

Object-Oriented Programming

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria

Wolfgang.Schreiner@risc.jku.at

May 13, 2025

Abstract

This second part of our lecture notes covers more advanced features of the programming language Java: the concept of *object methods*, the datatype *array*, the implementation of fundamental *algorithms and data structures*, the organization of classes by *inheritance*, the use of the *Java Collections Framework* for processing various types of object containers, and finally the handling of special situations by *exceptions*.

Contents

1	Extra-Strong Java	6
1.1	Objects with Methods	6
1.1.1	Object Methods	7
1.1.2	Abstracting Data	10
1.1.3	The <code>this</code> Pointer	12
1.2	Class Libraries and Packages	19
1.2.1	Writing Packages	20
1.2.2	The Package <code>java.lang</code>	21
1.3	Arrays	24
1.3.1	Array Declaration and Creation	25
1.3.2	Array Referencing	26
1.3.3	Array Processing	28
1.3.4	Arrays and Objects	31
1.3.5	Multi-Dimensional Arrays	37
1.3.6	The Class <code>Vector</code>	41
2	Java with Taste	43
2.1	Binary Search	43
2.2	Simple Sorting	46
2.3	Dynamic Data Structures	49
2.3.1	List Nodes	49
2.3.2	List Type	51

2.3.3	Inserting List Elements	52
2.3.4	Searching for List Elements	54
2.3.5	Deleting List Elements	55
2.3.6	Keeping a List Sorted	57
2.4	Abstract Data Types	59
2.4.1	Interfaces	59
2.4.2	Stacks	64
2.4.3	Queues	68
3	Java with Caffeine	76
3.1	Inheritance	76
3.2	Example	77
3.3	Superclasses and Subclasses	78
3.4	Class Hierarchies	79
3.5	Interface Hierarchies	81
3.6	Constructors in Derived Classes	81
3.7	Code Sharing	84
3.8	Type Compatibility	84
3.9	Object Assignments	85
3.10	Static Types and Dynamic Types	87
3.11	Generic Methods	87
3.12	The Object Class	88
3.13	Method Overriding	90
3.14	Calling Overridden Methods	91
3.15	Shadowing Variables	92
3.16	Final Methods and Classes	93
3.17	Abstract Classes	94
3.18	Frameworks	95
3.19	Visibility Modifiers	97
3.19.1	Class Visibility	97
3.19.2	Field and Method Visibility	98
3.19.3	Rules of Thumb	98

4	Java in Pots	101
4.1	Generics	101
4.2	The Collections Framework	103
4.3	Collections	104
4.3.1	Lists	106
4.3.2	Sets	107
4.3.3	Queues and Deques	109
4.4	Maps	111
4.5	Algorithms	113
5	Spilt Java	117
5.1	Error Codes	117
5.2	Exceptions	121
5.3	Example	122
5.4	Handling Exceptions	124
5.5	Catching Multiple Exception Types	125
5.6	The finally Clause	126
5.7	Exceptions Thrown by Handlers	127
5.8	Runtime Exceptions	128
5.9	When Not to Use Exceptions	130
5.10	File Input/Output	132
5.11	The Input Class	134

Chapter 1

Extra-Strong Java

In this chapter, we will continue the discussion on those concepts of Java that enable us to write larger programs that solve more complex problems. These constructs mainly deal with the structuring of code (*methods*) and of data (*classes*). Their combined application already provides a good deal of the expressiveness of the programming language. Furthermore, we will introduce *arrays* as one of the most important data structures of Java. Having studied this section, one should be able to develop large programs that consist of multiple classes and call various methods that process collections of values.

What will be not yet discussed in this chapter is how to separate the internal implementation of a class from the public specification (*information hiding*) and how new classes can be constructed by extending existing ones (*inheritance*). These issues will be presented in a later chapter; they will complete the *object-oriented* features of Java.

1.1 Objects with Methods

Up to now, we have considered two kinds of classes. The first one are collections of methods, variables, and constants declared as

```
public class Class
{
    public static type variable = expression;
    public static final type constant = expression;
    public static type method(parameters)
    {
        ...
    }
}
```

```
    }  
    ...  
}
```

Here the name *Class* can be used to call a *method* from other modules by the qualified name

```
Class.method(arguments)
```

Thus *method* is bound to *Class*; it can refer to globally declared variables and constants which are also bound to the class.

The second kind of classes are types of objects whose declarations are of the form

```
public class Class  
{  
    public type variable = expression;  
    public final type constant = expression;  
    public Class(parameters)  
    {  
        ...  
    }  
    ...  
}
```

Here the name *Class* can be used to create an object by the expression

```
new Class(arguments)
```

which invokes the corresponding constructor. This constructor is a special method that is not bound to a class but to an *object* of this class; it can refer to the fields (variables/constants) of the object.

The main difference between both kinds of declarations is the use of the keyword `static`. An entity declared with this keyword is bound to the class in which it is declared; if `static` is omitted, the entity belongs to an object of this class. By omitting the keyword for method declarations, we can also declare methods which are bound to objects.

1.1.1 Object Methods

We can declare a method which is not bound to a class but to an object of a class as

```
public class Class
{
    public type method(parameters)
    {
        ...
    }
    ...
}
```

Such an *object method* can be invoked by the qualified name

```
object.method(arguments)
```

The body of *method* may refer to the fields of (the object referenced by) *object* and update their contents.

Example Take the class

```
public class Date
{
    public int day;
    public String month;
    public int year;

    public Date(int day, String mont, int year)
    {
        this.day = day;
        this.month = month;
        this.year = year;
    }

    public int getYear()
    {
        return year;
    }

    public void setYear(int year)
    {
        this.year = year;
    }

    public void increaseYear()
```

```
{
    year++;
}
```

The code

```
Date date = new Date(24, "December", 2001);
System.out.println(date.getYear());
```

gives output

```
2001
```

However, we may increase the year of the date by executing

```
date.setYear(date.getYear()+1);
System.out.println(date.getYear());
```

which gives output

```
2002
```

Actually, we may achieve the same effect by executing

```
date.increaseYear();
System.out.println(date.getYear());
```

which gives output

```
2003
```

It may look a bit awkward to have methods for manipulating individual object fields rather than referring to an object field directly as in

```
date.year = date.year+1;
```

However, it is a very good convention *not* to refer to object fields in methods that are not declared in the class itself. The object thus becomes an *abstract datatype* whose private *implementation* (the object fields) is hidden behind a public *interface* (the object methods). Later, we will show how this convention can be actually *enforced* by the implementor of a class.

Do not refer to object fields from the outside.

1.1.2 Abstracting Data

We return to our problem of analyzing exam grades and replace the declaration of class *Result* by the following declaration:

```
public class Result
{
    public int count = 0;           // number of grades
    public int sum = 0;            // sum of grades processed
    public int min = Integer.MAX_VALUE; // minimum grade processed
    public int max = Integer.MIN_VALUE; // maximum grade processed

    public int getCount()
    {
        return count;
    }

    public void process(int grade)
    {
        count++;
        sum += grade;
        if (grade < min) min = grade;
        if (grade > max) max = grade;
    }

    public void print()
    {
        System.out.println("\nNumber of grades: " + count);

        // these values are only valid, if there was at least one grade
        if (count > 0)
        {
            System.out.println("Best grade: " + min);
            System.out.println("Worst grade: " + max);
            System.out.println("Average grade: " +
                               (float)sum/(float)count);
        }

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

How a grade is processed and how a result is printed is now hidden within class

Result, therefore the methods `processGrade` and `printResult` are not necessary any more.

The main method `analyzeExamGrades` now becomes

```
public static void analyzeExamGrades()
{
    char answer = 0;
    do
    {
        Result result = analyzeExam();
        result.print();
        answer = askToContinue();
    }
    while (answer == 'y');
}
```

where method `print` is invoked on object *result*.

The method `analyzeExam` becomes

```
public static Result analyzeExam()
{
    Result result = new Result();

    // read and process exam
    while (true)
    {
        int grade = readGrade(result.getCount());
        if (grade == 0) return result;
        result.process(grade);
    }
}
```

where the methods `getCount` and `process` are invoked on object *result*.

Our application now consists of two classes. The first class is the modified application `Exams3`; the second class is class `Result` which we put (because of its size) into a separate file `Result.java`:

```
// -----
// Result.java
// the (intermediate) result of analyzing the grades of an exam
//
```

```

// Author: Wolfgang Schreiner <Wolfgang.Schreiner@fhs-hagenbe...
// Created: August 24, 2001
// -----
public class Result
{
    ...

    // -----
    // returns the number of grades processed so far
    // -----
    public int getCount() { ... }

    // -----
    // process a grade and incorporate it into the result
    // -----
    public void process(int grade) { ... }

    // -----
    // print the result
    // -----
    public void print() { ... }
}

```

You may take a look at the full source code of the application¹ and of the object type class².

1.1.3 The *this* Pointer

From the previous explanations, it becomes clear that there is no fundamental distinction between classes that represent applications and classes that represent object types. We have just a single construct `class` which may contain the following entities (omitting for the moment classes declared within other classes):

```

public class Class
{
    // class fields
    public static type variable = expression;
    public static final type constant = expression;

    // class methods

```

¹Exams3.java

²Result2.java

```
public static type method(parameters)
{
    ...
}

// object fields
public type variable = expression;
public final type constant = expression;

// constructors
public Class(parameters)
{
    ...
}

// object methods
public type method(parameters)
{
    ...
}
}
```

In order to explain which entity can refer to which other entity, we have to explain what the term “bound to an object” actually means, i.e., in which way entities declared with the keyword `static` (class fields and class methods) differ from entities declared without this keyword (object fields and object methods). This can be explained best with the concept of the *this* pointer. This pointer refers to the “current object” and is in Java represented by the special constant

`this`

which can be used in object field declarations, constructors, and object methods. The short explanation is now as follows.

Object fields and object methods can be only accessed by qualification with an object. For unqualified access (referring to a field or calling a method of the “current” object), we need the *this* pointer. Object field declarations and object methods receive the *this* pointer; they can therefore refer to object fields and call object methods in the “current” object. Class field declarations and class methods do not receive the *this* pointer; they can therefore not refer to object fields or call object methods in the “current” object.

A more detailed explanation with examples is given below; this may be omitted on first reading.

`static` entities do not have the *this* pointer.

Fields (Variables/Constants)

An *object field* is a field declared *without* the keyword `static`. For an object field, an instance is allocated in *every object* of the class. The object field is initialized when the object is created; the declaration of an object field receives in the *this* pointer a reference to the object which is being created.

An object field *name* is referenced as

object.name

where *object* denotes the object in which we want access to the field. Every unqualified reference

name

is automatically translated into a qualified reference

this.name

where the pointer *this* refers to the current object. Therefore an entity can refer to an object field without qualification only if it has the *this* pointer. Since an object field declaration has the *this* pointer, it can refer to other object fields in the same class without qualification.

Example The declaration

```
public class Numbers
{
    public int zero = 0;
    public int one = zero+1;
}
```

is identical to the declaration

```
public class Numbers
{
    public int zero = 0;
    public int one = this.zero+1;
}
```

A *class field* is a field declared *with* the keyword `static`. It is allocated in a special memory area reserved for the class. Therefore there exists only *one* instance of a class field. A class field is initialized when the JVM interpreter loads the corresponding class. A class field declaration does therefore *not* receive the *this* pointer and can consequently *not* refer to an object field without qualification.

Example The declaration

```
public class Numbers
{
    public int zero = 0;
    public static int one = zero+1;
}
```

results in the compiler error

```
Numbers.java:4: non-public static variable zero cannot be referenced
                from a public static context
    public static int one = zero+1;
                        ^
```

A class field *name* is referenced as

Class.name

where *Class* denotes the class in which we want access to the field. Every unqualified reference

name

is automatically translated into a qualified reference

Class.name

using the name *Class* of the current class. Any entity in a class declaration can therefore refer to a class field in the same class without qualification.

Example The declaration

```
public class Numbers
{
    public static int zero = 0;
    public int one = zero+1;
}
```

is identical to

```
public class Numbers
{
    public static int zero = 0;
    public int one = Numbers.zero+1;
}
```

Methods and Constructors

An *object method* is method declared *without* the keyword `public static` (also a constructor is considered as an object method). In addition to the declared parameters, an object method has an implicit parameter *this*. It can therefore refer to an object field in the same class without qualification.

Example The declaration

```
public class Numbers
{
    public int one = 1;
    public int add1(int n)
    {
        return n+one;
    }
}
```

is identical to declaration

```
public class Numbers
{
    public int one = 1;
    public int add1(int n)
    {
        return n+this.one;
    }
}
```

An object method *name* can be called as

```
object.name(...)
```

where *object* denotes the object in which we want to call the method. The parameter *this* of the method then receives the value of *object*. Every unqualified call of an object method

```
method(...)
```

is automatically translated into a qualified call

```
this.method(...)
```

Therefore an entity can call an object method without qualification only if it has the *this* pointer. Since an object method has the *this* pointer, it can call object methods in the same class without qualification. The value of the *this* pointer is thus implicitly forwarded from one object method of a class to another method of the same class.

Example Take the declaration

```
public class Numbers
{
    public int mult(int n, int m)
    {
        return n*m;
    }
    public int square(int n)
    {
        return mult(n, n);
    }
}
```

The second method is translated into

```
public int square(int n)
{
    return this.mult(n, n);
}
```

which is okay, because it is an object method which receives the *this* pointer.

A *class method* is a method declared *without* the keyword `static`. A class method does *not* receive the *this* pointer; it can therefore not refer to object fields or call object methods without qualification.

Example The declaration

```
public class Numbers
{
    public int one = 1;
    public static int add1(int n)
    {
        return n+one;
    }
}
```

results in the compiler error

```
Numbers.java:6: non-public static variable one cannot be referenced
                from a public static context
    return n+one;
               ^
```

A class method *name* can be called as

```
Class.name(...)
```

where *Class* denotes the class in which we want to call the method. Every unqualified call of a class method

```
name(...)
```

is automatically translated into a qualified call

```
Class.method(...)
```

using the name *Class* of the current class. Any entity in a class declaration can therefore refer to a class method in the same class without qualification.

Example The declaration

```
public class Numbers
{
    public int one = add(0, 1);
    public static int add(int n, int m)
    {
        return n+m;
    }
}
```

is identical to

```
public class Numbers
{
    public int one = Numbers.add(0, 1);
    public static int add(int n, int m)
    {
        return n+m;
    }
}
```

1.2 Class Libraries and Packages

A *class library* is a set of classes that support the development of programs. Every Java system is shipped with a class library called the *Java Core API (Application Programming Interface)*. The classes of this library are grouped into several *packages*; each package holds a set of classes with related functionality. Some important packages of the Java API³ are listed below:

Name	Description
java.applet	Create applets
java.awt	Abstract Windowing Toolkit
java.io	Input and output functions
java.lang	General language support
java.net	Computing across the network
java.util	General utilities

³<http://docs.oracle.com/javase/7/docs/api>

Package `java.lang` is special in the sense that we can use the classes in this package (e.g. class `String` or class `Integer`) without provision; this package is *automatically imported*.

To a class *Class* in any other package *package* we can refer by the qualified name *package.Class*, e.g. `java.util.Vector`. It is, however, more convenient to use an import declaration of one of the forms

```
import package.Class;
import package.*;

public class MyClass
{
    ...
}
```

The first form allows to refer to *Class* without qualification; e.g., the declaration

```
import java.util.Vector;
```

makes the class name `Vector` available. The second form makes all classes of *package* available, e.g.

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

allows to use `Vector`, `Hashtable`, `Stack`, and all other classes of `java.util`.

1.2.1 Writing Packages

The following examples use the Unix path separator `/`. In MS Windows, you have to use `\` instead.

You can write your own package by placing the source code of all classes of the package into a subdirectory of the same name. For instance, the directory

```
/home/ws/
  Main.java
cs1/
  Input.java
  A.java
```

defines a package `cs1` with classes `Input` and `A`. Each of the `class` declarations must be prefixed by a declaration

```
package package;
```

For instance, file `Input.java` may have content

```
package cs1;

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

public class Input
{
    ...
}
```

The declaration must import the classes used from other packages; it is not necessary to import the packages of the *current* package. The keyword `public` *exports* the class from the package, i.e., it allows classes from other packages to use `Input`. If `public` is omitted, only classes within the package `cs1` may use `Input`.

If the current working directory is `/home/ws`, you can then run

```
javac cs1/Input.java
```

to generate file `cs1/Input.class`. The main program stored in file `Main.java` may then have declaration

```
import cs1.*;

public class Main
{
    ...
}
```

to use the classes of package `cs1` without qualification.

One can construct hierarchies of packages with qualified names; the directory structure has to correspond to the individual names. For instance, the classes for package `cs1.basic` must be stored in subdirectory `cs1/basic/`.

1.2.2 The Package `java.lang`

The automatically imported package `java.lang` provides various classes which are so commonly used that they can be considered as part of the language. It is a good idea to study this package in some detail. For the beginning, the most important classes are the following:

Name	Description
Boolean	A wrapper for type <code>boolean</code>
Byte	A wrapper for type <code>byte</code>
Character	A wrapper for type <code>char</code> and utility methods
Double	A wrapper for type <code>double</code> and utility methods
Float	A wrapper for type <code>float</code> and utility methods
Integer	A wrapper for type <code>int</code> and utility methods
Long	A wrapper for type <code>long</code> and utility methods
Math	Various numerical (floating point) operations
Short	A wrapper for type <code>short</code>
String	Sequences of characters
StringBuffer	Mutable sequences of characters
System	Several useful class fields and class methods

We have already used various classes of this library. For instance, our standard output statement

```
System.out.println(string)
```

uses the class variable *out* in class `System` declared as

```
public static PrintStream out
```

where `PrintStream` is a class in package `java.io` which provides the object method

```
void println(String)
```

Various classes “wrap” a primitive Java type into an object type. For instance, the declarations

```
int val = 7;  
Integer i = new Integer(val);  
int value = i.intValue();
```

wraps an `int` value into an `Integer` object and then extracts the `int` value. Usually it is not necessary to deal with these classes explicitly because Java performs automatic wrapping/unwrapping from atomic types to/from the corresponding object types whenever needed (*autoboxing*).

Example (Separating Words)

We write a program that reads a string which contains multiple words (sequences of letters) separated by other characters. The program then prints all words of the text in separate lines in the order of their occurrence in the text. For instance, the input

```
One, two, and three!
```

shall result in output

```
One
two
and
three
```

For this purpose, we may use the `Character` method

```
public static boolean isLetter(char ch)
```

which returns true if and only if character *ch* denotes a letter.

The core task is thus described by the following method:

```
public static void printWords(String text)
{
    int i = 0;
    final int end = text.length();
    while (i < end)
    {
        int a = wordBegin(text, i);
        assert a == end || Character.isLetter(text.charAt(a));
        if (a == end) break;
        int b = wordEnd(text, a);
        assert b == end || !Character.isLetter(text.charAt(b));
        System.out.println(text.substring(a, b));
        i = b+1;
    }
}
```

We repeatedly look, from the current text position *i*, for the start position *a* of the next word. If no more word exists, we terminate the loop. Otherwise, we look, from position *a*, for the position *b* after the word and print the text from *a* to *b*. We continue our search with the next position after *b*.

The remaining methods can be implemented as follows:

```
public static int wordBegin(String text, int i)
{
    int a = i;
    final int end = text.length();
    while (a < end && !Character.isLetter(text.charAt(a)))
        a = a+1;
    return a;
}

public static int wordEnd(String text, int a)
{
    int b = a+1;
    final int end = text.length();
    while (b < end && Character.isLetter(text.charAt(b)))
        b = b+1;
    return b;
}
```

The main method then is:

```
public public static void main(String[] args)
{
    String text = Input.readString();
    printWords(text);
}
```

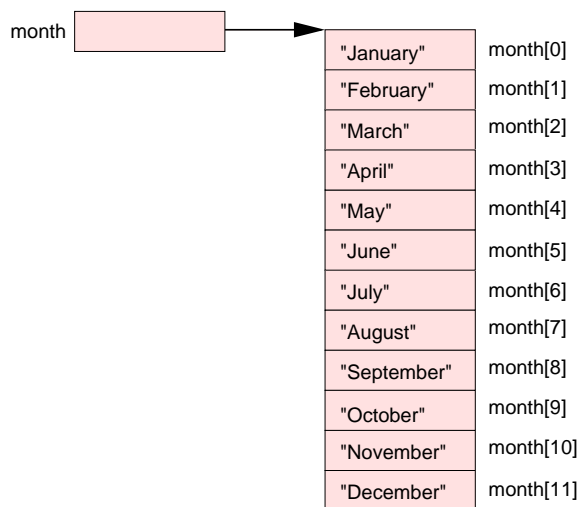
Please take a look at the commented source code⁴.

1.3 Arrays

We now consider a data structures which allows us to process not only individual data but whole collections of data organized in tables. An *array* is a such a table of homogeneous values, i.e., all table values are of the same type. Each value is stored at a particular table position denoted by a numerical *index*. Each table entry referenced by an index denotes a variable that can be individually read or written. Figure 1.1 shows the memory layout of an array *month* that holds the names of the months.

In Java, an array is represented like an object by a pointer that references the table data; assignments and comparisons of array variables are therefore assignments

⁴Words.java

Figure 1.1: An array *month*

and comparisons of array pointers, *not* of the actual array data. The special value `null` can be also used as an array values and represents the *default value* for array types.

1.3.1 Array Declaration and Creation

An *array declaration* is of the form

```
type[] name
```

and may appear in form of a variable/constant declaration. It declares a variable/constant *name*; the square brackets `[]` denote that *a* does not hold a value of the denoted *type* but holds a reference to an array of elements of this *type*. An alternative declaration form is

```
type name[]
```

which has the same effect.

An array is *created* by an expression

```
new type[length]
```

where *length* is an expression that evaluates to an integer value which denotes the number of elements in the array. Each array element is initialized with the default value of *type*.

The array in Figure 1.1 can be therefore declared and initialized as

```
String[] month = new String[12];
```

If a small array is to be created and initialized at its declaration, this can be written as

```
type[] name = {expressions};
```

where *expressions* denotes a list of values (of the denoted *type*) by which the array elements are initialized. Array *month* can be therefore also declared and initialized as

```
String[] month = {"January", "February", ..., "December"};
```

If at a later time another such small array is to be assigned to a previously declared array variable, this can be written as

```
name = new type[] {expressions};
```

For instance, we might write

```
month = new String[] {"January", "February", "March"};
```

1.3.2 Array Referencing

An individual array element is referenced by an expression

```
array[index]
```

where *array* is an expression that denotes an array and *index* is an integer expression that denotes the position of the element in the array. Indexing starts with 0, i.e., the elements of an array with *length* elements can be referenced by indices 0 to *length* - 1.

The array in Figure 1.1 can be thus written as

```
month[0] = "January";
month[1] = "February";
...
month[11] = "December";
```

Its elements can be printed by the code

```
System.out.print("Enter month (1-12): ");
int i = Input.readInt();
System.out.println("Month " + i + ": " + month[i-1]);
```

which gives rise to the following dialogue:

```
Enter month (1-12): 2
Month 2: February
```

It is an error to refer to an error element by an index which is outside the allocated range. For instance, the access

```
System.out.println(month[12]);
```

triggers the runtime error:

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException
    at Main.main(Main.java:6)
```

To allow such *automated bound checking*, each *array* holds its length in an object constant

```
array.length
```

which is especially useful if an array is passed as a parameter to a method (which does not know with which length the array was created).

Example (Command-Line Arguments)

The method `main` in an application has interface

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

The parameter `args` contains the arguments passed to the application from the command line. For instance, take the program

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

The body of this method shows how an array of length n is typically processed: by a for loop whose iteration variable i runs from index 0 and stops when i becomes n .

Executing this program as

```
java Main -option input 2
```

gives output

```
-option
input
2
```

1.3.3 Array Processing

To process all elements of an array a , we have in above example used a for loop of the form

```
for (int i = 0; i < a.length; i++)
{
    ... process a[i] ...
}
```

where an index i runs over all positions of a to retrieve every element $a[i]$.

However, Java provides an alternative syntax of the for loop where we can access the array elements without the use of an index:

```
for (T e : a)
{
    ... process e ...
}
```

Here the local loop variable e is subsequently set to every element of a ; the type T of e must therefore match the base type of the array. This form is always applicable when we are not interested in the actual index of e in a .

Example We can write a function `printArray` that prints all elements of an integer array as

```
public static void printArray(int[] a)
{
    for (int e : a)
        System.out.println(e);
}
```

However, in many cases we actually need to deal with the array indices as demonstrated by the following example.

Example (Searching in an Array)

We want a class function `search` which takes an integer array a and an integer x and which returns the index of the first occurrence of x in a . If x does not occur in a , -1 is returned.

This function can be implemented as follows:

```
public static int search(int[] a, int x)
{
    final int n = a.length;
    for(int i = 0; i < n; i++)
        if (a[i] == x) return i;
    return -1;
}
```

It is a good idea to perform in a loop (which may be iterated a large number of times) only those operations that are absolutely necessary. We therefore declare a constant n to hold the length of a rather than accessing the constant field `length` in each iteration.

Example (Prime Number Computation)

We are going to write a program which computes and prints all prime numbers less than a given number n . For this purpose, we hold an array p of primes computed so far and test every new candidate i whether it is divided by any element of this array. If not, it is added as a new prime to this table.

The core function can be written as follows

```
public static void printPrimes(int n)
{
    if (n < 2) return;
    System.out.println(2);           // 2 is prime

    int len = 1000;                  // size of p
    int[] p = new int[len];          // prime table
    int l = 0;                       // number of elements in p

    for (int c = 3; c < n; c += 2) // check odd candidates c
    {
        // c is not prime, next iteration
        if (!isPrime(c, p, l)) continue;

        System.out.println(c);
        if (l == len)                // table is full
        {
            len = 2*len;             // double size of table
            p = resize(p, len);
        }
        p[l] = c;                    // enter c into table
        l = l+1;
    }
}
```

Since 2 is the only even prime, we deal with it separately and further on only investigate odd candidates c . If method `isPrime` detects c as prime, we enter c into the table. However, if the table is already full, we first have to create a new table of double size by method `resize`.

The check of whether c is prime with respect to table p into which l primes have been entered is as follows:

```
public static boolean isPrime(int c, int[] p, int l)
```

```
{
  for (int i = 0; i < l; i++)
    if (c % p[i] == 0) return false;
  return true;
}
```

Finally, the following method takes table p and returns a new table of size len into which the elements of p have been copied:

```
public static int[] resize(int[] p, int len)
{
  int l = p.length;
  int[] q = new int[len];
  for (int i = 0; i < l; i++)
    q[i] = p[i];
  return q;
}
```

Please take a look at the documented source code⁵ of the application.

1.3.4 Arrays and Objects

By combinations of objects and arrays complex information structures can be constructed as it is typical in larger applications. In the following, we will outline two frequently occurring patterns.

Objects Containing Arrays

Frequently, a data structure does not only consist of an array but contains additional data (e.g. indices) that are required to process the array. Rather than keeping these pieces in separate variables, one may pack them into a single object that provides via methods access to the data structure.

Example (Prime Number Computation)

In the previous example, a prime number table was used to determine whether a new candidate number is prime. This table consisted of two pieces of data, an array p and a number l that denotes how many elements have been already entered into the table. We now declare a class that combines these pieces of information:

⁵Primes.java

```
public class Table
{
    public int[] p = new int[1000]; // the table data
    public int l = 0;                // number of elements in p

    public void add(int e)
    {
        if (l == p.length) resize();
        p[l] = e;
        l = l+1;
    }

    public void resize()
    {
        int l = p.length;
        int[] q = new int[2*l];
        for (int i = 0; i < l; i++)
            q[i] = p[i];
        p = q;
    }

    public int getNumber()
    {
        return l;
    }

    public int getElement(int i)
    {
        return p[i];
    }
}
```

All functionality of adding an element to the table and resizing it on demand is now hidden behind the object method `add`. The object method `getNumber` returns the number of elements in the table; the object method `get` allows to retrieve any element by its index.

The core function is thus considerably simplified because it does not have to deal with the current size of the table any more:

```
public static void printPrimes(int n)
{
    if (n < 2) return;
    System.out.println(2);        // 2 is prime
```

```
Table p = new Table();           // prime table
for (int c = 3; c < n; c += 2) // check odd candidates c
{
    if (!isPrime(c, p)) continue;
    System.out.println(c);
    p.add(c);
}
}
```

The prime check itself becomes:

```
public static boolean isPrime(int c, Table p)
{
    final int l = p.getNumber();
    for (int i = 0; i < l; i++)
        if (c % p.getElement(i) == 0) return false;
    return true;
}
```

Please take a look at the documented source code of the table type code⁶ and of the application⁷.

Arrays Containing Objects

Up to now arrays, we have only used arrays that contained values of primitive types. However, in general an array may also contain objects. For instance, the declaration

```
Class[] name = new Class[length];
```

declares and creates an array *name* that may hold *length* objects of type *Class*. However, the entries of *name* are initialized with the default value of *Class* which is the null pointer. Therefore we have the situation depicted in Figure 1.2, i.e., the array elements do not yet refer to any objects.

To create the array objects, we have to execute the following piece of code

```
for (int i = 0; i < length; i++)
    name[i] = new Class(arguments);
```

where each object is initialized by a call of a *Class* constructor (possibly the default constructor). We then have the situation depicted in Figure 1.3.

⁶Table.java

⁷Primes2.java

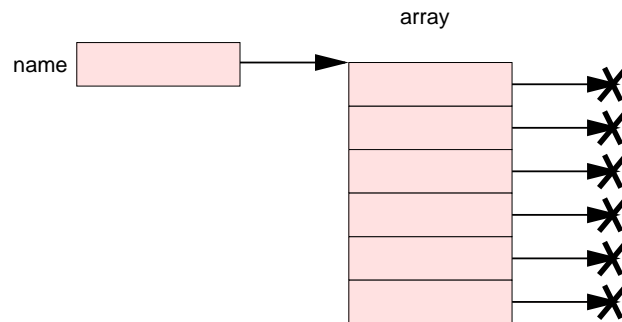


Figure 1.2: Array with null Pointers

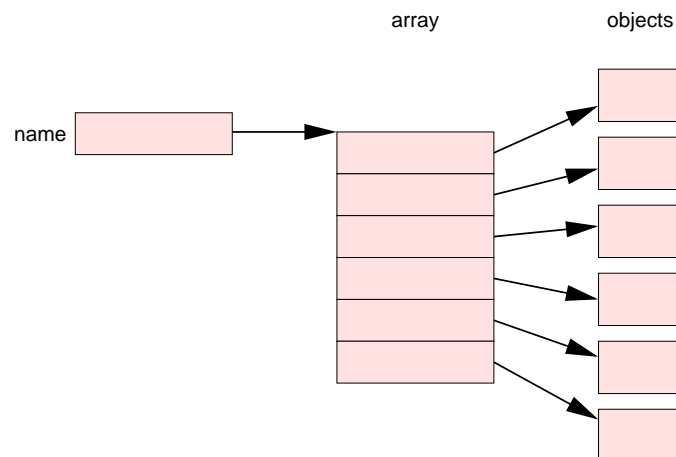


Figure 1.3: Array with Object Pointers

Example (Phone Book)

We write a program that reads a list of phone book entries, i.e., pairs of a name and a phone number. The program then reads sequences of names and prints the corresponding phone number. A phone book entry is an object of the following class:

```
public class Entry
{
    public String name;
    public String number;

    public Entry(String name, String number)
    {
        this.name = name;
        this.number = number;
    }

    public String getName()
    {
        return name;
    }

    public String getNumber()
    {
        return number;
    }
}
```

The following function asks the user for a phone book entry and then reads the entry (we omit error checking in this example):

```
public static Entry readEntry()
{
    System.out.print("Enter name: ");
    String name = Input.readString();
    System.out.print("Enter number: ");
    String number = Input.readString();
    return new Entry(name, number);
}
```

The function creating the phone book is then defined as follows:

```
public static Entry[] readPhoneBook()
{
    System.out.print("How many entries? ");
    int n = Input.readInt();
    Entry[] book = new Entry[n];
    for (int i=0; i<n; i++)
    {
        System.out.println("Entry " + (i+1) + ":");
        book[i] = readEntry();
    }
    return book;
}
```

Looking up for a name is then achieved by the following function:

```
public static Entry search(Entry[] book, String name)
{
    final int n = book.length;
    for(int i = 0; i < n; i++)
        if (book[i].getName().equals(name)) return book[i];
    return null;
}
```

This function takes `book[i]`, applies the object method `getName()` to extract the name and then calls the string method `equals` to compare this name with a given name. If the name is not found in the phone book, `null` is returned.

The complete procedure for phone book lookup is then the following (omitting error checking):

```
public static void usePhoneBook(Entry[] book)
{
    while (true)
    {
        System.out.print("Another lookup (y/n)? ");
        char ch = Input.readChar();
        if (ch == 'n') return;
        System.out.print("Name: ");
        String name = Input.readString();
        Entry entry = search(book, name);
        if (entry == null)
            System.out.println("Name not found.");
        else
```

```
        System.out.println("Number: " + entry.getNumber());
    }
}
```

The main program is then the following:

```
public static void main(String[] args)
{
    Entry[] book = readPhoneBook();
    usePhoneBook(book);
}
```

Please take a look at the documented source code⁸ of this application.

While arrays of objects are commonly used, we do not recommend to use the array itself in its “naked” form as the central data structure of an application. Better use a “wrapper object” that contains the array and provide array access via the methods of this object.

1.3.5 Multi-Dimensional Arrays

The arrays we have examined so far have all been *one-dimensional* in the sense that they represent a sequence of values referenced by a single index. A *two-dimensional* array is a matrix which is referenced by a pair of indices, the first one denoting a row of the matrix, the second one denoting a column within this row. The expression *matrix*[*i*][*j*] then denotes the element of *matrix* at row *i* and column *j* (see Figure 1.4).

In Java, a two-dimensional matrix is represented by a one-dimensional array whose elements refer to the matrix rows (see Figure 1.5). Such a two-dimensional array *name* is declared as

```
type[][] name
```

where *type* denotes the element type of the matrix; it is created by

```
new type[rows][columns]
```

where *rows* and *columns* are integer expressions that denote the number of matrix rows respectively columns. For instance,

⁸PhoneBook.java

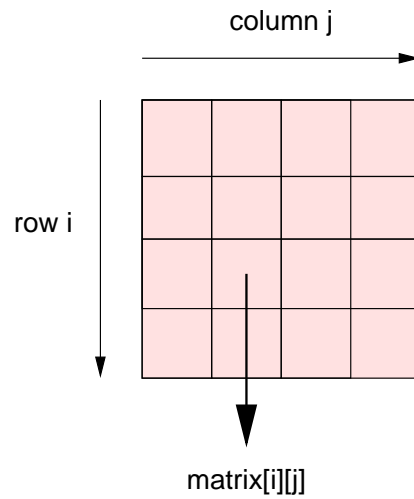


Figure 1.4: A Matrix

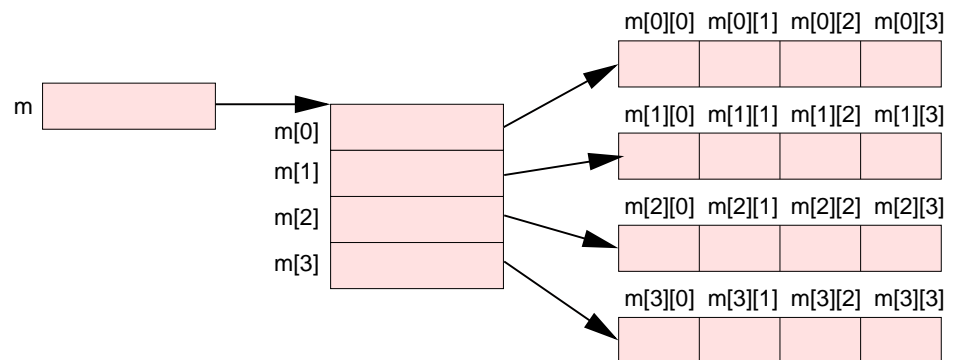


Figure 1.5: A Two-Dimensional Array

```
int[][] m = new int[2][3];
```

declares and allocates a matrix m with 2 rows and 3 columns.

We can access a matrix element as

```
name[row][column]
```

where row and $column$ are integer expressions that denote the row number respectively the column number of the element. The expression

```
name.length
```

returns the number of rows in the matrix, the expression

```
name[0].length
```

returns the number of columns in row 0 (which is the same for all rows if the matrix has been declared as shown above).

A two-dimensional array can be initialized in the declaration as

```
type[][] name = { {value, ...}, ... };
```

where each entry of the initializer list is a list of values for a particular row. For instance, the declaration

```
int[][] m = { {1, 2, 3}, {4, 5, 6} };
```

initializes m with 2 lines and 3 columns; matrix element $m[1][2]$ is 6. At a later time a matrix variable can receive a new value by an assignment of form

```
name = new type[][] { {value, ...}, ... };
```

for instance

```
m = new int[][] { {1, 2}, {3, 4}, {5, 6} };
```

Example (Matrix Multiplication)

The following program multiplies two matrices a and b giving a result c .

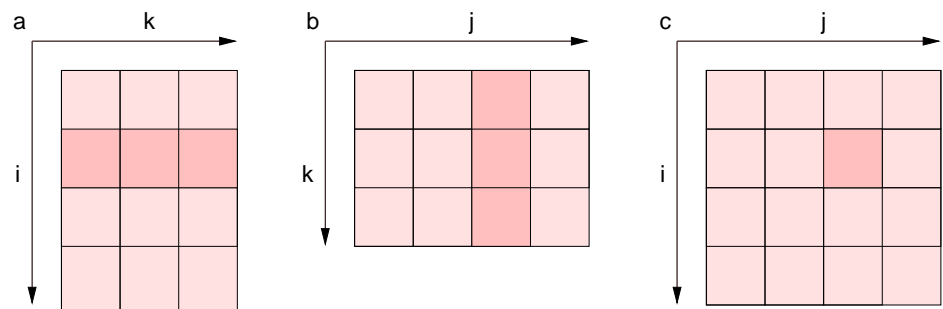


Figure 1.6: Matrix Multiplication

```

public static float[][] matMult(float[][] a, float[][] b)
{
    final int M = a.length;    // a is M x Q
    final int N = b[0].length; // b is Q x N
    final int Q = a[0].length; // c is M x N
    assert Q == b.length;

    float[][] c = new float[M][N];
    for (int i = 0; i < M; i++)
    {
        for (int j = 0; j < N; j++)
        {
            double sum = 0;
            for (int k = 0; k < Q; k++)
                sum = sum + a[i][k] * b[k][j];
            c[i][j] = (float)sum;
        }
    }

    return c;
}

```

Here matrix element $c[i][j]$ is the scalar product of row i of matrix a and column j of matrix b as denoted in Figure 1.6.

The result matrix has as many rows as matrix a and as many columns as matrix b ; the number of columns of a must be the same as the number of rows of matrix b . The outer loop with index i runs over all rows of a , the nested loop with index j runs over all columns of b . The innermost loop with index k multiplies the corresponding elements of row i of a and row j of b and computes the *sum* of

these products.

1.3.6 The Class Vector

The package `java.util` contains a class `Vector` which provides a service similar to an array in that it can store a sequence of values and reference them by an index. But whereas an array has fixed size throughout its existence, a `Vector` object can grow dynamically as needed. A `Vector` variable is created and initialized by a call of form

```
Vector<T> v = new Vector<T>();
```

where T can be an arbitrary object (not atomic) type that represents the type of the elements in the vector (class `Vector` is an example of a *generic* class which will be discussed in more detail in Chapter 4).

Some important methods of the class are the following:

Vector<T>() This constructor creates a vector that is initially empty.

void add(*T* object) This object method adds the specified *object* to the end of *this* vector.

***T* get(int index)** This object method returns the object at the specified *index*.

int size() This object method returns the number of elements in *this* vector.

For more information on the methods of this class, please consult the Java Manual⁹.

Example Take the program code

```
Vector<Integer> v = new Vector<Integer>();
v.add(0);           // autoboxing int -> Integer
v.add(1);
v.add(2);
int i = v.get(1);   // autounboxing Integer -> int
System.out.println(i);
```

This code adds three integer values to a vector and then retrieves the element at index 1; its output is

⁹<https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/Vector.html>

1

Please note that the vector has initially size 0 and is dynamically extended by the sequence of `add` operations. Before putting the `int` values into the vector, they are automatically boxed into `Integer` objects; when the vector elements are retrieved, they are automatically unboxed to `int` values again.

