developerWorks®

Intro to Java programming

Learn Java language basics and master the constructs for building and deploying real-world applications

J Steven Perry

September 14, 2016

Learn the structure, syntax, and programming paradigm of the Java platform and language. Start by mastering the essentials of object-oriented programming on the Java platform, and progress incrementally to the more-sophisticated syntax and libraries that you need in order to develop complex, real-world Java applications.

- **Skill level**: Beginner
- **Cost**: Free
- **Time commitment**: 15 - 30 minutes per unit (10 to 12 hours, total)
- **Get started**: Unit 1

This developerWorks learning path is for software developers who are new to Java™ technology and want to become productive Java programmers. Work through the 23-unit sequence, including videos and quizzes, to get up and running with object-oriented programming (OOP) and real-world application development using the Java language and platform.

Step-by-step instructions and video demos help you create and build on a simple Java object, developing it into a full application that — in the final unit — you deploy as a web app in the cloud. All along the way, you can test your newly acquired Java expertise with short quizzes and programming challenges.

To view this video, **Author's introduction** , please access the online version of the article. If this article is in the developerWorks archives, the video is no longer accessible.

Skills you'll gain

- Gain an understanding of the basics of OOP on the Java platform
- Have a fully functional Java development environment that uses the Eclipse IDE
- Become familiar with Java syntax and essential libraries
- Be ready to learn more-complex programming Java techniques
- Know where to find curated resources for bolstering your Java programming knowledge

System requirements

- A system supporting Java SE 8 with at least 2GB of memory. Java 8 is supported on Linux®, Windows®, Solaris®, and Mac OS X
- At least 200MB of disk space to install the software components and examples

You also need to install and set up a development environment consisting of JDK 8 from Oracle and the Eclipse IDE. Follow the download and installation instructions provided in the learning path.

Units in this learning path

Unit 1: Java platform overview

Learn the function of each of the Java platform's constituent components, see how the Java language is structured, and become familiar with navigating the Java API documentation.

Start Unit 1

Unit 2: Setting up your Java development environment

Install the Java Development Kit and the Eclipse IDE, become familiar with the main Eclipse components, and create a new Java project.

Start Unit 2

Unit 3: Object-oriented programming concepts and principles

Get an introduction to OOP concepts and understand the benefits of the OOP paradigm.

Start Unit 3

Unit 4: Getting started with the Java language

Recognize the reserved words in the Java language and learn the function and syntax for each construct within a Java class.

Start Unit 4

Unit 5: Your first Java class

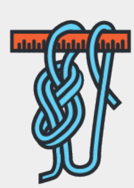
Create a package and declare a class, add variables and methods to your class, use the Eclipse code generator, and test your class with the JUnit framework from within Eclipse.

Start Unit 5

Unit 6: Adding behavior to a Java class

Learn the syntax for accessor method declarations and method calls.

Start Unit 6



Unit 7: Strings and operators

Learn how to instantiate and manipulate strings, do string concatenation and method chaining, and explore arithmetic operators.

Start Unit 7

Unit 8: Conditional operators and control statements

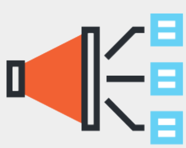
Use relational operators, conditional operators, and control statements for decision making.

Start Unit 8

Unit 9: Loops

Iterate over code or execute it repeatedly.

Start Unit 9



Unit 10: Java collections

Create and manage collections of objects.

Start Unit 10

Unit 11: Archiving Java code

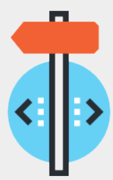
Import other developers' code and share yours.

Start Unit 11

Unit 12: Writing good Java code

Learn best practices for writing clean, easily maintainable Java code.

Start Unit 12



Unit 13: Next steps with objects

Enhance Java classes via method overloading, method overriding, and more.

Start Unit 13

Unit 14: Exceptions

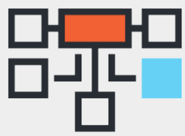
Use built-in Java platform mechanisms (checked and unchecked exceptions) to handle errors in your code.

Start Unit 14

Unit 15: Building Java applications

Create applications from collections of objects.

Start Unit 15



Unit 16: Inheritance

Enhance code reuse by deriving classes from other classes.

Start Unit 16

Unit 17: Interfaces

Learn the purpose of an interface, how to use one, and how to implement one.

Start Unit 17

Unit 18: Nested classes

Learn how to define tightly coupled classes, and consider the advantages and side effects.

Start Unit 18

Unit 19: Regular expressions

Describe and search for string patterns in your Java code.

Start Unit 19

Unit 20: Generics

Facilitate code reuse by defining classes with abstract type parameters.

Start Unit 20

Unit 21: I/O

Collect and manipulate external data in your Java programs.

Start Unit 21

Unit 22: Java serialization

Store object state in binary format for object remoting or object persistence.

Start Unit 22

Unit 23: Java in the cloud

Deploy Java applications to the IBM cloud.

Start Unit 23

Related topic

- IBM Code: Java

IBM

developerWorks®

Unit 2: Setting up your Java development environment

Install the JDK and Eclipse for your platform and get started with Eclipse

J Steven Perry

September 14, 2016

Learn the structure, syntax, and programming paradigm of the Java platform and language. Start by mastering the essentials of object-oriented programming on the Java platform, and progress incrementally to the more-sophisticated syntax and libraries that you need to develop complex, real-world Java applications.

Before you begin

This unit is part of the "Intro to Java programming" learning path. Although the concepts discussed in the individual units are standalone in nature, the hands-on component builds as you progress through the units, and we recommend that you review the prerequisites, setup, and unit details before proceeding.

Unit objectives

- Download and install the JDK and the Eclipse IDE
- Set up your Eclipse development environment
- Understand the main Eclipse components and how to use them for Java development
- Create a new Java project in Eclipse

Your development environment

The JDK includes a set of command-line tools for compiling and running your Java code, including a complete copy of the JRE. Although you can use these tools to develop your applications, an IDE gives you additional functionality along with task management and a visual interface.

In this learning path you use Eclipse, a popular open source IDE. Eclipse handles basic tasks, such as code compilation and debugging, so that you can focus on writing and testing code. In addition, you can use Eclipse to organize source code files into projects, compile and test those projects, and store project files in any number of source repositories. You need an installed JDK to use Eclipse for Java development.

Install the JDK

Follow these steps to download and install the JDK:

1. Browse to Java SE Downloads and click the **Java Platform (JDK)** box to display the download page for the latest version of the JDK.
2. Agree to the license terms for the version you want to download.
3. Choose the download that matches your operating system and chip architecture.

