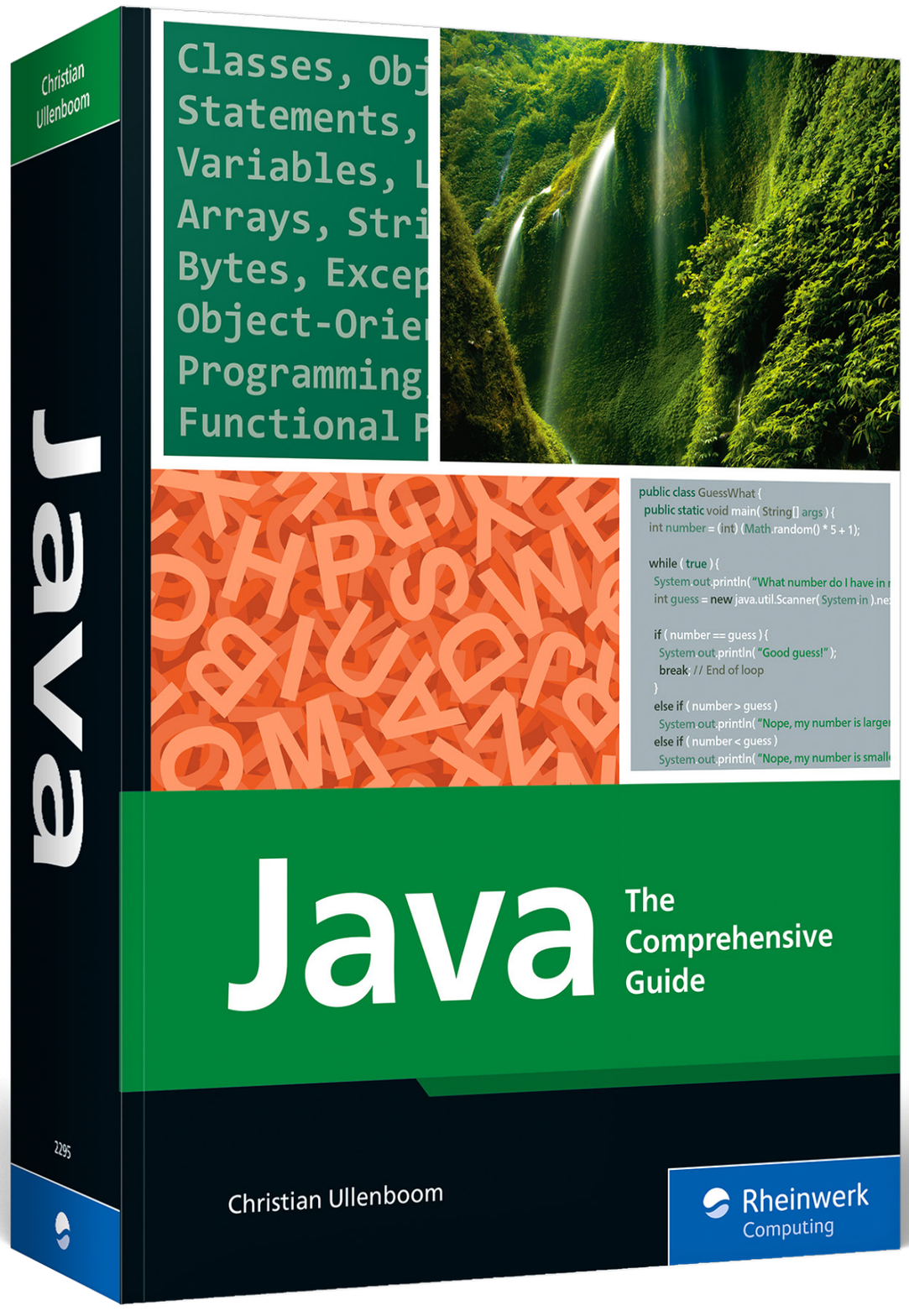


Reading Sample

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.

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.