In the following chapter, we will deal more with data structures that can store collections of objects.

Chapter 2

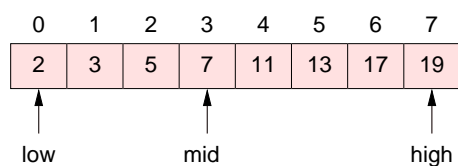
Java with Taste

While in the previous sections we have presented basic mechanisms of the Java programming language, we will in this section use the language to illustrate several important concepts in computer science. We will discuss some algorithms and solution strategies for fundamental problems in computer science and important data structures for storing and retrieving information. The concept of *abstract datatypes* is introduced to distinguish between the specification of a data structure and its implementation.

2.1 Binary Search

We have already seen in the previous section how to search in an array by sequentially running through the array and comparing each element with a given key. In worst case, we have to investigate the whole array to find the desired element (or find out that the element is not in the array). If the array consists of n elements, this requires n comparison operations; therefore we call the search algorithm *linear* in the length of the input array.

There is a better way of searching for an element in an array provided that the array is *sorted*, i.e., the elements of the array are arranged in some (e.g., ascending) order. Assume that we are looking for the element 13 in the array



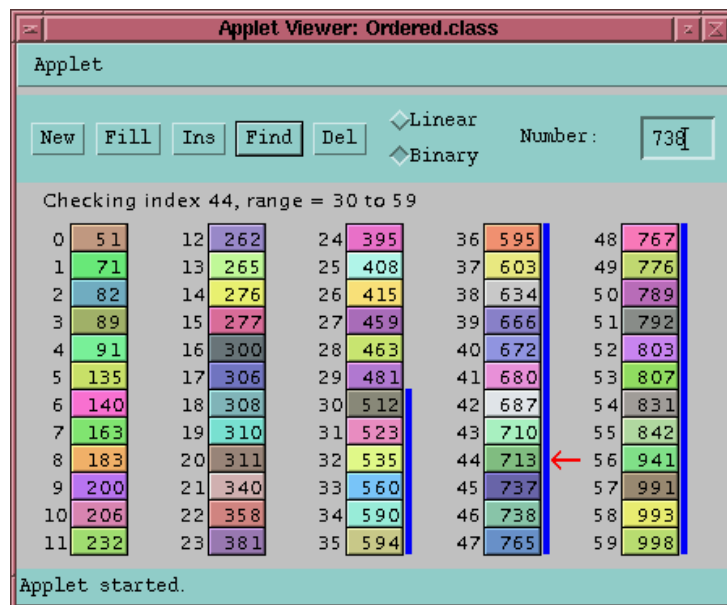


Figure 2.1: Binary Search

We use truncated integer division.

We denote the index range of the array by the variables *low* (initially 0) and *high* (initially 7) and take a look at the element in the middle referenced by the variable *mid* (initially $3 = (low + high)/2$). If this element is equal to 13, we are done. However, the element at this position is 7 which is less than 13. Since the array is sorted, we therefore know that the wanted element is to the right of *mid*. Thus we set the variable *low* to *mid* + 1 and continue the search:

0	1	2	3	4	5	6	7
2	3	5	7	11	13	17	19
				↑	↑		↑
				low	mid		high

The process is iterated until the element is found or $low > high$ (in this case the element is not in the array).

Figure 2.1 shows the screenshot of an applet embedded into the electronic version of this document; this applet illustrates the binary search for the value 738 in an array of 60 elements. The current search range is denoted by the blue line between from low index 30 to high index 49; the middle element at position 44 is denoted by the red arrow.

The algorithm is expressed by the following Java function:

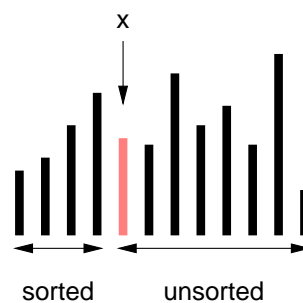
```
// -----  
// i = binarySearch(a, x)  
// 'i' is the index of 'x' in 'a'.  
//  
// Input:  
// 'a' is not null and sorted in ascending order.  
// Output:  
// 'a' remains unchanged.  
// if 'i' == -1, then 'x' is not in 'a'.  
// if 'i' != -1, then 'a[i] == x'.  
// -----  
public static int binarySearch(int[] a, int x)  
{  
    int low = 0;  
    int high = a.length-1;  
    while (low <= high)  
    {  
        // x does not occur in a in any position  
        // less than low or greater than high  
        int mid = (low + high)/2;  
        if (a[mid] == x) return mid;  
        if (x > a[mid])  
            low = mid+1;  
        else  
            high = mid-1;  
    }  
    // x does not occur in a  
    return -1;  
}
```

Since in every iteration the size of the search range (the difference between *high* and *low*) is halved, the function needs at most as many comparisons as correspond to the binary logarithm of the length of the array; we call the algorithm *logarithmic* in the array length. For instance, if the array is $1024 = 2^{10}$ elements long, at most 10 iterations are required to find an element or determine that the element is not in the array. Sorted arrays can therefore be searched very quickly compared with arrays that are not sorted.

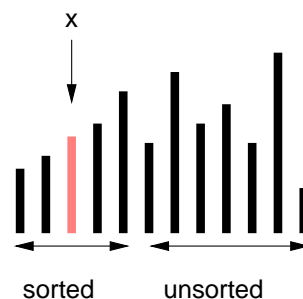
2.2 Simple Sorting

As we have seen in the previous section, having an array sorted is an important prerequisite for later searching it. In general, when using an array as a “database” it is a good idea to keep the elements sorted according to some criterion: names can be sorted in alphabetical order, products by registration numbers, cities by zip codes, and so on. Entities may be also sorted according to different criteria, e.g., a database of books may be sorted according to their titles or their ISBN numbers (but not both at the same time). Because sorting is an important problem in computer science, it has been extensively studied and numerous methods of varying sophistication have been developed. In this section, we will investigate a basic sorting algorithm called *insertion sort* which is good for moderately sized arrays (up to some tens of elements). For larger arrays, we will see later a much faster algorithm.

The basic idea of insertion sort is as follows: we assume that at the start of iteration i of the algorithm, the first i elements of the array are already sorted. The array therefore consists of a left part which is sorted and a right part which is unsorted.



We then take the $(i + 1)$ -th element x and insert it into the appropriate position of the sorted part to the left of x by shifting the elements larger than x one position to the right.



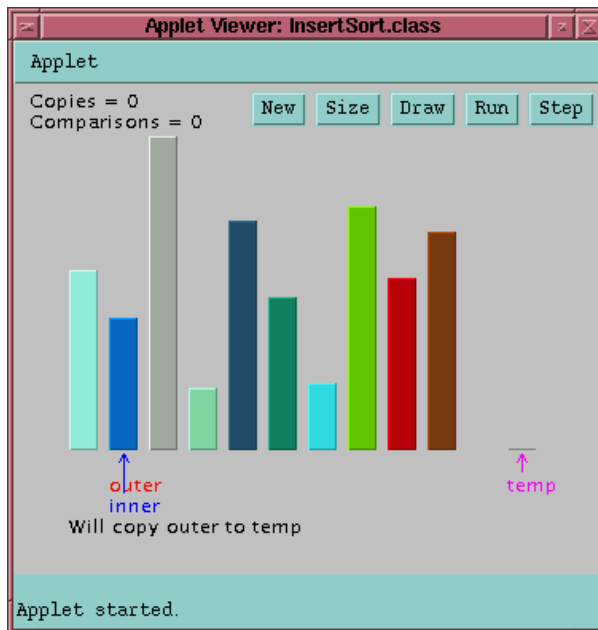


Figure 2.2: Insertion Sort

Now we have the first $i + 1$ elements sorted. After n iterations (where n is the length of the array), therefore the whole array is sorted.

Figure 2.2 shows the screenshot of an applet embedded into the electronic version of this document; this applet illustrates the insertion sort of an array of 10 elements. The arrow label *outer* represents the program variable i above, the arrow label *inner* the variable j (see below). The arrow label *temp* denotes the temporary variable x .

The algorithm is expressed by the following Java method:

```
// -----
// insertionSort(a)
// sort 'a' by the insertion sort algorithm.
//
// Input:
//   'a' is not null.
// Output:
//   'a' has the same elements as the old 'a'.
//   'a' is sorted in ascending order.
// -----
```

```

public static void insertionSort(int[] a)
{
    final int n = a.length;
    for (int i = 1; i < n; i++)
    {
        // a is sorted from 0 to i-1
        int x = a[i];
        int j = i-1;
        while (j >= 0 && a[j] > x)
        {
            a[j+1] = a[j];
            j = j-1;
        }
        a[j+1] = x;
    }
}

```

In this method, we use the variable `x` as a temporary buffer to hold the value of `a[i]` while the larger elements in the sorted part of the array are shifted one position to the right. When all elements have been shifted, this value is written back into the appropriate position of the array.

In the first iteration of the outer loop, the algorithm performs at most one comparison operation, in the second iteration at most two, and so on. In the worst case (the input array is sorted in *descending* order), we therefore have

$$1 + 2 + \dots + (n - 1) = n * (n - 1) / 2 \approx n^2 / 2$$

comparisons (the number of element swap operations is approximately the same). We therefore call the algorithm *quadratic* in the length n of the array. In *average*, however, in each iteration of the outer loop only half of the maximum number of comparisons have to be performed until the appropriate insertion point is found. The algorithm therefore needs in average approximately $n^2/4$ steps.

Why?

Sorting an array by this algorithm is therefore much slower than searching for an element in an array (which takes at most n steps, if the array is unsorted, and at most $\log_2 n$ steps, if the array is sorted). For instance, if the size of the array is 1024, approximately 262000 comparison operations have to be performed to sort the array. If the array has 8192 elements, approximately 16 million comparisons have to be performed.

Why?

However, for data that are already sorted or almost sorted, the insertion sort does much better. When data is in order, the inner loop is never executed. In this case, the algorithm needs only n steps. If the data is almost sorted, the inner loop is

only very infrequently executed and the algorithm runs in “almost” n steps. It is therefore a simple and efficient way to order a database that is only slightly out of order.

2.3 Dynamic Data Structures

Arrays have certain disadvantages as data structures. In an unordered array, searching is slow. In an ordered array, inserting a new element is slow (because all elements after the inserted element have to be shifted by one position to the right) and also deleting an element is slow (because we have to shift all elements after the deleted element one position to the left). Last but not least, the size of an array cannot be changed after creation.

In this section, we will look at a data structure that solves some of these problems: the *linked list*. After arrays, linked list are the second most commonly used general-purpose data structure. They are mainly used in cases where elements are frequently inserted respectively deleted and/or it is hard to predict the number of elements to be stored. They are *not* used for fast searching; for this purpose, there exist better data structures (search trees, hash tables) with which we will deal in a subsequent course.

Lists are used for frequently inserting and deleting an unpredictable number of elements.

2.3.1 List Nodes

A linked list consists of *nodes* each of which holds

- some *value*,
- a *link*, i.e., a reference to another node of the list.

Figure 2.3 illustrates such a list of linked nodes.

The following declaration introduces a corresponding Java class *Node*:

```
public class Node
{
    public int value;
    public Node next;
    public Node(int value, Node next)
    {
        this.value = value;
```

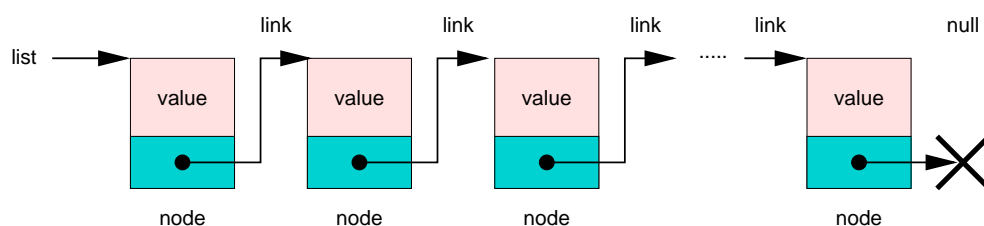


Figure 2.3: Linked List

```

        this.next = next;
    }
    public Node(int value)
    {
        this(value, null);
    }
}

```

A node value may be of any type.

Don't do this in real programs.

This class has an object field *value* carrying the information value (here of type `int`) and an object field *next* which represents the link to the next node. The first constructor initializes the value field to a given value; the link is automatically initialized to `null`. The second constructor initializes both the value and the link of the node. For sake of brevity, we do not introduce methods for returning or updating the content of the value field but will directly access the object field.

A list is just a (reference to a) node. Consequently, a *singleton* list [1], i.e., a list with a single node holding the value 1 can be declared as

```
Node one = new Node(1);
```

Here `one` is a reference to the list node (which itself contains a null reference). We can retrieve its value by the object field `one.value`.

The list [2, 1] can be now created by creating a new node holding the value 2 and linking it with the list [1], i.e.,

```
Node two = new Node(2, one);
```

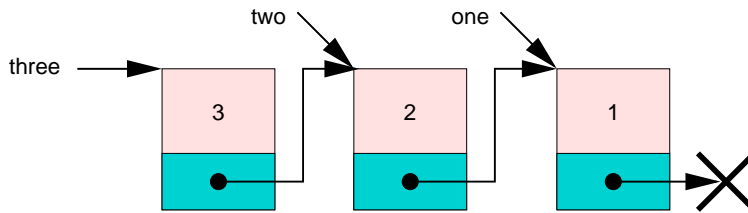
We can retrieve the value of the first node as `two.value` and the content of the second node as `two.next.value`.

Consequently, the list [3, 2, 1] can be created as

Why?

```
Node three = new Node(3, two);
```

We can retrieve the values of the list as `three.value`, `three.next.value`, and `three.next.next.value`. We then have the situation depicted in the following picture:



We see that list `three` consists of three linked nodes; however, we also see that different lists may have overlapping nodes, for instance, the second node of `three` is also the first node of `two`. Updating the content of this node as

```
two.value = -2;
```

has the effect that the statement

```
System.out.println(three.next.value);
```

gives output

```
-2.
```

The possibility of having multiple references to a list node gives a lot of flexibility but it can be also a source of errors. When dealing with dynamic data structures, one has to take care that not unwanted side effects occur by updating a data structure that is also shared with other structures.

What happens if we access `three.next.next.next.value`?

Dynamic data structures may have shared contents.

2.3.2 List Type

A list is actually an abstraction that should be separated from its implementation as a sequence of linked nodes. We therefore introduce the following class declaration:

```
public class List
{
    public Node head;
    public List()
    {
    }
    ...
}
```

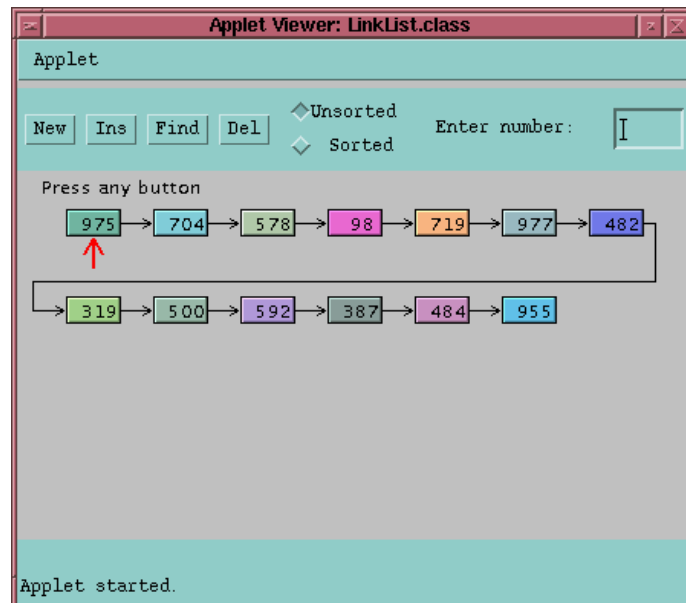


Figure 2.4: List Operations

where the object field `head` refers to the first node of the list (if any). This allows us to define an empty list as

```
List empty = new List();
```

In the following, we will develop various useful list operations as object methods of `List`.

Figure 2.4 shows the screenshot of an applet embedded into the electronic version of this document; this applet illustrates the list operations described in the following sections.

2.3.3 Inserting List Elements

The first operation is that of inserting a new element into a list. Let us for the moment assume that elements are always inserted at the head of the list. This allows us to define the method

```
public class List
{
```

```
...
public void insert(int value)
{
    Node node = new Node(value, head);
    head = node;
}
}
```

which allocates a new list node, sets its content to *value*, links it with the current *head* and then updates *head* to refer to the new node as the first node of the list. We can thus create the list [3, 2, 1] as

```
List l = new List();
l.insert(1);
l.insert(2);
l.insert(3);
```

To make the process of filling lists a bit more convenient, we may make `insert` a program function

```
// -----
// list = insert(value)
// 'list' is the result of inserting 'value' at the head of
// 'this' list.
//
// Output
// 'list == this'
// 'list.head.value == value'
// 'list.head.next == old this.head'
// all nodes referenced via 'old this' remain unchanged
// -----
public List insert(int value)
{
    Node node = new Node(value, head);
    head = node;
    return this;
}
```

which returns the list as its result. Then we can write

```
List l = new List();
l.insert(1).insert(2).insert(3);
```

which also gives list [3, 2, 1].

2.3.4 Searching for List Elements

Suppose we are looking for the first node of a list that holds a particular key value (for instance for retrieving other data stored in the node or for updating these data). The only possibility to find the node is to follow the chain of node links starting with `head`; the search stops when the node is found or when we reach the `null` reference. Thus we develop the following method:

```
// -----
// node = search(value)
// 'node' is the result of searching 'value' in 'this' list
//
// Output:
//   all nodes referenced via 'this' remain unchanged.
//   if 'node == null', then 'value' does not occur in 'this'
//   if 'node != null', then 'node.value == value' and 'node'
//     is the first node in 'this' for which this holds.
// -----
public Node search(int value)
{
    Node node = head;
    while (node != null && node.value != value)
        node = node.next;
    return node;
}
```

The local variable *node* holds a reference to the currently investigated node. Please note that it is *not* correct to exchange the conditions in the loop to

```
while (node.value != value && node != null)
    node = node.next;
```

since we *first* have to know that *node* is not `null` before we are allowed to refer to the object field *value* (short-circuited evaluation of `&&`). After termination of the loop, we know that *node* either is `null` or holds *value*. We can therefore just return the current value of *node* to the caller.

Since the loop structure in this program nicely matches the pattern of the `for`-loop, one may express the method also as

```
Node search(int value)
{
```

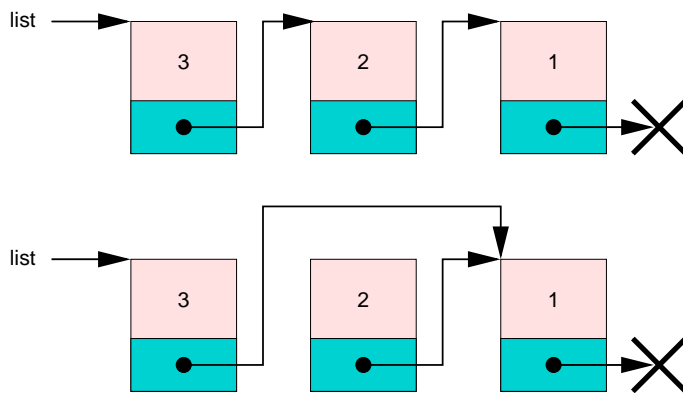


Figure 2.5: Deleting a List Element

```

for (Node node = head; node != null; node = node.next)
    if (node.value == value) return node;
return null;
}

```

In this way, searching for elements in a list closely resembles searching for elements in an unsorted array.

2.3.5 Deleting List Elements

Suppose we want to delete from a list the first node that holds a particular key value. For this purpose, we have to find the appropriate list node and update any reference to this node by a link to the successor node:

- If the deleted node was the first node, *head* has to be updated.
- If the deleted node was not the first node, the link of the predecessor node has to be updated.

Figure 2.5 illustrates the case where the node with value 2 is deleted from a list [3, 2, 1]. Please note that “deleting” a node actually means “unlinking” the node from the list; the node itself still exists (and is unchanged) and may be still referenced from other data structures.