Windows

1. Save the file to your hard drive when prompted.
2. When the download is complete, run the install program. Install the JDK to your hard drive in an easy-to-remember location such as C:\home\Java\jdk1.8.0_92. (As in this example, it's a good idea to encode the update number in the name of the install directory that you choose.)

OS X

1. When the download is complete, double-click it to mount it.
2. Run the install program. You do not get to choose where the JDK is installed. You can run `/usr/libexec/java_home -1.8` to see the location of JDK 8 on your Mac. The path that's displayed is similar to /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home.

See JDK 8 and JRE 8 Installation for more information, including instructions for installing on Solaris or Linux.

You now have a Java environment on your computer. Next, you'll install the Eclipse IDE and create a Java project in Eclipse

Install Eclipse

Follow along with this video demo to download and install Eclipse on your system, take a quick Eclipse tour, and create a Java project.

To view this video, **Getting started with Eclipse** , please access the online version of the article. If this article is in the developerWorks archives, the video is no longer accessible.

Recap: The Eclipse development environment

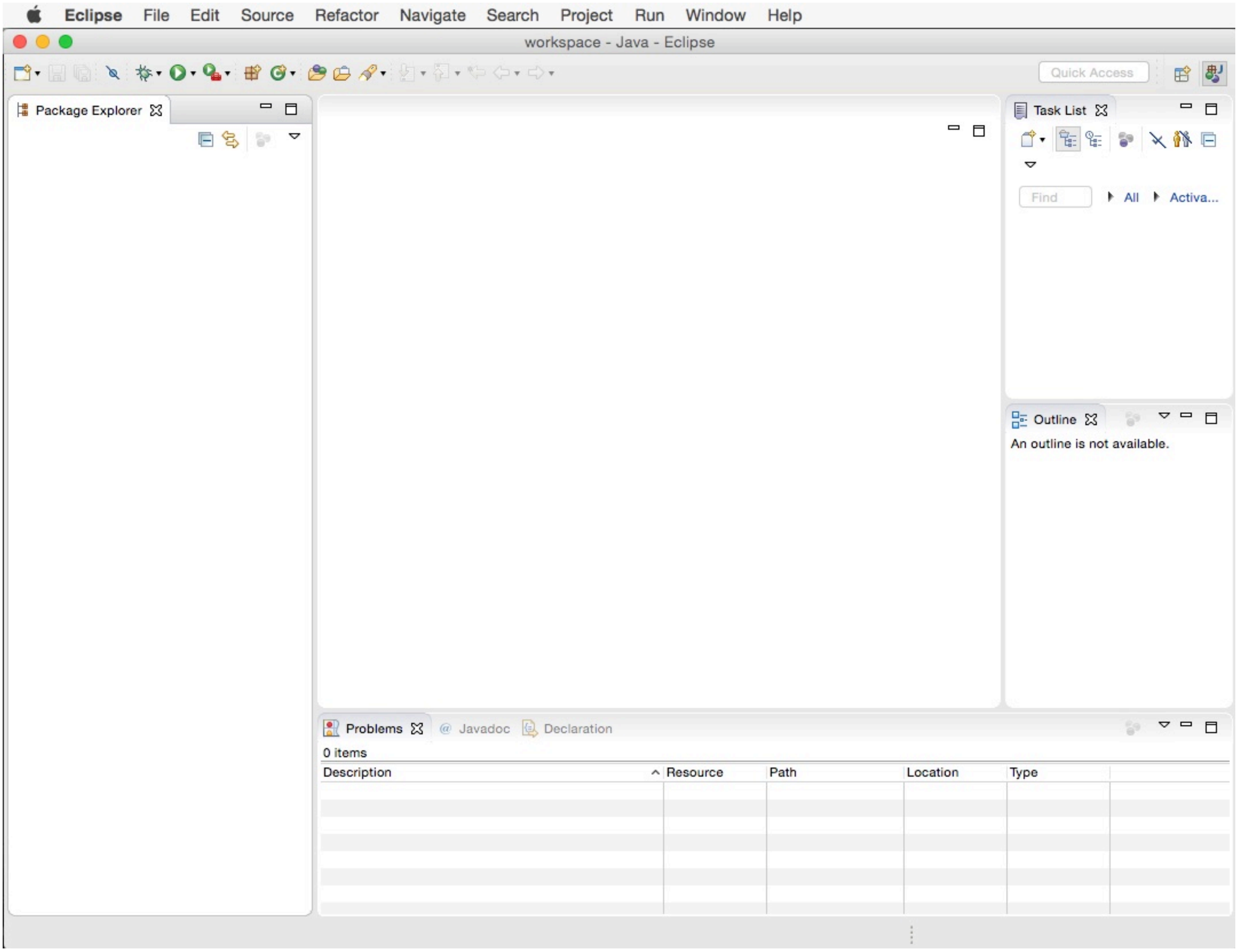
The Eclipse development environment has four main components:

- Workspace
- Projects
- Perspectives
- Views

The primary unit of organization in Eclipse is the *workspace*. A workspace contains all of your *projects*. A *perspective* is a way of looking at each project (hence the name), and within a perspective are one or more *views*.

Figure 1 shows the Java perspective, which is the default perspective for Eclipse. You see this perspective when you start Eclipse.

Figure 1. Eclipse Java perspective



The Java perspective contains the tools that you need to begin writing Java applications. Each tabbed window shown in Figure 1 is a view for the Java perspective. Package Explorer and Outline are two particularly useful views.

The Eclipse environment is highly configurable. Each view is dockable, so you can move it around in the Java perspective and place it where you want it. For now, though, stick with the default perspective and view setup.

You've now created a new Eclipse Java project and source folder. Your development environment is ready for action. However, an understanding of the OOP paradigm — covered in the next unit — is essential before you start coding in Java.

For further exploration

Learning Eclipse Java IDE by Brian Gorman

IBM Code: Java journeys

Previous: Java platform overview
Next: Object-oriented programming concepts and principles

IBM

developerWorks®

Unit 3: Object-oriented programming concepts and principles

Understand the Java language's object-oriented paradigm

J Steven Perry

September 14, 2016

Learn the structure, syntax, and programming paradigm of the Java platform and language. Start by mastering the essentials of object-oriented programming on the Java platform, and progress incrementally to the more-sophisticated syntax and libraries that you need to develop complex, real-world Java applications.

Before you begin

This unit is part of the "Intro to Java programming" learning path. Although the concepts discussed in the individual units are standalone in nature, the hands-on component builds as you progress through the units, and I recommend that you review the prerequisites, setup, and unit details before proceeding.

