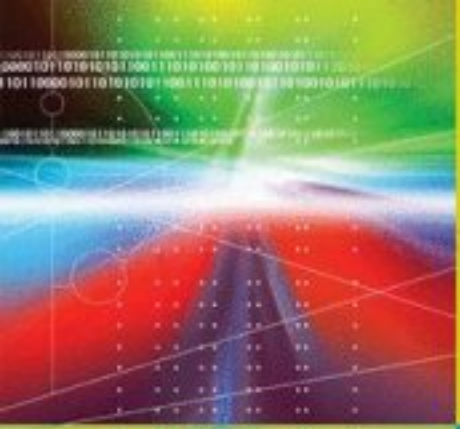


Frank Nielsen

UNDERGRADUATE TOPICS in COMPUTER SCIENCE

A Concise and Practical Introduction to Programming Algorithms in Java

Springer

UTiCS

Undergraduate Topics in Computer Science

Undergraduate Topics in Computer Science' (UTiCS) delivers high-quality instructional content for undergraduates studying in all areas of computing and information science. From core foundational and theoretical material to final-year topics and applications, UTiCS books take a fresh, concise, and modern approach and are ideal for self-study or for a one- or two-semester course. The texts are all authored by established experts in their fields, reviewed by an international advisory board, and contain numerous examples and problems. Many include fully worked solutions.

Also in this series

Iain D. Craig
Object-Oriented Programming Languages: Interpretation
978-1-84628-773-2

Max Bramer
Principles of Data Mining
978-1-84628-765-7

Hanne Riis Nielson and Flemming Nielson
Semantics with Applications: An Appetizer
978-1-84628-691-9

Michael Kifer and Scott A. Smolka
Introduction to Operating System Design and Implementation: The OSP 2 Approcah
978-1-84628-842-5

Phil Brooke and Richard Paige
Practical Distributed Processing
978-1-84628-840-1

Frank Klawonn
Computer Graphics with Java
978-1-84628-847-0

David Salomon
A Concise Introduction to Data Compression
978-1-84800-071-1

David Makinson
Sets, Logic and Maths for Computing
978-1-84628-844-9

Orit Hazzan
Agile Software Engineering
978-1-84800-198-5

Pankaj Jalote
A Concise Introduction to Software Engineering
978-1-84800-301-9

Alan P. Parkes
A Concise Introduction to Languages and Machines
978-1-84800-120-6

Gilles Dowek
Principles of Programming Languages
978-1-84882-031-9

Frank Nielsen

A Concise and Practical Introduction to Programming Algorithms in Java

Springer

Frank Nielsen
École Polytechnique
Paris
France

Sony Computer Science Laboratories, Inc.
Tokyo
Japan

Undergraduate Topics in Computer Science ISSN 1863-7310
ISBN 978-1-84882-338-9 e-ISBN 978-1-84882-339-6
DOI 10.1007/978-1-84882-339-6

British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library

Library of Congress Control Number: 2009921195

Printed on acid-free paper

Springer Science+Business Media
springer.com

To my family,
To Audrey and Julien.

"Dinosaur," Julien, October 4th, 2008 (painting, 5 years old).

Preface

This concise textbook has been primarily designed for undergraduate students as a very first course in programming. The book requires no prior knowledge of programming nor algorithms. It provides a gentle introduction to these topics. The contents of this book have been organized into ten chapters split over *two parts*, as follows:

– The first part is concerned with *getting ready* to program basic tasks using the modern language Java$^{\text{TM}}$. The fundamental notions of variables, expressions, assignments with type checking are first explained. We present the conditional and loop statements that allow programmers to control the instruction work flows. The concepts of functions with pass-by-value arguments and recursion are explained. We proceed by presenting arrays and data encapsulation using objects, and insist on the notion of references for the latter.

– The second part of the book focuses on *data-structures* and *algorithms*. We first describe the fundamental sequential and bisection search techniques, and analyze their respective efficiency using complexity analysis. Since the effective bisection search requires sorted data, we then explain basic iterative and recursive sorting algorithms. We follow by explaining linked lists and describe common insertion/deletion/merge operations on them. We then introduce the concept of abstract data-structures (illustrating them with queues and stacks) and explain how to program them in Java using the object-oriented style methods. Finally, the last chapter is an introduction to more evolved algorithmic tasks that tackle combinatorial optimization problems.

The goal of this book is *two-fold*: Namely, during the first part, novice programmers progressively learn the basic concepts underlying most imperative programming languages using Java. The second part then introduces fresh pro-

grammers with the very basic principles of thinking the algorithmic way, and explain how to turn these algorithms into programs using the programming concepts of Java. The book progressively conveys to the reader that "programming" is in fact a complex task that consists of modeling a given problem, designing algorithms and purposely structuring its data for solving the problem, coding the algorithm into a program, and finally, testing the program.
Each chapter of the book concludes with a set of exercises that lets students practice notions covered by the chapter. The third part of the book consists of an overall exam that allows readers to evaluate their assimilation level. A solution is provided. Exercises and sections that are recommended to be skimmed through in a first reading are indicated using the mark **.
Additional materials, including all Java source codes for each chapter, are available at the following book web page:

`http://www.lix.polytechnique.fr/Labo/Frank.Nielsen/JavaProgramming/`

Preface for Instructors

The Java programming curriculum (called INF311) from which this book has been prepared has been taught at École Polytechnique (Palaiseau, France) for many years. Every year about 250 students enroll into the curriculum. For most of them, it is their first experience with the Java programming language. Some of them have a prior programming experience using mathematical packages such as Maple^{TM} which are interpreters. This yields an important source of confusion not only from the standpoint of the language syntax but also from the conceptual sides of using an imperative programming language. The INF311 curriculum does not assume any prior programming experience and concentrates on teaching the fundamental notions of programming an imperative object-oriented language. This book is intended as a first programming course with two objectives in mind:

– Get a firsthand experience on programming basic algorithms using basic features of Java, and

– Introduce the very first fundamental concepts underlying computer science (that is, complexity analysis, decidability, abstract data-structures, etc.).

The curriculum consists of ten lectures (each of them lasts *90 minutes*) that deal with the following topics:

Lecture 1	Variables, expressions and assignments (with an introduction to the science of computing)
Lecture 2	Conditional and loop statements
Lecture 3	Static functions and recursions
Lecture 4	Arrays
Lecture 5	Objects (data encapsulation without object methods)
Lecture 6	Rehearsal for mid-term programming exam
Lecture 7	Searching and sorting
Lecture 8	Linked lists
Lecture 9	Data-structures and object methods
Lecture 10	Combinatorial optimization algorithms

A first course in programming without hands-on experience on writing programs by oneself is simply not conceivable. That is why each lecture is followed by a two-hour programming training class to let students become familiar with the notions covered during the lectures, and experience for themselves tracking and correcting bugs.
To control the level of assimilation by students, we organize at mid-term of the curriculum a two-hour programming exam that is semi-automatically checked using scripts and Java input/output redirections. The final exam is a two-hour paper exam that focuses more on checking whether students understand the basic data-structures and algorithms. A review exam with a detailed solution is provided in Chapter 11 (page 227).
The pedagogic resources, which include slides of each lecture and recorded videos of the lectures taught in the auditorium, are available at the following web page:

`http://www.enseignement.polytechnique.fr/informatique/INF311/`

Frank Nielsen December 2008.

École Polytechnique (Palaiseau, France)
Sony Computer Science Laboratories, Inc. (Tokyo, Japan)

Acknowledgments

It is my great pleasure to acknowledge my colleagues, Professors François Morain and Robert Cori at École Polytechnique (France), who taught this introductory course. I have greatly benefited from their course materials but more importantly from their feedback and advice. I express my deepest thanks to Philippe Chassignet and the full team of teaching assistants for their shared

passion of letting students get their first programs to compile and work; Many thanks to Bogdan Cautis, Guillaume Chapuy, Philippe Chassignet, Etienne Duris, Luca de Féo, Yann Hendel, Andrey Ivanov, Vincent Jost, Marc Kaplan, Gaëtan Laurent, David Monniaux, Giacomo Nannicini, Sylvain Pradalier, Stéphane Redon Maria Naya Plasencia, Andrea Roeck, David Savourey, and Olivier Serre.

This book became possible thanks to the warm encouragement and support from Sony Computer Science Laboratories, Inc. I express my gratitude to Dr. Mario Tokoro and Dr. Hiroaki Kitano, as well as all other members of Sony Computer Science Laboratories, Inc.

I thank Ian Mackie for asking me to write this undergraduate textbook for the Undergraduate Topics in Computer Science series on behalf of Springer-Verlag.

I apologize for any (involuntary) name omissions and remaining errors.

Finally my deepest thanks and love go to my family for their everlasting support.

Contents

Part II. Data-Structures & Algorithms

List of Figures

List of Tables

Listings

1
Expressions, Variables and Assignments

1.1 Introduction

In this chapter, we cover the very basics lying at the heart of every Java program. That is, we introduce the minimalist program that forms the skeleton common to all programs, and explain the programming cycle of writing, compiling, editing and executing code. We provide an overview of the key concepts of typed language and describe the primitive types of Java. Finally, we present the syntax of arithmetic operators and give details on basic input/output operations for writing our first programs.

1.2 My first Java programs

1.2.1 A minimalist program

When learning any new programming language, it is traditional to start by first looking at what programmers call the *basic skeleton* of any program. This skeleton lets them gain some understanding of the language syntax and presents the necessary wrapping code. First programs look often mystic, if not weird, since they reveal at once some of the key syntax components of the programming language. In its simplest form, consider the following "shortest" Java program:

F. Nielsen, *A Concise and Practical Introduction to Programming Algorithms in Java*, Undergraduate Topics in Computer Science, DOI 10.1007/978-1-84882-339-6_1,

Program 1.1 My first Java program — A minimalist approach

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

At first, everything in the above code seems strange to novice programmers. In fact, we will have to wait until Chapter 5 to get a full understanding of what the first two lines actually mean. For now let us ignore these lines. To execute this program, we need to perform the following three steps:

- **Type** this program into any text editor[1] and store this file under filename `MyFirstProgram.java`. Extensions `.java` means that the file is a Java *source code*. The name of the file should bear the name appearing after the `class` keyword.
- **Compile** the program by typing at the console: `javac MyFirstProgram.java`. This will generate a compiled file, called the *bytecode*, bearing the name `MyFirstProgram.class`. Check in the directory[2] that this file was created.
- **Execute** the program by typing the command `java MyFirstProgram` at the console prompt. This will execute the bytecode of `MyFirstProgram.class`. That is, it will run the program on the java virtual machine[3] (JVM).

Well, the program ran correctly but produced no apparent result, so that it is not very convincing to us that something really took place on the processor. In fact, the `main` function of the program was called upon execution, and the set of instructions contained inside the function was executed stepwise. Here, there was nothing to execute since the body of the `main` function is empty; the body of the `main` function is encapsulated into the program class `MyFirstProgram` and is delimited by the opening/closing of braces {}, also called curly brackets. So let us spice up this program by writing a message on the console output.

1.2.2 Hello World

The "Hello Word" program is *the* emblematic program for comparing at a glance the syntax of different programming languages. What we need to do is to add a single *instruction* line inside the former program to *display* the message

[1] Like `Notepad` under Windows™ or `Nedit` in Linux-based KDE environments.
[2] In Windows, type `dir` at the console prompt. In Unix, use the equivalent `ls` command.
[3] Java bytecode is cross-platform, which means that provided that there exists a JVM implementation for the target machine, you will be able to run your bytecode on that environment. This is one of the key strengths of Java in today's market.

"Hello World." Let us choose the program name to be `HelloWorld.java`. Then we need to label the program class with that name too. To display a message, we use the instruction System.out.println(message);. Instructions are followed by a semi-colon ";" mark. Thus we get our slightly revised program:

Program 1.2 The Hello World Java program

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

Again, let us go through the workflow of editing/compiling and running the program as follows:

- **Type** this program and store it under filename `HelloWorld.java`.
- **Compile** this program by typing in the console: `javac HelloWorld` (implicitly meaning `javac HelloWorld.java`). A bytecode `HelloWorld.class` is produced in the same directory. The bytecode is not human readable, only machine readable! Try to open the file and look at the strange sequence of symbols that encode this compiled program. At this stage, it should be clear that files ending with the extension `.java` are source codes, and files ending with extensions `.class` are bytecodes.
- **Execute** the program by typing the command `java HelloWorld` (implicitly meaning `java HelloWorld.class`) at the console prompt.

You will now get the result visible in the output console as follows:

```
Hello World
```

Congratulations, you successfully wrote your first program. Let us now see how to perform arithmetic calculations.

1.3 Expressions and programs as calculators

The first program you may think of is to compute formulas[4] for various input. Assume, for example, we are given a 3D box with the following dimensions:

[4] Historically, this objective was one of the very first motivations for programming languages. FORTRAN, which stands for FORmula TRANslation, is such an example that was widely used by physicists for simulation purposes. Although FORTRAN is nowadays a bit outdated, there is still a lot of legacy code running daily (for example, weather forecasting).

50cm in width, 100cm in height and 20cm in depth. We simply do compute the volume in cubic meters m^3 by converting the dimension units into meter equivalents and performing the product of these dimensions to get the volume. Thus the volume of that 3D box is

$$0.5m \times 1m \times 0.2m = 0.1m^3.$$

How do we program this? We simply need to *evaluate* the arithmetic expression $0.5 \times 1 \times 0.2$

Program 1.3 Expression: Evaluating the volume of a 3D box

```
class VolumeBox{
public static void main (String[ ] args)
{
System.out.println(0.5*1*0.2);
}
}
```

Storing the above program into filename `VolumeBox.java`, compiling and executing it, we get the expected output:

```
0.1
```

Thus we can *calculate* the value of *any* arithmetic expression, say the generic `myExpression` expression, and *display* its value by executing the instruction System.out.println(myExpression); Of course, this straight number alone is not very informative, so it is better to display a message on the console telling what the number really means. We do this by printing the message without the return carriage (line return) using the instruction System.out.print.

```
class VerboseVolumeBox{
public static void main (String[ ] args)
{
System.out.print("Volume of the box (in cubic meters):");
System.out.println(0.5*1*0.2);
}
}
```

Compiling and running this program yields the better verbose output:

```
Volume of the box (in cubic meters):0.1
```

1.3.1 Arithmetic operations and priority order

The arithmetic operations used in expressions are:

– The addition (+)

– The subtraction (-)

– The multiplication (*)

– The division (/)

– The modulo (remainder, %)

These operations depend on the type of operands, and yield the usual bugs. For example, consider the variable q defined as:

```
int q=2/3;
```

This initializes the integer variable q to 0 since the division / is the Euclidean division.[5]
Programmers have to take care with variable initializations, especially when implicit casting occurs. To illustrate these points, consider the following code snippet:

```
double qq=2/3;
double qqq=2/3.0;
```

Although variable qq is declared of type double the division operands 2 and 3 have been identified as integers so that the compiler will compute the Euclidean division and get the integer 0, which will then be implicitly cast into a double (see Figure 1.1): 0.0. However, when declaring and initializing double variable qqq, since the second operand is of type double, the first operand will be cast into a double, and the double division will be performed yielding the expected result: 0.6666666666666666.
The operators *unambiguously* satisfy priority rules so that parentheses may be omitted when forming expressions for ease of reading. For example, consider the expression 7+9*6. This expression admits two kinds of parentheses: (7+9)*6 and 7+(9*6). But since the multiplication has higher priority over the addition, it is understood that the expression 7+9*6 is meant to be 7+(9*6), and evaluates to 61.
As we will see in the next chapter, an important class of expressions are *boolean expressions*, which are used in program control structures. Boolean expressions admit only two outcomes: true or false. The most common *logical operators* are && for AND and for OR. Thus for boolean variables a and b, the boolean expression a&&b; is evaluated to true if and only if both a and b are true. The following program presents the use of boolean expressions and boolean variable assignments:

[5] In Euclidean division, x/y computes the Euclidean ratio of x by y, and x%y computes the remainder.

Program 1.4 Boolean expressions and boolean variable assignments

```
class BooleanExpression{

public static void main (String[ ] args)
{
boolean a=true;
boolean b=false;

boolean expr1=a||b;
boolean expr2=a&&b;

System.out.print("a||b=");
System.out.println(expr1);

System.out.print("a&&b=");
System.out.println(expr2);
}
}
```

Compiling and running this program, we get the following console output:

```
a||b=true
a&&b=false
```

Boolean expressions and their role in program workflows will be presented in the next chapter.

1.3.2 Mathematical functions

In Java, mathematical functions including trigonometric sine and cosine functions and constants are all encapsulated into the `Math` class. The most usual functions[6] are summarized in the table below:

[6] Refer to the on-line documentation at `http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Math.html` for complete description of the available set of functions.

`Math.PI`	π (3.141592...)
`Math.E`	e (2.71828...)
`Math.abs(x)`	`x` (absolute value)
`Math.ceil(x)`	$\lceil x \rceil$ (ceil function)
`Math.round(x)`	round to nearest integer
`Math.sqrt(x)`	$\sqrt{x}$ (square root function)
`Math.log(x)`	$\ln x$ ($\log_e x$)
`Math.log10(x)`	$\log_{10} x$
`Math.pow(x,y)`	x^y (power function)
`Math.cos(x)`	$\cos x$ (cosine function)
`Math.sin(x)`	$\sin x$ (sine function)
`Math.tan(x)`	$\tan x$ (tangent function)

The program below demonstrates a more elaborate program for computing a formula:

Program 1.5 Program demonstrating the use of mathematical functions

```
class MathFunction{
public static void main (String[ ] args)
{
double x=Math.E;
double fx=Math.log(x);
System.out.print("Is this precisely 1 or are there numerical
    errors? ");
System.out.println(fx);


x=Math.PI/15.0;
fx=Math.sin(x)*Math.sin(x)+Math.cos(x)*Math.cos(x);
System.out.print("What about this trigonometric equality (
    should be 1) ?");
System.out.println(fx);
}
}
```

The output of the program is:

```
Is this precisely 1 or are there numerical errors? 1.0
What about this trigonometric equality (should be 1) ?1.0000000000000002
```

Note that the constant `Math.E` is defined as the `double` value that is closer than any other to e. Observe that in the second case, we should mathematically have the identity $\sin^2\theta + \cos^2\theta = 1$, but due to numerical imprecisions, we end up only with a very close number: 1.0000000000000002.

1.3.3 Declaring constants

Constants are declared by prepending to the type of variables the keyword `static`. That is, constants are *immutable variables*. Constants are also typed too. For instance, we may wish to define our own approximation of the mathematical constant π as follows:

```
final double PI = 3.14;
```

The declaration of constants should be made inside the body of the class (and not inside functions).

Program 1.6 Declaring constants

```
class ConstantDeclaration
{
final static int One=1;
final static int Two=2;

public static void main(String [] args)
{
int Three=One+Two;
System.out.println(Three);
}
}
```

1.4 Commenting Java programs

It is useful to comment programs to describe parts of the code and the various input/output formats of programs. Not only does it become crucial for us when revisiting a program months later, it also becomes absolutely necessary when sharing a program with others. Java has two types of comments: single lines and multiple lines illustrated as below:

```
// This is a single line comment
/* I can also write comments on
several lines
by using these delimiters */
```

Let us illustrate the comment syntax by commenting the former `VerboseVolumeBox.java` program:

```
/*
   This program computes the volume of a 3D box
*/
class VerboseVolumeBox{
```

```
public static void main (String[ ] args)
{
System.out.print("Volume of the box (in cubic meter):");
// Result is in cubic meter unit
System.out.println(0.5*1*0.2);
}
}
```

1.5 Indenting programs

Java source codes are parsed by the compiler `javac`, which checks the *syntax* of the program and generates an overly optimized bytecode that is a low-level machine-interpretable bytecode. Indenting a program consists of making the source code prettier by adding a few extra spaces or return lines. Indenting consists of applying several conventions like aligning columns of opening/closing braces, etc. Code indentation does not change the bytecode. For example, the following badly indented source code will produce the same bytecode although it obfuscates its readibility.

```
class NotIndentedVolumeBox{public static void main (String[ ]
    args){System.out.print("Volume of the box (in cubic
   meter):");System.out.println(0.5*1*0.2);}}
```

As a rule of thumb, it is important for programmers to comment and indent source codes well to improve their readibility. Indenting also helps browsing lengthy source codes.

1.6 Variables, assignments and type checking

Suppose now that we would like to compute the balance of a set of credits with a set of debits. We first need to compute the total credit figure, then the total debit figure and subtract these two figures to get the balance. Computing the total credit and debit can be done and displayed using the former syntax System.out.println(myExpression);. But what about the balance? For example, consider we have two credit lines (say, 100 and 150 dollars), and three debit lines (50, 25 and 100 dollars). Then, we would have the following code:

Program 1.7 Sketch of the balance sheet program

```
class BalanceSheet{
public static void main (String[ ] args)
```

```
{
System.out.print("Total credit (in US dollars):\t");
System.out.println(100+150);

System.out.print("Total debit (in US dollars):\t");
System.out.println(50+25+100);

System.out.print("Balance:");
}
}
```

Running this program, we get:

```
Total credit (in US dollars):   250
Total debit (in US dollars):    175
```

Note that the \t inside the string `"Total credit (in US dollars):\t"` denotes the *tabulation* character that allows one to nicely align the latter numbers. The problem is to get the balance we need to subtract 175 from 250. Of course, this could be computed and displayed with the instruction:

```
System.out.println((100+150)-(50+25+100));
```

...but this will not yield an efficient approach since we would again compute the two cumulative credit/debit sums. A much better strategy consists of storing intermediate calculations in containers by using variables, as explained next.

1.6.1 Variables for storing intermediate values

In Java, variables are all *typed*. Java belongs to the large category of typed programming languages. This means that we need to specify the type of variables when declaring variables. To declare a variable named `credit` for storing the overall credit modeled as an integer number, we use the syntax:

```
int credit;
```

Variables always need to be declared before use. By convention, we choose in this textbook to declare all variables at the beginning of the `main` block (delimited by the braces {}). In this application, we consider that credit/debit numbers are natural numbers (integers), so that we declare the variable to be of type `int`.

Program 1.8 Balance sheet using integer variables

```
class BalanceSheet2{
public static void main (String[ ] args)
{
int credit;
int debit;
```

```
credit=100+150;
System.out.print("Total credit (in US dollars):\t");
System.out.println(credit);

debit=50+25+100;
System.out.print("Total debit (in US dollars):\t");
System.out.println(debit);

System.out.print("Balance:");
System.out.println(credit-debit);
}
}
```

Running the above program, we get the console output:

```
Total credit (in US dollars):   250
Total debit (in US dollars):    175
Balance:75
```

In Java, we can choose to declare variables using one of the following *primitive types*[7]: `int` (integer simple precision stored onto a machine word of 32 bits), `long` (integer double precision stored onto two machine words, 64 bits), `float` (simple precision real stored onto a machine word), `double` (double precision real), `char` (character encoding worldwide language characters using 16 bits[8]) and `boolean` (two states: `true` or `false`). Table 1.1 lists the primitive types of Java with their respective encoding representations and range of values. All primitive types of Java are manipulated by value.

Boolean `boolean`	1 bit	`true` or `false`
Character `char`	2 bytes	UNICODE
Integer `byte`	1 byte	$[-128, 128]$
Integer `short`	2 bytes	$[-32768, 32767]$
Integer `int`	4 bytes	$[-2^{31} = -2147483648, 2^{31} - 1 = 21477483647]$
Integer `long`	8 bytes	$[-2^{63} = -9223372036854775808, 2^{63} - 1 = 9223372036854775807]$
Real `float`	4 bytes	$[1.40129846432481707e - 45f, 3.40282346638528860e + 38f]$
Real `double`	8 bytes	$[2.2250738585072014e - 308d, 1.79769313486231570e + 308d]$

Table 1.1 Primitive data types of Java.

We can also compactly declare *and* initialize variables at once as follows:

[7] There is also the less used `byte` and `short` primitive types that we voluntarily omit in this book.

[8] Java uses the UNICODE standard for coding characters using 16 bits (2 bytes). This allows programmers to encode a wide range of characters including Chinese/Korean and Japanese kanji. The traditional 8-bit ASCII encoding is limited to the Roman alphabet.

```
int credit1=100, credit2=150;
int credit=credit1+credit2;
int debit1=50, debit2=25, debit3=100;
int debit=debit1+debit2+debit3;
int balance=credit-debit;

System.out.print("Balance:");
System.out.println(balance);
```

This example illustrates two constructions:

– Variable declaration and assignment to a constant (a *basic expression* — see for example, variable `credit1`)

– Variable declaration and assignment to an *arithmetic expression* (see for example, variable `debit1`).

As we have formerly seen, variables are useful for storing initial values. Thus these variables play an important role in the initialization of programs. For example, the former program for computing the volume of a 3D box can be rewritten as:

Program 1.9 Volume of a 3D box using `double` type variables

```
class VerboseVolumeBox2{
public static void main (String[ ] args)
{
double width=0.5, height=1.0, depth=0.2;

System.out.print("Volume of the box (in cubic meter):");
System.out.println(width*height*depth);
}
}
```

Java distinguishes upper from lower case in variable names, as exemplified below:

Program 1.10 Java distinguishes upper/lower cases

```
class UpperLowerCase
{
public static void main (String arguments[])
{
int MyVar;
// myvar variable is different from MyVar
int myvar;
// Generate a syntax error at compile time:
// cannot find symbol variable myVar
System.out.println(myVar);
}
}
```

That is, Java is *case-sensitive*.

1.6.2 Type checking for assignments and casting

1.6.2.1 Type checking. Whenever assigning a variable, the compiler `javac` checks that the type of the expression coincides with the type of the variable. If not, the compiler reports an error message and aborts (without generating proper bytecode). For example, consider the code below:

```
class TypeError{
public static void main (String[ ] args)
{
double myFavoriteReal=3.141592;
int myFavoriteNat=2.71;
}
}
```

Compiling this code will generate an error as follow:

```
prompt%javac TypeError.java
TypeError.java:5: possible loss of precision
found   : double
required: int
int myFavoriteNat=2.71;
                  ^
1 error

prompt%
```

That is, the compiler performed the type checking and found that the type of the expression (here, the real constant 2.71 of type `double`) does not agree with the type of the `int` variable `myFavoriteNat`. Typing provides an essential *safeguard mechanism* for writing more coherent programs, that is programs without obvious bugs.

1.6.2.2 Casting types. We can eventually transform that real 2.71 by truncating it into the integer 2 by a casting operation:

```
int myFavoriteNat=(int)2.71;
```

This results in a loss of precision, as noticed by the compiler.
In general, we can explicitly cast a type `TypeExpr` into a `TypeVar` using the following syntax:

```
TypeVar myVar=(TypeExpr)(Expression);
```

Typed languages such as Java are useful to abstract notions and types of variables characterizing semantic parameters. In college, we are familiar with similar notions to assign quantity of the same units. For example, it does not make sense to assign to a velocity (type $m \times s^{-1}$) an acceleration (type $m \times s^{-2}$). In fact, some earlier casting operations were already carried out when declaring and initializing constants:

```
double x=2; // implicit casting  int->double
double x=(double)2;// explicit casting
long x=2.0; // implicit casting double->long
double x=2.0d; // The constant 2.0 is declared of type double
                            // add a 'd' after the number
```

The implicit casting rules of primitive types are summarized in Figure 1.1.

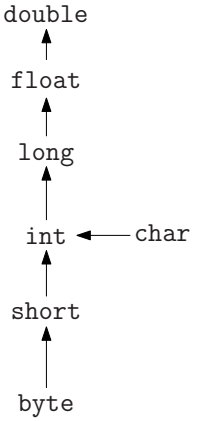


Figure 1.1 Implicit casting rules of primitive types

For example, consider the following code snippet that implicitly casts a character (of type `char`) into its corresponding ASCII integer code:

```
char c='X';
int code=c;
System.out.println(code);
```

We get the ASCII code 88 for the capital letter 'X'.
There are slight restrictions on the names of variables that should not begin with a digit, nor bear the name of a reserved keyword of the language. Trying to use a reserved keyword for the name of a variable will result in a compiler error as illustrated in the following code:

Program 1.11 Variable names should not belong to the list of reserved keywords

```
class ReservedKeyword
{public static void main (String arg[]){
double x,y;
// Generate a syntax error:
// "not a statement"
int import;
}
}
```

The list of reserved keywords for Java is: `abstract`, `assert`, `boolean`, `break`, `byte`, `case`, `catch`, `char`, `class`, `const`, `continue`, `default`, `do`, `double`, `else`, `extends`, `false`, `final`, `finally`, `float`, `for`, `goto`, `if`, `implements`, `import`, `instanceof`, `int`, `interface`, `long`, `native`, `new`, `null`, `package`, `private`, `protected`, `public`, `return`, `short`, `static`, `strictfp`, `super`, `switch`, `synchronized`, `this`, `throw`, `throws`, `transient`, `true`, `try`, `void`, `volatile`, `while` .

1.6.3 The inner mechanisms of assignments

The prototype instruction for assigning a variable `var` of type `TypeVar` is the following atomic instruction:

```
var=Expression;
```

The compiler proceeds the following three steps:

- **Evaluate** the expression `Expression` into a value,
- **Check** that the type of the value coincides with the type of the variable (type checking). If not, check whether some implicit casting rules apply (see Figure 1.1). If types cannot coincide, then the compiler generates an error message and aborts.
- **Store** the value of the expression at the *memory location* referenced by the variable.

We further explain these two sides value/memory location of variables in the next section dealing with special assignment instructions: variable increments.

1.7 Incrementing/decrementing variables

1.7.1 General mechanism for incrementation

Incrementing a variable `x` by an amount `increment` (that is, incrementing the variable `x` by a step of width `increment`) consists of adding to the value of the variable the value of the `increment`, and in storing the result at the memory location referenced by variable `x`. Incrementing a variable without specifying its amount means to add one to its value. Incrementation is thus nothing other than a particular form of assignment where a variable appears on *both sides* of the equality sign =:

```
x=x+increment;
```

The variable on the left hand side means "**store** at the memory location referenced by that variable," while the variable on the right hand side means "**get** the value stored at the memory location referenced by that variable." For programmer novices, the instruction `x=x+increment;` is quite confusing at first since it makes no mathematical sense. Let us deconstruct the action taken by the compiler when encountering such an instruction:

– Evaluate arithmetic expression `x+increment`:
  – Perform type checking of `increment` with `x` (cast `increment` type if necessary),
  – Get the value `xVal` stored at memory location referenced by `x`,
  – Get the value `incrementVal` stored at memory location referenced by `increment`,
  – Return the value `xVal+incrementVal`.
– Finally, store the expression value at memory location referenced by `x`.

Instead of writing `x=x+increment`, we can equivalently write this instruction compactly using the following shortcut:

```
x+=increment
```

Similarly, we can decrement a variable by a given step as follows:

```
x-=increment
```

These basic shortcuts extend[9] also to the / and ∗ operators:

```
int y=1;
y*=3; // y=3, multiplication assignment
int z=12;
z/=2; // z=6, division assignment
int p=23;
p%=2; // p=1, modulo assignment
```

1.7.2 Pre-incrementation and post-incrementation

Often we need to add or subtract *one* to (the value of) a variable, say `x`. This can be done as follows:

```
x=x+1;
x+=1; // compact form
x=x-1;
x-=1; // compact form
```

[9] These shortcut instructions are, however, rarely used in practice.

These incrementation instructions are so frequent that they can be either written compactly as `++x` or `x++` (for pre-incrementation and post-incrementation). Let us explain the difference between *pre-incrementation* and *post-incrementation*:
Consider the following code:

```
i=2;
j=i++;
```

This gives the value 3 to `i` and 2 to `j` as we do a *post-incrementation*. That is, we increment after having evaluated the expression `i++`. The above code is equivalent to:

```
i=2;
j=i;
i=i+1; // or equivalently i+=1;
```

Alternatively, consider the pre-incrementation code:

```
i=2;
j=++i;
```

This code is equivalent to...

```
i=2;
i=i+1;
j=i;
```

... and thus both the values of i and j are equal to 3 in this case. The same explanations hold for pre-decremention $--$i and post-decrementation i$--$. Note that technically speaking the `++` syntax can be seen as a *unary operator*.

1.7.3 A calculator for solving quadratic equations

Let us put altogether the use of mathematical functions with variable initializations and assignments in a simple program that computes the roots $b \pm \frac{\sqrt{b^2-4ac}}{2a}$ of a quadratic equation $ax^2 + bx + c = 0$:

Program 1.12 A quadratic equation solver

```
class QuadraticEquationSolver
{
public static void main(String[] arg)
{
double a,b,c;
a=Math.sqrt(3.0);
b=2.0;
c=-3.0;
double delta=b*b-4.0*a*c;
double root1, root2;
```

```
root1= (-b-Math.sqrt(delta))/(2.0*a);
root2= (-b+Math.sqrt(delta))/(2.0*a);
System.out.println(root1);
System.out.println(root2);
System.out.println("Let us check the roots:");
System.out.println(a*root1*root1+b*root1+c);
System.out.println(a*root2*root2+b*root2+c);
}
}
```

Note that for this particular initialization, we have `delta>0`. Otherwise, there are imaginary roots, and the program will crash in the `Math.sqrt` function.

1.8 Basics of Java input/output (I/O)

In this section, we quickly review the elementary instructions for reading or writing on the console. We will then see how to redirect input/output from/to files.

1.8.1 Computing does not mean displaying

In Java, one needs to *explicitly* display results on the console to read them back. Printing the results on the console is one way to retrieve or export the results of programs. This is very different from most mathematical symbolic systems such as Maple®,[10] which computes *and* displays results at once. For example, in Maple computing $2 + 2$ will not only compute the sum but also display the result 4:

```
>2+2;
        4
```

This is one frequent source of confusion for beginners who have had prior "programming" experience with such mathematical packages. Remember that in Java, we need to *explicitly* display results.

1.8.1.1 Displaying results on the console. To display messages on the console in Java, one invokes either `System.out.println` for writing a string message *with* a return line, or `System.out.print` for writing a message without the return line. One can also display integers, reals or strings. Since it is quite cumbersome to write several `System.out.print[ln]` at a row like:

[10] `http://www.maplesoft.com/`

```
System.out.println("The value of x is ");
System.out.print(x);
```

To bypass these two instructions, one often uses string concatenations when formatting messages compactly.

1.8.1.2 Displaying & String concatenations. The former code is equivalent to the following:

```
System.out.println("The value of x is "+x);
```

For novice programmers, this code looks like a *magic* expression for outputting messages on the console. However, this can also at first yield unexpected results for beginners like:

```
int x=3;
System.out.println("value="+x+1);
```

Since the printed message on the console is 31, and not 4 as one would expect. The correct result would have been obtained by setting parentheses around `x+1` as follows:

```
System.out.println("value="+(x+1));
```

The trick is that the `+` between `"value="` and `(x+1)` has a different semantic than the `+` in `x+1`. What happens is that in the former case, the argument `"value="+x+1` of System.out.println is evaluated following the priority order rules. That is, the compiler cast the integer value `x` into a string "3" and appends it to string "value=" giving the string "value=3." Then the integer 1 is converted into the string "1" and appended to "value=3" giving the final string "value=31." Setting parenthesis around $x+1$ let us change the priority order, so that $x+1$ is first evaluated to give integer 4, and then string concatenation are performed after casting the 4 into the corresponding string "4."
The following program summarizes and exemplifies our discussion:

```
int a=1, b=-2;
System.out.print("a="); System.out.print(a);
System.out.print(" b="); System.out.println(b);
System.out.println("a="+a+" b="+b);
String s1="Lecture in", s2=" Java";
String s=s1+s2;// Perform explicit string concatenation
System.out.println(s);
```

1.8.2 Keyboard input

Programs often require users to give *interactive* parameters that play an important role in initialization of programs. At each run, the program asks for

some user keyboard input to deliver the solution. Java I/O is quite elaborate compared to other similar imperative languages like C++ because it has been designed to fit the object-oriented style that will be explained later on. For now, consider reading input from the console using the following syntax:

Program 1.13 Reading an integer value

```
import java.util.*;

class KeyboardIntInput{
public static void main(String [] args)
{
Scanner keyboard=new Scanner(System.in);
int val;

System.out.print("Enter an integer please:");
val=keyboard.nextInt();
System.out.print("I read the following value:");
System.out.println(val);
}


}
```

The output of a running session gives:

```
Enter an integer please:5
I read the following value:5
```

In the former code, we again used two *magic* lines, **import** java.util.*; and Scanner keyboard=**new** Scanner(System.in);, which will be fully explained later, in the second part of this book.
We can similarly read `float` and `double` values using keyboard.nextFloat(); and keyboard.nextDouble() instructions:

```
System.out.print("Enter a single-precision real please:");
valf=keyboard.nextFloat();
System.out.print("I read the following value:");
System.out.println(valf);

System.out.print("Enter a double precision please:");
vald=keyboard.nextDouble();
System.out.print("I read the following value:");
System.out.println(vald);
```

In case the above code snippet yields an error at run time, which is due to the number formatting conventions [11], add the instruction keyboard.useLocale(Locale.US);

[11] In the US, real numbers are written using the a decimal dot "." while in European countries it is a decimal comma ",." By default, Java is installed with the country local settings.

1.8.3 File redirections

Instead of writing messages to the console, one can redirect them to a text file without changing the source code, using the > symbol in the command line. For example consider the following toy program, which asks for an integer and squares it:

```
import java.util.*;

class SmallProg{
public static void main(String [] args)
{
Scanner keyboard=new Scanner(System.in);
int val;
System.out.print("Enter an integer please:");
val=keyboard.nextInt();
val*=val; // squared the input value
System.out.println("Squared value="+val);
}
}
```

Running the program from the console, we get:

```
prompt%java SmallProg
Enter an integer please:5
Squared value=25
```

Let us now store the output message into a text filename `output.txt`. We have

```
prompt%java SmallProg >output.txt
4
```

The above number 4 is the number input by the user when running the program. Let us inspect the file named `output.txt` by opening it:

```
Enter an integer please:Squared value=16
```

Thus all messages printed out using the instruction System.out.print[ln] have been redirected to filename `output.txt`. Similarly, we can set the input of a program from a filename by redirecting the input from the command line using the symbol <. Let us create a file `input.txt` containing the (string) value 4. We can now execute the program by redirecting its input as follows:

```
prompt%java SmallProg <input.txt
Enter an integer please:Squared value=16
```

... or redirect both input/output:

```
prompt%java SmallProg <input.txt >output.txt
```

This will overwrite the former output file `output.txt`.

1.9 Bugs and the art of debugging

Writing programs that are *syntaxically correct* straight from the keyboard is very rare. This seldom happens. This is why whatever textbook on Java one may read, it will *never* replace the experience of programming. The experience of programming consists of getting familiar with the way of thinking about the architecture of a program and translating these thoughts into corresponding language instructions. Sometimes a small typographic error in the source code yields many subsequent listed errors by the compiler. Indeed, Java compiler `javac` complains a lot and compiler comments are often verbose to help programmers get to the point. Consider this program to train yourself to spot the syntax errors:

```
class SyntaxBug{

public static void main(String [] args)
{
/*
 Ask for a value
 /
double val, fval;
fval=3*val*Val+2;
System.out.println("f(x)="+val);
}

}
```

Trying to compile this code will result in the following compiler error message:

```
prompt%javac SyntaxBug.java
SyntaxBug.java:5: unclosed comment
/*
^
SyntaxBug.java:13: reached end of file while parsing
}?
 ^
2 errors
```

Let us correct the multiple-line comment that lacks the closing $*/$

```
class SyntaxBug{

public static void main(String [] args)
{
/*
 Ask for a value
 /
double val, fval;
fval=3*val*Val+2;
System.out.println("f(x)="+val);
```

```
}

}
```

Recompiling this code will then reveal the uppercase mistyping of variable `Val`. Once corrected, the program can finally be compiled into an executable bytecode. This toy example emphasizes the incremental strategy for correcting programs.

In general, correcting a program consists of first making it syntactically correct so that the compiler can indeed parse the code and generate a corresponding bytecode. Then comes the task of making the program *semantically correct*. That is, to test and better prove that the program indeed solves the problem in *all* instances and not only on a particular test suite. This is a far more difficult and challenging task that has not yet been solved today. For very specific codes like embarked codes on airplanes, there exist mathematical techniques[12] to prove that these codes are indeed correct and resilient to all kinds of input. But these techniques hold only for these very specific codes. So be prepared in your programming life to discover bugs later on, even if you have already successfully used some code for many years. Remember that unless you prove your code, you cannot guarantee that it is crash-free.

Proving that a program is correct is all the more difficult since numbers are represented in machines using *finite representations* (4 or 8 bytes depending on whether single or double precision is used). Thus usually remarkable mathematical identities like $e^{\log x} = \log e^x = x$ on variables may fail as they do not detect potential *overflow* problems that yield not only incorrect result but may potentially crash a system. For example, consider the two primitives types for storing natural numbers in Java:

– `int`: the *maximum* machine-representable `int` is $2^{31} - 1$.

– `long`: the *maximum* machine-representable `long` is $2^{63} - 1$.

Let us now perform some incrementation instructions as follows:

```
class OverflowBug{

public static void main(String [] args)
{
int n=2147483647; //2^31-1
int nval=n+1;
System.out.println("If n="+n+" then n+1="+nval);

long m=9223372036854775807L; // 2^63-1
long mval=m+1;
System.out.println("If m="+m+" then m+1="+mval);
}
```

[12] For example, the branch of so-called *static analysis*.

```
}
```

Note that we initialized the `long` variable `m` by appending a "L" to the long digit to tell the compiler that this is a natural number using double precision. Otherwise, the compiler would have complained that our integer (`int`) was too big. Now, compile and execute this basic code to see the incoherent results:

```
prompt%java OverflowBug
If n=2147483647 then n+1=-2147483648
If m=9223372036854775807 then m+1=-9223372036854775808
```

One gets negative numbers, quite an unexpected result, isn't it? Loosely speaking, what happened is that the incrementation generated a carry that was propagated to the sign bit of the number type (the 32th and 64th bit respectively). These numbers were then decoded as negative numbers.

1.10 Integrated development environments (IDEs)

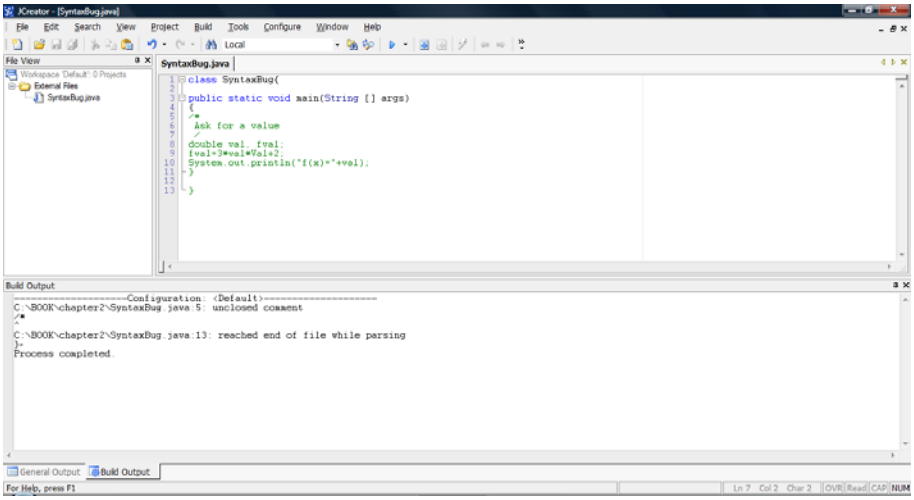


Figure 1.2 Snapshot of the `JCreator` integrated development environment

Once you get used to programming basic java programs, it will become more and more cumbersome to type the various compiler/execution commands. Even simple programs with small mistyped characters may generate long lists of

compiler errors that are difficult to browse. Programmers often used what is called an *integrated development environment* to increase the productivity cycle of writing programs. The most famous one is called `Eclipse`[13]. It is free of charge and used worldwide. Moreover, Eclipse is written using Java itself. Another free recommended IDE is `JCreator`. Using these IDEs, it is easy to compile and execute programs using the keyboard function F1-12 keys. Moreover, when the compiler reports a list of bugs, one can just click on the bug line to directly access the corresponding part of the source code. IDEs also help nicely indent code and highlight reserved keywords with various user personalized colors. It provides many functionalities (like expanding/shrinking code capabilities) to help navigate into codes. For example, looking at Figure 1.2, we would have found the bad closing comment even before compiling that code since the code would not have been nicely colored: That is, some instructions would have been colored using the comment green color instead of the regular black color.