The key idea in the deletion process is to traverse the list looking for the node to be deleted while keeping track of the *predecessor* node of the currently investigated

node. Once the right node is found, we have to update the link of the predecessor. The corresponding Java method is as follows:

```
// -----
// list = delete(value)
// 'list' is the result of deleting the first node that
// holds 'value' from 'this' list.
//
// Output
// 'list == this'.
// if 'value' was not in the old 'this', 'this' equals
// the old 'this'.
// if 'value' was in the old 'this', 'this' equals the
// old 'this' except that the first node with
// 'node.value == value' is unlinked.
// -----
public List delete(int value)
{
    Node prev = null; // previous node
    Node node = head; // current node
    while (node != null && node.value != value)
    {
        prev = node;
        node = node.next;
    }
    if (node != null)
    {
        if (prev == null)
            head = node.next;
        else
            prev.next = node.next;
    }
    return this;
}
```

The critical part of this method is the condition after the iteration. If *node* is not *null*, then it holds *value*, i.e., we have found the node to be deleted. Then, if *prev* is *null*, this is the first node in the list and *head* has to be updated. Otherwise, *prev* references the predecessor of *node* and its link to the next node has to be updated.

Example Since `delete` has been implemented as a program function that returns the updated list as its argument, we can now write statements like

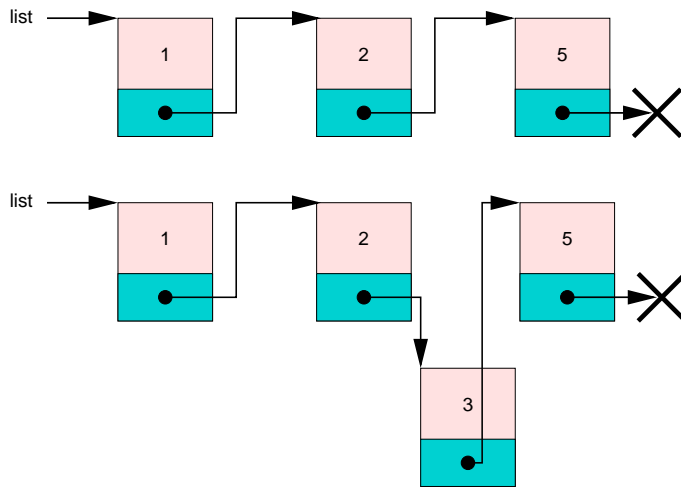


Figure 2.6: Inserting into a Sorted List

```
List l = new List();
l.insert(1).insert(2).insert(1).delete(1).delete(2).delete(2);
```

Which elements does *l* contain after these update operations?

2.3.6 Keeping a List Sorted

Suppose we want to keep a list sorted, i.e., we want to insert new elements not at the head of the list but at those positions that correspond to the order of their values. Figure 2.6 illustrates the case where a node with value 3 is inserted into the list [1, 2, 5] should result in the list [1, 2, 3, 5].

The insertion process is similar to the deletion process explained in the previous section. We have to maintain a reference to the node which is currently investigated. If we find out that the new node has to be inserted before the current node, we have to update the link in the predecessor node. The corresponding Java method is the following:

```
// -----
// list = insertSorted(value)
// 'list' is the result of inserting 'value' into
// the sorted 'this' list.
```

```

//
// Input
// 'this' is sorted.
// Output
// 'list == this' and 'this' is sorted.
// 'this' is the same as the old 'this' except that
// a new node with 'value' has been inserted.
// -----
public List insertSorted(int value)
{
    Node prev = null; // previous node
    Node node = head; // current node
    while (node != null && node.value < value)
    {
        prev = node;
        node = node.next;
    }
    Node n = new Node(value, node);
    if (prev == null)
        head = n;
    else
        prev.next = n;
    return this;
}

```

When we have found the *node* before which the new node is to be inserted, we allocate a new node *n* which holds *value* and link it to *node*. Then we update either the *head* reference (if the new node becomes the first node of the list) or the link in the predecessor node *prev*.

If a list is sorted, we can optimize the method `search` (see page 54) a bit by terminating the search when a node has been found that holds a value greater than the value looked for. The corresponding method becomes

```

// -----
// node = search(value)
// 'node' is the result of searching 'value' in 'this' list
//
// Output:
// all nodes referenced via 'this' remain unchanged.
// if 'node == null', then 'value' does not occur in 'this'
// if 'node != null', then 'node.value == value' and 'node'
// is the first node in 'this' for which this holds.

```

```
// -----  
public Node search(int value)  
{  
    for (Node node = head; node != null; node = node.next)  
    {  
        if (node.value == value) return node;  
        if (node.value > value) return null;  
    }  
    return null;  
}
```

In the unsorted case, always the whole list has to be traversed to find out that an element is not in the list. But also in the sorted case, in average half of the list has to be traversed; therefore the performance improvement is only moderate. Sorted lists are therefore used only, if we want for other reasons to process the elements in some order.

2.4 Abstract Data Types

Arrays and linked lists are data structures which can be used for the implementation of various kinds of *collections*, i.e., objects that serve as repositories for other objects and provide access methods with particular properties. For instance, the previously presented collection class `Table` was implemented by an array (but could be also implemented by a linked list). We call such data structures with multiple implementation possibilities *abstract data types (ADTs)*. In this section, we will discuss *stacks* and *queues* as two examples of abstract data types. However, first we are going to introduce another Java construct which becomes useful in this context.

2.4.1 Interfaces

Until now we have used the term *interface* to denote the public part of a class; this part consists of the methods by which we can interact with objects of the class. The interface hides the class *implementation*, i.e., the object fields which hold the contents of the object. This separation between interface and implementation is also supported by a special Java construct.

A Java *interface* is a collection of class constants and *abstract object methods* declared as follows:

We separate the interface of a class from its implementation.

```
public interface Name
{
    public static final type name = expression; // class constant
    public type name(parameters);             // abstract method
    ...
}
```

The Java identifier *Name* is the name of the interface; as a convention, we use names with upper-case capital letters. The keyword `public static final` is redundant and may be omitted. Please note that the abstract method declaration does not contain a body.

Such an interface expresses the public part of a class that represents the type of an object. A class *implements* such an interface if it is declared as

```
public class Class implements Name
{
    public type name(parameters)
    {
        ...
    }
}
```

This class contains a method implementation for every abstract method declared in the interface. The method declaration in the class must have the same return type and parameter list as the declaration in the interface. The method must be declared with the keyword `public`. The basic idea of this declaration is depicted in Figure 2.7: objects that are of type *Class* can “hide” behind a “shield” represented by the interface type *Name*. The details will be explained in the following section.

Example Take the interface

```
public interface Container
{
    public static final int size = 100;
    public void addElement(int x);
    public int getElement(int i);
}
```

This interface is implemented by the following class:

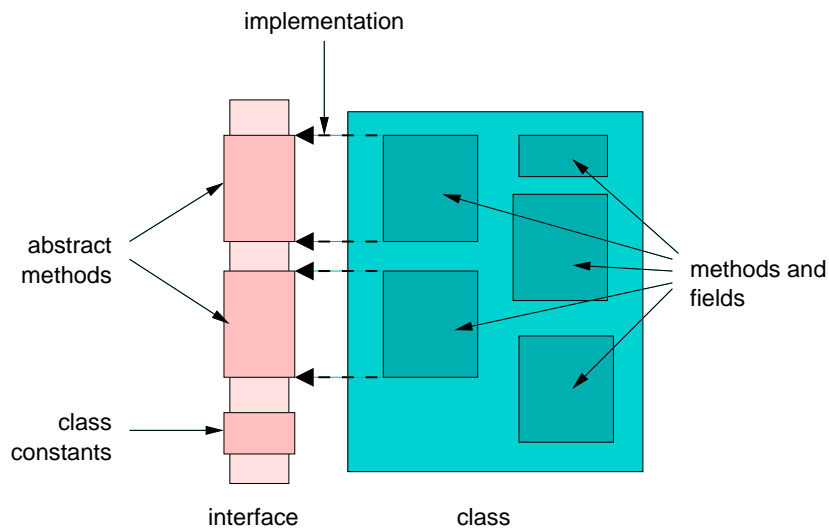


Figure 2.7: Interface and Class

```

public class Array implements Container
{
    public int[] a = new int[size];
    public int l = 0;

    public void setElement(int i, int x)
    {
        a[i] = x;
    }

    public void addElement(int x)
    {
        setElement(l, x);
        l++;
    }

    public int getElement(int i)
    {
        return a[i];
    }
}

```

Please note that the class implementation can refer to the class constant `size` declared in the interface. Furthermore, the implementation has fields `a` and `l` and

a method `setElement` that are not visible in the interface.

A class may also implement multiple interfaces by a declaration of the form

```
public class Class implements Name1, Name2, ...
{
    ...
}
```

In this case, the class must provide a method for every abstract method in every interface listed in the class declaration.

Polymorphism via Interfaces

An interface name may be used to declare the *type* of a variable; the *value* of the variable may be (a reference) to an object of *any* class that implements this interface. We may use the interface methods of the object without any more knowing the actual type of the object.

Example Using the interface and class declarations of the previous section, we may create and use a container object as follows:

```
Container c = new Array();
c.addElement(1);
```

However, it is *not* possible to execute

```
c.setElement(0, 1);
```

because `c` is of type `Container` which does not contain a method `setElement`. For the same reason, it is not possible to refer to the object field `c.l`.

A variable which has an interface type is called *polymorphic* (“of many forms”), because it may hold values of different types. This allows to write methods which operate in the same way on objects of different types.

Example Take the method

```
public int getFirst(Container c)
{
    return c.getElement(0);
}
```

This method may take any `Container` object and return its first element. Therefore, we may write

```
Container a = new Array();
a.addElement(1);
int i = getFirst(a);
```

Now assume we have a class declaration

```
public class List implements Container
{
    ...
}
```

Then also the following code is legal

```
Container l = new List();
l.addElement(1);
int i = getFirst(l);
```

If a class is a special implementation of a more general concept, it is a good idea to define the general concept by a Java interface and let the class implement this interface. When we use the object, we should then refer to the *interface* type and not to the object type. If we later decide to use another implementation of the concept, we may do this without changing the code that uses this concept.

Example Take the code

```
Container c = new Array();
c.addElement(1);
int i = getFirst(c);
```

We may replace the implementation of `c` by changing the first line to

```
Container c = new List();
```

The rest of the code remains unchanged.

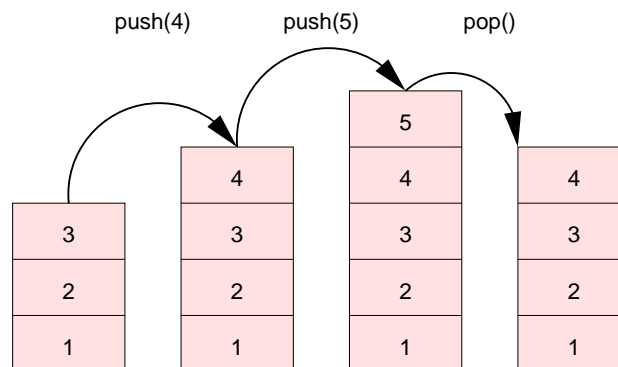


Figure 2.8: Stack Operations

2.4.2 Stacks

A *stack* is a pile of items. We may place a new item on the top (“push”) and remove an item from the top (“pop”), but we do not have access to the items below the top. The last item pushed is always the first item popped; a stack is therefore also called a *LIFO* data structure (“last in, first out”). This idea is illustrated by Figure 2.8. Figure 2.9 shows the screenshot of an applet embedded into the electronic version of this document; this applet demonstrates the usage of the various stack operations.

The abstract datatype “stack of integers” is represented by the following Java interface:

```
public interface Stack
{
    // -----
    // b = isEmpty()
    // 'b' is true exactly if 'this' stack has no elements
    // -----
    public boolean isEmpty();

    // -----
    // push(t)
    // push 't' on top of 'this' stack
    //
    // Output:
    // 'this' is the old stack with the new top 't' i.e.
    // 'pop() == t' and 'top() == t'
```

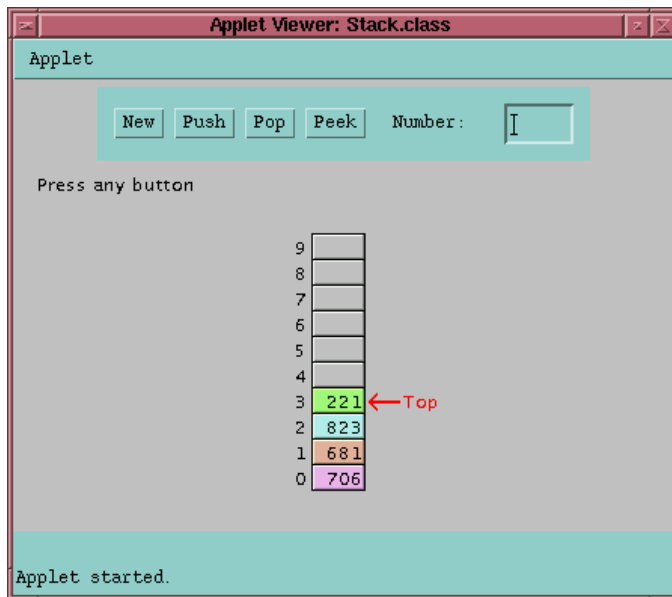


Figure 2.9: Stack Applet

```
// -----
public void push(int t);

// -----
// t = pop()
// 't' is the result of popping the top of 'this' stack.
//
// Input:
//   '!isEmpty()'
// Output:
//   't' is the last item pushed on the stack.
//   'this' is the old stack without the top element
// -----
public int pop();

// -----
// t = top()
// 't' is the top of 'this' stack.
//
// Input:
//   '!isEmpty()'
```

```
// Output:  
// 't' is the last item pushed on the stack.  
// 'this' stack remains unchanged.  
// -----  
public int top();  
}
```

We can easily implement a stack by a linked list (see page 49) as shown by the following declaration:

```
public class ListStack implements Stack  
{  
    public Node head;  
  
    public boolean isEmpty()  
    {  
        return head == null;  
    }  
  
    public void push(int t)  
    {  
        Node node = new Node(t, head);  
        head = node;  
    }  
  
    public int pop()  
    {  
        int value = top();  
        head = head.next;  
        return value;  
    }  
  
    public int top()  
    {  
        return head.value;  
    }  
}
```

We may then use the stack as follows:

```
Stack s = new ListStack();  
s.push(3); s.push(4); s.push(5);  
int v0 = s.pop(); System.out.println(v0);  
int v1 = s.pop(); System.out.println(v1);
```

which gives output

```
5
4
```

Example (Parenthesis Matching) We want to write a function that takes a string of parentheses, e.g.

```
((([()]][]))[])
```

and returns true if and only if the closing parentheses match the open parentheses (like in above example). It would return false if a closing parenthesis would not have an opening counterpart (or vice versa) or if it would encounter a mismatch as in

```
((([()]]_[]))[])
```

Our solution uses a stack: whenever we encounter an open parenthesis, we push it on the stack. When we encounter a closing parenthesis, we pop one element from the stack and check whether it has the right shape.

```
public static boolean checkParens(String word)
{
    int n = word.length();
    Stack stack = new ListStack();
    for (int i = 0; i < n; i++)
    {
        char ch = word.charAt(i);
        switch (ch)
        {
            case '(': case '[':
                stack.push(ch);
                break;
            case ')': case ']':
                if (stack.isEmpty()) return false;
                char ch0 = (char)stack.pop();
                if (ch == ')' && ch0 != '(') return false;
                if (ch == ']' && ch0 != '[') return false;
                break;
            default:
                break;
        }
    }
}
```

```

    }
    return stack.isEmpty();
}

```

When we encounter a closing parenthesis, we first check whether the stack is empty. If yes, there is a closing parenthesis that has no matching open parenthesis such that we return false. Otherwise, we pop the top element and check whether it has the right shape. If not, we return false. When we have processed all characters, we return true provided that the stack is empty. If not, there was an open parenthesis without a closing counterpart. When parsing the word `[() []]` the stack will take the following values:

Next Character	Stack Contents
[[
([(
)	[
[[[
]	[
]	

All parentheses have matched and the stack is finally empty. Therefore the function returns true.

The Java standard library provides a class `Stack` implemented on top of class `Vector`; for the class interface, please consult the Java Manual¹.

2.4.3 Queues

A *queue* is a sequence of elements accessed from both ends: on one end (the *tail* or *rear*) new elements are added (“enqueue”), from the other end (the *head* or *front*) elements are removed (“dequeue”). A queue is therefore a *FIFO* data structure (first in, first out): the first element enqueued is also the first element dequeued. This idea is illustrated by Figure 2.10. Figure 2.11 shows the screenshot of an applet embedded into the electronic version of this document; this applet demonstrates the various queue operations.

The abstract datatype “queue of integers” is represented by the following Java interface:

¹<https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/Stack.html>

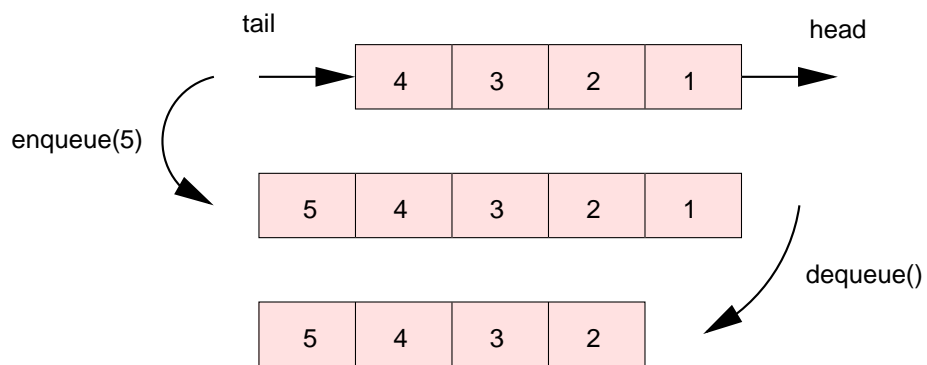


Figure 2.10: Queue Operations

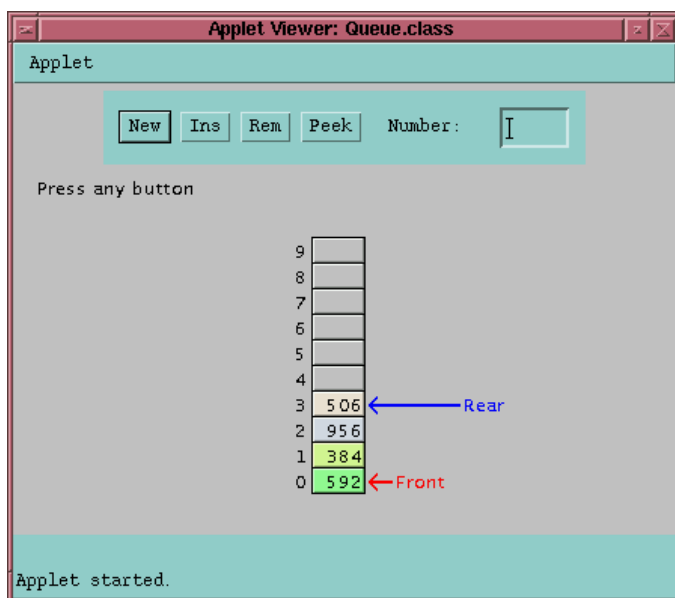


Figure 2.11: Queue Applet

```
public interface Queue
{
    // -----
    // b = isEmpty()
    // 'b' is true exactly if 'this' queue has no elements
    // -----
    public boolean isEmpty();

    // -----
    // enqueue(e)
    // enqueue 'e' to tail of 'this' queue
    //
    // Output:
    // 'this' is the old queue with 't' at the end
    // -----
    public void enqueue(int e);

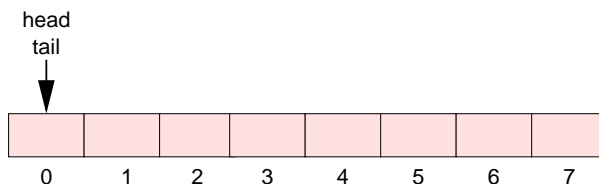
    // -----
    // t = dequeue()
    // 'e' is the result of removing the head of 'this'
    //
    // Input:
    // '!isEmpty()'
    // Output:
    // 't' is the head of 'this' queue.
    // 'this' is the old queue without the head
    // -----
    public int dequeue();

    // -----
    // t = peek()
    // 't' is the head of 'this' queue.
    //
    // Input:
    // '!isEmpty()'
    // Output:
    // 't' is the head of 'this' queue
    // 'this' queue remains unchanged.
    // -----
    public int peek();
}
```

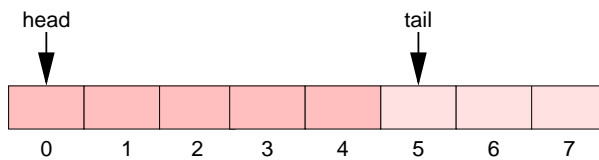
For the ADT queue, there exist two popular implementations.

