

Reading Sample

Java is an object-oriented programming language and the first sample chapter focuses on objects that are created by a blueprint (the class). Objects are addressed via references and through references objects can be passed on to other places and can be compared. The second sample chapter explores the Java class library.

-  **"Classes and Objects"**
-  **"The Class Library"**
-  **Contents**
-  **Index**
-  **The Author**

Christian Ullenboom

Java: The Comprehensive Guide

1126 pages, 2023, \$59.95

ISBN 978-1-4932-2295-7

 www.rheinwerk-computing.com/5557

Chapter 3

Classes and Objects

“Nothing is more fairly distributed than common sense: no one thinks he needs more of it than he already has.”
—René Descartes (1596–1650)

Java is an object-oriented programming language and this chapter focuses on objects that are created by a blueprint (the class). Objects are addressed via references and through references objects can be passed on to other places and can be compared.

3.1 Object-Oriented Programming

A book about Java programming must unite several parts:

- First, basic programming according to the imperative principle (variables, operators, case distinction, loops, and simple static methods) in a new grammar for Java
- Then, the object orientation (objects, classes, inheritance, interfaces) and extended possibilities of the Java language (exceptions, generics, lambda expressions)
- Finally, the libraries (string processing, input/output, etc.)

This chapter focuses on the paradigm of object orientation and demonstrates the syntax, such as how classes are implemented in Java and how class/object variables and methods are used.

Note

Java, of course, is not the first object-oriented (OO) language, nor was C++. Classically, Smalltalk and especially Simula-67 from 1967 are considered the progenitors of all OO languages. The concepts they introduced are still relevant today, including the four generally accepted principles of object-oriented programming (OOP): *abstraction*, *encapsulation*, *inheritance*, and *polymorphism*.¹



¹ Rest assured, all four basic pillars will be described in detail in the following chapters!

3.1.1 Why Object-Oriented Programming at All?

Since people perceive the world in objects, the analysis of systems is also often already modeled in an OO way. But with procedural systems that have only subroutines as a means of expression, mapping OO design into a programming language becomes difficult, and inevitably, a break will occur. Over time, documentation and implementation will diverge; the software then becomes difficult to maintain and extend. A better approach, therefore, is to think in an OO way and then use an OOP language to map these ideas.



Note

Bad code can be written in any language.

The objects mapped in the software have three important characteristics:

- Every object has an identity.
- Every object has a state.
- Every object has a behavior.

These three properties have important consequences: First, the identity of the object remains the same during its lifetime until its death and cannot change. Second, the data and the program code to manipulate that data are treated as belonging together. In procedural systems, you'll often find scenarios like a large memory area that can be accessed by all its subroutines in some way or other. For objects, this statement is not true since objects logically manage their own data and monitor the manipulation of that data.

So, OO software development is about modeling in objects and then programming. Design takes a central position in this process; large systems are decomposed and described in ever finer detail. The statement of the French writer François Duc de La Rochefoucauld (1613–1680) fits well here:

“Those who spend too much time on small things become incapable of great things.”

3.1.2 When I Think of Java, I Think of Reusability

With each new project, you may notice similar problems have already been solved in previous projects. Of course, problems that have already been solved shouldn't be reimplemented; instead, repetitive parts should be reused as best as possible in different contexts. The goal is the best possible reuse of components.

The reusability of program parts has existed since before OOP languages came into being; however, OOP languages facilitate the programming of reusable software components. Thus, the many thousands of classes in the library are examples of how

developers won't be constantly bothered about the implementation of data structures, for example, or the buffering of data streams.

Even though Java is an OOP language, being OO doesn't guarantee fancy design and ideal reusability. An OO language facilitates OOP, but OOP can also be achieved in a simple programming language such as C. In Java, programs are also possible that consist of only one class and accommodate 5,000 lines of program code with static methods. Bjarne Stroustrup (the creator of C++, also called Stumpy by his friends) aptly said this about comparing C and C++:

“C makes it easy to shoot yourself in the foot, C++ makes it harder, but when you do, it blows away your whole leg.”²

In the spirit of our didactic approach, this chapter will first use some classes from the standard library. We'll start with the `Point` class, which represents two-dimensional points. In a second step, we'll then program our own classes. Next, we'll focus on the concept of abstraction in Java, namely, how groups of related classes are designed.

3.2 Members of a Class

Classes are an important feature of OOP languages. A class defines a new type, describes the properties of the objects, and thus specifies the blueprint.

Each object is an *instance* of a class.

A class essentially declares two things:

- Attributes (what the object has)
- Operations (what the object can do)

Attributes and operations are also referred to as the *members* of an object. Which members a class should actually have is determined in the analysis and design phase. We won't describe this decision in this book; for us, the class descriptions are already available.

The operations of a class are implemented by the Java programming language through *methods*. The attributes of an object define its states, and they are implemented by class/object variables also referred to as *fields*.³

Note

The term “object-oriented programming” contains the term “object” but not the term “class,” which we've used quite a bit. So, why isn't OOP called “class-based programming” instead? The reason is that class declarations aren't mandatory for OO pro-

² Or as Bertrand Meyer put it, “Do not replace legacy software by lega-c++ software.”
³ We won't use the term “field” in this context because it is a reserved word for arrays.



grams. Another approach is *prototype-based object-oriented programming*. In this case, JavaScript is the best-known representative; only objects exist, which are concatenated with a kind of base type, the *prototype*.

To approach a class, let’s use a fun *first-person* approach (*object approach*), which is also used in the analysis and design phase. In this first-person approach, we put ourselves in the object and say “I am...” for the class, “I have...” for the attributes, and “I can...” for the operations. Readers should test this thought experiment on the human, car, worm, and cake classes.

Before we delve into custom classes, let’s first explore some classes from the standard library. A simple class is `Point`, which describes a point on a two-dimensional plane by the coordinates `x` and `y` and provides some operations to modify point objects. Let’s test a point again with the object approach.

Concept	Explanation
Class name	I am a point .
Attribute	I have an x and a y coordinate.
Operation	I can move myself and set my position .

Table 3.1 OOP Terms and Their Meanings

Regarding the point, in Oracle’s application programming interface (API) documentation (<https://docs.oracle.com/en/java/javase/17/docs/api/java.desktop/java/awt/Point.html>), you can read that it defines the object variables `x` and `y`, has a `setLocation(...)` method (among other things), and offers a constructor that takes two integers.

3.3 Natural Modeling Using Unified Modeling Language*

For the representation of a class, program code can be used (i.e., either text or a graphical notation). One of the available graphical description types is the Unified Modeling Language (UML). Graphic illustrations are much easier for people to understand and provide a broader overview.

In the first section of a UML diagram, you can read the attributes of an object; in the second section, its operations. The `+` before the members, as shown in Figure 3.1, indicates that they are public, and anyone can use them. Compared to Java, the type specification is reversed: First comes the name of the variable, then the type or (in the case of methods) the type of the return value. Other programming languages such as TypeScript or Kotlin also use this “flipped” type specification in the code.

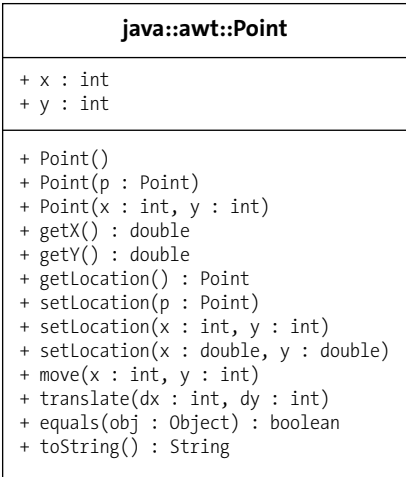


Figure 3.1 The `java.awt.Point` Class in a UML Representation

UML defines various diagram types that can describe different views of the software. Different diagrams are important for the individual phases in software design. Let’s take a brief look at four diagrams and their areas of use.

- **Use cases diagram**
A *use cases diagram* is usually created during the requirements phase and describes business processes by showing how people—or existing programs—interact with the system. The acting persons or active systems are called *actors* and are indicated in the diagram as small people. Use cases then describe the interaction with the system.
- **Class diagram**
For a static view of a program’s design, the *class diagram* is one of the most important diagram types. On one hand, a class diagram represents the elements of the class (i.e., its attributes and operations), and on the other hand, the relationships among the classes. Class diagrams are used most frequently in this book, especially to show association and inheritance to other classes. Classes are represented as rectangles in such a diagram, and the relationships between classes are indicated by lines.
- **Object diagram**
At first glance, a class diagram and an object diagram are quite similar. The main difference is that an *object diagram* visualizes the assignment of attributes (i.e., the object states). For this purpose, *instance specifications* are used, which include the relationships the object has with other objects at runtime. For example, if a class diagram describes a person, one rectangle appears in the diagram. If this person has friends at runtime (i.e., has associations to other person objects), then a great many people can be associated in an object diagram, while a class diagram can’t represent this instance.

■ Sequence diagram

The *sequence diagram* represents the dynamic behavior of objects. Thus, this diagram shows the order in which operations are called and when new objects are created. The individual objects are given a vertical lifeline, and horizontal lines between the lifelines of the objects describe the operations or object creations. Thus, the diagram is read from top to bottom.

Since the class diagram and object diagram tend to describe the structure of software, these models are also called *structure diagrams* (along with package diagrams, component diagrams, composition structure diagrams, and distribution diagrams). A use case diagram and a sequence diagram tend to show dynamic behavior and are therefore referred to as *behavior diagrams*. Other behavior diagrams include state diagrams, activity diagrams, interaction overview diagrams, communication diagrams, and timing diagrams. In UML, however, capturing the central statements of the system in a diagram is the goal, and thus, diagram types can be mixed without any problem.

In this book, you'll find mostly class diagrams.

3.4 Creating New Objects

A class describes what an object should look like. Expressed in a set or element relation, objects correspond to elements, and classes correspond to sets in which the objects are contained as elements. These objects have members that can be used. If a point represents coordinates, ways to query and change these states will be available.

In the following sections, we'll examine how instances of the `Point` class can be created at runtime and how the members of the `Point` objects can be accessed.

3.4.1 Creating an Instance of a Class Using the `new` Keyword

Objects must always be explicitly created in Java. For this purpose, the language defines the `new` keyword.

[Ex]

Example

The Java library declares the `Point` type for points. The following code will create a `Point` object:

```
new java.awt.Point();
```

Basically, `new` is something like a unary operator. The `new` keyword is followed by the name of the class of which an instance is to be created. The class name is fully qualified here because `Point` is in a `java.awt` package. (A package is a group of related classes; in

Section 3.6.3, you'll see how developers can also abbreviate this notation.) The class name is followed by a pair of parentheses for the *constructor call*. This call is a kind of method call that can be used to pass values for the initialization of the fresh object.

If Java's memory management can reserve free memory for the object to be created and if the constructor can be passed through validly, the `new` expression subsequently returns a *reference* to the fresh object to the program. If we don't remember this reference, automatic garbage collection can release the object.

3.4.2 Declaring Reference Variables

The result of `new` is a reference to the new object. The reference is usually held in a *reference variable* in order to be able to access properties of the object later.

Example

Let's declare the variable `p` of type `java.awt.Point`. The `p` variable then takes the reference from the new object created via `new`, as in the following example:

```
java.awt.Point p;  
p = new java.awt.Point();
```

The declaration and initialization of a reference variable can be combined (also a local reference variable is uninitialized at the beginning like a local variable of a primitive type) in the following example:

```
java.awt.Point p = new java.awt.Point();
```

The types must be compatible, of course, and a `Point` object won't pass as a `String`. Thus, attempting to assign a point object to an `int` or `string` variable will result in a compiler error. Consider the following examples:

```
int    p = new java.awt.Point(); // ☠ Type mismatch: cannot convert from  
                                     // Point to int  
String s = new java.awt.Point(); // ☠ Type mismatch: cannot convert from  
                                     // Point to String
```

So, a variable stores either a simple value (variable of type `int`, `boolean`, `double`, etc.) or a reference to an object. Ultimately, the reference is internally a pointer to a memory area but isn't visible to Java developers in this way.

Reference types are available in four designs: *class types*, *interface types*, *array types* (also called *field types*), and *type variables* (a special generic type). Our case represents an example of a class type.

[Ex]

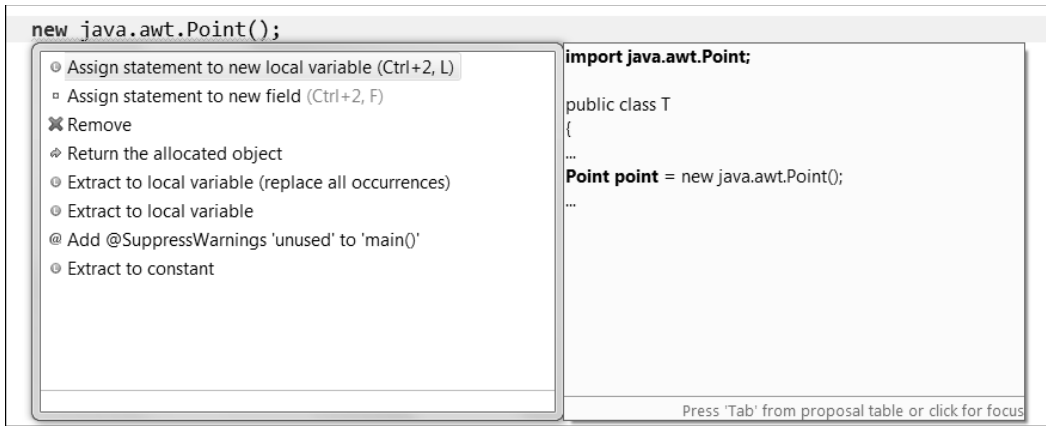


Figure 3.2 Pressing Ctrl+1 Allows You to Create Either a New Local Variable or an Object Variable for the Expression

3.4.3 Let’s Get to the Point: Accessing Object Variables and Methods

The variables declared in a class are called *object variables* or *instance variables*. Each created object has its own set of object variables,⁴ which make up the state of the object.

The dot operator allows you to access the states or call methods on objects. The dot is located between an expression that provides a reference and the object member. The API documentation describes which members exactly are provided by a class—if an object doesn’t have a member, the compiler will prohibit its use.

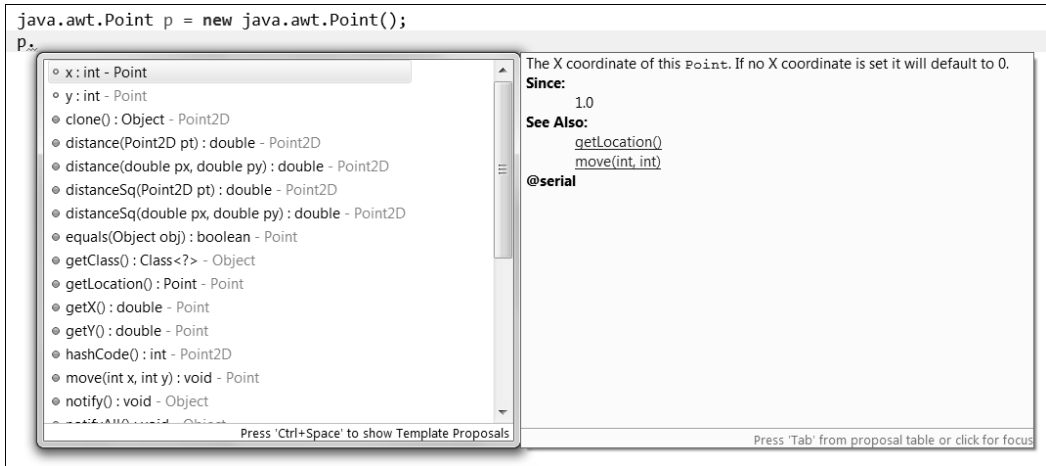


Figure 3.3 Ctrl+Space Displays the Possible Members of a Reference. Pressing Enter Selects the Member and, Especially for Methods, Places the Cursor between the Pair of Parentheses.

⁴ In some cases, several objects will share one variable, called *static variables*. We’ll look at this case in more detail later in Chapter 6, Section 6.3.

Example

The `p` variable references a `java.awt.Point` object. The object variables `x` and `y` are supposed to be initialized, as in the following example:

```
java.awt.Point p = new java.awt.Point();
p.x = 1;
p.y = 2 + p.x;
```

A method call is just as simple as an access to class or object variables. The expression with the reference is followed by the method name after the dot.

Door and Playing Piece on the Game Board

At first glance, point objects appear to be mathematical constructs, but they can be used universally. Anything that has a position in two-dimensional space can be represented by a point object. The point stores `x` and `y` for us, and if we didn’t have any point objects, we’d always have to store `x` and `y` separately.

Let’s now put a playing piece and a door on a game board. Of course, the two objects have positions. Without objects, the storage of these coordinates would perhaps look like the following example:

```
int playerX;
int playerY;
int doorX;
int doorY;
```

Modeling `x` and `y` separately isn’t ideal since a much better abstraction is available by using the `Point` class, which also provides several useful methods.

Without Abstraction, Only the Bare Data	Encapsulation of the States in an Object
<pre>int playerX; int playerY;</pre>	<pre>java.awt.Point player;</pre>
<pre>int doorX; int doorY;</pre>	<pre>java.awt.Point door;</pre>

Table 3.2 Objects Encapsulate States

The following example creates two points representing the `x/y` coordinates of a playing piece and a door on a game board. Once the points have been created, the coordinates are set, and a test is carried out to see how far apart the playing piece and the door in the following example:

```
class PlayerAndDoorAsPoints {

    public static void main( String[] args ) {
        java.awt.Point player = new java.awt.Point();
        player.x = player.y = 10;

        java.awt.Point door = new java.awt.Point();
        door.setLocation( 10, 100 );

        System.out.println( player.distance( door ) ); // 90.0
    }
}
```

Listing 3.1 PlayerAndDoorAsPoints.java

In the first case, we are explicitly assigning the variables `x` and `y` of the game. In the second case, we won't directly set the objects' states via the variables but instead change these states via the `setLocation(...)` method. The two objects have their own coordinates and won't get in each other's way.

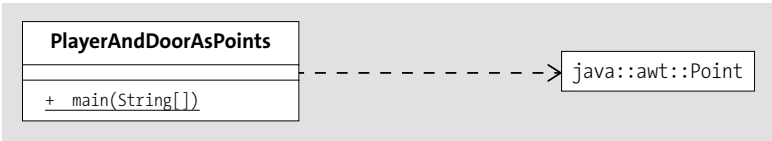


Figure 3.4 UML Diagram Showing the Dependency between a Class and `java.awt.Point` with a Dashed Line. Attributes and Operations of the Point Object Are Not Shown.

toString()

The `toString()` method returns a `String` object as the result to reveal the state of the point. This method is special in that every object has a `toString()` method—however, the output is not useful in every case.

```
class PointToStringDemo {

    public static void main( String[] args ) {
        java.awt.Point player = new java.awt.Point();
        java.awt.Point door = new java.awt.Point();
        door.setLocation( 10, 100 );

        System.out.println( player.toString() ); // java.awt.Point[x=0,y=0]
        System.out.println( door );             // java.awt.Point[x=10,y=100]
    }
}
```

Listing 3.2 PointToStringDemo.java

Tip

Instead of explicitly calling `println(obj.toString())` for the output, `println(obj)` works as well. This method is useful because the signature `println(Object)` accepts any object as an argument and automatically calls the `toString()` method on that object.

After the Dot, Life Goes On

As you've seen, the `toString()` method returns a `String` object as a result, as in the following code:

```
java.awt.Point p = new java.awt.Point();
String s = p.toString();
System.out.println( s ); // java.awt.Point[x=0,y=0]
```

The `String` object itself has methods too. One `String` object method is `length()`, which returns the length of the string, as in the following example:

```
System.out.println( s.length() ); // 23
```

You can combine the request of the `String` object and its length into a single expression, which is referred to as *cascaded calls*, as in the following example:

```
java.awt.Point p = new java.awt.Point();
System.out.println( p.toString().length() ); // 23
```

Object Creation without Variable Assignment

When using object members, the type to the left of the point must always be a reference. Whether the reference comes from a variable or is created on-the-fly does not matter. Consider the following example:

```
java.awt.Point p = new java.awt.Point();
System.out.println( p.toString().length() ); // 23
```

The following code does exactly the same thing:

```
System.out.println( new java.awt.Point().toString().length() ); // 23
```

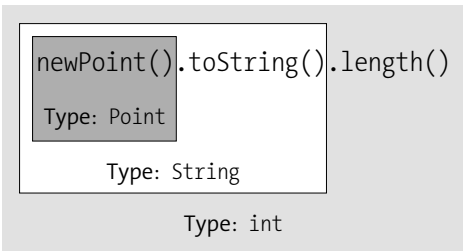


Figure 3.5 Each Nesting Results in a New Type

Basically, the following statement also works:

```
new java.awt.Point().x = 1;
```

However, this last option doesn't make sense in this context because, although the object is created and an object variable is set, the object is then fair game again for automatic garbage collection.



Example

You can use the `File` object to determine the size of a file with the following code:

```
long size = new java.io.File( "file.txt" ).length();
```

The return value of the `File` method `length()` is the length of the file in bytes.

3.4.4 The Connection between new, the Heap, and the Garbage Collector

If a runtime system receives a request to create an object via `new`, it reserves enough memory to accommodate all object members and management information. A `Point` object stores coordinates in two `int` values so at least 2×4 bytes are needed. The Java runtime environment obtains the memory space from the *heap*. The heap grows from a starting size to the maximum size allowed so that a Java program can't grab arbitrary amounts of memory from the operating system, which would probably cause the machine to crash. In the HotSpot Java virtual machine (JVM), the heap is $1/64$ of main memory at startup and then grows to the maximum size of $1/4$ of main memory.⁵



Note

Only a few special cases in Java exist where `new` objects aren't created via `new`. Thus, the `newInstance()` method, based on native code, creates a new object from the `Constructor` object. Also, `clone()` can create a new object as a copy of another object. With string concatenation via `+`, you don't see a `new`, but the compiler will build internal code to create a new `String` object.

If the system can't provide enough memory for a new object, automatic garbage collection tries to clear away everything unused in a final rescue operation. If still not enough free memory is available, the runtime environment will generate an `OutOfMemoryError` and terminate the entire program.⁶

⁵ <https://docs.oracle.com/en/java/javase/17/gctuning/ergonomics.html>

⁶ However, this particular exception can also be intercepted, which is important for server operation because if a buffer can't be created, for example, the whole JVM should not stop immediately.

Heap and Stack

The JVM specification provides for five different *runtime data areas*.⁷ In addition to *heap memory*, let's take a quick look at *stack memory*. The Java Runtime Environment (JRE) uses it for local variables, for example. Also, Java uses the stack when calling methods with parameters. The arguments go on the stack before the method call, and the called method can access the values by reading or writing through the stack. With endless recursive method calls, the maximum stack size will be reached at some point, and an exception of type `java.lang.StackOverflowError` will occur. Since a JVM stack is associated with each thread, the end of the thread has come, with other threads continuing without impact.

Automatic Garbage Collection: It's Gone

Let's consider the following scenario:

```
java.awt.Point binariumLocation;
binariumLocation = new java.awt.Point( 50, 9 );
binariumLocation = new java.awt.Point( 51, 7 );
```

In this code, we're declaring a `Point` variable, building an instance, and assigning the variable. Then, we create a new `Point` object and override the variable. But what about the first point?

If the object is no longer referenced by the program, the automatic garbage collector notices and releases the reserved memory.⁸ Automatic garbage collection regularly tests whether the objects on the heap are still needed. If they aren't needed, the object hunter deletes them. So, a kind of graveyard atmosphere always exists around the heap, and once the last reference has been taken from the object, it's already dead. Several garbage collection algorithms exist, and each JVM vendor has its own procedures.

3.4.5 Overview of Point Methods

A few methods of the `Point` class have already been mentioned, and the API documentation naturally enumerates all methods:

```
class java.awt.Point
```

The more interesting methods include the following:

- `double getX()`
- `double getY()`
Returns the *x* or *y* coordinate.

⁷ Section 2.5 of the JVM specification, available at <https://docs.oracle.com/javase/specs/jvms/se17/html/jvms-2.html#jvms-2.5>.

⁸ With the `java` switch `-verbose:gc` set, a console output will always be produced when the garbage collector detects objects that are no longer referenced and clears them away.

- ▶ void setLocation(double x, double y)
Sets the x and the y coordinates at the same time. The coordinates are rounded and stored in integers.
- ▶ boolean equals(Object obj)
Checks if another point has the same coordinates. If so, the return is true; otherwise, it's false. If something other than a point is passed, the compiler won't find fault with this, but the result will always be false.

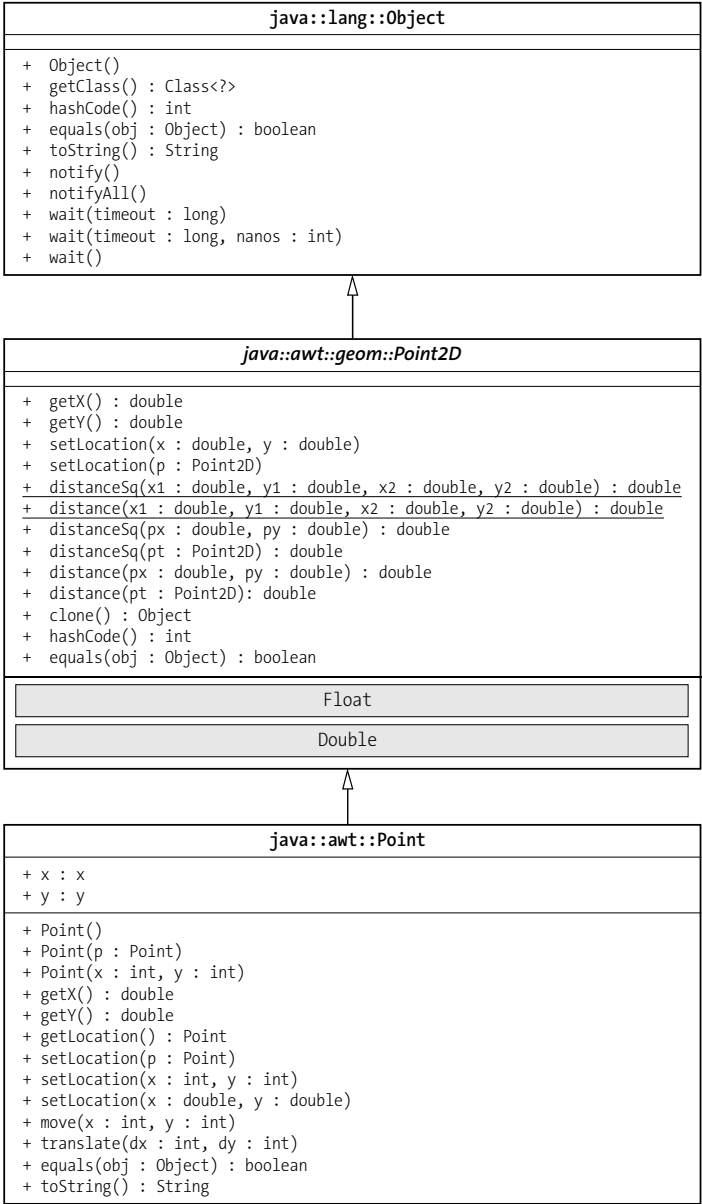


Figure 3.6 Inheritance Hierarchy in Point2D

Note

You may be surprised that a Point stores the coordinates as int, but the getX() and getY() methods return a double, and setLocation(double, double) takes the coordinates as double, rounds them, and stores them as int, thus losing precision. The reason relates to inheritance, which will be discussed in more detail in Chapter 7, Section 7.2. Point inherits from Point2D, and there's already double getX(), double getY(), and setLocation(double, double); the Point subclass can't simply turn double into int.

A Few Words about Inheritance and the API Documentation*

Not only does a class have its own members, it always inherits some members from its parents as well. In the case of Point, the superclass is Point2D—according to the API documentation. Even Point2D inherits from Object, the magic class that all Java classes have as a superclass. We'll devote Chapter 7, Section 7.2, to inheritance later, but right now, you must understand that the superclass passes object variables and methods to subclasses. Inherited object variables and methods are only briefly listed in the API documentation of a class in the block “Methods inherited from...” and are quickly forgotten. Therefore, developers must look not only at the methods of the class itself, but also at its inherited methods. So, for Point, we need to not just understand the methods of Point themselves, but also the methods from Point2D and Object.

Let's look at some methods of the superclass. The class declaration of Point contains an extends Point2D, which makes it explicitly clear that a superclass exists.⁹

```
class java.awt.Point
extends Point2D
```

- ▶ static double distance(double x1, double y1, double x2, double y2)
Calculates the distance between the given points according to Euclidean distance.
- ▶ double distance(double x, double y)
Calculates the distance of the current point to the specified coordinates.
- ▶ double distance(Point2D pt)
Calculates the distance of the current point to the coordinates of the passed point.

Are Two Points the Same?

The equals(...) method tells you whether two points are equal, and its use is pretty simple. Let's imagine managing the coordinates for a player, a door, and a snake and

⁹ However, the class declaration is not yet complete since an implements Serializable is missing, but we are not concerned about this for now.

then test whether the player is “on” the door and whether the snake is “on” the player’s position.

```
class PointEqualsDemo {

    public static void main( String[] args ) {
        java.awt.Point player = new java.awt.Point();
        player.x = player.y = 10;

        java.awt.Point door = new java.awt.Point();
        door.setLocation( 10, 10 );

        System.out.println( player.equals( door ) );    // true
        System.out.println( door.equals( player ) );    // true

        java.awt.Point snake = new java.awt.Point();
        snake.setLocation( 20, 22 );

        System.out.println( snake.equals( door ) );    // false
    }
}
```

Listing 3.3 PointEqualsDemo.java

Since player and door have the same coordinates, equals(...) returns true. Whether we have the distance from the player to the door calculated or the distance from the door to the player—the result with equals(...) should always be symmetrical.

Another test option results from distance(...) because, if the distance between points is zero, then the points naturally lie on top of each other and thus have no distance.

```
class Distances {

    public static void main( String[] args ) {
        java.awt.Point player = new java.awt.Point();
        player.setLocation( 10, 10 );
        java.awt.Point door = new java.awt.Point();
        door.setLocation( 10, 10 );
        java.awt.Point snake = new java.awt.Point();
        snake.setLocation( 20, 10 );

        System.out.println( player.distance( door ) );          // 0.0
        System.out.println( player.distance( snake ) );          // 10.0
        System.out.println( player.distance( snake.x, snake.y ) ); // 10.0
    }
}
```

```
}
}
```

Listing 3.4 Distances.java

Player, door, and snake are again represented as Point objects and preassigned with positions. For the player, we then call the distance(...) method and pass the reference to the door and the snake.

3.4.6 Using Constructors

When objects are created with new, a constructor gets called. A constructor has the task of putting an object into a start state, for example, initializing the object variables. A constructor is a good approach to initialization because a constructor is always called first, even before any other method is called. The initialization in the constructor makes sure that the new object has a meaningful initial state:

```
class java.awt.Point
    extends Point2D
```

Three constructors can be found in the API documentation for Point:

- Point()
Creates a point with the coordinates (0, 0).
- Point(int x, int y)
Creates a new point and initializes it with the values from x and y.
- Point(Point p)
Creates a new point and initializes it with the coordinates the passed point has as well. This kind of constructor is called a *copy constructor*.

A constructor without arguments is at *parameterless constructor*, sometimes also referred to as a *no-arg constructor*. Each class can have at most one parameterless constructor. You can also have a class that doesn’t declare a parameterless constructor, only constructors with parameters (i.e., parameterized constructors).

Example

The following three variants create a Point object with the same coordinates (1, 2). Note that java.awt.Point has been abbreviated to Point:

Point p = new Point(); p.setLocation(1, 2);
Point q = new Point(1, 2);
Point r = new Point(q);

The parameterless constructor is written first, while the second and third constructors are parameterized constructors.



3.5 ZZZZZnake

A classic computer game is *Snake*. On the screen is a player, a snake, some gold, and a door. The door and the gold are fixed, the player can be moved, and the snake moves independently towards the player. You must try to move the player to the gold and then to the door. If the snake catches you before you achieve these goals, you're unlucky, and it's game over.

This game may sound complex at first glance, but you already have all the building blocks to program this game:

- Player, Snake, Gold, and Door are `Point` objects preconfigured with coordinates.
- A loop runs through all coordinates. If a player, the door, the snake, or gold has been "hit," a symbolic representation of the figure is displayed.
- You'll test three conditions for the game status: 1) Has the player collected the gold and is standing on the door? (You've won the game.) 2) Does the snake bite the player? (You've lost the game.) 3) Does the player collect gold?
- The `Scanner` enables you to respond to keystrokes and move the player around the board.
- The snake must move in the direction of the player. While the player can only move horizontally or vertically, the snake can move diagonally.

The corresponding source code for this game follows:

```
public class ZZZZZnake {

    public static void main( String[] args ) {
        java.awt.Point playerPosition = new java.awt.Point( 10, 9 );
        java.awt.Point snakePosition  = new java.awt.Point( 30, 2 );
        java.awt.Point goldPosition   = new java.awt.Point( 6, 6 );
        java.awt.Point doorPosition   = new java.awt.Point( 0, 5 );
        boolean rich = false;

        while ( true ) {
            // Draw grid and symbols

            for ( int y = 0; y < 10; y++ ) {
                for ( int x = 0; x < 40; x++ ) {
                    java.awt.Point p = new java.awt.Point( x, y );
                    if ( playerPosition.equals( p ) )
                        System.out.print( '&' );
                    else if ( snakePosition.equals( p ) )
                        System.out.print( 'S' );
                    else if ( goldPosition.equals( p ) )
                        System.out.print( '$' );
```

```
                    else if ( doorPosition.equals( p ) )
                        System.out.print( '#' );
                    else System.out.print( '.' );
                }
                System.out.println();
            }

            // Determine status

            if ( rich && playerPosition.equals( doorPosition ) ) {
                System.out.println( "You won!" );
                return;
            }
            if ( playerPosition.equals( snakePosition ) ) {
                System.out.println( "SSSSSS. You were bitten by the snake!" );
                return;
            }
            if ( playerPosition.equals( goldPosition ) ) {
                rich = true;
                goldPosition.setLocation( -1, -1 );
            }

            // Console input and change player position
            // Keep playing field between 0/0.. 39/9
            switch ( new java.util.Scanner( System.in ).next() ) {
                case "u" /* p */ -> playerPosition.y = Math.max( 0, playerPosition.y - 1 );
                case "d" /* own */ -> playerPosition.y = Math.min( 9, playerPosition.y + 1 );
                case "l" /* eft */ -> playerPosition.x = Math.max( 0, playerPosition.x - 1 );
                case "r" /* ight */ -> playerPosition.x = Math.min( 39, playerPosition.x + 1 );
            }

            // Snake moves towards the player

            if ( playerPosition.x < snakePosition.x )
                snakePosition.x--;
            else if ( playerPosition.x > snakePosition.x )
                snakePosition.x++;
            if ( playerPosition.y < snakePosition.y )
                snakePosition.y--;
            else if ( playerPosition.y > snakePosition.y )
                snakePosition.y++;
        } // end while
    }
}
```

Listing 3.5 ZZZZZnake.java

The `Point` members in use include the following:

- The object states `x`, `y`: The player and the snake can be moved, and the coordinates must be reset.
- The `setLocation(...)` method: Once the gold has been collected, you set the coordinates so that the coordinate from the gold is no longer on our grid.
- The `equals(...)` method: Tests if a point is on top of another point.