Unit objectives

- Grasp how the object-oriented paradigm differs from the structured-programming paradigm
- Know the key characteristics of an object
- Understand the benefits that stem from the defining principles of object-oriented programming (OOP)

What is an object?

The Java language is (mostly) object oriented. This unit gives you a short introduction to OOP language concepts and principles, using structured programming as a point of contrast.

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.

Parent and child objects

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. With the object-oriented paradigm, you can reuse the common attributes and behavior of the parent object, adding to its child objects attributes and behavior that differ.

Object communication and coordination

Objects talk to other objects by sending messages (*method calls*, in Java parlance). Furthermore, in an object-oriented application, program code coordinates the activities among objects to perform tasks within the context of the specific application domain.

Object summary

A well-written object:

- Has well-defined boundaries
- Performs a finite set of activities
- Knows only about its data and any other objects that it needs to accomplish its activities

In essence, an object is a discrete entity that has only the necessary dependencies on other objects to perform its tasks.

It's time to see what a Java object looks like.

Example: A person object

My first example is based on a common application-development scenario: an individual being represented by a `Person` object.

You know from the definition of an object that an object has two primary elements: attributes and behavior. Here's how these elements apply to the `Person` object.

As a rule of thumb, think of the attributes of an object as **nouns** and behavior as **verbs**.

Attributes (nouns)

What attributes can a person have? Some common ones include:

- Name
- Age
- Height

- Weight
- Eye color
- Gender

You can probably think of more (and you can always add more attributes later), but this list is a good start.

Behavior (verbs)

An actual person can do all sorts of things, but object behaviors usually relate to application context of some kind. In a business-application context, for instance, you might want to ask your `Person` object, "What is your body mass index (BMI)?" In response, `Person` would use the values of its height and weight attributes to calculate the BMI.

More-complex logic can be hidden inside of the `Person` object, but for now, suppose that `Person` has the following behavior:

- Calculate BMI
- Print all attributes

State and string

State is an important concept in OOP. An object's state is represented at any moment in time by the values of its attributes.

In the case of `Person`, its state is defined by attributes such as name, age, height, and weight. If you wanted to present a list of several of those attributes, you might do so by using a `String` class, which you'll learn more about later.

Using the concepts of state and string together, you can say to `Person`, "Tell me all about you by giving me a listing (or `String`) of your attributes."

Principles of OOP

If you come from a structured-programming background, the OOP value proposition might not be clear yet. After all, the attributes of a person and any logic to retrieve (and convert) those values can be written in C or COBOL. The benefits of the OOP paradigm become clearer if you understand its defining principles: *encapsulation*, *inheritance*, and *polymorphism*.

Encapsulation

Recall that an object is above all discrete, or self-contained. This characteristic is the principle of *encapsulation* at work. *Hiding* is another term that's sometimes used to express the self-contained, protected nature of objects.

Regardless of terminology, what's important is that the object maintains a boundary between its state and behavior and the outside world. Like objects in the real world, objects used in computer programming have various types of relationships with different categories of objects in the applications that use them.

On the Java platform, you can use *access modifiers* (which you'll learn about later) to vary the nature of object relationships from *public* to *private*. Public access is wide open, whereas private access means the object's attributes are accessible only within the object itself.

The public/private boundary enforces the object-oriented principle of encapsulation. On the Java platform, you can vary the strength of that boundary on an object-by-object basis. Encapsulation is a powerful feature of the Java language.

Inheritance

In structured programming, it's common to copy a structure, give it a new name, and add or modify the attributes that make the new entity (such as an `Account` record) different from its original source. Over time, this approach generates a great deal of duplicated code, which can create maintenance issues.

OOP introduces the concept of *inheritance*, whereby specialized classes — without additional code — can "copy" the attributes and behavior of the source classes that they specialize. If some of those attributes or behaviors need to change, you override them. The only source code you change is the code needed for creating specialized classes. The source object is called the *parent*, and the new specialization is called the *child*— terms that you've already been introduced to.

Suppose that you're writing a human-resources application and want to use the `Person` class as the basis (also called the *super class*) for a new class called `Employee`. Being the child of `Person`, `Employee` would have all of the attributes of a `Person` class, along with additional ones, such as:

- Taxpayer identification number
- Employee number
- Salary

Inheritance makes it easy to create the new `Employee` class without needing to copy all of the `Person` code manually.

Polymorphism

Polymorphism is a harder concept to grasp than encapsulation and inheritance. In essence, polymorphism means that objects that belong to the same branch of a hierarchy, when sent the same message (that is, when told to do the same thing), can manifest that behavior differently.

Want more?
Dig into the details of polymorphism with this step-by-step recipe, which demonstrates how polymorphism works using the Java language.

To understand how polymorphism applies to a business-application context, return to the `Person` example. Remember telling `Person` to format its attributes into a `String`? Polymorphism makes it possible for `Person` to represent its attributes in various ways depending on the type of `Person` it is.

Polymorphism, one of the more complex concepts you'll encounter in OOP on the Java platform, is beyond the scope of this introductory course. You'll explore encapsulation and inheritance in more depth in subsequent units.

Not a purely object-oriented language

Two qualities differentiate the Java language from purely object-oriented languages such as Smalltalk. First, the Java language is a mixture of objects and **primitive types**. Second, with Java, you can write code that exposes the inner workings of one object to any other object that uses it.

The Java language does give you the tools necessary to follow sound OOP principles and produce sound object-oriented code. Because Java is not purely object oriented, you must exercise discipline in how you write code — the language doesn't force you to do the right thing, so you must do it yourself. You'll get tips in Unit 12, "Writing good Java code."

Test your understanding

Test your knowledge of what you've learned in this unit. Answers are below.

1. An object is a data-oriented entity, and any program that needs to can access any of that object's data.
   a. True
   b. False
   c. Sometimes
2. Which statement most accurately describes the relationship between a parent and child object?
   a. A parent object contains the exact same code as its child.
   b. A child object is unrelated to its parent.
   c. A parent object inherits attributes and behavior from its child.
   d. A child object contains code from its parent but is more specialized and can define additional attributes.
3. A well-written object can be described as:
   a. Has well-defined boundaries, does only a few things, and knows only about its own data.
   b. Is largely data oriented and open to any program that needs to access it.
   c. Has knowledge only of the other objects it needs to perform its functions.
   d. a and b
   e. a and c
   f. All of the above
4. Which statement(s) best describes an object's attributes?
   a. Attributes act like the nouns of an object.
   b. Attributes describe an object's behavior.
   c. Attributes can mean pretty much whatever you want them to mean — that's what we mean by "open source."
   d. a and c
   e. b and c
   f. All of the above.

