Learn Java in 42 minutes

Table of Contents

Java platform overview
Java technology is used to develop applications for a wide range of environments, from consumer devices to heterogeneous enterprise systems. In this section, get a high-level view of the Java platform and its components.

The Java language

Like any programming language, the Java language has its own structure, syntax rules, and programming paradigm. The Java language's programming paradigm is based on the concept of Object-oriented programming (OOP), which the language's features support.
The Java language is a C-language derivative, so its syntax rules look much like C's. For example, code blocks are modularized into methods and delimited by braces (`{` and `}`), and variables are declared before they are used.
Structurally, the Java language starts with `packages`. A package is the Java language's namespace mechanism. Within packages are classes, and within classes are methods, variables, constants, and more. You learn about the parts of the Java language in this tutorial.

The Java compiler

When you program for the Java platform, you write source code in `.java` files and then compile them. The compiler checks your code against the language's syntax rules, then writes out `bytecode` in .class files. Bytecode is a set of instructions targeted to run on a Java virtual machine (JVM). In adding this level of abstraction, the Java compiler differs from other language compilers, which write out instructions suitable for the CPU chipset the program will run on.

The JVM

At runtime, the JVM reads and interprets `.class` files and executes the program's instructions on the native hardware platform for which the JVM was written. The JVM interprets the bytecode just as a CPU would interpret assembly-language instructions. The difference is that the JVM is a piece of software written specifically for a particular platform. The JVM is the heart of the Java language's "write-once, run-anywhere" principle. Your code can run on any chipset for which a suitable JVM implementation is available. JVMs are available for major platforms like Linux, MacOS and Windows, and subsets of the Java language have been implemented in JVMs for mobile phones and hobbyist chips.

The garbage collector

Rather than forcing you to keep up with memory allocation (or use a third-party library to do so), the Java platform provides memory management out of the box. When your Java application creates an object instance at runtime, the JVM automatically allocates memory space for that object from the `heap` - a pool of memory set aside for your program to use. The Java `garbage collector` runs in the background, keeping track of which objects the application no longer needs and reclaiming memory from them. This approach to memory handling is called `implicit memory management` because it doesn't require you to write any memory-handling code. Garbage collection is one of the essential features of Java platform performance.

The Java Development Kit

When you download a Java Development Kit (JDK), you get — in addition to the compiler and other tools - a complete class library of prebuilt utilities that help you accomplish most common application-development tasks. The best way to get an idea of the scope of the JDK packages and libraries is to check out the JDK API documentation.

The Java Runtime Environment

The Java Runtime Environment (JRE; also known as the Java runtime) includes the JVM, code libraries, and components that are necessary for running programs that are written in the Java language. The JRE is available for multiple platforms. The JRE is included in the JDK.

Get to know the Java APIs

Most Java developers constantly reference the official online Java API documentation— also called the Javadoc. By default, you see three panes in the Javadoc. The top-left pane shows all of the packages in the API, and the bottom-left pane shows the classes in each package. The main pane (to the right) shows details for the currently selected package or class. For example, if you click the java.util package in the top-left pane and then click the ArrayList class listed below it, you see details about ArrayList in the right pane, including a description of what it does, how to use it, and its methods.

Compiling and running Java programs

In Java, every source file usually contains exactly one class. The file must have the same name as the class; a class named `Person` would be stored in the source file `Person.java`. This source file can then be compiled using the javac compiler:

```
% javac Person.java
```

The output of the compiler is a file with the same name as the source file, but with the extension *.class* instead of *.java* (i.e., `Person.class` in the above example). That class file contains the byte code mentioned earlier, so it cannot be executed right away. Instead it is executed using the JVM (byte code interpreter) as follows:

```
% java Person
```

This command loads the `Person` class and executes its main method (that is, starts the program). If the `Person` class in turn uses other classes, these are loaded automatically when needed. Since every class should be in its own file, several files can need to be recompiled at the same time. The `javac compiler` has a special option `-depend` to compile all files that depend on a particular file. The command

```
% javac -depend Person.java
```

will compile not only `Person.java`, but also all changed files it depends upon.