Extension

If you're interested in a little more programming on this task, consider the following enhancements:

- Player, snake, gold, and door should be set to random coordinates.
- Instead of just one piece of gold, there should be two pieces.
- Instead of one snake, there should be two snakes.
- With two snakes and two pieces of gold, things can get a little tight for the player. Let's give the player a head start of 5 moves at the beginning without the snakes moving.
- For advanced developers: The program, which is so far only contained in the `main` method, should be split into different methods.

3.6 Tying Packages, Imports, and Compilation Units

The class library of Java is rather extensive, with thousands of types, and covers everything developers of platform-independent programs need as a basis. The class library includes data structures, classes for date/time calculation, file processing, and more. Most types are implemented in Java itself (and the source code is usually directly available from the development environment), but some parts are implemented natively, for example, when reading from a file.

When you program your own classes, they supplement the standard library, so to speak; the bottom line is that creating your own classes increases the number of possible types a program can use.

3.6.1 Java Packages

A *package* is a group of thematically related types. Packages can be arranged in hierarchies so that one package can contain another package—similar to the directory structure of a file system. Examples of packages include the following:

- `java.awt`
- `java.util`

- `com.google`
- `org.apache.commons.math3.fraction`
- `com.tutego.insel`

The Java standard library classes are located in packages starting with `java` and `javax`. Google uses the `com.google` root; the Apache Foundation publishes Java code at `org.apache`. In this way, you can read from the outside which types your own class depends on.

3.6.2 Packages in the Standard Library

The logical grouping and hierarchy can be observed easily in the Java library. The Java standard library starts with the `java` root; some types are in `javax`. This package contains other packages, such as `awt`, `math`, and `util`. For example, `java.math` contains the classes `BigInteger` and `BigDecimal` because working with arbitrarily large integers and floats is part of mathematics. A point and a polygon, represented by the `Point` and `Polygon` classes, are part of the graphical user interface (GUI) package, which is the `java.awt` package.

If someone put custom classes in packages with the prefix `java`, for example, `java.tutego`, a program author might cause confusion since you could no longer easily see whether the package was a component of each distribution. For this reason, the prefix `java` is forbidden for custom packages.

Classes that are in a package starting with `javax` can be part of the Java SE like `javax.swing`, but these classes don't necessarily have to be part of the Java SE; more on this topic will follow in Chapter 16.

3.6.3 Full Qualification and Import Declaration

To use the `Point` class, which is located in the `java.awt` package, outside of the `java.awt` package (which is almost always the case), you must make this known to the compiler with the entire package specification. For this purpose, the class name alone isn't sufficient because the class name might be ambiguous and a class declaration might exist in different packages.

Types can't be uniquely identified until you specify their package. A dot separates packages, so you'll write `java.awt` and `java.util` instead of just `awt` or `util`. With an innumerable number of packages and classes worldwide, uniqueness would not be feasible at all if we didn't write it this way. A type with the same name may be in different packages, for example, `java.util.List` and `java.awt.List` or `java.util.Date` and `java.sql.Date`. Therefore, only the package and the type together form a unique identifier.

To enable the compiler to precisely assign a class to a package, two options are available: First, types can be fully qualified, as we’ve been doing up to now. An alternative and more practical approach is to make the compiler aware of the types in the package through an `import` declaration.

Full Qualificataion	Import Declaration
<pre>class AwtWithoutImport { public static void main(String[] args){ java.awt.Point p = new java.awt.Point(); java.awt.Polygon t = new java.awt.Polygon(); t.addPoint(10, 10); t.addPoint(10, 20); t.addPoint(20, 10); System.out.println(p); System.out.println(t.contains(15, 15)); } }</pre> <p>Listing 3.6: AwtWithoutImport.java</p>	<pre>import java.awt.Point; import java.awt.Polygon; class AwtWithImport { public static void main(String[] args){ Point p = new Point(); Polygon t = new Polygon(); t.addPoint(10, 10); t.addPoint(10, 20); t.addPoint(20, 10); System.out.println(p); System.out.println(t.contains(15, 15)); } }</pre> <p>Listing 3.7: AwtWithImport.java</p>

Table 3.3 Type Access via Full Qualification and with an import Declaration

The source code on the left uses the full qualification approach, and each reference to a type costs more writing effort; on the right, the `import` declaration only mentions the class name and “swaps out” the package specification to an `import`. All types named with `import` are remembered by the compiler for this file in a data structure. When the compiler arrives at the line with `Point p = new Point();`, it finds the `Point` type in its data structure and can assign the type to the `java.awt` package, thus again providing the indispensable qualification.



Note
The types from `java.lang` are automatically imported, and thus, `import java.lang.String;` isn’t needed.

3.6.4 Reaching All Types of a Package with Type-Import-on-Demand

If a Java class accesses several other types of the same package, the number of `import` declarations can become large. In our example, we’re only using two classes from `java.awt` (`Point` and `Polygon`), but you can easily imagine what happens when additional windows, labels, buttons, sliders, and more are included from the GUI package. In this case, an asterisk `*` is allowed as the last member in an `import` declaration:

```
import java.awt.*;
import java.math.*;
```

With this syntax, the compiler knows all the types in the `java.awt` and `java.math` packages, so the compiler can map the package for the `Point` and `Polygon` classes as well as the package for the `BigInteger` class.

«

Note

The `*` is only allowed on the last hierarchy level and always applies to all types in this package. The following examples are syntactically incorrect:

```
import *; // Syntax error on token "*", Identifier expected
import java.awt.Po*; // Syntax error on token "*", delete this token
```

A declaration like `import java.*;` is syntactically correct but has no effect because no type declarations exist in the package `java`, only subpackages.

The `import` declaration refers only to a directory (assuming that the packages are mapped to the file system) and doesn’t include subdirectories.

»

Although the `*` does shorten the number of individual `import` declarations, there are two things to keep in mind:

- If two different packages contain a type with the same name, for example, `Date` in both `java.util` and `java.sql` or `List` in both `java.awt` and `java.util`, an interpretation error will occur when the type is used because the compiler doesn’t understand the meaning. Full qualification will solve the problem.
- The number of `import` declarations tells you something about the degree of complexity. The more `import` declarations you use the greater the dependencies on other classes, which is generally a red flag. Although graphical tools can show dependencies accurately, `import *` could obscure the full scope of dependency.

+

Best Practice

Development environments usually set the `import` declarations automatically and usually cascade the blocks. Therefore, the `*` should be used sparingly because it “pollutes” the namespace by many types and increases the risk of collision.

+

3.6.5 Hierarchical Structures across Packages and Mirroring in the File System

The classes belonging to a package are usually located¹⁰ in the same directory. The name of the package is the same as the name of the directory (and vice versa, of course). Instead of the directory separator (such as “/” or “\”), a dot is used.

Let’s assume the following directory structure with a helper class:

com/tutego/insel/printer/DatePrinter.class

In this case, the package name is `com.tutego.insel.printer`, and thus the directory name is *com/tutego/insel/printer*. Umlauts and special characters should be avoided, because they always cause trouble in the file system. Identifiers should always be in English anyway.

The Structure of Package Names

Basically, a package name can be arbitrary, but hierarchies usually consist of inverted domain names. Thus, the domain of the website *http://tutego.com* becomes `com.tutego`. This naming ensures that classes remain unique worldwide. A package name is usually written entirely in lowercase.

3.6.6 The Package Declaration

To place the `DatePrinter` class in a `com.tutego.insel.printer` package, two things must be true:

- The package must be physically located in a directory (i.e., *com/tutego/insel/printer*).
- The source code contains a package declaration at the top.

The package declaration must be located at the very start; otherwise, an interpretation error will occur (of course, comments can be placed before the package declaration):

```
package com.tutego.insel.printer;

import java.time.LocalDate;
import java.time.format.*;

public class DatePrinter {
    public static void printCurrentDate() {
        DateTimeFormatter fmt =
            DateTimeFormatter.ofLocalizedDate( FormatStyle.MEDIUM );
```

¹⁰ I wrote “usually” because the package structure doesn’t necessarily need to be mapped to directories. Packages could be read from a database by the class loader, for example. In the following sections, however, we always want to start from directories.

```
        System.out.println( LocalDate.now().format( fmt ) );
    }
}
```

Listing 3.6 *src/main/java/com/tutego/insel/printer/DatePrinter.java*

The package declaration is followed by the `import` declaration(s) and the type declaration(s) as usual.

To use the class, you have two options, as you already know: either the full qualification or an `import` declaration. The following code is an example of the first variant.

```
public class DatePrinterUser1 {
    public static void main( String[] args ) {
        com.tutego.insel.printer.DatePrinter.printCurrentDate();
    }
}
```

Listing 3.7 *src/main/java/DatePrinterUser1.java*

The following code is an example of the second variant, with the `import` declaration:

```
import com.tutego.insel.printer.DatePrinter;

public class DatePrinterUser2 {
    public static void main( String[] args ) {
        DatePrinter.printCurrentDate();
    }
}
```

Listing 3.8 *src/main/java/DatePrinterUser2.java*

Tip

A development environment takes a lot of work off your hands, so you’ll rarely notice file operations like creating directories, for example. A modern integrated development environment (IDE) also takes care of moving types to other packages, the associated changes to the file system, and adjustments to the `import` and package declarations for you.

3.6.7 Unnamed Package (Default Package)

An *unnamed package* or *default package* contains a class without a package specification. A best practice is to always organize your own classes in packages. Doing so enables finer visibility, and conflicts with other companies and other programmers can be avoided. Big problems could arise if 1) every company were messy and put all their

classes in unnamed packages and then 2) tried to swap out libraries. Conflicts would essentially be preprogrammed.

A class located in the package can import any other visible class from other packages, but not classes from the unnamed package. Let's assume `Sugar` is in the unnamed package and `Chocolate` in the `com.tutego` package:

```
Sugar.class
com/tutego/insel/Chocolate.class
```

The `Chocolate` class can't use `Sugar` because classes from the unnamed package are not visible to subpackages. Only other classes in the unnamed package can use classes in the unnamed package.

If `Sugar` was in a package (or even in a superpackage!), `Chocolate` could import `Sugar`:

```
com/Sugar.class
com/tutego/insel/Chocolate.class
```

3.6.8 Compilation Unit

A `.java` file is a *compilation unit* that consists of three (optional) segments in the following order:

- 1. The package declaration
- 2. The import declaration(s)
- 3. The declaration(s)

Thus, a compilation unit consists of, at most, one package declaration (not necessary if the type is in the default package), any number of import declarations, and any number of type declarations. The compiler translates each type of compilation unit into its own `.class` file. A package is ultimately a collection of compilation units. Usually, the compilation unit is a source code file; in general, the lines of code could also come from a database or be generated at runtime.

3.6.9 Static Import*

The import declaration informs the compiler about the packages, so that a type no longer needs to be fully qualified if it's explicitly listed in the import section or if the package of the type is named via `*`.

If a class specifies static methods or constants, its members are always addressed by the type name. Java provides a *static import* as an option to use the static methods or variables immediately without prefixed type names. So, while the normal import names the types to the compiler, a static import makes class members known to the compiler, thus going one level deeper.

Example

Import the static variable `out` from `System` statically for the screen output with the following code:

```
import static java.lang.System.out;
```

With the otherwise usual output via `System.out.print*(...)`, the class name can be omitted after the static import so that you only need `out.print*(...)`.

Let's include several static members in the following example of a static import:

```
package com.tutego.insel.oop;

import static java.lang.System.out;
import static javax.swing.JOptionPane.showInputDialog;
import static java.lang.Integer.parseInt;
import static java.lang.Math.max;
import static java.lang.Math.min;

class StaticImport {

    public static void main( String[] args ) {
        int i = parseInt( showInputDialog( "First number" ) );
        int j = parseInt( showInputDialog( "Second number" ) );
        out.printf( "%d is greater than or equal to %d.%n",
                    max(i, j), min(i, j) );
    }
}
```

Listing 3.9 src/main/java/com/tutego/insel/oop/StaticImport.java

Importing Multiple Types Statically

The following static import imports the static `max(...)/min(...)` methods:

```
import static java.lang.Math.max;
import static java.lang.Math.min;
```

If you require more static methods, the wildcard variant goes beyond the individual enumeration, as in the following example:

```
import static java.lang.Math.*;
```

Best Practice

Even if Java provides this wildcard option, you should use it in moderation. The possibility of static imports is useful when classes want to use constants, but you run the



risk, with a missing type name, that where the member actually comes from will become invisible, as well as what dependency it is based on. Problems also arise with methods that have names that are homonyms: A method from its own class can overlay statically imported methods. So, if later in the custom class—or superclass—a method is included that has the same signature as a statically imported method, a compiler error will not arise, but the semantics will change because then the new custom method will be used instead of the statically imported one.

3.7 Using References, Diversity, Identity, and Equality

In Java, `null` is an incredibly special reference that can trigger a large number of problems. But you can't do without it, and the following section will demonstrate its importance. After this discussion, we'll look at how object comparisons work and the difference between identity and equivalence.

3.7.1 null References and the Question of Philosophy

In Java, three special references exist: `null`, `this`, and `super`. (We'll defer descriptions of `this` and `super` to Chapter 6, Section 6.1.4.) The special literal `null` can be used to initialize reference variables. The `null` reference is typeless and thus can be assigned to any reference variable and passed to any method that awaits an object.¹¹



Example

The declaration and initialization of two object variables with `null` is shown in the following example:

Point p = null;
String s = null;
System.out.println(p); // null

The console output in the last line briefly returns “null,” which is actually a string representation of the `null` type.

Since `null` is typeless, and there's only one `null`, `null` can be type-matched to any type. Thus, for example, `(String) null == null && (Point) null == null` returns the result `true`. The `null` literal is intended for references only and can't be converted to any primitive type such as the integer `0`.¹²

A whole lot can be done with `null`. The main purpose is to indicate uninitialized reference variables (i.e., to express that a reference variable doesn't reference any object). In

¹¹ `null` thus behaves as though it were a subtype of any other type.
¹² C(++) and Java differ in this regard.

lists or trees, for example, `null` indicates the absence of a valid successor or, in a graphical dialog box, that the user has aborted the dialog process. In these cases, `null` is a valid indicator, not an error.

Note

For a local variable initialized with `null`, the shortcut with `var` does not work. A compiler error would arise with the following example:

var text = null; // ⚠ Cannot infer type: variable initializer is 'null'



Nothing Works on null except NullPointerException

Since `null` doesn't hide an object, you cannot call a method or query an object variable from `null`. The compiler knows the type of each expression, but only the runtime environment (i.e., the JVM) knows what's being referenced. When attempting to access a member of an object via the `null` reference, the JVM throws a `NullPointerException`.¹³ Consider the following example:

```
package com.tutego.insel.oop;           // 1
public class NullPointer {             // 2
    public static void main( String[] args ) { // 3
        java.awt.Point p = null;        // 4
        String s = null;                // 5
        p.setLocation( 1, 2 );          // 6
        s.length();                    // 7
    }                                   // 8
}                                       // 9
```

Listing 3.10 `src/main/java/com/tutego/insel/oop/NullPointer.java`

Notice how we have a `NullPointerException` at runtime because the program terminates at `p.setLocation(...)` with the following output:

```
Exception in thread "main" java.lang.NullPointerException: Cannot invoke "java.awt.Point.setLocation(int, int)" because "p" is null
at com.tutego.insel.oop.NullPointer.main(NullPointer.java:6)
```

In the error message, the runtime environment tells you that the error, the `NullPointerException`, is in line 6. To correct the error, you have two options. First, you must either initialize the variables, that is, assign an object as in the following example:

¹³ The name is reminiscent of pointers. Although we aren't dealing with pointers in Java, but with references, it's called `NullPointerException` and not `NullReferenceException`. This distinction is a reminder that a reference identifies an object, and a reference to an object is a pointer. The .NET framework is more consistent in this regard and calls the exception `NullReferenceException`.


```
p = new java.awt.Point();
s = "";
```

Alternatively, before accessing the members, you can perform a test to determine whether object variables point to something or are `null` and, depending on the outcome of the test, allow access to the member or not.



null in Other Programming Languages*

Is Java a purely OOP language? No, because Java distinguishes between primitive types and reference types. Let’s assume for a moment that primitive types don’t exist. Would Java then be a pure OOP language, where each reference references a pure object? The answer is still “no” because `null` is something that enables you to initialize reference variables, but doesn’t represent an object and has no methods. This scenario can cause a `NullPointerException` when de-referencing.

Other programming languages have different approaches to solving the problem, and `null` referencing isn’t possible. For example, in Ruby, everything is always an object. Where Java uses `null` to express “unassigned,” Ruby does so with `nil`. The subtle difference is that `nil` is an instance of the class `NilClass` (strictly speaking, a singleton that exists only once in the system). `nil` also has some public methods like `to_s` (like Java’s `toString()`), which then returns an empty string. With `nil`, no `NullPointerException` exists anymore, but of course, an error can still arise if a method is called on this object of type `NilClass` that doesn’t exist. In Objective-C, the standard language for iOS programs (so far), you have the null object `nil`. Usually, nothing happens when a message is sent to the `nil` object; the message is simply ignored.

3.7.2 Everything to null? Testing References

With the comparison operator `==` or the test for inequality via `!=`, you can easily determine whether a reference variable really references an object or not, for example, through the following code:

```
if ( object == null )
    // variable references nothing, but is correctly initialized with null
else
    // variable references an object
```

null Test and Short-Circuit Operators

At this point, let’s come back to the usual logical short-circuit operators and the logical non-short-circuit operators. The former evaluates operands only from left to right until the result of the operation is fixed. At first glance, whether all subexpressions are evaluated or not doesn’t seem to matter. In some expressions, however, this evaluation is important. In the following example, the variable `s` is of type `String`, and the program should output the string length when a string is entered:

```
String s = javax.swing.JOptionPane.showInputDialog( "Input a string" );
if ( s != null && ! s.isEmpty() )
    System.out.println( "Length of string: " + s.length() );
else
    System.out.println( "Dialog canceled or no input given" );
```

Listing 3.11 `src/main/java/NullCheck.java`, `main`

The return value of `showInputDialog(...)` is `null` if the user cancels the dialog. Our program should take this possibility into account. Therefore, the `if` condition tests whether `s` references an object at all and, if so, additionally whether the string is non-empty. After that evaluation follows an output.

This notation occurs frequently, and the AND operator for linking must be a short-circuit operator since it’s explicit in this case that the length is determined only if the `s` variable references a `String` object at all and is not `null`. Otherwise, you’d get a `NullPointerException` on `s.isEmpty()` if every subexpression was evaluated and `s` was `null`.



The Luck of Others: null Coalescing Operator*

Since `null` occurs far too often, but `null` references must be avoided, you may see a lot of code like `o != null ? o : non_null_o`. Various programming languages, including JavaScript, Kotlin, Objective-C, PHP, and Swift provide a shortcut for this construct called the *null coalescing operator*. Sometimes written as `??` or as `?:`, in our example, the null coalescing operator is written as `o ?? non_null_o`. This construct is especially nice with sequential tests of the type `o ?? p ?? q ?? r`, where it says something like, “Return the first non-null reference.” Java doesn’t provide such an operator.

3.7.3 Assignments with References

A reference allows access to a referenced object, and a reference variable stores a reference. Multiple reference variables may store the same reference, similar to an object being addressed under different names—just as a person might be addressed as “Boss” by her co-workers, she might be called “Honey” by her husband. This nicknaming is also referred to as an *alias*.



Example

Let’s say you want to address a point object under an alternative variable name. Consider the following example:

```
Point p = new Point();
Point q = p;
```

A point object is created and referenced with the variable `p`. The second line now stores the same reference in the variable `q`. After that, `p` and `q` reference the same object. For

better understanding, what's important here is how often `new` occurs because that tells you how many objects the JVM will create. Since only one `new` exists in the two lines, only one point is created.



If two object variables reference the same object, a natural consequence is that object states can be read and modified via two paths. If the same person is consistently called “Boss” in the company and “Honey” at home, everyone is happy.

Let's continue with our example using point objects. If `p` and `q` point to the same point object, changes via `p` can also be observed via the `q` variable, as in the following example.

```
public static void main( String[] args ) {
    Point p = new Point();
    Point q = p;
    p.x = 10;
    System.out.println( q.x ); // 10
    q.y = 5;
    System.out.println( p.y ); // 5
}
```

Listing 3.12 `ItsTheSame.java`, `main`

3.7.4 Methods with Reference Types as Parameters

The fact that the same object can be addressed via two names (i.e., via two different variables) can be observed in methods. A method that receives an object reference via the parameter can access the passed object. As a result, the method can change this object with the provided methods or access the object variables.

In the following example, we'll declare two methods. The first method, `initializePosition(Point)`, is supposed to initialize a given point with random coordinates. Two `Point`

objects are later passed to the method in `main(...)`: one for the player and one for the snake. The second method, `printScreen(Point, Point)`, prints the playing field on the screen and then prints a `&` when the coordinate hits a player and an `S` when it hits the snake. If the player and the snake happen to meet, the snake “wins.”

```
package com.tutego.insel.oop;
import java.awt.Point;

public class DrawPlayerAndSnake {

    static void initializePosition( Point p ) {
        int randomX = (int)(Math.random() * 40); // 0 <= x < 40
        int randomY = (int)(Math.random() * 10); // 0 <= y < 10
        p.setLocation( randomX, randomY );
    }

    static void printScreen( Point playerPosition,
                           Point snakePosition ) {
        for ( int y = 0; y < 10; y++ ) {
            for ( int x = 0; x < 40; x++ ) {
                if ( snakePosition.distanceSq( x, y ) == 0 )
                    System.out.print( 'S' );
                else if ( playerPosition.distanceSq( x, y ) == 0 )
                    System.out.print( '&' );
                else System.out.print( '.' );
            }
            System.out.println();
        }
    }

    public static void main( String[] args ) {
        Point playerPosition = new Point();
        Point snakePosition = new Point();
        System.out.println( playerPosition );
        System.out.println( snakePosition );
        initializePosition( playerPosition );
        initializePosition( snakePosition );
        System.out.println( playerPosition );
        System.out.println( snakePosition );
        printScreen( playerPosition, snakePosition );
    }
}
```

Listing 3.13 `src/main/java/com/tutego/insel/ooop/DrawPlayerAndSnake.java`

The code should produce the following output:

```
java.awt.Point[x=0,y=0]
java.awt.Point[x=0,y=0]
java.awt.Point[x=38,y=1]
java.awt.Point[x=19,y=8]
.....
.....&.....
.....
.....
.....
.....
.....
.....S.....
.....
```

The moment `main(...)` calls the static method `initializePosition(Point)`, we have two names for the `Point` object: `playerPosition` and `p`. However, this double naming is only inside the JVM because `initializePosition(Point)` knows the object only via `p` but doesn't know the `playerPosition` variable. With `main(...)`, the reverse is true: Only the variable name `playerPosition` is known in `main(...)`, but it has no idea about the name `p`. The `Point` method `distanceSq(int, int)` returns the squared distance from the current point to the passed coordinates.



Note

The name of a parameter variable may well be the same as the name of the argument variable, which wouldn't change the semantics. The namespaces are completely separate, and misunderstandings don't exist because both the calling method and the called method have completely separate local variables.

Value Transfer and Reference Transfer via Call by Value

Primitive variables are always copied by value (*call by value*). The same applies to references, which are to be understood as a kind of pointer, which are basically only integers. For this reason, the following static method has no side effects:

```
package com.tutego.insel.oop;
import java.awt.Point;

public class JavaIsAlwaysCallByValue {

    static void clear( Point p ) {
        System.out.println( p ); // java.awt.Point[x=10,y=20]
```

```
p = new Point();
System.out.println( p ); // java.awt.Point[x=0,y=0]
}

public static void main( String[] args ) {
    Point p = new Point( 10, 20 );
    clear( p );
    System.out.println( p ); // java.awt.Point[x=10,y=20]
}
}
```

Listing 3.14 `JavalsAlwaysCallByValue.java`

After assigning `p = new Point()` in the `clear(Point)` method, the parameter variable `p` references another point object, and the reference passed to the method is thus lost. Of course, this change isn't visible from outside because the parameter variable `p` of `clear(...)` is only a temporary alternative name for the `p` from `main`; a reassignment to the `clear-p` doesn't change the reference from the `main-p`. As a result, the caller of `clear(...)`, which is `main(...)`, has no new object under it. If you want to initialize the point with null, you must access the states of the passed object directly, for instance, in the following way:

```
static void clear( Point p ) {
    p.x = p.y = 0;
}
```



Call by Reference Doesn't Exist in Java: A Look at C and C++*

In C++ , another way to pass arguments is called *call by reference*. A `swap(...)` function is a good example of the usefulness of call by reference:

```
void swap( int& a, int& b ) { int tmp = a; a = b; b = tmp; }
```

Pointers and references are something different in C++, which easily confuses beginners to the language. In C++ and also in C, a comparable `swap(...)` function could also have been implemented with pointers:

```
void swap( int *a, int *b ) { int tmp = *a; *a = *b; *b = tmp; }
```

The implementation provides a reference to the argument in C(++).

Final Declared Reference Parameter and the Missing const

As we've seen, final variables tell a programmer that variables must not be rewritten. Local variables, parameter variables, object variables, and class variables can be final. In any case, new assignments are taboo. Whether the parameter variable is of a primitive

type or a reference type does not matter. Thus, with a method declaration of the following type, an assignment to `p` and also to `value` would be forbidden:

```
public void clear( final Point p, final int value )
```

If the parameter variable isn't `final` and is a reference type, we'd lose the reference to the original object with an assignment, which would make little sense, as we saw in the previous example. Parameter variables declared `final` make it clear in the program code that changing the reference variable doesn't make any sense, and the compiler forbids an assignment. In the case of our `clear(...)` method, the initialization would have been noticed directly as a compiler error. Consider the following example:

```
static void clear( final Point p ) {
    p = new Point();    // ☠ Cannot assign a value to final variable 'p'
}
```

Let's recap: If a parameter is declared `final`, no assignments are possible. However, `final` doesn't prohibit changes to objects, and therefore, `final` could be understood as "definitive." With the reference of the object, you can very well change the state, as we did in the previous sample program.

Therefore, `final` doesn't fulfill the task of preventing write access to objects. A method with passed references can therefore modify objects if, for example, `set*(...)` methods or variables can be accessed. Thus, the documentation must always explicitly describe when the method modifies the state of an object.

In C++, the addition `const` for parameters enables the compiler to recognize that object states shouldn't be changed. A program is called *const-correct* if it never modifies a constant object. This `const` is an extension of the object type in C++, which doesn't exist in Java. Although Java's developers have reserved the `const` keyword, it's not used yet.

3.7.5 Identity of Objects

The comparison operators `==` and `!=` are defined for all data types in such a way that they test the complete correspondence of two values. With primitive data types, this correspondence is easy to see, and with reference types, it's basically the same. (Remember: references can be understood as pointers, which are integers.) The `==` operator tests references to see if they match (i.e., reference the same object). The `!=` operator tests the opposite (i.e., whether they do not match), so the references aren't equal. Accordingly, the test says something about the identity of the referenced objects, but nothing about whether two different objects may have the same content. The content of the objects doesn't matter for `==` and `!=`.

Example

The following example shows two objects with three different point variables (`p`, `q`, and `r`) and illustrates the meaning of `==`:

```
Point p = new Point( 10, 10 );
Point q = p;
Point r = new Point( 10, 10 );
System.out.println( p == q ); // true, because p and q reference the
                               // same object
System.out.println( p == r ); // false, because p and r reference two
                               // different point objects that happen to have
                               // the same coordinates
```

Since `p` and `q` reference the same object, the comparison returns `true`. `p` and `r` reference different objects, but they happen to have the same content. But how is the compiler supposed to know when two point objects are equal in content? Is it because a point is characterized by the object variables `x` and `y`? The runtime environment might hastily compare the assignment of each object variable, but this approach doesn't always correspond to the correct comparison we desire. For example, a point object could additionally record the number of method calls, which may not be taken into account in a comparison based on the location of two points.

3.7.6 Equivalence and the `equals(...)` Method

The universal solution is to let the class determine when objects are equal (i.e., have the same value). For this purpose, each class can implement a method called `equals(...)`, and with its help, each instance of this class can compare itself with any other object. The classes always decide, according to the use case, which object variables they refer to for an equality test, and `equals(...)` returns `true` if the desired states (object variables) match.

Example

Two non-identical point objects with the same content can be compared via `==` and `equals(...)`, as in the following example:

```
Point p = new Point( 10, 10 );
Point q = new Point( 10, 10 );
System.out.println( p == q );    // false
System.out.println( p.equals(q) ); // true, since symmetrically also
                                   // q.equals(p)
```

Only `equals(...)` tests content equivalence in this case.

[Ex]

[Ex]

Accordingly, due to the different meanings, we must carefully distinguish the concepts of *identity* and *equivalence* (also *equality*) of objects. For this reason, Table 3.4 shows a summary.

	Tested with	Implementation
Identity of the references	== or !=	Nothing to do
Equivalence of states	equals(...) or ! equals(...)	Depending on the class

Table 3.4 Identity and Equivamlence of Objects

equals(...) Implementation of Point*

The Point class declares equals(...), as described in the API documentation. Let’s look at an implementation for an idea of how it works next.

```
public class Point ... {

    public int x;
    public int y;
    ...
    public boolean equals( Object obj ) {
        ...
        Point pt = (Point) obj;
        return (x == pt.x) && (y == pt.y);    // (*)
        ...
    }
}
```

Listing 3.15 java/awt/Point.java (Snippet)

Although some things are new in this example, we recognize the comparison in line (*). In this case, the Point object compares its own object variables with the object variables of the point object passed as argument to equals(...).

There’s Always an equals(...): The Object Superclass and Its Equals*

Every class has a equals(...) method because of the universal superclass Object (see Chapter 7; for details). Thus, if a class doesn’t specify its own equals(...) method, it inherits an implementation from the Object class, as in the following example:

```
public class Object {
    public boolean equals( Object obj ) {
        return ( this == obj );
    }
}
```

```
}
...
}
```

Listing 3.16 java/lang/Object.java (Snippet)

In this case, the equivalence is mapped to the identity of the references, but no comparison of content occurs. This equivalence is the only thing the given implementation can do because, if the references are identical, the objects are naturally the same. The only thing is that the base class Object doesn’t “know” anything about is states.

Language Comparison

Programming languages generally provide their own operators for identity comparison and equivalence testing. == and equals(...) in Java is analogous to is and == in Python and === and == in Swift.



3.8 Further Reading

In this chapter, the topic of object orientation was introduced rather quickly, which doesn’t mean that OOP is easy. The road to good design is rocky and leads through many Java projects. Reading other programs and studying design patterns can be immensely helpful. Readers should also get comfortable with UML to sketch design ideas. An interesting approach is taken by PlantUML (<https://plantuml.com/>) with a text syntax that the tool converts into graphics.

Chapter 16

The Class Library

*“What we need is some crazy people;
look where the normal ones have taken us.”
—George Bernard Shaw (1856–1950)*

In this chapter you’ll learn about the Java class library.

16.1 The Java Class Philosophy

A programming language consists not only of a grammar, but also, as in the case of Java, of a programming library. A platform-independent language—as many imagine C or C++ to be—isn’t really platform independent if different functions and programming models are used on each computer, which is exactly the weak point of C(++). These algorithms, which aren’t dependent on the operating system, can be applied everywhere in the same way, but the result is realized in the end with inputs/outputs or graphical user interfaces (GUIs). The Java library, on the other hand, tries to abstract away from platform-specific features, and the developers have gone to great lengths to put all the important methods into well-formed object-oriented (OO) classes and packages. These elements cover in particular the central areas of data structures, input and output, graphics, and network programming.



16.1.1 Modules, Packages, and Types

At the top of the Java library are modules, which in turn consist of packages, which in turn contain types.

Modules of the Java SE

The *Java Platform, Standard Edition (Java SE)* application programming interface (API) consists of the following modules, all of which begin with the prefix `java`:

Module	Description
<code>java.base</code>	Fundamental types of Java SE
<code>java.compiler</code>	Java language model, annotation processing, and Java compiler API
<code>java.datatransfer</code>	The API for data transfer between applications, usually the clipboard
<code>java.desktop</code>	GUIs with Abstract Windowing Toolkit (AWT) and Swing, the <i>Accessibility</i> API, audio, printing, and JavaBeans
<code>java.instrument</code>	Instrumentalization is the modification of Java programs at run-time
<code>java.logging</code>	The <i>Logging</i> API
<code>java.management</code>	<i>Java Management Extensions (JMX)</i>
<code>java.management.rmi</code>	<i>Remote Method Invocation (RMI)</i> connector for remote access to the JMX beans
<code>java.naming</code>	The <i>Java Naming and Directory Interface (JNDI)</i> API
<code>java.prefs</code>	The <i>Preferences</i> API is used to store user preferences
<code>java.rmi</code>	Remote method calls with the RMI API
<code>java.scripting</code>	The <i>Scripting</i> API
<code>java.security.jgss</code>	Java binding of the <i>IETF Generic Security Services</i> API (GSS API)
<code>java.security.sasl</code>	Java support for <i>IETF Simple Authentication and Security Layer (SASL)</i>
<code>java.sql</code>	The JDBC API for accessing relational databases
<code>java.sql.rowset</code>	The <i>JDBC RowSet</i> API
<code>java.xml</code>	XML classes with the <i>Java API for XML Processing (JAXP)</i> , <i>Streaming API for XML (StAX)</i> , <i>Simple API for XML (SAX)</i> , and <i>W3C Document Object Model (DOM)</i> API
<code>java.xml.crypto</code>	The API for XML cryptography

Table 16.1 Modules of the Java SE

The `java.base` module—the most important module—contains core classes such as `Object` and `String`, among others. This module is the only module that doesn’t itself contain any dependency on other modules. Every other module, however, references at least `java.base`. The Javadoc contains a nice graphical representation, shown in Figure 16.1.

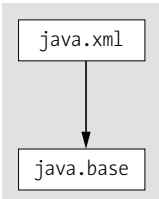


Figure 16.1 The `java.xml` Module Has a Dependency on the `java.base` Module

In some cases, more dependencies exist, such as with the `java.desktop` module, shown in Figure 16.2.