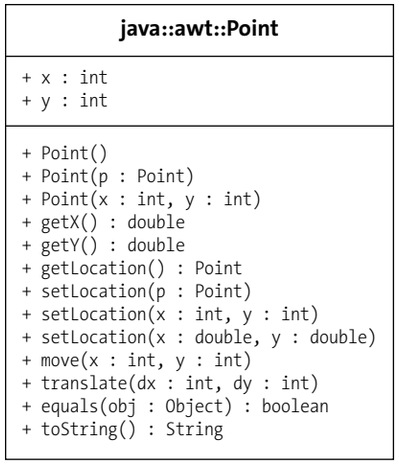


Figure 3.1 The java.awt.Point Class in a UML Representation

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.

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.

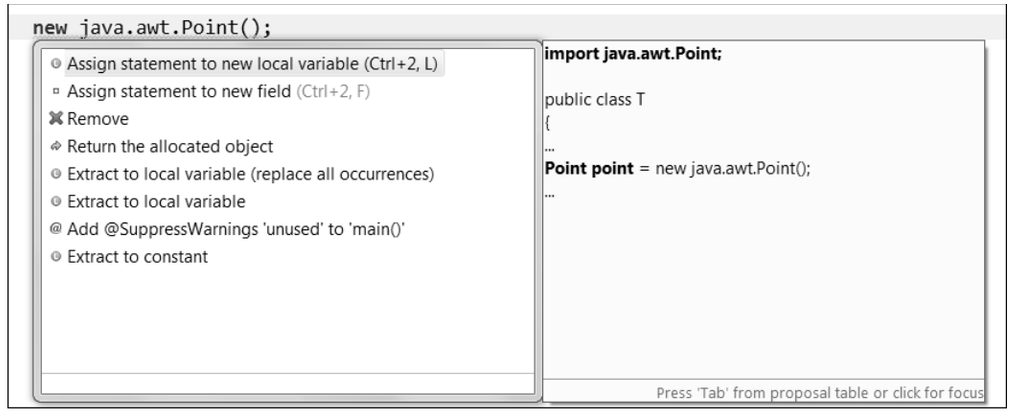


Figure 3.2 Pressing Ctrl+1 Allows You to Create Either a New Local Variable or an Object Variable for the Expression

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.

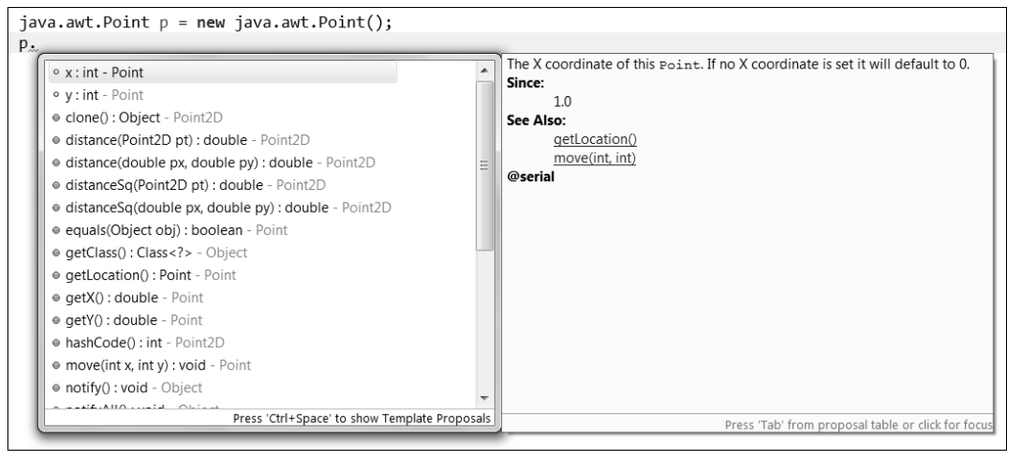


Figure 3.3 Ctrl+Space Displays the Possible Members of a Reference. Pressing Enter Selects the Member and, Especially for Methods, Places the Cursor between the Pair of Parentheses.