1.11 Exercises

1.11.1 Note to instructors

The exercise section is primarily designed for a two-hour machine session with small groups of students. For some of the students, the first session will also be their very first hands-on experience using a Unix OS machine; Indeed, universities often provide a Linux-based park of consoles.[14] It is thus worthwhile describing the basic OS commands like `cd`, `mkdir`, `pwd`, `ls`, and job task commands. For Windows users, we recommend the `JCreator` v3.5 free integrated development environment.[15]

In this chapter, we have described the very basics for writing simple Java programs that calculate formulas. We have provided an overview of the framework for developing Java applications.

For this first exercise session, it is thus worth spending some time to get students familiar with the programming environment (usually on Windows or Linux operating systems). For example, as a warm-up:

– Create a simple Java source code that computes and prints to the output $\sin \frac{\pi}{6}$. Compile this code and execute it.

[13] `http://www.eclipse.org/`

[14] At École Polytechnique (Paris, France), we use the KDE environment (`http://www.kde.org/`) with the `Nedit` text editor (`http://www.nedit.org/`).

[15] Available for download at `http://www.jcreator.com/download.htm`

– Edit the former file to add $\cos\frac{\pi}{6}$ and check numerically whether $\sin^2\frac{\pi}{6} + \cos^2\frac{\pi}{6} = 1$ or not, recompile the code, etc.

– Copy the file and rename it to implement another simple program that computes, say, a triangle area, etc.

Then we suggest readers become familiar with input/output (I/O) file redirections (< and > symbols in command line) before proceeding to the following exercises.

1.11.2 First set of exercises

Exercise 1.1 (Time conversion)

Write a program `Convert.java` that converts a given number of seconds **int** seconds; into the format of hours, minutes and seconds. Print this equivalent number of seconds using a formatting rule like HH:MM:SS.

Exercise 1.2 (Time conversion with user keyboard input)

Modify the previous program to read the number of seconds from the console. Let us call this program `ConvertIO.java`. For example, the program execution will yield the following session:

```
How many seconds? 4000
4000 s = 1 hour, 6 min and 40 sec
```

Create an input text file `inputsec.txt` that contains the line `4000`, and run the program by redirecting both input and output to text files:

```
prompt% java ConvertIO <inputsec.txt
...
prompt% java ConvertIO <inputsec.txt >result.txt
```

Exercise 1.3 (Time comparison)

Design a program that compares two time durations (in hour/min/sec format) by first converting them into their respective total number of seconds, and then comparing the two delays using the **if** (booleanPredicate) {} **else** {} conditional. This voluntarily anticipates the next chapter. Write another program for comparing the two durations but without converting them into a total number of seconds. Declare two boolean values `Smaller` and `Bigger` both initialized to `false`. Then compare the respective hours, and decide whether we need to assign one of these boolean values to `true` or not. For hours matching, test the minutes,

etc., up to the seconds. Finally, report on the console the outcome based on these boolean values.

Exercise 1.4 (From a linear to a quadratic solver)

We consider a linear equation $y = ax + b$ where a and b are given parameters (using `double` type). Write a program that calculates the root of $ax+b = 0$. That is, the y-intercept of the line with the x-axis. How to extend it to quadratic equations $ax^2 + bx + c = 0$? Observe numerical rounding phenomena that emphasize that roots slightly differ from their mathematical roots because of finite precision of number representations.

Exercise 1.5 (Java syntax for constant declarations)

Explain why the program below does not compile. How should it be modified in order to compile properly?

```
class ConstantExercise
{
static final int MAX = 10;

public static void main(String [] args)
{
double fi;
static final double GoldenRatio=(1.0d+Math.sqrt(5))/2.0;

for(int i=0;i<MAX;i++)
{
    fi=GoldenRatio*Math.cos(2.0*Math.PI*i/(double)MAX);
    System.out.println("i="+i+" f(i)="+fi);
}
}
}
```

2
Conditional Structures and Loops

2.1 Instruction workflow

In this chapter, we start by describing how programmers can control the execution paths of programs using various branching conditionals and looping structures. These branching/looping statements act on blocks of instructions that are sequential sets of instructions. We first explain the single choice `if else` and multiple choice `switch case` branching statements, and then describe the structural syntaxes for repeating sequences of instructions using either the `for`, `while` or `do while` loop statements. We illustrate these key conceptual programming notions using numerous sample programs that emphasize the program workflow outcomes. Throughout this chapter, the leitmotiv is that the execution of a program yields a workflow of instructions. That is, for short:

Program runtime = Instruction workflow

F. Nielsen, *A Concise and Practical Introduction to Programming Algorithms in Java*, Undergraduate Topics in Computer Science, DOI 10.1007/978-1-84882-339-6_2,

2.2 Conditional structures: Simple and multiple choices

2.2.1 Branching conditions: `if ... else ...`

Programs are not always as simple as plain sequences of instructions that are executed step by step, one at a time on the processor. In other words, programs are not usually mere monolithic blocks of instructions, but rather compact structured sets of instruction blocks whose executions are decided on the fly. This gives a program a rich set of different instruction workflows depending on initial conditions.
Programmers often need to check the status of a computed intermediate result to branch the program to such or such another block of instructions to pursue the computation. The elementary branching condition structure in many imperative languages, including Java, is the following `if else` instruction statement:

```
if (booleanExpression)
{BlockA}
else
{BlockB}
```

The boolean expression `booleanExpression` in the `if` structure is first *evaluated* to either `true` or `false`. If the outcome is `true` then `BlockA` is executed, otherwise it is `BlockB` that is selected (boolean expression evaluated to `false`). Blocks of instructions are delimited using braces {...}. Although the curly brackets are *optional* if there is only a single instruction, we recommend you set them in order to improve code readibility. Thus, using a simple `if else` statement, we observe that the same program can have different execution paths depending on its initial conditions. For example, consider the following code that takes two given dates to compare their order. We use the branching condition to display the appropriate console message as follows:

```
int h1 =..., m1 =..., s1 =...; // initial conditions
int h2 =..., m2 =..., s2 =...; // initial conditions
int hs1 = 3600*h1 + 60*m1 + s1;
int hs2 = 3600*h2 + 60*m2 + s2;
int d=hs2-hs1;

if (d>0) System.out.println("larger");
  else
 System.out.println("smaller or identical");
```

Note that there is no `then` keyword in the syntax of Java. Furthermore, the `else` part in the conditional statement is optional:

```
if (booleanExpression)
{BlockA}
```

Conditional structures allow one to perform various status checks on variables to branch to the appropriate subsequent block of instructions. Let us revisit the quadratic equation solver:

Program 2.1 Quadratic equation solver with user input

```
import java.util.*;
class QuadraticEquationRevisited
{
public static void main(String[] arg)
{
Scanner keyboard=new Scanner(System.in);

System.out.print("Enter a,b,c of equation ax^2+bx+c=0:");
double a=keyboard.nextDouble();
double b=keyboard.nextDouble();
double c=keyboard.nextDouble();

double delta=b*b-4.0*a*c;
double root1, root2;

if (delta>=0)
  {
  root1= (-b-Math.sqrt(delta))/(2.0*a);
  root2= (-b+Math.sqrt(delta))/(2.0*a);
  System.out.println("Two real roots:"+root1+" "+root2);
  }
  else
  {System.out.println("No real roots");}
}
}
```

In this example, we asserted that the computations of the roots `root1` and `root2` are possible using the fact that the discriminant `delta>=0` in the block of instructions executed when expression `delta>=0` is `true`. Running this program twice with respective user keyboard input `1 2 3` and `-1 2 3` yields the following session:

```
Enter a,b,c of equation ax^2+bx+c=0:1 2 3
No real roots
Enter a,b,c of equation ax^2+bx+c=0:-1 2 3
Two real roots:3.0 -1.0
```

In the `if else` conditionals, the boolean expressions used to select the appropriate branchings are also called *boolean predicates*.

2.2.2 Ternary operator for branching instructions: `Predicate ? A : B`

In Java, there is also a special compact form of the `if else` conditional used for variable assignments called a ternary operator. This ternary operator `Predicate ? A : B` provided for branching assignments is illustrated in the sample code below:

```
double x1=Math.PI; // constants defined in the Math class
double x2=Math.E;
double min=(x1>x2)? x2 : x1; // min value
double diff= (x1>x2)? x1-x2 : x2-x1; // absolute value
System.out.println(min+" difference with max="+diff);
```

Executing this code, we get:

```
2.718281828459045 difference with max=0.423310825130748
```

The compact instruction

```
double min=(x1>x2)? x2 : x1;
```

...is therefore equivalent to:

```
double min;
if (x1>x2) min=x2;
else min=x1;
```

Figure 2.1 depicts the schema for unary, binary and ternary operators.

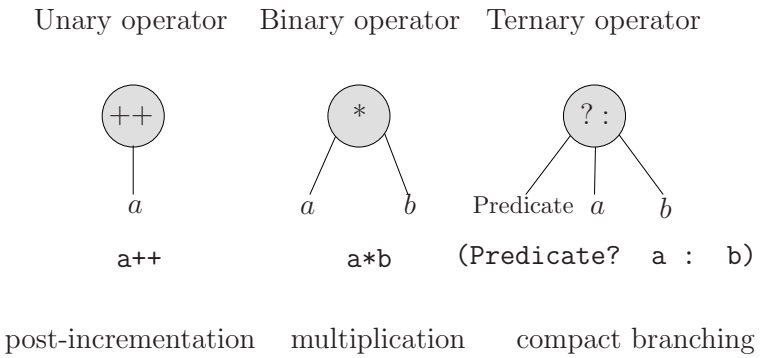


Figure 2.1 Visualizing unary, binary and ternary operators

2.2.3 Nested conditionals

Conditional structures may also be nested yielding various complex program workflows. For example, we may further enhance the output message of our former date comparison as follows:

```
int h1 =..., m1 =..., s1 =...;
int h2 =..., m2 =..., s2 =...;
int hs1 = 3600*h1 + 60*m1 + s1;
int hs2 = 3600*h2 + 60*m2 + s2;
int d=hs2-hs1;

if (d>0) {System.out.println("larger");}
  else
    {if (d<0)
      {System.out.println("smaller");}
      else
        {System.out.println("identical");}
    }
```

Since these branching statements are all single instruction blocks, we can also choose to remove the braces as follows:

```
if (d>0) System.out.println("larger");
else
if (d<0)
  System.out.println("smaller");
  else
  System.out.println("identical");
```

However, we do not recommend it as it is a main source of errors to novice programmers. Note that in Java there is *no* shortcut[1] for `else if`. In Java, we need to write plainly `else if`. There can be any arbitrary *level* of nested `if else` conditional statements, as shown in the generic form below:

```
if (predicate1)
    {Block1}
    else
    {
      if (predicate2)
        {Block2}
        else
        {
        if (predicate3)
          {Block3}
          else
            {
              ...
            }
        }
```

[1] In some languages such as Maple®, there exists a dedicated keyword like `elif`.

```
    }
```

In general, we advise to always take care with boolean predicates that use the equality tests `==` since there can be numerical round-off errors. Indeed, remember that machine computations on reals are done using single or double precision, and thus the result may be truncated to fit the formatting of numbers. Consider for example the following tiny example that illustrates numerical imprecisions of programs:

```
class RoundOff
{
public static void main(String[] arg)
{
double a=1.0d;
double b=3.14d;
double c=a+b;

if (c==4.14) // Equality tests are dangerous!
  {
  System.out.println("Correct");
  }
  else
  {
  System.out.println("Incorrect. I branched on the wrong
      block!!!");
  System.out.println("a="+a+" b="+b+" a+b=c="+c);
  // unexpected behavior may follow...
  }
}
}
```

Running this program, we get the surprising result caused by numerical precision problems:

```
Incorrect. I branched on the wrong block!!!
a=1.0 b=3.14 a+b=c=4.140000000000001
```

This clearly demonstrates that equality tests `==` in predicates may be harmful.

2.2.4 Relational and logical operators for comparisons

The *relational operators* (also called comparison operators) that evaluate to either `true` or `false` are the following ones:

<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	equal to
!=	not equal to

One of the most frequent programming errors is to use the equality symbol = instead of the relational operator == to test for equalities:

```
int x=0;

if (x=1) System.out.println("x equals 1");
else  System.out.println("x is different from  1");
```

Fortunately, the compiler generates an error message since in that case types `boolean/int` are incompatible when performing type checking in the expression `x=1`. But beware that this will not be detected by the compiler in the case of boolean variables:

```
boolean x=false;
if (x=true) System.out.println("x is true");
else  System.out.println("x is false");
```

In that case the predicate `x=true` contains an assignment `x=true` and the evaluated value of the "expression" yields `true`, so that the program branches on the instruction `System.out.println("x is true");`.
A boolean predicate may consist of several relation operators connected using *logical operators* from the table:

&&	and
||	or
!	not

The logic truth tables of these connectors are given as:

&&	true	false
true	true	false
false	false	false

||	true	false
true	true	true
false	true	false

Whenever logical operators are used in predicates, the boolean expressions are evaluated in a lazy fashion as follows:

– For the `&&` connector, in `Expr1 && Expr2` do not evaluate `Expr2` if `Expr1` is evaluated to `false` since we alreaady know that in that case that `Expr1 && Expr2 = false` whatever the `true/false` outcome value of expression `Expr2`.

– Similarly, for the `||` connector, in `Expr1 || Expr2` do not evaluate `Expr2` if `Expr1` is evaluated to `true` since we already know that in that case `Expr1 || Expr2 = true` whatever the `true/false` value of expression `Expr2`.

The lazy evaluation of boolean predicates will become helpful when manipulating arrays later on. For now, let us illustrate these notions with a simple example:

Program 2.2 Lazy evaluation of boolean predicates

```
class LazyEvaluation{
  public static void main (String[] args)
  {
    double x=3.14, y=0.0;
    boolean test1, test2;


    // Here division by zero yields a problem
    // But this is prevented in the && by first checking
        whether the denominator is
    // zero or not
    if ((y!=0.0) && (x/y>2.0))
          {// Do nothing
           ; }
          else
      {// Block
      test1=(y!=0.0);
      test2=(x/y>2.0);

      System.out.println("Test1:"+test1+" Test2:"+test2);
      System.out.println("We did not evaluate x/y that is
          equal to "+(x/y));
        }


    // Here, again we do not compute x/y since the first term
         is true
    if ((y==0.0) || (x/y>2.0))
      {// Block
      System.out.println("Actually, again, we did not
          evaluate x/y that is equal to "+(x/y));
        }
  }
}
```

Running this program, we get the following console output:

```
Test1:false Test2:true
We did not evaluate x/y that is equal to Infinity
Actually, again, we did not evaluate x/y that is equal to Infinity
```

2.2.5 Multiple choices: `switch case`

The nested `if else` conditional instructions presented in § 2.2.3 are somehow difficult to use in case one would like to check that a given variable is equal to such or such a value. Indeed, nested blocks of instructions are difficult to properly visualize on the screen. In the case of `multiple choices`, it is better to use the `switch case` structure that branches on the appropriate set of instructions depending on the value of a given expression. For example, consider the code:

```
class ProgSwitch
{public static void main(String arg[]){
System.out.print("Input a digit in [0..9]:");
Scanner keyboard=new Scanner(System.in);
int n=keyboard.nextInt();
switch(n)
{
case 0: System.out.println("zero"); break;
case 1: System.out.println("one"); break;
case 2: System.out.println("two"); break;
case 3: System.out.println("three"); break;
default: System.out.println("Above three!");
break;
}}}
```

The conditional statement `switch` consider the elementary expression n of type `int` and compare it successively with the first case: `case 0`. This means that **if** (n==0)Block1 **else** \{ ... \}. The set of instructions in a `case` should end with the keyword `break`. Note that there is also the `default` case that contains the set of instructions to execute when none of the former cases were met. The formal syntax of the multiple choice `switch case` is thus:

```
switch(TypedExpression)
{
case C1:
        SetOfInstructions1;
        break;
case C2:
        SetOfInstructions2;
        break;
...
case Cn:
        SetOfInstructionsn;
        break;
default:
        SetOfDefaultInstructions;
}
}
```

Multiple choice `switch` conditionals are often used by programmers for displaying messages[2]:

Program 2.3 Demonstration of the `switch case` statement

```
int dd=3; // 0 for Monday, 6 for Sunday

switch(dd)
{
case 0:
  System.out.println("Monday"); break;
case 1:
  System.out.println("Tuesday"); break;
case 2:
  System.out.println("Wednesday"); break;
case 3:
  System.out.println("Thursday"); break;
case 4:
  System.out.println("Friday"); break;
case 5:
  System.out.println("Saturday"); break;
case 6:
  System.out.println("Sunday"); break;
default:
  System.out.println("Out of scope!");

}
```

2.3 Blocks and scopes of variables

2.3.1 Blocks of instructions

A block of instructions is a set of instructions that is executed sequentially. Blocks of instructions are delimited by braces, as shown below:

```
{
// This is a block
// (There are no control structures inside it)
Instruction1;
Instruction2;
...
}
```

A block is semantically interpreted as an atomic instruction at a macroscopic level when parsing.

[2] Or used for translating one type to another when used in functions

2.3.2 Nested blocks and variable scopes

Blocks can be nested. This naturally occurs in the case of `if-else` structures that may internally contain other conditional structures. But this may also be possible without conditional structures for controlling the scope of variables. Indeed, variables defined in a block are defined for all its sub-blocks. Thus a variable cannot be redefined in a sub-block. Moreover variables defined in sub-blocks cannot be accessed by parent blocks as illustrated by the following example:

```
class NestedBlock
{
public static void main(String[] arg)
{
int i=3;
int j=4;

System.out.println("i="+i+" j="+j);

  {
  // Cannot redefine a variable i here
  int ii=5;
  j++;
  i--;
  }

System.out.println("i="+i+" j="+j);
// Cannot access variable ii here
}
}
```

```
i=3 j=4
i=2 j=5
```

Finally note that single instructions in control structures such as `if-else` are interpreted as implicit blocks where braces are omitted for code readibility.

2.4 Looping structures

Loop statements are fundamental structures for iterating a given sequence of instructions, repeating a block of instructions. Java provides three kinds of constructions for ease of programming, namely: `while`, `for` and `do-while`. Theoretically speaking, these three different constructions can all be emulated with a `while` statement. We describe the semantic of each structure by illustrating it with concrete examples.

2.4.1 Loop statement: `while`

The syntax of a `while` loop statement is as follows:

```
while (boolean_expression)
{block_instruction;}
```

This means that while the `boolean_expression` is evaluated to `true`, the sequence of instructions contained in the `block_instruction` is executed.
Consider calculating the *greatest common divisor* (gcd for short) of two integers a and b. That is, the largest common divisor c such that both a and b can be divided by c. For example, the GCD of $a = 30$ and $b = 105$ is 15 since $a = 2\times3\times5$ and $b = 5\times3\times7$. Euclid came up with a simple algorithm published for solving this task. The algorithm was reported in his books *Elements* around[3] 300 BC. Computing the GCD is an essential number operation that requires quite a large amount of computation for large numbers. The GCD problem is related to many hot topics of cryptographic systems nowadays. Euclid's algorithm is quite simple: If $a = b$ then clearly the GCD of a and b is $c = a = b$. Otherwise, consider the largest integer, say a without loss of generality, and observe the important property that

$$\mathrm{GCD}(a, b) = \mathrm{GCD}(a - b, b).$$

Therefore, applying this equality reduces the total sum $a + b$, and at some point, after k iterations, we will necessarily have $a_k = b_k$. Let us prove a stronger result: $\mathrm{GCD}(a, b) = \mathrm{GCD}(b, a \mod b)$.

Proof

To see this, let us write $a = qb + r$ where q is the quotient of the Euclidean division and r its reminder. Any common divisor c of a and b is also a common divisor of r: Indeed, suppose we have $a = ca'$ and $b = cb'$, then $r = a - qb = (a' - qb')c$. Since all these numbers are integers, this implies that r is divisible by c. It follows that the greatest common divisor g or a and b is also the greatest common divisor of b and r. □

Let us implement this routine using the `while` loop statement. The terminating state is when $a = b$. Therefore, while $a \neq b$ (written in Java as `a!=b`), we retrieve that smaller number to the larger number using a `if` conditional structure. This gives the following source code:

[3] It is alleged that the algorithm was likely already known in 500 BC.

Program 2.4 Euclid's Greatest Common Divisor (GCD)

```
class GCD {
public static void main(String[] arg)
{
int a;
int b;
while (a!=b)
  {
  if (a>b) a=a-b;
    else b=b-a;
  }
System.out.println(a); // or b since a=b
}
}
```

Running this program for $a = 30$ and $b = 105$ yields $\mathrm{GCD}(a, b) = 15$.
Euclid's algorithm has a nice *geometric* interpretation: Consider an initial rectangle of width a and height b. Bisect this rectangle as follows: Choose the smallest side, and remove a square of that side from the current rectangle. Repeat this process until we get a square: The side length of that square is the GCD of the initial numbers. Figure 2.2 depicts this "squaring" process.

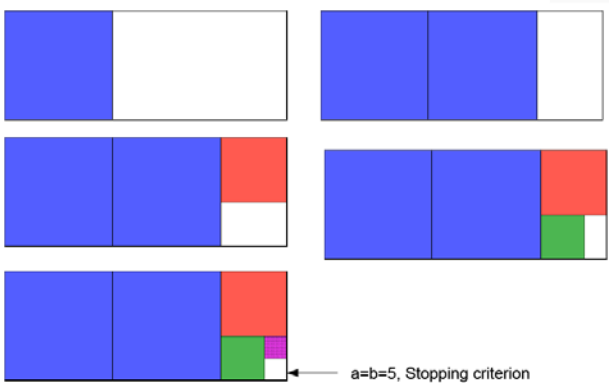


Figure 2.2 A geometric interpretation of Euclid's algorithm. Here, illustrated for $a = 65$ and $b = 25$ ($\mathrm{GCD}(a, b) = 5$)

2.4.2 Loop statement: `do-while`

Java provides another slightly different syntax for making iterations: the `do` loop statement. The difference with a `while` statement is that we execute at least once the sequence of instructions in a `do` statement, whereas this might not be the case of a `while` statement, depending on the evaluation of the boolean predicate. The syntax of a `do` structure is as follows:

```
do
{block_instruction;}
while (boolean_expression);
```

That is, the `boolean_expression` is evaluated after the block of instructions, and not prior to its execution, as it is the case for `while` structures. Consider computing the square root $\sqrt{a}$ of a non-negative number a using Newton's method. Newton's method finds the closest root to a given initial condition x_0 of a smooth function by iterating the following process: Evaluate the tangent equation of the function at x_0, and intersect this tangent line with the x-axis: This gives the new value x_1. Repeat this process until the difference between two successive steps go beyond a prescribed threshold (or alternatively, repeat k times this process). For a given value x_n, we thus find the next value x_{n+1} by setting the y-ordinate of the tangent line at $(x_n, f(x_n))$ to 0:

$$y = f'(x_n)(x - x_n) + f(x_n) = 0.$$

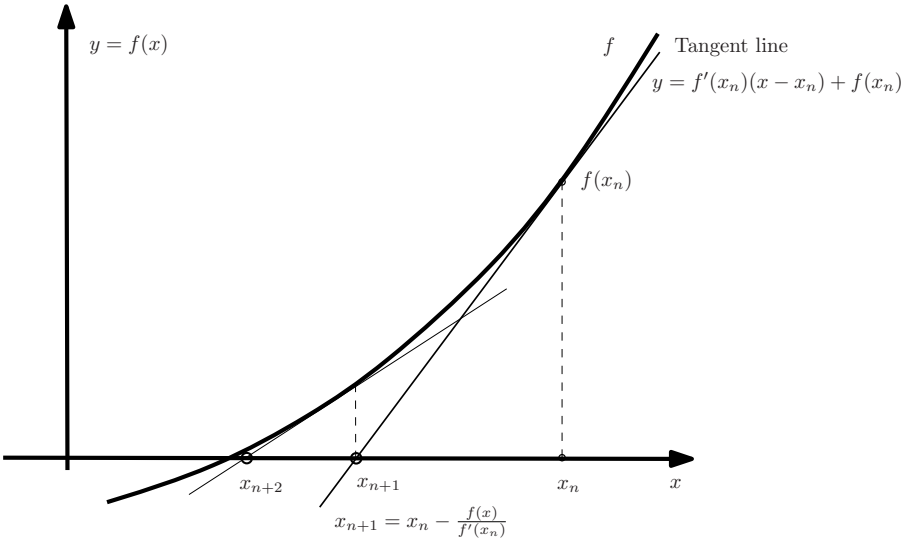


Figure 2.3 Newton's method for finding the root of a function

It follows that we get:

$$x_{n+1} = x_n - \frac{f(x)}{f'(x_n)}.$$

Figure 2.3 illustrates this root finding process. Let us use Newton's method to calculate the square root function of a by finding the root of equation $f(x) = x^2 - a$. We implement the loop using a `do` structure as follows:

Program 2.5 Newton's approximation algorithm

```
double a = 2.0, x, xold;
x = a;
do{
xold = x;
// compute one iteration
x = (xold+a/xold)/2.0;
System.out.println(x);
} while(Math.abs(x-xold) > 1e-10);
```

```
1.5
1.4166666666666665
1.4142156862745097
1.4142135623746899
1.4142135623730949
```

Netwon's method is provably converging very fast under mild assumptions.

2.4.3 Loop statement: `for`

Often programmers need to repeat a sequence of instructions by changing some variables by a given increment step. Although this can be done using the former `while`/`do` structures, Java provides a more convenient structure: the `for` loop. The generic syntax of a `for` structure is as follows:

```
for(initialCondition; booleanPredicate; update)
{
block_instructions;
}
```

For example, consider computing the cumulative sum S_n of the first n integers:

$$S_n = \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}.$$

We have the recurrence equation: $S_n = n - 1 + S_{n-1}$ with $S_0 = 0$. Therefore to compute this cumulative sum, we start in reverse from S_0 and get S_i by adding $i-1$ to S_{i-1} for all $i \in \{1, ..., n\}$. Let us use the `for` structure as follows:

Program 2.6 Cumulative sum

```
class ForLoop
{
public static void main(String args[])
{
int i, n=10;
int cumulLoop=0;
for(i=0;i<n;i++) {cumulLoop+=i;}
```

```
int cumul=(n*(n-1))/2; // closed-form solution
System.out.println(cumulLoop+" closed-form:"+cumul);
}
}
```

To give yet another usage of the `for` loop, consider computing an approximation of π by a Monte-Carlo simulation. We draw uniformly random points in a unit square, and count the number of points falling inside the unit disk centered inside the square. The ratio of the points falling inside this square to the overall number of points gives a good approximation of $\frac{\pi}{4}$. We draw a uniform random number in $[0, 1)$ in Java using the function `random()` of the `Math` class. This kind of Monte-Carlo simulation is extremely useful to compute complex integrals, and is often used in the finance industry. Let us give the code for approaching π:

Program 2.7 Approaching π by Monte-Carlo simulation

```
class MonteCarloPI
{
public static void main(String [] args)
{
int iter = 10000000; // # iterations
int hits = 0;
for (int i = 0; i < iter; i++)
{
double rX = 2*Math.random() - 1.0;
double rY = 2*Math.random() - 1.0;
double dist = rX*rX + rY*rY;
if (dist <= 1.0) // falls inside the disk
hits++;
}
double ratio = (double)hits/iter; // Ratio of areas
double area = ratio * 4.0;
System.out.println("Estimation of PI: " + area+ " versus
    library PI "+Math.PI);
}
}
```

Unfortunately running this code gives a poor approximation of π since we get only a few correct digits, even after drawing 10^7 points.

```
Estimation of PI: 3.1413544 versus library PI 3.141592653589793
```

2.4.4 Boolean arithmetic expressions

A category of arithmetic expressions that are especially useful for writing predicates in loop structures are boolean expressions. Although they are not

usually used in plain assignments, they also make perfect sense as illustrated by the program below:

Program 2.8 Boolean arithmetic expression

```
class Boolean{
public static void main(String[] args)
{
boolean b1 = (6-2) == 4;
boolean b2 = 22/7 == 3+1/7.0 ;
boolean b3 = 22/7 == 3+ 1/7;
System.out.println(b1); // true
System.out.println(b2); // false
System.out.println(b3); // true
}
}
```

2.5 Unfolding loops and program termination

2.5.1 Unfolding loops

When executing a program that contains loop structures, we can unroll these loops manually. Compilers actually do it sometimes to optimize the generated bytecode.

2.5.2 Never ending programs

Once programmers first experience loops, a major issue arises: Does the program terminate? It is indeed quite easy to write never ending programs by writing loops that execute forever as illustrated below:

```
int i=0;
while (true)
i++;
```

Always make sure when you write a `for` structure that the boolean expression will evaluate to `false` at some stage. Take care to avoid mistyping problems such as:

```
for(i=0;i>=0;i++)
; // common mistyping error in the boolean predicate
```

... and prefer to use curly brackets instead of the semi-colon for single-instruction blocks:

```
for(i=0;i>=0;i++)
{ }
```

2.5.3 Loop equivalence to universal `while` structures

As mentioned earlier, the three loop structures in Java are all equivalent to a universal `while` structure. These different loop syntaxes are provided to help programmers quickly write code.

```
for(instructionInit; booleanCondition; instructionUpdate) block_instruction;
```

```
instructionInit;
while (booleanCondition)
block_instruction3; instructionUpdate;
```

2.5.4 Breaking loops at any time with `break`

We can voluntarily escape loops at *any time* by using the keyword `break`. This special instruction is useful for example when we ask users to input any given number of data.

2.5.5 Loops and program termination

Consider the following sequence $\{u_i\}_i$ of integers numbers as follows:

$$u_{n+1} = \begin{cases} u_n/2 & \text{if } n \text{ is even,} \\ 3u_n + 1 & \text{otherwise.} \end{cases},$$

initialized for any given $u_0 \in \mathbb{N}$.
For example, let $u_0 = 14$. Then $u_1 = 14$, $u_2 = 7$, $u_3 = 22$, $u_4 = 11$, $u_5 = 34$, $u_6 = 17$, $u_7 = 52$, $u_8 = 26$, $u_9 = 13$, $u_{10} = 40$, $u_{11} = 20$, $u_{12} = 10$, $u_{13} = 5$, $u_{14} = 16$, $u_{15} = 8$, $u_{16} = 4$, $u_{17} = 2$, $u_{18} = 1$. Once 1 is reached the sequence cycles $1, 4, 2, 1, 4, 2 \ldots$. It is conjectured but not yet proved that for any $u_0 \geq 1$ the sequence reaches in a finite step number 1. We can numerically check that this conjecture holds for a given number $u_0 = n$ using the following `do` loop structure:

Program 2.9 Syracuse's conjecture

```
do{
if ((n%2)==0)
```

```
n/=2;// divide by 2
else
n=3*n+1;
} while (n>1);
```

However one has not yet managed to successfully prove that this program will eventually stop for *any* given n. It is a hard mathematical problem that is known in the literature by the name of Syracuse's conjecture or $3x + 1$ problem.[4]
This simple toy problem raises the fundamental *halting problem* famous in theoretical computer science. Loosely speaking, Gödel proved that there is *no* program that can decide whether any given program will stop after a finite number of instructions or not. This important theoretical result, which is one of the pillars of computer science, will be further explained using a simple contradiction argument in Chapter 3.8.

2.6 Certifying programs: Syntax, compilation and numerical bugs

A program that compiles without reporting *any* error message is a *syntactically* correct program. Beware that because of the language flexibility provided by its high-level semantic, some obscure codes[5] compile. These obscure codes are often very difficult for humans to understand. To get a flavor, consider for example the snippet:

Program 2.10 Syntactically correct program

```
int i=3;
// syntax below is valid! guess its result?
int var=i+++i;
```

This program compiles and is valid since the expression `i+++i` is well-formed. How did the compiler interpret it? Well, first the compiler put parenthesis from the operator priority rule: `(i++)+i`. Then it first evaluated this expression by performing the post-incrementation `i++` (so that it returns 3 for this expression but now i stores value 4). Finally, it adds to the value of `i` 3 so that we get $3 + 4 = 7$.
Even when a simple human-readable program compiles, it becomes complex for humans to check whether the input fits all branching conditions. In other words, are all input cases considered so that the program does not have to

[4] See `http://arxiv.org/abs/math/0608208/` for some annotated bibliographic notes.
[5] Hackers love them.

process "unexpected" data? This can turn out to be very difficult to assert for moderate-size programs. For example, consider the quadratic equation solver:

Program 2.11 Quadratic equation solver

```
import java.util.*;
class QuadraticEquationScanner
{
public static void main(String[] arg)
{
double a,b,c; // choose a=1, b=1, c=1
Scanner input=new Scanner(System.in); input.useLocale(Locale.
    US);
a=input.nextDouble();
b=input.nextDouble();
c=input.nextDouble();
double delta=b*b-4.0*a*c;
double root1, root2;
// BEWARE: potentially Not a Number (NaN) for neg.
    discriminant!
root1= (-b-Math.sqrt(delta))/(2.0*a);
root2= (-b+Math.sqrt(delta))/(2.0*a);
System.out.println("root1="+root1+" root2="+root2);
}
}
```

The problem with that program is that we may compute roots of negative numbers. Although mathematically this makes sense with imaginary numbers $\mathbb{C}$, this is not the case for the function `Math.sqrt()`. The function returns a special number called `NaN` (standing for Not a Number) so that the two roots may be equal to `NaN`. It is much better to avoid that case by ensuring with a condition that `delta` is greater or equal to zero:

```
if (delta>=0.0d)
  {
  root1= (-b-Math.sqrt(delta))/(2.0*a);
  root2= (-b+Math.sqrt(delta))/(2.0*a);
  System.out.println("root1="+root1+" root2="+root2);
  }
  else
  {System.out.println("Imaginary roots!");}
```

The rule of thumb is to write easy-to-read code and adopt conventions once and for all. For example, always put a semi-colon at the end of instructions, even if it is not required (atomic blocks). Always indent the source code to better visualize nested structures with braces {}. Take particular care of equality test `==` with assignment equal symbol `=` (type checking helps find some anomalous situations but not all of them).

Finally, let us insist that even if we considered all possible input cases and wrote our codes keeping in mind that they must also be human-readable, it

is impossible for us to consider all numerical imprecisions that can occur.[6] Consider the following example:

Program 2.12 A simple numerical bug

```
// Constant
final double PI = 3.14;
int a=1;
double b=a+PI;
if (b==4.14) // Equality test are dangerous!!!
  System.out.println("Correct result");
else
{System.out.println("Incorrect result");
System.out.println("a="+a+" b="+b+" PI="+PI);
}
```

This code is dangerous because, mathematically speaking, it is obvious that $a+b = 4.14$ but because of the finite representation of numbers in machine (and their various formatting), this simple addition yields an approximate result. In practice, the first lesson we learn is that we always need to very cautiously use equality tests on reals. The second lesson is that proofs of programs should be fully automated. This is a very active domain of theoretical computer science that will bring novel solutions in the 21st century.

2.7 Parsing program arguments from the command line

So far we have initialized programs either by interactively asking users to enter initial values at the console, or by plugging these initial values directly into the source code. The former approach means that we have high-latency programs since user input is "slow." The latter means that programs lack flexibility since we need to recompile the code every time we would like to test other initial parameter conditions.
Fortunately, programs in Java can be executed with *arguments* given in the command line. These arguments are stored in the array `arg` of the `main` function:

```
public static void main (String[] args)
```

These arguments are stored as strings `args[0]`, `args[1]`, etc. Thus even if we enter numbers like "120" and "28" in the command line:

[6] Some software packages such as Astrée used in the airplane industry do that automatically to certify code robustness. See `http://www.astree.ens.fr/`

```
prompt gcd 120 28
```

These numbers are in fact plain sequences of characters that are stored in Java strings. Thus the program needs at first to reinterpret these strings into appropriate numbers (integers or reals), prior to assigning them to variables. To parse a string and get its equivalent integer (`int`), one uses `Integer.parseInt(stringname);`. For reals, to parse and create the corresponding `float` or `double` from a given string `str`, use the following functions: `Float.parseFloat(str)` or `Double.parseDouble(str)`. Let us revisit Euclid's GCD program by taking the two numbers a and b from the program arguments:

```
class gcd {
public static void main(String[] arg)
{
// Parse arguments into integer parameters
int a= Integer.parseInt(arg[0]);
int b= Integer.parseInt(arg[1]);
System.out.println("Computing GCD("+a+","+b+")");

while (a!=b)
{
if (a>b) a=a-b;
else b=b-a;
}
// Display to console
System.out.println("Greatest common divisor is "+a);
}
}
```

Compiling and running this program yields:

```
prompt%java gcd 234652 3456222
Computing GCD(234652,3456222)
Greatest common divisor is 22
```

But there is more. In Chapter 1.8, we explained the basic mechanism of input/output redirections (I/O redirections). Using I/O redirections with program arguments yields an efficient framework for executing and testing programs. Let us export the result to a text file named `result.txt`:

```
prompt%java gcd 234652 3456222 >result.txt
```

Then we saved the texts previously written on the console to that file. We can visualize its contents as follows:

```
prompt%more result.txt
Computing GCD(234652,3456222)
Greatest common divisor is 22
```

We are now ready to proceed to the next chapter concentrating on functions and procedures.

2.8 Exercises

Exercise 2.1 (Integer parity)

Write a program that interactively asks for an integer at the console and reports its odd/even parity. Modify the code so that the program first asks the user how many times it would like to perform parity computations, and then iteratively asks for a number, compute its parity, and repeat until it has performed the required number of parity rounds. Further modify this program so that now both input and output are redirected into text files, say `input.txt` and `output.txt`.

Exercise 2.2 (Leap year)

A leap year is a year with 366 days that has a 29th February in its calendar. Years whose division by 4 equals an integer are leap years except for years that are evenly divisible by 100 unless they are also evenly divisible by 400. Write a program that asks for a year and report on whether it is a leap year or not. Modify this code so that the program keeps asking for years, and compute its leap year property until the user input -1.

Exercise 2.3 (Displaying triangles)

Write a program that asks for an integer `n`, and write on the output a triangle as illustrated for the following example (with $n = 5$):

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
1 2 3 4
1 2 3
1 2
1
```

Exercise 2.4 (Approximating a function minimum)

Let $f(x) = \sin(\frac{25}{x^2-4x+6}) - \frac{x}{3}$ for $x \in [0, \pi]$ be a given function. Write a program that takes as argument an integer `n` and returns the minimum of $f(x)$ on the range $x \in [0, \pi]$ evenly sampled by steps $\frac{1}{n}$. Execute the program for $n = 10$, $n = 100$ and $n = 100000$ and check that the root is about -1.84318.

Exercise 2.5 (Computing $\sqrt{a}$ numerically)

Consider Newton's root finding method to compute $\frac{1}{\sqrt{a}}$ by choosing function $f(x) = a - \frac{1}{x^2}$. Show that we obtain the following sequence:

$x_{n+1} = \frac{x_n}{2}(3 - ax_n^2$. Write the corresponding program. Does it converge faster or slower than Newton's method for $f(x) = a - x^2$?

Exercise 2.6 (Variable scope)

Consider the following program:

```
class ScopeExercise
{
public static void main(String[] a)
{
int j=5;
for(int i=0;i<10;i++)
  System.out.println("i="+i);
j+=i+10;
System.out.println("j="+j);
}
}
```

Explain what is wrong with this program. How do we change the scope of variable `i` in order to compile?

*Exercise 2.7 (Chevalier DeMere and the birth of probability **)*

In the 17th century, gambler Chevalier De Méré asked the following question of Blaire Pascal and Pierre de Fermat: How can one compare the following probabilities

– Getting at least one ace in four rolls of a dice,

– Getting at least one double ace using twenty-four rolls of two dices.

Chevalier De Méré thought that the second chance game was better but lost constantly. Using the function `Math.random()` and loop statements, experiment with the chance of winning for each game. After running many trials (say, a million of them), observe that the empirical probability of winning with the first game is higher. Prove that the probability of winning for the first and second games are respectively $\left(\frac{5}{6}\right)^4$ and $\left(\frac{35}{36}\right)^{24}$.

*Exercise 2.8 (Saint Petersburg paradox **)*

The following game of chance was introduced by Nicolas Bernoulli: A gamer pays a fixed fee to play, and then a fair coin is tossed repeatedly until, say, a tail first appears. This ends the game. The pot starts at 1 euro and is doubled every time a head appears. The gamer wins whatever is in the pot after the game ends. Show that you win 2^{k-1} euros if the coin is tossed k times until the first tail appears. The paradox is that whatever the initial fee, it is worth playing this game. Indeed, prove that the expected gain is $\sum_{k=1}^{\infty} \frac{1}{2^k} 2^{k-1} = \sum_{k=1}^{\infty} \frac{1}{2} = \infty$. Write a program

that simulates this game, and try various initial fees and number of rounds to see whether you are winning or not. (Note that this paradox is mathematically explained by introducing an expected utility theory.)

3
Functions and Recursive Functions

3.1 Advantages of programming functions

The concept of functions in programming languages is quite different from the usual mathematical notions, although it bears some similarity. Therefore it can be confusing at first sight to make an explicit comparison. We rather prefer to introduce the syntax of functions in Java, and show its two essential merits:

– Functions as *subroutines* for enhancing program *modularity* and code re-use,

– Functions defined *recursively* by themselves for novel computation paradigms.

Last but not least, by introducing functions, we will explain the difference between local (say, usual block variables) and global memory variables (say, static class variables). Java passes arguments in functions by value only. This is a major fundamental difference with C++ that allows both by-value and by-reference variable passing. Finally, describing recursion will help us explain the *function call stack* of Java where local variables are temporarily allocated.

F. Nielsen, *A Concise and Practical Introduction to Programming Algorithms in Java*, Undergraduate Topics in Computer Science, DOI 10.1007/978-1-84882-339-6_3,

3.2 Declaring and calling functions

3.2.1 Prototyping functions

Functions should always be declared inside the body of a class: The program class (for now). The generic syntax for declaring a *function* `F` that takes N arguments `arg1`, ..., `argN` of respective types `Type1`, ...,`TypeN` and return a result of type `TypeR` is:

```
static TypeR F(Type1 arg1, Type2 arg2, ..., TypeN argN)
{
TypeR result;
block_of_instructions;
return result;
}
```

Procedures are special functions that do not return any result. In Java, this is specified by using the `void` keyword for the function return type. Thus the declaration of a *procedure* is as follows:

```
static void Proc(Type1 arg1, Type2 arg2, ..., TypeM argM)
{
block_of_instructions;
return;
}
```

The last instruction statement `return;` in the procedure `Proc` may be omitted. We always put the keyword `static` in front of procedure/function declarations: `static void Proc`. We will explain why this is necessary in Chapter 5 dealing with objects and methods. Since procedures and functions shall be attached to the body of class (that is, encapsulated into the class), we have basically the following program skeleton:

Program 3.1 Basic program skeleton for defining static functions

```
class ProgramSkeleton{

static  TypeF F(Type1 arg1, ..., TypeN argN)
          {
          TypeF result;
          // Description
          Block-of-instructions;
          return result;
          }

static void Proc(Type1 arg1, Type2 arg2, ..., TypeM argM)
        {
        block_instructions;
        return;
        }
```

```
public static void main(String[] arguments)
{...}
}
```

Let us see some concrete examples.

3.2.2 Examples of basic functions

Let us create a demonstration program class `FunctionDeclaration` by typing in the `FunctionDeclaration.java` text file the following code:

Program 3.2 A basic demonstration class for defining and calling static functions

```
class FunctionDeclaration{

   static int square(int x)
            {return x*x;}

   static boolean isOdd(int p)
            {if ((p%2)==0) return false;
                      else return true;}

   static double distance(double x, double y)
            {if (x>y) return x-y;
                 else return y-x;}

   static void display(double x, double y)
              {System.out.println("("+x+","+y+")");
               return; // return void
              }

  public static void main (String[] args)
  {
  display(3,2);
  display(square(2),distance(5,9));

  int p=123124345;
  if (isOdd(p))
      System.out.println("p is odd");
        else  System.out.println("p is even");
  }
}
```

Compiling and executing this code, we get:

```
(3.0,2.0)
(4.0,4.0)
p is odd
```

As shown in this example, the functions and procedures are defined in the *scope* of class `FunctionDeclaration`. These functions are called in the `main` function of class `FunctionDeclaration` but this is not a restriction: Functions can also be called inside the body of *other* functions. Observe that arguments are *expressions* that may contain function calls themselves too, as in the following example: `display(square(2),distance(5,9));`. This first warm-up example shows that functions/procedures help in writing code subroutines and thus provide the basics for code modularity and code re-use. Code re-use facilitates certification and correctness of programs by breaking them down into elementary units that can be proved more easily in turn.
Since functions are encapsulated into classes, we can call any arbitrary function `F` declared in another class, say `OtherClass`, by using the syntax `OtherClass.F`. We already used this syntax when calling mathematical functions such as `Math.cos(x);` that are encapsulated into the class `Math`. Class `Math` is part of a huge set of standardized Java application programming interfaces (APIs) that come installed with the Java development kit (JDK). We can omit the class name when calling a function provided that the function is declared inside the body of the *same* class. This is the case of all functions of our program `FunctionDeclaration`. For example, `display(3,2);` is equivalent to the function call `FunctionDeclaration.display(3,2);`. Observe that the `main` function of the program class is the *default procedure* called upon when executing the program.

3.2.3 A more elaborate example: The iterative factorial function

Let us now look at how implementing the factorial function

$$n! = 1 \times 2 \times ... \times n = \prod_{i=1}^{n} i$$

for any given $n \in \mathbb{N}^+$ (by convention $0! = 1$)). Since the factorial function does not belong to the basic static functions of the `Math` class, let us write our own function `factorial` in a toolbox class named `Toolbox`. We use a `while` statement for accumulating the multiplications $n \times ... \times 1$ as follows:

Program 3.3 Implementing the factorial function $n!$

```
class Toolbox{

  static int factorial(int n)
  {int result=1;
```

```
  while(n>0){
    result*=n; // similar to result=result*n;
    n--;
  }
    return result; /
    }
}

class ExampleFactorial{

  public static void main(String[] args)
  {
  System.out.println("6!="+Toolbox.factorial(6));
  }
}
```

Compiling and executing this program yields `6!=720`.

3.2.4 Functions with conditional statements

The former `distance` function of § 3.2.2 takes two *arguments* `x` and `y` of type `double` and returns the absolute value of their difference in a result of type `double`. Let us recall the previous code...