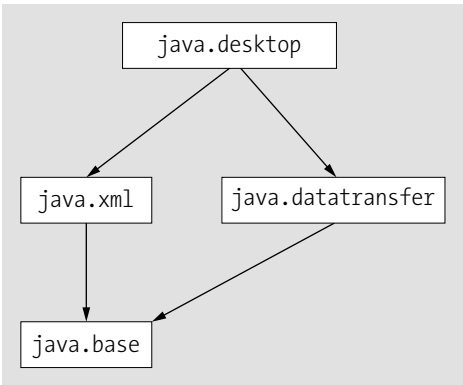


Figure 16.2 Dependencies of the `java.desktop` Module

The `java.se` Module

One special module is `java.se`, which doesn’t declare its own packages or types but merely groups other modules together. The name for such a construction is *aggregator module*. The `java.se` module defines the API for the Java SE platform in this way, as shown in Figure 16.3.

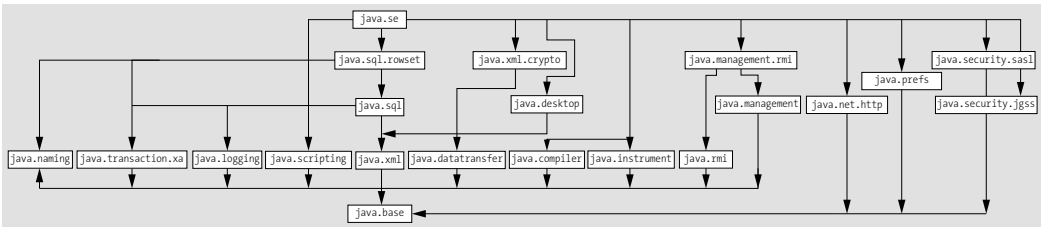


Figure 16.3 Dependencies of the `java.se` Module



Note

In the following sections, we won't discuss the Java SE types in terms of the module from which they originate. You only need to know in which module a type is located when building smaller subsets of Java SE.

Other Modules

Two other modules that also start with `java` but aren't part of the Java SE standard are `java.jnlp` for the Java Network Launch Protocol (JNLP) and `java.smartcardio`, which is the Java API for communication with smart cards according to the international standard ISO/IEC 7816-4.¹

The Java Development Kit (JDK) is the standard implementation of Java SE. This implementation provides developers with more packages and classes, such as with an HTTP server or with Java tools like the Java compiler and the Javadoc tool. In this implementation, several modules will start with the prefix `jdk`.

16.1.2 Overview of the Packages of the Standard Library

The *Java 11 Core Java SE API* consists of the following modules and packages:

Modules	Packages Included
<code>java.base</code>	<code>java.io</code> , <code>java.lang</code> , <code>java.lang.annotation</code> , <code>java.lang.invoke</code> , <code>java.lang.module</code> , <code>java.lang.ref</code> , <code>java.lang.reflect</code> , <code>java.math</code> , <code>java.net</code> , <code>java.net.spi</code> , <code>java.nio</code> , <code>java.nio.channels</code> , <code>java.nio.channels.spi</code> , <code>java.nio.charset</code> , <code>java.nio.charset.spi</code> , <code>java.nio.file</code> , <code>java.nio.file.attribute</code> , <code>java.nio.file.spi</code> , <code>java.security</code> , <code>java.security.cert</code> , <code>java.security.interfaces</code> , <code>java.security.spec</code> , <code>java.text</code> , <code>java.text.spi</code> , <code>java.time</code> , <code>java.time.chrono</code> , <code>java.time.format</code> , <code>java.time.temporal</code> , <code>java.time.zone</code> , <code>java.util</code> , <code>java.util.concurrent</code> ,
<code>java.base</code>	<code>java.util.concurrent.atomic</code> , <code>java.util.concurrent.locks</code> , <code>java.util.function</code> , <code>java.util.jar</code> , <code>java.util.regex</code> , <code>java.util.spi</code> , <code>java.util.stream</code> , <code>java.util.zip</code> , <code>javax.crypto</code> , <code>javax.crypto.interfaces</code> , <code>javax.crypto.spec</code> , <code>javax.net</code> , <code>javax.net.ssl</code> , <code>javax.security.auth</code> , <code>javax.security.auth.callback</code> , <code>javax.security.auth.login</code> , <code>javax.security.auth.spi</code> , <code>javax.security.auth.x500</code> , <code>javax.security.cert</code>

Table 16.2 Packages in the Modules of the Java 17 Core Java SE API

¹ https://en.wikipedia.org/wiki/ISO/IEC_7816

Modules	Packages Included
<code>java.compiler</code>	<code>javax.annotation.processing</code> , <code>javax.lang.model</code> , <code>javax.lang.model.element</code> , <code>javax.lang.model.type</code> , <code>javax.lang.model.util</code> , <code>javax.tools</code>
<code>java.datatransfer</code>	<code>java.awt.datatransfer</code>
<code>java.desktop</code>	<code>java.applet</code> , <code>java.awt</code> , <code>java.awt.color</code> , <code>java.awt.desktop</code> , <code>java.awt.dnd</code> , <code>java.awt.event</code> , <code>java.awt.font</code> , <code>java.awt.geom</code> , <code>java.awt.im</code> , <code>java.awt.im.spi</code> , <code>java.awt.image</code> , <code>java.awt.image.renderable</code> , <code>java.awt.print</code> , <code>java.beans</code> , <code>java.beans.beancontext</code> , <code>javax.accessibility</code> , <code>javax.imageio</code> , <code>javax.imageio.event</code> , <code>javax.imageio.metadata</code> , <code>javax.imageio.plugins.bmp</code> , <code>javax.imageio.plugins.jpeg</code> , <code>javax.imageio.plugins.tiff</code> , <code>javax.imageio.spi</code> , <code>javax.imageio.stream</code> , <code>javax.print</code> , <code>javax.print.attribute</code> , <code>javax.print.attribute.standard</code> , <code>javax.print.event</code> , <code>javax.sound.midi</code> , <code>javax.sound.midi.spi</code> , <code>javax.sound.sampled</code> , <code>javax.sound.sampled.spi</code> , <code>javax.swing</code> , <code>javax.swing.border</code> , <code>javax.swing.colorchooser</code> , <code>javax.swing.event</code> , <code>javax.swing.filechooser</code> , <code>javax.swing.plaf</code> , <code>javax.swing.plaf.basic</code> , <code>javax.swing.plaf.metal</code> , <code>javax.swing.plaf.multi</code> , <code>javax.swing.plaf.nimbus</code> , <code>javax.swing.plaf.synth</code> , <code>javax.swing.table</code> , <code>javax.swing.text</code> , <code>javax.swing.text.html</code> , <code>javax.swing.text.html.parser</code> , <code>javax.swing.text.rtf</code> , <code>javax.swing.tree</code> , <code>javax.swing.undo</code>
<code>java.instrument</code>	<code>java.lang.instrument</code>
<code>java.logging</code>	<code>java.util.logging</code>
<code>java.management</code>	<code>java.lang.management</code> , <code>javax.management</code> , <code>javax.management.loading</code> , <code>javax.management.modelmbean</code> , <code>javax.management.monitor</code> , <code>javax.management.openmbean</code> , <code>javax.management.relation</code> , <code>javax.management.remote</code> , <code>javax.management.timer</code>
<code>java.management.rmi</code>	<code>javax.management.remote.rmi</code>
<code>java.naming</code>	<code>javax.naming</code> , <code>javax.naming.directory</code> , <code>javax.naming.event</code> , <code>javax.naming.ldap</code> , <code>javax.naming.spi</code>
<code>java.prefs</code>	<code>java.util.prefs</code>
<code>java.rmi</code>	<code>java.rmi</code> , <code>java.rmi.activation</code> , <code>java.rmi.dgc</code> , <code>java.rmi.registry</code> , <code>java.rmi.server</code> , <code>javax.rmi.ssl</code>

Table 16.2 Packages in the Modules of the Java 17 Core Java SE API (Cont.)

Modules	Packages Included
java.scripting	javax.script
java.security.jgss	javax.security.auth.kerberos,org.ietf.jgss
java.security.sasl	javax.security.sasl
java.sql	java.sql,javax.sql,javax.transaction.xa
java.sql.rowset	javax.sql.rowset,javax.sql.rowset.serial,javax.sql.rowset.spi
java.xml	javax.xml,javax.xml.catalog,javax.xml.datatype,javax.xml.namespace,javax.xml.parsersjavax.xml.stream,javax.xml.stream.events,javax.xml.stream.util,javax.xml.transform,javax.xml.transform.dom,javax.xml.transform.sax,javax.xml.transform.stax,javax.xml.transform.stream,javax.xml.validation,javax.xml.xpath,org.w3c.dom,org.w3c.dom.bootstrap,org.w3c.dom.events,org.w3c.dom.ls,org.w3c.dom.ranges,org.w3c.dom.views,org.xml.sax,org.xml.sax.ext,org.xml.sax.helpers
java.xml.crypto	javax.xml.crypto,javax.xml.crypto.dom,javax.xml.crypto.dsig,javax.xml.crypto.dsig.dom,javax.xml.crypto.dsig.keyinfo,javax.xml.crypto.dsig.spec

Table 16.2 Packages in the Modules of the Java 17 Core Java SE API (Cont.)

Developers should be able to map the following packages according to their respective capabilities:

Package	Description
java.awt	The AWT package provides classes for graphics output and GUI usage.
java.awt.event	Interfaces for the various events in GUIs.
java.io java.nio	Input and output options. Files are represented as objects. Data streams allow sequential access to file contents.
java.lang	A package that’s automatically included. Contains indispensable classes like string, thread, or wrapper classes.
java.net	Communication via networks. Provides classes for building client and server systems that can connect to the internet via TCP and IP, respectively.

Table 16.3 Important Packages in the Java SE

Package	Description
java.text	Support for internationalized programs. Provides classes for handling text and formatting dates and numbers.
java.util	Provides types for data structures, space and time, and parts of internationalization, as well as random numbers. Subpackages take care of regular expressions and concurrency.
javax.swing	Swing components for GUIs. This package has various subpackages.

Table 16.3 Important Packages in the Java SE (Cont.)

For a developer, you can’t avoid studying the Java API documentation at <https://docs.oracle.com/en/java/javase/17/docs/api/index.html>.

Official Interface (java and javax Packages)

The list provided by the Java documentation represents the permitted access to the library. The types are basically designed to last forever, so developers can count on still being able to run their Java programs in 100 years. But who defines the API? In essence, three sources define APIs:

- Oracle developers put new packages and types into the API.
- The *Java Community Process (JCP)* adopts a new API. Then, Oracle is not acting alone, but instead, a group works out a new API and defines its interfaces.
- The *World Wide Web Consortium (W3C)* provides an API for XML Document Object Model (DOM), for example.

A good mnemonic is that anything starting with java or javax is a permitted API, and anything else can lead to non-portable Java programs. Some classes are supported that aren’t part of the official API. These classes include, for example, various Swing classes for controlling the appearance of the interface.

Standard Extension API (javax Packages)

Some Java packages start with javax. Originally, these extension packages were intended to complement the core classes. Over time, however, many packages that initially had to be included have now migrated to the standard distribution, so that today, a fairly large proportion start with javax, but no longer represent extensions that need to be additionally installed. Sun didn’t want to rename the packages at that time, so as not to make migration more difficult. If you notice a package name with javax in the source code today, therefore, you can no longer easily determine whether an external source must be included or whether the package is already part of the distribution (and since Java version).

Truly external packages include the following packages:

- The *Java Communications API* for serial and parallel interfaces
- The *Java Telephony API*
- Speech input/output with the *Java Speech API*
- *JavaSpaces* for shared memory of different runtime environments
- *JXTA* for establishing P2P networks

The bottom line is that developers are dealing with the following libraries:

- With the official Java API
- With APIs from Java Specification Request (JSR) extensions
- With unofficial libraries, such as open-source solutions, for example, to access PDF files or control ATMs

An important role is also played by types from the `jakarta` package, which is part of Jakarta EE (formerly Java EE) and semi-official.

16.2 Simple Time Measurement and Profiling*

In addition to the convenient classes for managing date values, two static methods provide simple ways to measure times for program sections:

```
final class java.lang.System
```

- ▶ `static long currentTimeMillis()`
Returns the milliseconds elapsed since 1/1/1970, 00:00:00 Coordinated Universal Time (UTC).
- ▶ `static long nanoTime()`
Returns the time from the most accurate system timer. This method has no reference point to any date.

The difference between two time values can be used to roughly estimate the execution times of programs.



Tip

The values of `nanoTime()` are always ascending, which isn't necessarily true for `currentTimeMillis()` because Java gets the time from the operating system. System times can change, for example, when a user adjusts the time. Differences of `currentTimeMillis()` timestamps are then completely wrong and could even be negative.

16.2.1 Profilers

Where the Java virtual machine (JVM) does waste clock cycles in a program is shown by a *profiler*. Optimization can then begin at those points. *Java Mission Control* is a powerful program of the JDK and integrates a free profiler. *Java VisualVM* is another free program that can be obtained from <https://visualvm.github.io/>. On the professional and commercial side, *JProfiler* (<https://www.ej-technologies.com/products/jprofiler/overview.html>) and *YourKit* (<https://www.yourkit.com/java/profiler>) are competitors. The *Ultimate Version* of IntelliJ also includes a profiler.

16.3 The Class Class

Let's suppose we want to write a class browser. This program should display all classes belonging to the running program and furthermore additional information, such as variable assignment, declared methods, constructors, and some information about the inheritance hierarchy. For this purpose, you'll need the library class, `Class`. Instances of `Class` are objects that, for example, represent a Java class, record or a Java interface.

In this respect, Java differs from many conventional programming languages because the members of classes can be queried by the currently running program using the `Class` objects. The instances of `Class` are a restricted kind of meta-object²—containing the description of a Java type but revealing only selected information. Besides normal classes, interfaces are also represented by a `Class` object, and even arrays and primitive data types—instead of `Class`, the class name `Type` would probably have been more appropriate.

16.3.1 Obtaining a Class Object

First, for a given class, you must identify the associated `Class` object. `Class` objects themselves can only be created by the JVM. (We can't create instances because the constructor of `Class` is private.) To obtain a reference to a `Class` object, the following solutions are available:

- If an instance of the class is available, you can call the `getClass()` method of the object and get the `Class` instance of the associated class.
- Each type contains a static variable named `.class` of type `Class`, which references the associated `Class` instance.
- The ending `.class` is also permitted for primitive data types. The same `Class` object returns the static variable `TYPE` of the wrapper classes. Thus, `int.class == Integer.TYPE` is true.

² True metaclasses are classes whose only instance in each case is the regular Java class. Then, for example, the regular class variables would actually be object variables in the metaclass.

- The class method `Class.forName(String)` can query a class, and you'll obtain the associated `Class` instance as a result. If the type hasn't been loaded yet, `forName(String)` searches for and binds the class. Because searching can go wrong, a `ClassNotFoundException` is possible.
- If you already have a `Class` object but are interested in its ancestors instead, you can simply get a `Class` object for the superclass via `getSuperclass()`.

The following example shows three ways to obtain a `Class` object for `java.util.Date`:

```
Class<Date> c1 = java.util.Date.class;
System.out.println( c1 );           // class java.util.Date
Class<?> c2 = new java.util.Date().getClass();
// or Class<? extends Date> c2 = ...

System.out.println( c2 );           // class java.util.Date
try {
    Class<?> c3 = Class.forName( "java.util.Date" );
    System.out.println( c3 );       // class java.util.Date
}
catch ( ClassNotFoundException e ) { e.printStackTrace(); }
```

Listing 16.1 `src/main/java/com/tutego/insel/meta/GetClassObject.java, main()`

The variant with `forName(String)` is useful if the name of the desired class wasn't determined when the program was translated.

Otherwise, the previous technique is more catchy, and the compiler can check if the type exists. A full qualification is needed: `Class.forName("Date")` would only search for `Date` in the default package, and the return isn't a collection after all.



Example

Note that class objects for primitive elements aren't returned by `forName(String)`. The two expressions `Class.forName("boolean")` and `Class.forName(boolean.class.getName())` lead to a `ClassNotFoundException`.

```
class java.lang.Object
```

- `final Class<? extends Object> getClass()`
Returns the `Class` instance at runtime which represents the class of the object.

```
final class java.lang.Class<T>
implements Serializable, GenericDeclaration, Type, AnnotatedElement
```

- `static Class<?> forName(String className)` throws `ClassNotFoundException`
Returns the `Class` instance for the class, record or interface with the specified fully qualified name. If the type hasn't yet been required by the program, the class loader searches for and loads the class. The method never returns `null`. If the class couldn't be loaded and included, a `ClassNotFoundException` will occur. The alternative method, `forName(String name, boolean initialize, ClassLoader loader)`, also allows loading with a desired class loader. The class name must always be fully qualified.

`ClassNotFoundException` and `NoClassDefFoundError`*

A `ClassNotFoundException` can be thrown by any of the following methods:

- `forName(...)` from `Class`
- `loadClass(String name [, boolean resolve])` from `ClassLoader`
- `findSystemClass(String name)` from `ClassLoader`

An exception occurs whenever the class loader can't find the class by its class name. Thus, the trigger is when an application wants to load types dynamically, but those types aren't present.

In addition to `ClassNotFoundException`, the `NoClassDefFoundError` is a hard `LinkageError` that the JVM raises whenever it can't load a class referenced in the bytecode. For example, let's consider an expression like `new MyClass()`. When the JVM executes this code, it attempts to load the bytecode from `MyClass`. If the bytecode for `MyClass` has been removed after compilation, the JVM raises the `NoClassDefFoundError` due to the unsuccessful load attempt. Also, the error occurs if the `MyClass` class was found when loading the bytecode, but `MyClass` has a static initialization block that in turn references a class for which no class file exists.

While `ClassNotFoundException` is more common than `NoClassDefFoundError`, the exception is generally an indication that a Java Archive file (JAR file) is missing in the module path.

Problems after Applying an Obfuscator*

The fact that the compiler automatically generates bytecode according to this modified source code only leads to unexpected problems if you run an obfuscator over the program text, which subsequently modifies the bytecode and thus obscures the meaning of the program or the bytecode and renames types in the process. Obviously, an obfuscator must not rename types whose `Class` instances are requested.

Otherwise, the obfuscator must correctly replace the corresponding strings as well (but of course not replace all strings that happen to match class names).

16.3.2 A Class Is a Type

In Java, different types exist, and classes, records, interfaces, and enumeration types are represented by the JVM as `Class` objects. In the Reflection API, the `Type` interface represents all types and the only implementing class is `Class`. Below `Type` there are some subinterfaces:

- `ParameterizedType` represents generic types like `List<T>`.
- `TypeVariable<D>` represents, for example, `T` extends `Comparable<? super T>`.
- `WildcardType` represents `? super T`.
- `GenericArrayType` represents something like `T[]`.

The only method of `Type` is `getTypeName()`, and this method is just a default method that calls `toString()`. `Type` is the return of various methods in the Reflection API, such as `getGenericSuperclass()` and `getGenericInterfaces()` of the `Class` class, and many other methods listed in the Javadoc under **USE**.

16.4 The Utility Classes System and Members

In the `java.lang.System` class, methods exist for requesting and changing system variables, for redirecting the standard data streams, for determining the current time, for terminating the application, and for several other tasks. All methods are exclusively static, and an instance of `System` can't be created. In the `java.lang.Runtime` class, additional helper methods are available, such as for starting external programs or for requesting memory requirements. Unlike `System`, only one method is static in this class, namely, the singleton method `getRuntime()`, which returns the instance of `Runtime`.

java.lang.System	java.lang.Runtime
<div>+ err : PrintStream + in : InputStream + out : PrintStream + arraycopy(src : Object, srcPos : int, dest : Object, destPos : int, length : int) + clearProperty(key : String) : String + console() : Console + currentTimeMillis() : long + exit(status : int) + gc() + getProperties() : Properties + getProperty(key : String, def : String) : String + getProperty(key : String) : String + getSecurityManager() : SecurityManager + getenv(name : String) : String + getenv() : Map + hashCode(x : Object) : int + inheritedChannel() : Channel + load(filename : String) + loadLibrary(libname : String) + mapLibraryName(libname : String) : String + nanoTime() : long + runFinalization() + runFinalizersOnExit(Value : boolean) + setErr(err : PrintStream) + setIn(in : InputStream) + setOut(out : PrintStream) + setProperties(props : Properties) + setProperty(key : String, value : String) : String + setSecurityManager(s : SecurityManager)</div>	<div>+ addShutdownHook(hook : Thread) + availableProcessors() : int + exec(cmdarray : String[], envp : String[], dir : File) : Process + exec(command : String, envp : String[], dir File) : Process + exec(command : String) : Process + exec(cmdarray : String[]) : Process + exec(cmdarray : String[], envp : String[]) : Process + exec(command : String, envp : String[]) : Process + exit(status : int) + freeMemory() : long + gc() + getLocalizedInputStream(in : InputStream) : InputStream + getLocalizedOutputStream(out : OutputStream) : OutputStream + getRuntime() : Runtime + halt(status : int) + load(filename : String) + loadLibrary(libname : String) + maxMemory() : long + removeShutdownHook(hook : Thread) : boolean + runFinalization() + runFinalizersOnExit(value : boolean) + totalMemory() : long + traceInstructions(on : boolean) + traceMethodCalls(on : boolean)</div>

Figure 16.4 Members of the System and Runtime Classes

Remark

All in all, the `System` and `Runtime` classes don't seem particularly orderly, as shown in Figure 16.4; you may think everything that doesn't fit elsewhere can be found in these two classes. Also, some methods of one class would be just as good in the other class. The fact that the static method `System.arraycopy(...)` for copying arrays isn't located in `java.util.Arrays` can only be explained historically. Furthermore, `System.exit(int)` redirects to `Runtime.getRuntime().exit(int)`. Some methods are obsolete and distributed differently: The `exec(...)` of `Runtime` to start external processes is handled by a new class (`ProcessBuilder`), and the question about the memory state or the number of processors is answered by `MBeans`, such as `ManagementFactory.getOperatingSystemMXBean().getAvailableProcessors()`. But API design is like gambling: One rash action, and you've lost the farm.

16.4.1 Memory of the Java Virtual Machine

The `Runtime` object includes the following three methods that provide information about the memory of the JVM:

- `maxMemory()` returns the maximum number of bytes available for the JVM. The value can be set when calling the JVM with `-Xmx` in the command line.
- `totalMemory()` is what is currently used and can grow to `maxMemory()`. Basically, this memory limit can also shrink again. The following applies: `maxMemory() > totalMemory()`.
- `freeMemory()` indicates the memory that is free for new objects and also provokes the automatic garbage collection process. The following applies: `totalMemory() > freeMemory()`. However, `freeMemory()` isn't the entire freely available memory area because the "share" of `maxMemory()` is still missing.

Two pieces of information are missing and therefore must be calculated:

- **Used memory:**
`long usedMemory = Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory();`
- **Free total memory:**
`long totalFreeMemory = Runtime.getRuntime().maxMemory() - usedMemory;`

Example

The following example outputs information about the memory on a computer:

```
long totalMemory    = Runtime.getRuntime().totalMemory();
long freeMemory     = Runtime.getRuntime().freeMemory();
long maxMemory      = Runtime.getRuntime().maxMemory();
long usedMemory     = totalMemory - freeMemory;
```



```
long totalFreeMemory = maxMemory - usedMemory;

System.out.printf(
    "total=%d MiB, free=%d MiB, max=%d MiB, used=%d MiB, total free=%d MiB%n",
    totalMemory >> 20, freeMemory >> 20, maxMemory >> 20,
    usedMemory >> 20, totalFreeMemory >> 20 );

The result is the following output:

total=126 MiB, free=124 MiB, max=2016 MiB, used=1 MiB, total free=2014 MiB
```

16.4.2 Number of CPUs or Cores

The Runtime method availableProcessors() returns the number of logical processors or cores.

[Ex]

Example

Print the number of processors/cores:

```
System.out.println( Runtime.getRuntime().availableProcessors() ); // 4
```

16.4.3 System Properties of the Java Environment

The Java environment manages system properties, such as path separators or virtual machine versions in the java.util.Properties object. The static method System.getProperties() queries these system properties and returns the filled Properties object. However, the Properties object isn't absolutely necessary for querying individual properties: System.getProperty(...) directly queries a property.

[Ex]

Example

The following example outputs the name of the operating system:

```
System.out.println( System.getProperty( "os.name" ) ); // e.g., Windows 10
```

The following example outputs all system properties on the screen:

```
System.getProperties().list( System.out );
```

An excerpt of the result is shown in the following output:

```
-- listing properties --
sun.desktop=windows
awt.toolkit=sun.awt.windows.WToolkit
java.specification.version=9
file.encoding.pkg=sun.io
sun.cpu.isalist=amd64
...
```

Table 16.4 shows a list of the important standard system properties.

Key	Meaning
java.version	Version of the Java Runtime Environment (JRE)
java.class.path	The current classpath
java.library.path	Path for native libraries
java.io.tmpdir	Path for temporary files
os.name	Name of the operating system
file.separator	Separator of the path segments, for example / (Unix) or \ (Windows)
path.separator	Separator for path specifications, such as : (Unix) or ; (Windows)
line.separator	Newline character (string)
user.name	Name of the logged-on user
user.home	Home directory of the user
user.dir	Current directory of the user

Table 16.4 Standard System Properties

Application Programming Interface Documentation

A few more keys are listed in the API documentation at System.getProperties(). Some variables are also accessible in other ways, such as through the File class.

```
final class java.lang.System
```

- ▶ static String getProperty(String key)
Returns the assignment of a system property. If the key is null or empty, a NullPointerException or an IllegalArgumentException, respectively, will occur.
- ▶ static String getProperty(String key, String def)
Returns the assignment of a system property. If the property isn't present, the method returns the string def, which is the default value. For the exceptions, the same applies as for getProperty(String).
- ▶ static String setProperty(String key, String value)
Reassigns a system property. The return is the previous assignment or null if no previous assignment exists.
- ▶ static String clearProperty(String key)
Deletes a system property from the list. The return is the previous assignment or null if no previous assignment exists.

► `static Properties getProperties()`

Returns a `Properties` object filled with the current system assignments.

16.4.4 Setting Custom Properties from the Console*

Properties can also be set from the console during the program startup. This approach is convenient for a configuration that controls, for example, the behavior of a program. On the command line, `-D` specifies the name of the property and, after an equal sign (without whitespace), its value. Consider the following example:

```
$ java -DLOG -DUSER=Chris -DSIZE=100 com.tutego.insel.lang.SetProperty
```

The `LOG` property is “simply exists” but with no assigned value. The next two properties, `USER` and `SIZE`, are associated with values that are first of type `String` and must be further processed by the program. This information doesn’t appear with the argument list in the static `main(String[])` method because it precedes the name of the class and is already processed by the JRE.

To read the properties, we’ll use the familiar `System.getProperty(...)` method in the following example:

```
Optional<String> logProperty = ofNullable( System.getProperty( "LOG" ) );
Optional<String> user nameProperty = ofNullable( System.getProperty( "USER" ) );
Optional<String> sizeProperty = ofNullable( System.getProperty( "SIZE" ) );
```

```
System.out.println( logProperty.isPresent() );           // true
user nameProperty.ifPresent( System.out::println );     //
Chris
sizeProperty.map( Integer::parseInt ).ifPresent( System.out::println ); // 100
System.out.println( System.getProperty( "DEBUG", "false" ) ); //
false
```

Listing 16.2 `com/tutego/insel/lang/SetProperty.java, main()`

In return, you’ll receive a string indicating the value via `getProperty(String)`. If no property of that name exists at all, you’ll get `null` instead. In this way, we can know if this value was set at all. So, a simple `null` test tells us whether `logProperty` is present or not. Instead of `-DLOG`, `-DLOG=` also returns the same result because the associated value is the empty string. Since all properties are of type `String` to begin with, `user nameProperty` is easy to output, and you’ll get either `null` or the string specified after `=`. If the types aren’t strings, they must be processed further, for example, with `Integer.parseInt()`, `Double.parseDouble()`, and so on. The `System.getProperty(String, String)` method, which is passed two arguments, is pretty useful in this case because the second argument represents a default value. Thus, a default value can always be assumed.

`Boolean.getBoolean(String)`

In the case of properties that are assigned truth values, the following statement can be written:

```
boolean b = Boolean.parseBoolean( System.getProperty( property ) ); // (*)
```

There’s another variant for the truth values. The static method `Boolean.getBoolean(String)` searches for a property with the specified name in the system properties. Thus, the following is analogous to the line (*):

```
boolean b = Boolean.getBoolean( property );
```

You might be surprised to find this static method in the wrapper class `Boolean` because property access has nothing to do with wrapper objects and the class actually goes beyond its area of responsibility in this case.

Compared to a separate, direct `System` query, `getBoolean(String)` also has a disadvantage in that, when it returns `false`, you can’t distinguish whether the property simply doesn’t exist or whether the property is assigned the value `false`. Also, incorrectly set values like `-DP=false` always result in `false`.³

```
final class java.lang.Boolean
implements Serializable, Comparable<Boolean>
```

► `static boolean getBoolean(String name)`

Reads a system property named `name` and returns `true` if the value of the property is equal to the string `"true"`. The return value is `false` if the value of the system property is `false`, if the property doesn’t exist, or if the property is `null`.

16.4.5 Newline Characters and `line.separator`

To move from the end of one line to the beginning of the next, a *newline* is inserted. The character for a new line doesn’t have to be a single character; several characters may also be necessary. Unfortunately for programmers, the number of characters for a new-line sequence depends on the architecture, for instance:

- Unix, macOS: Line feed (LF for short), `\n`
- Windows: Carriage return (CR for short) and line feed

The control code for a carriage return is 13 (0x0D); the control code for a line feed is 10 (0x0A). Java also assigns its own escape sequences for these characters: `\r` for carriage returns and `\n` for line feeds. (The `\f` sequence is for a form feed, also called a “page feed,” which doesn’t play any role in line breaks).

³ This confusion is due to the implementation: `Boolean.valueOf("false")` returns `false` just like `Boolean.valueOf("")` or `Boolean.valueOf(null)`.

In Java, you can obtain a newline character or a newline string from the system in one of the following three ways:

- By calling `System.getProperty("line.separator")`
- By calling `System.lineSeparator()`
- You don't always have to query the character (or, strictly speaking, a possible string of characters) individually. If the character is part of a formatted output at the formatter, `String.format(...)` or `printf(...)`, the format specifier `%n` stands for exactly the newline string stored in the system.

16.4.6 Environment Variables of the Operating System

Almost every operating system uses the concept of *environment variables*; for example, `PATH` is known for the search path for applications on Windows and Unix. Java enables access to these system environment variables. Two static methods are used for this purpose:

```
final class java.lang.System
```

- ▶ `static Map<String, String> getEnv()`
Reads a set of `<string, string>` pairs with all system properties.
- ▶ `static String getEnv(String name)`
Reads a system property named `name`. If the property doesn't exist, the return will be `null`.

Variable Name	Description	Example
COMPUTERNAME	Name of the computer	<i>MOE</i>
HOMEDRIVE	Drive of the user directory	<i>C:</i>
HOMEPATH	Path of the user directory	<i>\Users\Christian</i>
OS	Name of the operating system*	<i>Windows_NT</i>
PATH	Search path	<i>C:\windows\SYSTEM32; C:\windows ...</i>
PATHEXT	File extensions that represent executable programs	<i>.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC</i>
SYSTEMDRIVE	Drive of the operating system	<i>C:</i>
TEMP and also TMP	Temporary directory	<i>C:\Users\CHRIST~1\AppData\Local\Temp</i>
USERDOMAIN	Domain of the user	<i>MOE</i>

Table 16.5 Selection of Some Environment Variables Available in Windows

Variable Name	Description	Example
USERNAME	Name of the user	<i>Christian</i>
USERPROFILE	Profile directory	<i>C:\Users\Christian</i>
WINDIR	Directory of the operating system	<i>C:\windows</i>
* The result differs from <code>System.getProperty("os.name")</code> , which already returns “Windows 10” for Windows 10.		

Table 16.5 Selection of Some Environment Variables Available in Windows (Cont.)

Some variables are also accessible via the system properties, for instance, with `System.getProperties()`, `System.getProperty(...)`, and so on.

Example

The following example outputs the environment variables of the system:


```
Map<String,String> map = System.getenv();  
map.forEach( (k, v) -> System.out.printf( "%s=%s\n", k, v ) );
```

Ex

16.5 The Languages of Different Countries

When developers start with console or GUI output, they often hardwire the output to a local language. If the language changes, the software can't handle other country-specific rules, for example, when formatting floats. Developing “multilingual” programs that provide localized outputs in different languages is not too difficult. Basically, you'll replace all language-dependent strings and formatting of data with code that takes into account country-specific output formats and rules. Java offers a solution for these cases: on one hand, you can define a language that then specifies rules according to which the Java API can automatically format data, and on the other hand, you can allow language-dependent parts to be swapped out to resource files.

16.5.1 Regional Languages via Locale Objects

In Java, `Locale` objects represent languages in geographic, political, or cultural regions. The language and the region must be separated because a region or a country doesn't always clearly specify the language. For Canada, in the province of Quebec, the French edition is relevant, which, of course, differs from the English edition. Each of these language-specific properties can be encapsulated in a special object. These `Locale` objects are then passed to a `Formatter` that's located behind `String.format(...)` and `printf(...)` or passed to a `Scanner`. These outputs are referred to as *locale sensitive*.

Building Locale Objects

Locale objects are always created with the name of the language and optionally with the name of the country or a region and variant. The `Locale` class provides three ways to build the objects:

- Using the `Locale` constructor (deprecated in Java 19)
- Using the nested `Builder` class of `Locale` uses the builder pattern to build new `Locale` objects
- Using the `Locale` method `forLanguageTag(...)` and a string identifier



Example

Country abbreviations are specified in the constructor of the `Locale` class, for example, for a language object for Great Britain or France. Consider the following examples:

```
Locale greatBritain = new Locale( "en", "GB" );
Locale french      = new Locale( "fr" );
```

In the second example, we don't care about the country. We're simply choosing French as the language, no matter what part of the world.

Languages are identified by 2-letter abbreviations from the ISO 639 code⁴ (*ISO Language Code*), and country names are 2-letter abbreviations described in ISO 3166⁵ (*ISO Country Code*).



Example

Three variants for building `Locale.JAPANESE` are found in the following example:

```
Locale loc1 = new Locale( "ja" );
Locale loc2 = new Locale.Builder().setLanguage( "ja" ).build();
Locale loc3 = Locale.forLanguageTag( "ja" );
final class java.util.Locale
implements Cloneable, Serializable
```

- `Locale(String language)`
Creates a new `Locale` object for the language given by the ISO-693 standard. Invalid identifiers aren't recognized.
- `Locale(String language, String country)`
Creates a `Locale` object for a language according to ISO 693 and a country according to the ISO 3166 standard.