Implementation as an Array

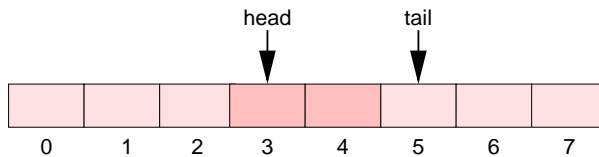
A simple implementation of a queue is by an array. The index *head* refers to the position of the head element of the queue, the index *tail* refers to the first free array slot where new elements can be enqueued. The following diagram illustrates the array in the initial state (*head* equals *tail*):



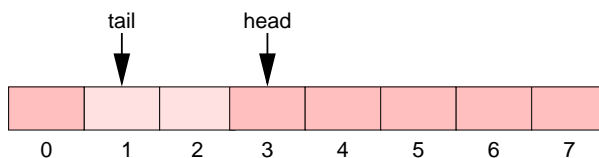
After five elements have been inserted we have the following state:



After two elements have been dequeued we have the state:



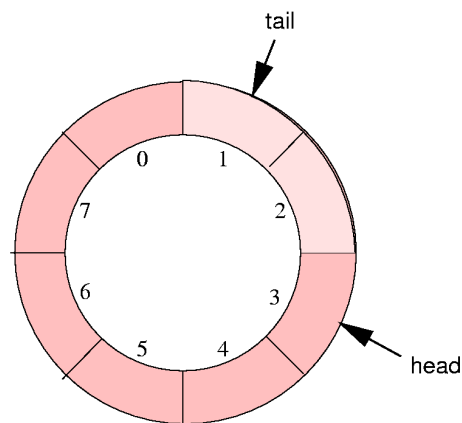
Now we would like to have four more elements inserted. However, there are only three more free slots at the end of the array. What shall we do? The answer is simple: whenever one of the indices *head* or *tail* runs out of the right end of the array it moves back into the array at the left end:



We can achieve this effect by computing the new index as

```
tail = (tail + 1) % n;
```

where n is the length of the array. By this trick we simulate a so called *circular array* as illustrated by the following diagram:



At no time, the queue can hold more than n elements but, as long as this condition is met, we can perform arbitrary many queue operations.

The following Java class implements this idea:

```
public class ArrayQueue implements Queue
{
    public int[] a;

    public int head = 0;
    public int tail = 0;
    public int count = 0;

    public ArrayQueue(int n)
    {
        a = new int[n];
    }

    public int isEmpty()
    {
        return count == 0;
    }

    public void enqueue(int value)
    {
```

```
        if (count == a.length) System.exit(-1);
        count = count+1;
        a[tail] = value;
        tail = (tail+1) % a.length;
    }

    public int dequeue()
    {
        count = count-1;
        int value = a[head];
        head = (head+1) % a.length;
        return value;
    }

    public int peek()
    {
        return a[head];
    }
}
```

In this implementation, we use the counter *count* to keep track of the number of elements stored in the array. Otherwise, there is no possibility to distinguish the full state from the empty state: in both situation *head* equals *tail*. If the array gets full, we simply abort the program.

With this implementation, we can write a program

```
Queue q = new ArrayQueue(8);
q.enqueue(2); q.enqueue(3); q.enqueue(5);
int v0 = s.dequeue(); System.out.println(v0);
int v1 = s.dequeue(); System.out.println(v1);
```

which gives output

```
2
3
```

Cyclic arrays are a popular implementation technique to implement queues, for instance in system software where incoming events have to be buffered until they can be processed in the order in which they have arrived. However, there are also other techniques as demonstrated by the following subsection.

In a real implementation, a larger array should be allocated.

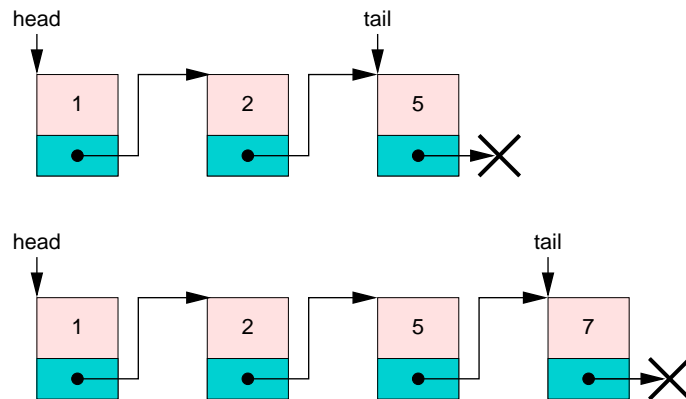


Figure 2.12: List Representation of a Queue

Implementation as a Linked List

In this section, we investigate the use of a linked list as the underlying representation of a queue. Here the main problem is that a list only provides access to its head, while in a queue elements have to be added at the tail. We can overcome this problem by maintaining, in addition to the pointer *head*, a pointer *tail* which references the last element of the list; whenever a new element is to be added, we link it to the last element and update the *tail* pointer. This idea is illustrated in Figure 2.12:

The following Java class implements this idea:

```
public class ListQueue implements Queue
{
    public Node head;
    public Node tail;

    public boolean isEmpty()
    {
        return head == null;
    }

    public void enqueue(int e)
    {
        Node node = new Node(e);
        if (head == null)
            head = node;
    }
}
```

```
        else
            tail.next = node;
        tail = node;
    }

    public int dequeue()
    {
        int value = head.value;
        head = head.next;
        if (head == null) tail = null;
        return value;
    }

    public int peek()
    {
        return head.value;
    }
}
```

In method `enqueue`, we set, if the list is empty, the *head* pointer to the new node. Otherwise, we link the last element with the new node. In both cases, the *tail* pointer is updated to refer to the new node. In `dequeue`, we remove the head element by updating the *head* pointer to point to the next node. If the list becomes empty, we also set the *tail* pointer to `null`.

When comparing both queue implementations, the implementation by linked lists seems a bit simpler and has the advantage that there is no a priori bound on the number of elements. However, the array implementation is a bit faster since it does not require the allocation of a new list cell on every `enqueue` operation; it is therefore preferred if utmost efficiency is required.

Chapter 3

Java with Caffeine

In this chapter, we are going to describe one of the most important aspects of Java that has not yet been discussed: the organization of classes by *inheritance*. This feature extends the previously introduced mechanisms of class declarations and method calls and makes Java a fully *object-oriented* programming language. At the end of this chapter, you essentially know what object-oriented programming is (although the pragmatics of how to write large object-oriented programs is a complex topic that cannot be discussed in these short notes).

3.1 Inheritance

Classes represent collections of uniform objects, i.e., objects with the same structure (fields and methods). However, in reality, things that belong to the same category are rarely that uniform but usually come in different *variants*; often these variants can be hierarchically classified.

Take as an example the kingdom of animals sketched in Figure 3.1. An animal may be for instance a fish, a reptile, or a mammal. A mammal again may be a predator, a rodent, or a hoofed animal. A predator may be a wolf, a bear, or a cat, while a rodent may be a mouse or a rat, and so on.

A wolf and a bear are both predators, and a wolf and a mouse are both mammals. When calling an animal a “mammal”, we highlight its property of breastfeeding its offspring and forget the more distinguishing properties of being a predator or being a rodent. Also in software we would like to group objects to a common class, if they have some common characteristics, even if they differ in some other features.

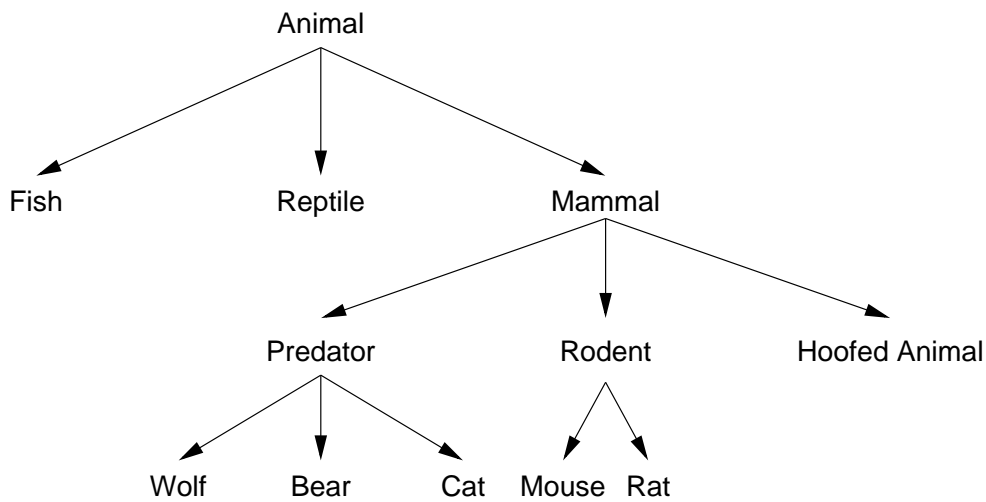


Figure 3.1: A Class Diagram

Similar to above animal classification, Java offers the possibility of categorizing classes in a hierarchical tree structure. If two classes (“predator” and “rodent”) have a common ancestor (“mammal”) in the tree, they share the common properties of this class but differ in other properties. If needed, objects of the different classes may be considered as objects of the ancestor class and treated alike.

3.2 Example

We are going to introduce the basics of inheritance in Java by the following example: Take an Internet shop that offers books as well as CDs. The shop stores the following pieces of information on books:

- article number
- title
- price
- author
- publisher
- ISBN number.

The following is stored on CDs:

- article number
- title
- price
- interpreter
- list of songs.

When a customer orders articles via the Internet, she wants to see a shopping cart with a list of articles including article number, title, and price. When clicking on the article, she should see the remaining information about the article (for books, the author, publisher, and ISBN number; for CDs, the interpreter and the list of songs).

3.3 Superclasses and Subclasses

The envisioned software should provide a class `Article` which offers the common functionality of books and CDs:

```
public class Article
{
    public int number;
    public String title;
    public int price;

    public Article (...) { ... }

    public int getNumber() { ... }
    public String getTitle() { ... }
    public int getPrice() { ... }
}
```

Books and CDs are now special cases of articles; they have the same fields and methods as articles but also provide additional fields and methods. A corresponding Java class `Books` for books is therefore the following one:

```
public class Book extends Article
{
    public String author;
    public String publisher;
    public String ISBN;
```

```
public Book(...) { ... }

public String getAuthor() { ... }
public String getPublisher() { ... }
public String getISBN() { ... }
}
```

The clause `extends` means that `Book` is derived from `Article`, i.e., it *inherits* all fields and methods of `Article` as if they were declared locally within the class. We call `Article` the *superclass* (or *parent class*) of `Book` and `Book` its *subclass* (or *child class*).

In general, we have the following form of class declaration

```
public class Subclass extends Superclass
{
    ...
}
```

If the class implements one more interfaces, its declaration looks like

```
public class Subclass extends Superclass implements Interface, ...
{
    ...
}
```

Please note that while a Java class may implement multiple interfaces, it may only inherit from *one* superclass (*single inheritance*).

3.4 Class Hierarchies

Returning to our example, we can define a subclass `CD` of `Article` which adds other fields and methods.

```
public class CD extends Article
{
    public String interpreter;
    public String[] songs;
}
```

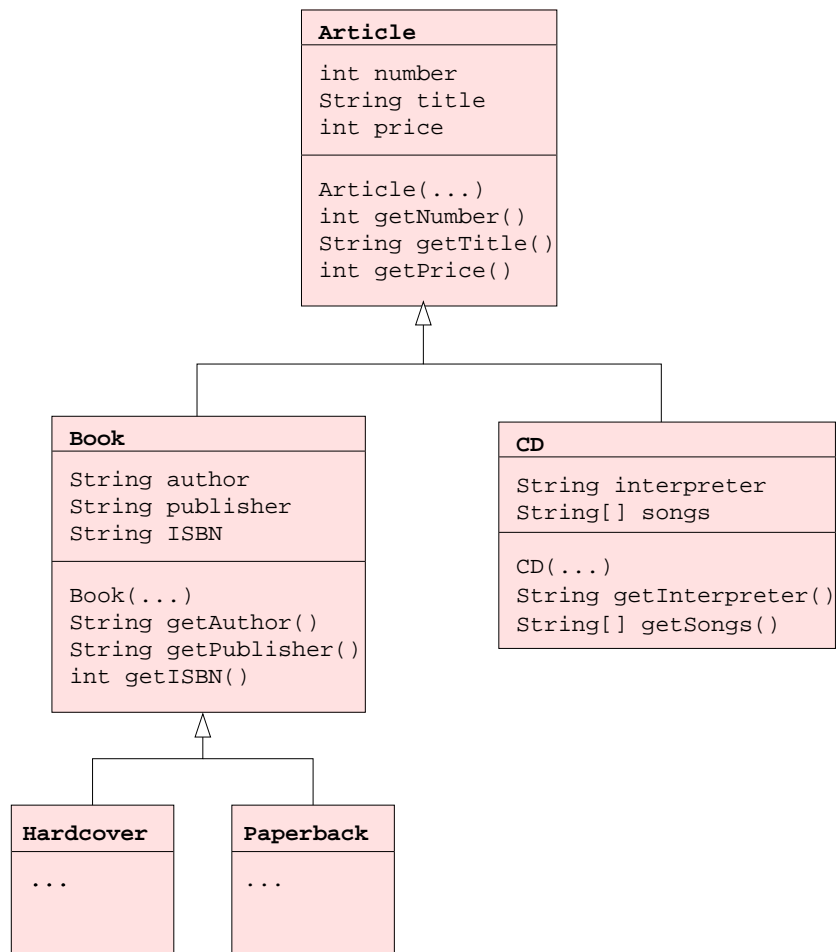


Figure 3.2: A Class Diagram

```

public CD(...) { ... }

public String getInterpreter() { ... }
public String[] getSongs() { ... }
}

```

Further on, we may create subclasses of **Book** like **Hardcover** and **Paperback** and construct a hierarchy as the one shown in Figure 3.2. In this *class diagram* superclasses and subclasses are organized in a tree where arrows are used to denote the relationship from the subclass to a superclass.

If a class is not explicitly derived from another class, it is implicitly considered

as a subclass of the predefined class `Object`. All classes are therefore directly or indirectly derived from `Object`. We will discuss this class in more detail on page 88.

3.5 Interface Hierarchies

Not only a Java class may be derived by inheritance from another class but also a Java interface may be derived by inheritance from another interface:

```
public interface I extends J
{
    ...
}
```

The derived interface *I* inherits all the constants and methods declared in the interface *J*.

Interfaces therefore can form also a tree-like hierarchy as the one for classes shown in the previous section. However, class hierarchies and interface hierarchies are separate and do not overlap (see Figure 3.3) An interface cannot be used to derive a class, and a class cannot be used to derive an interface. The only relationship between classes and interfaces is that a class may implement one or more interfaces.

3.6 Constructors in Derived Classes

As shown in above example, each class has its own constructors, e.g.

```
public class Article
{
    public int number;
    public String title;
    public int price;

    public Article(int n, String t, int p)
    {
        number = n;
        title = t;
        price = p;
    }
}
```

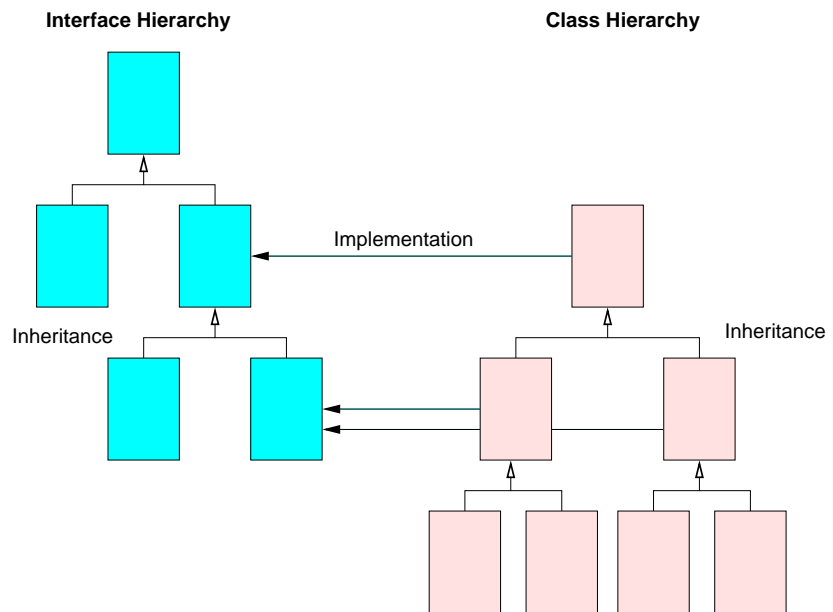


Figure 3.3: Interface and Class Hierarchy

```

}

public class Book extends Article
{
    public String author;
    public String publisher;
    public String ISBN;

    public Book(int n, String t, int p, String a, String pb, String i)
    {
        number = n;
        title = t;
        price = p;
        author = a;
        publisher = pb;
        ISBN = i;
    }
}

```

Please note that class `Book` inherits all the fields of `Article`, therefore its constructor must also initialize these fields in addition to the fields declared locally.

However, above solution is rather clumsy because it duplicates the code of the `Article` constructor. It would be better if the `Book` constructor could call the `Article` constructor.

Java allows a constructor to call a constructor of the base class by a statement

```
super(...);
```

as the *first* statement of the constructor body (like we can write `this(...)` to call another constructor of class `Book`).

Above class can therefore be rewritten as

```
public class Book extends Article
{
    public String author;
    public String publisher;
    public String ISBN;

    public Book(int n, String t, int p, String a, String pb, String i)
    {
        super(n, t, p);
        author = a;
        publisher = pb;
        ISBN = i;
    }
}
```

The constructor now calls the `Article` constructor to initialize the fields declared in `Article` and itself initializes only those fields declared in `Book`.

Above example describes the typical form of the initialization of objects of derived classes. A constructor only cares for the fields declared locally in its class and delegates the initialization of the fields of the superclass to the constructor of this class. If there is no call to a constructor of the superclass, then the compiler automatically inserts a call

```
super();
```

to the default constructor of the superclass. In this case, the superclass must explicitly contain such a constructor without parameters (the implicitly generated default constructor does not suffice).

3.7 Code Sharing

What are the advantages of inheritance? One aspect that has been illustrated above is the reduction in the amount of code that has to be written. A subclass can inherit fields and methods from a superclass without declaring it locally. This aspect is especially important if there may be multiple subclasses that inherit from the same superclass. Rather than defining the common fields and method in each subclass, they are defined once and for all in the superclass. One should therefore obey the following strategy:

Whenever there are two or more classes that share common functionality, this functionality should be put in a separate superclass from which the other classes are derived by inheritance.

In this way, code may be shared among multiple classes, i.e., the duplication of code can be avoided.

3.8 Type Compatibility

Sharing code among multiple classes is only one aspect of inheritance. Another one that is even more important is that a subclass is *compatible* with the superclass. As a general rule:

Every program that is able to work with objects of the superclass can automatically also work with objects of the subclass.

Let us illustrate the consequences of this rule by our example of the Internet shop. We can write a program that works with objects of type `Article`, i.e., it can place them into a shopping cart, print the contents of a shopping cart, and compute the total price of an order without making a difference between the various kinds of article.

Later on, we may implement various concrete articles by deriving subclasses of `Article`, e.g., `Book`, `CD`, or `Video`. Therefore objects of these classes are special variants of articles, and the program can immediately work with them (place them into the shopping cart, extract their price and so on) as if they had existed when the program was written. The program need *not* be modified to work with the new kinds of articles.