```
static double distance(double x, double y)
{double result;
if (x>y) result=x-y;
 else result=y-x;
return result;
 }
```

This could have been rewritten more compactly as follows:

```
 static double distance(double x, double y)
{if (x>y) return x-y;
 else return y-x;}
```

In case functions or procedures use branching conditionals (such as `if else` or `switch case` statements), we always have to make sure that whatever the instruction workflow, the function will always reach an appropriate `return` statement. The compiler checks all these different execution paths and may complain with a `missing return statement` message error if this property is not met. For example, try to replace the `distance` function by this erroneous code:

```
static double distance(double x, double y)
{double result;
if (x>y) result=x-y; // forgot voluntarily the return statement
 else return y-x;}
```

As an other example, consider the following code snippet, which uses a `switch case` statement:

Program 3.4 Function with branching structures

```
class FunctionWithConditionalStatement{
  public static void main (String[] arguments)
  {
    double x=Math.E;

    System.out.println("Choose function to evalute for x="+x)
        ;
    System.out.print("(1) Identity, (2) Logarithm, (3) Sinus.
         Your choice ?");

    int t=TC.lireInt();

    System.out.println("F(x)="+F(t,x));
  }

  // Observe that here we deliberately chose the function to
     be declared after the main body


  public static double F(int generator, double x)
  {double v=0.0;

    switch(generator)
    {
      case 1: v=x; break;
      case 2: v=Math.log(x); break;
      case 3: v=Math.sin(x); break;
    }

    return v;
  }
}
```

This is one very nice feature of typed programming languages that check that all return paths of functions have appropriate return type `TypeR` (eventually by implicitly casting types).

3.3 Static (class) variables

Once functions have been introduced, we see that variable declarations potentially appear in the body of all respective functions, and are no longer found only in the `main` function. The *variable scopes* are nevertheless restricted to the body of the function delimited by the braces; These variables cannot be

accessed from outside the function bodies.
Suppose now that we would like to count the number of times a given function is called. We need to declare a kind of persistent variable that we can attach to the class encapsulating the function: These kinds of persistent variables are called *static* variables. Using static variables, it becomes easy, say, to count the overall number of function calls as illustrated by the following code:

Program 3.5 An example using a static (class) variable

```
class StaticVariable{
static int nbfunccalls=0;

   static int square(int x)
            {nbfunccalls++;
            return x*x;}

   static boolean isOdd(int p)
            {nbfunccalls++;
            if ((p%2)==0) return false;
                      else return true;}

   static double distance(double x, double y)
            {double result;
            nbfunccalls++;
            if (x>y) result=x-y;
                 else result=y-x;
            return result;
                 }


   static void display(double x, double y)
              {nbfunccalls++;
              System.out.println("("+x+","+y+")");
              }

  public static void main (String[] args)
  {
  FunctionDeclaration.display(3,2);
  display(square(2),distance(5,9));

  int p=123124345;
  if (isOdd(p))
      System.out.println("p is odd");
        else  System.out.println("p is even");

        System.out.println("Total number of function calls:"+
            nbfunccalls);
  }
}
```

Running this program, we get the following message after completion:

```
Total number of function calls:4
```

Static variables are persistent variables attached to the class. These static variables are stored in the *global* memory and can be accessed at *any* time from *any* function. Thus static variables prove very useful for sharing information between different blocks of instructions encapsulated into functions.

3.4 Pass-by-value of function arguments

3.4.1 Basic argument passing mechanism

Whenever calling a function in Java, the arguments (which are potentially arbitrarily complex expressions that may also include other function calls) are evaluated sequentially. Java compiler `javac` then performs the necessary implicit casting operations, if necessary, of these evaluated expressions, and complain with error messages in case the types of *evaluated expressions* do not match the definition types of function arguments. Thus a generic *function call* prototyping syntax is as follows:

```
F(ExprArg1, ..., ExprArgN)
```

Even if the function `F` is called with plain variables as is usually the case (say, function call `F(var1, ..., varN)`), these variables are in fact *basic expressions* that are evaluated in this case to their stored values. Once the body of the function is entered, there is no longer direct access to these variables. Executing the function call thus amounts to executing this function where argument variables and evaluated expressions have been *binded*:

```
static  TypeF F()
{
Type1 arg1=ExprArg1;
...
TypeN argN=ExprArgN;
// Body of the function
...
}
```

3.4.2 Local memory and function call stack

In Java, variables declared in instruction blocks are all created and stored in the *local* memory of the corresponding function. This local memory is named the function call stack. Whenever a function is called, the required memory for the local variables is allocated into the *function stack*, and the arguments are passed by *value* as illustrated in Figure 3.1. Whenever inside the body of the

current function another function, say `G`, is called, the same mechanism applies: First, local memory for storing all block variables declared in the function block of `G` is allocated, and arguments are passed by value. Once `G` is completed, the local memory allocated for its execution is released. These various levels of nested function calls yield the *function call stack*.

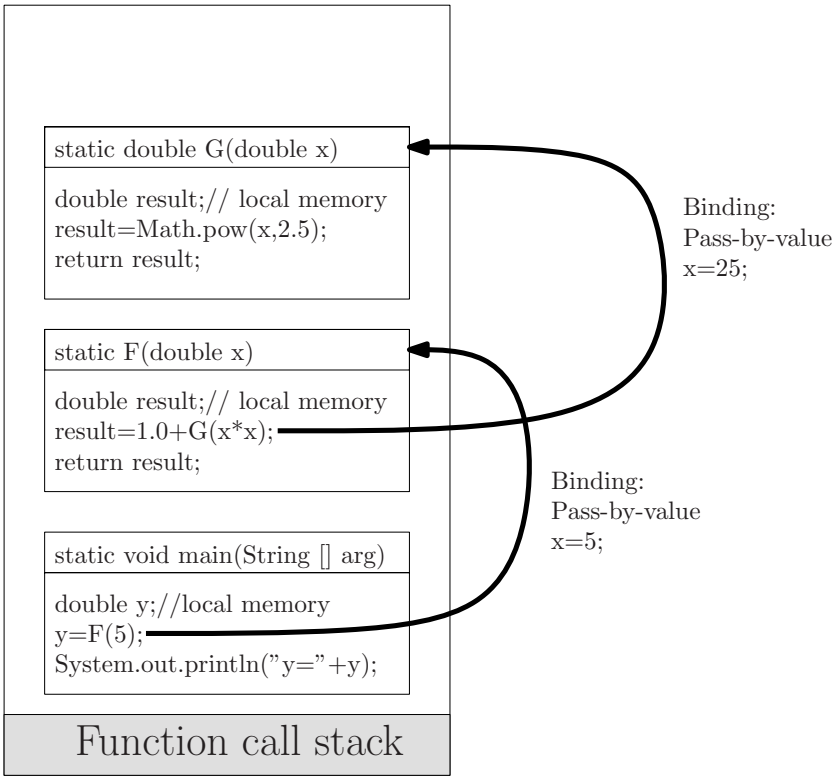


Figure 3.1 Illustrating the function call stack: The function variables are allocated into the local memory stack and are thus not visible to other functions. The pass-by-value mechanism binds the function arguments with the respective expression values at calling time

Program 3.6 Illustrating the function call stack

```
class FunctionStack{

  static double G(double x)
  {
  double result;
  result=Math.pow(x,2.5);
```

```
  return result;
  }

  static double F(double x)
  {
  double result;
  result=1.0+G(x*x);
  return result;
  }

  public static void main (String[] args)
  {
  double y;
  y=F(5.0);
  System.out.println("y="+y);
  }
}
```

For example, considering the above program, the function call stack evolves as follows:

Step	1	2	3	4	5	6	7
Action:	run	call `main`	call `F`	call `G`	finish `G`	finish `F`	finish `main`
Function stack	∅	`main`	`F` `main`	`G` `F` `main`	`F` `main`	`main`	∅

Note that since arguments are passed by values, the local variables of the *calling* functions are not changed. To emphasize this point, consider the following toy sample program:

Program 3.7 Pass-by-value does not change local variable of calling functions

```
class PassByValue{
  static void F(double x)
  {
  x=3; // Here is the catch. Take care of pass-by-value.
  System.out.println("Before exiting F, value of x:"+x);
  }
  public static void main (String[] args)
  {
  double x=5;
  F(x);
  System.out.println("value of x:"+x);
  }
}
```

We get from the pass-by-value mechanism the following output:

```
Before exiting F, value of x:3.0
value of x:5.0
```

That is, when running the program `PassByValue`, the block of instructions in the `main` procedure is executed as follows:

- Local variable `x` of `main` is assigned to value 5,
- Function `F` is called with basic expression `x` that is evaluated to 5 and passed by value to `F` (expression/argument binding),
- Local variable `x` of `F` (not of `main`) is assigned to 3, and the message `Before exiting F, value of x:3.0` is displayed,
- Function `F` is completed, and its local allocated memory is removed from the function stack,
- Message `value of x:5.0` is displayed from the last instruction of the `main` procedure.

In a similar way, swapping two integer variables cannot be achieved by the following erroneous code since Java is pass-by-value:

```
public void badSwap(int var1, int var2)
{
  int temp = var1;
  var1 = var2;
  var2 = temp;
} // at the end of this block, var1 and var2 keeps their original value
```

Loosely speaking, functions usually do not change the *calling* environment. Function `y=F(x)` returns a result that is taken into account in the calling environment. There are of course different ways for functions and procedures to alter the calling environment. This bears the name of *side-effect* functions.

3.4.3 Side-effects of functions: Changing the calling environment

We have previously seen that functions can share the static (class) variables. We gave an example where the overall number of function calls was updated every time a function was called. This is one example of a function side-effect of calling functions: Changing the global environment. Let us give yet another simple example to illustrate this point:

Program 3.8 Toy example for illustrating how functions can change the environment

```
class FunctionSideEffect{
static int x=0;

  static void F()
```

```
  {
  x++;
  }

  static void G()
  {
  --x;
  }

  public static void main (String[] args)
  {
  F(); // x=1
  G(); // x=0
  F(); // x=1
  System.out.println("value of x:"+x);
  }
}
```

The program displays in the console: `value of x:1`. The various program execution steps are visualized as shown in Figure 3.2. Note that in this diagram we split the memory into two parts: The local function call stack memory (function variables) and the global memory (static variables and referenced structures). Thus in Java, we can voluntarily alter the calling environment by handling arrays and other non-primitive user-defined typed objects that are manipulated by their *references*. We will explain in further detail these function side-effects in the following two chapters dealing with arrays and objects.

3.4.4 Function signatures and function overloading

Java is a typed programming language, and functions/procedures declared in Java are also typed by their signature. The signature of the following declared function:

```
static TypeR F(Type1 arg1, Type2 arg2, ..., TypeN argN)
{
TypeR result;
block_instructions;
return result;
}
```

is the Cartesian product of the type of its arguments:

$$\text{Type1} \times \text{Type1} \times \text{TypeN}.$$

In other words, the signatures of functions are their ordered sequence of parameter types. Thus provided that functions have different signatures, they can actually bear potentially the same name; That is, a given function can be

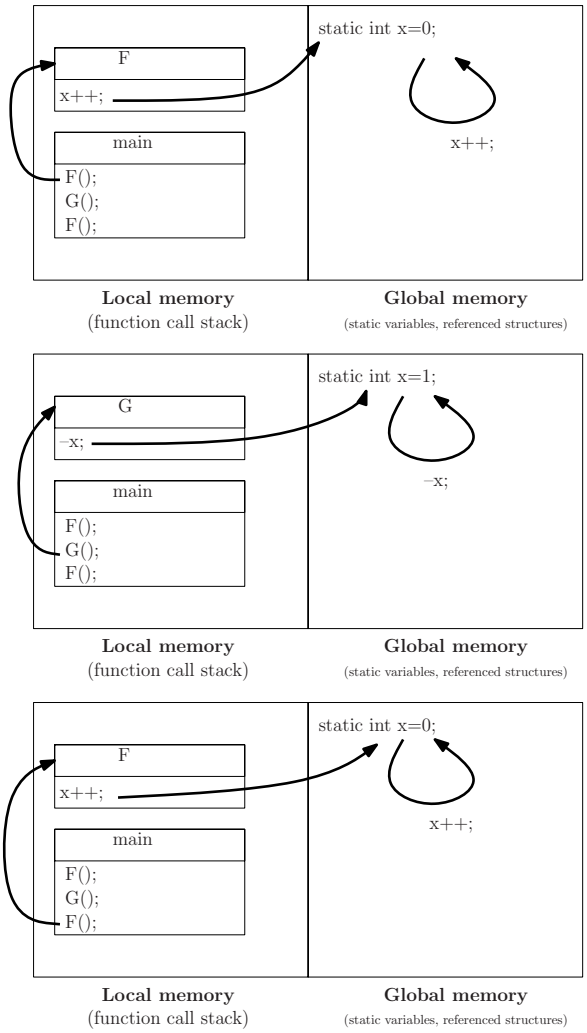


Figure 3.2 Visualizing the local function call stack and the global memory for program `FunctionSideEffect.java`

overloaded provided its argument types differ from the original function. This is useful in a number of situations where we would like to have a function perform the same "function" on different kinds of typed arguments, as illustrated by the following program:

Program 3.9 Function signatures and overloading

```
class PlusOne{
```

```
  static double plusone(int n)
  {return n+1.0;
  }

  static double plusone(double x)
  {return x+1.0;
  }

  static double plusone(String s)
  {
    return Double.parseDouble(s)+1.0;
  }

  public static void main(String[] args)
  {
    System.out.println(plusone(5));
    System.out.println(plusone(6.23));
    System.out.println(plusone("123.2"));
  }
}
```

We get the following output:

```
6.0
7.23
124.2
```

However Java does not take into account the type of the result for creating signatures associated with functions, so that the following code will not properly compile and generates the following error message:
`plusone(int) is already defined in SignatureError static double plusone(int n)`:

Program 3.10 Function signatures do not take into account the return type

```
class SignatureError{
public static int plusone(int n)
{return n+1;}
public static double plusone(int n)
{return n+1.0;}
public static void main(String args[])
{} }
```

3.5 Recursion

Recursion is a powerful principle for writing functions that call themselves. Recursion provides a methodology to get compact, simple and yet elegant algorithms. Since these recursive functions call themselves, we shall take

particularl care of the behavior of the function stack. In particular, we shall focus on program/function termination by inspecting the *terminal states* that correspond to the trivial cases that do not need to call functions.

3.5.1 Revisiting the factorial function: A recursive function

We have seen in §3.2.3 an iterative algorithm for computing the factorial $n! = n \times (n-1) \times ... \times 1$. Let us now write the factorial function the recursive way. First, we need to find a recurrence equation for the factorial function. We simply get this relationship as:

$$n! = n \times (n-1) \times ... \times 1 = n \times (n-1)!$$

with $0! = 1$ by definition (terminal state).

Program 3.11 Recursive implementation of the factorial function

```
class RecursiveFactorial
{
  public static int Factorial(int n)
  {
    if (n==0) return 1; // Terminal stage
    else return n*Factorial(n-1); // Apply recurrence
        equation
  }

  public static void main(String[] arg)
  {System.out.println("10!="+Factorial(10));}
}
```

Compiling and executing this code, we get `10!=3628800`. Does this recursive function always terminate? For a given integer $n \geq 0$, the function `Factorial` calls itself until we reach at some stage the argument $n = 0$. Thus the function stack has piled $n+1$ calls with the last call being `Factorial(0)`. At this stage, we enter the terminal stage and return 1. `Factorial(1)` can return its result $1 \times 1 = 1$ and it is removed from the stack. Then `Factorial(2)` can return its result 2×1, and so on, until we return the result of `Factorial(n)`. Figure 3.3 illustrates the pile on process of the function call stack until it reaches the terminal state $n = 0$.

Note that if we call `Factorial(-1)`, the program will pile-on the function stack the function calls `Factorial(-2)`, `Factorial(-3)`, etc. and stop once the function stack gets full, generating an overflow memory problem: A *stack overflow*.

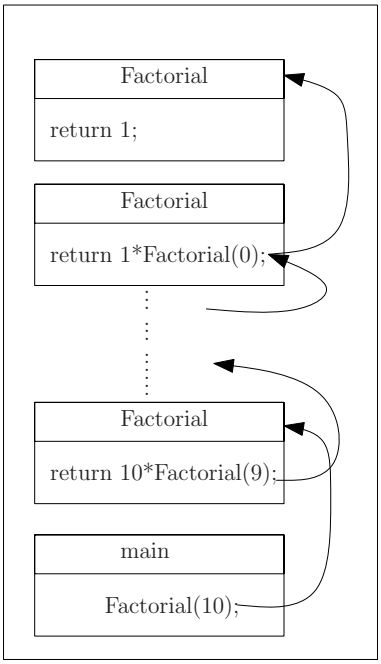


Figure 3.3 Visualizing the function call stack for the recursive factorial function.

3.5.2 Fibonacci sequences

Fibonacci sequences were introduced by Leonard de Pise[1] as the following sequence $\{F_i\}_i$ of natural numbers:

$$F_i = \begin{cases} 1 & \text{for } i = 1 \text{ or } i = 2, \\ F_{i-1} + F_{i-2} & \text{otherwise.} \end{cases}$$

The first Fibonacci numbers are given by $1, 1, 2, 3, 5, 8, 13, 21, 34, 55...$ Fibonacci sequences are important for studying the characteristics of *population growth* under a basic model. Let us explain this model for a population of male/female rabbits:

– First, put a newly born pair of male/female rabbits in an enclosed field.

– Assume that newly born rabbits take a month to become mature, after which time...

– ... Each pair of mature rabbits produces a new pair of baby rabbits every month.

[1] Leonard de Pise (1170-1245) is better known as Fibonacci.

The question Fibonacci originally raised was:

"How many pairs of rabbits will there be in the following months?"

Obviously at first, we put a single pair of newly born rabbits and have the first month $F_1 = 1$ ($\rightarrow$ terminal state). Then we need to wait for another month for these rabbits to become mature so that $F_2 = 1$ ($\rightarrow$ terminal state). After which, the number of rabbits F_i at month i is the number of rabbits at month $i-1$ plus the number of rabbits newly generated by mature rabbits. The number of mature rabbits at month i is precisely F_{i-2}. Thus comes the following recurrence relationship: $F_i = F_{i-1} + F_{i-2}$ with $F_1 = F_2 = 1$ (terminal states).

Program 3.12 Displaying Fibonacci sequences using recursion

```
class FibonacciSequence{
  public static int Fibonacci(int n)
  {
    if (n<=1) return 1;
    else
      return Fibonacci(n-1)+Fibonacci(n-2);
    }

  public static void main(String[] args)
  {int i;
    for(i=0;i<=30;i++)
      System.out.print(Fibonacci(i)+" ");
  }
}
```

Running the above program, we get the first 31 Fibonacci numbers: `1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711 28657 46368 75025 121 393 196418 317811 514229 832040 1346269`

3.5.3 Logarithmic mean

The logarithmic mean is often used in engineering to measure temperature differences. There are several non-equivalent definitions proposed in the literature. Here, we consider for our programming purpose the following recursive definition:

$$L(x_1, ..., x_n) = (n-1)\frac{L(x_2, ..., x_n) - L(x_1, ..., x_{n-1})}{\log x_n - \log x_1},$$

with $L(x) = x$ being the terminal case. Hence, for two elements x and y, we have the logarithmic mean expressed as

$$L(x, y) = \frac{x - y}{\log x - \log y}.$$

The recursive program below implements the logarithmic mean computation:

Program 3.13 Logarithmic mean

```
static double LogarithmicMean(double [] array, int i, int j)
{
// Terminal case: One element. The mean is always this
    element
if (j-i==0) return array[i];
else
{
int n=j-i+1;
return (n-1)*(LogarithmicMean(array,i+1,j)-LogarithmicMean(
    array,i,j-1))/(Math.log(array[j])-Math.log(array[i])) ;
}
}

static double LogarithmicMean(double [] array)
{
return LogarithmicMean(array,0,array.length-1);
}
```

3.6 Terminal recursion for program efficiency **

Although it is quite easy to write recursive functions once recurrence equations and terminal states are clearly identified, the main drawback of using recursive programs in practice is efficiency. Indeed, every time we call a function, the Java Virtual Machine (JVM) has to allocate some local memory for the local variables declared inside the body of the function. Then it has to perform the pass-by-value of arguments. Another major issue is that the function stack is limited in its size, and it is quite easy in practice to reach stack overflow problems. Terminal recursion is a principle that solves these time and memory problems while keeping the essence of recursion. A recursive function is said to be *terminal* if and only if wherever the function calls itself it is with the following simple expression `f(...);`. In other words, a function is *terminal recursive* if all its *return paths* are of the form `return f(...);`. Terminal recursion is effective because it simply allows one to branch former arguments into new arguments by performing various expression evaluations on *parameters* only. Therefore there is no need to use the stack function calls, and no stack overflow problems are occurring when using terminal recursion. In other words, arguments of terminal recursive functions play the role of *accumulators*.
Let us reconsider the factorial recursive function:

```
if (n<=1) return 1; else
return n*f(n-1);
```

...and write it using terminal recursion as follows:

Program 3.14 Writing the factorial function using terminal recursion

```
class TerminalRecursionFactorial{

  static long FactorialRecTerminal(int n, int i, int result)
  {
    if (n==i) return result;
      else
        return FactorialRecTerminal(n,i+1,result*(i+1));
  }

  static long FactorialLaunch(int n)
  {
    if (n<=1) return n;
    else return FactorialRecTerminal(n,1,1);
  }

  public static void main(String[] args)
  {
    System.out.println("Factorial 10!="+FactorialLaunch(10));
  }
}
```

Similarly, we can revisit the Fibonacci recursive program to write using terminal recursion as follows:

Program 3.15 Fibonacci calculation using terminal recursion

```
class FibonacciTerminalRecursion{

  static int FibonacciRecTerm(int n, int i, int a, int b)
  {
    if (n==i) return a;
        else return FibonacciRecTerm(n,i+1,a+b,a);
  }

  static int FibonacciLaunch(int n)
  {if (n<=1) return n;
        else return FibonacciRecTerm(n,0,0,1);
  }

  public static void main(String[] arg)
  {
    System.out.println("Fibonacci(7)="+FibonacciLaunch(7));
  }
}
```

3.7 Recursion and graphics **

Beautiful recursive geometric patterns are often observed in nature. To give a single example, consider the shapes of snowflakes: They look similar to Koch's mathematical snowflakes illustrated in Figure 3.4. These patterns are called *fractal* patterns since they are defined by a recursive process.

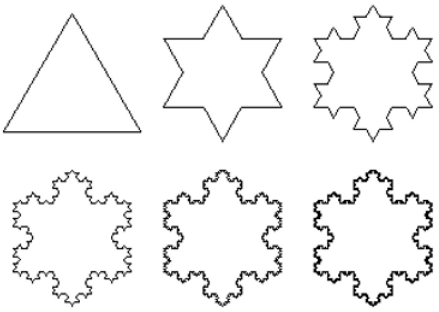


Figure 3.4 Koch's mathematical recursive snowflakes

One famous fractal mathematician is Waclaw Sierpinski (1882-1969), who studied such recursive patterns. For example, the Sierpinski's triangle recursive pattern is given by the following simple *rewriting* rule:

Replace a given (parent) triangle by three (children) triangles defined by the midpoints of the edges of the parent triangle. Figure 3.5 shows Sierpinski's fractal triangle obtained after a few recursive graphics rewriting operations. The figure was produced by the following more elaborate Java program which recursively draws the fractal pattern using procedure `sierpinski_draw`.

Program 3.16 Sierpinski's fractal triangles

```
import javax.swing.*;
import java.awt.*;
public class Sierpinski extends JFrame {
  public static final int WINDOW_SIZE = 512;
  public static final int THRESHOLD=10; // stopping criterion
      for recursion
  public static int P1_x, P1_y, P2_x, P2_y, P3_x, P3_y;

  public Sierpinski() {
    super("Sierpinski");
    setSize(WINDOW_SIZE, WINDOW_SIZE);
    // A simple triangle
    P1_x = (int)getSize().getWidth()/2;;
    P1_y = 40;
    P2_x = 20;
    P2_y = (int)getSize().getHeight() - 20;
    P3_x = (int)getSize().getWidth() - 20;
```

```
    P3_y = (int)getSize().getHeight() - 20;

    setVisible(true); setDefaultCloseOperation(JFrame.
        EXIT_ON_CLOSE);
  }
  // Compute the midpoint
  public Point getMiddle(Point p1, Point p2) {
    return new Point((int)(p1.getX() + p2.getX())/2, (int)(p1
        .getY() + p2.getY())/2);
  }
  public void paint(Graphics g) {
    super.paint(g);
    sierpinski_draw(new Point(P1_x, P1_y),new Point(P2_x,
        P2_y), new Point(P3_x,P3_y));
  }
  public void sierpinski_draw(Point p1, Point p2, Point p3) {
    //termination condition
    if (p1.distance(p2) < THRESHOLD && p1.distance(p3) <
        THRESHOLD &&
      p2.distance(p3) < THRESHOLD) return; // stop recursion
    //draw the current triangle
    Graphics g = getGraphics();
    g.drawLine((int)p1.getX(),(int)p1.getY(),(int)p2.getX(),(
        int)p2.getY());
    g.drawLine((int)p2.getX(),(int)p2.getY(),(int)p3.getX(),(
        int)p3.getY());
    g.drawLine((int)p3.getX(),(int)p3.getY(),(int)p1.getX(),(
        int)p1.getY());
    //recursively draw the 3 smaller corner triangles
    Point m12 = getMiddle(p1, p2);
    Point m23 = getMiddle(p2, p3);
    Point m31 = getMiddle(p3, p1);
    // Recursive calls
    sierpinski_draw(p1, m12, m31);
    sierpinski_draw(p2, m23, m12);
    sierpinski_draw(p3, m31, m23);
  }
  public static void main(String[] args) {
     new Sierpinski();
  }
}
```

3.8 Halting problem: An undecidable task

We have shown that recursion is a very powerful concept for writing compact functions that calculates results using a recurrence relation with associated terminal states. A major problem is to know whether all terminal states are properly studied. Otherwise, the function might call itself forever, or at least

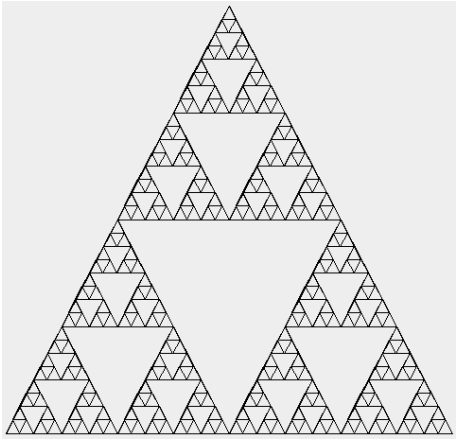


Figure 3.5 Sierpinski's triangle fractal obtained by the program `Sierpinski.java`

until the function call stack gets full. However even if all terminal states are scrupusously taken care of, it is *impossible* to prove that such a function will terminate. This crucial limitation is known by the name of the *halting problem* in *computer science*.

Even for simple recurrence relations, it is very challenging to mathematically prove termination. For example, consider the Syracuse sequence of numbers defined in §2.5.5 as follows:

$$u_{n+1} = \begin{cases} u_n/2 & \text{if } n \text{ is even,} \\ 3u_n + 1 & \text{otherwise.} \end{cases},$$

initialized for any given $u_0 \in \mathbb{N}$. It is conjectured but *not yet proved* that for any $u_0 \geq 1$ the sequence reaches in a finite step number 1. The following program tests experimentally the termination of the recursive function:

Program 3.17 Recursive Syracuse: Testing for termination

```
class RecursiveSyracuse
{
  public static double Syracuse(int n)
  {
  if (n==1) return 1;
  else
   if (n%2==0) return 1+Syracuse(n/2); // even
   else return (1+Syracuse(3*n+1)/2);
  }
  public static void main(String[] args)
  {
    for(int i=1; i<=10000; i++)
    {
      System.out.println("Test termination for "+i);
       Syracuse(i);
    }
```

```
    }
}
```

Gödel[2] formally proved that there is *no* program that can decide whether or not any given program entered as an argument will stop after a finite number of instructions. Indeed, loosely speaking, suppose that we have at our disposal a "special" Java program `Terminate(Prog)` that returns `true` if and only if a program `Prog` terminates, and `false` otherwise. Then we could design the following function:

```
public static void UndecidableProg()
{
while(Terminate(UndecidableProg))
{}
}
```

Does the function `UndecidableProg` terminate or not? If it terminates, then `Terminate(UndecidableProg)` is `true`, and the function loops forever, thus not terminating. Or it does not terminate but `Terminate(UndecidableProg)` is `true` so that it terminates. This yields a contradiction. Note that Gödel's Proof of Incompleteness Theorem[3] is way beyond this informal sketch.

3.9 Exercises

Exercise 3.1 (Computing function values)

Write a function that computes $f(x) = \sqrt{x+1}$ for x of type `double`. Test this function in the `main` program function by displaying the result of calling $f(2)$ and $f(3)$. Then use a `for` loop statement to display $f(i)$ for $i \in \{0, ..., 9\}$. Finally, modify this program to display $f(x)$ for all $x \in [0, 1)$ by increment of step size 0.1.

Exercise 3.2 (Power function and one of its properties)

Write a function `power` that takes two integer arguments `a` and `b` (with $b \geq 0$), and that returns the result a^b. The exponentiation shall be computed by accumulating b multiplications. Then use this exponentiation function in the `main` program to compute a^b for a and b given interactively by the user.

For c, a non-negative integer, we have the following property: $a^{b^c} = a^{bc}$. Indeed, we have $a^{b^c} = e^{c \log a^b} = e^{bc \log a} = a^{bc}$. Write a function `check`

[2] Kurt Gödel (1906-1978).
[3] `http://en.wikipedia.org/wiki/Halting_problem`

that uses former function `power` that returns true if and only if this equality is numerically satisfied (and `false` otherwise). Modify the `main` function so that the user can also input the value of `c` interactively at the console. When might this inequality fail in your program?

Exercise 3.3 (Binary representation)

Give a *recursive* function `DisplayBase2` that takes as its argument a non-negative number, and report its binary decomposition. For example, the binary decomposition of the decimal number 11 is 1011: $(11)_{10} = (1011)_2$. That is, we have the following unique decomposition: $11 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$. This binary decomposition can be obtained by successively dividing by 2 and taking the remainder 0 or 1 each time.

Exercise 3.4 (Euclid's greatest common divisor)

Use the property that $\mathrm{GCD}(a, b) = \mathrm{GCD}(b, a \mod b)$ and $\mathrm{GCD}(a, 0) = |a|$ for designing a recursive function `static int GCD(int a, int b)`. One can use the function `Math.abs(x)` of the `Math` class to calculate the absolute value of a given number x. Visualize the recursive function calls by displaying the various function call parameters `a` and `b`. Illustrate the recursive calls and the function call stack for $a = 15$ and $b = 21$. Compare your code with the following one that uses a ternary operator:

Program 3.18 Euclid's greatest common divisor using recursion

```
static int gcd(int a, int b)
{
   return ( b != 0 ? gcd(b, a % b) : a );
}
```

Exercise 3.5 (Week day calendar)

The *week day* X of a given date given as M/D/Y (for month/day/year) can be encoded as an integer X in range $X \in \{0, ..., 6\}$ with Sunday being encoded by 0. The week day is calculated by the following intricate formula:

$$X = \left(D + \lfloor 2.6 * M' - 0.2 \rfloor + E + \left\lfloor \frac{E}{4} \right\rfloor + \left\lfloor \frac{S}{4} \right\rfloor - 2S \right) \mod 7,$$

where

$$(M', Y') = \begin{cases} (M-2, Y) & \text{if } M > 2, \\ (M+10, Y-1) & \text{if } M \leq 2. \end{cases}$$

and $Y' = 100S + E$ with $0 \leq E \leq 100$. Write a program that takes as input a date formatted using the M/D/Y style and report

the corresponding week day ("Sunday", ..., "Saturday") to the output console using a `switch` conditional statement.

Exercise 3.6 (Calculating $\sin nx$ and $\cos nx$)

Write *recursive* functions `sin(int n, double x)` and `cos(int n, double x)` that respectively compute $\sin nx$ and $\cos nx$ using the following trigonometric formula:

$$\sin(nx) = \sin x \cos(n-1)x + \cos x \sin(n-1)x,$$
$$\cos(nx) = \cos x \cos(n-1)x - \sin x \sin(n-1)x.$$

(*Hint:* Consider $\sin 0 = 0$ and $\cos 0 = 1$ as the terminal cases.)

*Exercise 3.7 (Palindrome **)*

A palindrome is a word or phrase that can be read in *either* direction like "radar" or "Was it a rat I saw." Write an iterative function `CheckingPalindrome(String str)` that checks whether a word/phrase stored in `str` is a palindrome or not. One shall use the `charAt(int pos)` method of the `String` class that reports the character of type `char` located at position `pos` in the string. Checking whether a string is a palindrome or not can also be done *recursively* as follows: A word `s` is a palindrome if there exists a *word* `w` and a *character* `c` such that $s = cwc$ and w is a palindrome (of smaller length). Design a recursive function `CheckingPalindromeRec(String str, int left, int right)` that checks whether the portion of the string `str` delimited by the index range `left` and `right` is a palindrome or not. Test your program by calling the function `CheckingPalindromeRec(str, 0, str.length()-1);`

4 Arrays

4.1 Why do programmers need arrays?

Arrays are useful for processing *many* data or generating *many* results at once into a compact contiguous structure. Loosely speaking, array structures allow one to manipulate *many* variables at once. An array is an *indexed* sequence of components. In mathematics, one is familiar with variables bearing indices like, for example, the vector coordinates x_i or matrix elements $m_{i,j}$. In most programming languages, indices start at zero and *not* at one as is often the case in mathematics. This simple 1-or-0 convention actually yields confusion to many novice programmers, and is therefore an important source of bugs to watch for.

4.2 Declaring and initializing arrays

4.2.1 Declaring arrays

In Java, arrays are also *typed* structures: For a given type, say `TYPE`, `TYPE[]` is the type of arrays storing a collection of homogeneous elements of type `TYPE`. Local arrays are declared within the body of functions (delimited by braces) as follows:

```
int [ ] x; // array of integers
```

F. Nielsen, *A Concise and Practical Introduction to Programming Algorithms in Java*, Undergraduate Topics in Computer Science, DOI 10.1007/978-1-84882-339-6_4,

```
boolean [ ] prime; // array of booleans
double [ ] coordinates; // arrays of reals formatted using double precision
```

Similarly, static class arrays are declared inside the body of a class using the keyword `static`:

```
class Example{
static int [ ] x;
static boolean [ ] prime;
...
}
```

These (class) static array variables can then be used and shared by *any* function of the class. In both local/class cases, arrays are allocated into the *global* memory, and not into the function call stack. Only the references of arrays may be stored into the function call stack for locally declared arrays.

4.2.2 Creating and initializing arrays

Arrays are *created* and *initialized* by default using the Java reserved keyword `new`:

```
int [ ] x = new int[5];
boolean [ ] prime = new boolean[16];
```

The size of arrays needs to be specified:

```
x=new int [32];
```

The size of an array can also be given as an integer arithmetic expression like `2*n`, etc:

```
x=new int [2*n+3];
```

Arrays can only be declared once but may eventually be created and initialized several times. This recreation process overrides the former creation/initialization:

```
x=new int [2*n+3];
...
x=new int [4*n-2]; // overrides the former creation and initialization
```

By default, initialization of arrays is performed by Java by filling all its elements with 0, or by casting this 0 to the equivalent array element type: For example, `false` for booleans, `0.0d` for `double`, `0.0f` for `float`, etc. Initialization can also be explicitly done by *enumerating* all its elements separated by comas "," within curly brackets {} as follows:

```
int [ ] prime={2, 3, 5, 7, 11, 13, 17, 19};
boolean prime[]={ false, true, true, true, false, true, false, true};
```

In that case, the size of the array is *deduced* from the number of elements in the set, and should therefore not be explicitly specified. Nor shall there be an explicit creation using the keyword `new`. Here are a few examples illustrating static array declarations, creations and initializations. Arrays can be declared within the body of a function or globally as a static variable of the class. To illustrate global static arrays, consider the following program:

Program 4.1 Static array declarations and creations

```
class ArrayDeclaration{
  static int digit [] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0};
  static double x [] = {Math.PI, Math.E, 1.0, 0.0};
  static boolean prime[]={ false, true, true, true, false,
     true, false, true, false, false };
  static int y[];
  static void MyFunction(int n)
  {
    // Allocate an array of size n
    y=new int[2*n];
  }
  public static void main (String[] args)
  {
  MyFunction(15);
  // We recreate and initialize array y;
  MyFunction(20); }
}
```

Observe that in this sample program, array `y` is created twice. In any case, arrays are allocated and stored in the global memory, and not stored in the local function call stack. Only the references of arrays are stored in the function stack for arrays declared within functions (without the keyword `static`).

4.2.3 Retrieving the size of arrays: `length`

Arrays in Java carry additional information[1] about themselves: Their types and lengths. The size of an array `array` is accessed by using the keyword `length`, post-appended to the array name with a "." dot:

```
array.length
```

Observe that there are no parenthesis used in conjunction with the keyword `length`.

```
static boolean prime[]={ false, true, true, true, false, true
    , false, true, false, false };
```

[1] Technically speaking, we say that arrays in Java are reflexive since they contain additional information. This is to contrast with arrays in C or C++ that are *non-reflexive* since they contain only their components.