⁴ https://en.wikipedia.org/wiki/List_of_ISO_639-1_codes
⁵ https://en.wikipedia.org/wiki/ISO_3166-1

- `Locale(String language, String country, String variant)`
Creates a `Locale` object for a language, a country, and a variant. `variant` is a vendor-dependent specification like "WIN" or "MAC."

The static `Locale.getDefault()` method returns the currently set language. For the running JVM, `Locale.setDefault(Locale)` can change the language.

The `Locale` class has more methods; developers should study the Javadoc for the `Builder`, for `forLanguageTag(...)` and the new extensions and filter methods.⁶

Constants for Some Languages

The `Locale` class has constants for commonly occurring languages, with an option for specifying countries. The constants for countries and languages include, for example, `CANADA`, `CHINA`, `FRENCH`, `GERMAN`, `ITALIAN`, `KOREAN`, `TAIWAN`, `UK` and `US`. Behind an abbreviation like `Locale.UK`, nothing else exists except for the initialization with `new Locale("en", "GB")`.

Methods That Accept Instances of Locale

`Locale` objects are actually not interesting as objects—they do have methods, but more exciting is its use as a type for the identification of a language. Dozens of methods in the Java library accept `Locale` objects and adjust their behaviors based on them. Examples include `printf(Locale, ...)`, `format(Locale, ...)`, and `toLowerCase(Locale)`.

Tip

If no variant of a format or parse method exists with a `Locale` object; the method usually doesn't support language-dependent behavior. The same limitation applies to objects that don't accept a `Locale` via a constructor or setter. `Double.toString(...)` is one such example, as is `Double.parseDouble(...)`. In internationalized applications, these methods will rarely be found. Also, string concatenation with, for example, a float isn't permitted (because a `Double` method is called internally), and using `String.format(...)` is definitely a better option.

Methods of Locale*

`Locale` objects provide a number of methods that reveal the ISO-639 code of the country, for example.

Example

The following example outputs the `Locale` information accessible for languages in selected countries. The objects `System.out` and `Locale.*` are imported statically:

⁶ Oracle's Java tutorial describes these extensions at <http://docs.oracle.com/javase/tutorial/i18n/locale/index.html>.

```
out.println(GERMANY.getCountry());           // DE
out.println(GERMANY.getLanguage());          // de
out.println(GERMANY.getVariant());           //
out.println(GERMANY.getISO3Country());       // DEU
out.println(GERMANY.getISO3Language());      // deu
out.println(CANADA.getDisplayCountry());     // Canada
out.println(GERMANY.getDisplayLanguage());   // German
out.println(GERMANY.getDisplayName());       // German (Germany)
out.println(CANADA.getDisplayName());        // English (Canada)
out.println(GERMANY.getDisplayName(FRENCH)); // allemand (Allemagne)
out.println(CANADA.getDisplayName(FRENCH));  // anglais (Canada)
```

Listing 16.3 src/main/java/com/tutego/insel/locale/GermanyLocal.java, main()

Static methods also exist for querying `Locale` objects:

```
final class java.util.Locale
implements Cloneable, Serializable
```

- ▶ `static Locale getDefault()`
Returns the language preset by the JVM, which defaults to the operating system.
- ▶ `static Locale[] getAvailableLocales()`
Returns a list of all installed `Locale` objects. The field contains at least `Locale.US` and about 160 entries.
- ▶ `static String[] getISOCountries()`
Returns an array of all 2-letter ISO-3166 country codes.
- ▶ `static Set<String> getISOCountries(Locale.IsoCountryCode type)`
Returns a set with all ISO-3166 country codes, where the `IsoCountryCode` list determines the following: `PART1_ALPHA2` returns the code of 2 letters, `PART1_ALPHA3` of 3 letters, `PART3` of 4 letters.

On the other hand, other methods provide abbreviations according to ISO standards:

```
final class java.util.Locale
implements Cloneable, Serializable
```

- ▶ `String getCountry()`
Returns the country abbreviation according to the ISO-3166 2-letter code.
- ▶ `String getLanguage()`
Returns the abbreviation of the language in ISO-639 code.
- ▶ `String getISO3Country()`
Returns the ISO abbreviation of the country of these settings and throws a `MissingResourceException` if the ISO abbreviation isn't available.

- ▶ `String getISO3Language()`
Returns the ISO abbreviation of the language of these settings and throws a `MissingResourceException` if the ISO abbreviation isn't available.
- ▶ `String getVariant()`
Returns the abbreviation of the variant or an empty string.

These methods provide abbreviations, but they aren't intended for human-readable output. For various `get*()` methods, therefore, corresponding `getDisplay*()` methods exist:

```
final class java.util.Locale
implements Cloneable, Serializable
```

- ▶ `String getDisplayCountry(Locale inLocale)`
`final String getDisplayCountry()`
Returns the name of the country for screen outputs for a language or `Locale.getDefault()`.
- ▶ `String getDisplayLanguage(Locale inLocale)`
`String getDisplayLanguage()`
Returns the name of the screen output language for a given `Locale` or `Locale.getDefault()`.
- ▶ `String getDisplayName(Locale inLocale)`
`final String getDisplayName()`
Returns the name of the settings for a language or `Locale.getDefault()`.
- ▶ `String getDisplayVariant(Locale inLocale)`
`final String getDisplayVariant()`
Returns the name of the variant for a language or `Locale.getDefault()`.

16.6 Overview of Important Date Classes

Because date calculations are convoluted entities, we can be grateful to the developers of Java for providing many classes for date calculation and formatting. The developers have kept the classes abstract enough to allow for local specifics like output formatting, parsing, time zones, or daylight saving time/winter time in different calendars.

Prior to Java 1.1, only the `java.util.Date` class was available for displaying and manipulating date values, and this class had to carry out several tasks:

- Creation of a date/time object from year, month, day, minute, and second
- Querying day, month, year, and so on with an accuracy of milliseconds
- Processing and output of date strings

Since the `Date` class wasn't quite bug-free and internationalized, new classes were introduced in JDK 1.1, namely, the following:

- `Calendar` takes on `Date`'s task of converting between different date representations and time scales. The `GregorianCalendar` subclass is created directly.
- `DateFormat` breaks up date strings and formats the output. Date formats also depend on the country, which Java represents through `Locale` objects, and on a time zone, which is represented by the instances of the `TimeZone` class.

In Java 8, another date library was added with entirely new types. Finally, date and time can be represented separately:

- `LocalDate`, `LocalTime`, and `LocalDateTime` are the temporal classes for a date, for a time, and for a combination of date and time, respectively.
- `Period` and `Duration` represent intervals.

16.6.1 Unix Time: January 1, 1970

January 1, 1970, was a Thursday with groundbreaking changes: The British rejoiced that the age of majority dropped from 24 to 18, and as in every year, and people everywhere woke up to massive hangovers from the night before. For us, however, a technical innovation is of concern: The date of 1/1/1970, 0:00:00 UTC is also referred to as the *Unix epoch*, and a *Unix time* is described in relation to this time in terms of seconds. For example, 100,000,000 seconds after 1/1/1970 is March 3, 1973, at 09:46:40. The *Unix Billionnium* was celebrated 1,000,000,000 seconds after Jan. 1, 1970, namely, on Sept. 9, 2001, at 01:46:40.

16.6.2 System.currentTimeMillis()

Unix time is also an important concept for us Java developers because many times in Java are relative to this date. The timestamp 0 refers to 1/1/1970 0:00:00 Greenwich Mean Time. The `System.currentTimeMillis()` method returns the past milliseconds—not seconds!—relative to 1/1/1970, 00:00 UTC, although your operating system's clock may not be that accurate. The number of milliseconds is represented in a `long` (i.e., in 64 bits), which will suffice for about 300 million years.



Warning

The values of `currentTimeMillis()` don't necessarily ascend because Java gets the time from the operating system, where the system time can change. A user can adjust the time, or a service such as the *Network Time Protocol (NTP)* takes over this task. Differences of `currentTimeMillis()` timestamps are then completely wrong and could even be negative. An alternative is `nanoTime()`, which has no reference point, is more precise, and is always ascending.⁷

⁷ <http://stackoverflow.com/questions/351565/system-currenttimemillis-vs-system-nanotime> goes into more details and provides links to internal implementations.

16.6.3 Simple Time Conversions via TimeUnit

A time duration in Java is often expressed in terms of milliseconds. 1,000 milliseconds correspond to 1 second, 1,000 × 60 milliseconds to 1 minute, and so on. However, all those large numbers aren't easy to read, which is why `TimeUnit` objects are used with their `to*(...)` methods for the purpose of conversion. Java declares the following constants in `TimeUnit`: `NANOSECONDS`, `MICROSECONDS`, `MILLISECONDS`, `DAYS`, `HOURS`, `SECONDS`, and `MINUTES`.

Each of the enumeration elements defines the conversion methods, for instance, `toDays(...)`, `toHours(...)`, `toMicros(...)`, `toMillis(...)`, `toMinutes(...)`, `toNanos(...)`, and `toSeconds(...)`. These methods receive a `long` and return a `long` in the corresponding unit. In addition, two `convert(...)` methods convert from one unit to another.

Example

The following example converts 23,746,387 milliseconds to hours:

```
int v = 23_746_387;
System.out.println( TimeUnit.MILLISECONDS.toHours( v ) ); // 6
System.out.println( TimeUnit.HOURS.convert( v,
    TimeUnit.MILLISECONDS ) ); // 6
enum java.util.concurrent.TimeUnit
extends Enum<TimeUnit>
implements Serializable, Comparable<TimeUnit>
```

- ▶ `NANOSECONDS`, `MICROSECONDS`, `MILLISECONDS`, `SECONDS`, `MINUTES`, `HOURS`, `DAYS` `TimeUnit` enumeration elements.
- ▶ `long toDays(long duration)`
- ▶ `long toHours(long duration)`
- ▶ `long toMicros(long duration)`
- ▶ `long toMillis(long duration)`
- ▶ `long toMinutes(long duration)`
- ▶ `long toNanos(long duration)`
- ▶ `long toSeconds(long duration)`
- ▶ `long convert(long sourceDuration, TimeUnit sourceUnit)`
Returns `sourceUnit.to*(sourceDuration)`, where `*` represents the respective unit. For example, this method returns `HOURS.convert(sourceDuration, sourceUnit)`, then `sourceUnit.toHours(1)`. The readability of this method isn't ideal, so the other methods should be preferred. Results may be truncated, not rounded. If an overflow occurs, no `ArithmeticException` will follow.



► `long convert(Duration duration)`
Converts the passed duration into the time unit that represents the current `TimeUnit`. For example, `TimeUnit.MINUTES.convert(Duration.ofHours(12))` returns 720. Thus, for example, `aunit.convert(Duration.ofNanos(n))` and `aunit.convert(n, NANOSECONDS)` are the same.

16.7 Date-Time API

The `java.time` package is based on the standardized calendar system of ISO-8601, and this covers how a date is represented, including time, date and time, UTC, time intervals (duration/time span), and time zones. The implementation is based on the Gregorian calendar, although other calendar types are also conceivable. Java’s calendar system can use other standards or implementations as well, including the *Unicode Common Locale Data Repository (CLDR)* for localizing days of the week or the *Time-Zone Database (TZDB)*, which documents all time zone changes since 1970.

16.7.1 Initial Overview

The central temporal types from the Date-Time API can be quickly documented:

Type	Description	Field(s)
<code>LocalDate</code>	Represents a common date	Years, months, and days
<code>LocalTime</code>	Represents a common time	Hours, minutes, seconds, and nanoseconds
<code>LocalDateTime</code>	Combination of date and time	Years, months, days, hours, minutes, seconds, and nanoseconds
<code>Period</code>	Duration between two Local-Dates	Years, months, and days
<code>Year</code>	Year only	Year
<code>Month</code>	Month only	Month
<code>MonthDay</code>	Month and day only	Month, day
<code>OffsetTime</code>	Time with time zone	Hours, minutes, seconds, nanoseconds, and zone offset
<code>OffsetDateTime</code>	Date and time with time zone as UTC offset	Year, month, day, hours, minutes, seconds, nanoseconds, and zone offsets

Table 16.6 All Temporal Classes from `java.time`

Type	Description	Field(s)
<code>ZonedDateTime</code>	Date and time with time zone as ID and offset	Year, month, day, hours, minutes, seconds, nanoseconds, and zone info
<code>Instant</code>	Time (continuous machine time)	Nanoseconds
<code>Duration</code>	Time interval between two instants	Seconds/nanoseconds

Table 16.6 All Temporal Classes from `java.time` (Cont.)

16.7.2 Human Time and Machine Time

Date and time, which we as humans understand in units such as days and minutes, is referred to as *human time*, while the continuous time of the computer, which has a resolution in the nanosecond range, is called *machine time*. The machine time begins at a time we call an *epoch*, namely, the Unix epoch.

From Chapter 7, Section 7.2.5, you learned how most classes are made for humans and that only `Instant/Duration` refers to machine time. `LocalDate`, `LocalTime`, and `LocalDateTime` represent human time without reference to a time zone, whereas `ZonedDateTime` does reference a time zone. When choosing the right time classes for a task, the first consideration is, of course, whether to represent human time or machine time. This choice is followed by questions about exactly which fields are needed and whether a time zone is relevant or not. For example, if the execution time is to be measured, you don’t need to know on which date the measurement started and ended; in this case, `Duration` would be correct, unlike `Period`.

Example

Examples for explicit formatting and default formatting for the US locale:

```
LocalDate now = LocalDate.now();
System.out.println( now ); // e.g., 2023-01-31
System.out.printf( "%d. %s %d%n",
                    now.getDayOfMonth(), now.getMonth(), now.getYear() );
// e.g., 31. JANUARY 2023
LocalDate bdayMLKing = LocalDate.of( 1929, Month.JANUARY, 15 );
DateTimeFormatter formatter =
    DateTimeFormatter.ofLocalizedDate( FormatStyle.MEDIUM );
System.out.println( bdayMLKing.format( formatter ) ); // Jan 15, 1929
```

The `getMonth()` method on a `LocalDate` returns a `java.time.Month` object as the result, and these are enumerations. The `toString()` representation returns the constant in uppercase letters.

All classes are based on the ISO system by default. Other calendar systems, such as the Japanese calendar, are created using types from `java.time.chrono`, and of course, entirely new systems are also possible.



Example

Output for the Japanese calendar:

```
ChronoLocalDate now = JapaneseChronology.INSTANCE.dateNow();
System.out.println( now );           // Japanese Reiwa 4-01-31
```

Package Overview

The types of the Date-Time API are distributed among different packages:

- `java.time`
Contains the standard classes like `LocalTime` and `Instant`. All types are based on the ISO-8601 calendar system, commonly known as the “Gregorian calendar.” This calendar is extended by the *Proleptic Gregorian Calendar*. This calendar is also valid for the time before 1582 (the introduction of this calendar), so that a consistent timeline can be used.
- `java.time.chrono`
In this package, you’ll find predefined alternative (i.e., non-ISO) calendar systems, such as the Japanese calendar, the Thai-Buddhist calendar, the Islamic calendar, and a few others.
- `java.time.format`
Classes for formatting and parsing date and time, such as the `DateTimeFormatter`.
- `java.time.zone`
Supporting classes for time zones, such as `ZonedDateTime`.
- `java.time.temporal`
Deeper API that allows access and modification of individual fields of a date/time value.

Design Principles

Before we get into the individual classes, let’s look at some design principles because all types of the Date-Time API follow recurring patterns. The first and most important property is that all objects are *immutable*; that is, they can’t be changed. In contrast, with the “old” API, `Date` and the `Calendar` classes were mutable, with sometimes devastating consequences. If these objects are passed around and changed, incalculable side effects can occur. The classes of the new Date-Time API are immutable, and so the date/time classes like `LocalTime` or `Instant` are opposed to mutable types like `Date` or `Calendar`. All methods that look as if they permitted changes now instead create new objects with the desired changes. Side effects are therefore absent, and all types are thread safe.

Immutability is a design property as is the fact that `null` isn’t permitted as an argument. In the Java API, `null` is often accepted because it expresses something optional, but the Date-Time API usually penalizes `null` with a `NullPointerException`. The fact that `null` isn’t in use as an argument and not as a return benefits another property: The code can be mostly written with a fluent API (i.e., cascaded calls) since many methods return the `this` reference, as is known from `StringBuilder`.

Added to these more technical features is a consistent naming that is different from the naming of the well-known `JavaBeans`. So, no constructors and no setters exist (immutable classes don’t need them), but instead, patterns adhere to many types from the Date-Time API:

Method	Class/Instance Method	Basic Meaning
<code>now()</code>	Static	Returns an object with current time/current date
<code>of*()</code>	Static	Creates new objects
<code>from</code>	Static	Creates new objects from other representations
<code>parse*()</code>	Static	Creates a new object from a string representation
<code>format()</code>	Instance	Formats and returns a string
<code>get*()</code>	Instance	Returns fields of an object
<code>is*()</code>	Instance	Queries the status of an object
<code>with*()</code>	Instance	Returns an instance of the object with a changed state
<code>plus*()</code>	Instance	Returns an instance of the object with a totaled state
<code>minus*()</code>	Instance	Returns an instance of the object with a reduced state
<code>to*()</code>	Instance	Converts an object to a new type
<code>at*()</code>	Instance	Combines this object with another object
<code>*Into()</code>	Instance	Combines an own object with another target object

Table 16.7 Name Patterns in the Date-Time API

You’ve already used the `now()` method in one of the first examples in this section, and this method returns the current date, for example. Other creator methods are prefixed with `of`, `from`, or `with`; no constructors exist. The methods of the `with*()` type assume the role of setters.

16.7.3 The `LocalDate` Date Class

A date (without a time zone) is represented by the `LocalDate` class. This class can be used to represent a birth date, for example.

A temporal object can be created using the static of (...) factory methods and derived via `ofInstant(Instant instant, ZoneId zone)` or from another temporal object. Interesting are the methods that work with a `TemporalAdjuster`.

Equipped with these objects, you can use various getters and query individual fields, such as `getDayOfMonth()`; `getDayOfYear()` (return `int`); `getDayOfWeek()`, which returns an enumeration of type `DayOfWeek`; and `getMonth()`, which returns an enumeration of type `Month`. Furthermore, other methods include `long toEpochDay()` and `long toEpochSecond(LocalTime time, ZoneOffset offset)`.



Example

Find the next Saturday from now:

```
LocalDate today = LocalDate.now();
LocalDate nextSaturday =
    today.with( TemporalAdjusters.next(DayOfWeek.SATURDAY) );
System.out.printf( "Today is %s, and next Saturday is %s",
                    today, nextSaturday );
```

In addition, some methods return new `LocalDate` objects with `minus*()` or `plus*()` if, for example, a number of years should be returned with `minusYear(long yearsToSubtract)`. By negating the sign, the opposite method can also be used. In other words, `LocalDate.now().minusMonths(1)` provides the same result as `LocalDate.now().plusMonths(-1)`. The `with*()` methods reassign a field and return a modified new `LocalDate` object.

From a `LocalDate` object, you can create other temporal objects: `atTime(...)`, for example, returns `LocalDateTime` objects in which certain time fields are assigned. `atTime(int hour, int minute)` is such an example. With `until(...)`, a time duration of the `Period` type can be returned. Two methods that provide a stream of `LocalDate` objects up to an endpoint are also interesting:

- `Stream<LocalDate> datesUntil(LocalDate endExclusive)`
- `Stream<LocalDate> datesUntil(LocalDate endExclusive, Period step)`

16.8 Logging with Java

Logging information about program states is important for reconstructing and understanding the flow and states of a program at a later time. A logging API can write messages to the console or to external storage, such as text files, XML files, and databases, or to distribute these messages via chat.

16.8.1 Logging Application Programming Interfaces

Regarding logging libraries and APIs, the Java world is unfortunately divided. Since the Java standard library didn't provide a logging API in its first versions, the open-source library *log4j* quickly filled this gap. This library is used in almost every major Java project today. When the Logging API moved into Java 1.4 with Java Specification Request (JSR) 47, the Java community was surprised to find that `java.util.logging (JUL)` was neither API-compatible with the popular *Log4j* nor as powerful as *Log4j*.⁸

Over the years, the picture has changed. While in the early day's developers relied exclusively on *log4j*, more and more projects are now using *JUL*. One of the reasons is that some developers want to avoid external dependencies (although this doesn't really work since almost every included Java library is based on *log4j*). Another reason is that for many projects *JUL* is simply sufficient. In practice, for larger projects, as a result, multiple logging configurations overcrowd their own programs, as each logging implementation is configured differently.

16.8.2 Logging with java.util.logging

The Java logging API can write a message that you can then use for maintenance or security checks. The API is simple:

```
package com.tutego.insel.logging;
```

```
import static java.time.temporal.ChronoUnit.MILLIS;
import static java.time.Instant.now;
import java.time.Instant;
import java.util.logging.Level;
import java.util.logging.Logger;
```

```
public class JULDemo {
```

```
    private static final Logger log = Logger.getLogger( JULDemo.class.getName() );
```

```
    public static void main( String[] args ) {
        Instant start = now();
        log.info( "About to start" );
```

```
        try {
            log.log( Level.INFO, "Lets try to throw {0}", "null" );
            throw null;
        }
```

⁸ The standard logging API, on the other hand, provides only basics like hierarchical loggers. Standard logging doesn't come close to the power of *log4j* with its large number of writers in files, syslog/NT loggers, databases, and dispatch over the network.

```
        catch ( Exception e ) {
            log.log( Level.SEVERE, "Oh Oh", e );
        }
        log.info( () -
> String.format( "Runtime: %s ms", start.until(now(), MILLIS )) );
    }
}
```

Listing 16.4 src/main/java/com/tutego/insel/logging/CULDemo.java, JULDemo

When you run the example, the following warning appears on the console:

```
Jan. 24, 2022 7:47:46 PM com.tutego.insel.logging.JULDemo main
INFO: About to start
Jan. 24, 2022 7:47:46 PM com.tutego.insel.logging.JULDemo main
INFO: Lets try to throw null
Jan. 24, 2022 7:47:46 PM com.tutego.insel.logging.JULDemo main
SEVERE: Oh Oh
java.lang.NullPointerException: Cannot throw exception because "null" is null
at com.tutego.insel.logging.JULDemo.main(JULDemo.java:20)

Jan. 24, 2022 7:47:46 PM com.tutego.insel.logging.JULDemo main
INFO: Runtime: 35 ms
```

The Logger Object

The `Logger` object is a central element that can be retrieved via `Logger.getAnonymousLogger()` or via `Logger.getLogger(String name)`, where `name` is usually assigned the fully qualified class name. Often, the `Logger` object is declared as a private static final variable in the class.

Logging with Log Level

Not every message is equally important. Some messages are useful for debugging or because of timing measurements, but exceptions in the `catch` branches are hugely important. To support different levels of detail, you can specify a *log level*. This level determines how “serious” the error or a message is, which is important later when errors are sorted according to their urgency. Log levels are declared as constants in the `Level` class⁹ in the following ways:

- `FINEST` (smallest level)
- `FINER`
- `FINE`

⁹ Since the logging framework joined Java in version 1.4, it doesn’t yet use typed enumerations, which have only been available since Java 5.

- `CONFIG`
- `INFO`
- `WARNING`
- `SEVERE` (highest level)

For the logging process itself, the `Logger` class provides the general method `log(Level level, String msg)` or a separate method for each level.

Level	Call via log(...)	Special Log Method
SEVERE	<code>log(Level.SEVERE, msg)</code>	<code>severe(String msg)</code>
WARNING	<code>log(Level.WARNING, msg)</code>	<code>warning(String msg)</code>
INFO	<code>log(Level.INFO, msg)</code>	<code>info(String msg)</code>
CONFIG	<code>log(Level.CONFIG, msg)</code>	<code>config(String msg)</code>
FINE	<code>log(Level.FINE, msg)</code>	<code>fine(String msg)</code>
FINER	<code>log(Level.FINER, msg)</code>	<code>finer(String msg)</code>
FINEST	<code>log(Level.FINEST, msg)</code>	<code>finest(String msg)</code>

Table 16.8 Log Levels and Methods

All these methods send a message of type `String`. If an exception and the associated stack trace must be logged, developers must use the following logger method, which is also used in the example:

► `void log(Level level, String msg, Throwable thrown)`

The variants of `severe(...)`, `warning(...)`, and so on are not overloaded with a `Throwable` parameter type.

16.9 Maven: Resolving Build Management and Dependencies

In Chapter 1, Section 1.9.1, we created a Maven project, but never really benefited from using Maven. Two things stand out:

1. Dependencies can be easily declared, and they are automatically downloaded by Maven, including all sub-dependencies. Maven’s particular strength lies in resolving transitive dependencies.
2. During the build, Java source code alone doesn’t make a project; the sources must be compiled, test cases must be run, and Javadoc should be generated. At the end, the final result is usually a compressed JAR file.

16.9.1 Dependency to Be Accepted

As an example, let's create a dependency on the small web framework *Spark* (<https://sparkjava.com>). Let's open the Project Object Model (POM) file *pom.xml* and add the code lines in bold for the dependency:

```
<project ...>
...
<properties>
  <maven.compiler.target>17</maven.compiler.target>
  <maven.compiler.source>17</maven.compiler.source>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>
<dependencies>
  <dependency>
    <groupId>com.sparkjava</groupId>
    <artifactId>spark-core</artifactId>
    <version>2.9.3</version>
  </dependency>
</dependencies>
</project>
```

Listing 16.5: pom.xml

All dependencies are located in a special XML element named `<dependencies>`. Below that element, you can then have any number of `<dependency>` blocks.

Now that everything is prepared, let's write the main program:

```
public class SparkServer {
  public static void main( String[] args ) {
    spark.Spark.get( "/hello", ( req, res ) -
> "Hello Browser " + req.userAgent() );
  }
}
```

Listing 16.5 src/main/java/SparkServer.java

When you start the program as usual, a web server starts as well, and you can read the output via the URL <http://localhost:4567/hello>. (You can ignore the logger outputs.)

16.9.2 Local and the Remote Repository

Resolving dependent JAR files takes longer the first time because Maven contacts a remote repository and always pulls the latest JAR files from that location and then stores these files locally. The extensive remote repository stores almost all versions of JAR files for many well-known open-source projects. The *Central Repository* can be found <https://repo.maven.apache.org/maven2/>.

The downloaded resources themselves aren't stored in the project but in a local repository located in the user's home directory and named *.m2*. In this way, all Maven projects share the same JAR files, and they don't have to be reobtained and updated on a project-by-project basis.

16.9.3 Lifecycles, Stages, and Maven Plugins

A Maven build consists of a three-stage lifecycle: *clean*, *default*, and *site*. Within this lifecycle are *stages*. For example, *default* contains the *compile* stage for translating the sources. Everything Maven runs are *plugins*, such as compilers, and many others that are listed at <https://maven.apache.org/plugins/>. A plugin can execute different *goals*. For example, the Javadoc plugin (described at <https://maven.apache.org/components/plugins/maven-javadoc-plugin/>) currently knows 16 goals. A goal can be accessed subsequently via the command line or via the integrated development environment (IDE).

For example, a JAR file is created via the *package* stage:

```
$ mvn package
```

The command-line tool must be called in the directory where the POM file is located.

16.10 Further Reading

The Java library provides a large number of classes and methods, but not always exactly what's required by the current project. Some problems, such as the structure and configuration of Java projects, object-relational mappers (www.hibernate.org), or command-line parsers, may require various commercial or open-source libraries and frameworks. With purchased products, licensing issues are obvious, but with open-source products, integration into one's own closed source project isn't always a given. Various types of licenses (<https://opensource.org/licenses>) for open-source software with always different specifications—whether the source code is changeable, whether derivatives must also be free, whether mixing with proprietary software possible—complicate the choice, and violations (<https://gpl-violations.org/>) are publicly denounced and unpleasant. Java developers should increasingly focus their attention on software under the Berkeley Source Distribution (BSD) license (the Apache license belongs in this group) and under the LGPL license for commercial distribution. The Apache group has assembled a nice collection of classes and methods named *Apache Commons* (<http://commons.apache.org>), and studying these sources are recommended for software developers. The website <https://www.openhub.net> is exceptionally well suited for this purpose and enables searching via specific keywords through more than 1 billion source code lines of various programming languages—amazing how many developers use profanities!

Chapter 16

The Class Library

*“What we need is some crazy people;
look where the normal ones have taken us.”
—George Bernard Shaw (1856–1950)*

In this chapter you’ll learn about the Java class library.

16.1 The Java Class Philosophy

A programming language consists not only of a grammar, but also, as in the case of Java, of a programming library. A platform-independent language—as many imagine C or C++ to be—isn’t really platform independent if different functions and programming models are used on each computer, which is exactly the weak point of C(++). These algorithms, which aren’t dependent on the operating system, can be applied everywhere in the same way, but the result is realized in the end with inputs/outputs or graphical user interfaces (GUIs). The Java library, on the other hand, tries to abstract away from platform-specific features, and the developers have gone to great lengths to put all the important methods into well-formed object-oriented (OO) classes and packages. These elements cover in particular the central areas of data structures, input and output, graphics, and network programming.



16.1.1 Modules, Packages, and Types

At the top of the Java library are modules, which in turn consist of packages, which in turn contain types.

Modules of the Java SE

The *Java Platform, Standard Edition (Java SE)* application programming interface (API) consists of the following modules, all of which begin with the prefix `java`:

Module	Description
<code>java.base</code>	Fundamental types of Java SE
<code>java.compiler</code>	Java language model, annotation processing, and Java compiler API
<code>java.datatransfer</code>	The API for data transfer between applications, usually the clipboard
<code>java.desktop</code>	GUIs with Abstract Windowing Toolkit (AWT) and Swing, the <i>Accessibility</i> API, audio, printing, and JavaBeans
<code>java.instrument</code>	Instrumentalization is the modification of Java programs at run-time
<code>java.logging</code>	The <i>Logging</i> API
<code>java.management</code>	<i>Java Management Extensions (JMX)</i>
<code>java.management.rmi</code>	<i>Remote Method Invocation (RMI)</i> connector for remote access to the JMX beans
<code>java.naming</code>	The <i>Java Naming and Directory Interface (JNDI)</i> API
<code>java.prefs</code>	The <i>Preferences</i> API is used to store user preferences
<code>java.rmi</code>	Remote method calls with the RMI API
<code>java.scripting</code>	The <i>Scripting</i> API
<code>java.security.jgss</code>	Java binding of the <i>IETF Generic Security Services</i> API (GSS API)
<code>java.security.sasl</code>	Java support for <i>IETF Simple Authentication and Security Layer (SASL)</i>
<code>java.sql</code>	The JDBC API for accessing relational databases
<code>java.sql.rowset</code>	The <i>JDBC RowSet</i> API
<code>java.xml</code>	XML classes with the <i>Java API for XML Processing (JAXP)</i> , <i>Streaming API for XML (StAX)</i> , <i>Simple API for XML (SAX)</i> , and <i>W3C Document Object Model (DOM)</i> API
<code>java.xml.crypto</code>	The API for XML cryptography

Table 16.1 Modules of the Java SE

The `java.base` module—the most important module—contains core classes such as `Object` and `String`, among others. This module is the only module that doesn’t itself contain any dependency on other modules. Every other module, however, references at least `java.base`. The Javadoc contains a nice graphical representation, shown in Figure 16.1.

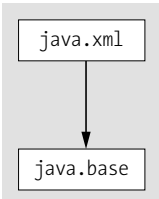


Figure 16.1 The `java.xml` Module Has a Dependency on the `java.base` Module

In some cases, more dependencies exist, such as with the `java.desktop` module, shown in Figure 16.2.

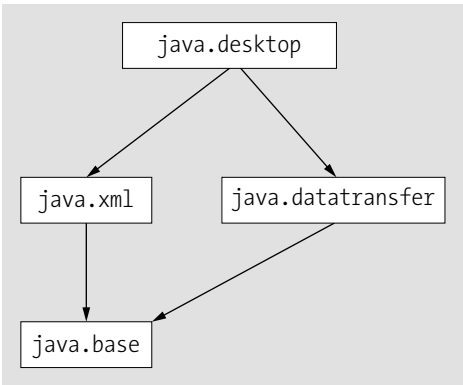


Figure 16.2 Dependencies of the `java.desktop` Module

The `java.se` Module

One special module is `java.se`, which doesn’t declare its own packages or types but merely groups other modules together. The name for such a construction is *aggregator module*. The `java.se` module defines the API for the Java SE platform in this way, as shown in Figure 16.3.

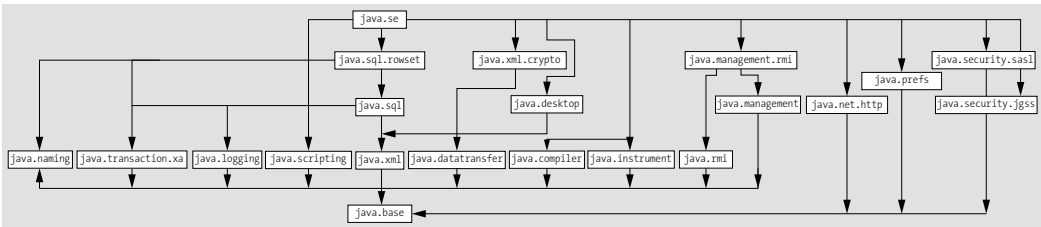


Figure 16.3 Dependencies of the `java.se` Module



Note

In the following sections, we won't discuss the Java SE types in terms of the module from which they originate. You only need to know in which module a type is located when building smaller subsets of Java SE.

Other Modules

Two other modules that also start with java but aren't part of the Java SE standard are java.jnlp for the Java Network Launch Protocol (JNLP) and java.smartcardio, which is the Java API for communication with smart cards according to the international standard ISO/IEC 7816-4.¹

The Java Development Kit (JDK) is the standard implementation of Java SE. This implementation provides developers with more packages and classes, such as with an HTTP server or with Java tools like the Java compiler and the Javadoc tool. In this implementation, several modules will start with the prefix jdk.