Simple declarations and expressions

This section shows how to declare and use variables of the simple types, such as integers or Booleans.

Simple declarations

Java supports the usual set of simple types, such as integer, boolean, and real variables. Here are a few of the most common ones:

```
int m, n;          // Two integer variables
double x, y;       // Two real coordinates
boolean b;         // Either 'true' or 'false'
char ch;           // A character, such as 'P' or '@'
```

Numeric expressions and assignments

Numeric expressions are written in much the same way as in other languages.

```
n = 3 * (5 + 2);
x = y / 3.141592653;
n = m % 8;              // Modulo, i.e. n is now (m mod 8)
b = true;
ch = 'x';
```

Another symbol == is used to compare two values to each other. If you try to compare two values using = you will get an error.

It is possible to assign a variable an initial value directly when declaring it.

```
double f = 0.57;
boolean flag = true;
```

Pitfall: differences between integer and real division The Java division operator / can actually mean two different things: real division for real numbers, and integer division for integers. Usually this is not a problem, but it can occasionally lead to some surprising results:

```
double f; f = 1 / 3; // f is now 0.0
f = 1.0 / 3.0; // f is now 0.33333333...
```

In the first case an integer division is performed, giving an integer result (0). To get the result 0.33333, the 1 and 3 are expressed as real values (1.0 and 3.0), which means the division becomes a real division.

Type conversion (casting)

In some languages it is possible to assign, for instance, a real value to an integer variable. The value is then automatically converted (in this case, rounded) to the right type. Java does not perform all such conversions automatically. Instead the programmer must indicate where the conversions must be made by writing the desired type in parentheses before the expression. In Java, such a conversion is called a cast.
Example:

```
double radians;
int degrees;
degrees = radians * 180 / 3.141592653; // Error
degrees = (int) (radians * 180 / 3.141592653); // OK
```

It is, however, possible to assign an integer value to a real variable without casting. In general, no cast is necessary as long as the conversion can be made without any loss of information.

Data types

```
private String name;
private int age;
```

A `dataType` can be either a primitive type or a reference to another object. For example, notice that `age` is an `int` (a primitive type) and that `name` is a `String` (an object). The JDK comes packed full of useful classes like `java.lang.String`, and those in the `java.lang` package do not need to be imported (a shorthand courtesy of the Java compiler). But whether the `dataType` is a JDK class such as `String` or a user-defined class, the syntax is essentially the same.

Primitive data types

The eight primitive data types you're likely to see on a regular basis, including the default values that primitives take on if you do not explicitly initialize a member variable's value:

Type	**Size**	**Default value**	**Range of values**
boolean	n/a	false	true or false
byte	8 bits	0	-128 to 127
char	16 bits	(unsigned)	\u0000' \u0000' to \uffff' or 0 to 65535
short	16 bits	0	-32768 to 32767
int	32 bits	0	-2147483648 to 2147483647
long	64 bits	0	-9223372036854775808 to 9223372036854775807
float	32 bits	0.0	-9223372036854775808 to 9223372036854775807
double	64 bits	0.0	4.9e-324 to 1.7976931348623157e+308

Boxing and unboxing

Every primitive type in the Java language has a JDK counterpart class, as shown:

Primitive	JDK counterpart
boolean	java.lang.Boolean
byte	java.lang.Byte
char	java.lang.Character
short	java.lang.Short
int	java.lang.Integer
long	java.lang.Long
float	java.lang.Float
double	java.lang.Double

Each JDK class provides methods to parse and convert from its internal representation to a corresponding primitive type. For example, this code converts the decimal value 238 to an `Integer`:

```
int value = 238;
Integer boxedValue = Integer.valueOf(value);
```

This technique is known as `boxing`, because you're putting the primitive into a wrapper, or box.

Similarly, to convert the Integer representation back to its int counterpart, you `unbox` it:

```
Integer boxedValue = Integer.valueOf(238);
int intValue = boxedValue.intValue();
```

Autoboxing and auto-unboxing

Strictly speaking, you don't need to box and unbox primitives explicitly. Instead, you can use the Java language's autoboxing and auto-unboxing features:

```
int intValue = 238;
Integer boxedValue = intValue;
intValue = boxedValue;
```

I recommend that you avoid autoboxing and auto-unboxing, however, because it can lead to code-readability issues. The code in the boxing and unboxing snippets is more obvious, and thus more readable, than the autoboxed code; I believe that's worth the extra effort.