5. Choose the option(s) that potentially contains some additional attributes of `Person`.
   a. Calculate age in weeks
   b. Number of siblings
   c. Resting pulse (in beats per minute)
   d. a and c
   e. b and c
   f. All of the above.
6. Which statement(s) best describes an object's behavior?
   a. Behavior acts like the nouns of an object.
   b. Behavior, like attributes, can be defined however we want — that's what we mean by "open source."
   c. Behavior acts like the verbs of an object.
   d. All of the above.
7. Encapsulation means "open to all" and indicates that the internal workings of an object are entirely visible to the outside world.
   a. True
   b. False
   c. Sometimes
8. Which statement(s) best describes inheritance?
   a. One object (the child) generalizes the attributes and behavior of another object (the parent), making the object more generalized.
   b. The child object inherits from its parent, adding specialized attributes and behavior.
   c. When a child object is created, a well-known best practice is for the developer to manually copy/paste all of the source code from the parent object to ensure that nothing gets broken.
   d. When a child object inherits from its parent, only the additional attributes and behavior must be coded, which is one of the benefits of OOP.
   e. a and b
   f. b and d
   g. All of the above.
9. Java is a pure object-oriented programming language like Scala or Smalltalk.
   a. True
   b. False
   c. Sometimes

Answers to quiz

1. b **2.** d **3.** e **4.** a **5.** e **6.** c **7.** b **8.** f **9.** b

For further exploration

OO Design Process: The object primer

Object-Oriented Programming Concepts

Object-oriented language basics

IBM Code: Java journeys

Previous: Setting up your Java development environment
Next: Getting started with the Java language

IBM

developerWorks®

Unit 4: Getting started with the Java language

Find out how Java objects are structured

J Steven Perry

September 14, 2016

Learn the structure, syntax, and programming paradigm of the Java platform and language. Start by mastering the essentials of object-oriented programming on the Java platform, and progress incrementally to the more-sophisticated syntax and libraries that you need to develop complex, real-world Java applications.

Before you begin

This unit is part of the "Intro to Java programming" course. Although the concepts discussed in the individual units are standalone in nature, the hands-on component builds as you progress through the units, and I recommend that you review the prerequisites, setup, and unit details before proceeding.

Unit objectives

- Recognize the reserved words in the Java language
- Understand the relationship between a Java class and a Java object
- Know the function and syntax for each construct within a Java class

OOP is all about objects. This unit covers two topics specifically related to how the Java language handles objects: reserved words and the structure of a Java class.

Reserved words

Like any programming language, the Java language designates certain words that the compiler recognizes as special. For that reason, you're not allowed to use them for naming your Java constructs. The list of reserved words (also called *keywords*) is surprisingly short:

```
abstract
assert
boolean
break
```

```
byte
case
catch
char
class
const
continue
default
do
double
else
enum
extends
final
finally
float
for
goto
if
implements
import
instanceof
int
interface
long
native
new
package
private
protected
public
return
short
static
strictfp
super
switch
synchronized
this
throw
throws
transient
try
void
volatile
while
```

You also may not use `true`, `false`, and `null` (technically, **literals** rather than keywords) to name Java constructs

One advantage of programming with an IDE is that it can use syntax coloring for reserved words, as you'll see later.

Structure of a Java class

A class is a blueprint for a discrete entity (object) that contains attributes and behavior. The class defines the object's basic structure; at runtime, your application creates an *instance* of the object. An object has a well-defined boundary and a state, and it can do things when correctly asked. Every object-oriented language has rules about how to define a class.

In the Java language, classes are defined as shown in Listing 1:

Listing 1. Class definition

```
package packageName;
import ClassNameToImport;
accessSpecifier class ClassName {
  accessSpecifier dataType variableName [= initialValue];
  accessSpecifier ClassName([argumentList]) {
    constructorStatement(s)
  }
  accessSpecifier returnType methodName ([argumentList]) {
    methodStatement(s)
  }
  // This is a comment
  /* This is a comment too */
  /* This is a
     multiline
     comment */
}
```

Note
In Listing 1 and some other code examples in this unit, square brackets indicate that the constructs within them are not required. The brackets (unlike { and }) are not part of the Java syntax.

Listing 1 contains various types of constructs, including `package` in line 1, `import` in line 2, and `class` in line 3. Those three constructs are in the list of reserved words, so they must be exactly what they are in Listing 1. The names that I've given the other constructs in Listing 1 describe the concepts that they represent.

Notice that lines 11 through 15 in Listing 1 are comment lines. In most programming languages, programmers can add comments to help document the code. Java syntax allows for both single-line and multiline comments:

```
// This is a comment
/* This is a comment too */
/* This is a
multiline
comment */
```

A single-line comment must be contained on one line, although you can use adjacent single-line comments to form a block. A multiline comment begins with `/*`, must be terminated with `*/`, and can span any number of lines.

Next, I'll walk you through the constructs in Listing 1 in detail, starting with `package`.

Packaging classes

With the Java language, you can choose the names for your classes, such as `Account`, `Person`, or `LizardMan`. 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 `makotojava`, `oracle`, or `ibm`.
- *`appName`* is the name of the application, abbreviated.
- *`compName`* is the name of the component.

You'll use this convention throughout this course, and I recommend that you keep using it to define all of your Java classes in packages. (The Java language doesn't force you to follow this package convention. You don't need to specify a package at all, in which case all of your classes must have unique names and are in the default package.)

Import statements

Eclipse simplifies imports
When you write code in the Eclipse editor, you can type the name of a class you want to use, followed by Ctrl+Shift+O. Eclipse figures out which imports you need and adds them automatically. If Eclipse finds two classes with the same name, Eclipse asks you which class you want to add imports for.

Up next in the class definition (referring back to Listing 1) is the *import statement*. 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.makotojava` package:

```
import com.makotojava.*;
```

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.

Class declaration

To define an object in the Java language, you must declare a class. Think of a class as a template for an object, like a cookie cutter.

Listing 1 includes this class declaration:

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

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.

Variables and methods

Classes can have two types of *members*—*variables* and *methods*.

Variables

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 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.

- `public`: Any object in any package can see the variable. (Don't ever use this value; see the **Public variables** sidebar.)

- `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.

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: Class definition for Person

Here's an example that summarizes what you've learned so far. Listing 2 is a class definition for `Person`.