```
System.out.println(prime.length);
```

We cannot change the size of arrays once initialized. For example, trying to force the length of array `array` by setting `array.length=23;` will generate the following compiler message error: `cannot assign a value to final variable length`.

4.2.4 Index range of arrays and out-of-range exceptions

Arrays created with the syntax `array=new TYPE[Expression]` have a fixed length determined at the instruction call time by evaluating the expression `Expression` to its integer value, say l (with `l=array.length`). The elements of that array are accessed by using an index ranging from 0 (lower bound) to $l-1$ (upper bound):

```
array[0]
...
array[l-1]
```

A frequent programming error is to try to access an element that does not belong to the array by giving an inappropriate index falling out of the range $[0, l-1]$. The following program demonstrates that Java raises an exception if we try to use out-of-range indices:

Program 4.2 Arrays and index out of bounds exception

```
class ArrayBound{
  public static void main (String[] args)
  {
    int [ ] v={0,1,2,3,4,5,6,7,8};
    long l=v.length;
    System.out.println("Size of array v:"+l);
    System.out.println(v[4]);
    System.out.println(v[12]);
  }
}
```

Running the above program yields the following console output:

```
Size of array v:9
4
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 12
        at ArrayBound.main(ArrayBound.java:8)
```

That is, index out of bounds cannot be checked by the compiler and may happen at *any* time when executing the program.

A *subarray* is a *consecutive portion* of an array: For example, `array[3..7];`. Java does not provide language support for manipulating subarrays so that one has to explicitly keep track of the lower and upper bounds of subarrays.

4.2.5 Releasing memory and garbage collector

In Java, one does not need to explicitly release memory of unused memory structures such as arrays. The Java virtual machine (JVM) does it *fully automatically* using the *garbage collector*. Once the JVM detects that elements of an array cannot be accessed anymore because the reference of that array has been released by, say, the function call stack, the garbage collector will free that memory. This is a key difference with another popular programming language: C++. The garbage collector checks at any time whether elements of a given array can still be accessed by some variables holding a reference to that array or not. If not, the garbage collector releases that global memory and will perform some memory cleaning operations. Nevertheless, we can also *explicitly* indicate to the JVM that we do not want the array anymore by setting the reference of that array to `null`, meaning that we erase the array reference:

```
int [] array=new int[32];
array=null; // explicitly indicate to the JVM to release the array
```

4.3 The fundamental concept of array references

Whether the array is declared as a local variable or as a global (static/class) variable, all its elements are stored in the program global memory. That is, even if local array variables are declared, created and initialized within a function, their elements may still be accessed by the calling function once the function is completed. This provides an essential mechanism for voluntarily having side-effect phenomena in functions that can therefore potentially change the (global) program environment. An array variable `array` is dealt as a *reference* to that array, a single machine word from which its indexed elements can be accessed. The notion of reference for non-primitive types in Java is essential. It can be quite delicate to grasp at first for novice programmers but nevertheless is essential. The main advantages of handling array variables (whatever their sizes) as references (a single machine word using four bytes[2]) are as follows:

- References provide a mechanism for functions to access and modify elements of arrays that are preserved when functions exit.
- When calling a function with array arguments, Java does not need to allocate the full array on the *function call stack*, but rather pass a single reference to that array. Therefore it is computationally and memory efficient.

[2] That is equivalently 32 bits to reference a given memory location.

Furthermore, this pass-by-reference mechanism limits the risk of function stack overflow.

To illustate the notion of array references, consider the following set of instructions:

Program 4.3 Arrays and references

```
int [ ] v = {0, 1, 2, 3, 4};
// That is, v[0]=0, v[1]=1, v[2]=2, v[3]=3, v[4]=4;
int [ ] t =v;
// Reference of t is assigned to the reference of v so that t
    [i]=v[i]
t[2]++; // Post-incrementation: t[2]=v[2]=3
System.out.println(t[2]++);
// Display 3 and increment t[2]=v[2]=4 now
```

The result displayed in the console is 3. In summary, an array is allocated as a single contiguous memory block. An array variable stores a reference to the array: This reference of the array links to the symbolic memory address of its first element (indexed by 0).

Program 4.4 Assign an array reference to another array: Sharing common elements

```
class ArrayReference{
  public static void main (String[] args)
  {
    int [] v={0,1,2,3,4};
    System.out.println("Reference of array u in memory:"+v);
    System.out.println("Value of the 3rd element of array v:"
       +v[2]);
    // Declare a new array and assign its reference to the
       reference of array v
    int [] t =v;
    System.out.println("Reference of array v in memory:"+v);
       // same as u
    System.out.println(v[2]);
    t[2]++;
    System.out.println(v[2]);
    v[2]++;
    System.out.println(t[2]);
  }
}
```

Running this program, we notice that the reference of the array u coincides with the reference of array v:

```
Reference of array u in memory:[I@3e25a5
Value of the 3rd element of array v:2
Reference of array v in memory:[I@3e25a5
2
```

```
3
4
```

The `I` in `[I@3e25a5` indicates that this is a reference to an array of integers. The forefront letter varies according to the type of array elements as illustrated by the following code:

Program 4.5 Printing the references of various typed arrays

```
class ArrayDisplay{
  public static void main (String[] args)
  {
  int [] x=new int[10]; System.out.println(x);
  double [] y=new double[20]; System.out.println(y);
  float [] z=new float[5]; System.out.println(z);
  boolean [] b=new boolean[7];System.out.println(b);
  }
}
```

We get[3]:

```
[I@3e25a5
[D@19821f
[F@addbf1
[Z@42e816
```

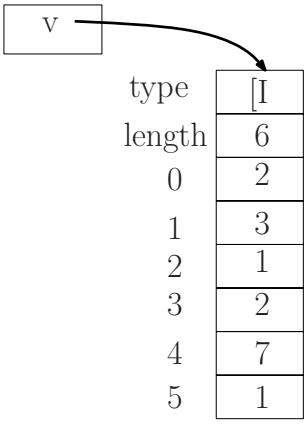


Figure 4.1 A way to visualize arrays in Java, explained for the array declaration and assignment: **int** [] v={2,3,1,2,7,1};.

One way to depict arrays is shown in Figure 4.1. Note that the length and type of the array are indicated in this representation (reflexive arrays).

[3] An array of strings has type `[Ljava.lang.String;`

4.4 Arrays as function arguments

Functions and procedures can have arrays as arguments too. Remember that arrays of element types `TYPE` are themselves of type `TYPE []`, so that the syntax for declaring a function using an array argument and calling it is:

```
static void MyFunction(int [ ] x)
{...}
...
int [] v=new int[10];
// Calling the function with an array argument
MyFunction(v);
```

For example, consider implementing a function that returns the minimum element of an array of integers provided as a function argument:

Program 4.6 Array argument in functions: Minimum element of an array

```
class ArrayMinimum{
  static int minArray(int [] t)
  {
    int m=t[0];
    for(int i=1;i<t.length; ++i)
      if (t[i]<m)
        m=t[i];
        return m;
  }

  public static void main(String[] args)
  {
  int [] v=new int [23];

  for(int i=0;i<23;i++)
      v[i]=(int)(Math.random()*100); // int from 0 to 99

  System.out.println("The minimum of the array is :"+minArray
      (v));
  }
}
```

Since we initialize the array by filling it with random elements using the `Math.random()` function, running the code several times may yield different outputs. For example, running the compiled bytecode three times in a row yields the following results:

```
The minimum of the array is :4
The minimum of the array is :2
The minimum of the array is :1
```

We say that the code is *non-deterministic* because it uses some randomness[4] provided by the function `Math.random()`. The following example demonstrates that only references of arrays are passed by functions:

Program 4.7 Creating and reporting array information using functions

```
class ArrayInFunction{
  public static void MyFunction(int n)
  {
    int array []=new int [n];
    int i;
    InformationArray(array);
  }
  public static void InformationArray(int [] t)
  {System.out.println("Size of array given in argument is:"+t
      .length);}

  public static void main (String[] args)
  {
    MyFunction(2312);
    MyFunction(2008);
    int x []=new int [12];
  }
}
```

Running the program, we get:

```
Size of array given in argument is:2312
Size of array given in argument is:2008
```

Arrays are useful structures for storing coordinates of vectors. Let us consider programming the inner product of two vectors *modeled* as arrays. We end-up with the following code:

Program 4.8 Calculating the inner product of two vectors given as arrays

```
class VectorInnerProduct{

  static double innerproduct(int [] x, int [] y)
  {
  double sum=0.0;
  System.out.println("Dim of vector x:"+x.length+ " Dim of
      vector y:"+y.length);

  for(int i=0;i<x.length; ++i)
        sum=sum+x[i]*y[i];
  return sum;
  }

  public static void main(String[] args)
```

[4] Since the randomness is emulated by some specific algorithms, we prefer to use the term *pseudo-randomness*.

```
  {
  int dimension=30;
  int [] v1,  v2;

  v1=new int[dimension]; v2=new int[dimension];

  for(int i=0;i<dimension;i++)
    {v1[i]=(int)(Math.random()*100);// random int [0,99]
     v2[i]=(int)(Math.random()*100);// random int [0,99]
    }
  System.out.println("The inner product of v1 and v2 is "+
      innerproduct(v1,v2));
  }
}
```

Running this program, we get:

```
Dim of vector x:30 Dim of vector y:30
The inner product of v1 and v2 is 80108.0
```

Static (class) functions may also return an array as a result of their calculation. A typical example is the addition of two vectors that yields *another* vector of the same dimension:

Program 4.9 Function returning an array: Addition of vectors

```
class VectorAddition{

  static int [] VectorAddition(int [] u, int [] v)
  {
    int[] result=new int[u.length];

    for(int i=0;i<u.length;i++)
      result[i]=u[i]+v[i];

      return result;
  }

  public static void main(String[] args)
  {
  int [] x={1, 2, 3}; int [] y={4, 5, 6};
  int [] z= VectorAddition(x,y);

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

The following example demonstrates how one can persistently modify inside a function the contents of an array passed as an argument. That is, this array element swapping program shows that the values of the elements of the array can be changed after exiting the function.

Program 4.10 Swapping array elements by calling a function

```
class ModifyArray{
static void swap(int [] t, int i, int j)
{
  int tmp;
  tmp=t[i];
  t[i]=t[j];
  t[j]=tmp;
}

static void DisplayArray(int [] x)
{ for(int i=0;i<x.length;i++)
    System.out.print(x[i]+" ");
  System.out.println();
}

public static void main(String[] args)
{
int [] t={1,2,3,4,5,6,7,8,9};
DisplayArray(t);
swap(t,2,3);
DisplayArray(t);
  }
}
```

We observe that the third and fourth element (corresponding respectively to index 2 and 3) have indeed been swapped:

```
1 2 3 4 5 6 7 8 9
1 2 4 3 5 6 7 8 9
```

4.5 Multi-dimensional arrays: Arrays of arrays

4.5.1 Multi-dimensional regular arrays

We have so far considered linear arrays (also called 1D arrays). These 1D arrays have proved useful for storing vector coordinates and processing arithmetic operations on them (see, for example, the former scalar product and vector addition programs). What about manipulating 2D matrices $M = [m_{i,j}]$ with n rows and m columns? Of course, once the dimensions n and m are known, we can map the elements $m_{i,j}$ of a 2D matrix to a 1D vector in $\mathbb{R}^{n \times m}$ by linearizing the matrix and using the following index correspondence:

$$(i, j) \Leftrightarrow i \times m + j.$$

This index remapping[5] is quite cumbersome to use in practice and may yield various insidious bugs. Fortunately in Java, we can also create *multi-dimensional* arrays easily; Java will perform the necessary index remapping accordingly. A regular bi-dimensional array consists of n lines, each line being itself an array of m elements. A 2D matrix of integers has type `int [] []` and is declared, created and initialized as follows:

```
int [ ] [ ] matrix;
matrix=new int[n][m];
```

By default at the initialization stage, the array `matrix` is filled up with all zero elements. We can change the contents of this 2D array using two *nested* loops, as follows:

```
for(int i=0; i<n; i++)
  for(int j=0; j<m;j++)
      matrix[i][j]=i*j+1;
```

These constructions extend to arbitrary array dimensions. For example, a 3D array may be defined as follows:

```
int [ ] [ ] [] volume;
volume=new double[depth][height][width];
```

Let us illustrate the manipulations of linear and bi-dimensional arrays by implementing the matrix vector product operation. Observe the declaration/creation and initialization of a 2D array by enumerating all its elements:

```
int [][] M={{1, 2, 3}, {4,5,6}, {7,8,9}};
```

Program 4.11 Matrix-vector product function

```
class MatrixVectorProduct{
  static int [] MultiplyMatrixVector(int [][] mat, int [] v)
  {
    int[] result;
    result=new int[mat.length];

    for(int i=0;i<result.length;i++)
    {
      result[i]=0;
      for(int j=0;j<v.length;j++)
        result[i]+= mat[i][j]*v[j];
    }
      return result;
  }
  public static void main(String[] args)
  {
    int [][] M={{1, 2, 3}, {4,5,6}, {7,8,9}};
    int [] v={1,2,3};
```

[5] We arbitrarily chose row major order. We can also choose the column major order.

```
    int [] z= MultiplyMatrixVector(M,v);

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

Thus it is quite easy to write basic functions of *linear algebra*. Note that in Java, it is not necessary[6] to provide the function with the array dimensions since we can retrieve these dimensions with the `length` keyword, as shown below:

Program 4.12 Creating multidimensional arrays and retrieving their dimensions

```
class MultidimArrays
{
static void f2D(double [] [] tab)
{
  System.out.println("Number of lines:"+tab.length);
  System.out.println("Number of columns:"+tab[0].length);
}
static void f3D(double [] [] [] tab)
{
  System.out.println("Number of lines X:"+tab.length);
  System.out.println("Number of columns Y:"+tab[0].length);
  System.out.println("Number of depths Z:"+tab[0][0].length);
}
public static void main(String[] args)
{
  double [] [] var=new double[3][4];
  f2D(var);
  double [] [] [] tmp=new double[4][5][7];
  f3D(tmp);
}
}
```

Running this program, we see that we correctly retrieved the 2D and 3D array dimensions given as function arguments:

```
Number of lines:3
Number of columns:4
Number of lines X:4
Number of columns Y:5
Number of depths Z:7
```

4.5.2 Multi-dimensional ragged arrays **

Multi-dimensional arrays need not be regular: They can be completely irregular. That is, a multi-dimensional array can also be defined as a 1D array of arrays,

[6] In the C programming language, one *has* to pass these dimensions as arguments.

each array "element" being itself an array with its own dimensions. However, these arrays should all store the same type of elements. To create such *ragged* arrays, we first need to declare the 1D array of arrays, and then proceed by declaring each individual array using a loop statement. For example, to declare and create a 2D ragged array of integers, we write the following statements:

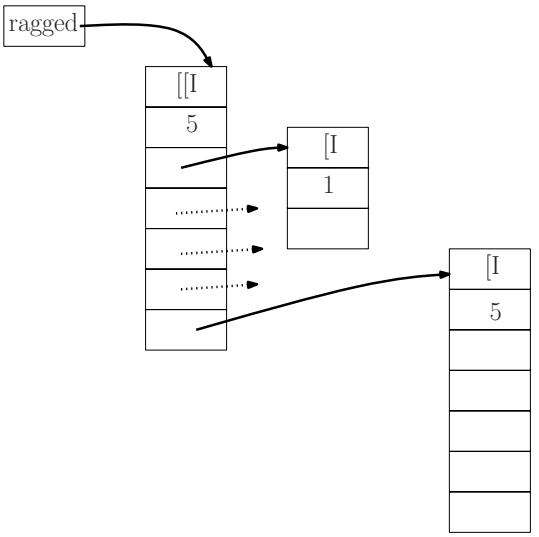


Figure 4.2 Visualizing the structure of a ragged array in the global memory

```
int ragged[][] = new int[5][];
for (int i = 0; i < 5; i++)
 {ragged[i] = new int[i + 1];}
```

The elements of the ragged arrays are either initialized by default (value zero) or by using nested loops as follows:

```
for (int i = 0; i < 5; i++)
{for (int j = 0; j < ragged[i].length; j++)
{        ragged[i][j] = (int)(10*Math.random());  // random init.
 }
}
```

Note that `ragged[i]` stores *references*[7] to linear arrays of integers. To visualize the entries of the ragged array, consider the following instructions:

```
System.out.println("type:"+ragged+" "+ragged.length);
 for (int i = 0; i < 5; i++)
  System.out.println("type:"+ragged[i]+" "+ragged[i].length);
```

[7] In general, the type of elements contained in the ragged array may be retrieved using `array.getClass();`

We get the following output:

```
type:[[I@addbf1 5
type:[I@42e816 1
type:[I@9304b1 2
type:[I@190d11 3
type:[I@a90653 4
type:[I@de6ced 5
```

Observe that array `ragged` is printed as a bi-dimensional array of integers using the "[[" notational convention: `[[I@addbf1`. Similarly to Figure 4.1, we can visualize ragged arrays as depicted in Figure 4.2.

4.6 Arrays of strings and `main` function

Strings of type `String` are *not* primitive types of Java. Though they can be constructed from literals and are immutable, strings are considered as special Java objects. Strings are *not* arrays of characters `char`. In other words,

$$\texttt{String} \neq \texttt{char []}.$$

These object notions shall be explained in the next chapter. We can also build arrays of strings that are of type `String []`, and functions may have string arrays as arguments. Actually, we are already very familiar with the `main` function of all Java programs that take as argument an array of strings:

```
class ProgramName
{
public static void main(String[ ] args)
{
...
}
}
```

For example, the following program lists all string arguments given in the line command when invoking the java virtual machine on the bytecode:

Program 4.13 Writing to the output the arguments of the invoked `main` function

```
class ParsingMain
{
  public static void main(String[] args)
  {
  for(int i=0;i<args.length;i++)
    System.out.println(i+":"+args[i]);
  }
}
```

After compiling this code, let us execute the bytecode using `java` as follows:

```
prompt%java ParsingMain Once upon a time there was a programming language
named Java!

0:Once
1:upon
2:a
3:time
4:there
5:was
6:a
7:programming
8:language
9:named
10:Java!
```

We can use the string array passed as argument of the `main` function of programs, to process inputs. Since these elementary inputs are considered as strings, we may eventually need to re-interpret them into the appropriate type before processing them. For example, consider the following program that seeks for the smallest integer entered in the arguments of the command line:

Program 4.14 Array of strings in `main`

```
class ParseArgumentsMin{
  public static void main(String[] args)
  {
    int indexMin=0;
    for(int i=1; i<args.length;i++)
    if (Integer.parseInt(args[indexMin])>Integer.parseInt(
        args[i]))
        indexMin=i;

    System.out.println("Maximum argument found at index:"+
        indexMin+" :"+args[indexMin]);
  }
}
```

Compiling and running this program with argument strings “345”, “546”, “234”, “2” and “45”, we get:

```
prompt%javac ParseArgumentsMinInt.java

prompt%java ParseArgumentsMin 345 546 234 2 45
Maximum argument found at index:3 :2
```

Once the strings are converted into corresponding integers using the library function `Integer.parseInt`, we get the index of the smallest argument: 2. Indeed `args[3]` corresponds to the string “2.”

4.7 A basic application of arrays: Searching **

Consider the following simple search problem encountered very day by in programmers: We are given a set $\mathcal{E}$ of n integers $\mathcal{E} = \{E_1, ..., E_n\}$, and we would like to know whether a given query element E belongs to that set or not: That is mathematically for short, $E \in \mathcal{E}$?. This search task is essential to decide whether we should add this element to the set or not. Let the data-structure for storing the n elements of $\mathcal{E}$ be an array named `array`.
The *sequential search* inspects in turn all the array elements `array[i]` and performs a comparison with the query element `E` to check for equality or not. If for a given index position i the query element matches the array element (that is, predicate `array[i]==E` is evaluated to `true`) then the element is found and the index of its position in the array is reported. Otherwise, we need to browse the full array before answering that `E` was not found in the array. This sequential search approach is summarized by the following program:

Program 4.15 Sequential search: Exhaustive search on arrays

```
class SequentialSearch{

static int SequentialSearch(int [] array, int key)
{int i;
for(i=0;i<array.length;i++)
  if (array[i]==key)
      return i;

return -1;
}

public static void main (String[] args)
{
int [] v={1,6,9 ,12 ,45, 67, 76, 80, 95};

System.out.println("Seeking for element 6: Position "+
    SequentialSearch(v,6));
System.out.println("Seeking for element 80: Position "+
    SequentialSearch(v,80));
System.out.println("Seeking for element 33: Position "+
    SequentialSearch(v,33));
  }
}
```

Running the program, we get the following output:

```
Seeking for element 6: Position 1
Seeking for element 80: Position 7
Seeking for element 33: Position -1
```

For query elements that are not present inside the array, we have to wait to reach the end of array to return -1. This explains why this sequential search is

also called the *linear* search since it takes time *proportional* to the array length. The algorithmic question raised is to know whether there exists or not a faster method? Observe that in the above program the array elements were ordered in increasing order. We should try to use this *extra* property to speed-up the search algorithm. The idea is to skip browsing some portions of the arrays for which we know that the query element cannot be found for sure. Start with a search interval [`left`, `right`] initialized with the extremal array indices: `left`$= 0$ and `right`$= n-1$ where n denote the array length `array.length`. Let `m` denote the index of the *middle* element of this range: $m = (\text{left} + \text{right})/2$. Then execute *recursively* the following steps:

- If `array[m]==E` then we are done, and we return index `m`,
- If `array[m] <E`, then if the element is inside the array, it is necessarily within range $[m+1, \text{right}]$,
- If `array[m] >E`, then if the element is inside the array, it is necessarily within range $[\text{left}, m+1]$.

The search algorithm terminates whenever we find the element, or if at some point `left>right`. In that latter case, we return index -1 for reporting that we did not find the query element. Thus the dichotomic search (also called binary search) is a provably *fast* method for searching whether or not a query element is inside a *sorted* array by successively halving the index range. The number of steps required to answer an element membership is thus proportional to $\log_2 n$. The dichotomic search is said to have *logarithmic time complexity*. These time complexity notions will be further explained in Chapter 6. We summarize the bisection search by the following code:

Program 4.16 Binary search: Fast dichotomic search on sorted arrays

```
class BinarySearch{
static int Dichotomy(int [] array, int left, int right, int
    key)
{
if (left>right)
    return -1;
int m=(left+right)/2;
if (array[m]==key)
  {return m;}
  else
  {
  if (array[m]<key) return Dichotomy(array,m+1, right, key);
    else  return Dichotomy(array,left,m-1, key);
  }
}
static int DichotomicSearch(int [] array, int key)
{return Dichotomy(array,0,array.length-1, key);}
public static void main (String[] args)
```

```
{
int [] v={1,6,9 ,12 ,45, 67, 76, 80, 95};
System.out.println("Seeking for element 6: Position "+
    DichotomicSearch(v,6));
System.out.println("Seeking for element 80: Position "+
    DichotomicSearch(v,80));
System.out.println("Seeking for element 33: Position "+
    DichotomicSearch(v,33));
  }
}
```

We get the following console output:

```
Seeking for element 6: Position 1
Seeking for element 80: Position 7
Seeking for element 33: Position -1
```

4.8 Exercises

Exercise 4.1 (Array of strings)

Write a static function `DisplayArray` that reports the number of elements in an array of strings, and displays in the console output all string elements. Give another function `DisplayReverseArray` that displays the array in reverse order.

Exercise 4.2 (Pass-by-value array arguments)

Explain why the following `permute` function does *not* work:

Program 4.17 Permuting strings and Java's pass-by-reference

```
class ExoArray{

static void permute(String s1, String s2)
{
String tmp=s1;
s1=s2;
s2=tmp;
}
public static void main(String args[])
{
String [] array={"shark", "dog", "cat", "crocodile"};
permute(array[0],array[1]);
System.out.println(array[0]+" "+array[1]);
}
}
```

Give a static function `static void permute(String [] tab, int i, int j)` that allows one to permute the element at index position i with

the element at index position j. Explain the fundamental differences with the former `permute` function.

Exercise 4.3 (Searching for words in dictionary)

Consider a dictionary of words stored in a plain array of strings: `String [] dictionary`. Write a function `static boolean isInDictionary` that takes as argument a given word stored in a `String` variable, and report whether the word is already defined inside the dictionary or not. Explain your choice for performing equality tests of words.

Exercise 4.4 (Cumulative sums: Sequential and recursive)

Write a function that takes a *single* array argument of elements of type `double`, and returns its cumulative sum by iteratively adding the elements altogether. Computing the cumulative sum of an array can also be done *recursively* by using, for example, the following function prototype `CumulativeSumRec(double array, int left, int right)`. Implement this function and test it using `CumulativeSumRec(array, 0, array.length-1);`

Exercise 4.5 (Chasing bugs)

The following program when executed yields the following exception:

```
Exception in thread "main" java.lang.NullPointerException
        at BugArrayDeclaration.main(BugArrayDeclaration.java:8)
```

Program 4.18 Bug in array declaration

```
class BugArrayDeclaration
{
public static void main(String [] t)
{
int [] array;
int [] array2=null;
array=array2;
array[0]=1;
}
}
```

Find the bug and correct the program so that it runs without any bug.

Exercise 4.6 (Sieve of Eratosthenes)

One wants to compute all prime integers falling within range $[2, N]$ for a prescribed integer $N \in \mathbb{N}$. The sieve of Eratosthenes algorithm uses a boolean array to mark prime numbers, and proceeds as follows:

– First, the smallest prime integer is 2. Strike off 2 and all multiples of 2 in the array (setting the array elements to `false`),
– Retrieve the smallest remaining prime number p in the array (marked with boolean `true`), and strike off all multiples of p,
– Repeat the former step until we reach at some stage $p > \sqrt{N}$, and list all prime integers.

Design a function `static int[] Eratosthene(int N)` that returns in an integer array all prime numbers falling in range $[2, N]$.

Exercise 4.7 (Image histogram)

Consider that an image with grey level ranging in $[0, 255]$ has been created and stored in the regular bi-dimensional data-structure `byte [] [] img;`. How do we retrieve the image dimensions (width and height) from this array? Give a procedure that calculates the histogram distribution of the image. (*Hint:* Do not forget to perform the histogram normalization so that the cumulative distribution of grey colors sums up to 1.)

Exercise 4.8 (Ragged array for symmetric matrices)

A d-dimensional symmetric matrix M is such that $M_{i,j} = M_{j,i}$ for all $1 \leq i, j \leq d$. That is, matrix M equals its transpose matrix: $M^T = M$. Consider storing only the elements $M_{i,j}$ with $d \geq i \geq j \geq 1$ into a *ragged* array: `double [] [] symMatrix=new double [d][];`. Write the array allocation instructions that create a 1D array of length i for each row of the `symMatrix`. Provides a static function that allows one to multiply two such symmetric matrices stored in "triangular" bi-dimensional ragged arrays.

*Exercise 4.9 (Birthday paradox **)*

In probability theory, the birthday paradox is a mathematically well-explained phenomenon that states that the probability of having at least two people in a group of n people having the same birthday is above $\frac{1}{2}$ for $n \geq 23$. For $n = 57$ the probability goes above 99%. Using the `Math.random()` function and a `boolean` array for modeling the 365 days, simulate the birthday paradox experiment of having at least two people having the same birthday among a set of n people. Run this birthday experiment many times to get empirical probabilities for various values of n. Then show mathematically that the probability of having at least two person's birthdays falling the same day among a group of n people is exactly $1 - \frac{365!}{365^n(365-n)!}$.

5
Objects and Strings

5.1 Why do programmers need objects?

We have so far presented the basic primitive types (say, `boolean`, `int` and `long`, `float` and `double`) of Java and explained how to build homogeneous arrays of these primitive types. Objects are useful for two main reasons:

– (1) Objects allow one to encapsulate and structure a set of data,

– (2) Objects provide a set of functions called *methods* acting on the encapsulated data.

For example, we would like to create objects for storing dates and write an agenda program that manipulates these "date" objects. A date can be defined as a triplet of numbers, say MM/DD/YYYY for storing respectively the month, day and year of the considered date. This triplet of numbers represents the core data-structure of dates. The object-oriented programming framework allows one to define and work on complex entities by encapsulating the various concepts at the *object level*. For example, we may like to define the following object entities:

– A data-structure for manipulating 2D colored points,

– A data-structure for manipulating students.

The various data encapsulated in objects are called *records*, and the variables to access them are called *fields*. For a 2D colored point object, we may wish

F. Nielsen, *A Concise and Practical Introduction to Programming Algorithms in Java*, Undergraduate Topics in Computer Science, DOI 10.1007/978-1-84882-339-6_5,

to dispose of the point coordinates (x, y) and a triplet red/green/blue, say, for encoding its color properties (R, G, B). For a student object, we may rather define the following fields: Lastname, firstname, group, etc. Java is such an object-oriented (OO) programming language.
We first start by presenting how to define and create objects, and then describe how to write *non-static* functions acting on the object's records: methods. Furthermore, we shall see how to create arrays of non-primitive elements: arrays of objects. Finally, we will present the `String` class, which is a particular class standardized in Java. We will quickly overview a set of methods that that class offers.

5.2 Declaring classes and creating objects

To create objects, we first need to declare them by creating a new *class* for structuring them. Objects are also typed structures: The class name defines the type of objects. Once the object records are clearly identified in the design stage, we create corresponding variable fields in the class by defining variables without using the keyword `static`. Fields are also called *object variables* and do not have the leading keyword `static`. Thus to create an object for storing dates, we first define the corresponding class `Date`, which contains the fields `dd`, `mm` and `yyyy` for storing respectively the day, month and year of the date. The class declaration for `Date` is as follows:

```
class Date
{
int dd;     // field for day (no static keyword here!)
int mm;    // field for month
int yyyy; // field for year
}
```

Let us suppose that we created an object `day` of type `Date`. Then the variable `dd` of `Date` (its record) and can be accessed by the programmer by writing `day.dd`: That is, the record `dd` of the object `day`. Similarly, the month and year object records are accessed by writing `day.mm` and `day.yyyy`.
An object storing records of students can be similarly created from a corresponding class `Student` that encapsulates the various data attached to a student identity. Fields may not be all of primitive types. For example, we may also attach to a student object an array of `double` for storing his/her exam marks. Thus we may come up with the following class structure for defining the class `Student`:

```
class Student
{
```

```
String Lastname;
String Firstname;
int Company;
double [ ] Marks;
...
}
```

5.2.1 Constructor and object creation

To use an object, we first need to *create* it by instantiating the various fields of the class by a special method called the *constructor*. We build an object using the `new` keyword that will call the constructor method of the class. A constructor is an object method bearing the class name. A constructor as all other object methods is a *non-static* function. To access a record of an object inside the current object, we use the keyword `this`. Thus `this.field` will return the value of the `field` variable of the *current* object. The program below shows how to define a class `Date` with its constructor:

Program 5.1 A class for storing dates with a constructor method

```
class Date
{
int dd;
int mm;
int yyyy;
// Constructor
Date(int day, int month, int year)
{this.dd=day;
this.mm=month;
this.yyyy=year;}
}
```

An object `day` of type `Date` is created by the following instruction:

```
Date day=new Date(23,12,1971);
```

The various records of that date object are accessed by `day.dd`, `day.mm` and `day.yyyy`. The program below shows how to put altogether the newly created class `Date` for encapsulating a date with the class program that creates an object of this type and displays its various records:

Program 5.2 A small demonstration program using the `Date` class

```
class Date
{
int dd;
int mm;
int yyyy;
```

```
// Constructor (method)
Date(int day, int month, int year)
{
this.dd=day;
this.mm=month;
this.yyyy=year;
}

}

class DemoDate{
public static void main(String[] args)
{
Date day=new Date(23,12,1971);
System.out.println("Date:"+day.dd+"/"+day.mm+"/"+day.yyyy);
}
}
```

A class may potentially supply *several constructors* provided that they all bear different signatures. For now, it is best to define a single constructor that initializes all the fields of the object. Fields of the current object are accessed by using the keyword `this`. Although it is not mandatory to use it explicitly inside the constructor, we recommend initializing the object records in the constructor by using `this`:

```
Date(int day, int month, int year)
{this.dd=day;
this.mm=month;
yyyy=year; // we omitted the keyword this here
}
```

Note that in cases where no constructor is provided, Java will use a default constructor common to all objects. Fields can still be assigned after the object creation as follows:

```
Date day=new Date();// default constructor called
day.yyyy=1971;
...
```

We can visualize the object creation and its relationship as depicted in Figure 5.1.

5.2.2 The common `null` object

Similar to arrays, we can first define object variables and create/initialize them later. For example, consider the following code:

```
Date day;
...// do some instructions here
Date day=new Date(23,12,1971);
```

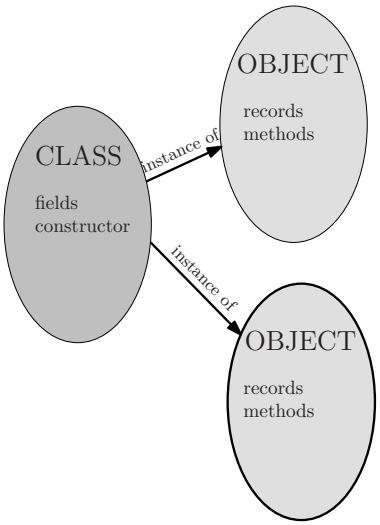


Figure 5.1 Visualizing the class fields and the object records when creating object instances of a class

Thus it might be the case that some instructions try to access the records of an object that is not yet created or initialized. This will raise an exception `nullPointerException` and provoke an abnormal program termination. Indeed, objects not yet created have by default value `null`. The `null` object is common to all classes and is used as the default value of object variables not yet created. To avoid `nullPointerException` exception, it is therefore recommended to *test* whether the object is `null` or not, as follows:

```
Student stud=null;
...
if (stud!=null)
stud.group=2;
```

5.2.3 Static (class) functions with objects as arguments

Objects can also be parameters of functions. That is, a static function may have both primitive and non-primitive type objects. The generic prototype of a function having object arguments is:

```
static TypeF F(Object1 obj1, ..., ObjectN objN)
```

For example, we may declare the following class function `static boolean isBefore (Date d1, Date d2)`. Functions may also return an object as a result as in `static Date readDate()`. The program below demonstrates these flexibilities:

Program 5.3 Objects as function parameters and returned results

```
class Date
{
int dd;
int mm;
int yyyy;

static final String[ ] months={
  "January", "February", "March", "April", "May",
   "June", "July", "August", "September", "October",
    "November", "December"
  };

// Constructor
Date(int day, int month, int year)
{
this.dd=day;
this.mm=month;
this.yyyy=year;
}

}
```

Observe the construction of a *constant array* of strings `months` using the syntax `static final String []`. The keyword `public` can be omitted in this textbook (except for the `main` program class function) since we do not deal with object inheritance, another important concept of object-oriented programming languages.

Program 5.4 Testing the `Date` class

```
class TestDate{
static void Display(Date d){
  System.out.println("The "+d.dd+" "+Date.months[d.mm-1]+" of
       "+d.yyyy);
  }

static boolean isBefore(Date d1, Date d2)
{
boolean result=true;
if (d1.yyyy>d2.yyyy) result=false;
if (d1.yyyy==d2.yyyy && d1.mm>d2.mm) result=false;
if (d1.yyyy==d2.yyyy && d1.mm==d2.mm && d1.dd>d2.dd) result=
    false;
return result;
}

public static void main(String[] args)
{
Date day1=new Date(23,12,1971);
Display(day1);
Date day2=new Date(23,6,1980);
System.out.println(isBefore(day1,day2));
```

```
}
}
```

5.3 Objects and references

Variables of a given object type are allocated into the *global* memory, and references to these memory structures are stored into the variables. This is to contrast with variables of primitive types that are manipulated by *value*. Similar to array variables, a variable of type object is a *reference* to that object. That is, loosely speaking, the variable stores the memory address of this referenced object. Thus when we write:

```
Date day1=new Date(23,12,1971);
Date day2=day1;
Display(day2);
day2.mm=6;
Display(day1);
```

We get:

```
Date:23/12/1971
Date:23/6/1971
```

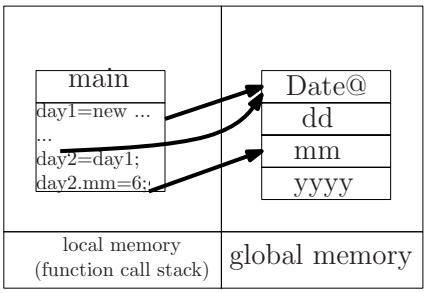


Figure 5.2 Objects are non-primitive typed structures that are stored in the program global memory and manipulated by references (and not by values)

The date `day1` is *not* copied to `day2`, but only their references are copied. That is, the reference of `day2` is assigned to the reference of `day1` so that their object records necessarily match since they refer to the same memory location as depicted in Figure 5.2.

5.3.1 Copying objects: Cloning

To copy or clone an object to another one, we need to do it *field-wise*. Otherwise we will copy only the references (as it was the case for `day2=day1`). There are two scenarii, depending on whether the second object has already been created (formerly using the keyword `new`) or not:

Program 5.5 Cloning objects: Two scenarii

```
// Two Scenarii:
// Here, we assume that day2 has already been created...
day2.dd=day1.dd;
day2.mm=day1.mm;
day2.yyyy=day1.yyyy;

// day2 object has not yet been created...
static Date Copy(date day1)
{
Date newdate=new Date (day1.dd,day1.mm,day1.yyyy);
return newdate;
}
...
Date d2=Copy(d1);
```

Copying verbatim all records of one object to another object is also called *deep copying*. Copying just their reference is *shallow copying*.

5.3.2 Testing for object equality

When writing programs, we often need to *test* for object equality. Do not use the regular syntax `==` for object equality since it will only compare their references. Of course, if references match then their fields also necessarily match. But the general case is to have two objects created with their respective fields, for which we would like to test for equality, field by field. Thus to physically compare objects, we need to define and use a *tailored predicate* that checks the objects field by field. For example, considering the `Date` objects, we may design the following predicate:

Program 5.6 Predicate for checking whether two dates of type `Date` are identical or not

```
static boolean isEqual(Date d1, Date d2)
{
return (d1.dd == d2.dd &&
        d1.mm == d2.mm &&
        d1.yyyy== d2.yyyy);
}
```

Here is a test program for using the `isEqual` predicate:

```
public static void main(String[] args)
{
Date day1=new Date(23,12,1971);
Date day2=day1; // beware not copying here: shallow copying
Date day3=new Date(23,12,1971);
System.out.println(isEqual(day1,day3));
System.out.println(day1);
System.out.println(day2);
System.out.println(day3);
}
```

Running this program, we get the following console output:

```
true
Date@3e25a5
Date@3e25a5
Date@19821f
```

This shows that `day1` and `day3` are identical (that is, they match field by field) although their references are different. Figure 5.3 illustrates the object creations: Their references are stored into the local stack function and the objects are stored into the global (persistent) memory.

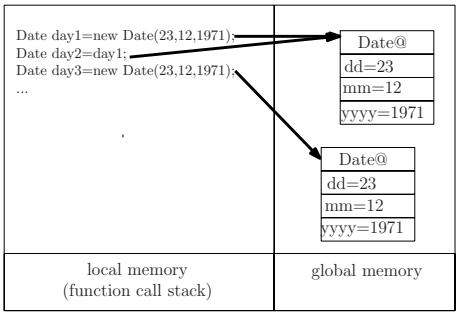


Figure 5.3 Testing for object equality using a tailored predicate that compares field by field the objects.

5.4 Array of objects

Since newly created classes also define new corresponding types, we can create an array of objects. For example, to create an array of objects of type `Date`, we use the following syntax:

```
Date [ ] tabDates=new Date[31];
```

When an array of objects is built, all its elements `Date[i]` are initialized to the `null` object.[1] Let us create a new kind of object for storing "events" that are annotated dates, by defining the class `XEvent` as follows:

```
public class XEvent
{
Date when;
String what;

// Constructor
public XEvent(Date d, String text)
{
this.when=d;
this.what=text;
}
}
```

We can now write a static class function `static XEvent oldest(XEvent[] tab)` attached to the class `TestXEvent` for seeking in an array and reporting the oldest event as follows:

Program 5.7 The class `XEvent` and arrays of objects

```
class TestXEvent
{static void Display(XEvent e)
  { System.out.print(e.what+": ");
    e.when.Display(); }

   static boolean older(XEvent e1, XEvent e2)
  {return Date.isBefore(e1.when,e2.when);}

 static XEvent oldest(XEvent[] tab)
  {
    XEvent result=tab[0];
    for(int i=1;i<tab.length;++i)
      if (older(tab[i],result)) result=tab[i];
    return result;
  }

  public static void  main(String [] args)
  {
    Date d1=new Date(26,6,2003);
    XEvent e1=new XEvent(d1,"Birthday Julien");
    Date d2=new Date(20,11,2000);
    XEvent e2=new XEvent(d2,"Birthday Audrey");
    Date d3=new Date(23,6,1971);
    XEvent e3=new XEvent(d3,"Birthday Me");
    Display(e1);
    XEvent [] tabEvent=new XEvent[3];
    tabEvent[0]=e1;
```

[1] This can be interpreted as the equivalent of the zero initialization of primitive types.

```
    tabEvent[1]=e2;
    tabEvent[2]=e3;
    System.out.print("Oldest person::");
    Display(oldest(tabEvent));
    }
}
```

Running this program yields the following console output:

```
Birthday Julien: 26 June 2003
Oldest person::Birthday Me: 23 June 1971
```

5.5 Objects with array members

Records of objects may also contain arrays. These arrays should always be built by the object constructor as follows: `new Type[sizeArrayExpression]`. Thus it is not necessary at compile time to know the array sizes provided that at run time the expression `sizeArrayExpression` can be evaluated to an integer value. In the chapter introduction, we mentioned that for student objects, we would like to store their grades using an array of `double`. Another example is a data-structure for polynomials that may be created using the following class:

```
class Polynome{
int degree;
double [ ] coefficients; // Array declaration
};
```

5.6 The standardized `String` objects

5.6.1 Declaring and assigning `String` variables

In Java, a string of characters is manipulated with an object of type `String`. The `String` type does *not* belong to the primitive types (`boolean`, `int`, `double`, etc.), but is rather a non-primitive object type. Therefore, a variable of type `String` is a reference to that object in the global memory. Thus for the same reasons as for the case of integers, the following `badSwap` function will not swap the string variables since once the function is executed the values of `p` and `q` (that is, the respective references) are kept unchanged: Java is pass-by-value/reference only. For non-primitive types such as arrays, objects and strings, the stored value is a reference.