Parsing and converting boxed types

You've seen how to obtain a boxed type, but what about parsing a numeric `String` that you suspect has a boxed type into its correct box? The JDK wrapper classes have methods for that, too:

```
String characterNumeric = "238";
Integer convertedValue = Integer.parseInt(characterNumeric);
```

You can also convert the contents of a JDK wrapper type to a String:

```
Integer boxedValue = Integer.valueOf(238);
String characterNumeric = boxedValue.toString();
```

Note that when you use the concatenation operator in a `String` expression, the primitive type is autoboxed, and wrapper types automatically have `toString()` invoked on them.

Statements

Java statements are written in much the same way as in other languages. Just like in C++ or JavaScript, statements can be grouped together in blocks using `{` and `}` (corresponding to begin and end in these languages).

If statements and boolean expressions

A simple if statement is written as follows:

```
if (n == 3) x = 3.2;
```

Note:

- There is no `then` keyword
- The condition must be of boolean type and written within parentheses
- Comparison is made using ==

There are of course a number of other comparison operators, such as <, >, <=, >=, ==, `!=` and so on.

```
if (x != 0) y = 3.0 / x; // Executed when x is non-zero
else y = 1;              // Executed when x is zero
```

Pitfall: semicolons and `else` statements Note that there should be a semicolon before the `else` keyword in the example above. However, when one uses braces (`{` and `}`) to form a block of statements, the right brace should **NOT** be followed by a semicolon. (In fact, a right brace is never followed by a semicolon in Java.)

```
if (x != 0) {
      y = 3.0 / x; x = x + 1;
} else // <--- Note: no semicolon
      y = 1;
```

It is common practice to always include the braces, even if they only contain a single statement. This avoids forgetting them whenever another statement is added.

More about boolean expressions

For boolean expressions, one needs to use logical operators corresponding to `and`, `or`, and `not`. In Java, they are written as follows:

and	&&
or	||
not	!

For example:

```
int x, y;
boolean b;
if ((x <= 9 || y > 3) && !b) {
      b = true;
}
```

While and for statements

```
// End when the term is less than 0.00001
double sum = 0.0;
double term = 1.0;
int k = 1;
while (term >= 0.00001) {
      sum = sum + term;
      term = term / k;
      k++; // Shortcut for k = k + 1
}
```

As the example shows, there is nothing special about Java's while statement.

`for( Start ; End ; In-/Decrement )` The for statement is quite general and can be used in some very advanced ways. However, the most common use is to repeat some statement a known number of times:

```
// Calculate 1 + (1/2) + (1/3) + ... + (1/100)
int i;
double sum = 0.0;
for (i = 1; i <= 100; i++) {
      sum = sum + 1.0 / i;
}
```

As indicated in these examples, the statement `i++` is a shortcut for `i = i + 1`. Actually, there are at least four ways to increment an integer variable1: `i = i + 1; i++; ++i; i += 1;` As long as these statements are not used as parts of a larger expression, they mean exactly the same thing. There corresponding operators for decrementing variables are `--` and `-=`.
But there are additonal ways to use a for (some of them will be more understandable later on)

```
// dynamic List, with the collection datatype vector
Vector vector = new Vector();

// wir hängen ein paar Namen an die Liste an
vector.add("Alpha");
vector.add("Beta");
vector.add("Charlie");

// Vector.iterator() // Returns a list iterator over the elements in this
list (in proper sequence).

for( Iterator it = vector.iterator(); it.hasNext(); )
{
      it.next() //gets the next value out of vector
      System.out.println(it.next());
}
```

Or the ForEach

```
// Array of integer with 9 values
int[] array = new int[]{4, 8, 4, 2, 2, 1, 1, 5, 9};

// ForEach Loop
for( int k: array )
{
      System.out.println("k =" + k);
}
```

Classes and objects

The Java language is (mostly) object oriented. This section is an introduction to OOP language concepts, using structured programming as a point of contrast.
As already mentioned, one file normally contains one class!