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
int playerX; int playerY;	java.awt.Point player;
int doorX; int doorY;	java.awt.Point door;

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
```

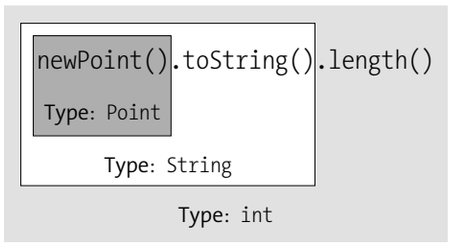


Figure 3.5 Each Nesting Results in a New Type

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`.

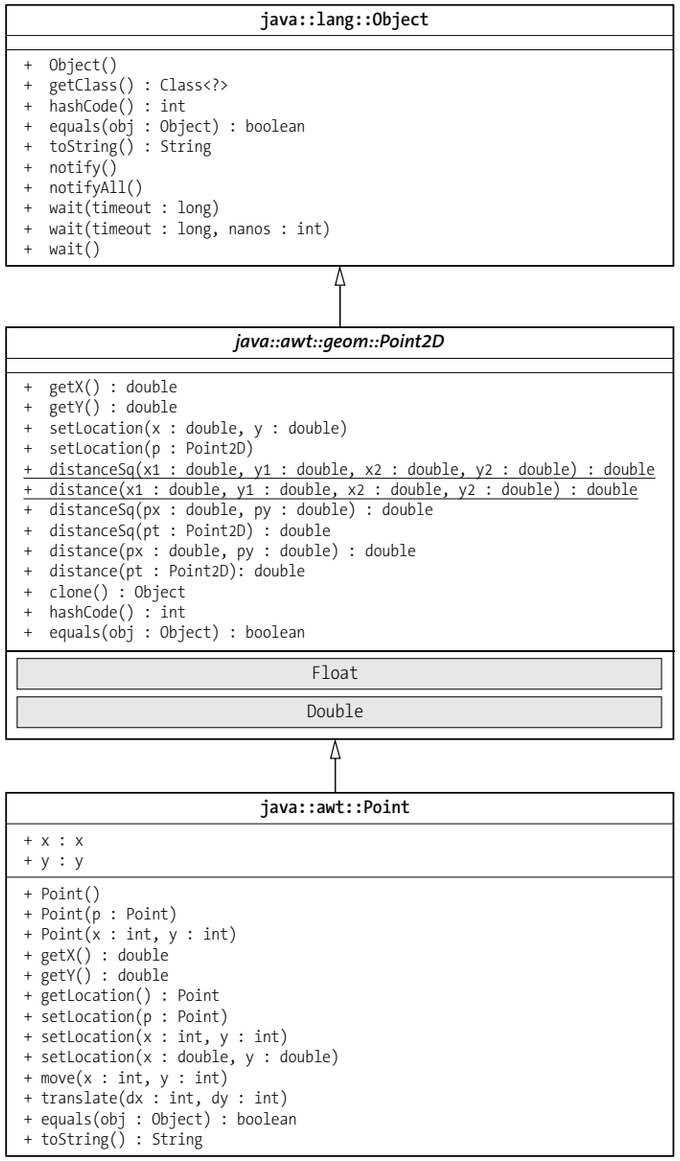


Figure 3.6 Inheritance Hierarchy in Point2D

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

```
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) );
 }
}
```

Listing 3.6: AwtWithoutImport.java

Import Declaration