Listing 2. Basic class definition for `Person`

```
package com.makotojava.intro;

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

This basic class definition for `Person` isn't useful at this point, because it defines only `Person`'s attributes (and private ones at that). To be more complete, the `Person` class needs behavior — and that means *methods*.

Methods

A class's methods define its behavior.

Methods fall into two main categories: *constructors*; and all other methods, which come in many types. A constructor method is used only to create an instance of a class. Other types of methods can be used for virtually any application behavior.

The class definition back in Listing 1 shows the way to define the structure of a method, which includes elements like:

- *`accessSpecifier`*

- *`returnType`*
- *`methodName`*
- *`argumentList`*

The combination of these structural elements in a method's definition is called the method's *signature*.

Now take a closer look at the two method categories, starting with constructors.

Constructor methods

You use constructors to specify how to instantiate a class. Listing 1 shows the constructor-declaration syntax in abstract form, and here it is again:

```
accessSpecifier ClassName([argumentList]) {
  constructorStatement(s)
}
```

Constructors are optional
If you don't use a constructor, the compiler provides one for you, called the default (or *no-argument* or *no-arg*) constructor. If you use a constructor other than a no-arg constructor, the compiler doesn't automatically generate one for you.

A constructor's *`accessSpecifier`* is the same as for variables. The name of the constructor must match the name of the class. So if you call your class `Person`, the name of the constructor must also be `Person`.

For any constructor other than the default constructor (see the **Constructors are optional** sidebar), you pass an *`argumentList`*, which is one or more of:

```
argumentType argumentName
```

Arguments in an *`argumentList`* are separated by commas, and no two arguments can have the same name. *`argumentType`* is either a primitive type or another class type (the same as with variable types).

Class definition with a constructor

Now, see what happens when you add the capability to create a `Person` object in two ways: by using a no-arg constructor and by initializing a partial list of attributes.

Listing 3 shows how to create constructors and also how to use `argumentList`:

Listing 3. `Person` class definition with a constructor

```
package com.makotojava.intro;
public class Person {
  private String name;
  private int age;
  private int height;
  private int  weight;
```

```
  private String eyeColor;

  private String gender;
  public Person() {
    // Nothing to do...
  }

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

Note the use of the `this` keyword in making the variable assignments in Listing 3. The `this` keyword is Java shorthand for "this object," and you must use it when you reference two variables with the same name. In this case, `age` is both a constructor parameter and a class variable, so the `this` keyword helps the compiler to tell which is which.

The `Person` object is getting more interesting, but it needs more behavior. And for that, you need more methods.

Other methods

A constructor is a particular kind of method with a particular function. Similarly, many other types of methods perform particular functions in Java programs. Exploration of other method types begins in this unit and continues throughout the course.

Back in Listing 1, you saw how to declare a method:

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

Other methods look much like constructors, with a couple of exceptions. First, you can name other methods whatever you like (though, of course, certain rules apply). I recommend the following conventions:

- Start with a lowercase letter.
- Avoid numbers unless they are absolutely necessary.
- Use only alphabetic characters.

Second, unlike constructors, other methods have an optional *return type*.

Person's other methods

Armed with this basic information, you can see in Listing 4 what happens when you add a few more methods to the `Person` object. (I've omitted constructors for brevity.)

Listing 4. `Person` with a few new methods

```
package com.makotojava.intro;

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

   public String getName() { return name; }
   public void setName(String value) { name = value; }
   // Other getter/setter combinations...
}
```

Notice the comment in Listing 4 about "getter/setter combinations." You'll work more with getters and setters later. For now, all you need to know is that a *getter* is a method for retrieving the value of an attribute, and a *setter* is a method for modifying that value. Listing 4 shows only one getter/setter combination (for the `Name` attribute), but you can define more in a similar fashion.

Note in Listing 4 that if a method doesn't return a value, you must tell the compiler by specifying the `void` return type in its signature.

Static and instance methods

Generally, two types of (nonconstructor) methods are used: *instance methods* and *static methods*. Instance methods depend on the state of a specific object instance for their behavior. Static methods are also sometimes called *class methods*, because their behavior isn't dependent on any single object's state. A static method's behavior happens at the class level.

Static methods are used largely for utility; you can think of them as being global methods (à la C) while keeping the code for the method with the class that defines it.

For example, throughout the coming units, you'll use the JDK `Logger` class to output information to the console. To create a `Logger` class instance, you don't instantiate a `Logger` class; instead, you invoke a static method named `getLogger()`.

The syntax for invoking a static method on a class is different from the syntax used to invoke a method on an object. You also use the name of the class that contains the static method, as shown in this invocation:

```
Logger l = Logger.getLogger("NewLogger");
```

In this example, `Logger` is the name of the class, and `getLogger(...)` is the name of the method. So to invoke a static method, you don't need an object instance, just the name of the class.

Test your understanding

1. What are import statements used for?
    a. They serve no real purpose other than to alert other programmers to your class dependencies, as a courtesy.

   b. They are used for web software to connect to certain sites and download relevant code.
   c. They are used to tell the compiler about which classes your class references.
   d. None of the above
2. If you want a variable to be visible only within the class that defines it, what *`accessSpecifier`* should you give it?
   a. `public`
   b. `private`
   c. None, because this is the compiler's default *`accessSpecifier`*
   d. `protected`
   e. None of the above
3. What are reserved words in the Java language?
   a. Certain words or phrases that are reserved for the Java runtime, so the compiler strips them out before compiling your code.
   b. Words (also known as *keywords*) that you may not use to name your constructs, and that have special meaning in the Java language.
   c. There's no such thing as a "reserved word" in the Java language.
   d. Certain words that are ignored by the compiler, sort of like comments.
   e. None of the above.
4. A package is used as a way to create an area (called a namespace) in which names are unique but outside of which they might not be.
   a. It depends on the compiler context.
   b. False
   c. There is no such thing as a "package" in the Java language.
   d. True
   e. None of the above
5. Listing 4 adds several attributes to the `Person` class. As an exercise, write the getter/setter code to access/set those values. Use the `getName()`/`setName()` methods as an example.
6. In Listing 3, an alternative constructor is also defined, along with the no-arg constructor, even though the no-arg constructor does nothing. Why might you declare the no-arg constructor in this scenario?
   a. If you don't define the no-arg constructor in every class, you'll get a compiler error.
   b. Providing the no-arg constructor allows you to create an empty `Person` object whose attributes are set to default values.
   c. The code in Listing 3 is in error. It won't compile because you cannot have two constructors on the same class.
   d. There's no such thing as a no-arg constructor. This is a trick question.
   e. None of the above
7. What's the difference between static methods and instance methods?
   a. There is virtually no difference between the two.
   b. A static method can be called from any language, whereas instance methods are only allowed in a single object.
   c. Invoking an instance method on an object requires a reference to the object, whereas invoking a static method does not.
   d. A static method "sticks" to the object it is defined on, whereas an instance method can float from one object to another.