```
static void badSwap(String p, String q)
```

```
{ String tmp;
tmp=p;
p=q;
q=tmp;}
```

To declare and initialize a string[2], we proceed as follows

```
String title= "First Hands-on Programming!";
String anotherTitle=title;
...
title="Extreme programming sounds better!"; // String can be modified
```

To print a string, use the regular `System.out.println()` function:

```
System.out.println(title);
```

Java proposes many standard functions for efficiently manipulating strings. Let us next review only the very fundamental functions that operate on `String` objects.

5.6.2 Length of a string: `length()`

The length of a string object is defined as its number of characters. The length of a string is reported by calling the `length()` *method* on the `String` object, as shown below:

```
String s= "Supercalifragilisticexpialidocious";
System.out.println(s.length());
```

We get 34, which is the longest *invented* word in a song from *Mary Poppins* written by the Sherman Brothers in 1964. Observe that the syntax is different from arrays. For an array `array`, we get its length (that is, its number of elements) by the following syntax: `array.length`. But for a string, we use the *method* `length()` (with no argument, which explains the () parentheses).

5.6.3 Equality test for strings: `equals(String str)`

To test whether two strings `s1` and `s2` are identical or not, do not use the regular `==` equality test for primitive types. Indeed testing whether two strings are identical or not by using the comparison `==` will check whether the references of these strings are identical or not. Instead, to test whether the contents of

[2] Use `String` objects only for moderate length strings, otherwise use the better class `StringBuffer`. See Java API at `http://java.sun.com/j2se/1.4.2/docs/api/java/lang/StringBuffer.html`

the two strings are identical or not, use the method `equals(str)` with a string argument `str`. Considers for example, the following sequence of instructions:

```
String s1="java";
String s2="JAVA";
System.out.println(s1.equals(s2));
```

We get `false` on the console output since Java is *case-sensitive*: That is, upper cases are considered different from lower cases in Java.
We can access the character of the string at position $k-1$ by calling the method `charAt(k)`. Again, indexes of characters range from 0 to the string length minus one. For example, consider the following code:

```
String s= "3.14159265";
System.out.println(s.charAt(1));
```

Then the executed program writes the separation "." on the console. We can assign a character variable with that string character as follows:

```
char c=s.charAt(1);
```

5.6.4 Comparing strings: Lexicographic order

Before comparing strings of various lengths, we first need to define what is meant by the comparison of two elementary characters. The "distance" between two characters is defined as the span in the ASCII[3] code table. Since Java is case-sensitive, lower cases are different from upper cases (they have different integer codes), and the same character appears twice in the ASCII table: once for the upper case and once for the lower case. The following code demonstrates the lower/upper case-sensitivity of Java:

Program 5.8 Lower/upper cases and ASCII codes of characters

```
char c1,c2;
c1='a';
c2='z';
// Compare character code
if (c1<c2)
{System.out.println(c1+" is before "+c2);}
else
{System.out.println(c1+" is after or equal to "+c2);}
int codec1=c1; // type casting/conversion
int codec2=c2; // type casting conversion
System.out.println("Code ASCII for "+c1+":"+codec1);
System.out.println("Code ASCII for "+c2+":"+codec2);
```

[3] American Standard Code for Information Interchange (ASCII). See `AmericanStandardCodeforInformationInterchange(ASCII)`. In fact, Java encodes characters using two bytes for encoding many different language character sets and kanjis. See UNICODE at `http://en.wikipedia.org/wiki/Unicode`

Running this code, we get

```
a is before z
Code ASCII for a:97
Code ASCII for z:122
```

One can convert a lower-case string to an upper-case string by performing the following *arithmetic* operations on characters (with `char`⇔`int` casting):

Program 5.9 Lower-case to upper-case string conversion

```
static String LowerToUpper(String s)
{
String result="";
char c;
for(int i=0;i<s.length();i++)
{
c=(char)(s.charAt(i)-32); //32=2^5
result+=c; //concatenation, append c to result
}
return result;
}
...
String s=LowerToUpper("convert a simple sentence");
```

To compare string s_1 with string s_2, we first need to define a *total order* between these strings so that we can report whatever the input strings whether

- $s_1 < s_2$,
- $s_1 = s_2$ or
- $s_1 > s_2$.

This total order is called the *lexicographic order* and is defined for strings as follows:

- If one string is a *substring* of another one, then report the length difference: Positive or negative number. Zero if and only if the strings are exactly the same.
- Otherwise, there necessarily exists an index where the two string characters differ. Report then the difference in ASCII code of these two different characters.

To lexicographically compare two strings in Java, use the method `compareTo(str)` method as follows:

```
String u="Polyhedron", v="Polyhedral";
System.out.println(u.compareTo(v));
// => 14= 'o'-'a'=111-97;
```

p o l y h e d r o n
p o l y h e d r a l

```
System.out.println("o:"+(int)'o');
System.out.println("a:"+(int)'a');
int diff='o'-'a'; // implicit casting char->int
System.out.println(diff);
```

We get:

```
o:111
a:97
14
```

Now consider the case of one string being the substring of another one:

```
String a="polyhedral",b="polyhedralization";
System.out.println(a.compareTo(b));
System.out.println(a.length()-b.length());
```

We find both times -7. This is the difference between the length of string `a` and the length of string `b`.
We can implement our own static function for performing the lexicographic order on strings as follows:

Program 5.10 Implementation of the lexicographic order on strings

```
static int LexicographicOrder(String p, String q)
{
int i=0;
while(i<p.length() && i<q.length())
{if (p.charAt(i)==q.charAt(i))
i++;
else
return p.charAt(i)-q.charAt(i);
}
return p.length()-q.length();
}
```

As a final example of the use of lexicographic order of strings in programs, let us play again with the string array argument of the `main` function (`public static void main(String[ ] args)`). The following program finds among the string arguments of the `main` function the lexicographically *smallest* string:

Program 5.11 Reporting the lexicographically minimum string of the command line arguments

```
class ParsingArgument{
  public static void main(String[] args)
  {
    String minimum=args[0];

    for(int i=1; i<args.length;i++)
```

```
        if (minimum.compareTo(args[i])>0)
          minimum=args[i];

      System.out.println("Lexicographically minimum string is:"
          +minimum);
    }
}
```

For example, calling the above compiled program with arguments `rock doll dance dancing`, we get:

```
prompt%java ParsingArgument rock doll dance dancing
Lexicographically minimum string is:dance
```

5.7 Revisiting a basic program skeleton

Let us close this chapter by writing a small skeleton program that defines a *new* class and uses this class in the `main` function of the *program* class:

Program 5.12 A more evolved basic skeleton program that also defines classes

```
class Point
{
int x,y;
Point(int xx, int yy){x=xx;y=yy;}
} // end of class Point
class Skeleton
{
// Static class variables
static int nbpoint=0;
static double x;
static boolean [] prime;
static int f1(int p){return p/2;}
static int f2(int p){return 2*p;}

public static void Display(Point p)
{System.out.println("("+p.x+","+p.y+")");}

public static void main(String [] argArray)
{
System.out.println(f2(f1(3))+" versus (!=) "+f1(f2(3)));
Point p,q;
p=new Point(2,1); nbpoint++;
q=new Point(3,4); nbpoint++;
Display(p);
Display(q);
} }
```

Compiling and running this program, we obtain the following console output:

```
2 versus (!=) 3
(2,1)
(3,4)
```

5.8 Exercises

Exercise 5.1 (A class for storing supermarket product items)

In a supermarket, each product is typically labeled with:

– (1) a string depicting the product name,

– (2) a price tag, and

– (3) a date of validity that informs when the product expires.

Write a class `Date` for storing dates with a constructor that initializes these various fields. Write another class `Product` for storing information related to products that use the class `Date`. Store a collection of product items into an array of `Product` elements. Given a product query element `Element`, write a function that checks whether the product is inside the array or not. The equality test shall be defined by using the product strings only. Write a function that, given a price range `low` and `high`, reports all product items with prices falling inside this range. Finally, describe a function that takes as its argument a given date, and report all elements of the array expiring before this date.

Exercise 5.2 (Lexicographic order on 2D points)

Write a class `Point2D` for storing the (x, y) coordinates of points. We would like to totally order points so that given two points $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$, $P_1 < P_2$ if and only if $x_1 < x_2$ or $x_1 = x_2$ and $y_1 < y_2$. Write a predicate function `static boolean LessThan(Point2D P1, Point2D P2)` that implements this lexicographic order. Consider now an array of 2D points. Give a function that reports the smallest point defined with respect to that lexicographic order. Extend these functions to 3D point sets stored in an array.

Exercise 5.3 (Lexicographic order for the Olympic games)

Design a class `Medals` for storing the number of gold, silver and bronze medals of a given country. Provide a constructor for that class. Define then a lexicographic order on two objects `a` and `b` of type `Medals` as follows: `a<b` if and only if:

– the number of gold medals of `a` is less than the number of gold medals of `b`, or

– the number of gold medals are identical but the number of silver medals of `a` is less than the number of silver medals of `b`, or

– the number of gold and silver medals are identical but the number of bronze medals of `a` is less than the number of silver medals of `b`.

Exercise 5.4 (A class for polynomials)

Consider defining a class for storing polynomials of arbitrary degree as follows:

Program 5.13 A class for polynomials

```
class Polynomial
{
int degree;
long [] coefficient;
...
}
```

Observe that a polynomial of degree d has $d + 1$ coefficients. Provide a constructor `Polynomial(int deg)` for that class that initializes all `deg+1` coefficients to zero. Then design and implement the following *static* functions:

– A clone function `static Polynomial copy(Polynomial source);`

– A display procedure `static void display(Polynomial source);`

– An equality predicate:
`static boolean equalTo(Polynomial P, Polynomial Q);`

– A function `static Polynomial derivative(Polynomial source);` that computes the derivative of a polynomial.

– A function `static Polynomial add(Polynomial P, Polynomial Q);` that adds two polynomials. Similarly, a function that subtracts two polynomials:
`static Polynomial subtract(Polynomial P, Polynomial Q);`

– A function `static long evaluate(Polynomial source, long x);` that evaluates the polynomial at value `x`. What is the time complexity of this function? Horner proposed a scheme for evaluating polynomials using only d multiplications and d additions using the following schema:

$$P(x) = (...((p_d x + p_{d-1})x + p_{d-2})x + ...)x + p_0$$

Implement the Horner evaluation scheme in function `static long HornerScheme(Polynomial source, long x);`

– A function `static Polynomial multiply(Polynomial P, Polynomial Q);` for multiplying[4] two polynomials. What is the time complexity of that polynomial multiplication?

Exercise 5.5 (Chevalier de Méré's probability games)

During the 17th century, gambler de Méré asked Blaise Pascal an explanation for his game losses. His question was solved by Pascal and Fermat, and yielded the foundations of the theory of probability. Gamblers used to bet on the event of getting at least one ace in four rolls of a dice. As a game variation, De Méré proposed to use two die and roll them 24 times with a bet on having at least one double ace. De Méré thought the games were equivalent but he lost consistently. Explain why? Write a simulation program to empirically determine which one dice/two die game is more likely to win.

Use the following classes:

```
import java.util.Random;

class Dice
{
Random rand;

Dice()
{ rand=new Random();}

// Return the elementary outcome of tossing a dice: A
    face between 1 and 6
int Toss()
{return 1+rand.nextInt(6);}
}

// One dice, four rolls
class DeMereGame1
{
Dice dice;

DeMereGame1()
{dice=new Dice(); }

// Return true if the gamer win and false otherwise
boolean Experiment()
{int i;
```

[4] Note that we can perform faster multiplication of polynomials using the Karatsuba algorithm ($O(n^{1.585})$ time complexity) or the FFT algorithm ($O(n \log n)$-time).

```
for(i=1;i<=4;i++)
  {if (dice.Toss()==6) return true; // win
  }
return false; // loose
}
}

class DeMereGame2
{
Dice dice1, dice2;

DeMereGame2()
{dice1=new Dice();dice2=new Dice();}

boolean Experiment()
{
int i;

for(i=1;i<=24;i++)
  {
  if ((dice1.Toss()==6)&&(dice2.Toss()==6)) return true;
       // true
  }

return false; // loose
}
}
```

6
Searching and Sorting

6.1 Overview

We have quickly sketched in Chapter 4.7 the sequential and dichotomic/bisection search strategies to answer *element membership* queries: Does an element already belong to the data-set or not? In this chapter, we further describe the *generic* framework for inserting/deleting or modifying attributes of elements of a data-set. We revisit the linear sequential and logarithmic bisection searches on sets of complex data-structured elements: sets of objects. Since the bisection search requires arrays to be totally ordered, and since raw arrays of elements are very unlikely to be already sorted beforehand, we then present two methods for sorting data-sets:

– (1) The *selection sort*, which iteratively selects the current smallest element of remaining sub-arrays, and

– (2) The *quicksort*, which recursively sorts the arrays by partitioning elements.

Finally, to conclude this chapter, we present the hashing method that is often used in practice to find elements in almost constant time, and summarize the time complexity of these various methods.

F. Nielsen, *A Concise and Practical Introduction to Programming Algorithms in Java*, Undergraduate Topics in Computer Science, DOI 10.1007/978-1-84882-339-6_6,

6.2 Searching information

When organizing data into objects by defining their records, we access the various object attributes by defining a key and additional fields. That is, complex objects are accessed via their corresponding *keys* that represent handles. We suppose that our data-sets fit the local volatile memory so that all objects can be loaded in the main memory without external input/output. Further, we first suppose *static* data-structures that are not updated. We will later consider the more challenging problem of adding/removing or changing object attributes dynamically.
To fix ideas, consider searching for a word in a dictionary. The basic container for storing information is the object that consists of a pair of "word" with its corresponding "definition." In Java, we thus declare the following class that contains the two fields: `word` and `definition`:

```
class DictionaryEntry
{
String word;
String definition;
}
```

To create a dictionary, we first create an array of `DictionaryEntry` elements, and then assign to each element an entry of the dictionary. The following program illustrates this process:

Program 6.1 Structuring data of a dictionary into basic object elements

```
class DictionaryEntry
{
String word;
String definition;
DictionaryEntry(String w, String def)
{this.word=new String(w); // Clone the strings
this.definition=new String(def);}
}

class TestMyDictionary
{public static void main(String[] args){
DictionaryEntry [] Dico =new DictionaryEntry[10];
Dico[0]=new DictionaryEntry("Java","A modern object-oriented
    programming language");
Dico[1]=new DictionaryEntry("C++","An effective object-
    oriented programming language");
Dico[2]=new DictionaryEntry("FORTRAN","FORTRAN stands for
    FORmula TRANSlation. Often used for simulation.");
//...
}
}
```

In practice, elementary data can be arbitrary complex. For example, consider representing an individual. In that case, the key (handle) of objects can be either the lastname or the firstname, and additional fields could be the address, phone number, age, etc. Thus all information characterizing an individual can be modeled by a class with the following set of attributes:

```
class Individual {
String Firstname;
String Lastname;

String Address;
String Phonenumber;

int Age;
boolean Sex;
}
```

Let us take a last geometric example for storing points. Each object denotes then a 2D point, and the key can be the name attached to that point or the x or y coordinates, etc. Further, additional information fields may be the point color attribute, its name, etc. A class for storing points can thus be modeled as:

```
class Color {int Red, Green, Blue;}
class Point2D
{
double x;
double y;
String name;
Color color;
}
```

6.3 Sequential search

Once the class for storing object records has been designed, we consider representing a set of objects by an array of such objects. Objects are stored in the array in any order. That is, the set of objects stored in the array is not sorted into any particular order. To seek whether a given query object is inside the array or not, we simply browse the array sequentially until we find the object, or we report that it has not been found inside the array. This explains why the sequential search is also called the linear search since in the worst-case we need to browse all the array elements to report that the query object was not found. The basic primitive operation is to compare the key of the query element with the respective keys of objects of the array. Let us consider a listing of people, each person modeled by the following class:

```
class Person{
String firstname, lastname; // key for searching
String phone; // additional fields go here

// Constructor
Person(String l, String f, String p)
{this.firstname=f; this.lastname=l; this.phone=p;}
};
```

Let us consider the keys of objects as the `lastname` and `firstname` attributes, and report the phone number record of the object in case the object is found in the array. The sequential search algorithm is implemented by the following `LinearSearch` function:

Program 6.2 Linear search on objects

```
static Person[] array=new Person[5];

static String LinearSearch(String lastname, String firstname)
  {
  for(int i=0;i<array.length;i++)
  {
    if ((lastname.equals(array[i].lastname) &&
       (firstname.equals(array[i].firstname))))
          {return array[i].phone;}
  }
  return "Not found in my dictionary";
}
```

Observe that the predicate for checking whether the keys of the current object with the given query is not a single `==` comparison test but rather uses the `String` method `equals` to check that the string contents are identical. The program below demonstrates the sequential search on an array of `Person` initialized. Observe that the array is defined as a `static` array so that its data are attached to the program class and can therefore be accessed from the various `static` functions of the class.

Program 6.3 A demonstration program using the `Person` object array

```
class LinearSearchProgram{
  static Person[] array=new Person[5];

  static String LinearSearch(String lastname, String
     firstname)
  {...}

  public static void main (String [] args)
  {
    array[0]=new Person("Nielsen","Frank","0169364089");
    array[1]=new Person("Nelson","Franck","04227745221");
    array[2]=new Person("Durand","Paul","0381846245");
    array[3]=new Person("Dupond","Jean","0256234512");
```

```
    array[4]=new Person("Tanaka","Ken","+81 3 1234 4567");

    System.out.println(LinearSearch("Durand","Paul"));
    System.out.println(LinearSearch("Tanaka","Ken"));
    System.out.println(LinearSearch("Durand","Jean"));
  }
}
```

Compiling and running this program, we get:

```
0381846245
+81 3 1234 4567
Not found in my dictionary
0199989796
```

6.3.1 Complexity of sequential search

How efficient is the `LinearSearch` function? Clearly the *best-case* is when the query object keys match the keys of the objects stored at the first position of the array (index 0). The worst-case is when the query object keys do not match the keys of all objects contained in the array. We then need to fully browse the array, and it takes time proportional to the number of elements in the array: its length. Let n denote the array size (`n=array.size`). Then we say that the worst-case requires time proportional[1] to n. In case the query object is not found, we thus spend *linear time*. Now consider that the query keys match some of the array elements. This element can be at index position 0 up to $n-1$ with even probability. In case the matching keys are found at index i, it takes time proportional to i. Thus a successful search requires *on average* a time proportional to $\frac{1}{n}\sum_{i=1}^{n} i = \frac{(n+1)n}{2n} = \frac{n+1}{2}$. Using the time complexity big Oh-notation, we say that the average complexity time of a successful sequential search is $O(n)$.

6.3.2 Dynamically adding objects

It is rarely the case that our data-sets are fixed once for all. They evolve with time as some of the people may leave or others join. We thus need to update the data-structure. Storing objects in arrays does not offer much flexibility. For adding new people to the data-set we can first create a large array and keep an indicator of the position of the last inserted individual in the array. To add new objects, we then add them to the free array cells and consequently

[1] In computer science, we use the notation $O(n)$ to denote this time complexity.

increment the indicator index of the current number of elements. This dynamic add operation code is summarized in the following listing:

Program 6.4 Adding new `Person` to the array

```
public static final int  MAX_ELEMENTS=100;
  static int nbelements=0;
  static Person[] array=new Person[MAX_ELEMENTS];

  static String LinearSearch(String lastname, String
      firstname)
  {
  for(int i=0;i<nbelements;i++)
  {
    if ((lastname.equals(array[i].lastname) &&
       (firstname.equals(array[i].firstname))))
       return array[i].phone;
  }
  return "Not found in my dictionary";
  }

  static void AddElement(Person obj)
  {if (nbelements<MAX_ELEMENTS)
    array[nbelements++]=obj; // At most MAX_ELEMENTS-1 here
    // nbelements is at most equal to MAX_ELEMENTS now
  }
```

Let us illustrate the insertion operations by the following code:

```
public static void main (String [] args)
{
AddElement(new Person("Nielsen","Frank","0169334089"));
AddElement(new Person("Nelson","Franck","04227745221"));
AddElement(new Person("Durand","Paul","0381846245"));
AddElement(new Person("Dupond","Jean","0256234512"));
AddElement(new Person("Tanaka","Ken","+81 3 1234 4567"));
System.out.println(LinearSearch("Durand","Paul"));
System.out.println(LinearSearch("Tanaka","Ken"));
System.out.println(LinearSearch("Durand","Jean"));
AddElement(new Person("Durand", "Jean","0199989796"));
System.out.println(LinearSearch("Durand","Jean"));
}
```

Running this code, we get the following console output:

```
0381846245
+81 3 1234 4567
Not found in my dictionary
0199989796
```

The main problems of handling a collection of objects by using array data-structures are:

– We need to know *a priori* the maximum number of people (so that we do not encounter an array overflow),

– We cannot remove inserted `Person` objects. The array data-structure is semi-dynamic.

Furthermore, note that in practice we need to check if a person already belongs to the array before eventually adding his/her records.

6.3.3 Dichotomy/bisection search

The dichotomy search is also called bisection search. The method was sketched in chapter 4.7. To perform a bisection search, we need to have an array *fully sorted* with respect to the object keys. The bisection search proceeds as follows: Start with a search interval [`left`, `right`] initialized with `left`= 0 and `right`= $n-1$ where n denote the array length `array.length`. Let `m` denote the index of the middle element of this range: $m = (\text{left} + \text{right})/2$. Then execute the following recursive steps:

– If `array[m]=E` then we are done, and we return the index position m,

– If `array[m] <E`, then if the solution exists it is necessarily within range $[m+1, \text{right}]$,

– If `array[m]>E`, then if the solution exists it is necessarily within range $[\text{left}, m+1]$.

The search algorithm terminates whenever `left>right`. In that case, we return index -1 for signaling that we did not find the quey element. Thus a dichotomic search (also called binary or bisection search) is a fast method for searching whether a query element is inside a sorted array or not by successively halving the index range. The number of steps required to answer an element membership is thus proportional to $\log_2 n$. The dichotomic search is said to have *logarithmic* complexity.
The bisection search code is as follows:

Program 6.5 Bisection search on sorted arrays

```
static int Dichotomy(int [] array, int left, int right, int
    key)
  {
    if (left>right) return -1;

    int m=(left+right)/2; // !!! Euclidean division !!!

    if (array[m]==key) return m;
    else
    {
    if (array[m]<key) return Dichotomy(array,m+1, right, key)
        ;
```

```
        else  return Dichotomy(array,left,m-1, key);
    }
  }

static int DichotomicSearch(int [] array, int key)
  {
    return Dichotomy(array,0,array.length-1, key);
  }
```

For example, here is a demonstration program that creates a sorted array of integers and use the dichotomic search:

```
public static void main (String[] args)
  {
  int [] v={1, 6, 9, 12 , 45, 67, 76, 80, 95};

  System.out.println("Seeking for element 6: Position "+
      DichotomicSearch(v,6)); //1
  System.out.println("Seeking for element 80: Position "+
      DichotomicSearch(v,80));//7
  System.out.println("Seeking for element 33: Position "+
      DichotomicSearch(v,33)); //-1
  }
```

Running this program, the queries yield indices 1, 7 and -1, respectively. The last -1 is a code for signaling that the number was not found in the array.
The dichotomic search is exponentially more efficient than the linear search since the ratio of their time complexity is: $O(\frac{n}{\log_2 n})$. However to perform a dichotomic search we need to have sorted arrays. Since it is usually not the case, let us present now two common sorting methods: the selection sort and the so-called quicksort.

6.4 Sorting arrays

Given an unsorted array of elements, consider the task of sorting all its elements to get a sorted array in, say, increasing order. Formally, the only two primitive operations considered for sorting arrays are:

– comparisons and

– element swapping operations.

For example, these basic operations are implemented as follows for integers:

```
static boolean GreaterThan(int a, int b)
{return (a>b);}

static void swap (int [] array, int i, int j)
```

```
{
int tmp=array[i];
array[i]=array[j];
array[j]=tmp;
}
```

Sorting is an entire area field of computer science in itself with many algorithms and remaining challenges to tackle. Next, we present two flagship algorithms: The selection sort and quicksort.

6.4.1 Sorting by selection: `SelectionSort`

The selection sort is very natural and proceeds as follows:

- First, seek for the smallest element of the array: This is the SELECTION stage.
- Then, reiterate the minimum selection for the remaining sub-array:

```
array[1], ..., array[n-1]
```

That is, at stage i+1, the selection sort seeks for the smallest elements for the sub-array

```
array[j], array[j+1], ..., array[n-1]
```

To program the selection sort, we use two nested loops:

- The inner loop selects and puts the smallest element in front of the current array,
- The outer loop repeats the minimum selection/swap for all subarrays.

These steps are summarized in the source code below:

Program 6.6 Sorting by selecting iteratively the minimum elements

```
static void swap (int [] array, int i, int j)
  {int tmp=array[i];array[i]=array[j];array[j]=tmp;}

  static void SelectionSort(int [] array)
  {
  int n=array.length;

  for(int i=0;i<n-1;i++){
    for(int j=i+1;j<n;j++){
      if (GreaterThan(array[i],array[j]))
       swap(array,i,j);}
       }
  }
```

To use the selection sort routine, we first create an array of integers, call the static sort function, and then display the sorted array:

```
public  static void main(String[] args)
  {
    int [] array={22,35,19,26,20,13,42,37,11,24};

    SelectionSort(array);

    for(int i=0;i<array.length;i++)
      System.out.print(array[i]+" ");
      System.out.println("");
  }
```

We get the sorted array:

```
11 13 19 20 22 24 26 35 37 42
```

A more verbose output will display the status of the array at all stages:

```
Stage 1:11 35 22 26 20 19 42 37 13 24
Stage 2:11 13 35 26 22 20 42 37 19 24
Stage 3:11 13 19 35 26 22 42 37 20 24
Stage 4:11 13 19 20 35 26 42 37 22 24
Stage 5:11 13 19 20 22 35 42 37 26 24
Stage 6:11 13 19 20 22 24 42 37 35 26
Stage 7:11 13 19 20 22 24 26 42 37 35
Stage 8:11 13 19 20 22 24 26 35 42 37
Stage 9:11 13 19 20 22 24 26 35 37 42
```

Note that the sorting is performed *in-place*: That is, we do not need extra memory to sort the input array. Furthermore, the minimum elements are searched iteratively in sub-arrays: At stage $i + 1$, every time a new minimum is found, it is swapped with the head `array[i]` of the sub-array.

6.4.2 Extending selection sort to objects

Our implementation of the selection sort program works only for integer arrays. To extend it to arbitrary object types[2], we need to adjust the two basic primitives: predicate `GreaterThan` and swapping procedure `swap` to the type of considered object type. To illustrate how these changes may be performed, consider the following type `EventObject` for handling objects denoting events:

```
class EventObject
{
int year, month, day;
EventObject(int y, int m, int d)
```

[2] In fact, Java provides a framework called the *generics* for this purpose that is beyond the scope of this book.

```
{year=y; month=m; day=d;}

static void Display(EventObject obj)
{System.out.println(obj.year+"/"+obj.month+"/"+obj.day);}
}
```

Then the two primitives are redefined as follows:

Program 6.7 Redefining the predicate/swap primitives for sorting other types of elements

```
static boolean GreaterThan(EventObject a, EventObject b)
  {return (( a.year>b.year) ||
          ( (a.year==b.year) && (a.month>b.month)) ||
((a.year==b.year) && (a.month==b.month) && (a.day>b.day)) )
    ;}

static void swap (EventObject [] array, int i, int j)
{
EventObject tmp=array[i];
array[i]=array[j];
array[j]=tmp;
}
```

Observe once again that for non-primitive types, we design a tailor `>=` predicate since the usual predicate will only compare references on these objects. Let us put the pieces altogether and demonstrate selection sorting on a set of events:

Program 6.8 Selection sort on a set of `EventObject` elements

```
static void SelectionSort(EventObject [] array)
  {
  int n=array.length;

  for(int i=0;i<n-1;i++)
    for(int j=i+1;j<n;j++)
      if (GreaterThan(array[i],array[j]))
       swap(array,i,j);
  }

  public  static void main(String[] args)
  {
    EventObject [] array=new EventObject[5];

    array[0]=new EventObject(2008,06,01);
    array[1]=new EventObject(2005,04,03);
    array[2]=new EventObject(2005,05,27);
    array[3]=new EventObject(2005,04,01);
    array[4]=new EventObject(2005,04,15);

    SelectionSort(array);
    for(int i=0;i<array.length;i++)
      EventObject.Display(array[i]);
    System.out.println("");
```

```
}
```

We get the events properly sorted and displayed on the output console:

```
2005/4/1
2005/4/3
2005/4/15
2005/5/27
2008/6/1
```

6.4.3 Complexity of selection sorting

Let us now focus on the time complexity of performing a selection sort on an array of n elements. We shall count only the basic primitive operations `GreatherThan` and `swap` as constant operations, ignoring all other instructions (for example, increment the loop index, etc). The worst-case scenario of selection sort is to sort an input array that is sorted using the reverse order:

```
16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1
1, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2
1, 2, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3
...
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16
```

Indeed, we need to find the minimum of all sub-arrays in the outer loop. This is done by swapping the current minimum with the current element whenever the comparison predicate `GreaterThan` is `true`. For reverse order sorted arrays, it takes $2(n+1-i)$ primitive operations at stage i since the `GreaterThan` is always evaluated to `true` so that we need to perform the swapping. Thus the overall worst-case complexity is $\sum_{i=1}^{n} 2(n+1-i) = 2\sum_{i=1}^{n} i = n(n+1)$, which is of the order of n^2. Selection sort has *quadratic* complexity.
This can be checked experimentally by explicitly counting the number of operations for sorting the reverse ordered array:

Program 6.9 Experimentally calculating the worst-case complexity of selection sorting

```
class SelectionSortComplexity
{
static int nboperations;
static boolean GreaterThan(int a, int b)
{nboperations++; return (a>b);}
static void swap (int [] array, int i, int j)
{
nboperations++;
int tmp=array[i];array[i]=array[j];array[j]=tmp;}
public static void main(String[] args)
{
```

```
int [] array={16,15,14,13,12,11,10,9,8,7,6,5,4,3,2,1};
nboperations=0;
SelectionSort(array);
System.out.println("Number of operations:"+nboperations);
int nb=2*array.length*(array.length-1)/2;
System.out.println("Number of operations:"+nb);
}
}
```

We get (with $15 \times 16 = 240$):

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Number of operations:240
Number of operations:240
```

6.5 QuickSort: Recursive sorting

Let us now turn to a fast expected sorting algorithm often used in practice: QuickSort. QuickSort is a *recursive sorting* procedure that proceeds as follows:

- First, partition the elements according to the pivot `array[0]`:
  - those smaller than the pivot stored in `arrayleft`,
  - those greater than the pivot stored in `arrayright`,
  - those equal to the pivot (in case of ties) stored in `arraypivot`.
- Then solve recursively the sorting in arrays `arrayleft` and `arrayright`, and
- Recompose the array as follows:

```
arrayleft arraypivot arrayright
```

Note that in this recursive algorithm, the terminal case is sorting a single element. This is done trivially by doing nothing since a single element array is a sorted array by definition. The partition procedure is done in-place without using extra memory as follows:

Program 6.10 The partition procedure in QuickSort

```
static int partition(int [] array, int left, int right)
  {
    int m=left; // pivot
    for(int i=left+1; i<=right;i++)
    {
      if (GreaterThan(array[left],array[i]))
      {
        swap(array,i,m+1);
```

```
        m++;// we extend left side array
        }
    }
      // place pivot now
      swap(array,left,m);
      return m;
  }
```

Once this partition has been done according to the pivot element, we simply recurse on the left/right sub-arrays as follows:

Program 6.11 Recursive sorting

```
static void quicksort(int [] array, int left, int right)
  {
    if (right>left)
    {
      int pivot=partition(array,left,right);
      quicksort(array,left,pivot-1);
      quicksort(array,pivot+1,right);
    }
  }

static void QuickSort(int [] array)
  {quicksort(array,0, array.length-1);}
```

6.5.1 Complexity analysis of QuickSort

Let us analyze the running time of QuickSort on an arbitrary array of size n:

- The worst-case running time is quadratic: $O(n^2)$,
- The expected running time is $O(nlogn)$,
- The best case is linear $O(n)$. It is reached whenever all elements are identical.

In theoretical computer science, we furthermore analyze lower bounds, which are time complexity bounds that can not be beaten by *any* algorithm. It is known that sorting requires of the order of $n \log n$ time (comparison operations) whatever the method/algorithm. This lower bound complexity is denoted as follows: $\Omega(n \log n)$.

6.6 Searching by hashing

Hashing is a fundamental technique often used in practice since it is quite a simple technique to implement that has excellent empirical performances. The

two underlying ideas of hashing are:

– Convert object elements `X` into integer numbers `x` by a conversion function `x=I(X)`. The problem is then to transform a set of n integers sparsely lying in $\mathbb{N}$ into a compact array of size m much less than n (written as $m << n$).

– Use an array to store the elements. Since arrays are of prescribed sizes, use a `hashing function` that operates some modulo operation to map an integer `x` into its hash index `h(x)`.

The main problem hashing techniques have to deal with are *collisions*. Collision occurs whenever two objects `X` and `Y` have been hashed into the same index. That is, `h(I(X))` is identical to `h(I(Y))`. Often, the hashing function is taken as $h(k) = k \mod m$, where m is a prime number.
Finding good hashing functions that minimize the risk of collisions, and adopting a good search/store policy in cases of collisions are the two challenging tasks of hashing that we will describe in the next chapter. The following code shows a basic skeleton for hashing strings. It does not solve collisions but rather explicitly reports them on the console output.

```
static int m=23;
static int String2Integer(String s)
{
int result=0;
for(int j=0;j<s.length();j++)
result+=(int)s.charAt(j);
return result;
}
// Note that m is a static variable
static int HashFunction(int l)
{return l%m;}
```

Program 6.12 A demonstration code for hashing strings

```
  public static void main (String[] args)
  {
  String [] animals={"cat","dog","parrot","horse","fish",
   "shark","pelican","tortoise", "whale", "lion",
  "flamingo", "cow", "snake", "spider", "bee", "peacock",
   "elephant", "butterfly"};

  int i;
  String [] HashTable=new String[m];

  for(i=0;i<m;i++)
    HashTable[i]=new String("-->");

  for(i=0;i<animals.length;i++)
    {int pos=HashFunction(String2Integer(animals[i]));
    HashTable[pos]+=(" "+animals[i]);
```

```
    }

  for(i=0;i<m;i++)
    System.out.println("Position "+i+"\t"+HashTable[i]);
}
```

Running this code, we get the following output:

```
Position 0      --> whale
Position 1      --> snake
Position 2      -->
Position 3      -->
Position 4      -->
Position 5      -->
Position 6      -->
Position 7      --> cow
Position 8      --> shark
Position 9      -->
Position 10     -->
Position 11     -->
Position 12     --> fish
Position 13     --> cat
Position 14     -->
Position 15     --> dog tortoise
Position 16     --> horse
Position 17     --> flamingo
Position 18     -->
Position 19     --> pelican
Position 20     --> parrot lion
Position 21     -->
Position 22     -->
```

We will further describe hashing techniques once the linked lists are introduced in the next chapter.

6.7 Exercises

Exercise 6.1 (Sequential search)

Consider a collection of books in a library with each book modeled by its (1) title, (2) author(s) and (3) ISBN unique number. Provide a class `Book` with an appropriate constructor. Further equip this class with another constructor that takes as its argument a `Book` object. Consider an array of books `Book [] array`. Give a search function `SequentialSearch` that given a query book element searches inside the array whether the book is present or not by checking ISBN numbers. What is the time complexity of this method? Design a function `static Book []`

`BookByAuthor(String str)` that given a string storing an author name, collects all books written by the author.

Exercise 6.2 (Selection sort and bisection search)

We would like to speed up the former search function for librarians that manipulates large collections of books. Implement the selection sort algorithm on arrays of `Book` elements using the order of ISBN numbers (International Standard Book Numbers). What is the time complexity of this sorting algorithm? Consider the books stored in a sorted array according to their ISBNs. Provide a fast recursive dichotomic search method for checking whether a book is inside the collection or not. What is the time complexity of this bisection search?

Exercise 6.3 (QuickSort on arrays of strings)

Modify the QuickSort program so that it can operate on arrays of strings. We shall use the lexicographic order defined on strings. Measure the time spent by the various sorting and searching methods using the function System.nanoTime();

7
Linked Lists

7.1 Introduction

Linked lists allow us to structure input data sets into elementary units by chopping the data-sets into its many individual elements that are stored in corresponding *cells*. Cells are all chained together into a single thread of cells, starting from the head to the tail. These chained cells can be manipulated dynamically by either *adding* or *removing* elements. These operations can be carried out efficiently (in constant time $O(1)$) by creating new cells or deleting some cells of the list. Linked lists are therefore preferred to arrays whenever we do not know *a priori* the input size. This is all the more interesting for sorting and searching operations that consider dynamic data sets in practice.

7.2 Cells and lists

7.2.1 Illustrating the concepts of cells and lists

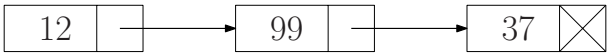
Consider an arbitrary set of objects $\mathcal{O} = \{O_1, ..., O_n\}$ for which we would like to create a linked list data-structure. A cell is an elementary structure consisting of *two fields*:

– The first field is used for storing the considered object. Thus the first field plays the role of container.

F. Nielsen, *A Concise and Practical Introduction to Programming Algorithms in Java*, Undergraduate Topics in Computer Science, DOI 10.1007/978-1-84882-339-6_7,

– The second field is used for storing the *reference* to the next cell. This allows it to point to the next cell.

A *linked list* is a set of *chained cells* that has two distinguished cells: the *head* and the *tail*. The head marks the beginning of the linked list. The tail does not point to the next cell; it thus signals that it is the last cell of the chained cells. That is, it stores the **null** reference signaling that the cell is a tail.
For example, consider the following linked list of 3 natural numbers $12 \rightarrow 99 \rightarrow 37$. We depict the associated chained cells as follows:



A cell can be interpreted as a pair of (container,reference)
Thus the above linked list can be parenthesized as (12 (99 (37 **null**))).

7.2.2 List as an abstract data-structure

The concept of lists is independent of the programming language. Lists belong to the fundamental ways of structuring data. They can be defined formally in a generic setting as follows:

Constant	:	Empty list listEmpty (**null** in Java)
Operations:		
Constructor	:	List x Object List x Object → List
Head	:	List → Object (not defined for listEmpty)
Tail	:	List → List (not defined for listEmpty)
isEmpty	:	List → Boolean
length	:	List → Integer
belongTo	:	List x Object → Boolean

7.2.3 Programming linked lists in Java

In Java, we need to create and declare a *new type* induced by a class for creating and initializing elementary cells. Let us denote this class by List. For the sake of simplicity, consider a linked list of integers. The cell container should therefore be of type **int**, and the next field shall refer to the next cell. That cell is also of type List. It is enough to define a variable of type List for pointing to the following cell. More precisely, that variable stores a reference in Java to that cell (a machine word coded using 32 bits, 4 bytes). Thus we declare the linked list structure by defining the following class:

Program 7.1 Declaration of a linked list

```
class List
{int container;
List next;}
```

We now need to provide:

– A *proper constructor* to initialize various objects of type List, and

– A few functions implementing the basic operations defined by the abstract framework of § 7.2.2.

Program 7.2 Linked list class with constructor and basic functions

```
public class List
{
int container;
List next;

// Constructor  List(head, tail)
List(int element, List tail)
  {this.container=element;
  this.next=tail;}

static boolean isEmpty(List list)
  {if (list==null) return true;
    else return false;}

static int head(List list)
{return list.container;}

static List tail(List list)
{return list.next;}
}
```

In Java, we do not need to explicitly free memory of cells not pointed by any other variables (meaning unreachable) as the garbage collector takes care of that. The type of the class `List` is recursive since it has one field `next` that defines a reference to the same type. That is, a *recursive type* is a data type with fields that may contain other values of the same type. The recursive type is self-referential.

7.2.4 Traversing linked lists

To search whether a given *query* element belongs to the elements stored in the list or not, we first start the exploration from the head cell of the list and compare the integer field of the cell with the query. If these integers match then we successfully have found the element and return **true**. Otherwise, we chain

to the next cell using the *reference* stored in field next. We proceed cell-wise until we eventually find the element, or we reach the tail cell of the list that has its next field set to **null**.

Program 7.3 Checking whether an element belongs to the list by traversing it

```
static boolean belongTo(int element, List list)
{
while (list!=null)
  {
    if (element==list.container)
      return true;
    list=list.next;
  }
  return false;
}
```

The time complexity of searching in a linked list of n cells is therefore linear ($O(n)$ using the big Oh-notation) since we need to fully explore the list to report that it is not inside. Note that since Java is pass-by-value only (with a pass-by-reference for non-primitive objects), at the end of calling static function belongTo the reference of list remains unchanged.

7.2.5 Linked lists storing String elements

List can store arbitrary types of objects. For example, the list declaration of storing String inside an individual cell is given below:

Program 7.4 Linked list storing String elements

```
class ListString
{
String name;
ListString next; // reference

// Constructor
ListString(String tname, ListString tail)
  {this.name=tname; this.next=tail;}

static boolean isEmpty(ListString list)
  {return (list==null);}

static String head(ListString list)
  {return list.name;  }

static ListString tail(ListString list)
  {return list.next;}
}
```

To detect whether or not a given string has already been inserted inside a linked list, we traverse again the list but now performs the *equality test* of strings as follows:

```
static boolean belongTo(String s, ListString list)
{
while (list!=null)
  {// Use equals method of String that take a String as its
      argument
    if (s.equals(list.name))
       return true;
    list=list.next;
  }
  return false;
}
```

7.2.6 Length of a linked list

The length of a linked list is defined as its number of elements. To determine the length of a given linked list, we browse the list starting from its head until we reach the tail, incrementing by one every time we traverse a cell. The static (class) function length takes as argument a reference to a list of strings: a variable of type ListString.

Program 7.5 Static (class) function computing the length of a list

```
static int length(ListString list)
{
  int l=0;
  while (list!=null)
    {l++;
    list=list.next;
    }
  return l;
}
```

Once again, observe that since Java is passing function arguments only by value (using references for objects such as lists), at the end of the function the original value (reference) of list is preserved although we perform list =list.next; instructions inside the body of the function. Thus we can call twice the static length function of class ListString; it will return the same length (as expected).

```
System.out.println(ListString.length(u));// reference to list
     u is kept preserved
System.out.println(ListString.length(u));
```

7.2.7 Dynamic insertion: Adding an element to the list

To add an element to an unordered list, we create a new cell storing that element and add that cell at the head. The static function Insert returns the reference to the newly created head cell. In other words, we create a new cell with its container assigned to the value of the element, and with its tail set to the head of the former list. Note that the element can already have been inserted.

Program 7.6 Inserting a new element to the list

```
static ListString Insert(String s, ListString list)
{return new ListString(s,list); }
```

Based on this function, we create a linked list by starting from the **null** cell and inserting successive elements to the list head.

```
myList=null;
myList=ListString.Insert("First", myList);//here, we
    explicitly means the Insert function of class ListString
myList=new ListString("Second",myList);//Since this code is
    located inside the ListString body, we can remove the
    class name
...
```

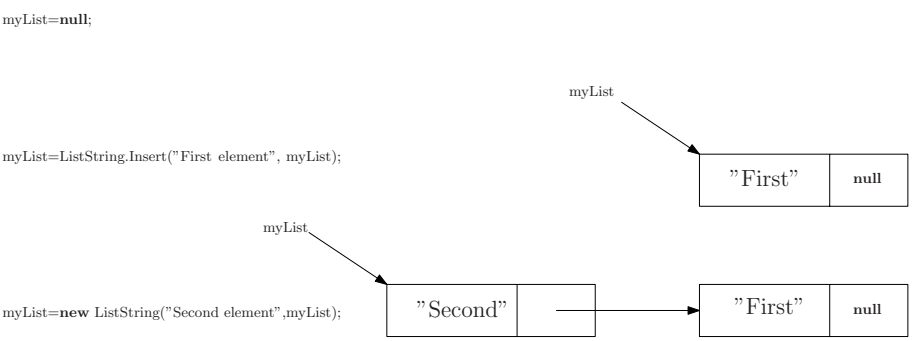


Figure 7.1 Creating a linked list by iteratively calling the static function `Insert`. The arrows anchored at variable myList mean references to objects of type `ListString`

7.2.8 Pretty printer for linked lists

Data-structures based on lists and algorithms processing lists are often more difficult to debug compared to similar algorithms based on arrays. Therefore, it is useful to print at the output console the chained cell structure of a given linked list. Procedure Display does this by traversing the list from its head.

Program 7.7 Pretty printer for lists

```
static void Display(ListString list)
{
  while(list!=null)
  {
  System.out.print(list.name+"-->");
  list=list.next;
  }
System.out.println("null");
}
```

We print a list referenced by its head element myList by invoking the instruction ListString.Display(myList); Instead of printing to the console at every cell visited, we could also have created a String variable storing the displayed list and return the string as output:

Program 7.8 Static function writing the structure of a linked list into a String object

```
static String Display(ListString list)
{String result="'"'
  while(list!=null)
  {
  result+=list.name+"-->";
  list=list.next;
  }
result+="null";
return result;
}
```

We print the list by calling the static function System.out.println(ListString.Display(myList));

7.2.9 Removing an element from a linked list

To remove an element from a linked list, we first need to search for *its position* in the linked list. If the element is found, we need to do a bit of list surgery to branch the former cell reference next to the next cell element. If the source list is empty (reference is `null`) or if the element is found at the head, we can easily

handle these cases. Thus we remove an element from the list by deconnecting its cell from the other chained cells as depicted by Figure 7.2.

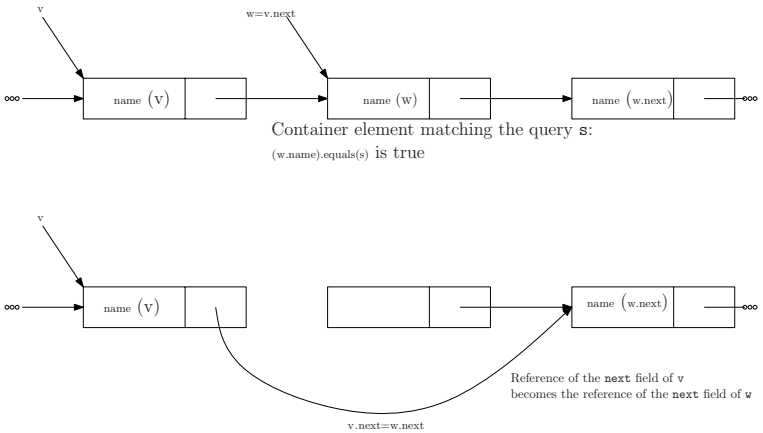


Figure 7.2 Removing an element to the list by deconnecting its cell from the other chained cells

Program 7.9 Removing an element from a list

```
static ListString Delete(String s, ListString list)
{
// Case: list is empty
if (list==null)
  return null;

// Case: element is at the head
if (list.name.equals(s))
  return list.next;

// Otherwise
ListString v=list;
ListString w=list.next; //tail

while( w!=null && !((w.name).equals(s)) )
  {v=w; w=v.next;}

// A bit of list surgery here
if (w!=null)
  v.next=w.next;

return list;
}
```

7.2.10 Common mistakes when programming lists

Programming linked lists involves handling objects that define structure cells. Since the tail is defined as the cell having its `next` field set to `null`, this can cause a problem for some code like:

```
result+=list.name+"-->";
list=list.next;
```

We need to be sure that variable `list` is never the object since we cannot access fields of the `null` object. Trying to do so will raise an exception (called here `nullPointerException`). Thus in general, take particular care when performing tests like **if** (currentCell!=**null**)... to detect whether the current object currentCell is the `null` object or not, before accessing its fields. For example, we better rewrite the static `head` function as follows:

```
static int head(List list)
  {if (list!=null)
      return list.container;
        else
          return -1; }
```

7.3 Recursion on linked lists

By its very nature defined in the abstract framework, the "self-definition" of lists yields practical recursive algorithms for designing various tasks. Codes for these algorithms are usually very compact. For example, let us revisit the static `length` function previously described above. We design a recursive function by considering as a terminal case the `null` list that is of length 0. Otherwise we return one plus the length of the linked list anchored at the tail for the current cell. This gives the following compact code:

Program 7.10 Recursive function for computing the length of a list

```
static int lengthRec(ListString list)
{ // terminal case?
  if (list==null)
    return 0;
  else
    return 1+lengthRec(list.next);
}
```

Let us attach this static function on class ListString by inserting the code inside the body of the class. Then we display at the output console the length of list

myList by executing the following instruction:
System.out.println(ListString.lengthRec(myList));
Similarly, we can rewrite with a recursive the previous functions, such as the `belongToRec` function that returns a boolean stating whether a given element is inside a list or not:

Program 7.11 Recursive membership function

```
static boolean belongToRec(String s, ListString list)
{
if (list==null) return false;
  else
  {
    if (s.equals(list.name))
         return true;
         else
         return belongToRec(s,list.next);
  }
}
```

The recursive display procedure is written as:

Program 7.12 Recursive display of a list

```
static void DisplayRec(ListString list)
{
  if (list==null)
    System.out.println("null");
    else
      {
        System.out.print(list.name+"-->");
        DisplayRec(list.next);
      }
}
```

Note that if we choose to call function DisplayRec *before* printing on the console the current cell by interchanging the two lines in the else-case as follows:

Program 7.13 Reversed recursive display of a list

```
static void DisplayRecRev(ListString list)
{
  if (list==null)
    System.out.print("null");
    else
      {
        DisplayRecRev(list.next);
        System.out.print("<--"+list.name);
      }
}
```

Then because of the mechanism of the function call stack, the list will be displayed from the tail to its head. Indeed, the first word written to the output

console is `null` since it is the terminal case. Then we pop the function call stack and write the last cell name string, and so on.

```
ListString l=new ListString("Paris",null);
    l=new ListString("Tokyo",l);
    l=new ListString("Berlin",l);
    l=new ListString("Porto",l);
    l=new ListString("Cambridge",l);
    l=new ListString("Roma",l);

    ListString.Display(l);
    ListString.DisplayRec(l);
    ListString.DisplayRecRev(l);
    System.out.println("");
```

```
Roma-->Cambridge-->Porto-->Berlin-->Tokyo-->Paris-->null
Roma-->Cambridge-->Porto-->Berlin-->Tokyo-->Paris-->null
null<--Paris<--Tokyo<--Berlin<--Porto<--Cambridge<--Roma
```

7.4 Copying linked lists

Copying or cloning objects is an essential operation frequently used in programs. To copy a list, we need to traverse the source list and create a new cell every time we visit a list. The snippet code below copy a source list `l`:

Program 7.14 Copying iteratively a source list but reversing its order

```
static ListString copy(ListString l)
{
ListString result=null;

while (l!=null)
{
  result=new ListString(l.name,result);
  l=l.next;
}
return result;

}
```

Note that the side-effect of the iterative function copy is to reverse the order of the linked list. Although it might not be important in some cases, we would rather get a perfect list clone by using the following recursive procedure:

Program 7.15 Copying a source list by maintaining the original head-tail order

```
static ListString copyRec(ListString l)
```

```
{
if (l==null)
  return null;
    else
    return new ListString(l.name,copyRec(l.next));
}
```

7.5 Creating linked lists from arrays

So far we have handled large (static) data-sets into arrays that are contiguously stored in the global memory. It is useful to start to build linked lists from arrays that will support editing operations. Therefore, we create the helper function, which creates a linked list from a `String` array as follows:

```
static ListString Build(String [] array)
{
ListString result=null;

// To ensure that list head is the first array element
   decrement
// Iterate from largest to smallest index
for(int i=array.length-1;i>=0;i--)
  result=new ListString(array[i],result);
return result;
}
```

Consider, for example, the following code snippet that shows how to use the function Build:

```
String [] colors={"green", "red", "blue", "purple", "orange",
    "yellow"};
ListString lColors=ListString.Build(colors);
ListString.Display(lColors);
```

The result displayed on the output console by executing the above instructions is:

```
green-->red-->blue-->purple-->orange-->yellow-->null
```

7.6 Sorting linked lists

Given an arbitrarily ordered linked list, we wish to sort the list elements, say in increasing order. We must first consider a simpler problem that consists of

merging two ordered lists without creating new cells. We will then make use of this merging function to provide an efficient sorting procedure.