```
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) );
 }
}
```

Listing 3.7: AwtWithImport.java

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 located[10] 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 );
    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

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.

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

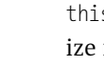
risk, with a missing type name, that where the member actually comes from will become invisible, as well as what dependency it is based on. Problems also arise with methods that have names that are homonyms: A method from its own class can overlay statically imported methods. So, if later in the custom 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.a
wt.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/oop/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 JavaIsAlwaysCallByValue.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.

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

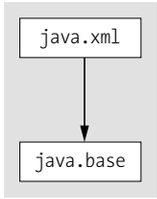


Figure 16.1 The java.xml Module Has a Dependency on the java.base Module

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

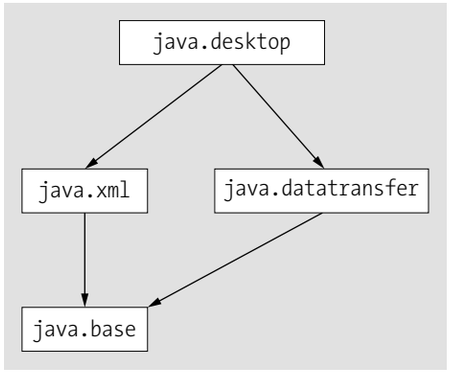


Figure 16.2 Dependencies of the java.desktop Module

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.

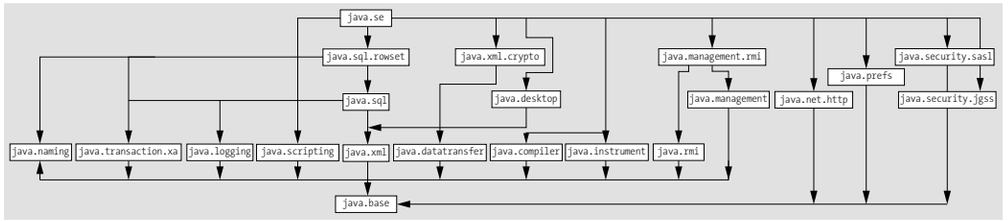


Figure 16.3 Dependencies of the java.se Module

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-object[2]—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`.

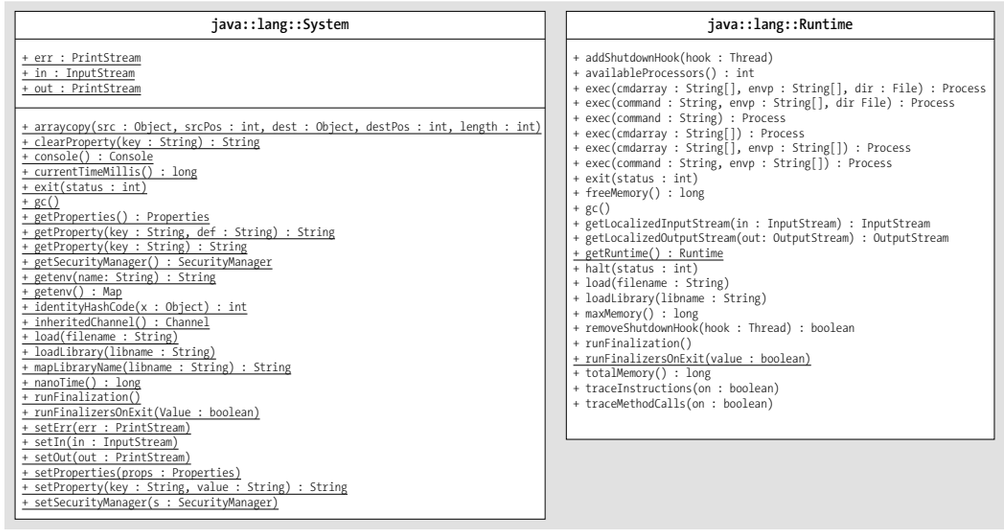


Figure 16.4 Members of the System and Runtime Classes

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.

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.

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=fa1se` 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 newline 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	*MOE*
HOMEDRIVE	Drive of the user directory	*C:*
HOMEPATH	Path of the user directory	*\Users\Christian*
OS	Name of the operating system*	*Windows_NT*
PATH	Search path	*C:\windows\SYSTEM32; C:\windows ...*
PATHEXT	File extensions that represent executable programs	*.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC*
SYSTEMDRIVE	Drive of the operating system	*C:*
TEMP and also TMP	Temporary directory	*C:\Users\CHRIST~1\AppData\Local\Temp*
USERDOMAIN	Domain of the user	*MOE*

Table 16.5 Selection of Some Environment Variables Available in Windows

Variable Name	Description	Example
USERNAME	Name of the user	*Christian*
USERPROFILE	Profile directory	*C:\Users\Christian*
WINDIR	Directory of the operating system	*C:\windows*

* The result differs from `System.getProperty("os.name")`, 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 ) );
```