Between subclass and superclass there is a *is-relationship* which can be summarized by the following rule:

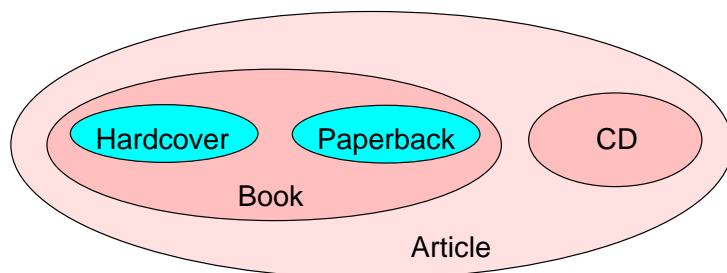


Figure 3.4: The Is-Relationship between Subclasses and Superclasses

Every object of a subclass *is* also an object of the superclass.

This rule (which implies the previously stated rule), can be illustrated by a *set diagram* as shown in Figure 3.4. The set of all objects of type `Article` has as subsets the set of all objects of type `Book` and the set of all objects of type `CD`. In other words, every object of type `Book` or of type `CD` is also an object of type `Article`.

3.9 Object Assignments

Above rule has a technical consequence in Java. A variable/constant whose type is a superclass may refer to an object of the subclass. For instance, since an object of type `Book` is also of type `Article`, we may write:

```
Article a = new Book(...);
```

However, since also an object of type `CD` is of type `Article`, we may change the assignment:

```
a = new CD(...);
```

We therefore don't see a priori whether a variable of type `Article` holds a book or a `CD`. However, this does not matter, because we can refer via this variable only to the fields and methods declared in class `Article` (which are inherited by both `Book` and `CD`).

```
String t = a.getTitle();
int p = a.getPrice();
```

If we want to know whether a variable of type `Article` holds a book, we can check this with the predicate `instanceof` and convert the type of the object by a *type cast* to `Book`.

```
if (a instanceof Book)
{
    Book b = (Book)a;
    ...
}
```

The `instanceof` check ensures that the type cast to `Book` is correct. If the check is omitted and the `Article` variable holds an object of type `CD`, the type cast triggers a runtime error.

Generalizing our discussion, let us assume that we have a subclass `D` of `C`:

```
public class D extends C
{
    ...
}
```

Then we can always assign to a variable of type `C` an object of type `D` (because a `D` object *is* also a `C` object):

```
D d = ...
C c = d;
```

However, we can assign to a variable of type `D` only a variable of type `C` with a type cast *and* after we have checked that the `C` variable actually holds a `D` object:

```
C c = ...;
if (c instanceof D)
{
    D d = (D)c;
    ...
}
```

3.10 Static Types and Dynamic Types

As demonstrated above, an object variable has two kinds of types:

1. A **static type**: this is the type with which the variable has been declared. It determines to which fields and methods of the variable we can refer.

In above example, the static type of variable `a` is `Article`.

2. A **dynamic type**: this is the type of the object to which the variable currently refers. The dynamic type can change with every assignment, but it always is a subtype of the static type. It determines which methods are actually called (see Section 3.13).

In above example, the dynamic type of variable `a` is `Book`.

We call object variables *polymorph* because they can refer to objects of different types (that are subtypes of a common supertype).

3.11 Generic Methods

Since objects variables may have multiple dynamic types, we can also write *generic methods* that operate in the same way on objects of these types.

For instance, we may write a method that prints the information available on an article:

```
public static void printInfo(Article a)
{
    int number = a.getNumber();
    String title = a.getTitle();
    int price = a.getPrice();
    System.out.print("Article '" + title);
    System.out.print("' (" + number + ") costs ");
    System.out.println(price/100 + "." + price%100 + " Euro.");
}
```

We may call this method for both books and CDs:

```
Book book = new Book(...);
CD cd = new CD(...);
printInfo(book);
printInfo(cd);
```

Since `Book` and `CD` are both subclasses of `Article`, it is legal to call `printInfo` with a `Book` object respectively with a `CD` object as an argument. The parameter a of public static type `Article` once receives the dynamic type `Book` and once the dynamic type `CD`.

To have such generic methods that operate on objects of different types is one of the main advantages of an object-oriented language like Java.

3.12 The Object Class

As already stated, all classes are derived ultimately from the `Object` class. If a class declaration omits the `extends` clause, it is derived from the `Object` class by default, i.e., the following two declarations are equivalent.

```
public class Class
{
    ...
}

public class Class extends Object
{
    ...
}
```

Class `Object` is declared in package `java.lang`; it has various methods that are not yet of interest to us.

The most important use of the `Object` class is the use as a parameter type in generic methods such that the method can operate on objects of *any* type.

For instance, we may implement a class `Stack` which can store objects of any object type as follows:

```
public class Stack
{
    public int N = 100;
    public Object[] stack = new Object[N];
    public int n = 0;

    public void push(Object o)
    {
```

```
        if (n == N) resize();
        stack[n] = o;
        n++;
    }

    public Object pop()
    {
        n--;
        return stack[n];
    }

    public void resize() { ... }
}
```

We may use this stack then to store `Integer` objects as in

```
Stack s = new Stack();
s.push(new Integer(2));
s.push(new Integer(3));
Integer i = (Integer)(s.pop());
```

Please note that the result of `pop` is an object, i.e., the caller has to cast it back to `Integer`. If the stack holds objects of multiple types, the caller may use the operator `instanceof` to select among these types:

```
Stack s = new Stack();
s.push(new Integer(2));
s.push("hello");
Object o = s.pop();
if (o instanceof Integer)
{
    Integer i = (Integer)o;
    ...
}
else if (o instanceof String)
{
    String s = (String)o;
    ...
}
```

Various container classes in the Java standard library operate in this fashion on arguments of type `Object`.

3.13 Method Overriding

We can make the static method `printInfo` presented above also a dynamic method of class `Article`:

```
public class Article
{
    public int number;
    public String title;
    public int price;
    ...
    public void printInfo()
    {
        System.out.print("Article '" + title);
        System.out.print( "' (" + number + ") costs ");
        System.out.println(price/100 + "." + price%100 + " Euro.");
    }
}
```

This method is inherited by `Book` and `CD` such that we may print books and CDs as follows:

```
Book book = new Book(...);
CD cd = new CD(...);
book.printInfo();
cd.printInfo();
```

However, this is actually not what we want. A book has more information than stored in class `Article` and we would like to have this additional information printed as well. For this reason, Java gives the possibility to *override* a methods declared in a superclass by a declaration of the same name in a subclass. For instance, we may define

```
public class Book extends Article
{
    public String author;
    public String publisher;
    public String ISBN;
    ...
    public void printInfo()
    {
```

```
        System.out.print("Article '" + title);
        System.out.print("' (" + number + ") costs ");
        System.out.println(price/100 + "." + price%100 + " Euro.");
        System.out.print("This is a book written by " + author);
        System.out.print(" and published by " + publisher);
        System.out.println(" with ISBN number " + ISBN + ".");
    }
}
```

When we now write

```
Book book = new Book(...);
book.printInfo();
```

it is the `printInfo` method of class `Book` which is called and not the corresponding method of class `Article`. The same is true if we write

```
Article book = new Book(...);
book.printInfo();
```

where variable `book` is declared of type `Article`; as mentioned in Section 3.10, it is not the *static* type of the variable (the type `Article` in the variable declaration) but its *dynamic* type (the type `Book` of the object referenced by the variable) that determines which definition of method `printInfo` is used!

3.14 Calling Overridden Methods

The method `printInfo` in `Book` shares a lot of functionality with the corresponding method of the superclass. It would be therefore good if the method in the subclass could invoke the method of the superclass. For this reason, Java has the pseudo-object `super` which refers to the current object (`this`) but considers it as an object of the superclass. We may then write

```
public class Book extends Article
{
    public String author;
    public String publisher;
    public String ISBN;
    ...
    public void printInfo()
```

```
    {
        super.printInfo();
        System.out.print("This is a book written by " + author);
        System.out.print(" and published by " + publisher);
        System.out.println(" with ISBN number " + ISBN + ".");
    }
}
```

The method `printInfo` in class `Book` invokes (with this object) the method `printInfo` of the superclass `Article`. In this fashion, functionality may be gradually extended starting with a simple general definition in the superclass that is appropriately refined in subclasses.

Any other method may use `super` to refer to a method of the superclass which was overridden by a definition in the current class. For instance, we may define

```
public class Book extends Article
{
    ...
    public void printInfo() { ... }
    ...
    public void printArticleInfo()
    {
        super.printInfo();
    }
}
```

The program fragment

```
Book book = new Book(...);
book.printArticleInfo();
```

now invokes the method `printInfo` in class `Article`.

3.15 Shadowing Variables

A child class may not only override methods of the parent class, it may also redefine variables inherited from the parent class, a technique that is called *shadowing a variable*. For instance, we may declare

```
public class Article
{
    public int number;
    public String title;
    public int price;
    ...
}

public class Book extends Article
{
    float price;
    public String author;
    public String publisher;
    public String ISBN;
    ...
}
```

Here the child class `Book` declares a new `float` variable `price` in addition to the `int` variable declared in the parent class. We now have *two* variables of the same name; any reference to `price` within class `Book` refers to the locally declared `float` variable which thus shadows the `int` variable declared in the class `Article`. Nevertheless, we may still refer to the `int` variable using the notation `super.price` (like we may refer to overridden methods).

Having said this, there is no good reason to ever shadow a variable. It makes code difficult to understand and we can always choose a new name for a variable in a child class.

Shadowing variables can and should be avoided.

3.16 Final Methods and Classes

If a method is declared as `final`, it cannot be overridden by a method in a subclass:

```
public final type method(...) { ... }
```

If a class is declared as `final`, no other class may inherit from this class.

```
public final class Class
{ ... }
```

The keyword `final` thus prevents unwanted extensions; furthermore it allows the compiler to generate more efficient code for method calls.

3.17 Abstract Classes

Classes may serve two purposes:

1. as templates for instantiating objects in new calls;
2. as parent classes for other classes by `extends` declarations.

The classes we have seen up to now served both purposes; however sometimes one wishes to declare a class that is only used for the second purpose.

For instance, let us assume that we want to write a generic method `printNice` which prints an object with a nice header line before and after the object contents. However, to be able to print the object contents, the object must provide a string representation of itself. We can express this by the fact that `printNice` operates on objects of type (respectively subtype of) `Printable` which provide via a method `getString` a string representation of themselves:

```
public static void printNice(Printable o)
{
    System.out.println("-- begin object ----- ");
    System.out.println(o.getString());
    System.out.println("-- end object ----- ");
}
```

The class `Printable` can be declared as

```
public abstract class Printable
{
    public abstract String getString();
}
```

The method `getString` is declared with the keyword `abstract` which says that the method does not contain an implementation (i.e., a body) in this class. Since the class `Printable` contains an abstract method, it must be also declared as `abstract`. Such an abstract class does not contain implementation for some of its methods, it cannot be used for instantiating objects in new calls; it can be only used for inheritance by other classes.

If a subclass of `Printable` provides implementations for all the abstract methods of the superclass, it is itself not abstract any more and can be used for instantiation. We may thus define a non-abstract subclass `MyInteger` of `Printable` as follows:

```
public class MyInteger extends Printable
{
    public int i;
    public MyInteger(int i) { this.i = i; }

    public String getString()
    {
        return Integer.toString(i);
    }
}
```

Then we may call

```
MyInteger i = new MyInteger(7);
printNice(i);
```

where `printNice` invokes the `getString` method declared in `MyInteger`. We may also write above example as

```
Printable i = new MyInteger(7);
printNice(i);
```

where we immediately forget that `i` is of type `MyInteger` but treat it as an object of type `Printable`. Thus we cannot create objects of type `Printable` (an abstract class cannot be the dynamic type of a variable) but we can declare variables and parameters of this type (an abstract class can be the public static type of a variable).

3.18 Frameworks

The only purpose of an abstract class is to serve as a placeholder in a class hierarchy in order to define a minimal interface for its subclasses. Methods may freely use abstract classes as types of parameters and variables.

The attentive reader may notice that in this respect abstract classes serve a purpose similar to interfaces. This is basically true but there are two important differences:

1. A class may only inherit from *one* other (abstract) class but it may implement *multiple* interfaces;
2. In an interface, all methods are abstracts; in an abstract class, some methods may be *not* abstract.

As for the second item, we may implement our class `Printable` also as

```
public abstract class Printable
{
    public abstract String getString();

    public void printNice(Printable o)
    {
        System.out.println("-- begin object ----- ");
        System.out.println(getString());
        System.out.println("-- end object ----- ");
    }
}
```

i.e., the class contains `printNice` as a dynamic method. This method invokes the (abstract) method `getString` to determine the actual text representation of the object.

Using this class, we can now declare the subclass `MyInteger` as shown above and call

```
Printable i = new MyInteger(7);
i.printNice();
```

The class `MyInteger` therefore *inherits* the functionality of `printNice` from `Printable` but it *provides* the functionality of `getString`.

An abstract class that provides by non-abstract methods functionality but leaves by abstract methods “gaps” to be filled by subclasses is also called a *framework*. It provides the skeleton functionality of an application but still needs to be customized: a subclass has to fill the skeleton with “flesh” by providing implementations of the abstract methods.

A framework may also give default implementations for the abstract methods such as in

```
public class Printable
{
    public String getString()
    {
        return "This object has no string representation."
    }

    public void printNice(Printable o)
```

```
{
    System.out.println("-- begin object ----- ");
    System.out.println(getString());
    System.out.println("-- end object ----- ");
}
```

Here we provide a body for `getString` (such that the method and the class are not abstract any more). Nevertheless, we want a subclass to override this method by a more specific one such as in the class `MyInteger` shown above.

3.19 Visibility Modifiers

Usually not all the fields and methods of a class are intended for general use, i.e., some of them may be just intended for internal purposes. In Java, we have several *visibility modifiers* that may be used to explicitly restrict the access to classes, fields, and methods. If no modifier (such as `public`) is given, an entity gets “default visibility”; it is then visible only to classes within the current package but not to classes outside the package. We have refrained from this kind of visibility (which is of questionable value anyway) but always used the keyword `public`; it is good practice to provide all entities with an explicit modifier to clearly indicate its visibility. In the following, we will investigate the meaning of other visibility modifiers as alternatives to `public`.

3.19.1 Class Visibility

If a class is declared with the keyword `public` such as in

```
public class Class
{
    ...
}
```

it becomes visible also outside the current package. If you organize your classes in packages, you have to use this identifier to denote those classes that are of general use.

3.19.2 Field and Method Visibility

For fields and methods, we have three visibility modifiers: `public`, `private`, `protected`.

- `public` indicates that the field/method can be used without restriction by anyone.
- `private` indicates that the field/method can be used only within the current class.
- `protected` indicates that the field/method can be used by the current class and by all classes that reside within the current package or that (directly or indirectly) inherit from the current class.

3.19.3 Rules of Thumb

The details of field visibility can become quite confusing; here are some simple rule of thumbs that are typically used:

- Make all classes `public` (the classes can be used from any package) unless you want to restrict access to the current package.
- Make all class fields `private` (class fields should be encapsulated by a class).
- Make all methods that are part of the class specification `public` (they provide functionality of general interest).
- Make all other methods `private`.

Thus a typical class is structured as follows:

```
// class can be used by any other class
public class Class
{
    // the private fields
    private type field;

    // the public constructors
    public Class (...) { ... }
```

```

// the public methods for the user
public type method (...) { ... }

// the private methods for the class
private type method (...) { ... }
}

```

Only in special situations one might want to use `protected` to expose some basically private methods to other classes within the current package.

Example Let us consider the class `Applet` that may be used to implement HTML-embedded programs with graphical output.

```

public class Name extends Applet
{
    public void paint(Graphics graph)
    {
        paint operations on graph
    }
}

```

Any applet is a subclass of class `Applet` provided in package `java.applet`. This class is part of the following subclass hierarchy

```

java.lang.Object
|
+--java.awt.Component
    |
    +--java.awt.Container
        |
        +--java.awt.Panel
            |
            +--java.applet.Applet

```

i.e., it itself inherits a lot of functionality from other classes. For instance, the class `Component` is implemented as

```

public abstract class Component implements ...
{

```

```
...
public void update(Graphics g)
{
    if (...)
    {
        g.setColor(getBackground());
        g.fillRect(0, 0, width, height);
        g.setColor(getForeground());
    }
    paint(g);
}

public void paint(Graphics g) { }
```

The class is declared as abstract, i.e., it is not intended for instantiation. It provides an `update` method, which clears the applet window and then invokes the `paint` method to draw the window contents. The `update` method is invoked by the browser whenever the window content is to be redrawn.

The method `paint` itself is declared with an empty body, i.e., it is expected that a subclass should override it with a corresponding functionality (it would have been better style to declare this method as abstract).

Chapter 4

Java in Pots

In this chapter, we will discuss the Java Collections Framework, a part of the Java library that in a uniform way supports the handling of various types of containers. Since this framework, which is heavily used in other libraries and many applications, is based on “generics”, i.e., entities that are parameterized over types, we will start with a presentation of this fundamental concept.

4.1 Generics

A *generic* is a type (class or interface) which is parameterized over other types (we will later also consider generic methods). Take for instance the following definition of a generic class:

```
// T is a type parameter
public class Box<T>
{
    private T t;
    public void set(T t) { this.t = t; }
    public T get() { return t; }
}
```

Objects of this generic class `Box<T>` are “boxes” that encapsulate values of some type T . The type parameter T may appear in the definition of the class everywhere a normal type is expected (with some restrictions, which we will discuss later).

To use a generic type in a program, it has to be instantiated by some concrete type for parameter T , e.g.

```
Box<String> sbox = new Box<String>();
```

creates a box that may hold `String` values. The box may be applied e.g. as

```
sbox.set("text");  
String s = sbox.get();
```

because also all occurrences of T in the declarations of methods `set` and `get` have been replaced by `String`.

The instantiation type of a generic type can be an arbitrary class, but not a primitive type; thus T may be instantiated with `Integer` but not with `int`. However, since Java performs automatic boxing/unboxing, the difference is not so big, as the following code snippet demonstrates:

```
Box<Integer> ibox = new Box<Integer>();  
ibox.set(5);          // autoboxing:   ibox.set(new Integer(5));  
int i = ibox.get(); // autounboxing: int i = ibox.get().intValue();
```

Also interfaces may be generic and generics may have multiple parameters. For instance, the following interface `Pair<A,B>` of a type of tuples with a first component of type A and a second component of type B is implemented by a corresponding class `PairClass<A,B>`:

```
public interface Pair<A,B>  
{  
    public A getFirst();  
    public B getSecond();  
}  
  
public class PairClass<A,B> implements Pair<A,B>  
{  
    private A first;  
    private B second;  
  
    public PairClass(A first, B second)  
    { this.first = first; this.second = second; }  
  
    public A getFirst() { return first; }  
    public B getSecond() { return second; }  
}
```

Then we may execute the following piece of code:

```
Pair<String,Integer> p = new PairClass<String,Integer>("key", 7);
String s = p.getFirst();
int i = p.getSecond(); // autounboxing
```

While there may exist many different instantiations of the same generic type, e.g. `Box<String>` or `Box<Integer>`, at runtime there exists only a single type `Box`, the *raw type* of the generic type `Box<T>`. In Java, type parameters are only used for compile-time type checking, at runtime they are replaced by the type `Object`. Thus e.g. the runtime representation of the generic type `Box<T>` is the same as that of the non-generic type

```
public class Box
{
    private Object t;
    public void set(Object t) { this.t = t; }
    public Object get() { return t; }
}
```

This implementation technique called “type erasure” imposes a couple of restrictions on generic types in Java, e.g. that generic types cannot be instantiated with primitive types, but also that it is not possible to use type parameters in object creations such as

```
public class Generic<T>
{
    public void method() { T x = new T(); } // illegal
}
```

or that it is illegal to create arrays of parameterized types:

```
Box<Integer>[] array = new Box<Integer>[10]; // illegal
```

However, rather than creating arrays of parameterized types, we may create all kinds of collections of parameterized types supported by the “Collection Framework” which we are now going to introduce.

4.2 The Collections Framework

The Java Collections Framework is a library for representing and manipulating various kinds of collections in a uniform way; a collection is a container that groups multiple elements into a single unit. The framework consists of the following entities:

- **Interfaces:** these are abstract data types that denote the collections.
- **Implementations:** these are classes that provide concrete implementations for the collection interfaces.
- **Algorithms:** these are methods that perform useful computations on collections independently of their concrete representation.

Since many parts of the Java library make use of the collection framework, it is a good idea to base new code also on this framework to make it interoperable with the library and thus reduce one's own programming effort.