What is an object?

Object-oriented languages follow a different programming pattern from structured programming languages like C and COBOL. The structured-programming paradigm is highly data oriented: You have data structures, and then program instructions act on that data. Object-oriented languages such as the Java language combine data and program instructions into `objects`.
An object is a self-contained entity that contains attributes and behavior, and nothing more. Instead of having a data structure with fields (attributes) and passing that structure around to all of the program logic that acts on it (behavior), in an object-oriented language, data and program logic are combined. This combination can occur at vastly different levels of granularity, from fine-grained objects such as a `Number`, to coarse-grained objects, such as a `FundsTransfer` service in a large banking application.

Structure of a class file

```
package packageName;
import ClassNameToImport;
accessSpecifier class ClassName {
  accessSpecifier dataType variableName [= initialValue];
  accessSpecifier ClassName([argumentList]) {
    constructorStatement(s)
  }
  accessSpecifier returnType methodName ([argumentList]) {
    methodStatement(s)
  }
}
```

Classes

To define an object in the Java language, you must declare a class.

A class's `accessSpecifier` can have several values, but usually it's `public`. You'll look at other values of `accessSpecifier` soon.

You can name classes pretty much however you want, but the convention is to use camel case: Start with an uppercase letter, put the first letter of each concatenated word in uppercase, and make all the other letters lowercase. Class names should contain only letters and numbers. Sticking to these guidelines ensures that your code is more accessible to other developers who are following the same conventions.

A class declaration typically contains a set of `attributes` (sometimes called `instance variables`) and `methods` (in other languages `functions`).
Attributes are declared almost as usual:

```
class Turtle {
      private boolean penDown;
      protected int x, y;
      // Declare some more stuff
}
```

The `private` and `protected` keywords require some explanation. The `private` declaration means that those attributes cannot be accessed outside of the class. In general, attributes should be kept `private` to prevent other classes from accessing them directly. There are two other related keywords: `public` and `protected`. The `public` keyword is used to declare that something can be accessed from other classes. The `protected` keyword specifies that something can be accessed from within the class and all its subclasses, but not from the outside.

Variables / Attributes

accessSpecifier dataType variableName [= initialValue];

The values of a class's variables distinguish each instance of that class and define its state. These values are often referred to as `instance variables`. A variable has:

- An `accessSpecifier`
- A `dataType`
- A `variableName`
- Optionally, an `initialValue`

The possible *accessSpecifier* values are:

- **public**: Any object in any package can see the variable.
- **protected**: Any object defined in the same package, or a subclass (defined in any package), can see the variable.
- **No specifier** (also called `friendly` or `package private access`): Only objects whose classes are defined in the same package can see the variable.
- **private**: Only the class containing the variable can see it.

Public variables

It's never a good idea to use public variables, but in extremely rare cases it can be necessary, so the option exists. The Java platform doesn't constrain your use cases, so it's up to you to be disciplined about using good coding conventions, even if tempted to do otherwise.

A variable's `dataType` depends on what the variable is - it might be a primitive type or another class type (more about this later).

The variableName is up to you, but by convention, variable names use the camel case convention, except that they begin with a lowercase letter. (This style is sometimes called lower camel case.)

Don't worry about the `initialValue` for now; just know that you can initialize an instance variable when you declare it. (Otherwise, the compiler generates a default for you that is set when the class is instantiated.)
Example:

```
package com.telekom.learnjava;

public class Person {
   private String name;
   private int age;
   private int height;
   private int weight;
   private String eyeColor;
   private String gender;
}
```