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 code[4] (*ISO Language Code*), and country names are 2-letter abbreviations described in ISO 3166[5] (*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 Billennium* 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)
`LocalDate`	Represents a common date	Years, months, and days
`LocalTime`	Represents a common time	Hours, minutes, seconds, and nanoseconds
`LocalDateTime`	Combination of date and time	Years, months, days, hours, minutes, seconds, and nanoseconds
`Period`	Duration between two `LocalDates`	Years, months, and days
`Year`	Year only	Year
`Month`	Month only	Month
`MonthDay`	Month and day only	Month, day
`OffsetTime`	Time with time zone	Hours, minutes, seconds, nanoseconds, and zone offset
`OffsetDateTime`	Date and time with time zone as UTC offset	Year, month, day, hours, minutes, seconds, nanoseconds, and zone offsets
`ZonedDateTime`	Date and time with time zone as ID and offset	Year, month, day, hours, minutes, seconds, nanoseconds, and zone info
`Instant`	Time (continuous machine time)	Nanoseconds
`Duration`	Time interval between two `instants`	Seconds/nanoseconds

Table 16.6 All Temporal Classes from java.time

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 `NullPointerExcpetion`. 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
`now()`	Static	Returns an object with current time/current date
`of*()`	Static	Creates new objects
`from`	Static	Creates new objects from other representations
`parse*()`	Static	Creates a new object from a string representation
`format()`	Instance	Formats and returns a string
`get*()`	Instance	Returns fields of an object
`is*()`	Instance	Queries the status of an object
`with*()`	Instance	Returns an instance of the object with a changed state
`plus*()`	Instance	Returns an instance of the object with a totaled state
`minus*()`	Instance	Returns an instance of the object with a reduced state
`to*()`	Instance	Converts an object to a new type
`at*()`	Instance	Combines this object with another object
`*Into()`	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 `LocaleDate` 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` class[9] 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	log(Level.SEVERE, msg)	severe(String msg)
WARNING	log(Level.WARNING, msg)	warning(String msg)
INFO	log(Level.INFO, msg)	info(String msg)
CONFIG	log(Level.CONFIG, msg)	config(String msg)
FINE	log(Level.FINE, msg)	fine(String msg)
FINER	log(Level.FINER, msg)	finer(String msg)
FINEST	log(Level.FINEST, msg)	finest(String msg)

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

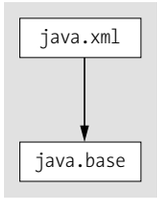


Figure 16.1 The java.xml Module Has a Dependency on the java.base Module

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

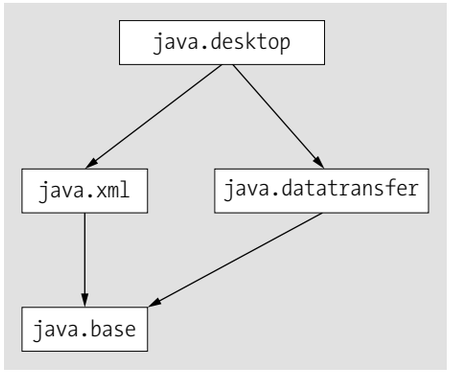


Figure 16.2 Dependencies of the java.desktop Module

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.

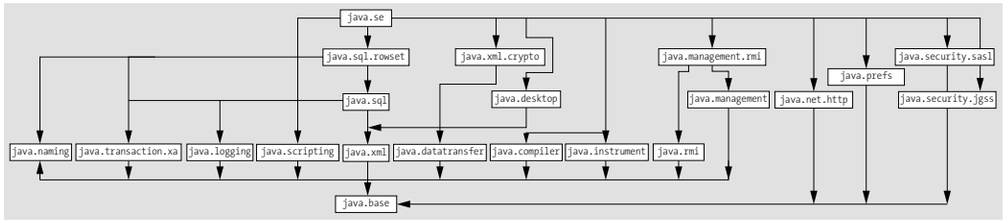


Figure 16.3 Dependencies of the java.se Module

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-object[2]—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`.