Check your answers.

For further exploration

The Java Language Specification

Java Language Basics

Classes and Objects

Java in a Nutshell, 6th Edition by Benjamin J. Evans and David Flanagan

Java: A Beginner's Guide, 6th Edition by Herbert Schildt

IBM Code: Java journeys

Previous: Object-oriented concepts and principles
Next: Your first Java class

IBM

developerWorks®

Unit 5: Your first Java class

Create and test a class in Eclipse

J Steven Perry

September 14, 2016

Learn the structure, syntax, and programming paradigm of the Java platform and language. Start by mastering the essentials of object-oriented programming on the Java platform, and progress incrementally to the more-sophisticated syntax and libraries that you need to develop complex, real-world Java applications.

Before you begin

This unit is part of the "Intro to Java programming" learning path. Although the concepts discussed in the individual units are standalone in nature, the hands-on component builds as you progress through the units, and I recommend that you review the prerequisites, setup, and unit details before proceeding.

Unit objectives

- Use the Eclipse Package Explorer to create a package and declare a class
- Add variables and methods to your class
- Learn how the `Logger` class helps you keep an eye on your application's behavior
- Use the Eclipse code generator
- Create a JUnit test case and run it from inside Eclipse

Your Person class

You're about to create and work with your first Java class — the same `Person` class that you've so far examined in the abstract. The following video takes you through all of this unit's steps. Watch first, and then I'll recap the steps briefly in the text while giving you a closer look at the code.

To view this video, **Your first Java class** , please access the online version of the article. If this article is in the developerWorks archives, the video is no longer accessible.

Step 1: Create a package

Rather than use the default package (almost always a bad idea), create one specifically for the code you're writing. Click **File > New > Package** to start the Java Package wizard. Type `com.makotojava.intro` into the Name text box and click **Finish**.

Step 2: Declare the class

Now you can see your new package in the Package Explorer. The easiest way to create a class from there is to right-click the package and choose **New > Class...**. In the New Class dialog box's Name text box, type `Person` and then click **Finish**.

Eclipse generates a shell class for you and shows the source code in the edit window:

```
package com.makotojava.intro;

public class Person {

}
```

Optionally, tidy up your workspace by closing views that you don't need.

Step 3: Add built-in logging

Before you do any further coding, you need to know how your programs tell you what they're 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`. For now, get in the habit of passing the name of the class that the code you're writing is located in. From any regular (that is, nonstatic) method, the preceding code always references the name of the class and passes that to the `Logger`.

If you're 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());
```

The code you're inside of is the `Person` class, so you reference a special literal called `class` that retrieves the `Class` object and gets its `Name` attribute. Add the logging code so that `Person` now looks like this:

```
package com.makotojava.intro;

import java.util.logging.Logger;

public class Person {

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

}
```

Step 4: Add class variables

In Unit 4, you saw how to begin fleshing out the `Person` class, but I didn't explain much of the syntax. Now you'll learn more formally how to add class variables.

For convenience, Listing 1 repeats the `Person` class code that you saw in Unit 4.

Listing 1. `Person` class definition with a constructor

```
package com.makotojava.intro;
public class Person {
  private String name;
  private int age;
  private int height;
  private int  weight;
  private String eyeColor;

  private String gender;
  public Person() {
    // Nothing to do...
  }

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

Recall that a variable has an *`accessSpecifier`*, a *`dataType`*, a *`variableName`*, and, optionally, an *`initialValue`*. In Unit 4, you looked briefly at how to define the *`accessSpecifier`* and *`variableName`*.

A *`dataType`* can be either a primitive type or a reference to another object. For example, notice in Listing 1 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 don't 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.

Table 1 shows the eight primitive data types you're likely to use on a regular basis, including the default values that primitives take on if you don't explicitly initialize a member variable's value.

Table 1. Primitive data types

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	1.17549435e-38 to 3.4028235e+38
`double`	64 bits	0.0	4.9e-324 to 1.7976931348623157e+308

Step 5: Add getters, setters, a constructor, and a main method

Back in the Eclipse source-code editor for `Person`, add lines 3 through 7 of the following code so that your class now looks like this:

```
package com.makotojava.intro;

import java.util.logging.Logger;

public class Person {

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

  private String name;
  private int age;
  private int height;
  private int weight;
  private String eyeColor;
  private String gender;
}
```

Eclipse has a handy code generator for generating getters and setters (among other items). Right-click the `Person` class in the Package Explorer and select **Source > Generate Getters and Setters...**. In the Generate Getters and Setters dialog box, select all of the attributes except `l` (which is the logger) and click **OK**.

Now, add a constructor to `Person` by typing the code from Listing 2 into your source window, just below the top part of the class definition (the line immediately beneath `public class Person ()`).

Listing 2. `Person` constructor

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

Make sure that you have no wavy lines indicating compile errors.

Using `main()` as a test harness
`main()` is a special method that you can include in any class so that the JRE can execute its code. A class isn't required to have a `main()` method — in fact, most never will — and a class can have at most one `main()` method. `main()` is a handy method to have because

it gives you a quick test harness for the class. In enterprise development, you would use test libraries such as JUnit, but using `main()` can be a quick-and-dirty way to create a test harness.

Step 6: Generate a JUnit test case

Now you can generate a JUnit test case where you instantiate a `Person`, using the constructor in Listing 2, and then print the state of the object to the console. In this sense, the "test" makes sure that the order of the attributes on the constructor call are correct (that is, that they're set to the correct attributes).

In the Package Explorer, right-click your `Person` class and select **New > JUnit Test Case**. In the first page of the New JUnit Test Case wizard, accept the defaults by clicking **Next**. In the Test Methods dialog box — the place where you select the methods that you want the wizard to build tests for — select only the constructor and click **Finish**. Eclipse then generates the JUnit test case.

Open `PersonTest`, go into the `testPerson()` method, and make it look like Listing 3.