This

Within an instance method or a constructor, `this` is a reference to the *current object* - the object whose method or constructor is being called. You can refer to any member of the current object from within an instance method or a constructor by using `this`.
The most common reason for using the `this` keyword is because a field is shadowed by a method or constructor parameter.
For example

```
public class Point {
    public int x = 0;
    public int y = 0;

    //constructor
    public Point(int a, int b) {
        x = a;
        y = b;
    }
}
```

but it should have been written like this:

```
public class Point {
    public int x = 0;
    public int y = 0;

    //constructor
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Each argument to the constructor shadows one of the object's fields — inside the constructor `x` is a local copy of the constructor's first argument. To refer to the `Point` field `x`, the constructor must use `this.x`.

Methods

accessSpecifier returnType methodName([argumentList]) {
methodStatement(s)
}

In Java, functions and procedures are called `methods`. A class's methods define its behavior. Methods are declared as follows:

```
package com.telekom.learnjava;

public class Person {
    private String name;
    private int age;
    private String gender;

    public Person() {
        // Nothing to do...
    }

    public Person(String name, int age, String gender) {
        this.name = name;
        this.age = age;
        this.gender = gender;
    }

    public void removeGender() {
        this.gender = "unknown";
    }
}
```

Before the method's name, a type is written to indicate the method's return type. The `removeGender` method does not return a value (i.e., it is a `procedure`, not a `function`). For this reason, it is declared as `void` (meaning `nothing`).

Objects

The `new` operator is used to create objects in much the same way as in other languages. If we assume the `Person` class requires two integer parameters (`age` and `height`) a Person object can be created as follows:

```
Person p;
p = new Person(42, 180);
```

The first line is a declaration of a reference variable to a Person object. The second line creates a new Person object and sets the `p` variable to refer to it. There's nothing strange about calling methods in objects, as the following examples show.

```
int age = p.getAge();
```

Parameters to classes: constructors

In the example above, the Turtle class was assumed to take two parameters. This must of course be specified in the class in some way, and in Java this is done in a special method called the `constructor`. The constructor is automatically called when an object is created, and the parameters of the constructor match those given when an object is created. The constructor is written just like any ordinary method but with the same name as the class, and no return type (not even void). The Turtle constructor could, for instance, look like this:

```
public Person(int age, int height) {
      this.age = age;
      this.height = height;
}
```

The main method

In Java, statements can only be written within methods in classes. This means that there must be some method which is called by the system when the program starts executing. This method is called `main` and must be declared in the class which is started from the command line (for example, in the `Person` class if one runs `java Person`). A main method usually creates a few objects and does some small work to get things going. For Person a simple main method may look as follows:

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

There are two new things about main, which can both safely be ignored for now. The first is the `static` keyword. It means that when the main method is called, it is not associated with an object, but with the class. (This implies that the method cannot access any attributes.) The other new thing is the parameter named `args`. If the Java interpreter is given any more information than the class name, this data is passed on to the main method in this parameter.

Inheritance

A `parent object` is one that serves as the structural basis for deriving more-complex `child objects`. A child object looks like its parent but is more specialized. You can reuse the common attributes and behavior of the parent object, adding to its child objects attributes and behavior that differ.
To declare a parent class (which also can be a child of a parent class):

```
class Turtle {
    String color;
    String name;
    int armor;
    double speed;
}
```

To declare a `subclass(child)` of another class, use the extends keyword in the class declaration:

```
class NinjaTurtle extends Turtle {
      Weapon[] weapons;
}
```

The object combined object now has the attributes: `color, name, armor, speed` and `weapons`!

If the superclass constructor has any parameters, it must be called first using the keyword `super`. The constructor for NinjaTurtle might look like this:
```
public NinjaTurtle(String color, String name) {
      super(color, name); // Call superclass constructor
      // ... do some more initialization stuff...
}
```

Virtual methods

In Java, all methods are virtual. A method in a subclass automatically overrides any method with the same name and parameters in any superclass. It is possible to declare `abstract` methods. Such methods are really just declarations without any associated implementation, meaning that the method must be implemented in some subclass. Consider, for example, a class for graphic figures: `Figure`. That class is then specialized into `Circle`, `Square` and so on. All figures can be drawn, but the implementation is left to subclasses. The draw method in Figure could be declared as follows:
```
public abstract void draw();
```
A class with one or more abstract methods is itself called abstract, and must be declared as such by writing abstract class instead of class. It is not possible to create objects from abstract classes.

Interfaces

An `interface` can be used to specify that a class has to provide a certain set of methods. This can be useful in a number of situations and is perhaps best shown with an example as follows. Programs with graphical user interfaces often need to be informed whenever the mouse is clicked. Usually the program has some method which should be automatically called by the system whenever the user clicks the mouse. Java provides a very flexible way of specifying an object and a method to call in such situations. Suppose the window system declares an interface, written as follows:
```
interface MouseListener {
      void processMouseClick(int x, int y);
}
```
This declaration essentially says that if an object should be used to handle mouse clicks, its class should contain a `processMouseClick` method with two integer parameters. A class can then be declared to implement that interface:
```
class SomeClass extends SomeOtherClass implements MouseListener {
      // ...declarations...
      public void processMouseClick(int x, int y) {
            // Do something sensible here
      }
}
```
Finally, the window system should have some method to register `MouseListener` objects to inform whenever a mouse is clicked. Such a method might look like as follows:
```
class WindowSystem {
      public void addMouseListener(MouseListener m) {
            // Insert m into some clever data structure
      }
      // ... and loads of more stuff...
}
```
Note that the type of the parameter `m` is not a class, but an interface. In other words, it does not matter of which class the listener is, as long as that class implements the MouseListener interface. This turns out to be a quite flexible technique in practice, since the different components need very little information about each other.

Using packages

With the Java language, you can choose the names for your classes, such as `Account`, `Person`, or `NinjaTurtle`. At times, you might end up using the same name to express two slightly different concepts. This situation, called a *name collision*, happens frequently. The Java language uses *packages* to resolve these conflicts.
A Java package is a mechanism for providing a *namespace* - an area inside of which names are unique, but outside of which they might not be. To identify a construct uniquely, you must fully qualify it by including its namespace.
Packages also give you a nice way to build more-complex applications with discrete units of functionality.

To define a package, use the package keyword followed by a legal package name, ending with a semicolon. Often package names follow this *de facto* standard scheme:

```
package  orgType.orgName.appName.compName;
```

This package definition breaks down as:

`orgType` is the organization type, such as com, org, or net.

`orgName` is the name of the organization's domain, such as telekom, oracle, or ibm.

`appName` is the name of the application, abbreviated.
`compName` is the name of the component.

Import statements

An import statement tells the Java compiler where to find classes that you reference inside of your code. Any nontrivial class uses other classes for some functionality, and the import statement is how you tell the Java compiler about them.
An import statement usually looks like this:

```
import ClassNameToImport;
```

You specify the import keyword, followed by the class that you want to import, followed by a semicolon. The class name should be **fully qualified**, meaning that it should include its package.

To import all classes within a package, you can put .* after the package name. For example, this statement imports every class in the `com.telekom` package:

```
import com.telekom.*;
```

Importing an entire package can make your code less readable, however, so I recommend that you import only the classes that you need, using their fully qualified names.

Arrays

A Java array is similar to an object in some ways. It is for example accessed using reference variables. Array references can be declared as follows:

```
int[] someInts; // Array of integer
array Person[] personArray; // An array of references to Person
```

Since these variables are only references to arrays, the array sizes are not given in the declarations, but when the arrays are actually created.

```
someInts = new int[30];
personArray = new Person[100];
```

The array elements can then be used as any simple scalar variables. (Note that indices always start at 0 and end at the size minus one, so the elements of the someInts array have the indices 0 to 29.)

```
int i; for (i = 0; i < someInts.length; i = i + 1) {
      someInts[i] = i * i;
}
```

The expression someInts.length means in the length of the vector, 30 in this case.

Lists

A `List` is an ordered collection, also known as a `sequence`. Because a `List` is ordered, you have complete control over where in the `List` items go. A Java `List` collection can only hold objects (not primitive types like `int`), and it defines a strict contract about how it behaves.

List is an interface, so you can't instantiate it directly. You'll work here with its most commonly used implementation, `ArrayList`. You can make the declaration in the explicit syntax:

```
List<String> listOfStrings = new ArrayList<String>(); //creates a list of Strings
```

Note that I assigned the `ArrayList` object to a variable of type `List`. With Java programming, you can assign a variable of one type to another, provided the variable being assigned to is a superclass or interface implemented by the variable being assigned from. In a later section, you'll look more at the rules governing these types of variable assignments.

Formal type <Object>

The `<Object>` in the preceding code snippet is called the *formal type*. `<Object>` tells the compiler that this `List` contains a collection of type `Object`, which means you can pretty much put whatever you like in the `List`.

If you want to tighten up the constraints on what can or cannot go into the List, you can define the formal type differently:

```
List<Person> listOfPersons = new ArrayList<Person>();
```

Now your `List` can only hold `Person` instances.

Using lists

Using `Lists` — like using Java collections in general — is super easy. Here are some of the things you can do with `Lists`:

- Put something in the `List`.
- Ask the `List` how big it currently is.
- Get something out of the List.

To put something in a List, call the `add()` method:

```
List<Integer> listOfIntegers = new ArrayList<>();
listOfIntegers.add(Integer.valueOf(238));
l.info("Current List size: " + listOfIntegers.size());
```

Sets

A `Set` is a collections construct that by definition contains unique elements — that is, no duplicates. Whereas a `List` can contain the same object maybe hundreds of times, a `Set` can contain a particular instance only once. A Java `Set` collection can only hold objects, and it defines a strict contract about how it behaves.

Because Set is an interface, you can't instantiate it directly. One of my favorite implementations is `HashSet`, which is easy to use and similar to `List`.

Here are some things you do with a `Set`:

- Put something in the `Set`.
- Ask the `Set` how big it currently is.
- Get something out of the Set.

A `Set`'s distinguishing attribute is that it guarantees uniqueness among its elements but doesn't care about the order of the elements. Consider the following code:

```
Set<Integer> setOfIntegers = new HashSet<Integer>();
setOfIntegers.add(Integer.valueOf(10));
setOfIntegers.add(Integer.valueOf(11));
setOfIntegers.add(Integer.valueOf(10));
for (Integer i : setOfIntegers) {
  l.info("Integer value is: " + i);
}
```

You might expect that the `Set` would have three elements in it, but it only has two because the `Integer` object that contains the value `10` is added only once.

Maps

A `Map` is a handy collection construct that you can use to associate one object (the `key`) with another (the *value*). As you might imagine, the key to the `Map` must be unique, and it's used to retrieve the value at a later time. A Java `Map` collection can only hold objects, and it defines a strict contract about how it behaves.

Because `Map` is an interface, you can't instantiate it directly. One of my favorite implementations is `HashMap`.

Things you do with `Maps` include:

- Put something in the `Map`.
- Get something out of the `Map`.
- Get a `Set` of keys to the `Map`— for iterating over it.

To put something into a `Map`, you need to have an object that represents its key and an object that represents its value:

```
public Map<String, Integer> createMapOfIntegers() {
  Map<String, Integer> mapOfIntegers = new HashMap<>();
  mapOfIntegers.put("1", Integer.valueOf(1));
  mapOfIntegers.put("2", Integer.valueOf(2));
  mapOfIntegers.put("3", Integer.valueOf(3));
  //...
  mapOfIntegers.put("168", Integer.valueOf(168));
return mapOfIntegers;
}
```

Exceptions

Many things can go wrong during the execution of a program. These run-time errors can be divided into two

- Faults introduced by the programmer, such as division by zero or calling a method with a null referen
- Things out of the program's control, such as a user entering a garbage on the keyboard when the prog

The latter category is the one that programmers usually take care of. The traditional way of handling these err some method which returns a value to indicate whether whings went well or not. A method to read a positive instance, look like this:

```
public int getNatural() { ... }
```

Suppose the special value -1 is used to indicate that an invalid number was entered by the user. The code that check the return value with an if statement. If that code is part of some method, that method may in turn have things went wrong. This kind of programming can easily turn into a lot of if statements and special return valu actually do something.

Catching exceptions

Using Java exceptions, the method above could be declared as follows:

```
public int getNatural() throws IOException { ... }
```

This method is said to throw an exception (more specifically, an `IOException`) when something else than a n keyboard. The code that calls the method could look like this:

```
int m, n;
try {
      n = getNatural(); //reads the garbage from the user input
      m = n * 2; // If an exception is thrown, this is not executed
} catch (IOException e) {
      // The user entered something wrong. Use 1 as default.
      n = 1;
      m = 2;
}
```
The statement(s) within the `try` clause are executed as usual, but whenever an exception occurs, the `try` clau
within the corresponding `catch` clause are executed.
The execution then continues after the `try/catch` clauses. The try clause can contain many statements that `t`
several different `catch` clauses. This means that the error handling is separated from the code that actually do
reducing the complexity of the error handling code. If a method does not handle an exception (for instance, if
without any try/catch clauses), the exception must be passed on to the calling method.
This is done using a `throws` declaration as indicated above:
```
public void doStuff() throws IOException {
      int n = getNatural(); // May throw an exception
      // Clever calculations using n
      ...
}
```
The method that calls `doStuff()` must in turn either catch the exception or pass it on. If the exception is **not**
it on), the execution is aborted with an error message.

Throwing exceptions

It is of course also possible to throw exceptions yourself when things go wrong. The `getNatural()` method c
pseudo-code):
```
public int getNatural() throws IOException {
      char ch;
      while (more input) {
            ch = (read character);
            if (ch < '0' || ch > '9') {
                  throw new IOException("bad natural number");
            }
            ...
      }
      ...
}
```
Note the new keyword in the throw statement above. This reveals that the exception is actually an object whi
particular, it contains a string describing the error, as indicated above.

Declaring new exceptions

Although there are a number of pre-defined exception classes in Java, you may occasionally need to declare y
done by creating a subclass of the existing Exception class. Suppose we want to throw an exception when so
Such an exception could hold information about the current temperature, as follows:
```
class OverheatedException extends Exception {
      public OverheatedException(String s, double temp) {
            super(s);
            myTemperature = temp;
      }
      public double getTemperature() {
            return myTemperature;
      }
      private double myTemperature;
}
```

Unchecked exceptions

Some exceptions do not have to be caught: the so-called unchecked exceptions which are thrown by the run-t
For instance, when an integer division by zero occurs, an ArithmeticException is thrown by the system.
This exception is normally not caught anywhere, so when it is thrown the execution of the program is aborted
most other languages). It is possible to catch these unchecked exceptions, which can occasionally be useful.

Miscellaneous

This section contains a few details about Java that might be useful to know when writing Java programs.

Comments

One kind of comments has already been shown: the line comment, which starts with `//` and extends to the end of a line.
Multi-line comments are written using `/*` and `*/`.
A special case of such multi-line comments are the documentation comments. They are written immediately before classes and methods and begin with `/**` (two asterisks) and end, as usual, with `*/` (one asterisk). Such comments are used by a special tool, javadoc, to automatically generate low-level documentation of the program.

```
/* This is a comment which
continues on to a second line */

//This is a single line comment

/**
  * Generate the cipher as a HashMap.
  * @param offset offset for the alphabet cipher
  * @return HashMap, Like: {[a]=[c], [b]=[d], [c]=[e]}
  */
public HashMap<String,String> generateCipher(int offset) {
    return new HashMap<String,String>();
};
```

Logging

Before going further into coding, you need to know how your programs tell you what they are doing.
The Java platform includes the java.util.logging package, a built-in logging mechanism for gathering program information in a readable form. Loggers are named entities that you create through a static method call to the Logger class:

```
import java.util.logging.Logger;
Logger l = Logger.getLogger(getClass().getName());
```

When calling the getLogger() method, you pass it a String, always references the name of the class and passes that to the Logger.
If you are making a Logger call inside of a static method, reference the name of the class you're inside of:

```
Logger l = Logger.getLogger(Person.class.getName());
```

Writing to the terminal

To write something to the terminal, call one of the methods `print` and `println` in the object `System.out`. They both write the argument (a String) to the terminal. The latter method, println, also ends the line. Example:

```
System.out.print(“Something “);
System.out.println(“Something else“);
```

Variable values can be printed like this:

```
int a;
a = 6 * 7;
System.out.println(“6 * 7 = “ + a);
```