16.1.2 Overview of the Packages of the Standard Library

The *Java 11 Core Java SE API* consists of the following modules and packages:

Modules	Packages Included
java.base	java.io, java.lang, java.lang.annotation, java.lang.invoke, java.lang.module, java.lang.ref, java.lang.reflect, java.math, java.net, java.net.spi, java.nio, java.nio.channels, java.nio.channels.spi, java.nio.charset, java.nio.charset.spi, java.nio.file, java.nio.file.attribute, java.nio.file.spi, java.security, java.security.cert, java.security.interfaces, java.security.spec, java.text, java.text.spi, java.time, java.time.chrono, java.time.format, java.time.temporal, java.time.zone, java.util, java.util.concurrent,
java.base	java.util.concurrent.atomic, java.util.concurrent.locks, java.util.function, java.util.jar, java.util.regex, java.util.spi, java.util.stream, java.util.zip, javax.crypto, javax.crypto.interfaces, javax.crypto.spec, javax.net, javax.net.ssl, javax.security.auth, javax.security.auth.callback, javax.security.auth.login, javax.security.auth.spi, javax.security.auth.x500, javax.security.cert

Table 16.2 Packages in the Modules of the Java 17 Core Java SE API

¹ https://en.wikipedia.org/wiki/ISO/IEC_7816

Modules	Packages Included
java.compiler	javax.annotation.processing, javax.lang.model, javax.lang.model.element, javax.lang.model.type, javax.lang.model.util, javax.tools
java.datatransfer	java.awt.datatransfer
java.desktop	java.applet, java.awt, java.awt.color, java.awt.desktop, java.awt.dnd, java.awt.event, java.awt.font, java.awt.geom, java.awt.im, java.awt.im.spi, java.awt.image, java.awt.image.renderable, java.awt.print, java.beans, java.beans.beancontext, javax.accessibility, javax.imageio, javax.imageio.event, javax.imageio.metadata, javax.imageio.plugins.bmp, javax.imageio.plugins.jpeg, javax.imageio.plugins.tiff, javax.imageio.spi, javax.imageio.stream, javax.print, javax.print.attribute, javax.print.attribute.standard, javax.print.event, javax.sound.midi, javax.sound.midi.spi, javax.sound.sampled, javax.sound.sampled.spi, javax.swing, javax.swing.border, javax.swing.colorchooser, javax.swing.event, javax.swing.filechooser, javax.swing.plaf, javax.swing.plaf.basic, javax.swing.plaf.metal, javax.swing.plaf.multi, javax.swing.plaf.nimbus, javax.swing.plaf.synth, javax.swing.table, javax.swing.text, javax.swing.text.html, javax.swing.text.html.parser, javax.swing.text.rtf, javax.swing.tree, javax.swing.undo
java.instrument	java.lang.instrument
java.logging	java.util.logging
java.management	java.lang.management, javax.management, javax.management.loading, javax.management.modelmbean, javax.management.monitor, javax.management.openmbean, javax.management.relation, javax.management.remote, javax.management.timer
java.management.rmi	javax.management.remote.rmi
java.naming	javax.naming, javax.naming.directory, javax.naming.event, javax.naming.ldapjavax.naming.spi
java.prefs	java.util.prefs
java.rmi	java.rmi, java.rmi.activation, java.rmi.dgc, java.rmi.registry, java.rmi.server, javax.rmi.ssl

Table 16.2 Packages in the Modules of the Java 17 Core Java SE API (Cont.)

Modules	Packages Included
java.scripting	javax.script
java.security.jgss	javax.security.auth.kerberos,org.ietf.jgss
java.security.sasl	javax.security.sasl
java.sql	java.sql,javax.sql,javax.transaction.xa
java.sql.rowset	javax.sql.rowset,javax.sql.rowset.serial,javax.sql.rowset.spi
java.xml	javax.xml,javax.xml.catalog,javax.xml.datatype,javax.xml.namespace,javax.xml.parsersjavax.xml.stream,javax.xml.stream.events,javax.xml.stream.util,javax.xml.transform,javax.xml.transform.dom,javax.xml.transform.sax,javax.xml.transform.stax,javax.xml.transform.stream,javax.xml.validation,javax.xml.xpath,org.w3c.dom,org.w3c.dom.bootstrap,org.w3c.dom.events,org.w3c.dom.ls,org.w3c.dom.ranges,org.w3c.dom.views,org.xml.sax,org.xml.sax.ext,org.xml.sax.helpers
java.xml.crypto	javax.xml.crypto,javax.xml.crypto.dom,javax.xml.crypto.dsig,javax.xml.crypto.dsig.dom,javax.xml.crypto.dsig.keyinfo,javax.xml.crypto.dsig.spec

Table 16.2 Packages in the Modules of the Java 17 Core Java SE API (Cont.)

Developers should be able to map the following packages according to their respective capabilities:

Package	Description
java.awt	The AWT package provides classes for graphics output and GUI usage.
java.awt.event	Interfaces for the various events in GUIs.
java.io java.nio	Input and output options. Files are represented as objects. Data streams allow sequential access to file contents.
java.lang	A package that’s automatically included. Contains indispensable classes like string, thread, or wrapper classes.
java.net	Communication via networks. Provides classes for building client and server systems that can connect to the internet via TCP and IP, respectively.

Table 16.3 Important Packages in the Java SE

Package	Description
java.text	Support for internationalized programs. Provides classes for handling text and formatting dates and numbers.
java.util	Provides types for data structures, space and time, and parts of internationalization, as well as random numbers. Subpackages take care of regular expressions and concurrency.
javax.swing	Swing components for GUIs. This package has various subpackages.

Table 16.3 Important Packages in the Java SE (Cont.)

For a developer, you can’t avoid studying the Java API documentation at <https://docs.oracle.com/en/java/javase/17/docs/api/index.html>.

Official Interface (java and javax Packages)

The list provided by the Java documentation represents the permitted access to the library. The types are basically designed to last forever, so developers can count on still being able to run their Java programs in 100 years. But who defines the API? In essence, three sources define APIs:

- Oracle developers put new packages and types into the API.
- The *Java Community Process (JCP)* adopts a new API. Then, Oracle is not acting alone, but instead, a group works out a new API and defines its interfaces.
- The *World Wide Web Consortium (W3C)* provides an API for XML Document Object Model (DOM), for example.

A good mnemonic is that anything starting with java or javax is a permitted API, and anything else can lead to non-portable Java programs. Some classes are supported that aren’t part of the official API. These classes include, for example, various Swing classes for controlling the appearance of the interface.

Standard Extension API (javax Packages)

Some Java packages start with javax. Originally, these extension packages were intended to complement the core classes. Over time, however, many packages that initially had to be included have now migrated to the standard distribution, so that today, a fairly large proportion start with javax, but no longer represent extensions that need to be additionally installed. Sun didn’t want to rename the packages at that time, so as not to make migration more difficult. If you notice a package name with javax in the source code today, therefore, you can no longer easily determine whether an external source must be included or whether the package is already part of the distribution (and since Java version).

Truly external packages include the following packages:

- The *Java Communications API* for serial and parallel interfaces
- The *Java Telephony API*
- Speech input/output with the *Java Speech API*
- *JavaSpaces* for shared memory of different runtime environments
- *JXTA* for establishing P2P networks

The bottom line is that developers are dealing with the following libraries:

- With the official Java API
- With APIs from Java Specification Request (JSR) extensions
- With unofficial libraries, such as open-source solutions, for example, to access PDF files or control ATMs

An important role is also played by types from the `jakarta` package, which is part of Jakarta EE (formerly Java EE) and semi-official.

16.2 Simple Time Measurement and Profiling*

In addition to the convenient classes for managing date values, two static methods provide simple ways to measure times for program sections:

```
final class java.lang.System
```

- ▶ `static long currentTimeMillis()`
Returns the milliseconds elapsed since 1/1/1970, 00:00:00 Coordinated Universal Time (UTC).
- ▶ `static long nanoTime()`
Returns the time from the most accurate system timer. This method has no reference point to any date.

The difference between two time values can be used to roughly estimate the execution times of programs.



Tip

The values of `nanoTime()` are always ascending, which isn't necessarily true for `currentTimeMillis()` because Java gets the time from the operating system. System times can change, for example, when a user adjusts the time. Differences of `currentTimeMillis()` timestamps are then completely wrong and could even be negative.

16.2.1 Profilers

Where the Java virtual machine (JVM) does waste clock cycles in a program is shown by a *profiler*. Optimization can then begin at those points. *Java Mission Control* is a powerful program of the JDK and integrates a free profiler. *Java VisualVM* is another free program that can be obtained from <https://visualvm.github.io/>. On the professional and commercial side, *JProfiler* (<https://www.ej-technologies.com/products/jprofiler/overview.html>) and *YourKit* (<https://www.yourkit.com/java/profiler>) are competitors. The *Ultimate Version* of IntelliJ also includes a profiler.

16.3 The Class Class

Let's suppose we want to write a class browser. This program should display all classes belonging to the running program and furthermore additional information, such as variable assignment, declared methods, constructors, and some information about the inheritance hierarchy. For this purpose, you'll need the library class, `Class`. Instances of `Class` are objects that, for example, represent a Java class, record or a Java interface.

In this respect, Java differs from many conventional programming languages because the members of classes can be queried by the currently running program using the `Class` objects. The instances of `Class` are a restricted kind of meta-object²—containing the description of a Java type but revealing only selected information. Besides normal classes, interfaces are also represented by a `Class` object, and even arrays and primitive data types—instead of `Class`, the class name `Type` would probably have been more appropriate.

16.3.1 Obtaining a Class Object

First, for a given class, you must identify the associated `Class` object. `Class` objects themselves can only be created by the JVM. (We can't create instances because the constructor of `Class` is private.) To obtain a reference to a `Class` object, the following solutions are available:

- If an instance of the class is available, you can call the `getClass()` method of the object and get the `Class` instance of the associated class.
- Each type contains a static variable named `.class` of type `Class`, which references the associated `Class` instance.
- The ending `.class` is also permitted for primitive data types. The same `Class` object returns the static variable `TYPE` of the wrapper classes. Thus, `int.class == Integer.TYPE` is true.

² True metaclasses are classes whose only instance in each case is the regular Java class. Then, for example, the regular class variables would actually be object variables in the metaclass.

- The class method `Class.forName(String)` can query a class, and you'll obtain the associated `Class` instance as a result. If the type hasn't been loaded yet, `forName(String)` searches for and binds the class. Because searching can go wrong, a `ClassNotFoundException` is possible.
- If you already have a `Class` object but are interested in its ancestors instead, you can simply get a `Class` object for the superclass via `getSuperclass()`.

The following example shows three ways to obtain a `Class` object for `java.util.Date`:

```
Class<Date> c1 = java.util.Date.class;
System.out.println( c1 );           // class java.util.Date
Class<?> c2 = new java.util.Date().getClass();
// or Class<? extends Date> c2 = ...

System.out.println( c2 );           // class java.util.Date
try {
    Class<?> c3 = Class.forName( "java.util.Date" );
    System.out.println( c3 );       // class java.util.Date
}
catch ( ClassNotFoundException e ) { e.printStackTrace(); }
```

Listing 16.1 `src/main/java/com/tutego/insel/meta/GetClassObject.java, main()`

The variant with `forName(String)` is useful if the name of the desired class wasn't determined when the program was translated.

Otherwise, the previous technique is more catchy, and the compiler can check if the type exists. A full qualification is needed: `Class.forName("Date")` would only search for `Date` in the default package, and the return isn't a collection after all.



Example

Note that class objects for primitive elements aren't returned by `forName(String)`. The two expressions `Class.forName("boolean")` and `Class.forName(boolean.class.getName())` lead to a `ClassNotFoundException`.

```
class java.lang.Object
```

- `final Class<? extends Object> getClass()`
Returns the `Class` instance at runtime which represents the class of the object.

```
final class java.lang.Class<T>
implements Serializable, GenericDeclaration, Type, AnnotatedElement
```

- `static Class<?> forName(String className)` throws `ClassNotFoundException`
Returns the `Class` instance for the class, record or interface with the specified fully qualified name. If the type hasn't yet been required by the program, the class loader searches for and loads the class. The method never returns `null`. If the class couldn't be loaded and included, a `ClassNotFoundException` will occur. The alternative method, `forName(String name, boolean initialize, ClassLoader loader)`, also allows loading with a desired class loader. The class name must always be fully qualified.

`ClassNotFoundException` and `NoClassDefFoundError`*

A `ClassNotFoundException` can be thrown by any of the following methods:

- `forName(...)` from `Class`
- `loadClass(String name [, boolean resolve])` from `ClassLoader`
- `findSystemClass(String name)` from `ClassLoader`

An exception occurs whenever the class loader can't find the class by its class name. Thus, the trigger is when an application wants to load types dynamically, but those types aren't present.

In addition to `ClassNotFoundException`, the `NoClassDefFoundError` is a hard `LinkageError` that the JVM raises whenever it can't load a class referenced in the bytecode. For example, let's consider an expression like `new MyClass()`. When the JVM executes this code, it attempts to load the bytecode from `MyClass`. If the bytecode for `MyClass` has been removed after compilation, the JVM raises the `NoClassDefFoundError` due to the unsuccessful load attempt. Also, the error occurs if the `MyClass` class was found when loading the bytecode, but `MyClass` has a static initialization block that in turn references a class for which no class file exists.

While `ClassNotFoundException` is more common than `NoClassDefFoundError`, the exception is generally an indication that a Java Archive file (JAR file) is missing in the module path.

Problems after Applying an Obfuscator*

The fact that the compiler automatically generates bytecode according to this modified source code only leads to unexpected problems if you run an obfuscator over the program text, which subsequently modifies the bytecode and thus obscures the meaning of the program or the bytecode and renames types in the process. Obviously, an obfuscator must not rename types whose `Class` instances are requested.

Otherwise, the obfuscator must correctly replace the corresponding strings as well (but of course not replace all strings that happen to match class names).

16.3.2 A Class Is a Type

In Java, different types exist, and classes, records, interfaces, and enumeration types are represented by the JVM as `Class` objects. In the Reflection API, the `Type` interface represents all types and the only implementing class is `Class`. Below `Type` there are some subinterfaces:

- `ParameterizedType` represents generic types like `List<T>`.
- `TypeVariable<D>` represents, for example, `T` extends `Comparable<? super T>`.
- `WildcardType` represents `? super T`.
- `GenericArrayType` represents something like `T[]`.

The only method of `Type` is `getTypeName()`, and this method is just a default method that calls `toString()`. `Type` is the return of various methods in the Reflection API, such as `getGenericSuperclass()` and `getGenericInterfaces()` of the `Class` class, and many other methods listed in the Javadoc under **USE**.

16.4 The Utility Classes System and Members

In the `java.lang.System` class, methods exist for requesting and changing system variables, for redirecting the standard data streams, for determining the current time, for terminating the application, and for several other tasks. All methods are exclusively static, and an instance of `System` can't be created. In the `java.lang.Runtime` class, additional helper methods are available, such as for starting external programs or for requesting memory requirements. Unlike `System`, only one method is static in this class, namely, the singleton method `getRuntime()`, which returns the instance of `Runtime`.

java.lang.System	java.lang.Runtime
<div>+ err : PrintStream + in : InputStream + out : PrintStream + arraycopy(src : Object, srcPos : int, dest : Object, destPos : int, length : int) + clearProperty(key : String) : String + console() : Console + currentTimeMillis() : long + exit(status : int) + gc() + getProperties() : Properties + getProperty(key : String, def : String) : String + getProperty(key : String) : String + getSecurityManager() : SecurityManager + getenv(name : String) : String + getenv() : Map + identityHashCode(x : Object) : int + inheritedChannel() : Channel + load(filename : String) + loadLibrary(libname : String) + mapLibraryName(libname : String) : String + nanoTime() : long + runFinalization() + runFinalizersOnExit(Value : boolean) + setErr(err : PrintStream) + setIn(in : InputStream) + setOut(out : PrintStream) + setProperties(props : Properties) + setProperty(key : String, value : String) : String + setSecurityManager(s : SecurityManager)</div>	<div>+ addShutdownHook(hook : Thread) + availableProcessors() : int + exec(cmdarray : String[], envp : String[], dir : File) : Process + exec(command : String, envp : String[], dir File) : Process + exec(command : String) : Process + exec(cmdarray : String[]) : Process + exec(cmdarray : String[], envp : String[]) : Process + exec(command : String, envp : String[]) : Process + exit(status : int) + freeMemory() : long + gc() + getLocalizedInputStream(in : InputStream) : InputStream + getLocalizedOutputStream(out : OutputStream) : OutputStream + getRuntime() : Runtime + halt(status : int) + load(filename : String) + loadLibrary(libname : String) + maxMemory() : long + removeShutdownHook(hook : Thread) : boolean + runFinalization() + runFinalizersOnExit(value : boolean) + totalMemory() : long + traceInstructions(on : boolean) + traceMethodCalls(on : boolean)</div>

Figure 16.4 Members of the System and Runtime Classes

Remark

All in all, the `System` and `Runtime` classes don't seem particularly orderly, as shown in Figure 16.4; you may think everything that doesn't fit elsewhere can be found in these two classes. Also, some methods of one class would be just as good in the other class. The fact that the static method `System.arraycopy(...)` for copying arrays isn't located in `java.util.Arrays` can only be explained historically. Furthermore, `System.exit(int)` redirects to `Runtime.getRuntime().exit(int)`. Some methods are obsolete and distributed differently: The `exec(...)` of `Runtime` to start external processes is handled by a new class (`ProcessBuilder`), and the question about the memory state or the number of processors is answered by `MBeans`, such as `ManagementFactory.getOperatingSystemMXBean().getAvailableProcessors()`. But API design is like gambling: One rash action, and you've lost the farm.

16.4.1 Memory of the Java Virtual Machine

The `Runtime` object includes the following three methods that provide information about the memory of the JVM:

- `maxMemory()` returns the maximum number of bytes available for the JVM. The value can be set when calling the JVM with `-Xmx` in the command line.
- `totalMemory()` is what is currently used and can grow to `maxMemory()`. Basically, this memory limit can also shrink again. The following applies: `maxMemory() > totalMemory()`.
- `freeMemory()` indicates the memory that is free for new objects and also provokes the automatic garbage collection process. The following applies: `totalMemory() > freeMemory()`. However, `freeMemory()` isn't the entire freely available memory area because the "share" of `maxMemory()` is still missing.

Two pieces of information are missing and therefore must be calculated:

- **Used memory:**
`long usedMemory = Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory();`
- **Free total memory:**
`long totalFreeMemory = Runtime.getRuntime().maxMemory() - usedMemory;`

Example

The following example outputs information about the memory on a computer:

```
long totalMemory    = Runtime.getRuntime().totalMemory();
long freeMemory     = Runtime.getRuntime().freeMemory();
long maxMemory      = Runtime.getRuntime().maxMemory();
long usedMemory     = totalMemory - freeMemory;
```



```
long totalFreeMemory = maxMemory - usedMemory;

System.out.printf(
    "total=%d MiB, free=%d MiB, max=%d MiB, used=%d MiB, total free=%d MiB%n",
    totalMemory >> 20, freeMemory >> 20, maxMemory >> 20,
    usedMemory >> 20, totalFreeMemory >> 20 );

The result is the following output:

total=126 MiB, free=124 MiB, max=2016 MiB, used=1 MiB, total free=2014 MiB
```

16.4.2 Number of CPUs or Cores

The Runtime method `availableProcessors()` returns the number of logical processors or cores.

[Ex]

Example

Print the number of processors/cores:

```
System.out.println( Runtime.getRuntime().availableProcessors() ); // 4
```

16.4.3 System Properties of the Java Environment

The Java environment manages system properties, such as path separators or virtual machine versions in the `java.util.Properties` object. The static method `System.getProperties()` queries these system properties and returns the filled `Properties` object. However, the `Properties` object isn't absolutely necessary for querying individual properties: `System.getProperty(...)` directly queries a property.

[Ex]

Example

The following example outputs the name of the operating system:

```
System.out.println( System.getProperty( "os.name" ) ); // e.g., Windows 10
```

The following example outputs all system properties on the screen:

```
System.getProperties().list( System.out );
```

An excerpt of the result is shown in the following output:

```
-- listing properties --
sun.desktop=windows
awt.toolkit=sun.awt.windows.WToolkit
java.specification.version=9
file.encoding.pkg=sun.io
sun.cpu.isalist=amd64
...
```

Table 16.4 shows a list of the important standard system properties.

Key	Meaning
java.version	Version of the Java Runtime Environment (JRE)
java.class.path	The current classpath
java.library.path	Path for native libraries
java.io.tmpdir	Path for temporary files
os.name	Name of the operating system
file.separator	Separator of the path segments, for example / (Unix) or \ (Windows)
path.separator	Separator for path specifications, such as : (Unix) or ; (Windows)
line.separator	Newline character (string)
user.name	Name of the logged-on user
user.home	Home directory of the user
user.dir	Current directory of the user

Table 16.4 Standard System Properties

Application Programming Interface Documentation

A few more keys are listed in the API documentation at `System.getProperties()`. Some variables are also accessible in other ways, such as through the `File` class.

```
final class java.lang.System
```

- ▶ `static String getProperty(String key)`
Returns the assignment of a system property. If the key is null or empty, a `NullPointerException` or an `IllegalArgumentException`, respectively, will occur.
- ▶ `static String getProperty(String key, String def)`
Returns the assignment of a system property. If the property isn't present, the method returns the string `def`, which is the default value. For the exceptions, the same applies as for `getProperty(String)`.
- ▶ `static String setProperty(String key, String value)`
Reassigns a system property. The return is the previous assignment or null if no previous assignment exists.
- ▶ `static String clearProperty(String key)`
Deletes a system property from the list. The return is the previous assignment or null if no previous assignment exists.

- `static Properties getProperties()`
Returns a `Properties` object filled with the current system assignments.

16.4.4 Setting Custom Properties from the Console*

Properties can also be set from the console during the program startup. This approach is convenient for a configuration that controls, for example, the behavior of a program. On the command line, `-D` specifies the name of the property and, after an equal sign (without whitespace), its value. Consider the following example:

```
$ java -DLOG -DUSER=Chris -DSIZE=100 com.tutego.insel.lang.SetProperty
```

The `LOG` property is “simply exists” but with no assigned value. The next two properties, `USER` and `SIZE`, are associated with values that are first of type `String` and must be further processed by the program. This information doesn’t appear with the argument list in the static `main(String[])` method because it precedes the name of the class and is already processed by the JRE.

To read the properties, we’ll use the familiar `System.getProperty(...)` method in the following example:

```
Optional<String> logProperty = ofNullable( System.getProperty( "LOG" ) );
Optional<String> user nameProperty = ofNullable( System.getProperty( "USER" ) );
Optional<String> sizeProperty = ofNullable( System.getProperty( "SIZE" ) );

System.out.println( logProperty.isPresent() );           // true
user nameProperty.ifPresent( System.out::println );     //
Chris
sizeProperty.map( Integer::parseInt ).ifPresent( System.out::println ); // 100
System.out.println( System.getProperty( "DEBUG", "false" ) ); //
false
```

Listing 16.2 `com/tutego/insel/lang/SetProperty.java, main()`

In return, you’ll receive a string indicating the value via `getProperty(String)`. If no property of that name exists at all, you’ll get `null` instead. In this way, we can know if this value was set at all. So, a simple `null` test tells us whether `logProperty` is present or not. Instead of `-DLOG`, `-DLOG=` also returns the same result because the associated value is the empty string. Since all properties are of type `String` to begin with, `user nameProperty` is easy to output, and you’ll get either `null` or the string specified after `=`. If the types aren’t strings, they must be processed further, for example, with `Integer.parseInt()`, `Double.parseDouble()`, and so on. The `System.getProperty(String, String)` method, which is passed two arguments, is pretty useful in this case because the second argument represents a default value. Thus, a default value can always be assumed.

`Boolean.getBoolean(String)`

In the case of properties that are assigned truth values, the following statement can be written:

```
boolean b = Boolean.parseBoolean( System.getProperty( property ) ); // (*)
```

There’s another variant for the truth values. The static method `Boolean.getBoolean(String)` searches for a property with the specified name in the system properties. Thus, the following is analogous to the line (*):

```
boolean b = Boolean.getBoolean( property );
```

You might be surprised to find this static method in the wrapper class `Boolean` because property access has nothing to do with wrapper objects and the class actually goes beyond its area of responsibility in this case.

Compared to a separate, direct `System` query, `getBoolean(String)` also has a disadvantage in that, when it returns `false`, you can’t distinguish whether the property simply doesn’t exist or whether the property is assigned the value `false`. Also, incorrectly set values like `-DP=false` always result in `false`.³

```
final class java.lang.Boolean
implements Serializable, Comparable<Boolean>
```

- `static boolean getBoolean(String name)`
Reads a system property named `name` and returns `true` if the value of the property is equal to the string `"true"`. The return value is `false` if the value of the system property is `false`, if the property doesn’t exist, or if the property is `null`.

16.4.5 Newline Characters and `line.separator`

To move from the end of one line to the beginning of the next, a *newline* is inserted. The character for a new line doesn’t have to be a single character; several characters may also be necessary. Unfortunately for programmers, the number of characters for a new-line sequence depends on the architecture, for instance:

- Unix, macOS: Line feed (LF for short), `\n`
- Windows: Carriage return (CR for short) and line feed

The control code for a carriage return is 13 (0x0D); the control code for a line feed is 10 (0x0A). Java also assigns its own escape sequences for these characters: `\r` for carriage returns and `\n` for line feeds. (The `\f` sequence is for a form feed, also called a “page feed,” which doesn’t play any role in line breaks).

³ This confusion is due to the implementation: `Boolean.valueOf("false")` returns `false` just like `Boolean.valueOf("")` or `Boolean.valueOf(null)`.

In Java, you can obtain a newline character or a newline string from the system in one of the following three ways:

- By calling `System.getProperty("line.separator")`
- By calling `System.lineSeparator()`
- You don't always have to query the character (or, strictly speaking, a possible string of characters) individually. If the character is part of a formatted output at the formatter, `String.format(...)` or `printf(...)`, the format specifier `%n` stands for exactly the newline string stored in the system.

16.4.6 Environment Variables of the Operating System

Almost every operating system uses the concept of *environment variables*; for example, `PATH` is known for the search path for applications on Windows and Unix. Java enables access to these system environment variables. Two static methods are used for this purpose:

```
final class java.lang.System
```

- ▶ `static Map<String, String> getEnv()`
Reads a set of `<string, string>` pairs with all system properties.
- ▶ `static String getEnv(String name)`
Reads a system property named `name`. If the property doesn't exist, the return will be `null`.

Variable Name	Description	Example
COMPUTERNAME	Name of the computer	<i>MOE</i>
HOMEDRIVE	Drive of the user directory	<i>C:</i>
HOMEPATH	Path of the user directory	<i>\Users\Christian</i>
OS	Name of the operating system*	<i>Windows_NT</i>
PATH	Search path	<i>C:\windows\SYSTEM32; C:\windows ...</i>
PATHEXT	File extensions that represent executable programs	<i>.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC</i>
SYSTEMDRIVE	Drive of the operating system	<i>C:</i>
TEMP and also TMP	Temporary directory	<i>C:\Users\CHRIST~1\AppData\Local\Temp</i>
USERDOMAIN	Domain of the user	<i>MOE</i>

Table 16.5 Selection of Some Environment Variables Available in Windows

Variable Name	Description	Example
USERNAME	Name of the user	<i>Christian</i>
USERPROFILE	Profile directory	<i>C:\Users\Christian</i>
WINDIR	Directory of the operating system	<i>C:\windows</i>
* The result differs from <code>System.getProperty("os.name")</code> , which already returns “Windows 10” for Windows 10.		

Table 16.5 Selection of Some Environment Variables Available in Windows (Cont.)

Some variables are also accessible via the system properties, for instance, with `System.getProperties()`, `System.getProperty(...)`, and so on.

Example

The following example outputs the environment variables of the system:


```
Map<String,String> map = System.getenv();  
map.forEach( (k, v) -> System.out.printf( "%s=%s\n", k, v ) );
```

Ex

16.5 The Languages of Different Countries

When developers start with console or GUI output, they often hardwire the output to a local language. If the language changes, the software can't handle other country-specific rules, for example, when formatting floats. Developing “multilingual” programs that provide localized outputs in different languages is not too difficult. Basically, you'll replace all language-dependent strings and formatting of data with code that takes into account country-specific output formats and rules. Java offers a solution for these cases: on one hand, you can define a language that then specifies rules according to which the Java API can automatically format data, and on the other hand, you can allow language-dependent parts to be swapped out to resource files.

16.5.1 Regional Languages via Locale Objects

In Java, `Locale` objects represent languages in geographic, political, or cultural regions. The language and the region must be separated because a region or a country doesn't always clearly specify the language. For Canada, in the province of Quebec, the French edition is relevant, which, of course, differs from the English edition. Each of these language-specific properties can be encapsulated in a special object. These `Locale` objects are then passed to a `Formatter` that's located behind `String.format(...)` and `printf(...)` or passed to a `Scanner`. These outputs are referred to as *locale sensitive*.

Building Locale Objects

Locale objects are always created with the name of the language and optionally with the name of the country or a region and variant. The `Locale` class provides three ways to build the objects:

- Using the `Locale` constructor (deprecated in Java 19)
- Using the nested `Builder` class of `Locale` uses the builder pattern to build new `Locale` objects
- Using the `Locale` method `forLanguageTag(...)` and a string identifier



Example

Country abbreviations are specified in the constructor of the `Locale` class, for example, for a language object for Great Britain or France. Consider the following examples:

```
Locale greatBritain = new Locale( "en", "GB" );
Locale french       = new Locale( "fr" );
```

In the second example, we don't care about the country. We're simply choosing French as the language, no matter what part of the world.

Languages are identified by 2-letter abbreviations from the ISO 639 code⁴ (*ISO Language Code*), and country names are 2-letter abbreviations described in ISO 3166⁵ (*ISO Country Code*).



Example

Three variants for building `Locale.JAPANESE` are found in the following example:

```
Locale loc1 = new Locale( "ja" );
Locale loc2 = new Locale.Builder().setLanguage( "ja" ).build();
Locale loc3 = Locale.forLanguageTag( "ja" );
final class java.util.Locale
implements Cloneable, Serializable
```

- `Locale(String language)`
Creates a new `Locale` object for the language given by the ISO-693 standard. Invalid identifiers aren't recognized.
- `Locale(String language, String country)`
Creates a `Locale` object for a language according to ISO 693 and a country according to the ISO 3166 standard.

⁴ https://en.wikipedia.org/wiki/List_of_ISO_639-1_codes
⁵ https://en.wikipedia.org/wiki/ISO_3166-1

- `Locale(String language, String country, String variant)`
Creates a `Locale` object for a language, a country, and a variant. `variant` is a vendor-dependent specification like "WIN" or "MAC."

The static `Locale.getDefault()` method returns the currently set language. For the running JVM, `Locale.setDefault(Locale)` can change the language.

The `Locale` class has more methods; developers should study the Javadoc for the `Builder`, for `forLanguageTag(...)` and the new extensions and filter methods.⁶

Constants for Some Languages

The `Locale` class has constants for commonly occurring languages, with an option for specifying countries. The constants for countries and languages include, for example, `CANADA`, `CHINA`, `FRENCH`, `GERMAN`, `ITALIAN`, `KOREAN`, `TAIWAN`, `UK` and `US`. Behind an abbreviation like `Locale.UK`, nothing else exists except for the initialization with `new Locale("en", "GB")`.

Methods That Accept Instances of Locale

`Locale` objects are actually not interesting as objects—they do have methods, but more exciting is its use as a type for the identification of a language. Dozens of methods in the Java library accept `Locale` objects and adjust their behaviors based on them. Examples include `printf(Locale, ...)`, `format(Locale, ...)`, and `toLowerCase(Locale)`.

Tip

If no variant of a format or parse method exists with a `Locale` object; the method usually doesn't support language-dependent behavior. The same limitation applies to objects that don't accept a `Locale` via a constructor or setter. `Double.toString(...)` is one such example, as is `Double.parseDouble(...)`. In internationalized applications, these methods will rarely be found. Also, string concatenation with, for example, a float isn't permitted (because a `Double` method is called internally), and using `String.format(...)` is definitely a better option.

Methods of Locale*

`Locale` objects provide a number of methods that reveal the ISO-639 code of the country, for example.

Example

The following example outputs the `Locale` information accessible for languages in selected countries. The objects `System.out` and `Locale.*` are imported statically:

⁶ Oracle's Java tutorial describes these extensions at <http://docs.oracle.com/javase/tutorial/i18n/locale/index.html>.


```
out.println(GERMANY.getCountry());           // DE
out.println(GERMANY.getLanguage());          // de
out.println(GERMANY.getVariant());           //
out.println(GERMANY.getISO3Country());       // DEU
out.println(GERMANY.getISO3Language());      // deu
out.println(CANADA.getDisplayCountry());     // Canada
out.println(GERMANY.getDisplayLanguage());   // German
out.println(GERMANY.getDisplayName());       // German (Germany)
out.println(CANADA.getDisplayName());        // English (Canada)
out.println(GERMANY.getDisplayName(FRENCH)); // allemand (Allemagne)
out.println(CANADA.getDisplayName(FRENCH));  // anglais (Canada)
```

Listing 16.3 src/main/java/com/tutego/insel/locale/GermanyLocal.java, main()

Static methods also exist for querying Locale objects:

```
final class java.util.Locale
implements Cloneable, Serializable
```

- ▶ static Locale getDefault()
Returns the language preset by the JVM, which defaults to the operating system.
- ▶ static Locale[] getAvailableLocales()
Returns a list of all installed Locale objects. The field contains at least Locale.US and about 160 entries.
- ▶ static String[] getISOCountries()
Returns an array of all 2-letter ISO-3166 country codes.
- ▶ static Set<String> getISOCountries(Locale.IsoCountryCode type)
Returns a set with all ISO-3166 country codes, where the IsoCountryCode list determines the following: PART1_ALPHA2 returns the code of 2 letters, PART1_ALPHA3 of 3 letters, PART3 of 4 letters.

On the other hand, other methods provide abbreviations according to ISO standards:

```
final class java.util.Locale
implements Cloneable, Serializable
```

- ▶ String getCountry()
Returns the country abbreviation according to the ISO-3166 2-letter code.
- ▶ String getLanguage()
Returns the abbreviation of the language in ISO-639 code.
- ▶ String getISO3Country()
Returns the ISO abbreviation of the country of these settings and throws a MissingResourceException if the ISO abbreviation isn't available.

- ▶ String getISO3Language()
Returns the ISO abbreviation of the language of these settings and throws a MissingResourceException if the ISO abbreviation isn't available.
- ▶ String getVariant()
Returns the abbreviation of the variant or an empty string.

These methods provide abbreviations, but they aren't intended for human-readable output. For various get*() methods, therefore, corresponding getDisplay*() methods exist:

```
final class java.util.Locale
implements Cloneable, Serializable
```

- ▶ String getDisplayCountry(Locale inLocale)
final String getDisplayCountry()
Returns the name of the country for screen outputs for a language or Locale.getDefault().
- ▶ String getDisplayLanguage(Locale inLocale)
String getDisplayLanguage()
Returns the name of the screen output language for a given Locale or Locale.getDefault().
- ▶ String getDisplayName(Locale inLocale)
final String getDisplayName()
Returns the name of the settings for a language or Locale.getDefault().
- ▶ String getDisplayVariant(Locale inLocale)
final String getDisplayVariant()
Returns the name of the variant for a language or Locale.getDefault().