Listing 3. The `testPerson()` method

```
@Test
public void testPerson() {
  Person p = new Person("Joe Q Author", 42, 173, 82, "Brown", "MALE");
  Logger l = Logger.getLogger(Person.class.getName());
  l.info("Name: " + p.getName());
  l.info("Age:" + p.getAge());
  l.info("Height (cm):" + p.getHeight());
  l.info("Weight (kg):" + p.getWeight());
  l.info("Eye Color:" + p.getEyeColor());
  l.info("Gender:" + p.getGender());
  assertEquals("Joe Q Author", p.getName());
  assertEquals(42, p.getAge());
  assertEquals(173, p.getHeight());
  assertEquals(82, p.getWeight());
  assertEquals("Brown", p.getEyeColor());
  assertEquals("MALE", p.getGender());
}
```

Step 7: Run your unit test in Eclipse

In Eclipse, right-click PersonTest.java in the Package Explore and select **Run As** > **JUnit Test**. The Console view opens automatically to show `Logger` output, and the JUnit view indicates that the test ran without errors.

For further exploration

Lesson: Classes and Objects

Pragmatic Unit Testing in Java 8 with JUnit by Dave Thomas, Andy Hunt, and Jeff Langr

JUnit Pocket Guide by Kent Beck

Think Java by Allen B. Downey and Chris Mayfield (see Chapter 11, "Classes")

Core Java by Cay Horstman (see "Lesson 4: Objects and Classes")

IBM Code: Java journeys

Previous: Getting started with the Java language
Next: Adding behavior to a Java class

Unit 6: Adding behavior to a Java class

Learn syntax for accessor method declarations and method calls

J Steven Perry

September 14, 2016

Learn the structure, syntax, and programming paradigm of the Java platform and language. Start by mastering the essentials of object-oriented programming on the Java platform, and progress incrementally to the more-sophisticated syntax and libraries that you need to develop complex, real-world Java applications.

Before you begin

This unit is part of the "Intro to Java programming" learning path. Although the concepts discussed in the individual units are standalone in nature, the hands-on component builds as you progress through the units, and I recommend that you review the prerequisites, setup, and unit details before proceeding.

Unit objectives

- Know how to hand-code the declaration of an accessor method pair
- Learn correct syntax for calling methods with or without parameters
- Know how to nest method invocations

Accessor methods

The getters and setters that you saw in action at the end of Unit 5 are called *accessor methods*. (Quick review: A getter is a method for retrieving the value of an attribute; a setter is a method for modifying that value.) To encapsulate a class's data from other objects, you declare its variables to be `private` and then provide accessor methods.

The naming of accessors follows a strict convention known as the *JavaBeans pattern*. In this pattern, any attribute `Foo` has a getter called `getFoo()` and a setter called `setFoo()`. The JavaBeans pattern is so common that support for it is built into the Eclipse IDE, as you saw when you generated getters and setters for `Person`.

Accessors follow these guidelines:

- The attribute is always declared with `private` access.
- The access specifier for getters and setters is `public`.
- A getter doesn't take any parameters, and it returns a value whose type is the same as the attribute it accesses.
- Setters take only one parameter, of the type of the attribute, and do not return a value.

Declaring accessors

By far the easiest way to declare accessors is to let Eclipse do it for you. But you also need to know how to hand-code a getter-and-setter pair.

Suppose I have an attribute, `Foo`, whose type is `java.lang.String`. My complete declaration for `Foo` (following the accessor guidelines) is:

```
private String foo;
public String getFoo() {
  return foo;
}
public void setFoo(String value) {
  foo = value;
}
```

Notice that the parameter value passed to the setter is named differently than if it had been Eclipse-generated (where the parameter name would be the same as the attribute name — for example, `public void setFoo(String foo)`). On the rare occasions when I hand-code a setter, I always use `value` as the name of the parameter value to the setter. This eye-catcher — my own convention, and one that I recommend to other developers — reminds me that I hand-coded the setter. If I don't use Eclipse to generate getters and setters for me, I have a good reason. Using `value` as the setter's parameter value reminds me that this setter is special. (Code comments can serve the same purpose.)

Calling methods

Invoking — or *calling*— methods is easy. The `testPerson` method at the end of Unit 5, for example, invokes the various getters of `Person` to return their values. Now you'll learn the formal mechanics of making method calls.

Method invocation with and without parameters

To invoke a method on an object, you need a reference to that object. Method-invocation syntax comprises:

- The object reference
- A literal dot
- The method name
- Any parameters that need to be passed

The syntax for a method invocation without parameters is:

```
objectReference.someMethod();
```

Here's an example:

```
Person p = new Person("Joe Q Author", 42, 173, 82, "Brown", "MALE");
p.getName();
```

The syntax for a method invocation with parameters is:

```
objectReference.someOtherMethod(parameter1, parameter2, . . ., parameterN);
```

And here's an example (setting the `Name` attribute of `Person`):

```
Person p = new Person("Joe Q Author", 42, 173, 82, "Brown", "MALE");
p.setName("Jane Q Author");
```

Remember that constructors are methods, too. And you can separate the parameters with spaces and newlines. The Java compiler doesn't care. These next two method invocations are equivalent:

```
new Person("Joe Q Author", 42, 173, 82, "Brown", "MALE");
```

```
new Person("Joe Q Author",// Name
  42,     // Age
  173,    // Height in cm
  82,     // Weight in kg
  "Brown",// Eye Color
  "MALE");// Gender
```

Notice how the comments in the second constructor invocation make it more readable for the next person who might work with this code. At a glance, that developer can tell what each parameter is for.

Nested method invocation

Method invocations can also be nested:

```
Logger l = Logger.getLogger(Person.class.getName());
l.info("Name: " + p.getName());
```

Here you pass the return value of `Person.class.getName()` to the `getLogger()` method. Remember that the `getLogger()` method call is a static method call, so its syntax differs slightly. (You don't need a `Logger` reference to make the invocation; instead, you use the name of the class as the left side of the invocation.)

That's all there is to method invocation.

Test your understanding

1. Which answer best describes the Java Beans pattern?
    a. It's where you use the Java API to write any application that's compliant with the Java specification.

   b. It's a particular way of writing code with regard to how to declare, access, and set values on the attributes of a Java object.
   c. The JavaBeans pattern cannot actually be coded; it exists only as an ideal standard.
   d. None of the above.
2. What's the difference between invoking a method and calling a method?
   a. Invoking a method is writing the method's code to use the `static` keyword. Calling a method is only for instance-based objects.
   b. Calling a method means to document it thoroughly, whereas invoking the method is done through Java source code.
   c. There's no difference; calling a method and invoking a method are synonyms for "executing a method."
   d. The difference only manifests when the code is written in Python or Ruby.
   e. None of the above
3. Constructors are **not** methods.
   a. It depends on the compiler context.
   b. False
   c. Sometimes, except that constructors cannot have parameters.
   d. True
4. **Programming problem**: Implement the `toString()` method on `Person`. Use the code for the `Person` class from Question 5 in Unit 4's quiz. Use the Javadoc for `Object.toString()` for a description of what `toString()` is supposed to do.
5. **Programming problem**: Write a JUnit test case to test the `toString()` method on `Person` from Question 4. Add your test method to the `PersonTest` class that you created in Unit 5.

