات تؤثر على الدماغ، وتسبب الأعراض التالية:

- إحباط وقهر.
- اعتقادات وهمية ("الحاسب يحقد علي") وأفكار سحرية ("يعمل البرنامج عندما ألبس قبعتي بالمقلوب فقط").
- الاقتناع بأن العنب حامض ("هذا البرنامج سيء على أية حال").

إذا عانيت أحد هذه الأعراض، انهض وتمشى. عندما تهدأ، فكر بالبرنامج. ماذا يفعل؟ ما هي الأسباب المحتملة لهذا السلوك؟ متى كانت آخر مرة اشتغل فيها البرنامج، وما الذي فعلته بعدها؟

أحياناً يحتاج العثور على الخطأ البرمجي بعض الوقت وحسب. غالباً ما أعثر على الأخطاء عندما أترك عقلي هائماً على غير هدى. من الأماكن المناسبة للعثور على الأخطاء البرمجية القطارات، والحمامات، والسرير.

لا، أنا أحتاج المساعدة بحق

ذلك يحصل. حتى أمهر المبرمجين يعلقون. أحياناً تحتاج إلى عينين جديدتين.

قبل أن تدعو أحداً آخر لمساعدتك، تأكد من تجريب كل الأساليب الواردة أعلاه. يجب أن يكون برنامجك بسيطاً بقدر المستطاع، وعليك أن تعمل مع أبسط الحالات التي تسبب الخطأ. يجب أن تكتب تعليمات طباعة في الأماكن المناسبة (ويجب أن يكون الخرج الذي تولده مفهوماً). عليك أن تفهم المشكلة بشكل جيد حتى تصفها بدقة.

عندما تطلب أحداً ليساعدك، أعطه المعلومات التي يحتاجها.

- ما نوع الخطأ البرمجي؟ نحوي، منطقي أم خطأ تشغيل؟
- ما هو آخر شيء عملته قبل حدوث هذا الخطأ؟ ما هي أسطر الشفرة الأخيرة التي كتبتها، أو ما هي حالة الاختبار الجديدة التي فشل البرنامج فيها؟
- إذا حدثت المشكلة أثناء تجميع البرنامج أو تشغيله، ما هي رسالة الخطأ، وإلى أي جزء من البرنامج تشير؟
- ماذا جرّبت لحل المشكلة، وماذا تعلمت؟

قد ترى الحل أثناء شرحك للمشكلة إلى شخص آخر. هذه الظاهرة شائعة لدرجة أن بعض الناس ينصح بأسلوب لتصحيح الأخطاء يتم بالاستعانة ببطة مطاطية وهو يدعى بالإنكليزية ("rubber ducking"). وهذه الطريقة تتم كما يلي:

1. اشتر بطة مطاطية من النوع العادي.
2. عندما تواجه مشكلة حقيقية فعلاً، ضع البطة على المكتب أمامك وقل لها، "أيتها البطة، أنا عالق فعلاً في مشكلة. إليكِ ما يحدث معي..."
3. اشرح المشكلة للبطة.
4. استنتج الحل.
5. اشكر البطة المطاطية.

أنا لا أمزح. انظر http://en.wikipedia.org/wiki/Rubber_duck_debugging.

لقد عثرت على الحشرة!

عندما تعثر على الخطأ البرمجي، عادة ما يكون إصلاحه واضحاً. لكن ليس دائماً. أحياناً يكون الشيء الذي تعتقد أنه خطأ برمجي عادي إشارة لعدم فهمك للبرنامج، أو لوجود خطأ في خوارزمية الحل. في هذه الحالات، قد يتوجب عليك إعادة النظر في الخوارزمية، أو تعديل النموذج العقلي للبرنامج. خذ بعض الوقت بعيداً عن الحاسب للتفكير، نفذ حالات الاختبار يدوياً، أو ارسم مخططات تمثل الحسابات.

بعد إصلاح الخطأ البرمجي، لا تنطلق مباشرة لصنع المزيد من الأخطاء. خذ دقيقة لتفكر: ما كان نوع الخطأ، لم حدث هذا الخطأ، كيف خبّأ الخطأ نفسه، ما الذي أمكنك فعله حتى تجده أسرع. عندما ترى شيئاً مشابهاً في المرة القادمة، ستقدر على إيجاد الخطأ بسرعة أكبر.

تَمَّ وَالحَمْدُ لِلَّهِ