16.6 Overview of Important Date Classes

Because date calculations are convoluted entities, we can be grateful to the developers of Java for providing many classes for date calculation and formatting. The developers have kept the classes abstract enough to allow for local specifics like output formatting, parsing, time zones, or daylight saving time/winter time in different calendars.

Prior to Java 1.1, only the java.util.Date class was available for displaying and manipulating date values, and this class had to carry out several tasks:

- Creation of a date/time object from year, month, day, minute, and second
- Querying day, month, year, and so on with an accuracy of milliseconds
- Processing and output of date strings

Since the `Date` class wasn't quite bug-free and internationalized, new classes were introduced in JDK 1.1, namely, the following:

- `Calendar` takes on `Date`'s task of converting between different date representations and time scales. The `GregorianCalendar` subclass is created directly.
- `DateFormat` breaks up date strings and formats the output. Date formats also depend on the country, which Java represents through `Locale` objects, and on a time zone, which is represented by the instances of the `TimeZone` class.

In Java 8, another date library was added with entirely new types. Finally, date and time can be represented separately:

- `LocalDate`, `LocalTime`, and `LocalDateTime` are the temporal classes for a date, for a time, and for a combination of date and time, respectively.
- `Period` and `Duration` represent intervals.

16.6.1 Unix Time: January 1, 1970

January 1, 1970, was a Thursday with groundbreaking changes: The British rejoiced that the age of majority dropped from 24 to 18, and as in every year, and people everywhere woke up to massive hangovers from the night before. For us, however, a technical innovation is of concern: The date of 1/1/1970, 0:00:00 UTC is also referred to as the *Unix epoch*, and a *Unix time* is described in relation to this time in terms of seconds. For example, 100,000,000 seconds after 1/1/1970 is March 3, 1973, at 09:46:40. The *Unix Billionnium* was celebrated 1,000,000,000 seconds after Jan. 1, 1970, namely, on Sept. 9, 2001, at 01:46:40.

16.6.2 System.currentTimeMillis()

Unix time is also an important concept for us Java developers because many times in Java are relative to this date. The timestamp 0 refers to 1/1/1970 0:00:00 Greenwich Mean Time. The `System.currentTimeMillis()` method returns the past milliseconds—not seconds!—relative to 1/1/1970, 00:00 UTC, although your operating system's clock may not be that accurate. The number of milliseconds is represented in a `long` (i.e., in 64 bits), which will suffice for about 300 million years.



Warning

The values of `currentTimeMillis()` don't necessarily ascend because Java gets the time from the operating system, where the system time can change. A user can adjust the time, or a service such as the *Network Time Protocol (NTP)* takes over this task. Differences of `currentTimeMillis()` timestamps are then completely wrong and could even be negative. An alternative is `nanoTime()`, which has no reference point, is more precise, and is always ascending.⁷

⁷ <http://stackoverflow.com/questions/351565/system-currenttimemillis-vs-system-nanotime> goes into more details and provides links to internal implementations.

16.6.3 Simple Time Conversions via TimeUnit

A time duration in Java is often expressed in terms of milliseconds. 1,000 milliseconds correspond to 1 second, 1,000 × 60 milliseconds to 1 minute, and so on. However, all those large numbers aren't easy to read, which is why `TimeUnit` objects are used with their `to*(...)` methods for the purpose of conversion. Java declares the following constants in `TimeUnit`: `NANOSECONDS`, `MICROSECONDS`, `MILLISECONDS`, `DAYS`, `HOURS`, `SECONDS`, and `MINUTES`.

Each of the enumeration elements defines the conversion methods, for instance, `toDays(...)`, `toHours(...)`, `toMicros(...)`, `toMillis(...)`, `toMinutes(...)`, `toNanos(...)`, and `toSeconds(...)`. These methods receive a `long` and return a `long` in the corresponding unit. In addition, two `convert(...)` methods convert from one unit to another.

Example

The following example converts 23,746,387 milliseconds to hours:

```
int v = 23_746_387;
System.out.println( TimeUnit.MILLISECONDS.toHours( v ) ); // 6
System.out.println( TimeUnit.HOURS.convert( v,
    TimeUnit.MILLISECONDS ) ); // 6
enum java.util.concurrent.TimeUnit
extends Enum<TimeUnit>
implements Serializable, Comparable<TimeUnit>
```

- ▶ `NANOSECONDS`, `MICROSECONDS`, `MILLISECONDS`, `SECONDS`, `MINUTES`, `HOURS`, `DAYS` `TimeUnit` enumeration elements.
- ▶ `long toDays(long duration)`
- ▶ `long toHours(long duration)`
- ▶ `long toMicros(long duration)`
- ▶ `long toMillis(long duration)`
- ▶ `long toMinutes(long duration)`
- ▶ `long toNanos(long duration)`
- ▶ `long toSeconds(long duration)`
- ▶ `long convert(long sourceDuration, TimeUnit sourceUnit)`
Returns `sourceUnit.to*(sourceDuration)`, where `*` represents the respective unit. For example, this method returns `HOURS.convert(sourceDuration, sourceUnit)`, then `sourceUnit.toHours(1)`. The readability of this method isn't ideal, so the other methods should be preferred. Results may be truncated, not rounded. If an overflow occurs, no `ArithmeticException` will follow.



► `long convert(Duration duration)`
Converts the passed duration into the time unit that represents the current `TimeUnit`. For example, `TimeUnit.MINUTES.convert(Duration.ofHours(12))` returns 720. Thus, for example, `aunit.convert(Duration.ofNanos(n))` and `aunit.convert(n, NANOSECONDS)` are the same.

16.7 Date-Time API

The `java.time` package is based on the standardized calendar system of ISO-8601, and this covers how a date is represented, including time, date and time, UTC, time intervals (duration/time span), and time zones. The implementation is based on the Gregorian calendar, although other calendar types are also conceivable. Java’s calendar system can use other standards or implementations as well, including the *Unicode Common Locale Data Repository (CLDR)* for localizing days of the week or the *Time-Zone Database (TZDB)*, which documents all time zone changes since 1970.

16.7.1 Initial Overview

The central temporal types from the Date-Time API can be quickly documented:

Type	Description	Field(s)
<code>LocalDate</code>	Represents a common date	Years, months, and days
<code>LocalTime</code>	Represents a common time	Hours, minutes, seconds, and nanoseconds
<code>LocalDateTime</code>	Combination of date and time	Years, months, days, hours, minutes, seconds, and nanoseconds
<code>Period</code>	Duration between two <code>LocalDates</code>	Years, months, and days
<code>Year</code>	Year only	Year
<code>Month</code>	Month only	Month
<code>MonthDay</code>	Month and day only	Month, day
<code>OffsetTime</code>	Time with time zone	Hours, minutes, seconds, nanoseconds, and zone offset
<code>OffsetDateTime</code>	Date and time with time zone as UTC offset	Year, month, day, hours, minutes, seconds, nanoseconds, and zone offsets

Table 16.6 All Temporal Classes from `java.time`

Type	Description	Field(s)
<code>ZonedDateTime</code>	Date and time with time zone as ID and offset	Year, month, day, hours, minutes, seconds, nanoseconds, and zone info
<code>Instant</code>	Time (continuous machine time)	Nanoseconds
<code>Duration</code>	Time interval between two instants	Seconds/nanoseconds

Table 16.6 All Temporal Classes from `java.time` (Cont.)

16.7.2 Human Time and Machine Time

Date and time, which we as humans understand in units such as days and minutes, is referred to as *human time*, while the continuous time of the computer, which has a resolution in the nanosecond range, is called *machine time*. The machine time begins at a time we call an *epoch*, namely, the Unix epoch.

From Chapter 7, Section 7.2.5, you learned how most classes are made for humans and that only `Instant/Duration` refers to machine time. `LocalDate`, `LocalTime`, and `LocalDateTime` represent human time without reference to a time zone, whereas `ZonedDateTime` does reference a time zone. When choosing the right time classes for a task, the first consideration is, of course, whether to represent human time or machine time. This choice is followed by questions about exactly which fields are needed and whether a time zone is relevant or not. For example, if the execution time is to be measured, you don’t need to know on which date the measurement started and ended; in this case, `Duration` would be correct, unlike `Period`.

Example

Examples for explicit formatting and default formatting for the US locale:

```
LocalDate now = LocalDate.now();  
System.out.println( now ); // e.g., 2023-01-31  
System.out.printf( "%d. %s %d%n",  
                    now.getDayOfMonth(), now.getMonth(), now.getYear() );  
// e.g., 31. JANUARY 2023  
LocalDate bdayMLKing = LocalDate.of( 1929, Month.JANUARY, 15 );  
DateTimeFormatter formatter =  
    DateTimeFormatter.ofLocalizedDate( FormatStyle.MEDIUM );  
System.out.println( bdayMLKing.format( formatter ) ); // Jan 15, 1929
```

The `getMonth()` method on a `LocalDate` returns a `java.time.Month` object as the result, and these are enumerations. The `toString()` representation returns the constant in uppercase letters.

All classes are based on the ISO system by default. Other calendar systems, such as the Japanese calendar, are created using types from `java.time.chrono`, and of course, entirely new systems are also possible.



Example

Output for the Japanese calendar:

```
ChronoLocalDate now = JapaneseChronology.INSTANCE.dateNow();
System.out.println( now );           // Japanese Reiwa 4-01-31
```

Package Overview

The types of the Date-Time API are distributed among different packages:

- `java.time`
Contains the standard classes like `LocalTime` and `Instant`. All types are based on the ISO-8601 calendar system, commonly known as the “Gregorian calendar.” This calendar is extended by the *Proleptic Gregorian Calendar*. This calendar is also valid for the time before 1582 (the introduction of this calendar), so that a consistent timeline can be used.
- `java.time.chrono`
In this package, you’ll find predefined alternative (i.e., non-ISO) calendar systems, such as the Japanese calendar, the Thai-Buddhist calendar, the Islamic calendar, and a few others.
- `java.time.format`
Classes for formatting and parsing date and time, such as the `DateTimeFormatter`.
- `java.time.zone`
Supporting classes for time zones, such as `ZonedDateTime`.
- `java.time.temporal`
Deeper API that allows access and modification of individual fields of a date/time value.

Design Principles

Before we get into the individual classes, let’s look at some design principles because all types of the Date-Time API follow recurring patterns. The first and most important property is that all objects are *immutable*; that is, they can’t be changed. In contrast, with the “old” API, `Date` and the `Calendar` classes were mutable, with sometimes devastating consequences. If these objects are passed around and changed, incalculable side effects can occur. The classes of the new Date-Time API are immutable, and so the date/time classes like `LocalTime` or `Instant` are opposed to mutable types like `Date` or `Calendar`. All methods that look as if they permitted changes now instead create new objects with the desired changes. Side effects are therefore absent, and all types are thread safe.

Immutability is a design property as is the fact that `null` isn’t permitted as an argument. In the Java API, `null` is often accepted because it expresses something optional, but the Date-Time API usually penalizes `null` with a `NullPointerException`. The fact that `null` isn’t in use as an argument and not as a return benefits another property: The code can be mostly written with a fluent API (i.e., cascaded calls) since many methods return the `this` reference, as is known from `StringBuilder`.

Added to these more technical features is a consistent naming that is different from the naming of the well-known `JavaBeans`. So, no constructors and no setters exist (immutable classes don’t need them), but instead, patterns adhere to many types from the Date-Time API:

Method	Class/Instance Method	Basic Meaning
<code>now()</code>	Static	Returns an object with current time/current date
<code>of*()</code>	Static	Creates new objects
<code>from</code>	Static	Creates new objects from other representations
<code>parse*()</code>	Static	Creates a new object from a string representation
<code>format()</code>	Instance	Formats and returns a string
<code>get*()</code>	Instance	Returns fields of an object
<code>is*()</code>	Instance	Queries the status of an object
<code>with*()</code>	Instance	Returns an instance of the object with a changed state
<code>plus*()</code>	Instance	Returns an instance of the object with a totaled state
<code>minus*()</code>	Instance	Returns an instance of the object with a reduced state
<code>to*()</code>	Instance	Converts an object to a new type
<code>at*()</code>	Instance	Combines this object with another object
<code>*Into()</code>	Instance	Combines an own object with another target object

Table 16.7 Name Patterns in the Date-Time API

You’ve already used the `now()` method in one of the first examples in this section, and this method returns the current date, for example. Other creator methods are prefixed with `of`, `from`, or `with`; no constructors exist. The methods of the `with*()` type assume the role of setters.

16.7.3 The `LocalDate` Date Class

A date (without a time zone) is represented by the `LocalDate` class. This class can be used to represent a birth date, for example.

A temporal object can be created using the static of (...) factory methods and derived via `ofInstant(Instant instant, ZoneId zone)` or from another temporal object. Interesting are the methods that work with a `TemporalAdjuster`.

Equipped with these objects, you can use various getters and query individual fields, such as `getDayOfMonth()`; `getDayOfYear()` (return `int`); `getDayOfWeek()`, which returns an enumeration of type `DayOfWeek`; and `getMonth()`, which returns an enumeration of type `Month`. Furthermore, other methods include `long toEpochDay()` and `long toEpochSecond(LocalTime time, ZoneOffset offset)`.



Example

Find the next Saturday from now:

```
LocalDate today = LocalDate.now();
LocalDate nextSaturday =
    today.with( TemporalAdjusters.next(DayOfWeek.SATURDAY) );
System.out.printf( "Today is %s, and next Saturday is %s",
                    today, nextSaturday );
```

In addition, some methods return new `LocalDate` objects with `minus*()` or `plus*()` if, for example, a number of years should be returned with `minusYear(long yearsToSubtract)`. By negating the sign, the opposite method can also be used. In other words, `LocalDate.now().minusMonths(1)` provides the same result as `LocalDate.now().plusMonths(-1)`. The `with*()` methods reassign a field and return a modified new `LocalDate` object.

From a `LocalDate` object, you can create other temporal objects: `atTime(...)`, for example, returns `LocalDateTime` objects in which certain time fields are assigned. `atTime(int hour, int minute)` is such an example. With `until(...)`, a time duration of the `Period` type can be returned. Two methods that provide a stream of `LocalDate` objects up to an endpoint are also interesting:

- `Stream<LocalDate> datesUntil(LocalDate endExclusive)`
- `Stream<LocalDate> datesUntil(LocalDate endExclusive, Period step)`

16.8 Logging with Java

Logging information about program states is important for reconstructing and understanding the flow and states of a program at a later time. A logging API can write messages to the console or to external storage, such as text files, XML files, and databases, or to distribute these messages via chat.

16.8.1 Logging Application Programming Interfaces

Regarding logging libraries and APIs, the Java world is unfortunately divided. Since the Java standard library didn't provide a logging API in its first versions, the open-source library *log4j* quickly filled this gap. This library is used in almost every major Java project today. When the Logging API moved into Java 1.4 with Java Specification Request (JSR) 47, the Java community was surprised to find that `java.util.logging (JUL)` was neither API-compatible with the popular *Log4j* nor as powerful as *Log4j*.⁸

Over the years, the picture has changed. While in the early day's developers relied exclusively on *log4j*, more and more projects are now using *JUL*. One of the reasons is that some developers want to avoid external dependencies (although this doesn't really work since almost every included Java library is based on *log4j*). Another reason is that for many projects *JUL* is simply sufficient. In practice, for larger projects, as a result, multiple logging configurations overcrowd their own programs, as each logging implementation is configured differently.

16.8.2 Logging with java.util.logging

The Java logging API can write a message that you can then use for maintenance or security checks. The API is simple:

```
package com.tutego.insel.logging;
```

```
import static java.time.temporal.ChronoUnit.MILLIS;
import static java.time.Instant.now;
import java.time.Instant;
import java.util.logging.Level;
import java.util.logging.Logger;
```

```
public class JULDemo {
```

```
    private static final Logger log = Logger.getLogger( JULDemo.class.getName() );
```

```
    public static void main( String[] args ) {
        Instant start = now();
        log.info( "About to start" );
```

```
        try {
            log.log( Level.INFO, "Lets try to throw {0}", "null" );
            throw null;
        }
```

⁸ The standard logging API, on the other hand, provides only basics like hierarchical loggers. Standard logging doesn't come close to the power of *log4j* with its large number of writers in files, syslog/NT loggers, databases, and dispatch over the network.


```
        catch ( Exception e ) {
            log.log( Level.SEVERE, "Oh Oh", e );
        }
        log.info( () -
> String.format( "Runtime: %s ms", start.until(now(), MILLIS )) );
    }
}
```

Listing 16.4 src/main/java/com/tutego/insel/logging/CULDemo.java, JULDemo

When you run the example, the following warning appears on the console:

```
Jan. 24, 2022 7:47:46 PM com.tutego.insel.logging.JULDemo main
INFO: About to start
Jan. 24, 2022 7:47:46 PM com.tutego.insel.logging.JULDemo main
INFO: Lets try to throw null
Jan. 24, 2022 7:47:46 PM com.tutego.insel.logging.JULDemo main
SEVERE: Oh Oh
java.lang.NullPointerException: Cannot throw exception because "null" is null
at com.tutego.insel.logging.JULDemo.main(JULDemo.java:20)

Jan. 24, 2022 7:47:46 PM com.tutego.insel.logging.JULDemo main
INFO: Runtime: 35 ms
```

The Logger Object

The `Logger` object is a central element that can be retrieved via `Logger.getAnonymousLogger()` or via `Logger.getLogger(String name)`, where `name` is usually assigned the fully qualified class name. Often, the `Logger` object is declared as a private static final variable in the class.

Logging with Log Level

Not every message is equally important. Some messages are useful for debugging or because of timing measurements, but exceptions in the `catch` branches are hugely important. To support different levels of detail, you can specify a *log level*. This level determines how “serious” the error or a message is, which is important later when errors are sorted according to their urgency. Log levels are declared as constants in the `Level` class⁹ in the following ways:

- `FINEST` (smallest level)
- `FINER`
- `FINE`

⁹ Since the logging framework joined Java in version 1.4, it doesn’t yet use typed enumerations, which have only been available since Java 5.

- `CONFIG`
- `INFO`
- `WARNING`
- `SEVERE` (highest level)

For the logging process itself, the `Logger` class provides the general method `log(Level level, String msg)` or a separate method for each level.

Level	Call via log(...)	Special Log Method
SEVERE	<code>log(Level.SEVERE, msg)</code>	<code>severe(String msg)</code>
WARNING	<code>log(Level.WARNING, msg)</code>	<code>warning(String msg)</code>
INFO	<code>log(Level.INFO, msg)</code>	<code>info(String msg)</code>
CONFIG	<code>log(Level.CONFIG, msg)</code>	<code>config(String msg)</code>
FINE	<code>log(Level.FINE, msg)</code>	<code>fine(String msg)</code>
FINER	<code>log(Level.FINER, msg)</code>	<code>finer(String msg)</code>
FINEST	<code>log(Level.FINEST, msg)</code>	<code>finest(String msg)</code>

Table 16.8 Log Levels and Methods

All these methods send a message of type `String`. If an exception and the associated stack trace must be logged, developers must use the following logger method, which is also used in the example:

► `void log(Level level, String msg, Throwable thrown)`

The variants of `severe(...)`, `warning(...)`, and so on are not overloaded with a `Throwable` parameter type.

16.9 Maven: Resolving Build Management and Dependencies

In Chapter 1, Section 1.9.1, we created a Maven project, but never really benefited from using Maven. Two things stand out:

1. Dependencies can be easily declared, and they are automatically downloaded by Maven, including all sub-dependencies. Maven’s particular strength lies in resolving transitive dependencies.
2. During the build, Java source code alone doesn’t make a project; the sources must be compiled, test cases must be run, and Javadoc should be generated. At the end, the final result is usually a compressed JAR file.

16.9.1 Dependency to Be Accepted

As an example, let's create a dependency on the small web framework *Spark* (<https://sparkjava.com>). Let's open the Project Object Model (POM) file *pom.xml* and add the code lines in bold for the dependency:

```
<project ...>
...
<properties>
  <maven.compiler.target>17</maven.compiler.target>
  <maven.compiler.source>17</maven.compiler.source>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>
<dependencies>
  <dependency>
    <groupId>com.sparkjava</groupId>
    <artifactId>spark-core</artifactId>
    <version>2.9.3</version>
  </dependency>
</dependencies>
</project>
```

Listing 16.5: pom.xml

All dependencies are located in a special XML element named `<dependencies>`. Below that element, you can then have any number of `<dependency>` blocks.

Now that everything is prepared, let's write the main program:

```
public class SparkServer {
  public static void main( String[] args ) {
    spark.Spark.get( "/hello", ( req, res ) -
> "Hello Browser " + req.userAgent() );
  }
}
```

Listing 16.5 src/main/java/SparkServer.java

When you start the program as usual, a web server starts as well, and you can read the output via the URL <http://localhost:4567/hello>. (You can ignore the logger outputs.)

16.9.2 Local and the Remote Repository

Resolving dependent JAR files takes longer the first time because Maven contacts a remote repository and always pulls the latest JAR files from that location and then stores these files locally. The extensive remote repository stores almost all versions of JAR files for many well-known open-source projects. The *Central Repository* can be found <https://repo.maven.apache.org/maven2/>.

The downloaded resources themselves aren't stored in the project but in a local repository located in the user's home directory and named *.m2*. In this way, all Maven projects share the same JAR files, and they don't have to be reobtained and updated on a project-by-project basis.

16.9.3 Lifecycles, Stages, and Maven Plugins

A Maven build consists of a three-stage lifecycle: *clean*, *default*, and *site*. Within this lifecycle are *stages*. For example, *default* contains the *compile* stage for translating the sources. Everything Maven runs are *plugins*, such as compilers, and many others that are listed at <https://maven.apache.org/plugins/>. A plugin can execute different *goals*. For example, the Javadoc plugin (described at <https://maven.apache.org/components/plugins/maven-javadoc-plugin/>) currently knows 16 goals. A goal can be accessed subsequently via the command line or via the integrated development environment (IDE).

For example, a JAR file is created via the package stage:

```
$ mvn package
```

The command-line tool must be called in the directory where the POM file is located.

16.10 Further Reading

The Java library provides a large number of classes and methods, but not always exactly what's required by the current project. Some problems, such as the structure and configuration of Java projects, object-relational mappers (www.hibernate.org), or command-line parsers, may require various commercial or open-source libraries and frameworks. With purchased products, licensing issues are obvious, but with open-source products, integration into one's own closed source project isn't always a given. Various types of licenses (<https://opensource.org/licenses>) for open-source software with always different specifications—whether the source code is changeable, whether derivatives must also be free, whether mixing with proprietary software possible—complicate the choice, and violations (<https://gpl-violations.org/>) are publicly denounced and unpleasant. Java developers should increasingly focus their attention on software under the Berkeley Source Distribution (BSD) license (the Apache license belongs in this group) and under the LGPL license for commercial distribution. The Apache group has assembled a nice collection of classes and methods named *Apache Commons* (<http://commons.apache.org>), and studying these sources are recommended for software developers. The website <https://www.openhub.net> is exceptionally well suited for this purpose and enables searching via specific keywords through more than 1 billion source code lines of various programming languages—amazing how many developers use profanities!

Contents

Preface	31
1 Introduction	43
1.1 Historical Background	43
1.2 On the Popularity of Java: The Key Features	45
1.2.1 Bytecode	46
1.2.2 Executing the Bytecode via a Virtual Machine	46
1.2.3 Platform Independence	46
1.2.4 Java as a Language, Runtime Environment, and Standard Library	47
1.2.5 Object Orientation in Java	47
1.2.6 Java Is Widespread and Well Known	48
1.2.7 Java Is Fast: Optimization and Just-In-Time Compilation	48
1.2.8 Pointers and References	50
1.2.9 Take Out the Trash, Garbage Collector!	51
1.2.10 Exception Handling	52
1.2.11 The Range of Libraries and Tools	52
1.2.12 Comparably Simple Syntax	53
1.2.13 Abandoning Controversial Concepts	53
1.2.14 Java Is Open Source	54
1.2.15 What Java Is Less Suitable for	55
1.3 Java versus Other Languages*	56
1.3.1 Java and C(++)	56
1.3.2 Java and JavaScript	57
1.3.3 A Word about Microsoft, Java, and J++	57
1.3.4 Java and C#/.NET	58
1.4 Further Development and Losses	59
1.4.1 The Development of Java and Its Future Prospects	59
1.4.2 Features, Enhancements, and Specification Requests	60
1.4.3 Applets	61
1.4.4 JavaFX	61
1.5 Java Platforms	62
1.5.1 Java Platform, Standard Edition	62
1.5.2 Java Platform, Micro Edition: Java for the Little Ones	65
1.5.3 Java for the Very, Very Little Ones	65
1.5.4 Java for the Big Ones: Jakarta EE (Formerly Java Platform, Enterprise Edition)	65
1.5.5 Real-Time Java	66

1.6	Java Platform, Standard Edition, Implementations	67
1.6.1	OpenJDK	67
1.6.2	Oracle JDK	68
1.7	Installing the Java Development Kit	69
1.7.1	Installing Oracle JDK on Windows	70
1.8	Compiling and Testing the First Program	71
1.8.1	A Square Numbers Program	72
1.8.2	The Compiler Run	73
1.8.3	The Runtime Environment	74
1.8.4	Common Compiler and Interpreter Issues	74
1.9	Development Environments	75
1.9.1	IntelliJ IDEA	76
1.9.2	Eclipse Integrated Development Environment	80
1.9.3	NetBeans	81
1.10	Further Reading	81
2	Imperative Language Concepts	83
2.1	Elements of the Java Programming Language	83
2.1.1	Tokens	84
2.1.2	Text Encoding by Unicode Characters	85
2.1.3	Identifiers	85
2.1.4	Literals	87
2.1.5	(Reserved) Keywords	87
2.1.6	Summary of the Lexical Analysis	88
2.1.7	Comments	89
2.2	From Classes to Statements	91
2.2.1	What Are Statements?	91
2.2.2	Class Declaration	92
2.2.3	The Journey Begins with main(String[])	93
2.2.4	The First Method Call: println(...)	93
2.2.5	Atomic Statements and Statement Sequences	95
2.2.6	More about print(...), println(...), and printf(...) for Screen Output	95
2.2.7	Application Programming Interface Documentation	97
2.2.8	Expressions	98
2.2.9	Expression Statements	98
2.2.10	First Insights into Object Orientation	99
2.2.11	Modifiers	100
2.2.12	Grouping Statements with Blocks	101

2.3	Data Types, Typing, Variables, and Assignments	102
2.3.1	Overview of Primitive Data Types	104
2.3.2	Variable Declarations	106
2.3.3	Automatic Type Detection with var	109
2.3.4	Final Variables and the final Modifier	110
2.3.5	Console Inputs	110
2.3.6	Truth Values	112
2.3.7	Integer Data Types	112
2.3.8	Underscores in Numbers	114
2.3.9	Alphanumeric Characters	115
2.3.10	The float and double Data Types	115
2.3.11	Good Names, Bad Names	117
2.3.12	No Automatic Initialization of Local Variables	118
2.4	Expressions, Operands, and Operators	119
2.4.1	Assignment Operator	119
2.4.2	Arithmetic Operators	121
2.4.3	Unary Minus and Plus	124
2.4.4	Prefix or Postfix Increment and Decrement	124
2.4.5	Assignment with Operation (Compound Assignment Operator)	126
2.4.6	Relational Operators and Equality Operators	128
2.4.7	Logical Operators: NOT, AND, OR, and XOR	129
2.4.8	Short-Circuit Operators	130
2.4.9	The Rank of Operators in Evaluation Order	132
2.4.10	Typecasting (Casting)	135
2.4.11	Overloaded Plus for Strings	140
2.4.12	Operators Missing*	141
2.5	Conditional Statements or Case Distinctions	142
2.5.1	Branching with the if Statement	142
2.5.2	Choosing the Alternative with an if-else Statement	145
2.5.3	The Condition Operator	148
2.5.4	The Switch Statement Provides an Alternative	151
2.5.5	Switch Expressions	157
2.6	Always the Same with Loops	160
2.6.1	The while Loop	161
2.6.2	The do-while Loop	163
2.6.3	The for Loop	164
2.6.4	Loop Conditions and Comparisons with ==*	168
2.6.5	Loop Termination with break and back to Test with continue	171
2.6.6	break and continue with Labels*	174
2.7	Methods of a Class	177
2.7.1	Components of a Method	177

2.7.2	Signature Description in the Java Application Programming Interface Documentation	179
2.7.3	Calling a Method	180
2.7.4	Declaring Methods without Parameters	181
2.7.5	Static Methods (Class Methods)	182
2.7.6	Parameters, Arguments, and Value Transfers	183
2.7.7	Ending Methods Prematurely with return	185
2.7.8	Unreachable Source Code for Methods*	185
2.7.9	Methods with Returns	186
2.7.10	Overloading Methods	191
2.7.11	Scope	193
2.7.12	Default Values for Unlisted Arguments*	195
2.7.13	Recursive Methods*	195
2.7.14	Towers of Hanoi*	198
2.8	Further Reading	200

3 **Classes and Objects** 201

3.1	Object-Oriented Programming	201
3.1.1	Why Object-Oriented Programming at All?	202
3.1.2	When I Think of Java, I Think of Reusability	202
3.2	Members of a Class	203
3.3	Natural Modeling Using Unified Modeling Language*	204
3.4	Creating New Objects	206
3.4.1	Creating an Instance of a Class Using the new Keyword	206
3.4.2	Declaring Reference Variables	207
3.4.3	Let's Get to the Point: Accessing Object Variables and Methods	208
3.4.4	The Connection between new, the Heap, and the Garbage Collector	212
3.4.5	Overview of Point Methods	213
3.4.6	Using Constructors	217
3.5	ZZZZZnake	218
3.6	Tying Packages, Imports, and Compilation Units	220
3.6.1	Java Packages	220
3.6.2	Packages in the Standard Library	221
3.6.3	Full Qualification and Import Declaration	221
3.6.4	Reaching All Types of a Package with Type-Import-on-Demand	223
3.6.5	Hierarchical Structures across Packages and Mirroring in the File System	224
3.6.6	The Package Declaration	224

3.6.7	Unnamed Package (Default Package)	225
3.6.8	Compilation Unit	226
3.6.9	Static Import*	226
3.7	Using References, Diversity, Identity, and Equality	228
3.7.1	null References and the Question of Philosophy	228
3.7.2	Everything to null? Testing References	230
3.7.3	Assignments with References	231
3.7.4	Methods with Reference Types as Parameters	232
3.7.5	Identity of Objects	236
3.7.6	Equivalence and the equals(...) Method	237
3.8	Further Reading	239

4 **Arrays and Their Areas of Use** 241

4.1	Simple Field Work	241
4.1.1	Basic Components	242
4.1.2	Declaring Array Variables	243
4.1.3	Creating Array Objects with new	244
4.1.4	Arrays with { contents }	245
4.1.5	Reading the Length of an Array via the Object Variable Length	246
4.1.6	Accessing the Elements via the Index	246
4.1.7	Typical Array Errors	248
4.1.8	Passing Arrays to Methods	250
4.1.9	Multiple Return Values*	250
4.1.10	Preinitialized Arrays	251
4.2	The Extended for Loop	252
4.2.1	Using Anonymous Arrays in the Extended for Loop	253
4.2.2	Example: Searching Arrays with Strings	254
4.2.3	Creating Random Player Positions	255
4.3	A Method with a Variable Number of Arguments	256
4.3.1	System.out.printf(...) Accepts Any Number of Arguments	257
4.3.2	Finding the Average of Variable Arguments	257
4.3.3	Vararg Design Tips*	259
4.4	Multidimensional Arrays*	259
4.4.1	Nonrectangular Arrays*	262
4.5	Library Support for Arrays	264
4.5.1	Cloning Can Be Worthwhile: Propagating Arrays	264
4.5.2	Why Can Arrays "Do" So Little?	265
4.5.3	Copying Array Contents	266

4.6	Using the Arrays Class for Comparing, Filling, Searching, and Sorting	267
4.6.1	String Representation of an Array	267
4.6.2	Sorting	268
4.6.3	Parallel Sorting	269
4.6.4	Comparing Arrays of Primitives with Arrays.equals(...) and Arrays.deepEquals(...)*	269
4.6.5	Comparing Object Arrays Using Arrays.equals(...) and Arrays.deepEquals(...)*	271
4.6.6	Searching Differences Using Mismatch (...)*	272
4.6.7	Filling Arrays*	272
4.6.8	Copying Array Sections*	273
4.6.9	Binary Search*	275
4.6.10	Lexicographic Array Comparisons Using compare(...) and compareUnsigned(...)	276
4.6.11	Arrays for Lists with Arrays.asList(...): Convenient for Searching and Comparing*	277
4.6.12	A Long Snake	278
4.7	The Entry Point for the Runtime System: main(...)	281
4.7.1	Correct Declaration of the Start Method	281
4.7.2	Processing Command Line Arguments	282
4.7.3	The Return Type of main(...) and System.exit(int)*	283
4.8	Further Reading	285
5	Handling Characters and Strings	287
5.1	From ASCII via ISO-8859-1 to Unicode	287
5.1.1	ASCII	287
5.1.2	ISO/IEC 8859-1	288
5.1.3	Unicode	289
5.1.4	Unicode Character Encoding	291
5.1.5	Escape Sequences	292
5.1.6	Notation for Unicode Characters and Unicode Escapes	292
5.1.7	Java Versions Go Hand in Hand with the Unicode Standard*	294
5.2	Data Types for Characters and Strings	295
5.3	The Character Class	296
5.3.1	Is That So?	296
5.3.2	Converting Characters to Uppercase/Lowercase	298
5.3.3	From Character to String	299
5.3.4	From char to int: From Character to Number*	299