The collection framework is part of the package `java.util`, all interfaces and classes that we are now going to discuss are contained in this package. There are two fundamental sets of interfaces and associated classes:

- **Collections:** these are groups of objects called “elements” of some type E . They are described by generic interfaces which are derived from the interface `Collection<E>` and represented by generic classes that implement these interfaces.
- **Maps:** these are objects that map keys of some type K to values of some type V . They are described by generic interfaces which are derived from the interface `Map<K, V>` and implemented by generic classes that implement these interfaces.

We are now going to discuss these two parts of the framework. The presentation is necessarily sketchy; for details, refer to the Java API documentation.

4.3 Collections

A collection is a group of “elements” of some type E . It provides the interface

```
public interface Collection<E> extends Iterable<E>
{
    boolean add(E e);                // add e to this collection
    boolean addAll(Collection<E> c); // add all elements of c
    void clear();                    // make collection empty
    boolean contains(Object o);      // is o in this collection?
    boolean containsAll(Collection<E> c); // are all elems of c in this?
    boolean isEmpty();               // is this collection empty?
```

```
boolean remove(Object o);          // remove o from this collection
boolean removeAll(Collection<E> c); // remove all elems of c
int size();                        // number of elements
...
}
```

by which in particular elements may be added to respectively removed from the collection. Since `Collection<E>` extends the interface `Iterable<E>`, every collection is iterable, i.e., can be used in a for-each statement:

```
Collection<String> c = ...
for (String e : c)
    System.out.println(e);
```

If a collection has a well-defined order in which its elements occur, the collection implements this more special interface:

```
public interface SequencedCollection<E> extends Collection<E>
{
    void addFirst(E e); // add e as first element to this collection
    void addLast(E e); // add e as last element to this collection
    E getFirst();      // get first element of this collection
    E getLast();       // get last element of this collection
    E removeFirst();  // remove first element from this collection
    E removeLast();   // remove last element from this collection
    SequencedCollection<E> reversed();
                        // get reverse-ordered view of this collection
}
```

This interface additionally allows to add/get/remove elements from both “ends” of the collection and provides a view on the reversed version of this collection. This view is not itself a new collection but is backed by the original one; to really construct a new collection, an appropriate constructor has to be called on this view. For example, the following code snippet constructs a new collection from such a reversed view by application of the constructor `ArrayList` (which will be subsequently explained):

```
SequencedCollection<String> seq = ...
ArrayList<String> list = new ArrayList<String>(seq.reversed());
```

We are now going to investigate the most important kinds of collections and describe their constructions.

4.3.1 Lists

A list is a sequence of elements of some type E with the following interface:

```
public interface List<E> extends SeencedCollection<E>
{
    E get(int index);           // get element at index >= 0
    E set(int index, E element); // set element at index >= 0
    void add(int index, E element); // insert element at index >= 0
    int indexOf(Object o);      // index of first occurrence of o (or -1)
    int lastIndexOf(Object o); // index of last occurrence of o (or -1)
    List<E> subList(int from, int to); // sublist view of range [from,to[
    ...
}
```

In addition to the sequenced collection operations, a list thus supports the operations `get` and `set` that allow to access respectively update elements by indices. Thus an object of type `List<E>` can be used similarly to an array of type `E[]`. Furthermore, the operation `add` inserts an element at a given position shifting all subsequent elements by one; operations `indexOf` and `lastIndexOf` return the position of the first respectively last occurrence of an element in the list (-1, if there is no such occurrence).

The operation `subList` returns a view of the portion of the list from index *from* (inclusive) to index *to* (exclusive). This view is not a new list but backed by the original list; all changes in the return value are thus reflected in the original list. For instance, the snippet

```
list.subList(from, to).clear();
```

removes from *list* the range of elements starting at index *from* and ending before index *to*.

Furthermore, the class `Arrays` provides an operation `asList(a)` that creates a view of an array as a list (backed by the array) that may be e.g., used to create a new list with the subsequently introduced classes:

```
String[] a = new String[...];
List<String> l = ArrayList<String>(Arrays.asList(a));
```

There are various classes that implement these interfaces; two widely used ones are the following:

```
// implementation by an array
public class ArrayList<E> ... implements List<E>, ...
{
    public ArrayList();
    public ArrayList(Collection<E> c);
    ...
}
// implementation by a linked list
public class LinkedList<E> ... implements List<E>, ...
{
    public LinkedList();
    public LinkedList(Collection<E> c);
    ...
}
```

`ArrayList<E>` implements a list by an array that is initially empty and grows when new elements are added (similar to the already introduced class `Vector<E>`). Here both operations `get` and `set` are executed in constant time (independently of the index).

`LinkedList<E>` implements a list by a sequence of linked nodes. While it supports the same operations `get` and `set` as `ArrayList<E>`, the execution time of these operations is linear in the index.

Both implementations support a constructor that takes an arbitrary collection *c* and puts its element into the list (in the order in which they are returned by iterating over *c*); thus any collection can be converted to a list by e.g. executing

```
Collection<String> c = ...
List<String> list = new ArrayList<String>(c);
```

4.3.2 Sets

A set is a collection of elements of some type *E* where no element may occur more than once and where the order of elements is arbitrary (two sets are identical, if they have the same elements in arbitrary order). The interface

```
public interface Set<E> extends Collection<E> { ... }
```

is essentially the same as `Collection<E>`. The more special interface

```
public interface SequencedSet<E> extends SequencedCollection<E>, Set<E> { ... }
```

describes sets whose elements occur in some well-defined order; these sets provide the additional operations of sequenced collections.

Three important implementations of this interface are

```
// implementation by a hash table
public class HashSet<E> ... implements Set<E>, ...
{
    public HashSet();
    public HashSet(Collection<E> c);
    ...
}
// implementation by a tree
public class TreeSet<E> ... implements SequencedSet<E>, ...
{
    public TreeSet();
    public TreeSet(Collection<E> c);
    ...
}
// implementation by a hash table and a linked list
public class LinkedHashSet<E> ... implements SequencedSet<E>, ...
{
    public LinkedHashSet();
    public LinkedHashSet(Collection<E> c);
    ...
}
```

whose constructors allow to create empty sets respectively convert existing collections to sets (removing any duplicates). The specifics of the implementations are as follows:

- `HashSet<E>` implements a set by a hash table, i.e., an array that stores every element at that positions that is computed from the element by application of a hash function; this implementation is very fast but the order of elements when iterating over the set is random.

For this, type *E* must override the definition of method `int hashCode()` inherited from class `Object` (the hash function) by a definition that returns the same integer for elements considered as “equal”.

- `TreeSet<E>` implements a set by a binary tree, a dynamic data structure that generalizes linked lists in that every node has two successor nodes; this implementation is slower but iterating over such a set provides the elements in a natural order, e.g., integers are returned in increasing order.

For this, type E must implement the `Comparable` interface, i.e., provide a method `int compareTo(E x)` which returns a negative integer, zero, or a positive integer, as this object is less than, equal to, or greater than x .

- `LinkedHashSet<E>` implements a set, in addition to a hash table, by a linked list where new elements are added at the back; iterating over such a set returns the elements in the order in which they were inserted.

Like for `HashSet<E>`, type E must override the definition of method `int hashCode()` appropriately.

Thus the code snippet

```
Collection<String> c = ...
Set<String> set = new TreeSet<String>(c);
for (String s : set) System.out.println(s);
```

prints all elements of c where (by conversion to a set of type `TreeSet<String>`) every element is printed exactly once and the order in which the elements are printed is the natural lexicographic string order.

4.3.3 Queues and Deques

As already discussed, a queue is a sequence of elements with two ends: new elements are added to the “tail” of the queue while elements are removed from its “head”; thus elements are retrieved from a queue in the order in which they have been added. A deque (speak: “deck”) is a double ended queue where new elements can be also added to the head and removed from its tail.

The corresponding core interfaces are:

```
public interface Queue<E> extends Collection<E>
{
    public E remove(); // retrieve and remove head
    public E element(); // retrieve head without removing it
    ...
}
public interface Deque<E> extends Queue<E>, SeencedCollection<E>
{
    ...
}
```

The method `add(E e)` inherited from `Collection<E>` adds an element e to the tail of a queue; the new method `remove()` returns the element from the head of the queue while `element()` returns this element without removing it. A deque also supports the sequenced collection operations which symmetrically operate on both ends.

There are various classes that implement these interfaces; two widely used ones that implements deques (and thus also queues) are

```
// implementation by an array
public class ArrayDeque<E> ... implements Deque<E>, ...
{
    public ArrayDeque();
    public ArrayDeque(Collection<E> c);
    ...
}
// implementation by a linked list
public class LinkedList<E> ... implements List<E>, Deque<E>, ...
{
    public LinkedList();
    public LinkedList(Collection<E> c);
    ...
}
```

Thus the already discussed class `LinkedList<E>` implements both lists and deques (and queues).

As an example, take the following code snippet:

```
int n = ... ;
Queue<Integer> queue = new LinkedList<Integer>();
for (int i = 0; i < n; i++)
    queue.add(i);
while (!queue.isEmpty())
{
    int i = queue.remove();
    System.out.println(i);
}
```

This program adds n integers $0, \dots, n - 1$ to a queue and prints them out in the order in which they were inserted.

4.4 Maps

A map is an object that maps keys of some type K to values of some type V . Each key is mapped to at most one value, i.e., a map cannot contain multiple values with the same key. The corresponding interface is:

```
public interface Map<K,V>
{
    // maps key to value, returns previous value (or null, if none)
    public V put(K key, V value);

    // returns value to which key is mapped (null, if none)
    public V get(K key);
    ...

    // set of keys, collection of values, set of key/value pairs
    public Set<K> keySet();
    public Collection<V> values();
    public Set<Entry<K,V>> entrySet();

    // the type of a key/value pair
    public static interface Entry<K,V>
    {
        public K getKey();
        public V getValue();
    }
}
```

The methods `put` and `get` extend a map by a new value for a given key respectively return the value to which a key is mapped. The methods `keySet`, `values`, and `entrySet` return collections of the keys, values, respectively key/value pairs stored in a map; each pair is represented by a value of type `Map.Entry<K, V>`.

If the entries of a map occur in some well-defined order, the map has the following more special interface:

```
public interface SequencedMap<K,V> extends Map<K,V>
{
    ...
    // *sequenced* set of keys, collection of values, set of key/value pairs
    public SequencedSet<K> sequencedKeySet();
    public SequencedCollection<V> sequencedValues();
    public SequencedSet<Entry<K,V>> sequencedEntrySet();
}
```

```

    // get reverse-ordered view of this map
    SequencedMap<K,V> reversed();
}

```

Three important implementations of these interfaces are

```

// implementation by a hash table
public class HashMap<K,V> ... implements Map<K,V>, ...
{
    public HashMap();
    public HashMap(Map<K,V> m);
    ...
}
// implementation by a tree
public class TreeMap<K,V> ... implements SequencedMap<K,V>, ...
{
    public TreeMap();
    public TreeMap(Map<K,V> m);
    ...
}
// implementation by a hash table and a linked list
public class LinkedHashMap<K,V> ... implements SequencedMap<K,V>, ...
{
    public LinkedHashMap();
    public LinkedHashMap(Map<K,V> m);
    ...
}

```

whose constructors allow to create empty maps respectively duplicate existing maps. The specifics of the implementations are for as for the already discussed classes `HashSet<E>`, `TreeSet<E>`, and `LinkedHashSet<E>` with respect to the implementation and resulting time complexity of the map operations and with respect to the order in which keys, values, respectively entries are returned by the corresponding map operations.

As an example, take the following code fragment:

```

// maps words to number of occurrences in their text
// map entries are returned in lexicographical order
Map<String,Integer> map = new TreeMap<String,Integer>();

// reads words from individual lines and puts them in map

```

```
while (true)
{
    String word = Input.readString();
    if (!Input.isOkay()) break;
    Integer count = map.get(word);
    if (count == null)
        map.put(word, 1);
    else
        map.put(word, count+1);
}

// prints words and their occurrences in lexicographic order
for (Map.Entry<String,Integer> entry : map.entrySet())
    System.out.println(entry.getKey() + ":" + entry.getValue());

... // to be continued
```

4.5 Algorithms

The interface `Collection<E>` contains a method `toArray` that allows to convert any collection to an array:

```
public interface Collection<E> extends Iterable<E>
{
    ...
    <T> T[] toArray(T[] a); // copy this collection to a
}
```

This is an example of a *generic method* which is parameterized by a type parameter `<T>`; it can be thus applied for every instance of `T`, as demonstrated by the following piece of code where `T` is instantiated by `String`:

```
Collection<String> c = ...
String[] a = c.toArray(new String[c.size()]);
```

After the execution of this code, array `a` holds the same elements as collection `c`. In combination with the already discussed operation `Arrays.asList`, this method provides a bridge between Java arrays and collections, by supporting conversions in both directions. The reason that `toArray` is parameterized over a type parameter `T` (rather than the type parameter `E` of the generic interface) is that `T` may be

also a superclass of E such that `toArray` returns an array whose base type is this superclass.

More typically, however, generic methods are methods that are declared in a non-generic class. For instance, the class `java.util.Collections` provides various generic static methods that operate on various kinds of collections, such as:

```
// adds elements from array a to collection c
public static <T> boolean addAll(Collection<T> c, T[] a);
```

Also this method can be applied for every instance of T , as demonstrated by the following piece of code where T is instantiated by `String`:

```
String[] a = new String[...];
...
List<String> l = new LinkedList<String>();
l.add("word");
Collections.addAll(l, a);
```

After the execution of this code, list l holds first the string “word” and then all elements of array a .

The class `java.util.Collections` contains generic static methods for various of the generic interface types of the Java Collections framework, for example the following methods operating on values of type `List<T>`:

```
// reverses the order of the elements in the list
public static <T> void reverse(List<T> list);

// replaces all elements of the list by the denoted value
public static <T> void fill(List<T> list, T value);

// copies all elements from src into dest replacing original elements
// if dest is longer than src, the remaining elements remain unchanged
public static <T> void fill(List<T> dest, List<T> src);

// swaps in list elements at position i and j
public static <T> void swap(List<T> list, int i, int j);

// returns position of key in list (result is negative, if none)
// the list must be sorted in ascending order according to natural order
public static <T> int binarySearch(List<T> list, T key);

// returns position of key in list T (result is negative, if none)
```

```
// the list must be sorted in ascending order according comparator c
public static <T> int binarySearch(List<T> list, T key, Comparator<T> c);

// sorts list in ascending order according to the natural order of T
public static <T> void sort(List<T> list);

// sorts list in ascending order according to comparator c
public static <T> void sort(List<T> list, Comparator<T> c);
```

The operations `binarySearch` and `sort` for searching in a sorted list respectively sorting a list optionally take a *comparator* `c` which describes the order of the elements of the list by the following interface:

```
public interface Comparator<T>
{
    // returns -1, 0, 1, if o1 is less than, equal, or greater than o2
    public int compare(T o1, T o2);
}
```

Given these concepts, we can now continue the example of the previous section by printing the words of the text in the descending order of the number of their occurrences. For this purpose, we define the class

```
public class GreaterCounter
    implements Comparator<Map.Entry<String,Integer>>
{
    // returns -1, if entry1 has a greater counter than entry2
    public int compare(Map.Entry<String,Integer> entry1,
        Map.Entry<String,Integer> entry2)
    {
        Integer value1 = entry1.getValue();
        Integer value2 = entry2.getValue();
        return value2.compareTo(value1);
    }
}
```

An object of this class represents a comparator that orders word/counter entries in the descending order of their counters. Then we can solve our problem by the following piece of code:

```
... // continued from last section

// create duplicate of entry set as a list
```

```
List<Map.Entry<String,Integer>> entries =
    new ArrayList<Map.Entry<String,Integer>>(map.entrySet());

// sort list in descending order of the word counters
Collections.sort(entries, new GreaterCounter());

// print words in descending order of the number of their occurrences
for (Map.Entry<String,Integer> entry : entries)
    System.out.println(entry.getKey() + ":" + entry.getValue());
```

Thus with the help of Java collections and algorithms, problems can be quickly solved whose manual coding would require substantially more effort.

Chapter 5

Spilt Java

Programs have to handle *errors* by recovering from them or, if this is not possible, by terminating in a graceful way. In this chapter, we will show how the explicit checking of error codes returned by a function can be replaced by the much more convenient handling of *exceptions* raised by a function. We will conclude by demonstrating how the class `Input` that provides the input facilities used so far has been implemented with the help of exceptions.

5.1 Error Codes

A method is called and encounters a problem during its execution. Rather than terminating the program, the method wants to inform the caller about this problem and thus to delegate the decision how to handle the error. One possibility is to let the method return a special value, an *error code*, that describes whether the method has successfully completed its operations, or, if not, which error has occurred. For instance, we may define a set of error codes

```
final int OKAY = 0;           // result was okay
final int IOERROR = 1;       // input/output error
final int ARITHERROR = 2;    // arithmetic error
```

If our original method call was

```
m(...);
```

we now have to write

```
int result = m(...);
if (result != OKAY)
{
    ... // error handling
}
else
{
    ... // normal operation
}
```

If we have three methods of this kind that were originally called in sequence as

```
m(...);
n(...);
o(...);
```

we now have to write

```
int result1 = m(...);
if (result1 != OKAY)
{
    ... // error handling
}
else
{
    int result2 = n(...);
    if (result2 != OKAY)
    {
        ... // error handling
    }
    else
    {
        int result3 = o(...);
        if (result3 != OKAY)
        {
            ... // error handling
        }
        else
        {
            ... // normal operation
        }
    }
}
```

The program gets a little bit simpler, if we can return after the error handling to the caller:

```
int result1 = m(...);
if (result1 != OKAY)
{
    ... // error handling
    return;
}
int result2 = n(...);
if (result2 != OKAY)
{
    ... // error handling
    return;
}
int result3 = o(...);
if (result3 != OKAY)
{
    ... // error handling
    return;
}
```

Nevertheless, the textual overhead of error checking is high.

Another problem arises, if the method that may trigger the error is actually a function such that the return value is already in use. In this case, the method may set a global result variable to denote the success of its operation:

```
public static int error;
public static int m(...) { ... }

public static void main(...)
{
    int r = m(...);
    if (error != OKAY)
    {
        ... // error handling
    }
    else
    {
        ... // normal operation
    }
    ...
}
```

However, this is not a very good style.

If the domain of the function result includes some values that are not returned by the function in normal situations, we may use some of these values as error values, e.g. a negative `int` value if the normal result is a non-negative `int` value or `null` if the result is an object pointer.

```
Object r = m(...);
if (r == null)
{
    ... // error handling
}
else
{
    ... // normal operation, use r
}
```

If this is not possible, the result domain has to be extended to a pair of normal value and error code. For instance, we may define a class

```
public class Result
{
    public int value;
    public int error;
}
```

such that we can call the function as follows:

```
Result r = m(...);
if (r.error != OKAY)
{
    ... // error handling
}
else
{
    int i = r.value; // normal value
    ...
}
```

However, this requires a change in the interface of the function.

A major problem with all these solutions is that they tend to yield ugly code with a lot of error checks such that the flow of control for the normal case can be hardly seen. This is the problem that is solved by *exceptions*.

5.2 Exceptions

The Java mechanism of *exceptions* is based on three concepts:

1. **A protected code block:** a code block can be protected from the occurrence of errors. If an error occurs during the execution of the code block, the error will be handled.
2. **An exception handler:** an *exception handler* is a code block that handles a particular error. Each protected code block has one or more exception handlers that cover errors that may occur in the block.
3. **An exception:** an error is signaled by triggering an exception which is an object that identifies the error. The exception is forwarded to an exception handler which deals with the error. After the exception has been handled, execution proceeds after the protected code block.

The Java construct for triggering an exception is the `throw` statement which is typically invoked as

```
throw new ExceptionClass(...)
```

This construct triggers an exception, i.e., it constructs an object of type *ExceptionClass* and aborts the normal execution. The exception is forwarded to the handler which is currently in charge of this exception. The class *ExceptionClass* must be a subclass of the Java standard class *Exception*.

The Java language construct for handling exceptions is the `try` statement which looks in its minimal form as follows:

```
try
{
    // protected code block
    ...
}
catch (ExceptionClass e)
{
    // exception handler, takes e as parameter
    ...
}
```

If an exception is thrown during the execution the protected code block, the execution of the protected code block is aborted. If the exception is of type *ExceptionClass*, it is handled by the exception handler to which the exception object is passed as a parameter. After the execution of the handler, normal execution continues with the first statement after the whole construct. If the exception is of a different type, the exception must be handled by another handler (see later).