7.6.1 Merging ordered lists

Let us consider two ordered linked lists u and v of arbitrary lengths with corresponding cells storing integers in increasing order. That is, the sequence of integers stored in cells from the head to the tail is increasing in both lists. We would like to create a *new* linked list merging the elementary cells of u and v in increasing order *without* creating any new cell. To make it plain, we need to recycle the elementary cells already available in u and v when building the merged list. To provide an illustrating example, the list merging operation will give the following result:

u	3−−>6−−>8−−>**null**
v	2−−>4−−>5−−>7−−>9−−>**null**
Merge(u,v)	2−−>3−−>4−−>5−−>6−−>7−−>8−−>9−−>**null**

We consider the following class List that defines the basic blocks of integer linked lists.

```
class List
{
int container;
List next;

// Constructor  List(head, tail)
List(int element, List tail)
  {this.container=element;
  this.next=tail;}

// Insert element at the head of the list
List insert(int el)
{return new List(el,this);}

void Display()
{List u=this;
while(u!=null)
  {
  System.out.print(u.container+"-->");
  u=u.next;
  }
System.out.println("null");
}
}
```

So that the above two example integer linked lists are created in the main procedure as follows:

```
public static void main(String [] args)
{
List u=new List(8,null);u=u.insert(6);u=u.insert(3);
u.Display();
List v=new List(9,null);v=v.insert(7);v=v.insert(5);v=v.
    insert(4);v=v.insert(2);
v.Display();
}
```

The easy case for merging lists is whenever one of the lists is empty (meaning that either u or v is equal to **null**). Indeed, in that case, we simply return the other list (which might also be empty, in which case, both lists are empty). Otherwise, both lists are considered non-empty, and we compare their head elements u.container and v.container. Assume without loss of generality that the integer of u.container is less than v.container. Then we need to set the reference to the next field of u to the result of merging the two ordered lists with heads referenced by u.next and v. That is where recursion beautifully kicks in. Of course, in case u.container>=v.container we do the converse. Hence, the recursive function for merging two ordered lists is quite simple as shown below:

Program 7.16 Merging two ordered linked lists

```
static List mergeRec(List u, List v)
{
if (u==null) return v;// terminal case
if (v==null) return u;// terminal case
// Recursion
if (u.container<v.container)
  {
  u.next=mergeRec(u.next,v);
  return u;
  }
 else
  {
  v.next=mergeRec(u,v.next);
  return v;
  }
}
```

7.6.2 Recursive sorting of lists

Now consider sorting any arbitrary linked list using the above recursive merging procedure. The sorting algorithm will not create any new cells but rather set the next field of every cell to the appropriate reference for linking appropriately to the next cell.

We consider the following recursive sorting scheme: Split the initial list u in

half and create from it two linked lists, say u1 and u2. Recursively sort both sub-lists u1 and u2 of roughly half size, and merge them using the previous recursive merging function. To split in half the source list u, we first compute the length of u and set u2 to the cell split met half way. We also need to create the tail of u1 by explicitly setting the next field of the previous cell (that we call prevSplit) to **null**. Overall the sorting code for linked list is given below:

Program 7.17 Recursively sorting an arbitrary linked list

```
static List sortRec(List u)
{
int l=Length(u);

if (l<=1)
  return u;
else
  {int i, lr;
  List u1, u2, split, prevSplit; // references to cells

  u1=u;
  prevSplit=split=u;
  i=0;lr=l/2;

  while (i<lr)
    {i++;
    prevSplit=split;
    split=split.next;}

  u2=split; // terminates with a null
  prevSplit.next=null; // ensure u1 terminates with a null

  return mergeRec( sortRec(u1), sortRec(u2) );
  }
}
```

Let u be the following linked list:

20−−>19−−>21−−>23−−>17−−>15−−>1−−>6−−>9−−>2−−>3−−>**null**

Then executing the following code...

```
u.Display();
List sortu=sortRec(u);
System.out.println("Sorted linked list:");
sortu.Display();
```

...will return the sorted linked list:

```
1-->2-->3-->6-->9-->15-->17-->19-->20-->21-->23-->null
```

Once again, we have only recycled the previously created cells by changing their `next` field. Procedures sortRec and mergeRec do not create new cell objects.

Indeed, you can check that there are no **new** keywords in the body of these functions.

7.7 Summary on linked lists

Linked lists allow one to consider fully dynamic data structures. It is, however, quite useful to start building a list from a former array. A list object is nothing more than a reference to the cell of the linked list, namely a reference to the head of the list. We have presented `singly chained linked list` for unidirectionally traversing elements from the head to the tail. It is also possible to create *doubly chained linked lists* that contain in their class definition two reference fields beyond the container: Say, prev and next references.

```
class DoublyLinkedList
{int container;
DoublyLinkedList prev, next;
}
```

Finally, notice that in Java, we do not need to free unused cells (for example cells discarded when removing elements) since the *garbage collector* (GC) will do it automatically on our behalf. Let us now consider an application using linked lists: hashing.

7.8 Application of linked lists: Hashing

Hashing is a technique for storing a collection of homogeneous objects taken from a gigantic space into a compact array. To make it more concrete, assume we have to deal with a collection of strings of type `String`. The potential full collection of strings is gigantic and is called the *universe* from which objects are taken. If we consider a given set of n object, we would like to store them efficiently in a data-structure so that we can insert new objects and search for objects in almost constant time. The underpinning principle of hashing is to first transcode the string into an integer using a conversion procedure named String2Integer.

```
static int String2Integer(String s)
  {
    int result=0;
    // this is the method s.hashCode()
    for(int j=0;j<s.length();j++)
      result=result*31+s.charAt(j);
```

```
    return result;}
```

Note that the above procedure is already implemented in Java in the `String` class by the `hashCode` method. With that conversion procedure, two strings are very likely to be converted into different integers. Now we create a *hash table* `HashTable` that is an array of given size m. Then we hash string `s` using its corresponding integer k into an index $h(k)$ of `HashTable`, ranging from 0 to $m-1$ using a *hash function* h. This is done simply by taking the modulo operation (symbol % in Java) of the integer resulting from the conversion procedure. In order to avoid hashing strings into the same array cell, we choose m to be prime (here, $m = 23$):

$$h(k) = k \mod m, \text{where } m \text{ is a prime number.}$$

The code for hashing a set of strings is given below:

Program 7.18 Hashing a set of strings into a hash table

```
static int m=23;
// Note that m is a static variable
static int HashFunction(int l)
{return l%m;}
```

```
  public static void main (String[] args)
  {
  String [] animals={"cat","dog","parrot","horse","fish",
   "shark","pelican","tortoise", "whale", "lion",
  "flamingo", "cow", "snake", "spider", "bee", "peacock",
   "elephant", "butterfly"};

  int i;
  String [] HashTable=new String[m];

  for(i=0;i<m;i++)
    HashTable[i]=new String("-->");

  for(i=0;i<animals.length;i++)
    {int pos=HashFunction(String2Integer(animals[i]));
    HashTable[pos]+=(" "+animals[i]);
    }

  for(i=0;i<m;i++)
    System.out.println("Position "+i+"\t"+HashTable[i]);
}
```

Compiling and executing the program, we get the following result:

```
Position 0      --> whale
Position 1      --> snake
Position 2      -->
Position 3      -->
```

```
Position 4      -->
Position 5      -->
Position 6      -->
Position 7      --> cow
Position 8      --> shark
Position 9      -->
Position 10     -->
Position 11     -->
Position 12     --> fish
Position 13     --> cat
Position 14     -->
Position 15     --> dog tortoise
Position 16     --> horse
Position 17     --> flamingo
Position 18     -->
Position 19     --> pelican
Position 20     --> parrot lion
Position 21     -->
Position 22     -->
```

Observe that for some indices of the array, there have been two strings assigned to that array cell. For example the `dog` and `tortoise` have been both assigned to index 15, while `parrot` and `lion` have been assigned to index 20. This is what is called ǫllision phenomenon. For strings, we just solved this problem by concatenating strings stored at the same position (for illustration purpose). In the ideal case, if no collision happens, we can insert a new string in constant time, and search for a given string in constant-time too. This hashing technique is therefore efficient compared to arrays/lists data-structures. This explains why hashing is often used in real-world applications (and standardized in Java API packages). Let us now see how to resolve the collision problems by using two different strategies: (1) the open address method, and (2) the chained list method.

7.8.1 Open address hashing

Let us resolve hashing collisions using the open address scheme. The method consists of first computing the array index of the hashed string. Then two cases occur:

– Either the hash table cell at that index is free, and then we store the string there, or

– There is already a string stored at that index. Then we proceed starting from that index to search for the first free position of the hash table by iteratively incrementing the index. We eventually find that position and store the string there.

```
String [] HashTable=new String[m];
  // By default HashTable[i]=null
    for(i=0;i<animals.length;i++)
    {
    int s2int=String2Integer(animals[i]);
    int pos=HashFunction(s2int);

    while (HashTable[pos]!=null)
      pos=(pos+1)%m;

    HashTable[pos]=new String(animals[i]);
    }
```

```
Position 0      whale
Position 1      snake
Position 2      bee
Position 3      spider
Position 4      butterfly
Position 5      null
Position 6      null
Position 7      cow
Position 8      shark
Position 9      null
Position 10     null
Position 11     null
Position 12     fish
Position 13     cat
Position 14     peacock
Position 15     dog
Position 16     horse
Position 17     tortoise
Position 18     flamingo
Position 19     pelican
Position 20     parrot
Position 21     lion
Position 22     elephant
```

To search whether a given string belongs to the hash table or not, we proceed by first converting it to an integer, hashing it to the hash array index, and checking the value at that location. If there is nothing, we report that the object is not present (all this in constant time). Otherwise, we iteratively visit the cells by incrementing the index until we find the string or we find an empty array cell.

The main drawback of open addressing is that we cannot insert for sure more strings than the array size. In case we cannot upper bound the maximal number of elements. It is better to consider an alternative technique for managing the hash table. This is where linked lists kicked in.

7.8.2 Solving collisions with linked lists

Instead of considering an array of strings as we previously did, we now consider an array of linked lists. Each cell of the array will store a reference to the head of the linked list it refers to. When there is hash collision, we just add the element to the linked list of its hashed index. The code for initializing the hash table and filling it with the data-set is given below:

```
ListString [] HashTable=new ListString[m];

  for(i=0;i<m;i++)
    HashTable[i]=null;

for(i=0;i<animals.length;i++)
{
int s2int=String2Integer(animals[i]);
int pos=HashFunction(s2int);
HashTable[pos]=ListString.Insert(animals[i],HashTable[pos]);
}

for(i=0;i<m;i++)
    ListString.Display(HashTable[i]);
```

Executing the above code yields:

```
whale-->null
bee-->snake-->null
null
spider-->null
butterfly-->null
null
null
cow-->null
shark-->null
null
null
null
fish-->null
peacock-->cat-->null
null
tortoise-->dog-->null
horse-->null
flamingo-->null
null
pelican-->null
lion-->parrot-->null
elephant-->null
null
```

7.9 Comparisons of core data-structures

We have described several data-structures for organizing a collection of objects that provide basic operations such as inserting/deleting and searching whether a given query element belongs to the collection. We have seen that arrays are well-suited to static data-sets but fail short in case we need to delete/add elements. Linked lists have then be introduced for fully maintaining the collection of elements. However, searching using linked lists could be tricky. We finally combined both array and linked list technique in the hashing framework showing clearly its advantages when no collisions occur. The table 7.1 quickly summarizes the various time complexity operations of respective data-structures.

Data-structure	Initialization	Search	Insert	Delete
Array	O(1)	O(n)	O(1)	O(n)
Sorted array	O(n log n)	O (log n)	O(n)	O(n)
Hashing (Chained list)	O(1)	Almost O(1)	Almost O(1)	Almost O(1)
List	O(1)	O(n)	O(1)	O(n)

Table 7.1 Performance of various data-structures

7.10 Exercises

Exercise 7.1 (Linked lists for intervals)

Write a class `Interval` that stores the range `min`/`max` of intervals. Create a linked list of intervals. Provide the usual functions: `add`, `remove`, `length` and `display`.

Exercise 7.2 (Doubly linked lists)

A polygon is a piecewise linear non-self-intersecting closed curve. We consider modeling a polygon by its set of vertices (2D points stored in an array) with a linked list of vertex indices (integers). Write a class `Polygon` with a constructor that takes as its argument an array of 2D points describing the cyclic sequence of vertices. The linked list shall be bidirectional using the `prev` and `succ` fields.

Exercise 7.3 (Dynamic insertion for sorting)

Consider sorting an array `Dictionary` of strings by successively inserting the array item `Dictionary[i]` into a linked list of `Strings`. Provide a static function static StringList Sort(String [] array) that sorts the string array into a linked list, and return its head element. What is the time complexity of this sorting method?

Exercise 7.4 (Sparse polynomial representation)

Consider the following recursive type for representing sparse polynomials (polynomials with many coefficients set to zero):

Program 7.19 A class for manipulating sparse polynomials

```
class SparsePolynomial
{
int degree;
double coefficient;
SparsePolynomial next;

SparsePolynomial(int d, double v, SparsePolynomial poly)
{
this.degree=d;
this.coefficient=v;
this.next=poly;
}
}
```

Write the following static functions:

- `static void Display(SparsePolynomial poly)` that displays a polynomial to the console,
- `static SparsePolynomial Add(SparsePolynomial p, SparsePolynomial q)` that performs the addition of two polynomials `p` and `q`,
- `static SparsePolynomial Multiply(SparsePolynomial p, SparsePolynomial q)` that calculates the product of two polynomials `p` and `q`.

Exercise 7.5 (String pattern matching)

Given two strings of characters stored in array of `char`, design a predicate `static boolean occurence(char [] T, char [] M, int i)` that reports whether string `M` is located at index `i` of string `T`. The arrays of characters can be extracted from corresponding strings of type `String` using the method `toCharArray()` (for example, `String stringt="A simple`

`test string";char [] T=stringt.toCharArray();`). Deduce a naive algorithm for reporting the potential occurences of `M` in `T`. Let m and t be the respective length of arrays `M` and `T`. What is the time complexity of this search procedure? Karp and Robin proposed an effective linear time algorithm that consists of using a signature function S. The principle is as follows: Compute once for all the signature of the motif pattern `M`, and get $S(M)$. Then for all i, check whether the signature of $T[i..i+m-1]$ matches the signature of M. If signatures do not match then there is no occurrence at that position. Otherwise, when signatures match, we still need to check whether it is a true occurence or not. Give an implementation of the Karp-Rabin algorithm using as a signature function the sum of the character ASCII codes of an array:

Program 7.20 An example of signature function for the Karp-Rabin pattern matching algorithm

```
static long signature(char [] X, int m, int i)
{
long result=0;
for(int j=i;j<i+m;j++)
  result+=X[j];
return result;
}
```

8
Object-Oriented Data-Structures

8.1 Introduction

In this chapter, we describe the two most fundamental data-structures usually encountered in applications: stacks and queues. We begin by explaining the queue data-structures and their generalization to priority queues. We then describe stacks. Furthermore, we introduce the notion of *abstract data-structures* defined by their prototype *interfaces*, and show how to program these abstract data-structures using the object-oriented functionalities of Java. We revisit some of the static functions handling data-structures covered in the previous chapters into the OO paradigm by designing class *methods* that act on objects themselves.

8.2 Queues: First in first out (FIFO)

8.2.1 Queues as abstract data-structures: Interfaces

A queue is a *generic abstract data type* for storing incoming objects while preserving their order of arrival. That is, incoming objects are time-stamped and processed according to their arrival date. The basic operations that queues

F. Nielsen, *A Concise and Practical Introduction to Programming Algorithms in Java*, Undergraduate Topics in Computer Science, DOI 10.1007/978-1-84882-339-6_8,

support are detailed by the following interface:[1]

– `add` or `queue`: Add an element at the *tail* of the queue,

– `process` or `remove`: Process the *first* element of the queue.

These queues are also equipped with a predicate `empty` that tells us whether the queue is currently empty or not. Queues are one example of abstract data-structures that can be implemented using various data-structures at their backbone. We start from the most common implementation that uses arrays of prescribed size: That is, arrays with sizes fixed once for all. We manipulate two indices in this array, say:

– `freePlace`: Denote the index of the *first* free remaining place in the array.

– `lastProcessed`: Denote the index of the *last* processed object initially set to -1).

Figure 8.1 illustrates these notations. The array acts as a buffer for storing the incoming objects.

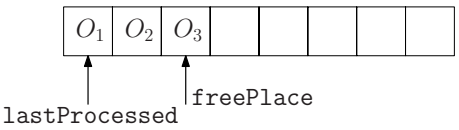


Figure 8.1 A queue implemented using an array requires two indices to indicate the last processed element and the first free location

8.2.2 Basic queue implementation: Static functions

We implement the `add`/`process` functions defined in the interface of queues using an array encapsulated into a tailored class. For example, let us consider queues for incoming sequences of `double` numbers. Whenever we call a `process` operation while the queue is empty, we choose by convention to return a special code: Here, chosen as -1.0. The straightforward code for implementing these `add`/`process` and `empty` operations is given below:

Program 8.1 A `double` queue with interface primitives implemented using static functions

```
class QueueDouble
{
static int lastProcessed=-1;
static int freePlace=0;
```

[1] Java provides the queue interface in its own API as explained in `http://java.sun.com/docs/books/tutorial/collections/interfaces/queue.html`

```
// Max objects is set to 1000
static double[] container=new double[1000];

// Stack in FIFO order
static void add(double a)
{
  if (freePlace<1000)
  {container[freePlace]=a;
  freePlace++;
  }
}

// Process in FIFO order
static double process()
{
if (freePlace-lastProcessed>1)
{
  lastProcessed++;
  return container[lastProcessed];
}
else
  return -1.0; // code for impossible to process
}

public static void main(String[] arg)
{
System.out.println("Queue demo:");
add(3.0); add(5.0); add(7.0);
System.out.println(process());
System.out.println(process());
System.out.println(process());
System.out.println(process());
System.out.println(process());
}

}
```

Running the above code yields:

```
Queue demo:
3.0
5.0
7.0
-1.0
-1.0
```

As it can be seen from the output, queues ensure that first-in elements are indeed first-out: That is, queues respect the order of arrival. Note that in this implementation all functions are static functions attached to the program class. Namely, they are class functions that access the buffer array defined as a static class array `static double[] container`.

8.2.3 An application of queues: Set enumeration

As a basic straightforward application of queues, consider the following mathematical puzzle:
Let $\mathcal{A} \subset \mathbb{N}$ be a set of integers such that:

– 1 belongs to $\mathcal{A}$, and

– If element a belongs to $\mathcal{A}$, then $2a + 1$ and $3a$ also belong to $\mathcal{A}$.

For a given n, we are asked to display all integers less or equal to n that belong to $\mathcal{A}$. To solve this problem algorithmically, consider starting with a queue initialized with element 1. We also use a boolean array to indicate whether integer a belongs to $\mathcal{A}$ or not. That is, we mark elements of $\mathcal{A}$ by tagging them using a boolean array. Initially, all elements except 1 are marked to `false`. Once this initialization is completed, for each element a of the queue, we perform in turn the following process:

– Compute $2a + 1$, add this element to the queue if a is less than n *and* not yet marked,

– Compute $3a$, add this element to the queue if a is less than n *and* not yet marked.

The algorithm terminates when the queue becomes empty, and we terminate by displaying all marked elements: the set $\mathcal{A}$ restricted to integers less or equal to n.
The listing below describes this simple application of queues:

Program 8.2 Enumerating set elements using queues

```
class QueueIntGame
{
final static int n=1000;
static int lastProcessed=-1;
static int freePlace=0;
static int[] container=new int[n];
static boolean[] mark=new boolean[n];

static void add(int a)
{ if (freePlace<n)
  {container[freePlace]=a;
  freePlace++;
  }
}

static boolean Empty()
{return ((freePlace-lastProcessed)==1); }

static void process()
```

```
{
int a;
    lastProcessed++;
    a=container[lastProcessed];
    if (a<n) mark[a]=true;
    if (2*a+1<n) add(2*a+1);
    if (3*a<n) add(3*a);
}

public static void main(String[] arg)
  {int i;

  for(i=0;i<n;i++)
    mark[i]=false;

  add(1);
  while(!Empty())
    process();

  for(i=0;i<n;i++)
    {if (mark[i])
      System.out.print(i+" ");}
  System.out.println("");
  }
}
```

Let us now run the program to list all integer elements less than 1000 and belonging to $\mathcal{A}$. We find:

```
1 3 7 9 15 19 21 27 31 39 43 45 55 57 63 79 81 87 91 93 111 115
117 127 129 135 159 163 165 171 175 183 187 189 223 231 235 237
243 255 259 261 271 273 279 319 327 331 333 343 345 351 367 375
379 381 387 405 447 463 471 475 477 487 489 495 511 513 519 523
525 543 547 549 559 561 567 639 655 663 667 669 687 691 693 703
705 711 729 735 751 759 763 765 775 777 783 811 813 819 837 895
          927 943 951 955 957 975 979 981 991 993 999
```

8.3 Priority queues and heaps

As we have already attested by ourselves in airports, queues that ensure the first in first out property are true only for passengers belonging to the same traveling class. Air flight companies implement several grades for their customers: Usually, it breaks down as economy, business and first classes. A first class client arriving at any time will be served first even if economy clients stand in the line long before. But another first class client will have to wait until the first first class client goes on board to embark. This property of

lining up according to grades is informally what is called *priority queues*, a generalization of queues[2]. These priority queues are yet another fundamental abstract data type lying at the heart of operating systems for performing "nice" job scheduling tasks. In a priority queue, it is always the object with highest priority that goes out of the queue first. In case several objects have the same priorities, the time arrival is taken into account and we operate along the guidelines of the queue then: "First in first out" for all objects with the same priority. For the case of a small fixed number of priorities, we may implement priority queues by declaring an array of queues. But this naive approach is rather inefficient and does not scale up with arbitrary large number of priority numbers. There are several ways to implement generic priority queues but these implementations basically all rely on the fundamental notion of *heaps*.
A heap is a special tree data-structure storing keys at its nodes. Heaps satisfy the so-called *heap property*. The heap property is described as follows: For any pair (P, Q) where Q is a child node of parent P, a heap must satisfy:

$$\text{key}(P) \geq \text{key}(Q).$$

Figure 8.2 depicts a heap. Observe that the maximal element of a heap is *always* located at the *root*. This explains why heaps are related to priority queues since it is straightforward to get the maximal priority by looking at the root. Heaps are also *complete binary trees*, meaning that nodes are added from left to right until the current level becomes complete (that is, saturated). We then create a next level and start adding nodes from the left to right side. Figure 8.2 depicts a heap. Observe that at a given layer the nodes are *not* ordered.

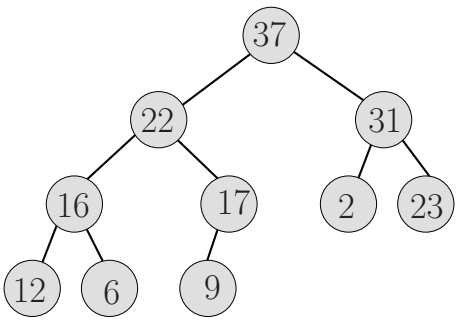


Figure 8.2 A heap is a binary tree specialized so that the keys of all children are less than the keys of their parents

Heaps can themselves be compactly *embedded* into arrays since the binary tree is perfect. For example, the heap of Figure 8.2 can be stored in an array by

[2] In simple queues, elements have the same priority.

reading its keys from the root to the leaves, level by level. This is a serialization operation. For the heap of Figure 8.2, we get the following sequence: 37, 22, 31, 16, 17, 2, 23, 12, 6, 9. This sequence implicitly encodes the perfect binary tree. But this does not mean that any array of integers is a heap in disguise. Indeed, remember that the heap property of child/parent keys should be satisfied. For arrays, this core property translates as:

$$1 \leq i, j \leq n, j = \frac{i}{2} \Rightarrow \text{array}[j] \geq \text{array}[i]. \tag{8.1}$$

Thus the data-structure for representing a heap embedded in an array is as follows:

```
public class Heap
{
int size;
int [] label;

static final int MAX_SIZE=10000;

Heap()
{
this.size=0;
this.label=new int[MAX_SIZE];
}
```

Let us now see the various operations acting on the heap compactly encoded into the array `int [] label;`

8.3.1 Retrieving the maximal element

As noticed earlier, the maximal key of a heap is necessarily stored at its root. That is, the maximal priority is encoded in the first cell of the array. Therefore, the following static function returns the maximal element of a heap given as argument:

Program 8.3 Retrieving the maximal priority

```
static int maxHeap(Heap h)
{return h.label[0];}
```

8.3.2 Adding an element

To add an element to the heap, we just need to add it to the first array position not yet assigned: The first free location of the array corresponds to adding a

leaf at the rightmost position of the last level of the tree. By doing so, we may, however, violate the heap property of Eq. 8.1, so that we need to transform this tree into a heap by swapping child-parent keys until we again get a heap. For example, consider adding element 25 to the heap. Figure 8.3 depicts the three stages:

– (1) add node 25 at the first place available on the last level of the tree,

– (2) swap that node with parent element 17 (since $25 > 17$),

– (3) swap again with parent element 22, and reach the final heap status since $35 < 37$.

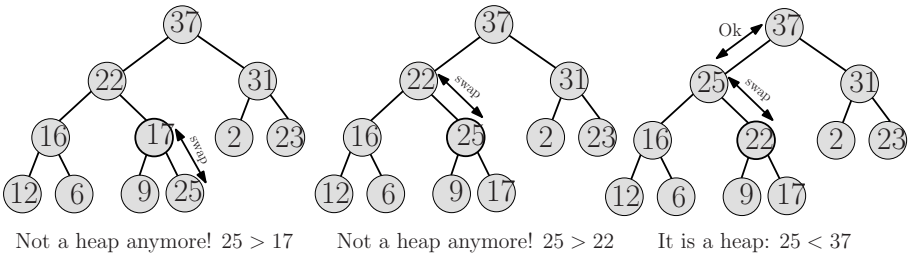


Figure 8.3 Adding element 25 to the heap: First, add a new node at the first immediate empty position; then eventually swap this node with its parent until the heap property is recovered

These basic step operations translate into corresponding operations on the array implicitly encoding the heap as follows:

Program 8.4 Adding an element to the heap

```
static void addHeap(int element, Heap h)
{
h.label[h.size]=element;
h.size++;

int i=h.size;
int j=i/2;

  while (i>1 && h.label[i]>h.label[j])
  {
    int tmp=h.label[i];
    h.label[i]=h.label[j];
    h.label[j]=tmp;
    i=j; // swap
    j=i/2;
  }
}
```

8.3.3 Removing the topmost element

We have previously seen that the maximal element is at the root of the heap. Let us explain now how to remove and update the heap. When we remove the maximal element located at the root (index 0 of the array), we choose the last inserted node and put this node at the the root position. We now similarly have to again ensure the heap property by eventually performing swapping operations: If the value of the current root node is below the value of its right child (the right subtree has less or exactly as many children as the left tree by construction), then we swap that node with the selected child, and reiterate until we reach the heap property. Figure 8.4 depicts these swapping operations when removing the topmost element.

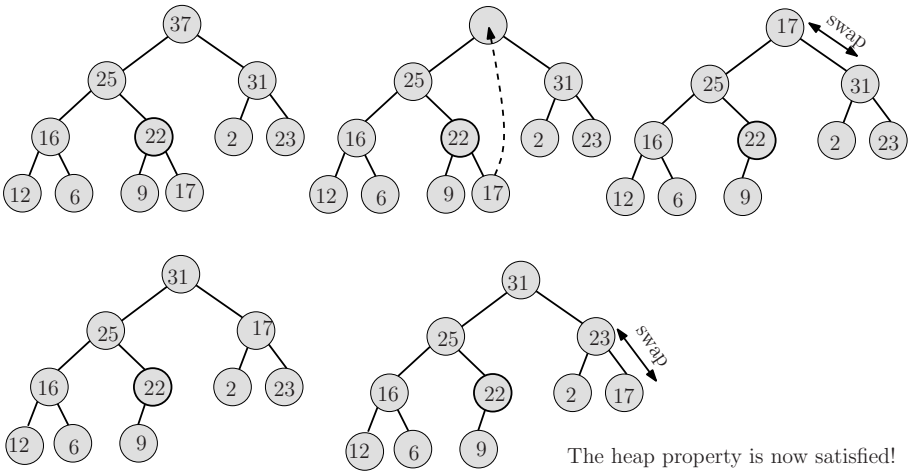


Figure 8.4 Removing the maximal element of the heap: First, replace the root by the last leaf node, and then potentially swap this node with its current children until the heap property is recovered

The portion of code performing this removal operation in the array encoding the heap is as follows:

Program 8.5 Removing the topmost element from the heap

```
static int removeHeap(int element, Heap h)
{
h.label[0]=h.label[h.size-1];
h.size--;

int i=0,j,k,tmp;
```

```
while(2*i<=h.size)
{
j=2*i;
if (j<h.size && h.label[j+1]>h.label[j])
  j++;

if (h.label[i]<h.label[j])
 {tmp=h.label[i];
 h.label[i]=h.label[j];
 h.label[j]=tmp;
 i=j;}
else break;
}

return h.label[h.size-1];
}
```

8.4 Object-oriented data-structures: Methods

We have so far explained data-structures and their basic operations using the basic concepts of *arrays* and *linked lists*. The functions manipulating these elementary data-structures have been all static functions so far. That means that *we had* to pass the data-structure as an argument of the function, or to declare the arrays/linked list as static variables of the class to be reached by functions. Although we attached these basic static functions to the class by inserting their code into the body of the class, this was not necessary. For example, we could have designed a class `Toolbox` that contains all functions operating on data-structures. Let us reexamine the code dealing with the linked list of the previous chapter:

Program 8.6 Linked list with static functions

```
class List
{
int container;
List next;

// Constructor  List(head, tail)
List(int element, List tail)
  {this.container=element;
  this.next=tail;}

// Static function is used here
static int head(List list)
{return list.container;}
}
```

We can remove all static functions from the scope of the `List` body and attach them to, say, a general-purpose class called `Toolbox` as follows:

Program 8.7 Linked list with static functions attached to a `Toolbox` class

```
class List
{
int container;
List next;

// Constructor  List(head, tail)
List(int element, List tail)
  {this.container=element;
  this.next=tail;}
}

class Toolbox{
static int head(List list)
{return list.container;}

static List insert(int s, List list)
{return new List(s,list); }
```

Then we could have created linked lists and called various related static functions by invoking static functions of the `Toolbox` class. For example, let us consider the following code snippet:

```
class List
{
int container;
List next;

// Constructor  List(head, tail)
List(int element, List tail)
  {this.container=element;
  this.next=tail;}
}

class Toolbox{

static int head(List list)
{return list.container;}

static List insert(int s, List list)
{return new List(s,list); }
}

class StaticFunction
{
  public  static void main(String [] args)
  {
  List myList=new List(3,null);
  myList=Toolbox.insert(6,Toolbox.insert(4,myList));
```

```
  System.out.println("Head:"+Toolbox.head(myList));
  }
}
```

Compiling and executing the program returns the expected result:

```
Head:6
```

Note that all usual mathematical functions such as exp (`Math.exp`), log (`Math.log`), $\sqrt{}$ (`Math.sqrt`), ... are defined similarly, as static functions, in the toolbox class `Math`. We can thus imitate this library behavior of classes by attaching common functions to a `Toolkit` class:

Program 8.8 Static functions attached to a library class

```
class Toolkit
{
static final double PI=3.14;

static double Square(double x)
{return x*x;}

static double Cubic(double x)
{return x*x*x;}
}

class StaticFuncStyle
{

public static void main(String[] s)
{
double radius=0.5;

double volSphere=(4/3.0)*Toolkit.PI*Toolkit.Cubic(radius);
double areaDisk=Toolkit.PI*Toolkit.Square(radius);
} }
```

A static function not accessing static class variables can therefore be attached to *any* class[3] but requires us to get as arguments the objects on which it processes on. The advantage of attaching all static functions to the same class as its argument object is that we can remove the class name. That is, if class `insert` were attached to the `List` class, then instead of having the instruction `myList=Toolbox.insert(6,Toolbox.insert(4,myList));`, we would have simplified it as `myList=insert(6,insert(4,myList));`

On the other hand, a *data-centric view* will consider the object `List` on which we can apply various processes called *methods*. These object methods do not require us to give as an argument the object itself as it is understood that these methods can access the object fields provided the function is a non-static

[3] And can thus be loosely called *class function*.

function: an object method. The object field is accessed using the reserved keyword `this`.

Program 8.9 Object methods for the list

```
class ListObj
{
int container;
ListObj next;

ListObj(int element, ListObj tail)
  {this.container=element;
  this.next=tail;}
ListObj insert(int s)
{return new ListObj(s,this);}

// Method
int head()
{return this.container;}
}
```

In the `main` procedure, we could have then used the following instructions:

```
ListObj list=new ListObj(7,null);
list=list.insert(4);
System.out.println("Head:"+list.head());
```

In the remainder, we will adopt this object-oriented framework when designing data-structures, as they correspond to a true data encapsulation.[4] We will first present the (class) static function and then describe the corresponding object-oriented data-structure.

As concisely mentioned in the introduction, we made a difference between data-structures with static functions operating on them, and encapsulated methods acting directly on the object itself. We shall further describe these concepts of object-oriented programming and revisit former data-structures using that framework (replacing static functions by equivalent object methods).

To start with, consider the following example where one has to compute the volume of a 3D box. First, we define a `Box` object by writing its class encapsulating its data members: `width`, `height`, and `depth`. So far, the usual way to process a `Box` has been using static functions that were necessarily attached to a class. So let us define the `static double Volume(Box box)` function inside the body of the main class program. Note that this static function needs a `Box` object as argument. The second (and much better way) to program this functionality is to provide a *method* to the object that will dispose of all the object fields at the time it is called. We access the various object fields

[4] A second more advanced course will then purposely describe the `public`/`private` keyword syntax.

using the keyword `this`. The example below illustrates the essential difference between the two ways of programming the `Volume` functionality.

Program 8.10 Object-oriented method for computing the volume of a 3D box

```
class Box
{
double width, height, depth;

Box(double w, double h, double d)
{
this.width=w; this.height=h; this.depth=d;
}

double Volume()
{return this.width*this.height*this.depth;}
}

class OOstyle
{
static double Volume(Box box)
{return box.width*box.height*box.depth;}

public static void main(String[] s)
{
Box myBox=new Box(5,2,1);
System.out.println("Volume by static method:"+Volume(myBox));
System.out.println("Volume by object method:"+myBox.Volume())
    ;
}
}
```

8.5 Revisiting object-oriented style data-structures

8.5.1 Object oriented priority queues

The former `maxHeap/add/removeTop` static functions operating on the array can thus be translated as equivalent methods acting on the object. A heap object now declares the array as an object field without that static keyword. Therefore, we can build several heaps at a time. The prototypes for the heap methods are as follows:

Program 8.11 Heap: Prototyping object methods

```
int maxHeap()
{
return this.label[0];
}


void add(int element)
{
...
}

void removeTop()
{
...
}
```

8.5.2 Object-oriented lists

Similarly, we can revisit the static functions formerly defined on linked lists to fit the object-oriented programming paradigm as follows:

Program 8.12 Linked list with object methods

```
public class List
{
int element;
List next;

List(int el, List l)
{
this.element=el;
this.next=l;
}

static List EmptyList()
{
return new List(0,null);
}

boolean isEmpty()
{
return (this.next==null);
}

}
```

The other various functions of linked list interface are then redefined as the following object methods:

Program 8.13 Linked list the object-oriented style: Methods

```
int length()
{
List u=this;
int res=0;
while(u!=null)  {res++;u=u.next;}
return res-1;
}

boolean belongsTo(int el)
{
List u=this.next;
while(u!=null)
  {
  if (el==u.element) return true;
  u=u.next;
  }

return false;
}

void add(int el)
{List u=this.next;
this.next=new List(el,u);
}


void delete(int el)
{
List v=this;
List w=this.next;

while(w!=null && w.element !=el)
  {
  v=w;
  w=w.next;
  }
if (w!=null) v.next=w.next;
}

void display()
{
List u=this.next;
while(u!=null)
  {System.out.print(u.element+"->");
  u=u.next;}
System.out.println("null");
}

static List FromArray(int [] array)
{
  List u=EmptyList();
  for(int i=array.length-1; i>=0; i--)
```

```
    u.add(array[i]);
  return u;
}
```

A simple code demonstrating various calls of object methods is given below. Observe that the data-structure is not passed as an argument since it is encapsulated into the object as a field.

Program 8.14 Demonstrating various calls of object methods

```
public static void main(String[] args)
{
int [] array={2,3,5,7,11,13,17,19,23};

List u=FromArray(array);

u.add(1);

u.display();

u.delete(5);
u.display();

System.out.println(u.belongsTo(17));
System.out.println(u.belongsTo(24));
}
```

The result displayed in the console output is:

```
1->2->3->5->7->11->13->17->19->23->null
1->2->3->7->11->13->17->19->23->null
true
false
```

8.6 Stacks: Last in first out (LIFO) abstract data-structures

Stacks are *generic* data-structures for storing incoming elements. However, stacks depart from queues in the sense that the last arrived element is the first pulled out. This is why stacks are also called LIFO data-structures: Last in first out structures. The basic *interface*[5] of a stack is described by the two following methods:

– `Push`: Add an element into the stack,

[5] Java also provides in its API a generic stack implementation. See `http://java.sun.com/j2se/1.4.2/docs/api/java/util/Stack.html`

– `Pull` (or `Pop`): Remove the last inserted element from the stack.

These two basic methods of stacks can be implemented using various schemes. We describe two of them to insist on the notion of abstract data-structures: Arrays and linked lists.

8.6.1 Stack interface and an array implementation

We may implement a stack using an array. The array is declared as an object field with an index variable that indicates the position of the last inserted element: the top of the stack. The `Push`/`Pull` primitives of the *stack interface* are then implemented as follows:

Program 8.15 Stack implementation using an array

```
class StackArray
{
int nbmax;
int index;
int [ ] array;

// Constructors
StackArray(int n)
{
this.nbmax=n;
array=new int[nbmax]; index=-1;
System.out.println("Succesfully created a stack array object
    ...");
}

// Methods
void Push(int element)
{
if (index<nbmax-1)
  array[++index]=element; }

int Pull()
{
if (index>=0 ) return array[index--];
else return -1;
}

}
```

Let us store the above `StackArray` class into a corresponding Java source code text file: `StackArray.java`. Then compile this code for manipulating stacks using arrays:

```
prompt% javac StackArray.java
```

A byte-code file named `StackArray.class` encapsulating the object characteristics and its methods is produced into the current directory. We can now use this bytecode and object interface into a demonstration program stored into another text file, say `DemoStack.java`:

Program 8.16 Stack in action: A simple demonstration program

```
class DemoStack{

public static void main(String [] args)
{
  StackArray myStack=new StackArray(10);
  int i;

  for(i=0;i<10;i++)
     myStack.Push(i);

  for(i=0;i<15;i++)
     System.out.println(myStack.Pull());

}
```

Compiling and running this demonstration program, we get the following console output:

```
Succesfully created a stack array object...
9
8
7
6
5
4
3
2
1
0
-1
-1
-1
-1
-1
```

Let us now see how to implement the two methods of the stack interface using another kind of data-structure for its backbone: linked lists.

8.6.2 Implementing generic stacks with linked lists

Instead of using an array, let us now use a linked list for storing incoming elements. We first recall the list declaration and its basic primitives:

Program 8.17 Minimal list implementation

```
class List
{
int element;
List next;

// Constructor
List(int el, List tail)
{
this.element=el;
this.next=tail;
}

List insertHead(int el)
{
return new List(el,this);
}

}
```

Let us now use this linked list data-structure for supporting the methods provided by the stack:

Program 8.18 Implementing the stack interface using a linked list as its backbone data-structure

```
class Stack
{
List list;

Stack()
{
list=null;
}

void Push(int el)
{
if (list!=null)
    list=list.insertHead(el);
    else
    list=new List(el,null);
}

int Pull()
{int val;
if (list!=null)
  {val=list.element;
    list=list.next;}
    else val=-1;

return val;
}
}
```

Finally, we can now run the same demonstration program as for arrays. The result is identical since the *semantic* of the methods declared in the stack interface is the same. We get the following Java program:

Program 8.19 Demonstration program: Stacks using linked lists

```
class DemoStackList
{

public static void main(String [] args)
{
  Stack myStack=new Stack();
  int i;

  for(i=0;i<10;i++)
     myStack.Push(i);

  for(i=0;i<15;i++)
     System.out.println(myStack.Pull());

}

}
```

In summary, objects provide a nice framework for encapsulating data. The methods acting on object are non-static functions that can access the fields of the current object. Thus methods can be interpreted as basic algorithms on structured data.

8.7 Exercises

Exercise 8.1 (Stacks of strings)

Implement a stack for storing strings. Write static functions `push` and `pop`. Then provide the equivalent methods for that class, and discuss the advantages of using object methods.