5.4	Strings	301
5.5	The String Class and Its Methods	303
5.5.1	String Literals as String Objects for Constant Strings	303
5.5.2	Concatenation with +	303
5.5.3	Multiline Text Blocks with “"""	304
5.5.4	String Length and Testing for Empty Strings	309
5.5.5	Accessing a Specific Character with charAt(int)	310
5.5.6	Searching for Contained Characters and Strings	311
5.5.7	The Hangman Game	314
5.5.8	Good That We Have Compared	316
5.5.9	Extracting String Sections	320
5.5.10	Appending Strings, Merging Strings, Case Sensitivity, and Whitespace	325
5.5.11	Searched, Found, and Replaced	328
5.5.12	Creating String Objects with Constructors and from Repeats*	330
5.6	Mutable Strings with StringBuilder and StringBuffer	333
5.6.1	Creating StringBuilder Objects	334
5.6.2	Converting StringBuilder to Other String Formats	335
5.6.3	Requesting Characters or Strings	335
5.6.4	Appending Data	335
5.6.5	Setting, Deleting, and Reversing Characters and Strings	337
5.6.6	Length and Capacity of a StringBuilder Object*	339
5.6.7	Comparison of StringBuilder Instances and Strings with StringBuilder	340
5.6.8	hashCode() with StringBuilder*	342
5.7	CharSequence as Base Type	342
5.7.1	Basic Operations of the Interface	343
5.7.2	Static compare(...) Method in CharSequence	344
5.7.3	Default Methods in the CharSequence Interface*	345
5.8	Converting Primitives and Strings	345
5.8.1	Converting Different Types to String Representations	345
5.8.2	Converting String Contents to a Primitive Value	347
5.8.3	String Representation in Binary, Hex, and Octal Formats*	349
5.8.4	parse*(...) and print*() Methods in DatatypeConverter*	353
5.9	Concatenating Strings	353
5.9.1	Concatenating Strings with StringJoiner	353
5.10	Decomposing Strings	355
5.10.1	Splitting Strings via split(...)	356
5.10.2	Yes We Can, Yes We Scan: The Scanner Class	356

5.11	Formatting Outputs	360
5.11.1	Formatting and Outputting via format()	361
5.12	Further Reading	367
6	Writing Custom Classes	369
6.1	Declaring Custom Classes with Members	369
6.1.1	Minimum Class	370
6.1.2	Declaring Object Variables	370
6.1.3	Declaring Methods	373
6.1.4	Shadowed Variables	375
6.1.5	The this Reference	377
6.2	Privacy and Visibility	380
6.2.1	For the Public: public	381
6.2.2	Not Public: Passwords Are private	381
6.2.3	Why Not Free Methods and Variables for All?	383
6.2.4	private Is Not Quite Private: It Depends on Who Sees It*	383
6.2.5	Declaring Access Methods for Object Variables	384
6.2.6	Setters and Getters according to the JavaBeans Specification	384
6.2.7	Package-Visibility	386
6.2.8	Visibility Summary	388
6.3	One for All: Static Methods and Class Variables	390
6.3.1	Why Static Members Are Useful	391
6.3.2	Static Members with static	392
6.3.3	Using Static Members via References?*	393
6.3.4	Why Case Sensitivity Is Important*	394
6.3.5	Static Variables for Data Exchange*	395
6.3.6	Static Members and Object Members*	396
6.4	Constants and Enumerations	397
6.4.1	Constants via Static Final Variables	397
6.4.2	Type-Unsafe Enumerations	398
6.4.3	Enumeration Types: Type-Safe Enumerations with enum	400
6.5	Creating and Destroying Objects	405
6.5.1	Writing Constructors	405
6.5.2	Relationship of Method and Constructor	406
6.5.3	The Default Constructor	407
6.5.4	Parameterized and Overloaded Constructors	408
6.5.5	Copy Constructors	410
6.5.6	Calling Another Constructor of the Same Class via this(...)	412

6.5.7	Immutable Objects and With Methods	414
6.5.8	We Don't Miss You: The Garbage Collector	416
6.6	Class and Object Initialization*	418
6.6.1	Initializing Object Variables	418
6.6.2	Static Blocks as Class Initializers	420
6.6.3	Initializing Class Variables	421
6.6.4	Compiled Assignments of the Class Variables	421
6.6.5	Instance Initializer	422
6.6.6	Setting Final Values in the Constructor and Static Blocks	425
6.7	Conclusion	426
7	Object-Oriented Relationship	427
7.1	Associations between Objects	427
7.1.1	Association Types	427
7.1.2	Unidirectional 1-to-1 Relationship	428
7.1.3	Becoming Friends: Bidirectional 1-to-1 Relationships	429
7.1.4	Unidirectional 1-to-n Relationships	431
7.2	Inheritance	436
7.2.1	Inheritance in Java	437
7.2.2	Modeling Events	438
7.2.3	The Implicit Base Class java.lang.Object	440
7.2.4	Single and Multiple Inheritance*	440
7.2.5	Do Children See Everything? The Protected Visibility	441
7.2.6	Constructors in Inheritance and super(...)	442
7.3	Types in Hierarchies	447
7.3.1	Automatic and Explicit Typecasting	447
7.3.2	The Substitution Principle	450
7.3.3	Testing Types with the instanceof Operator	452
7.3.4	Pattern Matching for instanceof	455
7.4	Overriding Methods	457
7.4.1	Providing Methods in Subclasses with a New Behavior	457
7.4.2	With super to the Parents	461
7.5	Testing Dynamic Bindings	463
7.5.1	Bound to toString()	463
7.5.2	Implementing System.out.println(Object)	465
7.6	Final Classes and Final Methods	466
7.6.1	Final Classes	466
7.6.2	Non-Overridable (final) Methods	466

7.7	Abstract Classes and Abstract Methods	468
7.7.1	Abstract Classes	468
7.7.2	Abstract Methods	470
7.8	Further Information on Overriding and Dynamic Binding	476
7.8.1	No Dynamic Binding for Private, Static, and final Methods	476
7.8.2	Covariant Return Types	476
7.8.3	Array Types and Covariance*	477
7.8.4	Dynamic Binding even with Constructor Calls*	478
7.8.5	No Dynamic Binding for Covered Object Variables*	480
7.9	A Programming Task	482
8	Interfaces, Enumerations, Sealed Classes, Records	483
8.1	Interfaces	483
8.1.1	Interfaces Are New Types	483
8.1.2	Declaring Interfaces	484
8.1.3	Abstract Methods in Interfaces	484
8.1.4	Implementing Interfaces	485
8.1.5	A Polymorphism Example with Interfaces	487
8.1.6	Multiple Inheritance with Interfaces	488
8.1.7	No Risk of Collision with Multiple Inheritance*	491
8.1.8	Extending Interfaces: Subinterfaces	493
8.1.9	Constant Declarations for Interfaces	494
8.1.10	Subsequent Implementation of Interfaces*	494
8.1.11	Static Programmed Methods in Interfaces	495
8.1.12	Extending and Modifying Interfaces	497
8.1.13	Default Methods	498
8.1.14	Declaring and Using Extended Interfaces	500
8.1.15	Public and Private Interface Methods	503
8.1.16	Extended Interfaces, Multiple Inheritance, and Ambiguities*	504
8.1.17	Creating Building Blocks with Default Methods*	508
8.1.18	Marker Interfaces*	512
8.1.19	(Abstract) Classes and Interfaces in Comparison	513
8.2	Enumeration Types	513
8.2.1	Methods on Enum Objects	514
8.2.2	Enumerations with Custom Methods, Constructors, and Initializers*	518
8.3	Sealed Classes and Interfaces	523
8.3.1	Sealed Classes and Interfaces	525
8.3.2	Subclasses Are Final, Sealed, and Non-Sealed	527
8.3.3	Abbreviated Notations	528

8.4	Records	529
8.4.1	Simple Records	529
8.4.2	Records with Methods	531
8.4.3	Customizing Record Constructors	532
8.4.4	Adding Constructors	534
8.4.5	Sealed Interfaces and Records	535
8.4.6	Records: Summary	536
9	There Must Be Exceptions	539
9.1	Fencing In Problem Areas	539
9.1.1	Exceptions in Java with try and catch	540
9.1.2	Checked and Unchecked Exceptions	540
9.1.3	A NumberFormatException (Unchecked Exception)	541
9.1.4	Appending a Date/Timestamp to a Text File (Checked Exception)	543
9.1.5	Repeating Canceled Sections*	545
9.1.6	Empty catch Blocks	546
9.1.7	Catching Multiple Exceptions	547
9.1.8	Combining Identical catch Blocks with Multi-Catch	548
9.2	Redirecting Exceptions and throws at the Head of Methods/Constructors	549
9.2.1	throws in Constructors and Methods	549
9.3	The Class Hierarchy of Exceptions	550
9.3.1	Members of the Exception Object	550
9.3.2	Base Type Throwable	551
9.3.3	The Exception Hierarchy	552
9.3.4	Catching Super-Exceptions	552
9.3.5	Already Caught?	554
9.3.6	Procedure of an Exceptional Situation	555
9.3.7	No General Catching!	555
9.3.8	Known RuntimeException Classes	557
9.3.9	Interception Is Possible, but Not Mandatory	558
9.4	Final Handling Using finally	558
9.4.1	The Ignorant Version	559
9.4.2	The Well-Intentioned Attempt	560
9.4.3	From Now On, Closing Is Part of the Agenda	560
9.4.4	Summary	562
9.4.5	A try without a catch, but a try-finally	562
9.5	Triggering Custom Exceptions	564
9.5.1	Triggering Exceptions via throw	564
9.5.2	Knowing and Using Existing Runtime Exception Types	566

9.5.3	Testing Parameters and Good Error Messages	568
9.5.4	Declaring New Exception Classes	570
9.5.5	Custom Exceptions as Subclasses of Exception or RuntimeException?	571
9.5.6	Catching and Redirecting Exceptions*	574
9.5.7	Changing the Call Stack of Exceptions*	575
9.5.8	Nested Exceptions*	576
9.6	try with Resources (Automatic Resource Management)	579
9.6.1	try with Resources	580
9.6.2	The AutoCloseable Interface	581
9.6.3	Exceptions to close()	582
9.6.4	Types That Are AutoCloseable and Closeable	583
9.6.5	Using Multiple Resources	583
9.6.6	Suppressed Exceptions*	585
9.7	Special Features of Exception Handling*	588
9.7.1	Return Values for Thrown Exceptions	588
9.7.2	Exceptions and Returns Disappear: The Duo return and finally	589
9.7.3	throws on Overridden Methods	590
9.7.4	Unreachable catch Clauses	592
9.8	Hard Errors: Error*	593
9.9	Assertions*	594
9.9.1	Using Assertions in Custom Programs	595
9.9.2	Enabling Assertions and Runtime Errors	595
9.9.3	Enabling or Disabling Assertions More Detailed	597
9.10	Conclusion	597
10 Nested Types		599
10.1	Nested Classes, Interfaces, and Enumerations	599
10.2	Static Nested Types	601
10.2.1	Modifiers and Visibility	602
10.2.2	Records as Containers	602
10.2.3	Implementing Static Nested Types*	602
10.3	Non-Static Nested Types	603
10.3.1	Creating Instances of Inner Classes	603
10.3.2	The this Reference	604
10.3.3	Class Files Generated by the Compiler*	605
10.4	Local Classes	605
10.4.1	Example with a Custom Declaration	606

10.4.2	Using a Local Class for a Timer	606
10.5	Anonymous Inner Classes	607
10.5.1	Using an Anonymous Inner Class for the Timer	608
10.5.2	Implementing Anonymous Inner Classes*	609
10.5.3	Constructors of Anonymous Inner Classes	609
10.5.4	Accessing Local Variables from Local and Anonymous Classes*	611
10.5.5	Nested Classes Access Private Members	612
10.6	Nests	613
10.7	Conclusion	614
11 Special Types of Java SE		615
11.1	Object Is the Mother of All Classes	616
11.1.1	Class Objects	616
11.1.2	Object Identification with toString()	617
11.1.3	Object Equivalence with equals(...) and Identity	619
11.1.4	Cloning an Object Using clone()*	625
11.1.5	Returning Hash Values via hashCode()*	630
11.1.6	System.identityHashCode(...) and the Problem of Non-Unique Object References*	636
11.1.7	Synchronization*	637
11.2	Weak References and Cleaners	638
11.3	The java.util.Objects Utility Class	639
11.3.1	Built-In Null Tests for equals(...)/hashCode()	639
11.3.2	Objects.toString(...)	640
11.3.3	null Checks with Built-In Exception Handling	640
11.3.4	Tests for null	641
11.3.5	Checking Index-Related Program Arguments for Correctness	642
11.4	Comparing Objects and Establishing Order	643
11.4.1	Naturally Ordered or Not?	643
11.4.2	compare*() Method of the Comparable and Comparator Interfaces	644
11.4.3	Return Values Encode the Order	645
11.4.4	Sorting Candy by Calories Using a Sample Comparator	645
11.4.5	Tips for Comparator and Comparable Implementations	647
11.4.6	Static and Default Methods in Comparator	648
11.5	Wrapper Classes and Autoboxing	651
11.5.1	Creating Wrapper Objects	653
11.5.2	Conversions to a String Representation	654

11.5.3	Parsing from a String Representation	655
11.5.4	The Number Base Class for Numeric Wrapper Objects	655
11.5.5	Performing Comparisons with compare*(...), compareTo(...), equals(...), and Hash Values	657
11.5.6	Static Reduction Methods in Wrapper Classes	660
11.5.7	Constants for the Size of a Primitive Type*	661
11.5.8	Handling Unsigned Numbers*	661
11.5.9	The Integer and Long Classes	663
11.5.10	The Double and Float Classes for Floats	664
11.5.11	The Boolean Class	664
11.5.12	Autoboxing: Boxing and Unboxing	665
11.6	Iterator, Iterable*	670
11.6.1	The Iterator Interface	670
11.6.2	The Supplier of the Iterator	673
11.6.3	The Iterable Interface	674
11.6.4	Extended for and Iterable	674
11.6.5	Internal Iteration	675
11.6.6	Implementing a Custom Iterable*	676
11.7	Annotations in Java Platform, Standard Edition	677
11.7.1	Places for Annotations	677
11.7.2	Annotation Types from java.lang	678
11.7.3	@Deprecated	678
11.7.4	Annotations with Additional Information	679
11.7.5	@SuppressWarnings	679
11.8	Further Reading	682
12	Generics<T>	683
12.1	Introduction to Java Generics	683
12.1.1	Man versus Machine: Type Checking of the Compiler and the Runtime Environment	683
12.1.2	Rockets	684
12.1.3	Declaring Generic Types	686
12.1.4	Using Generics	688
12.1.5	Diamonds Are Forever	690
12.1.6	Generic Interfaces	693
12.1.7	Generic Methods/Constructors and Type Inference	695
12.2	Implementing Generics, Type Erasure, and Raw Types	699
12.2.1	Implementation Options	699
12.2.2	Type Erasure	699

12.2.3	Problems with Type Erasure	700
12.2.4	Raw Types	704
12.3	Restricting Types via Bounds	706
12.3.1	Simple Restrictions with extends	707
12.3.2	Other Supertypes with &	709
12.4	Type Parameters in the throws Clause*	710
12.4.1	Declaring a Class with Type Variable <E extends Exception>	710
12.4.2	Parameterized Type for Type Variable <E extends Exception>	710
12.5	Inheritance and Invariance with Generics	713
12.5.1	Arrays Are Covariant	713
12.5.2	Generics Aren't Covariant, but Invariant	714
12.5.3	Wildcards with ?	715
12.5.4	Bounded Wildcards	717
12.5.5	Bounded Wildcard Types and Bounded Type Variables	720
12.5.6	The PECS Principle	722
12.6	Consequences of Type Erasure: Type Tokens, Arrays*	725
12.6.1	Type Tokens	725
12.6.2	Supertype Tokens	727
12.6.3	Generics and Arrays	728
12.7	Further Reading	729
13	Lambda Expressions and Functional Programming	731
13.1	Functional Interfaces and Lambda Expressions	731
13.1.1	Classes Implement Interfaces	731
13.1.2	Lambda Expressions Implement Interfaces	733
13.1.3	Functional Interfaces	734
13.1.4	The Type of a Lambda Expression Depends on the Target Type	735
13.1.5	@FunctionalInterface Annotations	740
13.1.6	Syntax for Lambda Expressions	741
13.1.7	The Environment of Lambda Expressions and Variable Accesses	745
13.1.8	Exceptions in Lambda Expressions	751
13.1.9	Classes with an Abstract Method as a Functional Interface?*	755
13.2	Method References	755
13.2.1	Motivation	755
13.2.2	Method References with ::	756
13.2.3	Variations of Method References	756

13.3	Constructor References	759
13.3.1	Writing Constructor References	760
13.3.2	Parameterless and Parameterized Constructors	761
13.3.3	Useful Predefined Interfaces for Constructor References	761
13.4	Functional Programming	762
13.4.1	Code = Data	762
13.4.2	Programming Paradigms: Imperative or Declarative	763
13.4.3	Principles of Functional Programming	764
13.4.4	Imperative Programming and Functional Programming	767
13.4.5	Comparator as an Example of Higher-Order Functions	769
13.4.6	Viewing Lambda Expressions as Mappings or Functions	769
13.5	Functional Interfaces from the java.util.function Package	770
13.5.1	Blocks with Code and the Functional Interface Consumer	771
13.5.2	Supplier	773
13.5.3	Predicates and java.util.function.Predicate	773
13.5.4	Functions via the Functional Interface java.util.function.Function	775
13.5.5	I Take Two	779
13.5.6	Functional Interfaces with Primitives	782
13.6	Optional Is Not a Non-Starter	784
13.6.1	Using null	785
13.6.2	The Optional Type	787
13.6.3	Starting Functional Interfaces with Optional	789
13.6.4	Primitive-Optional with Special Optional* Classes	792
13.7	What Is So Functional Now?	795
13.7.1	Recyclability	795
13.7.2	Stateless, Immutable	795
13.8	Further Reading	797
14	Architecture, Design, and Applied Object Orientation	799
14.1	SOLID Modeling	799
14.1.1	Three Rules	800
14.1.2	SOLID	800
14.1.3	Don't Be STUPID	802
14.2	Architecture, Design, and Implementation	803
14.3	Design Patterns	803
14.3.1	Motivation for Design Patterns	804
14.3.2	Singleton	805

14.3.3	Factory Methods	806
14.3.4	Implementing the Observer Pattern with Listeners	807
14.4	Further Reading	811
15	Java Platform Module System	813
15.1	Class Loader and Module/Classpath	813
15.1.1	Loading Classes per Request	813
15.1.2	Watching the Class Loader at Work	814
15.1.3	JMOD Files and JAR Files	815
15.1.4	Where the Classes Come from: Search Locations and Special Class Loaders	816
15.1.5	Setting the Search Path	817
15.2	Importing Modules	819
15.2.1	Who Sees Whom?	819
15.2.2	Platform Modules and a JMOD Example	821
15.2.3	Using Internal Platform Features: --add-exports	821
15.2.4	Integrating New Modules	824
15.3	Developing Custom Modules	825
15.3.1	Module com.tutego.candytester	825
15.3.2	Module Declaration with module-info.java and Exports	826
15.3.3	Module com.tutego.main	826
15.3.4	Module info File with requires	827
15.3.5	Writing Module Inserters: Java Virtual Machine Switches -p and -m	828
15.3.6	Experiments with the Module Info File	829
15.3.7	Automatic Modules	829
15.3.8	Unnamed Modules	830
15.3.9	Readability and Accessibility	831
15.3.10	Module Migration	832
15.4	Further Reading	833
16	The Class Library	835
16.1	The Java Class Philosophy	835
16.1.1	Modules, Packages, and Types	836
16.1.2	Overview of the Packages of the Standard Library	838

16.2 Simple Time Measurement and Profiling*	842
16.2.1 Profilers	843
16.3 The Class Class	843
16.3.1 Obtaining a Class Object	843
16.3.2 A Class Is a Type	846
16.4 The Utility Classes System and Members	846
16.4.1 Memory of the Java Virtual Machine	847
16.4.2 Number of CPUs or Cores	848
16.4.3 System Properties of the Java Environment	848
16.4.4 Setting Custom Properties from the Console*	850
16.4.5 Newline Characters and line.separator	851
16.4.6 Environment Variables of the Operating System	852
16.5 The Languages of Different Countries	853
16.5.1 Regional Languages via Locale Objects	853
16.6 Overview of Important Date Classes	857
16.6.1 Unix Time: January 1, 1970	858
16.6.2 System.currentTimeMillis()	858
16.6.3 Simple Time Conversions via TimeUnit	859
16.7 Date-Time API	860
16.7.1 Initial Overview	860
16.7.2 Human Time and Machine Time	861
16.7.3 The LocalDate Date Class	863
16.8 Logging with Java	864
16.8.1 Logging Application Programming Interfaces	865
16.8.2 Logging with java.util.logging	865
16.9 Maven: Resolving Build Management and Dependencies	867
16.9.1 Dependency to Be Accepted	868
16.9.2 Local and the Remote Repository	868
16.9.3 Lifecycles, Stages, and Maven Plugins	869
16.10 Further Reading	869
17 Introduction to Concurrent Programming	871
17.1 Concurrency and Parallelism	871
17.1.1 Multitasking, Processes, and Threads	872
17.1.2 Threads and Processes	873
17.1.3 How Concurrent Programs Can Increase Speed	874
17.1.4 How Java Can Provide for Concurrency	875

17.2 Generating Existing Threads and New Threads	875
17.2.1 Main Thread	876
17.2.2 Who Am I?	876
17.2.3 Implementing the Runnable Interface	876
17.2.4 Starting Thread with Runnable	878
17.2.5 Parameterizing Runnable	879
17.2.6 Extending the Thread Class*	880
17.3 Thread Members and States	882
17.3.1 The Name of a Thread	882
17.3.2 The States of a Thread*	882
17.3.3 Sleepers Wanted	883
17.3.4 When Threads Are Finished	885
17.3.5 Terminating a Thread Politely Using Interrupts	885
17.3.6 Unhandled Exceptions, Thread End, and UncaughtExceptionHandler	887
17.3.7 The stop() from the Outside and the Rescue with ThreadDeath*	889
17.3.8 Stopping and Resuming the Work*	891
17.3.9 Priority*	892
17.4 Enter the Executor	893
17.4.1 The Executor Interface	893
17.4.2 Happy as a Group: The Thread Pools	895
17.4.3 Threads with return via Callable	897
17.4.4 Memories of the Future: The Future Return	899
17.4.5 Processing Multiple Callable Objects	902
17.4.6 CompletionService and ExecutorCompletionService	903
17.4.7 ScheduledExecutorService: Repetitive Tasks and Time Controls	904
17.4.8 Asynchronous Programming with CompletableFuture (CompletionStage)	905
17.5 Further Reading	907
18 Introduction to Data Structures and Algorithms	909
18.1 Lists	909
18.1.1 First List Example	910
18.1.2 Selection Criterion ArrayList or LinkedList	911
18.1.3 The List Interface	911
18.1.4 ArrayList	918
18.1.5 LinkedList	919
18.1.6 The Array Adapter Arrays.asList(...)	921
18.1.7 ListIterator*	923

18.1.8	Understanding toArray(...) of Collection: Recognizing Traps	924
18.1.9	Managing Primitive Elements in Data Structures	927
18.2	Sets	928
18.2.1	A First Example of a Set	928
18.2.2	Methods of the Set Interface	930
18.2.3	HashSet	932
18.2.4	TreeSet: The Sorted Set	933
18.2.5	The Interfaces NavigableSet and SortedSet	935
18.2.6	LinkedHashSet	937
18.3	Associative Memory	938
18.3.1	The HashMap and TreeMap Classes and Static Map Methods	938
18.3.2	Inserting and Querying the Associative Memory	942
18.4	The Stream API	944
18.4.1	Declarative Programming	944
18.4.2	Internal versus External Iteration	945
18.4.3	What Is a Stream?	946
18.5	Creating a Stream	947
18.5.1	Stream.of(...)	948
18.5.2	Stream.generate(...)	949
18.5.3	Stream.iterate(...)	949
18.5.4	Parallel or Sequential Streams	950
18.6	Terminal Operations	951
18.6.1	Number of Elements	951
18.6.2	And Now All: forEach(...)	951
18.6.3	Getting Individual Elements from the Stream	952
18.6.4	Existence Tests with Predicates	953
18.6.5	Reducing a Stream to Its Smallest or Largest Element	953
18.6.6	Reducing a Stream with Its Own Functions	954
18.6.7	Writing Results to a Container, Part 1: collect(...)	956
18.6.8	Writing Results to a Container, Part 2: Collector and Collectors	957
18.6.9	Writing Results to a Container, Part 3: Groupings	959
18.6.10	Transferring Stream Elements to an Array or an Iterator	961
18.7	Intermediary Operations	962
18.7.1	Element Previews	963
18.7.2	Filtering Elements	963
18.7.3	Stateful Intermediary Operations	963
18.7.4	Prefix Operations	965
18.7.5	Images	966
18.8	Further Reading	968

19	Files and Data Streams	969
19.1	Old and New Worlds in java.io and java.nio	969
19.1.1	java.io Package with the File Class	969
19.1.2	NIO.2 and the java.nio Package	970
19.2	File Systems and Paths	970
19.2.1	FileSystem and Path	971
19.2.2	The Files Utility Class	977
19.3	Random Access Files	980
19.3.1	Opening a RandomAccessFile for Reading and Writing	980
19.3.2	Reading from RandomAccessFile	981
19.3.3	Writing with RandomAccessFile	983
19.3.4	The Length of the RandomAccessFile	983
19.3.5	Back and Forth within the File	984
19.4	Base Classes for Input/Output	985
19.4.1	The Four Abstract Base Classes	985
19.4.2	The Abstract Base Class OutputStream	986
19.4.3	The Abstract Base Class InputStream	988
19.4.4	The Abstract Base Class Writer	990
19.4.5	The Appendable Interface*	991
19.4.6	The Abstract Base Class Reader	992
19.4.7	The Interfaces Closeable, AutoCloseable, and Flushable	995
19.5	Reading from Files and Writing to Files	996
19.5.1	Obtaining Byte-Oriented Data Streams via Files	997
19.5.2	Obtaining Character-Oriented Data Streams via Files	997
19.5.3	The Function of OpenOption in the Files.new*(...) Methods	999
19.5.4	Loading Resources from the Module Path and from JAR Files	1001
19.6	Further Reading	1003
20	Introduction to Database Management with JDBC	1005
20.1	Relational Databases and Java Access	1005
20.1.1	The Relational Model	1005
20.1.2	Java Application Programming Interfaces for Accessing Relational Databases	1006
20.1.3	The JDBC API and Implementations: The JDBC Driver	1006
20.1.4	H2 Is the Tool in Java	1007

20.2 A Sample Query	1008
20.2.1 Steps to Query the Database	1008
20.2.2 Accessing the Relational Database with Java	1008
20.3 Further Reading	1009
21 Bits and Bytes, Mathematics and Money	1011
21.1 Bits and Bytes	1011
21.1.1 The Bit Operators: Complement, AND, OR, and XOR	1012
21.1.2 Representation of Integers in Java: Two's Complement	1013
21.1.3 The Binary, Octal, and Hexadecimal Place Value Systems	1014
21.1.4 Effect of Typecasting on Bit Patterns	1016
21.1.5 Working without Signs	1018
21.1.6 The Shift Operators	1021
21.1.7 Setting, Clearing, Reversing, and Testing a Bit	1023
21.1.8 Bit Methods of the Integer and Long Classes	1024
21.2 Floating Point Arithmetic in Java	1025
21.2.1 Special Values for Infinity, Zero, and Not a Number	1026
21.2.2 Standard Notation and Scientific Notation for Floats*	1029
21.2.3 Mantissas and Exponents*	1029
21.3 The Members of the Math Class	1031
21.3.1 Object Variables of the Math Class	1031
21.3.2 Absolute Values and Signs	1032
21.3.3 Maximums/Minimums	1032
21.3.4 Rounding Values	1033
21.3.5 Remainder of an Integer Division*	1035
21.3.6 Division with Rounding toward Negative Infinity and Alternative Remainders*	1036
21.3.7 Multiply-Accumulate	1038
21.3.8 Square Root and Exponential Methods	1038
21.3.9 The Logarithm*	1039
21.3.10 Angle Methods*	1040
21.3.11 Random Numbers	1041
21.4 Accuracy and the Value Range of Type and Overflow Control*	1042
21.4.1 The Largest and Smallest Values	1042
21.4.2 Overflow and Everything Entirely Exact	1042
21.4.3 What in the World Does the ulp Method Do?	1045
21.5 Random Numbers: Random, ThreadLocalRandom, and SecureRandom	1046
21.5.1 The Random Class	1047

21.5.2 ThreadLocalRandom	1050
21.5.3 The SecureRandom Class*	1050
21.6 Large Numbers*	1051
21.6.1 The BigInteger Class	1051
21.6.2 Example: Quite Long Factorials with BigInteger	1058
21.6.3 Large Floats with BigDecimal	1059
21.6.4 Conveniently Setting the Calculation Accuracy via MathContext	1062
21.6.5 Calculating even Faster with Mutable Implementations	1063
21.7 Money and Currency	1064
21.7.1 Representing Amounts of Money	1064
21.7.2 ISO 4217	1064
21.7.3 Representing Currencies in Java	1065
21.8 Further Reading	1066
22 Testing with JUnit	1067
22.1 Software Tests	1067
22.1.1 Procedure for Writing Test Cases	1068
22.2 The JUnit Testing Framework	1068
22.2.1 JUnit Versions	1069
22.2.2 Integrating JUnit	1069
22.2.3 Test-Driven Development and the Test-First Approach	1069
22.2.4 Test, Implement, Test, Implement, Test, Rejoice	1070
22.2.5 Running JUnit Tests	1072
22.2.6 assert*(...) Methods of the Assertions Class	1073
22.2.7 Testing Exceptions	1076
22.2.8 Setting Limits for Execution Times	1077
22.2.9 Labels with @DisplayName	1078
22.2.10 Nested Tests	1078
22.2.11 Ignoring Tests	1078
22.2.12 Canceling Tests with Methods of the Assumptions Class	1079
22.2.13 Parameterized Tests	1079
22.3 Java Assertion Libraries and AssertJ	1081
22.3.1 AssertJ	1081
22.4 Structure of Large Test Cases	1083
22.4.1 Fixtures	1083
22.4.2 Collections of Test Classes and Class Organization	1084
22.5 Good Design Enables Effective Testing	1085

22.6	Dummy, Fake, Stub, and Mock	1087
22.7	JUnit Extensions and Testing Add-Ons	1089
22.7.1	Web Tests	1089
22.7.2	Testing the Database Interface	1089
22.8	Further Reading	1089
 23 The Tools of the JDK		1091
23.1	Overview	1091
23.1.1	Structure and Common Switches	1092
23.2	Translating Java Sources	1092
23.2.1	The Java Compiler of the Java Development Kit	1092
23.2.2	Native Compilers	1092
23.3	The Java Runtime Environment	1093
23.3.1	Switches of the Java Virtual Machine	1093
23.3.2	The Difference between java.exe and javaw.exe	1095
23.4	Documentation Comments with Javadoc	1096
23.4.1	Setting a Documentation Comment	1096
23.4.2	Creating Documentation with the javadoc Tool	1098
23.4.3	HTML Tags in Documentation Comments*	1099
23.4.4	Generated Files	1099
23.4.5	Documentation Comments at a Glance*	1100
23.4.6	Javadoc and Doclets*	1101
23.4.7	Deprecated Types and Members	1101
23.4.8	Javadoc Verification with DocLint	1104
23.5	The JAR Archive Format	1105
23.5.1	Using the jar Utility	1105
23.5.2	The Manifest	1105
23.5.3	Launching Applications in Java Archives: Executable JAR Files	1106
23.6	Further Reading	1107
 The Author		1109
Index		1111

Index

^, logical operator	130	%f, format specifier	361
−, subtraction operator	121	%n, format specifier	361
!, logical operator	130	%s, format specifier	361
?, generics	716	%t, format specifier	361
..., variable argument list	257	%x, format specifier	361
.class	617, 843	+, addition operator	121
.NET Core	59	=, assignment operator	119
.NET Framework	58	==, reference comparison	619
@AfterAll	1084	== reference comparison	236
@AfterEach	1084	, logical operator	130
@author, Javadoc	1097	\$, inner class	602, 609
@BeforeAll	1084	1/1/1970	858
@BeforeEach	1084		
@code, Javadoc	1098	A	
@Deprecated	678		
@Deprecated, annotation	1103	abs(), math	1032
@deprecated, Javadoc	1102	Absolute value	149
@Disabled	1079	abstract, keyword	468, 470
@DisplayName	1078	Abstract class	468
@exception, Javadoc	1097	Accessibility, modular system	831
@FunctionalInterface	678, 740	Accessible, catch	592
@link, Javadoc	1098	Accessible source code	185
@linkplain, Javadoc	1098	Access method	384
@literal, Javadoc	1098	Actuator	205
@NonNull	786	Adapter	921
@Nullable	786	--add-exports, switch	823, 825
@Override	460, 485, 618, 678	Addition	121
@param, Javadoc	1097	--add-modules, switch	825
@return, Javadoc	1097	Adjusted exponent	1030
@SafeVarargs	690	Adobe Flash	61
@see, Javadoc	1097	AdoptOpenJDK	67
@SuppressWarnings	678, 679	Ahead-of-time-Compiler	1093
@Test	1072	Algebra	
@throws, Javadoc	1097	<i>Boolean</i>	130
@version, Javadoc	1097	<i>linear</i>	1066
*, multiplication operator	121	Alias	231
*7	44	American Standard Code for Information	
/, division operator	121	Interchange -> see ASCII	287
//, line comment	92	Amount	1032
&, generics	709	Ancestor, inheritance	438
&&, logical operator	130	And	
#ifdef	54	<i>bitwise</i>	133, 1012
%, Modulo operator	122, 1035	<i>logical</i>	133
%%, format specifier	361	Android	65
%b, format specifier	361	Angle function	1040
%c, format specifier	361	Annotation	459
%d, format specifier	361	Anonymous inner class	607
%e, format specifier	361	Aonix Perc Pico	67

Apache Commons CLI 282
Apache Commons Lang 619
Apache Maven 78
append(), StringBuilder/StringBuffer 335
Appendable, interface 336, 991
Applet 45
Application class loader 816
Arcus function 1040
Argument 93
 of functions 183
 optional 195
Argument count, variable 257
Ariane 5, crash 1017
ArithmeticException 557, 1054
Arithmetic operator 121
ARM -> see Automatic Resource Management
 (ARM) 580
Array 241
 multidimensional 259
Array bound 52
arraycopy(), system 266
ArrayIndexOutOfBoundsException 557
ArrayList, class 432, 909, 918
ArrayStoreException 478, 926
Array type 207
ASCII character 85
asin(), math 1040
asList(), arrays 277, 921
Assert, class 1073
Assertion 595
AssertionError 595
AssertJ 1081
assertXXX(), assert 1073
Assignment 119, 120, 133
 with operation 133
 with string concatenation 133
Association 427
 circular 429
 recursive 429
 reflexive 429
Assumptions, class 1079
Attributes 203, 204
Autoboxing 139, 665
Autobuild 80
Automatic garbage collection 393, 405
Automatic memory release 51
Automatic module 830
Automatic-Module-Name 832
Automatic Resource Management (ARM) 580
 block 580