Any method that may throw an exception has to be declared with a `throws` clause as in

```
public type method(...) throws ExceptionClass, ...
{
    ...
}
```

i.e., all the types of exceptions that the method may throw have to be listed in the method header.

5.3 Example

We will demonstrate the use of exceptions by protecting an input operation. First we define an appropriate exception class as

```
public class InputException extends Exception
{
    private String msg;

    public InputException(String msg)
    {
        super(msg);
    }
}
```

An object of this class stores in a string a description of the error that has occurred (we inherit the object representation of the base class).

We may now write a program function

```
public static public int readInt() throws InputException
{
    int i = Input.readInt();
    if (Input.hasEnded()) return -1;
}
```

```
    if (!Input.isOkay()) throw new InputException("some error");
    return i;
}
```

This program function reads an `int` value from the input stream and returns it as the result. However, if an input error has occurred, the method throws an `InputException` which contains the information whether the file has ended or another error has occurred.

We may now use the function in the following code block:

```
try
{
    int i = readInt(); if (Input.hasEnded()) return;
    int j = readInt(); if (Input.hasEnded()) return;
    int k = readInt(); if (Input.hasEnded()) return;
    System.out.println(i + " " + j + " " + k);
}
catch(InputException e)
{
    System.out.println("Input error: " + e.getMessage());
}
System.out.println("done.");
```

In normal execution (i.e., if no input error occurs), three integers *i*, *j*, and *k* are read and printed, i.e., the output may look like:

```
2 3 5
done.
```

Now let us assume that during the reading of the second integer an error has occurred. The second invocation of `readInt` throws an `InputException` which is forwarded to the handler which prints the content of the exception object `e` which describes what kind of error has occurred. The output of the program therefore is:

```
Input error: some error
done.
```

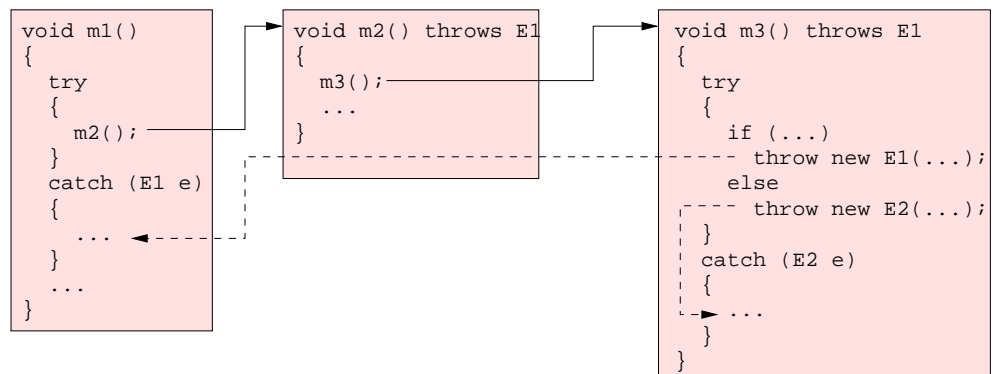


Figure 5.1: Searching for an Exception Handler

5.4 Handling Exceptions

In general, a try statement may have arbitrarily many exception handlers:

```

try
{
  ... // protected code block
}
catch (ExceptionClass1 e1)
{
  ... // exception handler 1
}
catch (ExceptionClass2 e2)
{
  ... // exception handler 1
}
...

```

If in the protected code block an exception is thrown, the corresponding exception handler is selected and executed. If there is no appropriate exception handler declared, the exception is propagated to the caller of the current method. For a more detailed explanation of this question, see Figure 5.1.

Here method `m1` calls in a protected code block `m2` which in turn calls `m3`. In the protected code block of `m3`, exceptions of type `E1` and of type `E2` may be thrown. Exceptions of type `E2` are caught by the handler in `m3` but exceptions of type `E1`

are propagated to the caller `m2`. `m2` however does not handle any exceptions; the exception is therefore propagated to `m1` where it is ultimately caught by a handler.

This figure also demonstrates another important principle: if a method `m` calls another method `n` that may throw an exception, `m` has two possibilities:

1. `m` catches the exception by a corresponding handler, i.e., it must call `n` within a `try` statement,
2. `m` propagates the exception to its caller, i.e., it must declare the exception type in its method header.

In above example, the body of `m3` may throw exceptions of type `E1` and `E2`; it only handles `E2` and must therefore declare `E1` in its header. Method `m2` calls `m1` which may throw exceptions of type `E1`; since `m2` does not handle this exception, it must declare `E1` in its header. Method `m1` calls `m2` which may throw exceptions of type `E1`; `m1` handles this type of exception and must therefore not declare any exception type in its header.

If the main method throws an exception, this exception is caught by the JVM interpreter.

We will later see that *runtime exceptions* are exempted from this rule.

5.5 Catching Multiple Exception Types

If multiple types of exceptions have a common supertype, it is not necessary to have an individual handler for each of these exceptions. Rather there may be a generic handler which catches all instances of this supertype. Since every exception is a subtype of `Exception`, we may thus catch *all* kinds of exceptions by a single handler:

```
try
{
    // may raise multiple kinds of exceptions
    ...
}
catch(Exception e)
{
    // catches all possible kinds of exceptions
    ...
}
```

This is useful if all different exceptions are handled by the same recovery mechanism, e.g., by printing an error message and returning a special value to the caller of the method.

If a try block has multiple exception handlers, the handlers are tried in sequence when an exception is thrown. We may therefore catch some exceptions by more specific handlers and have a “default handler” at the end catching all other exceptions:

```
try
{
    // may raise multiple kinds of exceptions
    ...
}
catch(ExceptionClass1 e)
{
    // catches exception type 1
    ...
}
catch(Exception e)
{
    // catches all other kinds of exceptions
    ...
}
```

5.6 The finally Clause

A try statement may be terminated by a finally clause:

```
try
{
    ... // protected code block
}
catch (Exception1 e1)
{
    ... // exception handler 1
}
...
finally
{
    ... // executed before try statement is exited
}
```

The code in the `finally` clause is executed before the `try` statement is exited, in *any* of the three possible situations:

1. if the protected code block does not throw an exception,
2. if an exception is thrown and caught by one of the handlers,
3. if an exception is thrown and not caught by any of the handlers.

For instance, if a file has been opened in the protected code block, the code in the `finally` clause can make sure that the file is closed in any (also the error) case. Please note that the same may be *not* achieved by placing such code after the `try` statement: if an exception is thrown and not caught by any handler of this statement, this code is not executed.

We will see an example of this later.

5.7 Exceptions Thrown by Handlers

An exception handler may itself throw an exception; either because an error has occurred in the handler itself or because the handler wants to further propagate a caught exception. An exception thrown by a handler is *not* handled by the handlers in the current `try` statement but propagated upwards.

The case of a handler first catching an exception and then propagating it further upwards is demonstrated by the following program:

```
public static void m1()
{
    try
    {
        m2();
    }
    catch (MyException e)
    {
        ... // m1-specific handling of e
    }
}

public static void m2() throws MyException
{
    try
    {
```

```

    ...
  }
  catch (MyException e)
  {
    ... // m2-specific handling of e
    throw e;
  }
}

```

The protected code block in `m2` may throw an exception which is caught by a corresponding handler and, after appropriate handling, further propagated upwards where it is again caught by the handler in `m1`. This demonstrates that the `throw` clause needs not always create a new exception object; it may use any already existing object for this purpose.

5.8 Runtime Exceptions

The Java standard library defines the following hierarchy of exceptions:

```

Exception
|
+-- RuntimeException
|   |
|   +-- ArithmeticException
|   +-- IndexOutOfBoundsException
|   +-- NullPointerException
|   +-- ...
+-- IOException
+-- InterruptedException
+-- ClassNotFoundException
+-- ...

```

Example The Java standard class `Thread` provides the method

```
public static void sleep(long ms) throws InterruptedException
```

which lets the program sleep (temporarily cease execution) for the specified number of milliseconds. This sleep may be interrupted in which case an `InterruptedException` is thrown. We may thus let the current program sleep for 100 milliseconds by a call

```
try
{
    Thread.sleep(100);
}
catch(InterruptedException e)
{
    System.out.println("Someone woke me up.");
}
```

This hierarchy has a sub-hierarchy of *runtime exceptions*, i.e., of all subclasses of `RuntimeException`. These exceptions are special in two senses:

1. Runtime exceptions are generated as the results of illegal JVM instructions (not as the results of `throw` statements):
 - `ArithmeticException`: triggered by a division by zero.
 - `IndexOutOfBoundsException`: triggered by an array access with an index outside the array bounds.
 - `NullPointerException`: triggered by an attempt to dereference a null pointer.
2. Runtime exceptions need not be declared in method headers; they can therefore be propagated by the main method to the JVM.

As an example, a program that executes a division by zero instruction

```
public class Main
{
    public static void main(String[] args)
    {
        int i = 0;
        int j = 1/i;
    }
}
```

aborts with the following exception (i.e., the JVM catches the exception and prints the corresponding information):

```
Exception in thread "main"
java.lang.ArithmeticException: / by zero
    at Main.main(Main.java:6)
```

If we wish, we may explicitly catch runtime exceptions as in

```
try
{
    int i = 0;
    int j = 1/i;
}
catch(ArithmeticException e)
{
    System.out.println("Arithmetic Exception: " + e.getMessage());
}
```

The `Exception` method `getMessage` (which is expected to be overwritten by any subclass of `Exception`) describes the exception in more detail (such as “/ by zero” in above case).

The programmer may also declare own runtime exceptions as subclasses of `RuntimeException` and explicitly raise them by `throw` statements. In this case, the exception type needs not be declared in method headers. However, we do not recommend this as a general strategy because the information which exceptions are thrown by a method has to be considered as part of the method specification and should therefore be made explicit in the method header.

5.9 When Not to Use Exceptions

We complete our discussion of exceptions by a word on the pragmatics of their use. In a nutshell, one should reserve exceptions for that purpose that is denoted by their name: situations that are not typical for the normal program flow. Normal and expected cases should be handled by function return values and/or transient parameters.

To demonstrate the difference, let us use the previously introduced method `readInt` to compute the sum of a stream of integers:

```
public static int readAndSum() throws InputException
{
    int s = 0;
    while (true)
    {
        int i = readInt();
        if (Input.hasEnded()) break;
    }
}
```

```
        s += i;
    }
    return s;
}
```

This code clearly shows the expected flow of control, the exceptional situation that an input error has occurred is treated as an exception.

Now let us assume that a clever programmer writes a version of `readInt` where also the end of a file is denoted by an exception:

```
public class EndOfFileException extends Exception { }

public static public int readInt()
    throws EndOfFileException, InputException
{
    int i = Input.readInt();
    if (Input.hasEnded()) throw new EndOfFileException();
    if (!Input.isOkay()) throw new InputException("some error");
    return i;
}
```

Now the programmer can process a stream of integers as follows:

```
public static int readAndSum() throws InputException
{
    int s = 0;
    try
    {
        while (true)
        {
            int i = readInt();
            s += i;
        }
    }
    catch(EndOfFileException e)
    {
        return s;
    }
}
```

This program correctly computes and returns the sum of a stream of integers. However, the expected flow of control is disguised. The program only makes sense if the

input stream eventually ends and a sum is returned. Looking at the protected code block above, it seems that the program under normal circumstances would never terminate! This is because we actually *expect* that the `EndOfFileException` will eventually be raised; an expected exception however is not an exception at all.

5.10 File Input/Output

An important use of exceptions is to handle input/output, e.g., from/to text files. Java provides very flexible input/output mechanisms, but this flexibility comes at a certain price: the Java standard library (concretely the package `java.io`) contains a large number of classes related to file input/output which can be combined in many ways; thus the beginner has a hard time to get the overall picture. In the following, we will therefore only focus on some simple examples as they frequently occur in practice.

As for file input, we can read from a text file named “in.txt” as depicted by the following code snippet:

```
String name = "in.txt";
try
{
    BufferedReader input = new BufferedReader(new FileReader(name));
    while (true)
    {
        String line = input.readLine();
        if (line == null) break;
        ...
    }
    input.close();
}
catch(FileNotFoundException e)
{
    System.out.println("File " + name + " does not exist.")
}
catch(IOException e)
{
    System.out.println("IO error: " + e.getMessage());
}
```

For reading the file, we have to create a corresponding object `input` of type `BufferedReader`. The construction of this object proceeds via an intermediate

object of type `FileReader` whose constructor in turn takes the name of the file (possibly including a directory path) as an argument; this constructor throws an exception, if the corresponding file could not be found. We may then call on `input` the method `readLine()` to read from the file one line after the other; finally we close the file by invoking the `close()` method. Since a process only may have a limited number of files in use, it is good practice to close a file after it has been processed. If some input error occurs, a corresponding exception is raised.

Having discussed how to read from a text file, the following example demonstrates how we can write to a text file. The following method takes a number n and a file name *name* and writes the first n positive integer values to the corresponding disk file. The success of the operation is denoted by the boolean return value.

```
public public static boolean writeTable(int n, String name)
{
    try
    {
        PrintWriter output =
            new PrintWriter(new BufferedWriter(new FileWriter(name)));
        for (int i=1; i<=n; i++)
        {
            output.println(i);
        }
        output.close();
    }
    catch(IOException e)
    {
        return false;
    }
    return true;
}
```

The method creates a `PrintWriter` object `output` via some intermediate objects of type `BufferedWriter` and `FileWriter`, where the constructor of the latter one is invoked with the name (generally path) of the file to write to. On `output`, the usual `print/println` methods can be invoked to write the numbers to the file; ultimately the file is closed by the `close()` operation (which ensures that all output is actually written to disk).

5.11 The Input Class

We finally show how the KWM class `Input` for line-oriented input has been implemented with the help of exceptions. The structure of this class is as follows:

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

public class Input
{
    // last read has returned end of stream
    private public static boolean endFlag = false;

    // message describing the error
    private public static String errorMessage = null;

    // buffered reader wrapped around standard input stream
    private public static BufferedReader stdReader =
        new BufferedReader(new InputStreamReader(System.in));

    // current reader
    private public static BufferedReader reader = stdReader;
    ...
}
```

We create a `BufferedReader` by wrapping the standard input stream `System.in`. This object is used by a private method `readLine` to return the next line of input:

```
private public static String readLine()
{
    try
    {
        String line = reader.readLine();
        if (line == null)
            endFlag = true;
        return line;
    }
    catch(Exception e)
    {
        errorMessage = e.getMessage();
    }
}
```

```
    return null;
}
```

This method catches any exception raised by `reader.readLine` and saves the exception text in a variable `errorMessage`. If a null line is returned, the `endFlag` is set to true. The methods `isOkay` and `hasEnded` check for these variables as follows:

```
public public static boolean isOkay()
{
    return !endFlag && errorMessage == null;
}

public public static boolean hasEnded()
{
    return endFlag;
}
```

The method `getError` returns the actual text of the error message.

```
public public static String getError()
{
    return errorMessage;
}
```

The method `clearError` resets the error variable:

```
public public static void clearError()
{
    errorMessage = null;
}
```

The internal method `readLine` can be immediately used to return a string to the caller:

```
public public static String readString()
{
    return readLine();
}
```

However, for returning say a `int` value, the text line has to be correspondingly converted, e.g. by the standard method `Integer.parseInt`:

```
public public static int readInt()
{
    String line = readLine();
    if (line == null) return Integer.MIN_VALUE;
    try
    {
        return Integer.parseInt(line);
    }
    catch(Exception e)
    {
        errorMessage = e.getMessage();
    }
    return Integer.MIN_VALUE;
}
```

Please note that the last return statement is only reached in the case that an exception has been thrown by the conversion.

To set the input source to a text file, the method `openInput` can be called:

```
public public static void openInput(String name)
{
    try
    {
        reader = new BufferedReader(new FileReader(name));
    }
    catch(FileNotFoundException e)
    {
        errorMessage = e.getMessage();
    }
}
```

This method sets the reader object to a new `BufferedReader` that is connected to a `FileReader` stream. When `closeInput` is called, the reader object is reset to the originally opened `stdReader`.

```
public public static void closeInput()
{
    if (reader == stdReader)
    {
        errorMessage = "No file has been opened!";
        return;
    }
}
```

```
try
{
    reader.close();
}
catch (IOException e)
{
    errorMessage = e.getMessage();
}
reader = stdReader;
endFlag = false;
}
```

While the use of class `Input` is convenient for simple line-oriented input, more sophisticated input handling requires the explicit use of the input/output facilities provided by the Java library. The source code given in the last two sections should be a suitable starting point of further study of these facilities.

Literature

The following books were used in the preparation of this course:

John Lewis and William Loftus *Java Software Solutions — Foundations of Program Design*. 7th edition, Addison-Wesley, Reading, Massachusetts, 2011.

A good introduction to learning programming in Java (800 pages with CD-ROM) supplemented by a Web site with examples and sources. Starts earlier with true object-oriented programming than we do in our course, focuses a bit more on technical details of Java, and discusses a bit less basic programming principles. A lot of material, easy to read.

Hanspeter Mössenböck *Sprechen Sie Java? — Eine Einführung in das systematische Programmieren*. 5. Auflage, dpunkt.verlag, Heidelberg, Germany, 2014.

Based on the lecture notes of a course on Java programming for computer science students at the Johannes Kepler University of Linz (360 pages). This book is less a book about Java than about programming in general (still all of the basics of Java are introduced). Does not focus on object-oriented programming (most of the programs are written in an imperative style) but more on the systematics of program design. Compact and easy to read.

Mitchell Waite and Robert Lafore *Data Structures & Algorithms in Java*. Waite Group Press, Corete Madera, California, 1998.

Many important data structures and algorithms of computer science are introduced and implemented in Java (600 pages with CD-ROM). No deep analysis but focus on basic understanding. The best aspect of this book is that most of the algorithms are visualized by Java applets (most applets presented in this course are from this book).

Index

- ArithmeticException exception, 129
- BufferedReader class, 134, 136
- Exception class, 121
- FileReader class, 136
- InputStreamReader class, 134
- Input class, 134
- Object class, 88
- close method, 132, 136
- extends clause, 79, 81
- finally clause, 126
- final, keyword, 93
- parseInt method, 135
- readLine method, 134
- super, statement, 91
- throws clause, 122
- throw statement, 121
- try statement, 121

- abstract classes, 94
- abstract data types, 59
- abstract methods, 94
- abstract object methods, 59
- adts, 59
- application programming interface, 19
- arithmetic exceptions, 129
- array, 24
- array declaration, 25
- assignment, of objects, 85
- automated bound checking, 27
- automatically imported, 20
- average, 48

- child class, 79

- exception, circular array, 72
- class, 12
- class diagram, 80
- class field, 15
- class hierarchies, 79
- class library, 19
- class method, 18
- code sharing, 84
- collections, 59
- constructors, in derived classes, 81

- descending, 48
- dynamic type, 87

- error code, 117
- error codes, 117
- errors, 117
- exception handler, 121
- exception handlers, multiple, 124
- exception, declaring of, 122
- exceptions, 121
- exceptions, abuse of, 130
- exceptions, default handlers, 126
- exceptions, Java standard, 128
- exceptions, propagating, 124
- exports, 21

- fifo, 68
- files, 132
- final classes, 93
- final methods, 93
- framework, 96

- generic methods, 87

- index, 24
- inheritance, 76
- insertion sort, 46
- interface, 59
- interface hierarchies, 81
- is-relationship, 84

- java core api, 19

- lifo, 64
- linear, 43
- link, 49
- linked list, 49
- logarithmic, 45

- method overriding, 90
- modifiers, visibility, 97

- nodes, 49

- object assignments, 85
- object field, 14
- object method, 8, 16
- object-oriented, 76
- one-dimensional, 37
- override, 90
- overriding of methods, 90

- packages, 19
- parent class, 79
- polymorph, 87
- polymorphic, 62
- protected code block, 121

- quadratic, 48
- queue, 68

- runtime exceptions, 129

- searching, 29
- set diagram, 85
- shadowing a variable, 92
- shadowing of variables, 92
- single inheritance, 79

- singleton, 50
- stack, 64
- static type, 87
- streams, 132
- subclass, 79
- superclass, 79

- this, 13
- two-dimensional, 37
- type cast, 86
- type compatibility, 84

- variables, shadowing, 92
- visibility modifiers, 97