Exercise 8.2 (Flood-filling binary images using stacks and queues)

Consider a binary image defined by a bi-dimensional array of booleans: `boolean [] [] image;`. Provide a function `drawRectangle` that draws the border of a rectangle given by its bottom left corner and with/height dimensions. Create a binary image by drawing at random n rectangle borders using the function `Math.random()`. We would like to perform the flood-filling at image pixel (x, y). Implement the recursive function `FloodFillingStack` by using a pixel stack. Similarly, implement the

recursive function `FloodFillingQueue` by using a queue. Which implementation technique is the most *memory* efficient?

Exercise 8.3 (Sorting using a priority queues)

Consider sorting a set of strings stored initially in an array `String [] array` using a priority queue defined with respect to the lexicographic order on strings. Implement the priority queue for `String` elements. Then sort the initial array of strings by first inserting all strings into the priority queue, and then successively retrieving them. What is the time and memory complexity of sorting using priority queues?

9
Paradigms for Optimization Problems

9.1 Introduction

In this chapter, we consider fundamental optimization problems that occur in many real-life applications. The few selected core optimization problems are very representative of a broad class of mathematical problems encountered in practice. A major characteristic of these problems is that they are all *combinatorial* by essence. This means that the optimization algorithms described in the following are not approaching an optimal solution numerically (by say, a Newton-like gradient descent method as seen previously in Chapter § 2), but rather exploring and searching for *exact solutions* in large but finite discrete *configuration spaces*. The optimization techniques presented in the remainder are broad enough that their underlying schemes can be used for solving various problems; these different kinds of solving methodologies are called *paradigms* since they yield *generic algorithms* for tackling many similar problems.

We first start by describing the naive brute force method, called *exhaustive search*, which consists of visiting all positions of configuration spaces. We then show how plugging a few structural constraints emanating directly from problems' properties allows one to significantly shrink the number of inspected configurations using a so-called *backtracking mechanism* implicitly implemented by recursion. As the size of problems grows, these exhaustive search and backtracking algorithms unfortunately suffer from browsing these gigantic domains. These exponentially large configuration spaces are too costly to explore, and solving these problems in practice become *intractable*. To

F. Nielsen, *A Concise and Practical Introduction to Programming Algorithms in Java*, Undergraduate Topics in Computer Science, DOI 10.1007/978-1-84882-339-6_9,

circumvent this inherent difficulty, we design faster algorithms that only report *approximate solutions* instead of too-costly exact solutions. We end up by presenting the most celebrated *heuristic* for getting *guaranteed* approximate solutions: the *greedy strategy*. We exemplify the greedy methodology on two problems: the knapsack and set cover problems,, which emphasize the important fact that the same paradigm applied to these two distinct problems yields different *approximation ratios*.

9.2 Exhaustive search

The paradigm of exhaustive search consists merely in exploring the full configuration space by enumerating one by one all possible configurations, and retaining the best one: the optimal solution. Eventually several optimal solutions may exist. Exhaustive search is clearly the *brute-force* paradigm that yields straightforward but often naive algorithms for solving a task at hand. Let us explain its implementation by taking a case study: the knapsack.

9.2.1 Filling a knapsack

Consider a set $\mathcal{O}$ of n objects $O_1, ..., O_n$ with corresponding weights $W_1, ..., W_n$. Suppose that object O_i weights W_i, where W_i is measured in kilogram units for all $i \in \{1, ..., n\}$. Given a knapsack that can carry an *overall* weight W, we are asked to *enumerate all possible choices* for *fully* filling this sack. Objects can be chosen only once; There is no allowed multiplicity of objects. In other words, we are asked to find all possible object *selections* that yield an overall weight of exactly W kilograms. W is called the *capacity* of the bag.
We can formulate this problem as computing

$$\mathcal{I}^* = \left\{ I \subseteq \{1, ..., n\}, \ \sum_{i \in I} W_i = W \right\},$$

where I denote a subset of indices representing the object selection. That is, find all index subsets so that their corresponding object weights sum up to the knapsack capacity.
If we were given *a priori* a fixed number of objects n, we could simply check all possible choices of selecting or not selecting objects by writing a sequence of nested loops. For example, consider completely filling a knapsack of capacity $W = 11$ by selecting objects in a set of $n = 5$ objects with the following

corresponding weights $W_1 = 3$, $W_2 = 4$, $W_3 = 5$, $W_4 = 6$ and $W_5 = 2$. Let us visualize the input as the following table:

O_i	1	2	3	4	5
W_i	3	4	5	6	3

The source code for fully exploring the combinatorial space of solutions by programming nested loops is presented below:

Program 9.1 Plain enumeration using nested loops

```
class KnapsackNestedLoops
{
public static void main(String [] argArray)
{
int W1=3, W2=4, W3=5, W4=6, W5=2; // respective weights
int W=11; // weight capacity of the knapsack
int bagWeight;//  weight of 'current' configuration

for(int i1=0;i1<=1; i1++)
  {for(int i2=0;i2<=1; i2++)
    {for(int i3=0;i3<=1; i3++)
      {for(int i4=0;i4<=1; i4++)
          {for(int i5=0;i5<=1;i5++)
            {
            bagWeight=i1*W1+i2*W2+i3*W3+i4*W4+i5*W5;

            // Does the current selection match the sack
                capacity
            if (bagWeight==W)
              {System.out.println("Solution:"+i1+" "+i2+" "+
                  i3+" "+i4+" "+i5);}
            }
          }
      }
    }
  }
}

}
```

Compiling the code (using console command `javac KnapsackNestedLoops.java`) and executing the above program (using console command `java KnapsackNestedLoops`), we get *all* possible solutions, reported using `true`/`false` memberships of objects:

```
Solution:0 0 1 1 0
Solution:0 1 1 0 1
Solution:1 0 0 1 1
```

That is, the first solution 0 0 1 1 0 reads as select object O_3 and O_4 (but not objects O_1 nor O_4): The subset $\{O_3, O_4\}$ of $\mathcal{O}$ encoded by the index subset

$\{3,4\}$ (see Eq. 9.2.1). We check that we indeed have $W_3+W_4 = 5+6 = 11 = W$. Thus the set of solutions can be read back from the bit memberships, as follows:

$$\mathcal{I}^* = \{\{3,4\},\{2,3,5\},\{1,4,5\}\}\,.$$

The exhaustive search algorithm seeks for all potential solutions by fully exploring the configuration space. Each configuration is modeled by a set of "loop indices" telling us whether or not we choose the corresponding object. This shows that there are 2^n possible configurations, ranging from the empty set[1] to the complete set $\mathcal{O}$, encoded by the full set of indices $\{1,...,n\}$. Observe that if we were to add a sixth object, we would need to add another nested loop directly *inside* the code in order to solve for solutions. This clearly stresses the limits of the "nested loop" method since we clearly do not want to manually edit the program source code every time we have a different problem size to solve. Furthermore, imagine if you were asked to write the nested loop program for $n = 100$. That would be daunting. Therefore, the number of objects n shall be considered as a parameter of the filling knapsack problem itself.

How do we cope with such a situation? We need to reconsider the configuration space, and model it appropriately. The potential solution space is modeled by the *power set* $2^{\mathcal{O}} = \{\mathcal{O}', \quad \mathcal{O}' \subseteq \mathcal{O}\}$ that includes all possible subsets of $\mathcal{O}$, including the empty set. As highlighted above, to each subset $\mathcal{O}' \subseteq \mathcal{O}$, we associate a binary signature $b(\mathcal{O}')$ of length n such that the l-th bit is set to 1 (or boolean `true` state) if and only if object O_i belongs to $\mathcal{O}'$. The configuration space is thus in a one-to-one mapping with the space of bit signatures: The set of binary numbers of n bits (with evaluated values ranging from 0 to $2^n - 1$). In other words, the binary signature depicts the configuration number whose binary representation is that signature. This shows that the size of the configuration space is 2^n; It grows *exponentially fast* with n. Incrementing n doubles the size of the configuration space.

To perform an exhaustive search of the configuration space for any arbitrary value n of objects, we first proceed by enumerating all possible binary representation numbers with n bits. Enumerating all binary representations with n bits can be done using the following recursive algorithms:

Program 9.2 Enumerating all 2^n binary number representations with n bits

```
class Enumeration
{

static void display(boolean [] tab)
{
for(int i=0;i<tab.length;i++)
```

[1] Obviously not a solution since its weight is trivially zero, but nevertheless considered when browsing the configuration space by these nested loops.

```
  {if (tab[i])
    System.out.print("1 ");
    else
    System.out.print("0 ");}

System.out.println("");
}

static void Enumerate(boolean [] selection, int pos)
{
if (pos==selection.length-1)
    {display(selection);} // terminal case, reach length n
  else
  {
  pos++;
  // Set the (pos+1)th bit to 1
  selection[pos]=true;
  Enumerate(selection,pos);
  // Set the (pos+1)th bit to 0
  selection[pos]=false;
  Enumerate(selection,pos);
  }
}

public static void main(String[] args)
{
int n=4; int i;
// Binary representation of numbers
boolean [] select=new boolean[n];
for(i=0;i<n;i++)
  {select[i]=false;} // optional since array creation set it
      to all zero
// Launching the enumeration from the first bit (index 0)
Enumerate(select,-1);
}

}
```

Running the program by executing the compiled byte code (javac Enumeration; java Enumeration), we get the following output:

```
1 1 1 1
1 1 1 0
1 1 0 1
1 1 0 0
1 0 1 1
1 0 1 0
1 0 0 1
1 0 0 0
0 1 1 1
0 1 1 0
0 1 0 1
0 1 0 0
0 0 1 1
```

```
0 0 1 0
0 0 0 1
0 0 0 0
```

To solve the filling knapsack problem, we now need to check at the terminal states whether the overall knapsack weight of currently selected objects matches the capacity weight of the knapsack or not.

Program 9.3 Exhaustive search for the perfect filling of a knapsack

```
class KnapSack
{
final static int n=10; // 10 objects
static int [] weight={2,3,5,7,9,11,4,13,23,27};

static void display(boolean [] selection, int val)
{String msg="";
  for(int i=0;i<selection.length;i++)
    {if (selection[i])
      msg=msg+weight[i]+" ";}
  System.out.println(msg+"="+val);
}

static void solveKnapSack(boolean [] chosen, int goal, int i,
    int total)
{
if ((i>=chosen.length)&&(total!=goal))
    return;

if (total==goal)
  {display(chosen, goal);}
else
  {
  chosen[i]=true;// add item first and proceed
  solveKnapSack(chosen,goal,i+1,total+weight[i]);

  chosen[i]=false;  // and then remove it and proceed
  solveKnapSack(chosen,goal,i+1,total);
  }
}


public static void main(String [] args)
{
int totalweight=51;
boolean [] chosen=new boolean[n];// initialized to all false

solveKnapSack(chosen, totalweight, 0, 0);
}

}
```

All solutions are then properly listed as below:

```
2 3 5 7 11 23 =51
2 5 7 9 11 4 13 =51
2 5 4 13 27 =51
2 7 11 4 27 =51
2 9 4 13 23 =51
2 9 13 27 =51
3 5 7 9 4 23 =51
3 5 7 9 27 =51
3 5 7 13 23 =51
3 5 9 11 23 =51
7 4 13 27 =51
9 11 4 27 =51
11 4 13 23 =51
11 13 27 =51
```

Exhaustive search is *not* an effective algorithmic technique since it fully explores the solution space to retrieve all potential solutions. The solution space is indeed a very sparse subset, a tiny fraction of the configuration space. For example, we found 14 solutions among a configuration space of size $2^{10} = 1024$. Furthermore, note that if all object weights add up to less than the capacity weight, we can obviously avoid performing the search since we already know that there is no possible solution. Thus the plain implementation of exhaustive search does extra work.

In order to avoid *some* of the unnecessary explorations, we can implement a *cut* in recursive function solveKnapSack. Indeed, if at some stage of the recursion, we already know that the current bag weight exceeds the capacity, it is not worth adding other remaining objects that can only add weight to the bag. We can therefore stop the recursion at these stages. To measure how many times this cut was applied in practice, we add a static (class) variable initialized at zero: **static int** nbCut=0; The exploration procedure with the cut now reads as:

Program 9.4 Adding a cut to the recursive exhaustive search

```
static void solveKnapSack(boolean [] chosen, int goal, int i,
    int total)
{
if (total>goal)
  {nbCut++;
  return; // cut recursive explorations
  }
if ((i>=chosen.length)&&(total!=goal))
   return;
if (total==goal)
  {display(chosen, goal);}
else
  {
  chosen[i]=true;// add item first and proceed
  solveKnapSack(chosen,goal,i+1,total+weight[i]);
  chosen[i]=false;  // and then remove it and proceed
```

```
    solveKnapSack(chosen, goal, i+1, total);
    }
}
```

Running this program, we find that it reported 396 cuts.

9.2.2 Backtracking illustrated: The eight queens puzzle

The queen puzzle consists of placing eight queens on an 8×8 chessboard so that the queens are in a *mutually* safe position. That is, none of the queens is able to capture any other. Figure 9.1 graphically depicts such a solution. Historically, finding whether there exist solutions was first asked by German chess player Max Bezzel in 1848. A brute force algorithm will enumerate all possible ways of putting the eight queens on a chessboard. The exhaustive search paradigm will thus explore...

$$\left(\begin{array}{c} 64 \\ 8 \end{array} \right) = \frac{64 \times 63 \times ... \times 57}{8!} = 283,274,583,552$$

...different configurations: There are 64 choices for placing the first queen, and then 63 choices for the second, up to $57 = 64 - 8 + 1$ choices for placing the last queen. Since queens are not distinguishable, we divide the former number by the number of queen permutations: 8!.

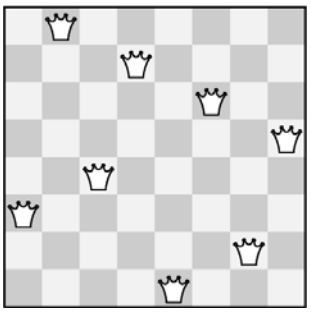


Figure 9.1 Illustrating one of the 92 distinct solutions to the eight queen puzzle

Thus a direct implementation of exhaustive search would not be efficient since the number of recursive calls is too important. We are going to exploit some *structural properties* of the eight queen problem to guide the exploration. Observe that trivially two queens cannot be located on the same row of the chessboard. Therefore a much better exhaustive search will consider only $8^8 = 2^{24} = 16,777,216$ configurations (for each queen, there are only 8 potential

positions on its given row). Let the ith queen column position be stored in an integer array queen. That is, the position of the ith queen is stored at coordinates (i,queen[i]) (with queen indices ranging from 0 to 7 to follow Java array conventions). Suppose we have already correctly placed l queens on the first l rows so that these l queens are pairwise safe. For the $(l+1)$th queen necessarily located at row $l+1$, we are going to check in turn all column positions, starting from column 0 up to column 7 until we find a safe position for the queen with respect to the previously located ones. Once such a position found, we proceed to the next queen lying on the next line, and so on. Whenever we reach at some point all column positions without finding a solution, then we need to *backtrack* by increasing the column of the previous queen (located at the row below the current queen), and so on. The recursive exploration with backtracking is detailed in the source code below. The recursive procedure is initially launched by calling in the main procedure function search(0).

Program 9.5 Eight queen puzzle: Search with backtracking

```
static boolean search(int row)
{
boolean result=false;
if (row==n)
  {displayChessboard();
  nbsol++; }
  else
  {int j=0;
    while(!result && j<n)
    {
      if (safeMove(row,j))
      {
        queen[row]=j;
        result=search(row+1);
      }
      // Backtracking here
      j++; // explore all columns
    }
    }
  return result;
}
```

To check whether two queens are in mutually safe positions or not, we check whether these two queens lie on the same row or column, or whether they share a common diagonal. This can be compactly written as:

Program 9.6 Check whether two queens are in mutually safe position or not

```
static boolean wrongPos(int i1, int j1, int i2, int j2)
{
return ( i1==i2 ||
         j1==j2 ||
         Math.abs(i1-i2) == Math.abs(j1-j2) );
```

```
}
```

It follows that we can check when we add the new i-th queen on the chessboard whether its position is safe or not with respect to all others ($k < i$) as follows:

Program 9.7 Check whether the i-th queen is safe wrt. the k-th queens, for $k < i$

```
static boolean safeMove(int i, int j)
{boolean result=true;
for(int k=0;k<i;k++)
  result=result && !wrongPos(i,j,k,queen[k]);
return result;  }
```

Finally, let us give the exhaustive search procedure with the recursive backtracking mechanism for the n queens, including a chessboard display procedure called every time a solution is found:

Program 9.8 Solving the 8-queen puzzle

```
class Queen
{
static final int n=8;
static int [] queen=new int[n]; // position (i,queen[i])
static int nbsol;

static void displayChessboard()
{
int i,j;

System.out.println("");

for(i=0;i<n;i++)
  {
    for(j=0;j<n;j++)
    {
      if (queen[i]!=j) System.out.print("0");
        else System.out.print("1");
    }
    System.out.println("");
  }
}

static boolean wrongPos(int i1, int j1, int i2, int j2){...}
static boolean safeMove(int i, int j){...}
static boolean search(int row){...}

public static void main(String [] arguments)
{
nbsol=0;
search(0);
System.out.println("Total number of solutions:"+nbsol);
}
```

```
}
```

Compiling and running this program, we obtain the well-known 92 distinct solutions that are displayed in the console using a 2D binary matrix indicating the queen locations as shown below:

...

```
00000001
00100000
10000000
00000100
01000000
00001000
00000010
00010000

00000001
00010000
10000000
00100000
00000100
01000000
00000010
00001000
Total number of solutions:92
```

It turns out that only 12 of these solutions are non-naive since others can be deduced from this reduced set of solutions by using rotation, flipping or symmetry operations. You can try the program on any arbitrary chessboard size, say of width size $n = 10$, and observe the number of solutions (724 distinct configurations for $n = 10$).

9.3 Greedy algorithms: Heuristics for guaranteed approximations

9.3.1 An approximate solution to the 0-1 knapsack problem

We revisit the former filling knapsack problem of § 9.2.1 where we are given a set $\mathcal{O}$ of n objects $O_1, ..., O_n$ called items and a knapsack to fill up to its maximal weight capacity W. But this time we consider that each object O_i, besides its weight W_i, has yet another attribute: U_i, its *utility*. The 0-1 knapsack

problem is a packing problem where one seeks for the best selection of objects such that:

1. the overall sum of the selected objects is less or equal to the knapsack capacity W, and
2. the overall utility of selected items is *maximized*.

This can be written mathematically as the following optimization problem:

$$\max_{I \subseteq \{1,\ldots,n\}} \sum_{i \in I} U_i, \tag{9.1}$$

$$\text{such that } \sum_{i \in I} W_i \leq W. \tag{9.2}$$

Figure 9.2 depicts an instance of the optimization knapsack problem.

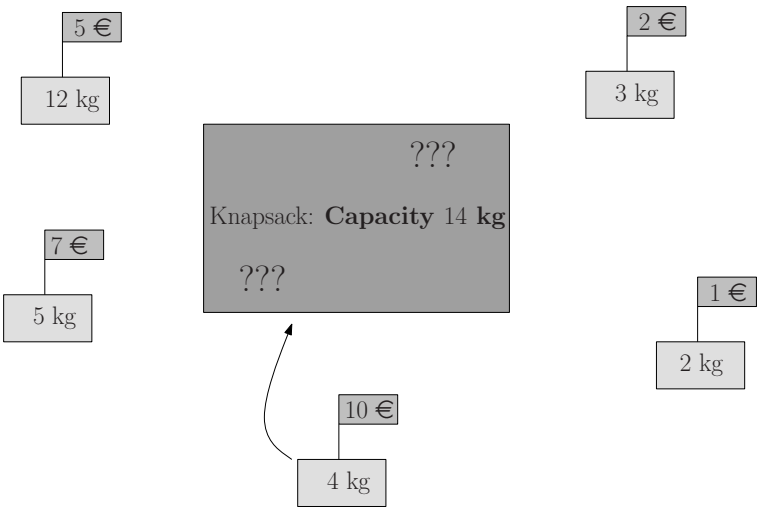


Figure 9.2 The 0-1 knapsack problem consists of finding the set of items that *maximizes* the overall utility while fitting the knapsack capacity

Exhaustively checking all 2^n subsets of $\mathcal{O}$ (power sets $2^{\mathcal{O}}$) as we did before takes too long and faces intrinsic limitations as soon as $n \sim 30 - 40$, due to the exponential size of the configuration space. We prefer to design a *heuristic* for the task that will report not an exact solution, but rather a guaranteed approximation. The basic idea of the greedy algorithm is to first choose the object O_l that maximizes the ratio $G_l = \frac{U_i}{W_i}$ of utility U_l over its weight W_l, remove this object from the set of items, and repeat until we reach the maximum capacity W of the knapsack, or when we cannot add any more objects without

exceeding the capacity. This is a greedy approximation heuristic that selects objects one by one. Once an object is taken it is not considered any more, and the optimization heuristic proceeds with the remaining objects. A choice is therefore definitive and irrevocable. The ratio factor of the utility over the weight is called the *gain* of objects.
Consider solving a 0-1 knapsack with capacity $W = 20$ and the following set of $n = 10$ (ten) items:

i	1	2	3	4	5	6	7	8	9	10
U_i	5	3	8	12	5	7	9	2	3	6
W_i	3	5	2	4	7	3	2	3	5	7
G_i	$\frac{5}{3}$	$\frac{3}{5}$	4	3	$\frac{5}{7}$	$\frac{7}{3}$	$\frac{9}{2}$	$\frac{2}{3}$	$\frac{3}{5}$	$\frac{6}{7}$

Program 9.9 Greedy approximation algorithm for solving the 0-1 knapsack

```
class GreedyKnapsack
{
static int n=10;
static int [] W={5, 3, 8, 12, 5, 7, 9, 2, 3, 6};
static int [] U={3, 5, 2, 4,  7, 3, 2, 3, 5, 7};
static int wKnapsack=20;

// Greedy algorithm for approximating a best solution
public static void main(String [] argArray)
{
double[] gain=new double[n];
double g;
int nSel=0;
int u=0;
int w=0;
int i, winner;
boolean [] chosen=new boolean[n];

System.out.println("Maximum weight of the knapsack:"+
    wKnapsack);

for(i=0;i<n;i++)
  {
  gain[i]=U[i]/(double)W[i];
  chosen[i]=false;
  }

  while(nSel<=n && w<wKnapsack)
  {
  g=0.0d; winner=-1;

  for(i=0;i<n;i++)
  {
```

```
  if (!chosen[i] && w+W[i]<=wKnapsack)
    System.out.println(i+" Weight "+W[i]+" Utility "+U[i]+"
        Gain["+i+"]="+gain[i]);
  }

  // Search best gain in the remaining collection of objects
  for(i=0;i<n;i++)
    if (!chosen[i] && gain[i]>g && wKnapsack-(w+W[i])>=0)
      {g=gain[i];winner=i;}

  // Can we add this object to the knapsack
  if (winner!=-1 && w+W[winner]<wKnapsack)
    {
    u+=U[winner];
    w+=W[winner];
    chosen[winner]=true;
    nSel++;
    System.out.println("Selected object "+winner+", updating
        #weights="+w+" #utility="+u);
    }
  else break;
  }
}
}
```

Running the greedy knapsack heuristic on the set of above items, we get the following output at the console:

```
Maximum weight of the knapsack:20
0 Weight 3 Utility 5 Gain[0]=1.6666666666666667
1 Weight 5 Utility 3 Gain[1]=0.6
2 Weight 2 Utility 8 Gain[2]=4.0
3 Weight 4 Utility 12 Gain[3]=3.0
4 Weight 7 Utility 5 Gain[4]=0.7142857142857143
5 Weight 3 Utility 7 Gain[5]=2.3333333333333335
6 Weight 2 Utility 9 Gain[6]=4.5
7 Weight 3 Utility 2 Gain[7]=0.6666666666666666
8 Weight 5 Utility 3 Gain[8]=0.6
9 Weight 7 Utility 6 Gain[9]=0.8571428571428571
Selected object 6, updating #weights=2 #utility=9
0 Weight 3 Utility 5 Gain[0]=1.6666666666666667
1 Weight 5 Utility 3 Gain[1]=0.6
2 Weight 2 Utility 8 Gain[2]=4.0
3 Weight 4 Utility 12 Gain[3]=3.0
4 Weight 7 Utility 5 Gain[4]=0.7142857142857143
5 Weight 3 Utility 7 Gain[5]=2.3333333333333335
7 Weight 3 Utility 2 Gain[7]=0.6666666666666666
8 Weight 5 Utility 3 Gain[8]=0.6
9 Weight 7 Utility 6 Gain[9]=0.8571428571428571
Selected object 2, updating #weights=4 #utility=17
0 Weight 3 Utility 5 Gain[0]=1.6666666666666667
1 Weight 5 Utility 3 Gain[1]=0.6
3 Weight 4 Utility 12 Gain[3]=3.0
4 Weight 7 Utility 5 Gain[4]=0.7142857142857143
```

```
5 Weight 3 Utility 7 Gain[5]=2.3333333333333335
7 Weight 3 Utility 2 Gain[7]=0.6666666666666666
8 Weight 5 Utility 3 Gain[8]=0.6
9 Weight 7 Utility 6 Gain[9]=0.8571428571428571
Selected object 3, updating #weights=8 #utility=29
0 Weight 3 Utility 5 Gain[0]=1.6666666666666667
1 Weight 5 Utility 3 Gain[1]=0.6
4 Weight 7 Utility 5 Gain[4]=0.7142857142857143
5 Weight 3 Utility 7 Gain[5]=2.3333333333333335
7 Weight 3 Utility 2 Gain[7]=0.6666666666666666
8 Weight 5 Utility 3 Gain[8]=0.6
9 Weight 7 Utility 6 Gain[9]=0.8571428571428571
Selected object 5, updating #weights=11 #utility=36
0 Weight 3 Utility 5 Gain[0]=1.6666666666666667
1 Weight 5 Utility 3 Gain[1]=0.6
4 Weight 7 Utility 5 Gain[4]=0.7142857142857143
7 Weight 3 Utility 2 Gain[7]=0.6666666666666666
8 Weight 5 Utility 3 Gain[8]=0.6
9 Weight 7 Utility 6 Gain[9]=0.8571428571428571
Selected object 0, updating #weights=14 #utility=41
1 Weight 5 Utility 3 Gain[1]=0.6
7 Weight 3 Utility 2 Gain[7]=0.6666666666666666
8 Weight 5 Utility 3 Gain[8]=0.6
Selected object 7, updating #weights=17 #utility=43
```

Here, the maximum utility reported for the input data set is 43. Let c^* denote the best overall utility of an optimal solution and c_G be the overall utility reported by the greedy algorithm. Then Dantzig proved in 1957 that $c_G \geq \frac{c^*}{2}$ (of course $c_G \geq c^*$ since an approximation can only be worse than an exact solution). For the 0-1 knapsack problem, we say that greedy approximation has an approximation factor of 2. Although the greedy heuristic proceeds by greedily choosing the current best element and reiterating until completion, its performance factor may vary according to the problem at hand. The following set cover problem explained in the next section illustrates this important fact.

9.3.2 A greedy algorithm for solving set cover problems

The set cover problem (SCP for short) is yet another fundamental combinatorial optimization problem that arises in many real-world applications. The problem is defined as follows; We are given a set $\mathcal{X}$ of n elements $\mathcal{X} = \{X_1, ..., X_n\}$ and a collection $\mathcal{S}$ of m subsets of $\mathcal{X}$: $\mathcal{S} = \{S_1, ..., S_m\}$ called *ranges*. The goal is to select a minimum number of sets of $\mathcal{S}$ so that their union covers all elements of $\mathcal{X}$. That is, we are asked to solve the following mathematical optimization problem:

$$\min_{I \subseteq \{1,\ldots,n\}} |I| \tag{9.3}$$

$$\text{such that } \bigcup_{i \in I} S_i = X. \tag{9.4}$$

In the literature, the input pair $(\mathcal{X}, \mathcal{S})$ is called a *range space*.

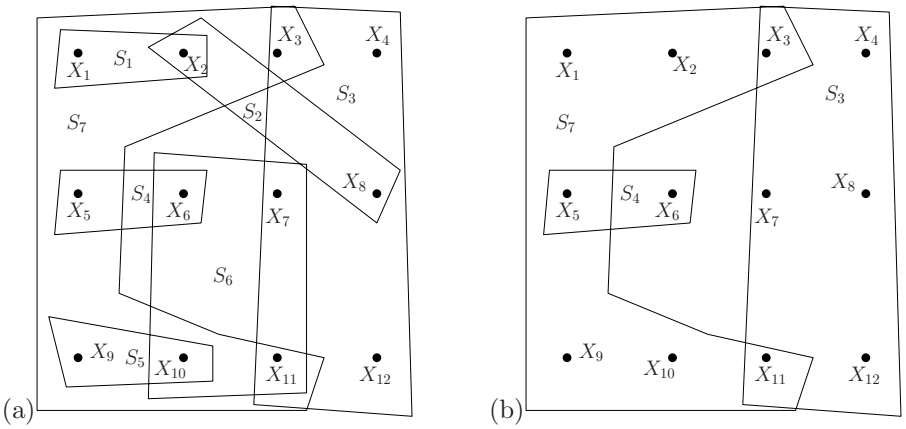


Figure 9.3 Example of an instance of a set cover problem: (a) input range space: set $\mathcal{X} = \{X_1, ..., X_{12}\}$ of 12 elements and a collection $\mathcal{S} = \{\{X_1, X_2\}, \{X_5, X_6\}, \{X_9, X_{10}\}, \{X_2, X_8\}, \{X_6, X_7, X_{10}, X_{11}\}, \{X_1, X_2, X_3, X_5, X_9, X_{10}, X_{11}\}, \{X_3, X_4, X_7, X_8, X_{11}, X_{12}\}\}$ of 7 subsets, and (b) optimal covering of size 3 since elements X_1, X_4 and X_6 are covered once by a different subset

As a real-world application of SCP, consider $\mathcal{X}$ denoting all grid elements of a digital terrain, and S_i denoting the grid cell covered by the ith base transceiver station. In the telecommunication industry, the goal is to minimize the number of selected base transceiver stations so that we can fully cover all grid cells (that is, deserve set $\mathcal{X}$). In practice, the problem is relaxed to the partial SCP by requiring only a fraction of grid cells to be covered, and associating various costs to selecting this or that base transceiver station. But this does not change in any way the essence of the SCP. Figure 9.4 displays the result of solving a (partial) set cover problem for this telecommunication problem on an urban city scene.

Once again, the greedy heuristic previously sketched yields a simple optimization algorithm for finding a covering: We choose the range of $\mathcal{S}$ that covers the most number of elements of $\mathcal{X}$, remove the covered elements from both $\mathcal{X}$ and all subsets $\mathcal{S}$, and reiterate until none of the grid elements remain. This greedy

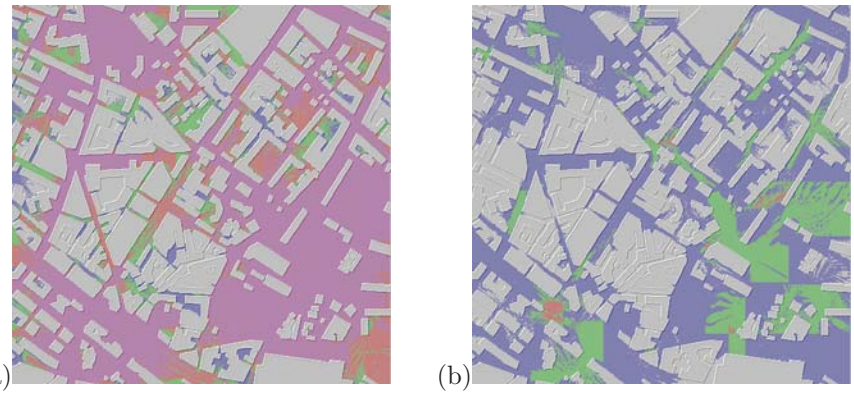


Figure 9.4 Set cover problem for urban radio network planning of mobile phones: (a) Covering of 99 base transceiver stations, and (b) covering using only 19 base stations. Here, some redundant areas covered more than four times

algorithm does not yield an optimal solution but guarantees an approximation solution. Let $c^* = |I^*|$ be the size of any optimal solution,[2] and $c = |I_G|$ denote the size of the solution reported by this greedy strategy. Then we have the following bounds:

$$c^* \leq c_G \leq c^*(1 + \log n).$$

To implement the greedy algorithm for the set cover problem, we first need to organize the data-structures for coding the range space $(\mathcal{X}, \mathcal{S})$. We choose to represent subsets of $\mathcal{X}$ using a *boolean incidence matrix* B. Matrix element $B[i, j]$ is set to `true` if and only if element X_j belongs to set S_i. The ranges are therefore encoded in the matrix rows while columns depict element membership in respective ranges. The basic code for creating and initializing an instance of a set cover problem is given in the following class SetCover.

Program 9.10 Initializing a set cover problem

```
class SetCover
{
int nbelements;
int nbsubsets;
boolean [][] incidenceMatrix;
SetCover(int nn, int mm)
{
this.nbelements=nn;
this.nbsubsets=mm;
incidenceMatrix=new boolean[nbsubsets][nbelements];
```

[2] Note that several solutions of same size may exist.

```
for(int i=0;i<nbsubsets;i++)
  for(int j=0;j<nbelements;j++)
    incidenceMatrix[i][j]=false;
}
void SetSubsets(int [] [] array)
{
for(int j=0;j<array.length;j++)
{
  for(int i=0;i<array[j].length;i++)
  incidenceMatrix[j][array[j][i]]=true;
}
}
void Display()
{
for(int i=0;i<nbsubsets;i++){

  for(int j=0;j<nbelements;j++)
    if (incidenceMatrix[i][j]) System.out.print("1");
      else System.out.print("0");
      System.out.println("");
      }
}
}
```

The greedy algorithm described above needs:

1. To find the set covering the greatest number of (not yet covered) elements,
2. To update the boolean incidence matrix once a maximal set of $\mathcal{S}$ has been chosen.

We implement these basic operations by inserting the following code in the body of class SetCover.

Program 9.11 Basic operations for supporting the greedy algorithm

```
// Number of covered element by subset i
int Cover(int i)
{
int nbEl=0;
for(int j=0;j<nbelements;j++)
    if (incidenceMatrix[i][j]) ++nbEl;
return nbEl;
}

// Report the current largest subset
int LargestSubset()
{
int i, nbel, max, select;
max=-1;select=-1;
for(i=0;i<nbsubsets;i++)
  {
  nbel=Cover(i);
  if (nbel>max) {max=nbel; select=i;}
```

```
  }
return select;
}

// Update the incidence matrix
void Update(int sel)
{
int i,j;
for(i=0;i<nbsubsets;i++)
{
  if (i!=sel)
    {
    for(j=0;j<nbelements;j++)
    if (incidenceMatrix[sel][j]) incidenceMatrix[i][j]=false;
    }
}
for(j=0;j<nbelements;j++)
    incidenceMatrix[sel][j]=false;
}
```

An initial set cover problem is created by building a new object of type SetCover, as follows:

Program 9.12 Creating an instance of a set cover problem

```
int [] [] subsets ={{0,1,3},{2,3,4},
   {0,2,5},{1,2,4},{3,4,5},{0,2}};
SetCover setcover=new SetCover(6,6);
setcover.SetSubsets(subsets);
System.out.println("Set cover problem:");
setcover.Display();
```

Finally, The greedy set cover algorithm is concisely written as follows:

Program 9.13 Greedy algorithm for solving SCPs

```
  static boolean [] GreedySCP(SetCover problem)
  {
  boolean [] result=new boolean[problem.nbsubsets];
  int cover=0;
  int select;

  for(int i=0;i<problem.nbsubsets;i++)
      result[i]=false;

  while(cover!=problem.nbelements)
  {
  // Choose largest not-yet covered subset
  select=problem.LargestSubset();
  result[select]=true;

  // Update covered matrix
  cover+=problem.Cover(select);
```

```
  // Update incidence matrix
  problem.Update(select);

  System.out.println("Selected "+select+" Number of covered
      elements="+cover);
  problem.Display();
  }

  return result;
  }
```

```
Set cover problem:
110100
001110
101001
011010
000111
101000
Selected 0 Number of covered elements=3
000000
001010
001001
001010
000011
001000
Selected 1 Number of covered elements=5
000000
000000
000001
000000
000001
000000
Selected 2 Number of covered elements=6
000000
000000
000000
000000
000000
000000
Solution: 0 1 2
```

To see that a greedy algorithm yields a $O(\log n)$ approximation factor, consider the SCP described in Figure 9.5. The two long subsets fully cover all elements, but the greedy heuristic will pick up the rightmost largest subset of 8 elements, and then the remaining largest subset of 4 elements, etc., until it picks the last subset of two elements. This instance is plugged into the program by replacing the former subsets array by:

```
int [] [] subsets={{0,1,2,3,4,5,6},{7,8,9,10,11,12,13},
            {0,7},{1,2,8,9},{3,4,5,6,10,11,12,13}};
SetCover setcover=new SetCover(14,5);
```

For this well-designed example, we have $c^* = 2$ but $c_G = \log \frac{n}{2}$, showing the upper bound of $O(\log n)$ approximation factor. Note that this approximation factor depends on n, the problem size. Remember that the 0-1 knapsack the same greedy heuristic yielded a *constant* approximation factor.

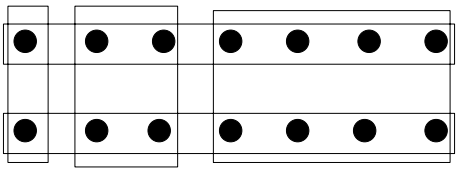


Figure 9.5 A bad instance for which the greedy set cover heuristic poorly behaves. This generic construction yields an approximation factor of $O(\log n)$

Although that we have noticed that the greedy set cover heuristic yields only a $O(\log n)$ competitive ratio, it is striking to know that no other algorithm can beat this simple heuristic provided that some problem complexity property holds in computer science, related to the famous $P \neq NP$? conjecture.

9.4 Dynamic programming: Optimal solution for the 0-1 knapsack problem

To close this chapter on programming various combinatorial optimization algorithms in Java, we will revisit the 0-1 knapsack problem. This time, we are looking for an *exact solution* (not the constant factor approximation solution we obtained from the greedy algorithm) without performing the prohibitive exhaustive search. The conceptual idea to make this possible is to come up with a mathematical *recurrence relation* that can then be used to *incrementally* build a bi-dimensional table from which we can retrieve the solution. Loosely speaking, the 2D table will store[3] intermediate solutions. Consider $u(i, j)$ the utility function defined by selecting items among the first i objects with an overall weight constraint of j units. The recurrence relation is found as follows:

- $u(1, w) = 0$ if $w < W_1$ (not enough room to put O_1 in the sack) and $u(1, w) = U_1$ whenever $w \geq W_1$.

[3] Eventually, only a subset of the table may be computed by a process called *memoization* in dynamic programming.

– At $u(i,j)$ either we did not select the i-th element and then $u(i,j) = u(i-1,j)$, or we selected it and then we need to know the utility of $u(i-1, j-W_i)$ to compute the total utility $U_i + u(i-1, j-W_i)$.

Thus we get the key recurrence relation:

$$u(i,j) = \begin{cases} u(i,j) = 0 \text{ for } j < W_1, \text{ and } u(i,j) = U_1 \text{ for } j \geq W_1, & i = 1 \\ u(i,j) = \max\{u(i-1,j) + u(i-1, i-W_i) + U_i\}, & i > 1. \end{cases} \tag{9.5}$$

Solving the knapsack problem by using dynamic programming consists of building a 2D table of size $n \times (W+1)$ (starting at weight 0) as follows:

Program 9.14 Dynamic programming for solving the 0-1 knapsack

```
static void SolveDP()
{
int i,j;
array=new int[nbObjects][weightmax+1];

// initialize the first row
for(j=0;j<=weightmax;j++)
  if (j<weight[0]) array[0][j]=0;
     else array[0][j]=utility[0];
// for all other rows
for(i=1;i<nbObjects;i++)
  {
  for(j=0;j<=weightmax;j++)
    if (j-weight[i]<0) array[i][j]=array[i-1][j];
     else
      array[i][j]=max( array[i-1][j],
              array[i-1][j-weight[i]]+utility[i]);
  }
}
```

The result of the optimization reads at the bottommost right cell of this array: $u(n,W)$. To illustrate this algorithm, consider the following input:

```
static int nbObjects=8;
static int [] weight={2,3,5,2,4,6,3,1};
static int [] utility={5,8,14,6,13,17,10,4};
static int weightmax=12;
static int [] [] array;
```

Then procedure SolveDP builds and gets the table of $u(i,j)$ evaluations (see Table 9.1).

We now retrieve the solution from the table by beginning from the bottom right cell: here, a maximum utility score of 38 for $i = n = 8$. If $u(i,j)$ has the same score as the precedent row $u(i-1,j)$ then we deduce that object O_i was not chosen. Otherwise, that means that we chose object O_i. Starting from the

Object Weight $u(i,j)$	0	1	2	3	4	5	6	7	8	9	10	11	12
1	0	0	5	5	5	5	5	5	5	5	5	5	5
2	0	0	5	8	8	13	13	13	13	13	13	13	13
3	0	0	5	8	8	14	14	19	22	22	27	27	27
4	0	0	6	8	11	14	14	20	22	25	28	28	33
5	0	0	6	8	13	14	19	21	24	27	28	33	35
6	0	0	6	8	13	14	19	21	24	27	30	33	36
7	0	0	6	10	13	16	19	23	24	29	31	34	37
8	0	4	6	10	14	17	20	23	27	29	33	35	38

Table 9.1 Table of $u(i,j)$ evaluations

last row, we thus deduce that we chose O_8 because $37 \neq 38$; We now *jump* to $u(i-1, j-W_8)$ and proceed similarly until we reach the first row.
The code for retrieving the solution from the 2D table by reading backward is thus:

Program 9.15 Extracting backward the optimal solution from the 2D table

```
static void InterpretArray()
{
int i,u,w;
u=0;
w=weightmax;

for(i=nbObjects-1;i>=1;i--)
{
  if (array[i][w]!=array[i-1][w])
      {System.out.print((i+1)+" ");
      w=w-weight[i];
      u=u+utility[i];
      }
}

if (array[0][w]!=0);
  {System.out.println("1");
  w=w-weight[0];
  u=u+utility[0];
  }

System.out.println("Cross check:"+u+" remaining weight "+w);
}
```

Running this procedure on our toy example, we get:

```
Reading solution:
8 7 5 4 1
```

```
Cross check:38 remaining weight 0
```

We interpret the procedure extracting the result by manually running procedure InterpretArray() in Table 9.2.

9.5 Optimization paradigms: Overview of complexity analysis

In this chapter, we have presented four basic combinatorial optimization problems:

1. plain knapsack filling problem,
2. eight queen puzzle,
3. set cover problem,
4. maximal utility knapsack problem.

For solving these tasks, we have considered heuristics for finding either an *exact* or an *approximation* of the optimal solution. The properties of these generic algorithms are quickly summarized below:

Exhaustive search. Explore the full configuration space using a simple recursive procedure, which usually requires exponential time for enumerating all potential solutions.

Backtracking. Structure the space of a potential solution and *incrementally* build the solution until we reach a dead end. Then backtrack by changing the configuration of previous states. Complexity does not asymptotically change; it is still exponential in the worst case.

Greedy algorithm. Construct the solution iteratively by choosing at each step the current best set. Greedy heuristics are *polynomial* in the input size. The greedy SCP algorithm takes cubic time.

Dynamic programing. Build a table from a recurrence relation and extract the solution by reading the table backward. The complexity is polynomial (that is, requires $O(Wn)$ time and space) if we consider W as a parameter. Dynamic programming faces limitations when W is too large. We then prefer to solve it using the constant 2-approximation greedy algorithm.