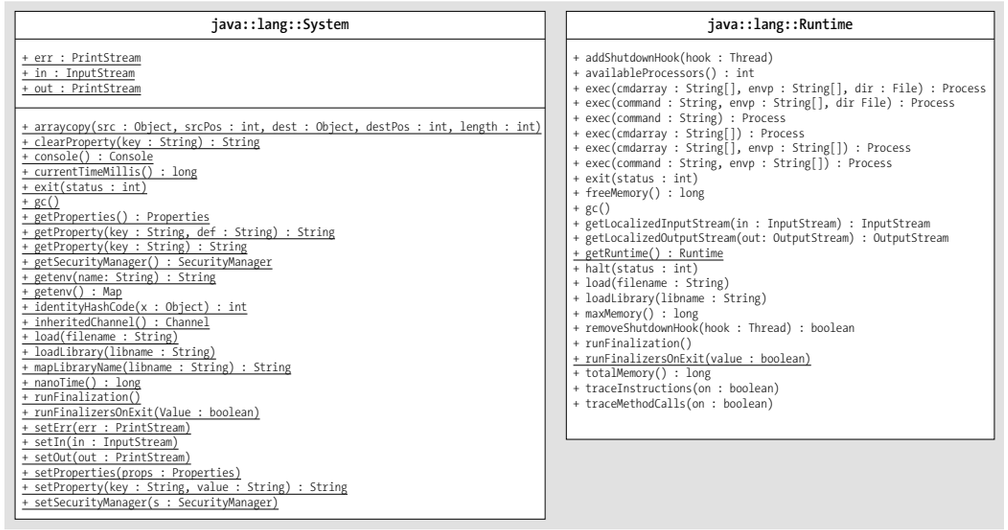


Figure 16.4 Members of the System and Runtime Classes

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.

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.

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=fa1se` 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 newline 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	*MOE*
HOMEDRIVE	Drive of the user directory	*C:*
HOMEPATH	Path of the user directory	*\Users\Christian*
OS	Name of the operating system*	*Windows_NT*
PATH	Search path	*C:\windows\SYSTEM32; C:\windows ...*
PATHEXT	File extensions that represent executable programs	*.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC*
SYSTEMDRIVE	Drive of the operating system	*C:*
TEMP and also TMP	Temporary directory	*C:\Users\CHRIST~1\AppData\Local\Temp*
USERDOMAIN	Domain of the user	*MOE*

Table 16.5 Selection of Some Environment Variables Available in Windows

Variable Name	Description	Example
USERNAME	Name of the user	*Christian*
USERPROFILE	Profile directory	*C:\Users\Christian*
WINDIR	Directory of the operating system	*C:\windows*