B

Base-ten system 1014
Basic Multilingual Plane (BMP) 294
Bias 1030
BiConsumer, interface 779
Bidirectional relationship 427
BiFunction, interface 780
BigDecimal, class 1051, 1059
Big endian 1053
BigInteger, class 1051
Billion-dollar mistake 787
Binary compatibility 497
Binary Floating-Point Arithmetic 1025
Binary name
 inner type 602
Binary operator 119
BinaryOperator, interface 780
Binary representation 349
binarySearch(), arrays 275
Binary system 1014
Binding, late dynamic 464
BiPredicate, interface 782
Bitwise exclusive OR 1012
Bitwise NOT 1012
Bitwise OR 1012
Blank character 298
Block 101
 empty 101
BMP -> see Basic Multilingual Plane
 (BMP) 294
Body 177
BOM -> see Byte Order Mark (BOM) 295
boolean, data type 104
Boolean algebra 130
Boolean class 664
Bootstrap class loader 816
Boxing 615, 666
Break 1066
break, keyword 171
Byte 1011
byte, data type 1014
Byte class 655
Bytecode 46
byte datatype 104, 112
Byte Order Mark (BOM) 295
BYTES, wrapper constant 661

C

C 43
C++ 44, 203

Calculation inaccuracy 169
Calendar, class 858
Call, cascaded 211
Callable, interface 897
Callable objects 902
Call by Reference 235
Call by Value 184, 234
Call stack 198
Camel case 86
CamelCase notation 87
Canonical constructor, record 529
Cardinality 427
Cascaded call 211
Case sensitivity 86, 320, 326
Cast 132, 135
catch, keyword 540
ceil(), math 1033
Central Repository, Maven 868
Chained list 919
char, data type 1014
Character 102, 115
 appending from 339
 replacing 328
Character string 93
 constant 303
 mutable 333
charAt(), string 310
char datatype 104, 112, 115
CharSequence, interface 302, 342
Checked exception 558
Child class 437
Class 47, 203
 abstract 468
 anonymous inner 607
 concrete 468
 final 466
 local 605
 nested top-level 601
 partially abstract 470
 pure abstract 470
 sealed 525
 static inner 601
Class, class 616, 843
class, keyword 369
ClassCastException 450, 557
Class concept 57
Class diagram 205
Class hierarchy 438
Class initializer 420
Class literal 616
Class loader 813
Class loading
 explicit 814
 implicit 814
Class method 182
ClassNotFoundException 844, 845
Class object 616
CLASSPATH 818, 1093, 1107
-classpath 818, 1093
Classpath-Hell 819
Class-Path-Wildcard 1095
Class property 391
Class type 207
Cleanable, interface 638
Cleaner class 638
clone(), arrays 264
clone(), object 625
Cloneable, interface 626
CloneNotSupportedException 626, 628
Cloning 625
Closeable, interface 995
COBOL 32
Code page 295
Code point 85, 287
Collator class 644
Command line parameter 282
Command not found 75
Comma operator 167
Comment 92
Commercial rounding 1034
Comparable, interface 489, 643, 657
Comparator, interface 643
compare(), comparator 645
compare(), wrapper classes 657
compareTo(), Comparable 645
compareTo(), string 318
compareToIgnoreCase(), string 318
Comparison operator 128
Comparison string 320
Compilation, conditional 54
Compilation unit 92, 226
Compiler 72
 incremental 80
 native 1093
Compiler error 74
Compile-time constant expression 421
Complement 1012
 bitwise 132
 logical 132
CompletableFuture, class 905
CompletionService, interface 903
CompletionStage, interface 905
Complex number 1066

Compound assignment operator, compound operator	126
Compound operator	126
Concatenation	303
Concrete class	468
Condition, composite	144
Conditional compilation	54
Conditional operator	148
Condition operator	133, 148
Configuration, reliable	831
Congruence, linear	1047
Conjunction	130
const, keyword	235, 236
Constant	
<i>inherited</i>	494
<i>symbolic</i>	397
Constant pool	332
const-correct	236
Constructor	217
<i>general</i>	408
<i>inheritance</i>	442
<i>parameterized</i>	408
<i>parameterless</i>	405
Constructor call	207
Constructor redirection	442
Constructor reference	759, 760
contains(), string	311
containsKey(), Map	944
contentEquals(), string	340
continue, keyword	173
Contravalance	130
Control structure	142
Converting radian measure	1041
Coordinates, Maven	77
Copy	
<i>deep</i>	628
<i>flat</i>	628
Copy constructor	217, 410, 625
Copy initialization block	610
copyOf(), arrays	273
copyOfRange(), arrays	273
CopyOnWriteArrayList, class	909
Copy variable	208, 390
cos(), math	1040
cosh(), math	1041
Cosine	1040
Covariance for arrays	477
Covariant, array	926
Covariant, generics	714
Covariant return type	477
Coverage, test	1071
Covered, attribute	457
Covered method	476
Covert method	476
-cp	818, 1093, 1095
Creating a stream	947
Currency, class	1065
currentThread(), Thread	876
currentTimeMillis(), system	842, 858
D	
-D	850
Dangling-else problem	145
Dangling pointer	417
Database query	1008
Database schema	1006
Database specification	1006
DataInput, interface	983
DataOutput, interface	983
Data pointer	984
Data structures	909
Data type	102
<i>integer</i>	112
Date, class	857
DateFormat, class	858
Date values	857
Daylight Saving Time/Winter Time	857
DbUnit	1089
Deadlock	873
Decimal point	1025
Decimal system	1014
Declarative programming	763, 944
Deconstructor	417
Decrement	132
Deep copy, clone()	628
deepEquals(), arrays	270, 635
deepHashCode(), arrays	635
default, switch-case, keyword	152
Default constructor	407
Default package	225
Deferred initialization	110
Degree	1041
delegate	57
delete(), StringBuffer/StringBuilder	338
Dependency inversion principle	801
Deployment	817
Deprecated	1102
-deprecation	1103
De-referencing	50, 141
Descendant, inheritance	438
Design by contract	594
Design pattern	804
Diamond operator	691

Diamond type	691
digit(), character	300
DIP -> see Dependency inversion principle	801
DirectX	58
Disjunction	130
Dispatcher, operating system	872
Dividend	121, 123, 1037
Division	121
<i>remainder</i>	1036
Division operator	121
Divisor	121, 123, 1037
Doc comment	1096
Doclet	1101
Documentation comment	1096
Don't Repeat Yourself -> see DRY	800
double, data type	1025
Double bracket initialization	610
Double class	655
double datatype	104
doubleToLongBits(), Double	636, 1030
do-while loop	163
DRY	800
Dummy object	1088
E	
-ea, switch	595, 1094
EasyMock	1088
Eclipse	80
Eclipse Adoptium	67
Eclipse Language Pack	81
ECMAScript	57
Effectively final	606, 745
Ego approach	204
else, keyword	145
EmptyStackException	557
Empty string	330, 332, 333
Enable assertions	595
End recursion	197
endsWith(), string	319
Enhanced interface for ILM	499
ensureCapacity(), list	919
Entities	1005
enum, keyword	400
Enum class	513
Enumeration type	400
Environment variables (operating system)	852
EOFException	982
Equality	238
equals(), arrays	269, 635
equals(), object	237, 619
equals(), string	340
equals(), URL	624
equalsIgnoreCase(), string	316, 317
Equivalence	130
Error	552
Error class	551, 593
Error code	539
Error message, non-static method	182
Escape character	312
Escape sequence	292
Escher, Maurits	200
Euler number	1031
Euro sign	293
EventListener, interface	808
EventObject, class	808
Exception	52
<i>nested</i>	577
<i>tested</i>	558
<i>testing</i>	1076
<i>untested</i>	558
<i>untreated</i>	887
Exception class	551
Exception handler	543
Exception parameter	543
Exception transparency	752
Execute-around-method pattern	773
Executor, interface	893
ExecutorCompletionService, class	903
ExecutorService, interface	895
exit(), system	283
Exit code	283
Explicit class loading	814
Exponent	1029
<i>adjusted</i>	1030
Exponential value	1039
Expression	98
<i>pure</i>	765
Expression specification	205
Expression statement	99, 120
Extended for loop	253
extends, keyword	437, 493
Extension class	437
Extension class loader	816
Extension method, virtual	499
F	
Factorial	1058
Factory	804
Factory method	806
Fairy, the good	195

Fake object	1087	Functional interface	734
Fall-through	153	Functional programming	762
FALSE, Boolean	664	Function descriptor	734
false, keyword	104	Future, interface	899
fastutil	928	FutureTask, class	901
Feature		G	
<i>object oriented</i>	47	Garbage collection, automatic	393
<i>static</i>	391	Garbage collector	393, 416
Fencepost error	166	Gaussian normal distribution	1048
Field		GC -> see Garbage collector	416
<i>non-primitive</i>	254	GCD	1051
<i>non-rectangular</i>	262	General constructor	408
<i>two-dimensional</i>	259	Generic method	695
File name extension	319	Generics	433, 686
FileNotFoundException	552	Generic type	686
FileSystem, class	970	GERMAN, locale	855
fill(), arrays	272	get(), list	910
fillInStackTrace(), Throwable	576	get(), map	943
final, keyword	110, 397, 466, 467	getBoolean(), Boolean	653
Final class	466	getChars(), string	324
finally, keyword	561	getClass(), object	843
Final method	467	getInteger(), integer	653
Final values	425	getNumericValue(), character	300
FindBugs	569	getPriority(), thread	892
First Person, Inc.	44	getProperties(), system	848
Fixture	1083	getResource(), class	1001
Flat copy, clone()	628	getResourceAsStream()	1001
Flat object copy	628	Getter	385
Float	102, 115, 1025	Global variable	193
float, data type	1025	Glyphs	290
Float class	655	Gosling, James	44
float datatype	104	goto, keyword	174
Floating point number	1025	GraalVM	50
floatToIntBits(), float	636	Grammar	83
floor(), math	1033	Granularity, threads	892
Flushable, interface	996	Greatest common divisor -> see GCD	1051
Fonts	290	Green OS	44
forDigit(), character	301	Green project	44
for-each loop	160	Groovy	60
for loop	164	H	
<i>extended</i>	253	H2, database	1007
format(), PrintWriter/PrintStream	362	Hamcrest	1081
format(), string	361	Hangman	314
Format specifier	361	Hash code	630
Format string	361	hashCode(), arrays	635
forName(), class	844	hashCode(), object	630
for-statement	165	Hash function	630
FRENCH, locale	855	HashMap, class	938
Function			
<i>first class</i>	768		
<i>higher order</i>	767		
<i>pure</i>	764		
<i>variadic</i>	257		

HashSet, class	932	Inheritance	215, 437
Hash table	938	Inherited constant	494
Hash value	630	Initialization, deferred	110
hasNextLine(), scanner	357	Initializing a class variable	421
Header	177	Initial module	828
Header, record	529	Inline tag	1100
Header file	54	Inner loop	167
Heap	212	Inner type	600
Helper class	408	InputStream, class	988
Hexadecimal number	1015	Instance	203
Hexadecimal representation	349	Instance initializer	422
Hexadecimal system	1015	Instance variables	208
HexFormat, class	353	int, data type	1014
Hidden variable	377	int datatype	104, 112
Hoare, Tony	787	Integer	102
HotJava	45	Integer class	655
HotSpot	49	Integration test	1070
Human time	861	IntelliJ IDEA	76
I		Interaction diagram	206
Identifier	85	Interface	57, 468, 483, 484
Identifiers	85	<i>enhanced</i>	499
Identity	238, 619	<i>functional</i>	734
identityHashCode(), system	632, 637	interface, keyword	484
IEEE 754	115, 123, 1025	Interface method, static	495
IEEEremainder(), math	1035	Interface segregation principle	801
if cascade	148	Interface type	207
if statement	142	Intermediary Operation	946, 962
<i>accumulated</i>	148	Intermediary operation	
Ignored status return value	546	<i>stateful</i>	963
IllegalArgumentException	557, 564, 566, 568, 570	Internal iteration	675
IllegalMonitorStateException	558	Interpreter	72
IllegalStateException	566	Interrupt	885
IllegalThreadStateException	878	interrupt(), thread	885
Imagination	44	interrupted(), thread	887
Immutable	414	InterruptedException	884, 887
immutable	301	Interval	170
Imperative programming language	91	Invariant, generics	714
Implication	130	IOException	552
Implicit class loading	814	Is-a-kind-of-relationship	468
import, keyword	222	isInterrupted(), thread	885
Import, static	226	is-method	385
Increment	132	isNaN(), double/float	1028
Incremental compiler	80	ISO/IEC 8859-1	288
Index	241, 248	ISO-639 code	854
Indexed variable	246	ISO 8859-1	85, 289
indexOf(), string	312	ISO abbreviation	856
IndexOutOfBoundsException	249, 567	ISO Country Code	854
Infinite	1026	ISO Language Code	854
Infinite loop	161	ISP -> see Interface segregation	
		principle	801
		ITALIAN, locale	855
		Iterable, interface	674, 675

Iteration	945
<i>internal</i>	675
<i>internal versus external</i>	945
Iterator, interface	670
iterator(), iterable	674
J	
J/Direct	57
Jakarta EE	65
JamaicaVM	67
JAR	1105
-jar, java	1107
jar, utility	1105
JAR file	815
jarsigner, utility	1091
Java	45
java, package	221
java, utility	1093
java.lang.ref, package	638
java.nio.file, package	970
javac, utility	1092
Java Card	65
Java Database Connectivity (JDBC)	1006
Javadoc, utility	1096, 1098
JavaFX platform	61
JavaFX script	62
Java Mission Control, software	843
Java Platform, Enterprise Edition (Java EE)	65
Java Platform, Micro Edition (Java ME)	62, 65
Java Platform, Standard Edition (Java SE)	61
Java Platform Module System (JPMS)	819
Java Plug-in	61
Java Runtime Environment (JRE)	1106
JavaScript	57
JavaSoft	45
Java Specification Request -> see JSR	60
Java TV	65
Java Virtual Machine (JVM)	46
javaw, utility	1095
javax, package	221, 841
Jigsaw	819
JIT -> see Just-in-time (JIT) compiler	49
join(), thread	890
Joy, Bill	43
JPMS -> see Java Platform Module System	
(JPMS)	819
JProfiler, software	843
IRuby	60
JSR	60
JSR 354	
<i>Money and Currency API</i>	1064

Jump label, switch	152
Jump target, switch	152
JUnit	1069
<i>execution times</i>	1077
<i>running test</i>	1072
<i>versions</i>	1069
Just-in-time (JIT) compiler	49
Jython	60

K

Keep It Simple, Stupid -> see KISS	800
keytool	1091
Keyword	87
<i>reserved</i>	87
KISS	800
KOREAN, locale	855
Kotlin	59, 76

L

Lambda expression	733
lastIndexOf(), string	313
Late dynamic binding	464
Latin-1	288
Left associativity	134
length(), string	309
LESS principle	722
Lexicon	83
Lifetime	193
line.separator	852
Linear algebra	1066
Linear congruence	1047
Line break	851
Line comment	92
lineSeparator(), system	852
LinkedHashSet, class	937
LinkedList, class	909, 919
Linking	813
Liskov, Barbara	452
Liskov substitution principle	452
Liskov substitution principle (LSP)	801
List	909
<i>chained</i>	919
List, interface	909
Listener	807
ListIterator, interface	923
Literal	87
Local class	605
Locale	855
Locale, class	326, 853
log(), math	1039

log4j	865
Logical operator	130
Log level	866
long, data type	104, 112, 1014
longBitsToDouble(), double	1030
Long class	655
Loop	160
<i>inner</i>	167
<i>outer</i>	167
Loop condition	163, 168
Loop counter	165
Loop increment	165
Loop test	165
Lower-bounded wildcard type	717
LSP -> see Liskov substitution principle	
(LSP)	801
M	
Machine time	861
Magic number	398
main()	74
Main-Class	1106
Main class	92
Main-Thread	876
MANIFEST.MF	1105
Mantissa	1029
Map, interface	938
Marker	174
Marker annotation	679
Marker interface	512
Masking	330
Math, class	1031
MathContext, class	1062
Maven	78
<i>Goals</i>	869
<i>Phases</i>	869
MAX_PRIORITY, thread	892
MAX_RADIX	301
max(), math	1032
Maximum	149, 1032
McNealy, Scott	44
Member class	600
Memory leak	417
Memory release, automatic	51
MESA, programming language	43
Metadata	459
META-INF/MANIFEST.MF	1105
Meta object	843
Method	177
<i>covered</i>	476
<i>covert</i>	476

Method (Cont.)	
<i>final</i>	467
<i>generic</i>	695
<i>native</i>	56
<i>overloaded</i>	96, 191
<i>overwriting</i>	457
<i>parameterized</i>	183
<i>recursive</i>	195
<i>static</i>	182
<i>synthetic</i>	602
method	203
Method body	177
Method call	93, 180, 374
Method header	177
Method reference	755, 756
<i>variants</i>	756
<i>with</i>	756
Microsoft Development Kit	57
MIN_PRIORITY, thread	892
MIN_RADIX	301
MIN_VALUE	1042
min(), math	1032
Minimum	149, 1032
Minus, unary	124, 132
Mixin	508
Mockito	1088
Mock object	1088
Model View Controller	807
Modifier	100, 179
Module	
<i>automatic</i>	830
<i>initial</i>	828
<i>readability</i>	831
<i>unnamed</i>	830
Module info file	826
Module system, accessibility	831
Moneta	1064
multicast	57
Multi-catch	548
Multidimensional array	259
Multilevel continue	174
Multiple branching	147
Multiple inheritance	488
Multiplication	121
Multiplicity	427
Multitasking	872
Multitasking capability	872
Mutator	374

N

NaN 122

nanoTime(), system 842

Narrowing conversion 137

native, keyword 628

Native compiler 1093

Native method 56

Natural order 643

Naughton, Patrick 44

NavigableMap, interface 940

NavigableSet, interface 935

NEGATIVE_INFINITY 1042

Negative sign 119

Nest 613

Nested exception 577

Nested tests 1078

Nested top level class 601

Nested type 600

NetBeans 81

Netscape 57

new, keyword 206, 405

Newline character 851

newOutputStream(), files 997

nextLine(), scanner 357

No-Arg constructor 217

no-arg-constructor 405

NoClassDefFoundError 845

Nominal tuple 529

Non-number 122

Non-primitive field 254

Normal distribution 1048

Gaussian 1048

NoSuchElementException 671

Not 130

bitwise 1012

Not a Number (NaN) 1026, 1042

quiet 1029

signaling 1029

Not a Number -> see NaN 1026

Notation, scientific 1029

NULL 539

null, keyword 228

Nullary constructor 405

Null object 474

Null-Object pattern 787

NullPointerException 229, 248, 558, 568, 784

null reference 228

Number

complex 1066

Euler 1031

Number (Cont.)

hexadecimal 1015

magic 398

Number class 655

NumberFormatException 348, 542, 546

Number value, Unicode character 85

Numeric promotion 121

O

Oak 44

Obfuscator 845

Object approach 204

Object class 440, 616

Object copy, flat 628

Object diagram 205

Object equivalence 619

Object graph 417

Object identification 617

Objective-C 57

Object orientation 47, 99

Object-oriented approach 57

Objects class 639

Object type 50, 449

Object variable

initializing 418

Observer pattern 807

Octal number representation 349

Octal system 1014

Off-by-one error 165

Olson, Ken 1067

Open-closed principle 801

OpenJDK 55, 67

OpenJFX 62

Operating system independence 46

Operation 203

intermediary 946, 962

reducing 946

terminal 946, 951

Operator 119

arithmetic 121

binary 119

logical 130

monadic 119

overloaded 54

question mark 119

relational 128

ternary 149

trinary 149

two-digit 119

unary 119

Operator precedence 132

Operator rank 132

Operator ranking 132

Optional class 787

OptionalDouble class 793

OptionalInt class 793

OptionalLong class 793

OR

bitwise 133, 1012

exclusive 130

exclusive bitwise 1012

logical 133

Oracle Corporation 45

Oracle JDK 68

Oracle OpenJDK 67

Order, natural 643

ordinal(), enum 517

Ordinal number, enum 516

Outer loop 167

OutOfMemoryError 212, 594, 626

Output formatting 857

OutputStream, class 986

Overflow 1042

Overloaded method 96, 191

Overloaded operator 54

Overwrite

method 457

P

Package 220

unnamed 225

package, keyword 224

package-private 381

Package visibility 386

Pair of parentheses 180

Parallel 871

Parameter 183

current 183

formal 183

Parameterized constructor 408

Parameterized tests 1079

Parameterized type 688

Parameterless constructor 405

Parameter list 177, 180

Parameter passing mechanism 184

parseBoolean(), Boolean 347

parseByte(), byte 347

parseDouble(), double 347

parseFloat(), float 347

parseInt(), integer 347, 351, 546, 663

parseLong(), long 347, 351

parseShort(), short 347

Partially abstract class 470

PATH 71

Path, interface 970

Paths, class 971

Payne, Jonathan 45

PDA 65

PhantomReference class 638

Pi 1031

Pipeline, stream API 946

Place value system 1014

PlantUML 239

Platform independence 46

Plugin

Maven 869

Plus, overloaded 140

Plus/minus, unary 132

Plus/minus sign reversal 124

Point, class 204, 206

Pointer 50

Point operator 208

Polar method 1048

Polymorphism 465

POM file 78

POSITIVE_INFINITY 1042

Post-decrement 125

Post-increment 125

Potency 1039

Pre-decrement 125

Prefix 319

Prefix operation 965

Pre-increment 125

print() 95, 192

printf() 96

printf(), PrintWriter/PrintStream 362

println() 95

printStackTrace(), throwable 550

Priority, thread 892

Priority queue 892

Privacy 380

private, keyword 381

Process 872

Profiler 843

Profiling 842

Program 92

Programming

against interfaces 488

declarative 763, 944

Programming against interfaces 802

Programming language, imperative 91

Programming paradigm 763

Promotion, numeric 121

Properties, class 848

Property 384
protected, keyword 442
Protocol 57
Prototype 411
Prototype-based object-oriented
 programming 204
Provider 773
Pseudo-prime number test 1051
Pseudo-random number 1047
public, keyword 381
Pure abstract class 470
Pure expression 765
Pure function 764
Purely abstract class 470
put(), map 942

Q

Quasi-parallelism 871
Question mark operator 119
Quotation marks 115
Quote, single 115
quote(), pattern 330

R

Random, class 1046
random(), math 255, 1041
RandomAccess, interface 918
RandomAccessFile, class 980
Random number 1041, 1053
Random number generator 1047
Range-Checking 52
Ranking 132
Raw type 704
Readability, module 831
readability, module 831
Reader, class 992
readUTF(), RandomAccessFile 982
Real-time Java 66
Real-time specification for Java ->
 see RTSJ 66
record, keyword 529
Records 529
Recursion type 197
Recursive method 195
Recursive type bound 709
Reducing operation 946
Reference 207
 abstract class 638

Reference (Cont.)

strong 638
 weak 638
Reference type 50, 98, 103, 449
 comparison with == 236
Reference variable 50, 207
Referencing 141
Referential transparency 765
regionMatches(), string 319
Relation 1005
Relational database 1005
 access with Java 1008
Relational database system 1006
Relationship, bidirectional 427
Reliable configuration 831
reliable configuration 831
Remainder of a division 1036
Remainder operator 121, 122, 1035
Removing whitespace 327
Rendezvous 889
replace(), string 328
replaceAll(), string 329
replaceFirst(), string 329
requireNonNull(), objects 569
Restricted identifiers 88
Result 98
Result type 177
resume(), thread 891
rethrow, exceptions 574
return, keyword 185, 250
Return type 177
 covariant 477
Reuse via copy & paste 1018
Reverse function 1040
Right associativity 134
rint(), math 1034
round(), math 1034
Rounding 1033
 commercial 1034
Rounding down 1033
RoundingMode, enumeration 1062
Rounding modes, BigDecimal 1061
Rounding up 1033
RTSJ 66
run(), runnable 877
Runnable 879
 parameterizing 879
Runnable, interface 876
Runtime environment 46
RuntimeException 541
Runtime interpreter 46

S

SapMachine 68
Scala 60
Scanner, class 356
ScheduledExecutorService, interface 904
ScheduledThreadPoolExecutor, class 894,
 904
Scheduler, operating system 872
Scientific notation 1029
Scope 193
Sealed class 525
Secondary 871
SecureRandom, class 1051
Seed 1047
Selenium 1089
Semantics 83
Separator 84, 355
Sequence diagram 206
Set, interface 928
setPriority(), thread 892
Setter 385
Set-top boxes 44
Sexadecimal system 1015
Shadowed variable 376
Sheridan, Mike 44
Shift 132
Shift operator 1021
short, data type 1014
Short-circuit operator 131
Short class 655
short datatype 104, 112
Side effect 374
Sign, negative 119
Signaling NaN 1029
Signature 178
Sign extension 132
Silent NaN 1029
Silverlight 61
Simula 57, 201
sin(), math 1040
Sine 1040
Single inheritance 440
Single quote 115
Singleton 401, 802, 805
Single-value annotation 679
sinh(), math 1041
sizeof 141
sleep(), thread 883
Smalltalk 47, 201
Smiley 293
sNaN 1029

SoftReference class 638
Software architecture 803
Software test 1067
SOLID 800
sort(), arrays 268
SortedMap, interface 940
Source code, accessible 185
Spacebar 298
split(), string 356
Square root 1038
SRP (Single Responsibility Principle) 800
Stack 198
Stack-case label 155
Stack memory 213
StackOverflowError 198, 594
Stack trace 545
Standard Directory Layout 77
Standard Extension API 841
Star Seven 44
start(), overwriting method 881
start(), thread 878
startsWith(), string 319
State, thread 882
Statement 91
 atomic 95
 blank 95
 elementary 95
 nested 147
Statement sequence 95
static, keyword 100, 392
Static (initialization) block 420
Statically typed 102
Static feature 391
Static import 226
Static inner class 601
Static interface method 495
Status return value, ignored 546
stop(), thread 885
Stream API 944
Stream class 985
Strictly typed 102
String
 appending to a 325
 length 309
StringBuffer 302, 333
StringBuilder 302, 333
String concatenation 132
StringIndexOutOfBoundsException ... 311, 322
String literal 303
String part
 comparing 319
 extracting 310, 320

Strongly typed 102

Strong reference 638

Stroustrup, Bjarne 203

Stub object 1088

STUPID 802

Subclass 437

Subinterface 493

Substitution principle 452

Substring 311

substring(), string 320

Subtraction 121

Suffix 319

Sun Microsystems 45

SunWorld 45

super, keyword 461

super() 442, 444, 446

Superclass 438

Supplier 773

Suppressed exception 563

Surrogate pair 295

suspend(), thread 891

switch statement 151

Symbolic constant 397

Symmetry, equals() 622

Synchronization 637

Syntax 83

Synthetic method 602

System.err 100, 550

System.in 988

System.out 99

System class loader 816

System property 848

T

TAIWAN 855

tan(), math 1040

tanh(), math 1041

Target type, lambda expression 736

TCFCTC 561

TCK -> see Technology Compatibility Kit (TCK) 67

TDD 1070

Technology Compatibility Kit (TCK) 67

Template pattern 988

Terminal operation 946, 951

Test

ignoring 1078

nested 1078

parameterized 1079

Test-Driven Development -> see TDD 1070

Test-First 1069

Test Runner 1068

this() 446

this(), constructor call 412

this\$, inner class 605

this reference 377, 446

inner class 604

Thread 873

condition 882

granularity 892

name 882

native 873

priority 892

properties 882

Thread, class 878

extending 880

Thread end 887

ThreadLocalRandom, class 950

Thread pool 893, 895

ThreadPoolExecutor, class 893

Three-way comparison 645

throw, keyword 564

Throwable class 550

throws, keyword 549

Time measurement 842

TimeUnit, enumeration 859

Time zone 857

toArray(), collection 924, 925

toBinaryString(), integer/long 349

toCharArray(), string 324

toHexString(), integer/long 349

Token 84

toLowerCase(), character 299

toLowerCase(), string 326

toOctalString(), integer/long 349

Top level class, nested 601

toString(), arrays 267

toString(), object 458, 617

toString(), point 210, 211

toUpperCase(), character 299

toUpperCase(), string 326

Towers of Hanoi 198

Trait 508

Translation error 74

Transparency, referential 765

TreeMap, class 939

TreeSet, class 933

trim(), string 327

TRUE, Boolean 664

true, keyword 104

Truth value 102

try, keyword 540

try block 543

try with resources 358, 580

Tuple 1005

nominal 529

Two's complement 1013

Two-dimensional field 259

Type

arithmetic 104

generic 686

inner 600

integral 104

nested 600

numeric 98

parameterized 688

primitive 103

Type, interface 846

TYPE, wrapper class 617, 843

Type annotation 677

Type bound, recursive 709

Typecast 135

Typecasting 135

automatic 135, 447

explicit 135, 449

Type comparison 133

Typed

static 102

strict 102

strong 102

Type deletion 699, 700

Type inference 691, 696

Type token 725

Type variable 686

U

U+, Unicode 289

UK, locale 855

UML -> see Unified Modeling Language (UML) 204

Umlaut 292

Unary minus 124

Unary operators 119

Unary plus/minus 132

Unboxing 666

UnhandledExceptionHandler, interface 888

Unchecked exception 571

UncheckedIOException 579

Underscore in numbers 114

Unicode 5.1 289

Unicode character 85

Unicode Consortium 289

Unicode-Escape 293

Unidirectional relationship 427

Unified Modeling Language (UML) 204

Unit test 1068

Unix epoch 858

Unix time 858

Unnamed module 830

Unnamed package 225

UnsupportedOperationException 558, 589, 673, 922

Untested exception 558

Upper-bounded wildcard type 717

Upward compatibility 64

US, locale 855

Use case 205

Use case diagram 205

UTF-16 encoding 291, 295

UTF-8 encoding 291, 982

Utility class 408

V

Value object 414, 654

valueOf(), enum 515

valueOf(), string 345

valueOf(), wrapper classes 653

Value operation 119

Value range 384

Value transfer 183, 184

Vararg 257

Variable

global 193

hidden 377

indexed 246

Variable declaration 106

Variable initialization 478

Variadic function 257

Vector, class 909

-verbose 1094

Virtual extension method 499

Virtual machine 46

Visibility 193, 380

Visibility modifier 381

Visual Age for Java 80

VisualVM, software 843

void, keyword 181

void-compatible 743

W

Weak reference 638

WeakReference class 638

Web 44

Web applet	45	XOR	130
WebRunner	45	<i>bitwise</i>	133
while loop	161	<i>logical</i>	133
Whitespace	298	-Xrs	1094
Widening conversion	136	-Xss	1094
Wildcard	716	<i>n</i>	198
Win32 API	57	-XX	
Windows-1252	289	<i>ThreadStackSize=n</i>	198
Wither method	415		
World Wide Web -> see Web	44		
Wrapper class	347, 651		
Writer, class	990		
writeUTF(), RandomAccessFile	982		
WWW -> see Web	44		

X

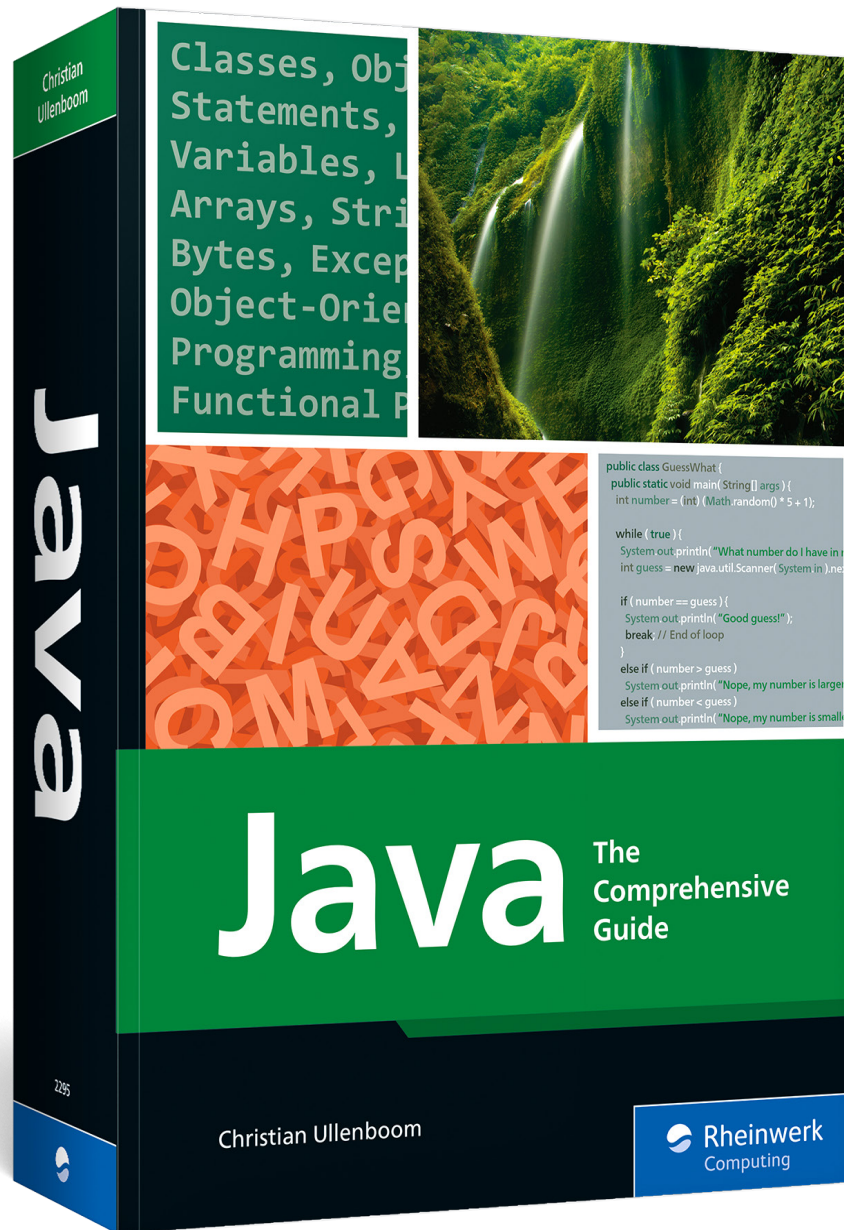
-Xms	1094
-Xmx	1094
-Xnoclassgc	1094

Y

YAGNI	800
Yoda style	129
You Ain't Gonna Need It -> see YAGNI	800
YourKit, software	843

Z

Zero-Assembler project	50
------------------------------	----



Christian Ullenboom is an Oracle-certified Java programmer and has been a trainer and consultant for Java technologies and object-oriented analysis and design since 1997.

Christian Ullenboom

Java: The Comprehensive Guide

1126 pages, 2023, \$59.95

ISBN 978-1-4932-2295-7



www.rheinwerk-computing.com/5557

We hope you have enjoyed this reading sample. You may recommend or pass it on to others, but only in its entirety, including all pages. This reading sample and all its parts are protected by copyright law. All usage and exploitation rights are reserved by the author and the publisher.