Finally, let us stress that even if the greedy heuristic seems a bit naive, computer scientists do not have better algorithms on hand. Finding a better algorithm is actually one of the hottest, if not the most challenging, open problem in

$u(i,j)$	0	1	2	3	4	5	6	7	8	9	10	11	12	Interpretation
1	0	0	5	5	5	5	5	5	5	5	5	5	5	O_1 $(u = 38, w = 0)$
2	0	0	5	8	8	13	13	13	13	13	13	13	13	$5 = 5 \Rightarrow \neg O_2$
3	0	0	5	8	8	14	14	19	22	22	27	27	27	$5 = 5 \Rightarrow \neg O_3$
4	0	0	6	8	11	14	14	20	22	25	28	28	33	$11 \neq 8 \Rightarrow O_4$ $(u = 33, w = 2)$
5	0	0	6	8	13	14	19	21	24	27	28	33	35	$24 \neq 22 \Rightarrow O_5$ $(u = 27, w = 4)$
6	0	0	6	8	13	14	19	21	24	27	30	33	36	$24 = 24 \Rightarrow \neg O_6$
7	0	0	6	10	13	16	19	23	24	29	31	34	37	$34 \neq 33 \Rightarrow O_7$ $(u = 14, w = 8)$
8	0	4	6	10	14	17	20	23	27	29	33	35	38	$38 \neq 37 \Rightarrow O_8$ $(u = 4, w = 11)$
Read back the table, starting at position $u(8, 12)$ (u=0, w=12)														

Table 9.2 Extracting the solution from the dynamic programming table. O_i and $\neg O_i$ meaning that we selected/did not select object O_i, respectively

computer science related to the $P \neq NP$? conjecture. Solving this conjecture is one of the seven key challenges proposed by the Clay Mathematics Institute[4] in its Millenium problem collection, with a rewarding of one million dollars.

9.6 Exercices

Exercise 9.1 (Binomial coefficients and Pascal triangles)

The *choose function*

$$C_{n,k} = \left(\begin{array}{c} n \\ k \end{array} \right)$$

is the number of different ways one can choose k objects among a set of n objects. We have

$$\left(\begin{array}{c} n \\ k \end{array} \right) = \frac{n!}{(n-k)!k!}$$

for $0 \leq k \leq n$, where $x!$ denotes the factorial of integer x. Binomial coefficients arise naturally in the polynomial expansion of

$$(x+1)^n = \sum_{k=0}^{n} \left(\begin{array}{c} n \\ k \end{array} \right) x^k.$$

Scholar Blaise Pascal (1623-1662) used the following recurrence relation:

- $C_{n,k} = C_{n-1,k-1} + C_{n-1,k}$, and
- $C_{n,n} = C_{n,0} = 1$ (terminal states)

to compute coefficients $C_{n,k}$ in a triangle shape depicted as below for $n = 5$. Rows are indexed by n, and columns by $k \leq n$.

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
```

- Give a plain recursive function for computing the $C_{n,k}$ coefficients, and use that procedure to compute the Pascal triangle.
- Try running the program with $n = 35$. What happens? Please explain.

[4] `http://www.claymath.org/millennium/P_vs_NP/`

– Write a function **public static int**[] createNextLine(**int**[] line) that takes as a parameter a reference to an array of integers storing the values of $C_{n-1,*}$ and returns a reference to a new array of integers containing the values $C_{n,*}$. Write a program for displaying the Pascal triangle of size 35 with this method. What happens? Please explain and compare with the fully recursive program. Deduce a simple program to display Pascal triangles of any order.

Exercise 9.2 (Fibonacci sequences)

Fibonacci sequences are defined by the recurrence relation $F(n) = F(n-1) + F(n-2)$ for $n \geq 2$ and $F(0) = 0$, $F(1) = 1$, otherwise.

– Give a recursive function for computing arbitrary $F(n)$.

– Using the memoization technique, create a static array **static long**[] memo;, and procedures for filling the array and computing (returning) a member of the sequence. What happens as n grows?

There exits a fast exponentiation method to compute Fibonacci numbers. Indeed, prove that

$$\begin{pmatrix} F(n+1) & F(n) \\ F(n) & F(n-1) \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n.$$

It follows that

$$\begin{pmatrix} F(n+1) & F(n) \\ F(n) & F(n-1) \end{pmatrix}^2 = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{2n} = \begin{pmatrix} F(2n+1) & F(2n) \\ F(2n) & F(2n-1) \end{pmatrix}.$$

Deduce that $F(2n) = F(n) \times (F(n) + 2F(n-1))$ and $F(2n-1) = F(n)^2 + F(n-1)^2$. Thus if n is even, then $F(n)$ and $F(n-1)$ can be obtained directly from $F(n/2)$ and $F(n/2-1)$. Similarly, if n is odd, then we get $F(n-1)$ and $F(n-2)$, and $F(n)$ is computed as the sum of $F(n-1) + F(n-2)$. Implement this algorithm using BigInteger number instead of **int** to circumvent numerical problems.

Exercise 9.3 (Hitting set problem)

We are given a range space $(\mathcal{X}, \mathcal{R})$ of $|\mathcal{X}| = n$ elements $X_1, ..., X_n$ and $|\mathcal{R}| = m$ subsets $R_1, ..., R_m$. We are asked to find a hitting set, that is a subset $\mathcal{X}' \subseteq \mathcal{X}$ of elements so that each range R_i contains at least one element of $\mathcal{X}'$.

– Design a greedy algorithm to find a not too large hitting set.

– Show that hitting set problems are dual to set cover problems by reversing the inclusion order. Show that the duality consists merely in transposing the incidence matrix.

Exercise 9.4 (Discriminating set problem)

Given a range space $(\mathcal{X}, \mathcal{R})$ of $|\mathcal{X}| = n$ elements $X_1, ..., X_n$ and $|\mathcal{R}| = m$ subsets $R_1, ..., R_m$, we would like to find a discriminating set $\mathcal{X}' \subseteq X$ such that $\forall i, j, i \neq jR_i \cap \mathcal{X}' \neq R_j \cap \mathcal{X}'$.

– Prove that finding a minimum size discriminating set $\mathcal{X}'$ amounts to solving a hitting set problem for the set of pairwise symmetric differences $R_i \triangle R_j = R_i \backslash R_j \cup R_j \backslash R_i$. Design a greedy algorithm for finding a not too large discriminating set. Deduce that memory requirement is quadratic.

– Instead of explicitly computing the full set of symmetric differences of ranges, design an algorithm that solves the hitting set problem using only linear memory (but more time).

10
The Science of Computing

10.1 The digital world

During the last three decades (1978–2008), we have witnessed worldwide an unprecedented revolution: the venue of the *digital world*. Nowadays there is no doubt that digital music players or digital cameras have replaced their analog counterparts forever (or at least until the next paradigm shift). But what are the benefits of this analog-to-digital wave since the industry is still working hard to reach the recording/rendering qualities of some prior analog devices.

The first merit of the digital world is to dissociate contents from its support. Once digitized contents are all binary strings of zeros and ones, this "binarization" process harmonizes all former kinds of contents (books, musics, pictures an videos) into a *universal representation*: strings of bits.

The second merit of this digital revolution is that there is a universal player: the "computer." Indeed, although there are a lot of dedicated devices in the consumer electronic industry optimized for reading such or such a kind of digital content, a "computer" can read/interpret and render appropriately all these digitized contents. That is, the computer became the *universal device*. Figure 10.1 illustrates this wave of universal content/universal players.

This raises the unsolved question of defining what information is. A key advantage of binarizing all contents is that we have at our disposal *generic algorithms* for handling these binary strings that can be used whatever the type of contents. Namely, we can indifferently:

F. Nielsen, *A Concise and Practical Introduction to Programming Algorithms in Java*, Undergraduate Topics in Computer Science, DOI 10.1007/978-1-84882-339-6_10,

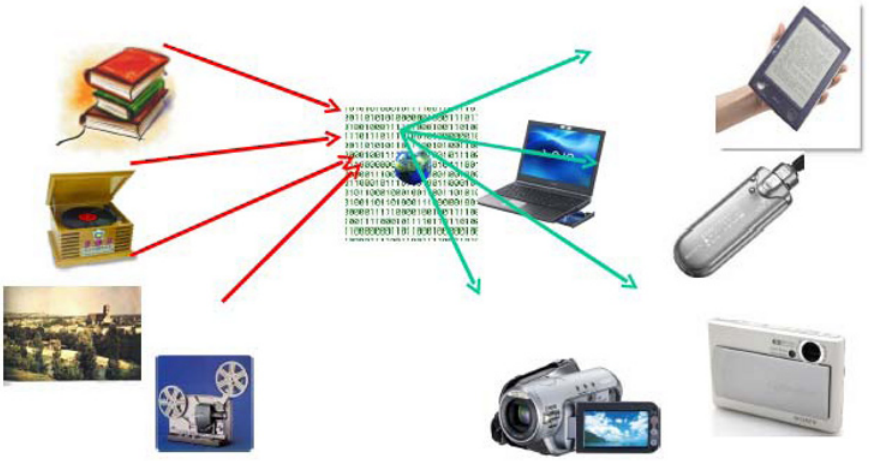


Figure 10.1 In the digital world, all kinds of content is represented using universal binary strings. Furthermore, all this content can be appropriately rendered using a *universal computer*. Nevertheless, for the consumer, industry proposes many tailored devices

– copy,

– compress,

– transmit,

– archive,

– ...

binary strings without knowing, say, that it is a music binary string or a book binary string. Figure 10.2 illustrates this powerful property of generic algorithms that act on all kinds of digitalized contents.

One of the frequent questions to ask is why humans calculate using decimal numbers using digits in $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ but computers only use bits, digits in $\{0, 1\}$. The answer is whatever base you choose, provided that it has at least two digits, you will get the same order of power to express *compactly* numbers. Indeed, unary numeral systems require a linear number of signs to express a number, while binary numeral systems only use $\log n$ of them. In other words, the binary numeral system is exponentially more powerful at compacting numbers than the unary numeral system.

In summary the digital world brought us these four advantages over the analog world:

– Universal content representation: binary strings.

– Universal rendering device: the computer.

– Generic (universal) algorithms.

– Universal binary base representation.

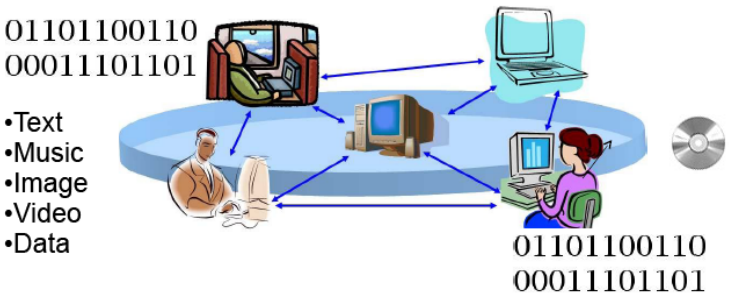


Figure 10.2 Once the content is available as a binary string we can apply generic algorithms such as copying, compressing, transmitting or archiving it without having to know, say, that it is a music or book string

10.2 Nature of computing?

Defining computing, finding its limitations and pondering whether nature is doing computing of some sort are puzzling questions. In 1953, James Watson and Francis Crick discovered the double-helix structure of DNA that encodes genes. They received the Nobel Prize for this spectacular discovery. The mechanism of DNA replication is well understood nowadays, and it is striking to see that in genetics, DNA sequences are copied, transmitted and archived using the same process carried out by the ribosome units. Historically speaking, it was first envisioned by Erwin Schroedinger[1] in his remarkable essay "What is life?" [5] published in 1944 that there should exist some "crystal" for transmitting information from cells to cells. This "crystal" revealed itself a decade later as the DNA.

[1] Another winner of the Nobel prize in 1933.

10.3 The digital equation

Mark Weiser envisioned the era of *ubiquitous computing*, where computing devices are omnipresent. We are experiencing more and more this life style with the Internet and current cell phones. How is it that even if content was binarized that computing became ubiquitous?
This is because of the digital equation:

$$\boxed{\text{Digital} = \text{Binary} + \text{Calculation}}$$

Indeed, consider pictures to illustrate this equation. At first, pictures were analog. We then digitized them into binary strings using various devices like photo scanners. But nowadays, we are yet a stage beyond: We have entered the era of *computational photography*. That is, photos are computed. You can buy digital cameras that implement *smile shutter* or other functionalities that could not exist in the analog era. The digital equation opened up a completely unknown world. There is a novel momentum of the media: What is a (digital) photo and how can we (computationally) capture/manipulate and share them? Computing not only allows us to experience novel experiences with former media, it also allows us to create brand new media that were quite unthinkable in the analog world. For example, within a couple a decades our flat TV will certainly be for sure replaced by a networked 3D holographic display. This will be possible not only because of hardware progress but also because a brand new type of media will be captured and processed: *geometry*. To simplify, we may say that the first media era was sound (1970's). Then came images (1980's) and videos (1990's). The 21st century (2000) with the rise of *geometry processing* will be about geometries (improperly called 3D videos). And who knows, this digital equation could also be one of the hidden laws of Nature.

10.4 Birth of algorithms and computers

The birth of algorithms is allegedly credited to Al-Khwarizmi (790-840), a Persian scholar working in the scientifically flourishing city of Bagdad. Al-Khwarizmi wrote important scientific treaties and is considered one of the fathers of algebra (Al jabr). The word "algorithm" stems from the latinization of "Algorithmi." Indeed, Al-Khwarizmi was concerned with providing his readers with generic pipelines to solve for solutions of quadratic equations.
A common misunderstanding is to associate computer science necessarily with computers. Computer science *is not* about programming personal computers. The difference engine, conceived by Charles Babbage in 1822, is one of the very

first computers. It was only physically built much later on. We refer to the textbook of Cormen et al. [1] for a nice introductory overview of algorithms. We emphasize that:

Computers = Computing Machineries

10.5 Computer science in the 21st century

Nowadays, computers (and computing) are omnipresent. The envisioned ubiquitous computing of Mark Weiser (Xerox Parc chief scientist) is here. Computers are abundant and versatile. They are abundant because we probably carry dozens of CPUs without being conscious of them (cell phones, cameras, music players, etc.). Furthermore, computing is impacting all traditional sciences. It is now common to speak of computational sciences. For example, biology is evolving as systems biology which allows researchers to perform a rich set of simulation-prediction-experiences (and finally cross-validate these numerical simulations in wet labs.

The science of computing is Computer Science (CS): Computer science raises deep theoretical questions. We recommend the Harel's book [3]: "Computers Ltd.: What They Really Can't Do." Computer science also brings important novel technologies. For example, in medical imaging a novel imaging technique is called diffusion tensor magnetic resonance imagery, or DT-MRI. Computing will definitively participate in the *integration science* of the 21st century.

11
Exam & Solution

The 2-hour exam below gives an overall review of the main concepts of programming in Java. The four independent exercises are given with their corresponding solutions.

Solution 11.1 (Mysterious recursive function)

– Consider the following program which compiles without any error:

```
public class MysteriousProgram {
  public static void display(int[] tab) {
    for (int i = 0; i < tab.length; i++)
      System.out.print(tab[i] + " ");
    System.out.println();
  }
  public static void swap2(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
  }
  public static void swap3(int[] tab, int i, int j) {
    int tmp = tab[i];
    tab[i] = tab[j];
    tab[j] = tmp;
  }

  public static void mysterious(int[] tab, int k) {
    for (int j = k; j < tab.length; j++) {
      swap3(tab, k, j);
      display(tab);
      swap3(tab, k, j);
    }
  }
```

F. Nielsen, *A Concise and Practical Introduction to Programming Algorithms in Java*, Undergraduate Topics in Computer Science, DOI 10.1007/978-1-84882-339-6_11,

```
  public static void mysteriousRecursive(int[] tab,
      int k) {
    if (k == tab.length - 1)
      display(tab);
    for (int j = k; j < tab.length; j++) {
      swap3(tab, k, j);
      mysteriousRecursive(tab, k + 1);
      swap3(tab, k, j);
    }
  }

  public static void init(int[] tab) {
    for (int i = 0; i < tab.length; i++)
      tab[i] = i + 1;
  }

  public static void main(String[] args) {
    int n = Integer.parseInt(args[0]);
    int[] t = new int[n];
    init(t);
    swap2(t[0], t[n - 1]);
    mysterious(t, 0);
    //  mysteriousRecursive(t, 0);
  }

}
```

Once this code compiled, what is the result displayed in the console output by invoking java MysteriousProgram 4? Describe explicitly the program execution steps and its outcome for the general case (for any given n $>$ 0)

SOLUTION:

For $n = 4$, the program displays the following output:

```
1 2 3 4
2 1 3 4
3 2 1 4
4 2 3 1
```

Function `init` fills array `t` that is given by reference with values ranging from 1 to n. Function `swap2` performs a permutation on the *copied values* of two array cells, and thus does not modify the original array cells since it works with copies. The array contents order is therefore preserved. In the `mysterious` function, the value of k does not change and denote a cell array. Here, we use function `swap3` that allows one to exchange the contents of two array cells. At the first loop round, we exchange an array cell with itself. Then, we successively exchange the contents of the cell with the contents of the following

cells, we display the array, and perform again a permutation to leave the cells as before. Here, $k = 0$ and all values are successively put in first position. Notice that when the `mysterious` function ends, the array order is thus reset to the original order.

– In the function main, we now replace the function call mysterious(t, 0); by mysteriousRecursive(t, 0);.
After having recompiled the program, what is the console result obtained by launching java MysteriousProgram 3?

Describe precisely the program execution steps and the obtained result in general case (any given n > 0)

SOLUTION:

For $n = 3$, calling `mysteriousRecursive(t, 0);` yields the following output:

```
1 2 3
1 3 2
2 1 3
2 3 1
3 2 1
3 1 2
```

The program displays all permutations of the array. Remark: The recursion always terminates since the body of the `for` loop is executed only for `k > tab.length-1`. Proof goes by induction.

Solution 11.2 (Modeling molecules)

In this exercise, we are first concerned with modeling molecules as arrays of atoms, where each atom is defined as a proper 3D sphere with a center and a radius (the Van der Waals radius). We will then study how to detect whether atoms and molecules collide or not.

– Design a class Point3D where each object is defined as a 3D point with coordinates x, y and z, all of type **double**. Further, provide this class with a constructor Point3D(**double** x0, **double** y0, **double** z0) that allows us to initialize a Point3D object.

– Write a static function **double** distance(Point3D p, Point3D q) that takes as arguments two points p and q, and returns the Euclidean distance $\|q - p\|$ between them. In order to compute the square function, we will use **static double** sqr(**double** x){ **return** x∗x; }, which is also inserted inside the class Point3D.

To compute the square root, we'll use function **static double** sqrt(**double** x) of the Math class.

– Give a static function Point3D add(Point3D p, Point3D q) that takes as arguments two points p and q, and return a **new** point equal to $p+q$. This function shall be inserted inside class Point3D.

– Give a static function **void** scale(Point3D p, **double** k) that multiplies the coordinates of point p by scalar number k. That is, p becomes $k.p$. This function shall be located inside class Point3D as well.

SOLUTION:

Program 11.1 The class `Point3D`

```
public class Point3D {
double x, y, z;
public Point3D ( double x0 , double y0 , double z0) {
this .x = x0;
this .y = y0;
this .z = z0;
}
static double sqr ( double x) { return x*x; }
public static double distance ( Point3D p, Point3D q)
    {
return Math . sqrt (sqr (q.x - p.x) + sqr (q.y - p.y)
+ sqr(q.z - p.z));
}
public static Point3D add ( Point3D p, Point3D q) {
return new Point3D (p.x + q.x, p.y + q.y, p.z + q.z);
}
public static void scale ( Point3D p, double k) {
p.x *= k;
p.y *= k;
p.z *= k;
}
}
```

– Give a class Atom that allows us to define atoms with **two** object fields:

  – center of type Point3D denoting the location (x, y, z) of the center of this atom, and

  – radius of type **double** that encodes the radius of this atom.

  Provide a constructor Atom(**double** x, **double** y, **double** z, **double** rad) to this class that initialize objects of this type.

  Further, add to this class two constants H_RADIUS = 1.2 and O_RADIUS = 1.5 that represent the radii in ångström for the hydrogen and oxygen atoms, respectively.

– Write a static function **boolean** bump(Atom a, Atom b) that takes as arguments two atoms a and b, and return **true** if and only if the distance between their centers is strictly less than the sum of their radii (this will represent a collision between two atoms). This function shall be defined inside the class Atom.

SOLUTION:

Program 11.2 Class `Atom` with the `bump` predicate

```
public class Atom {
Point3D center ;
double radius ;
public static final double H_RADIUS = 1.2;
public static final double O_RADIUS = 1.5;
public Atom ( double x, double y, double z, double rad
    ) {
this . center = new Point3D (x, y, z);
this . radius = rad ;
}
public static boolean bump ( Atom a, Atom b) {
return Point3D . distance (a.center , b. center )
< a. radius + b. radius ;
}
} // Temporary class; Shall be enhanced next
```

– Write a class Molecule that allows us to define a 3D molecule as an array of atoms, and provide the class with a constructor that takes as argument a reference to this array.

SOLUTION:

Program 11.3 Class `Molecule`

```
public class Molecule {
Atom [] atoms ;
public Molecule ( Atom [] t) {
this . atoms = t;
}
}
```

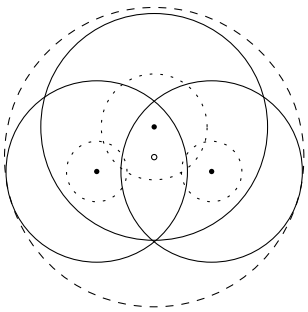
– Using a new class Test, write a program that builds a water molecule H20 with its oxygen atom located at $(0, 0.4, 0)$ and the two hydrogen atoms located at $(0.76, -0.19, 0)$ and $(-0.76, -0.19, 0)$ (units still being ångström).

SOLUTION:

Program 11.4 Test program for `Molecule`

```
public class Test {
public static void main ( String [] args ) {
Atom o = new Atom (0, 0.4 , 0, Atom . O_RADIUS );
Atom h1 = new Atom (0.76 , -0.19 , 0, Atom . H_RADIUS
    );
Atom h2 = new Atom ( -0.76 , -0.19 , 0, Atom .
    H_RADIUS );
Atom [] H2O = { o, h1 , h2 };
Molecule mol = new Molecule (H2O );
}
}
```

– For each molecule, we are now going to build an enclosing ball that will allow one to speed up the test for detecting potential collisions. To simplify, assume the center of that sphere is set as the centroid of atoms (barycenter of uniform weight). That is, we do not take into account respective masses. See figure as below. We shall use object Atom to represent such an enclosing ball.



• are atom centers. ∘ denote the centroid (barycenter with uniform weight).
Plain circles denote Van der Walls of respective atoms.

The enclosing sphere of a water molecule.

– Write a static function middle that takes as its argument an array of atoms (assume non-void) and that returns the centroid of these atoms. This function shall be inserted in the Atom class, and will use functions add and scale of class Point3D. This function shall not modify the atom coordinates of the array.

SOLUTION:

Program 11.5 The centroid of an `Atom` object

```
public static Point3D middle ( Atom [] t) {
Point3D middle = t [0]. center ;
for (int i = 1; i < t. length ; ++i)
middle = Point3D .add (middle , t[i]. center );
Point3D . scale (middle , 1.0/ t. length );
return middle ;
}
```

– Write a static function **double** maxDistance(Point3D p, Atom a) that returns the maximal distance between point p and any point on the sphere of atom a. This function shall be attached inside class Atom.

SOLUTION:

```
public static double maxDistance ( Point3D p, Atom a)
    {
return Point3D . distance (p, a. center ) + a. radius
    ;
}
```

– Describe now how to modify class Molecule for building its enclosing sphere and using it for checking for collisions. Note that if we do not intersect the enclosing ball of a molecule, it is not necessary to check for collisions of its atoms.

Write a static function **boolean** bump(Atom a, Molecule b) of class Molecule that allows us to check whether atom a is colliding with at least one of the atoms of molecule b or not.

Write a static function **boolean** bump(Molecule a, Molecule b) that extends this test to two molecules.

SOLUTION:

Program 11.6 The `Molecule` class equipped with the `bump` predicate

```
public class Molecule {
Atom [] atoms ;
Atom sphere ;
public Molecule ( Atom [] t) {
this . atoms = t;
Point3D center = Atom . middle ( atoms );
double r = 0;
for (int i = 0; i < atoms . length ; ++i) {
double ri = Atom . maxDistance (center , atoms [i]);
if (r < ri) r = ri;
```

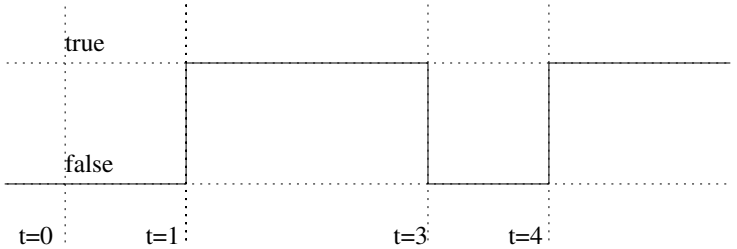
```
}

this . sphere = new Atom ( center .x, center .y,
    center .z, r);
}
public static boolean bump ( Atom a, Molecule b) {
if (! Atom . bump (a, b. sphere ))
return false ;
for (int i = 0; i < b. atoms . length ; ++i)
if ( Atom . bump (a, b. atoms [i]))
return true ;
return false ;
}
public static boolean bump ( Molecule a, Molecule b) {
if (! Atom . bump (a. sphere , b. sphere ))
return false ;
for (int i = 0; i < a. atoms . length ; ++i)
if ( bump (a. atoms [i], b))
return true ;
return false ;
}
}
```

Solution 11.3 (Coding electrical signals)

We are interested in modeling binary electrical signals with values ranging in {false, true} that only change a finite number of times. These changes occur only at clock ticks. Thus these events can be modeled as positive integers that encode the numbers of clock ticks since the starting time (time origin).

Such a signal is defined by its initial value v_0, and a sequence of strictly increasing transition states $\tau_1, \ldots, \tau_n \in \mathbb{N}$. In $]-\infty, \tau_1[$, the signal has value v_0. In $[\tau_1, \tau_2[$, the signal takes value $v_1 = \neg v_0$ (where $\neg x$ denotes the logical negate function of x, written as !x in Java). In $[\tau_2, \tau_3[$, the signal value is $v_2 = \neg v_1$, and so on (with $v_n = \neg v_{n-1}$ in time range $[\tau_n, +\infty[$). A constant signal shall be represented by its value v_0 and an empty sequence ($n = 0$).

For example, the depicted signal corresponds to the initial value false and the following sequence of transitions: $\tau_1 = 1, \tau_2 = 3, \tau_3 = 4$.



In the remainder, we shall use the following data structure:

```
class Transition {
  int time;
  Transition next;

  Transition(int time, Transition next) {
    this.time = time;
    this.next = next;
  }
}

class Signal {
  boolean initialValue;
  Transition transitions;

  Signal(boolean value, Transition transitions) {
    this.initialValue = value;
    this.transitions = transitions;
  }
}
```

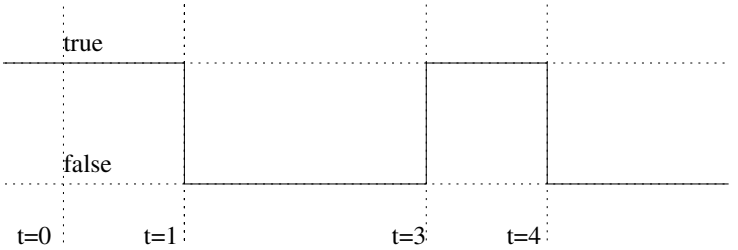
Field initialValue denotes the object value of type Signal at about $-\infty$. Field transitions denotes the beginning of the linked list of transition events for this signal. If that field is set to **null**, this means that the signal is constant. Otherwise, fields time of successive elements of the linked list indicate the respective values $\tau_1, \ldots, \tau_n$, with τ_1 denoting the head. We insist on the fact that the sequence τ_i is strictly increasing.

For example, we can create the signal depicted above as:

```
Signal signal1 = new Signal(false,
  new Transition(1, new Transition(3, new Transition(4,
      null))));
```

In the following questions, the functions shall be all located inside the body of a class, whose name can be arbitrarily chosen.

– Given a signal s, we first wish to invert it: that is, to obtain its logical negation. For example, the invert of signal1 is:



Write a function **static** Signal invert(Signal s) performing this process. Your function shall not modify the original signal. That is, you need to create a **new** Signal. The time complexity of this function should not depend on the length of the transition states in s.

– Given a signal s and time tick t, we want to compute $s(t)$ that is the value (true or false) of the signal s at time t. Write an **iterative** function **static boolean** valueAt(Signal s, **int** time) performing this task. You shall not modify signal s, and the time complexity should be proportional to the number of transition states in s.

– We would like a function that displays a signal as a sequence of intervals with respective values. For example,

```
-inf -> +inf : false
```

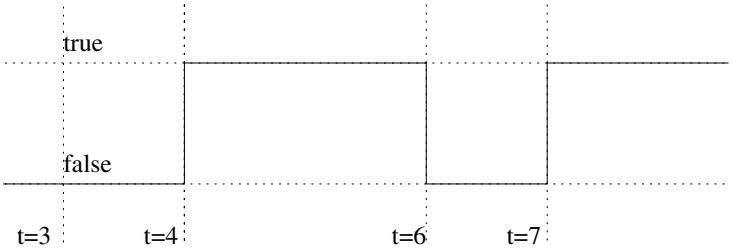
means that the signal uniformly takes value false. For signal1, the display function shall produce the output:

```
-inf -> 1.0 : false
1.0 -> 3.0 : true
3.0 -> 4.0 : false
4.0 -> +inf : true
```

Write such an **iterative** function **static void** print(Signal s).

– Given a signal s, we would like to produce another signal identical to s but shifted in time by a step $\delta > 0$. That is, we want a signal with the same initial value but with transition states (τ'_i) defined by $\tau'_i = \tau_i + \delta$, $\forall i \in \{1, \ldots, n\}$.

To illustrate this operation, the shifted signal of `signal1` with $\delta = 3$ produces the following result:



– Write a **recursive** function
static Transition shift(Transition t, **int** delta) that shall return a **new** list of transition states, identical to t but with values shifted by delta. You shall not modify the original list t.

  Then give a function **static** Signal shift(Signal s, **int** delta).

– A Signal object that does not have a strictly increasing sequence of transition states is not correct. In order to detect program errors, it is useful to have a function **static boolean** isWellFormed(Signal s) that returns **true** if and only if s is correct.

  Write a **recursive** function using an auxiliary function. Time complexity shall be linear to the list length of transition events stored in s.

– The commutable exclusive or operation (XOR for short) that takes two operands is defined by the following logic table:

XOR	false	true
false	false	true
true	true	false

  That is, b_1 XOR b_2 is true if and only if $b_1 = \neg b_2$. In Java, it can be written as b1 != b2.

  We now want to compute the output signal of the XOR of two input signals. We notice that at a given time step, if only one signal changes then the result changes, and if both signals change at a same time, then the result does not change.

  Write a function **static** Transition xorTransitions(Transition t1, Transition t2) that returns a **new** transition state list that corresponds to the output. This function shall not modify the source list t1 and t2, but in some cases, it is possible to share a sub-list. The time complexity shall be linear to the length of both lists.

Write now a function **static** Signal xorSignals(Signal s1, Signal s2).

SOLUTION:

Program 11.7 The various functions acting on `Signal` objects

```
public class WorkingSignals {

  public static Signal invert(Signal s) {
    return new Signal(!s.initialValue, s.transitions);
  }

  public static boolean valueAt(Signal s, int time) {
    boolean v = s.initialValue;
    Transition t = s.transitions;
    while (t != null) {
      if (time < t.time)
        return v;
      v = !v;
      t = t.next;
    }
    return v;
  }

  public static void print(Signal s) {
    boolean v = s.initialValue;
    Transition t = s.transitions;
    System.out.print("-inf -> ");
    while (t != null) {
      System.out.println(t.time + " : " + v);
      System.out.print(t.time + " -> ");
      v = !v;
      t = t.next;
    }
    System.out.println("+inf : " + v);
  }

  public static Transition shift(Transition t,
      int delta) {
    if (t == null)
      return null;
    else
      return new Transition(t.time + delta,
          shift(t.next, delta));
  }

  public static Signal shift(Signal s, int delta) {
    return new Signal(s.initialValue,
        shift(s.transitions, delta));
  }

  public static boolean isWellFormed(Transition t) {
    if (t == null || t.next == null)
      return true;
```

```
    else
      return (t.time < t.next.time)
        && isWellFormed(t.next);
  }

  static boolean isWellFormed(Signal s) {
    return isWellFormed(s.transitions);
  }

  public static Transition xorTransitions(Transition t1,
                                          Transition t2)
                                              {
    if (t1 == null)
      return t2;
    else if (t2 == null)
      return t1;
    else {
      int tt1 = t1.time;
      int tt2 = t2.time;
      if (tt1 < tt2)
        return new Transition(tt1,
            xorTransitions(t1.next, t2));
      else if (tt2 < tt1)
        return new Transition(tt2,
            xorTransitions(t1, t2.next));
      else
        return xorTransitions(t1.next, t2.next);
    }
  }

  public static Signal xorSignals(Signal s1, Signal s2)
      {
    return new Signal(
        (s1.initialValue) != (s2.initialValue),
        xorTransitions(s1.transitions, s2.transitions));
  }

  public static boolean recValueAt(boolean value,
      Transition t,
      int time) {
    if (t == null)
      return value;
    else if (time < t.time)
      return value;
    else
      return recValueAt(!value, t.next, time);
  }

  public static boolean recValueAt(Signal s, int instant
      ) {
    return recValueAt(s.initialValue, s.transitions,
        instant);
  }
```

```
  public static Signal renverser(Signal s) {
    boolean v = s.initialValue;
    Transition t = s.transitions;
    Transition t2 = null;
    while (t != null) {
      t2 = new Transition(-t.time, t2);
      v = !v;
      t = t.next;
    }
    return new Signal(v, t2);
  }

  public static void testValues(Signal s) {
    int[] t = { -1, 0, 1, 2, 4, 6 };
    for (int i = 0; i < t.length; ++i)
      System.out.println(t[i] + " : " + recValueAt(s, t[
          i]) + "  "
          + valueAt(s, t[i]));
  }

  public static void test(Signal s) {
    print(s);
    testValues(s);
    System.out.println(isWellFormed(s));
    System.out.println();
  }

  public static void testAll(Signal s) {
    test(s);
    Signal is = invert(s);
    test(is);
    System.out.println("XOR");
    print(xorSignals(s, is));
    System.out.println();
    Signal it = shift(is, 1);
    test(it);
    System.out.println("XOR");
    print(xorSignals(it, is));
    System.out.println();
    Signal ir = renverser(it);
    test(ir);
  }

  public static void main(String[] args) {
    Signal s1 = new Signal(false, null);
    Signal s2 = new Signal(false, new Transition(10, new
        Transition(50, null)));
    Signal s3 = new Signal(true, new Transition(10, new
        Transition(15,
        new Transition(30, null))));
    Signal signal1 = new Signal(false,
        new Transition(1, new Transition(3, new
```

```
            Transition(4, null))));
    System.out.println("\ns1");
     testAll(s1);
     System.out.println("\ns2");
     testAll(s2);
     System.out.println("\ns3");
     testAll(s3);

     System.out.println("s2 ^ s3");
     print(xorSignals(s2, s3));

     System.out.println("\nsigna1");
     testAll(signal1);

  }

}
```

Solution 11.4 (Nim game)

The game of Nim is a two-player game. We consider m bins (indexed from 0 to $m-1$), with bin i containing x_i fruits. A game configuration is therefore represented by an array of integers.

The two players are playing in turn. A player move consists of taking as many as wished (but at least one) fruit(s) in a **same** bin. The winner is the player who takes that last remaining fruit (hence all bins become empty).

The outcome of this game is fully predictable: One of the two players (depending on the initial bin configuration and the first player) has a winning strategy whatever the second player does. We shall study here this strategy.

We begin by writing two conversion functions as follows: The first function converts a number written in base 2 to a number written in base 10. The second function is the reciprocal function: conversion from base 10 to 2.

– An integer n represented in base 2 shall be modeled using an integer array binaryRepresentation of length k storing the k bits. We assume that all values in this array are equal to 0 or 1, and binaryRepresentation[i] is the i-th bit of the decomposition of n in base 2:

$$\texttt{n} = \sum_{\texttt{i=0}}^{\texttt{i<k}} \texttt{binaryRepresentation[i]} * 2^{\texttt{i}}$$

Write a function
static int binaryToDecimal(**int**[] binaryRepresentation) that returns the integer corresponding to the binary representation of binaryRepresentation

– Here, we assume that k is big enough, and we are given the function:

```
public static int[] decimalToBinary(int n, int k) {
  int[] binaryRepresentation = new int[k];
  decimalToBinaryAux(n, 0, binaryRepresentation);
  return binaryRepresentation;
}
```

Write the **recursive** function decimalToBinaryAux

– In the following, given a number y and an integer k, chosen large enough so that y can be written in base 2 using k bits, we denote by $y[i]$ the i-th bit of the binary decomposition of y. That is, we have $y = \sum_{i=0}^{k} y[i] * 2^i$.

We consider the following function called Grundy that takes as its argument a number of fruits in each bin and returns an integer $Grundy(x_0, x_2, \ldots, x_{m-1}) = a$ denoting the binary representation defined by $a[i] = \left(\sum_{l=0}^{m-1} x_l[i] \right) \mod 2$.

For example, if we want to compute $Grundy(6, 9, 1, 2)$, we first write 6, 9, 1 and 2 using base 2 (using the same number of k bits), and we compute the sum on each column modulo 2. This yields a representation in base 2 of a. We then convert a in base 10.

6	=	0	1	1	0
9	=	1	0	0	1
1	=	0	0	0	1
2	=	0	0	1	0
a	=	1	1	0	0

Thus we have $Grundy(6, 9, 1, 2) = a = 12$.

– Write function **static int** Grundy(**int**[] decimalTab) that returns the value of the Grundy function for the game configuration stored in array decimalTab

It is recommended to write a few auxiliary functions, as follows:
- compute the required number k of bits,
- build an array **int** [][] storing the binary decompositions of values stored in decimalTab,
- compute the Grundy function in binary.

Those functions can be reused in the remainder.

– We shall make use of the following (admitted without proof) result:

Let a Nim game configuration be denoted by x_0, x_1, ..., x_{m-1} fruits in respective bins. Then the current player has a winning strategy if and only if $Grundy(x_0, \ldots, x_{m-1}) \neq 0$.

In case we have $Grundy(x_0, \ldots, x_{m-1}) \neq 0$, the winning move is described by the following steps:

– We consider $Grundy(x_0, \ldots, x_{m-1})$ in base 2, and we select the maximum index j of a bit with value 1 in the decomposition. Such a bit necessarily exits because of $Grundy(x_1, \ldots, x_{m-1}) \neq 0$.

– We search for an index i corresponding to a bin containing x_i fruits, where j denotes the bit index of the binary decomposition of x_i with corresponding bit value 1. Such an index i necessarily exists from the Grundy function. Fruits will be removed from the bin i.

– We shall remove from bin i a number of fruits such that the remaining number of fruits x'_i shall satisfy for all rank h:

$$x'_i[h] = \begin{cases} 1 - x_i[h] & \text{if } grundy[h] = 1 \\ x_i[h] & \text{otherwise.} \end{cases}$$

In the example given in the former question, the index j is 3 and the bin to select fruits from has index 1 (containing 9 fruits). We remove 4 fruits so that it remains exactly 5 fruits.

– Write a function pick that takes as its argument an integer array denoting a game configuration, and returns an integer array of size 2: the first integer shall report the index of the bin to select fruits from, and the second integer shall give the number of fruits to remove from the selected bin. In case the Grundy function is zero (we know we are going to loose), we shall remove a fruit from the first non-empty bin.

SOLUTION:

Program 11.8 Nim game solution

```
public class Nim {

  public static int binaryToDecimal(
                        int[] binaryRepresentation) {
    int n = 0;
    int p2 = 1;
    for (int i=0; i < binaryRepresentation.length; i++)
        {
      n = n + binaryRepresentation[i] * p2;
      p2 = p2 * 2;
    }
    return n;
  }

  public static void decimalToBinaryAux(int n, int i,
      int[] binaryRepresentation) {
    if (n > 0) {
      binaryRepresentation[i] = n % 2;
      decimalToBinaryAux(n/2, i+1, binaryRepresentation)
          ;
    }
  }

  public static int[] decimalToBinary(int n, int k) {
    int[] binaryRepresentation = new int[k];
    decimalToBinaryAux(n, 0, binaryRepresentation);
    return binaryRepresentation;
  }

  public static int getBinaryLength(int[] decimalTab) {
    // search the largest value
    int max = decimalTab[0];
    for (int i = 1; i < decimalTab.length; i++) {
      if (decimalTab[i] > max) {
        max = decimalTab[i];
      }
    }
    int k = 0;
    int p = 1;
    while (p <= max) {
      ++k;
      p *= 2;
    }
    return k;
  }

  public static int[][] decomposition(int[] decimalTab)
      {
    int k = getBinaryLength(decimalTab);
    int[][] binaryTab = new int[decimalTab.length][];
```

```
    for (int i = 0; i < decimalTab.length; i++) {
      binaryTab[i] = decimalToBinary(decimalTab[i], k);
    }
    return binaryTab;
  }

  public static int[] binaryGrundy(int[][] binaryTab) {
    int k = binaryTab[0].length;
    int[] grundyBinaire = new int[k];
    for (int j = 0; j < k; j++) {
      for (int i = 0; i < binaryTab.length; i++) {
        grundyBinaire[j] =
          (grundyBinaire[j] + binaryTab[i][j]) % 2;
      }
    }
    return grundyBinaire;
  }

  public static int Grundy(int[] decimalTab) {
    int[][] binaryTab = decomposition(decimalTab);
    return binaryToDecimal(binaryGrundy(binaryTab));
  }

  public static int[] loserPick(int[] decimalTab) {
    int i = 0;
    while (decimalTab[i] == 0) {
      i++;
    }
    int[] play = { i, 1 };
    return play;
  }

  public static int[] pick(int[] decimalTab) {
    int[][] binaryTab = decomposition(decimalTab);
    int k = binaryTab[0].length;
    int[] grundyBin = binaryGrundy(binaryTab);

    if (binaryToDecimal(grundyBin) == 0) {
      return loserPick(decimalTab);
    }
    int j = k - 1;
    while (grundyBin[j] == 0) {
      j--;
    }
    int i = 0;
    while (binaryTab[i][j] == 0) {
      i++;
    }
    for (int h = 0; h < grundyBin.length; h++) {
      if (grundyBin[h] == 1) {
        binaryTab[i][h] = 1 - binaryTab[i][h];
      }
    }
```

```
    int[] play = new int[2];
    play[0] = i;
    play[1] = decimalTab[i] -
        binaryToDecimal(binaryTab[i]);
    return play;
  }

  public static void main(String[] args) {
    int[] game = { 6, 9, 1, 2 };
    do {
      int tt = 0;
      System.out.println();
      for (int i = 0; i < game.length; ++i) {
        System.out.print(game[i] + " ");
        tt += game[i];
      }
      System.out.println();
      if ( tt <= 0 )
        break;
      System.out.println(Grundy(game));
      int[] play = pick(game);
      System.out.println(play[0] + " " + play[1]);
      game[play[0]] -= play[1];
    } while (true);
  }
}
```

Bibliography

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition.* McGraw-Hill Science/Engineering/Math, July 2001.

[2] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, 3rd Edition.* Addison-Wesley Professional, 2005.

[3] David Harel. *Computers Ltd.: What They Really Can't Do.* Oxford University Press, Inc., New York, NY, USA, 2000.

[4] Frank Nielsen. *Visual Computing: Geometry, Graphics, and Vision.* Charles River Media/Thomson Delmar Learning, 2005.

[5] Erwin Schroedinger. *What is Life - The Physical Aspect of the Living Cell.* Cambridge University Press., 1944.

Index

Numbers written in italic refer to the page where the corresponding entry is described; numbers underlined refer to the definition; numbers in roman refer to the pages where the entry is used.