* The result differs from `System.getProperty("os.name")`, 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 ) );
```

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 code[4] (*ISO Language Code*), and country names are 2-letter abbreviations described in ISO 3166[5] (*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 Billennium* 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)
`LocalDate`	Represents a common date	Years, months, and days
`LocalTime`	Represents a common time	Hours, minutes, seconds, and nanoseconds
`LocalDateTime`	Combination of date and time	Years, months, days, hours, minutes, seconds, and nanoseconds
`Period`	Duration between two `LocalDates`	Years, months, and days
`Year`	Year only	Year
`Month`	Month only	Month
`MonthDay`	Month and day only	Month, day
`OffsetTime`	Time with time zone	Hours, minutes, seconds, nanoseconds, and zone offset
`OffsetDateTime`	Date and time with time zone as UTC offset	Year, month, day, hours, minutes, seconds, nanoseconds, and zone offsets
`ZonedDateTime`	Date and time with time zone as ID and offset	Year, month, day, hours, minutes, seconds, nanoseconds, and zone info
`Instant`	Time (continuous machine time)	Nanoseconds
`Duration`	Time interval between two `instants`	Seconds/nanoseconds

Table 16.6 All Temporal Classes from java.time

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 `NullPointerExcpetion`. 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
`now()`	Static	Returns an object with current time/current date
`of*()`	Static	Creates new objects
`from`	Static	Creates new objects from other representations
`parse*()`	Static	Creates a new object from a string representation
`format()`	Instance	Formats and returns a string
`get*()`	Instance	Returns fields of an object
`is*()`	Instance	Queries the status of an object
`with*()`	Instance	Returns an instance of the object with a changed state
`plus*()`	Instance	Returns an instance of the object with a totaled state
`minus*()`	Instance	Returns an instance of the object with a reduced state
`to*()`	Instance	Converts an object to a new type
`at*()`	Instance	Combines this object with another object
`*Into()`	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 `LocaleDate` 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` class[9] 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	log(Level.SEVERE, msg)	severe(String msg)
WARNING	log(Level.WARNING, msg)	warning(String msg)
INFO	log(Level.INFO, msg)	info(String msg)
CONFIG	log(Level.CONFIG, msg)	config(String msg)
FINE	log(Level.FINE, msg)	fine(String msg)
FINER	log(Level.FINER, msg)	finer(String msg)
FINEST	log(Level.FINEST, msg)	finest(String msg)

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

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

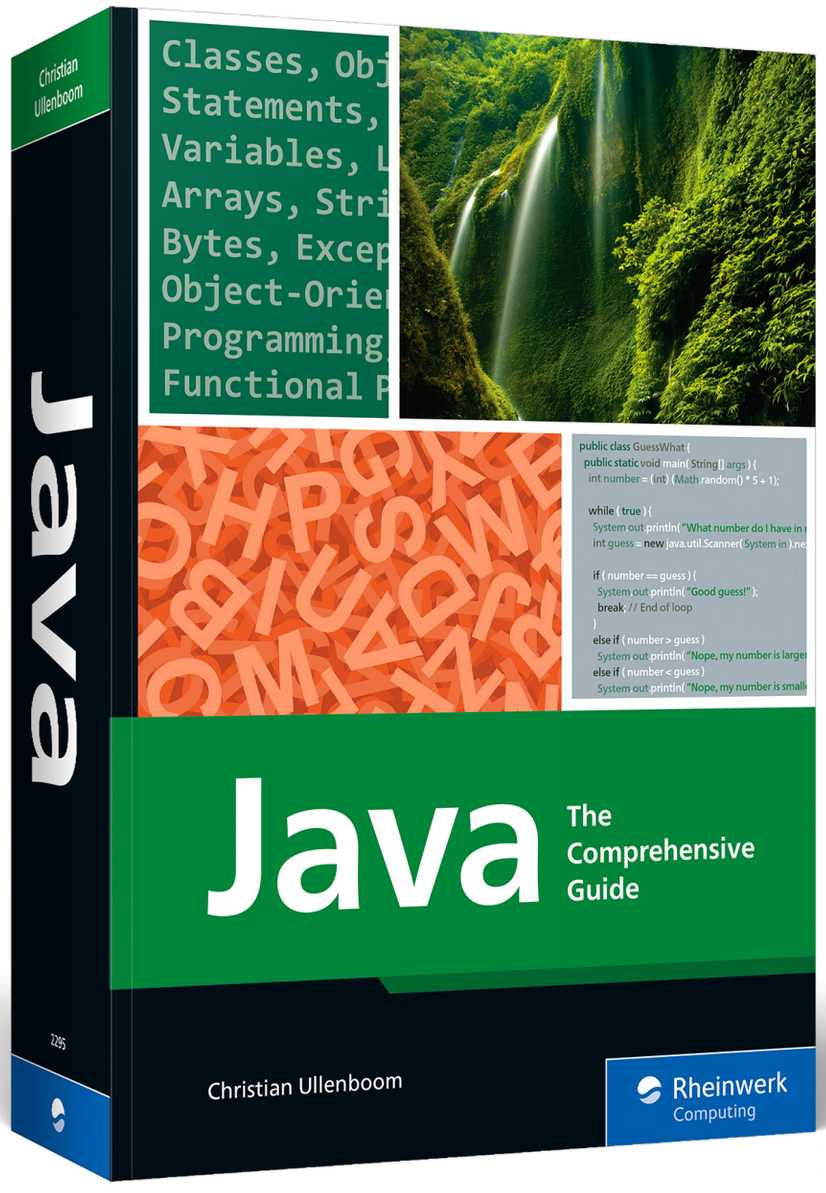
V

W

X

Y

Z



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