Check your answers.

For further exploration

The Java Tutorials: Defining Methods

Object-oriented language basics, Part 2: Fields and methods

IBM Code: Java journeys

Previous: Your first Java class
Next: Strings and operators

Unit 7: Strings and operators

Instantiate and manipulate strings, and explore arithmetic operators

J Steven Perry

September 14, 2016

Learn the structure, syntax, and programming paradigm of the Java platform and language. Start by mastering the essentials of object-oriented programming on the Java platform, and progress incrementally to the more-sophisticated syntax and libraries that you need to develop complex, real-world Java applications.

Before you begin

This unit is part of the "Intro to Java programming" learning path. Although the concepts discussed in the individual units are standalone in nature, the hands-on component builds as you progress through the units, and I recommend that you review the prerequisites, setup, and unit details before proceeding.

Unit objectives

- Learn multiple ways to instantiate a `String` object and set its value
- Understand string concatenation and method chaining
- Gain familiarity with the Java language's arithmetic operators

Strings

In C, string handling is labor intensive because strings are null-terminated arrays of 8-bit characters that you must manipulate. The closest Java code gets to the C world with regard to strings is the `char` primitive data type, which can hold a single Unicode character, such as *a*.

So far, you've encountered several variables of type `String`. In the Java language, strings are first-class objects of type `String`, with methods that help you manipulate them.

Here are a couple of ways to create a String, using the example of creating a `String` instance named `greeting` with a value of `hello`:

```
greeting = new String("hello");
```

```
String greeting = "hello";
```

Because `String`s are first-class objects, you can use `new` to instantiate them. Setting a variable of type `String` to a string literal has the same result, because the Java language creates a `String` object to hold the literal, and then assigns that object to the instance variable.

Concatenating strings

You can do many things with `String`, and the class has many helpful methods. Without even using a method, you've already done something interesting within the `Person` class's `testPerson()` method by concatenating, or combining, two `String`s:

```
l.info("Name: " + p.getName());
```

The plus (`+`) sign is shorthand for concatenating `String`s in the Java language. (You incur a performance penalty for doing this type of concatenation inside a loop, but for now, you don't need to worry about that.)

Concatenation exercise

Now, you can try concatenating two more `String`s inside of the `Person` class. At this point, you have a `name` instance variable, but it would be more realistic in a business application to have a `firstName` and `lastName`. You can then concatenate them when another object requests `Person`'s full name.

Return to your Eclipse project, and start by adding the new instance variables (at the same location in the source code where `name` is currently defined):

```
//private String name;
private String firstName;
private String lastName;
```

Comment out the `name` definition; you don't need it anymore, because you're replacing it with `firstName` and `lastName`.

Chaining method calls

Now, tell the Eclipse code generator to generate getters and setters for `firstName` and `lastName` (refer back to Unit 5 if necessary). Then, remove the `setName()` and `getName()` methods, and add a new `getFullName()` method to look like this:

```
public String getFullName() {
  return getFirstName().concat(" ").concat(getLastName());
}
```

This code illustrates *chaining* of method calls. Chaining is a technique commonly used with immutable objects like `String`, where a modification to an immutable object always returns the modification (but doesn't change the original). You then operate on the returned, changed value.

Operators

You've already seen that the Java language uses the `=` operator to assign values to variables. As you might expect, the Java language can do arithmetic, and it uses operators for that purpose too. Now, I give you a brief look at some of the Java language operators you need as your skills improve.

The Java language uses two types of operators:

- *Unary*: Only one operand is needed.
- *Binary*: Two operands are needed.

Table 1 summarizes the Java language's arithmetic operators:

Table 1. Java language's arithmetic operators

Operator	Usage	Description
`+`	`a + b`	Adds `a` and `b`
`+`	`+a`	Promotes `a` to `int` if it's a `byte`, `short`, or `char`
`-`	`a - b`	Subtracts `b` from `a`
`-`	`-a`	Arithmetically negates `a`
`*`	`a * b`	Multiplies `a` and `b`
`/`	`a / b`	Divides `a` by `b`
`%`	`a % b`	Returns the remainder of dividing `a` by `b` (the modulus operator)
`++`	`a++`	Increments `a` by 1; computes the value of `a` before incrementing
`++`	`++a`	Increments `a` by 1; computes the value of `a` after incrementing
`--`	`a--`	Decrements `a` by 1; computes the value of `a` before decrementing
`--`	`--a`	Decrements `a` by 1; computes the value of `a` after decrementing
`+=`	`a += b`	Shorthand for `a = a + b`
`-=`	`a -= b`	Shorthand for `a = a - b`
`*=`	`a *= b`	Shorthand for `a = a * b`
`%=`	`a %= b`	Shorthand for `a = a % b`

Additional operators

In addition to the operators in Table 1, you've seen several other symbols that are called operators in the Java language, including:

- Period (`.`), which qualifies names of packages and invokes methods
- Parentheses (`()`), which delimit a comma-separated list of parameters to a method

- `new`, which (when followed by a constructor name) instantiates an object

The Java language syntax also includes several operators that are used specifically for conditional programming — that is, programs that respond differently based on different input. You look at those in the next unit.

For further exploration

The Java Tutorials: Strings

Android API Reference: String

Think Java by Allen B. Downey and Chris Mayfield (see Chapter 9, "Strings and Things")

IBM Code: Java journeys

Previous: Adding behavior to a Java class
Next: Conditional operators and control statements

Unit 8: Conditional operators and control statements

Make decisions in your code

J Steven Perry

September 14, 2016

Learn the structure, syntax, and programming paradigm of the Java platform and language. Start by mastering the essentials of object-oriented programming on the Java platform, and progress incrementally to the more-sophisticated syntax and libraries that you need to develop complex, real-world Java applications.

Before you begin

This unit is part of the "Intro to Java programming" learning path. Although the concepts discussed in the individual units are standalone in nature, the hands-on component builds as you progress through the units, and I recommend that you review the prerequisites, setup, and unit details before proceeding.

Unit objectives

- Know when and how to use relational operators, conditional operators, and control statements
- Understand the concept of variable scope and its basic rules
- Become familiar with the ternary operator

Relational and conditional operators

The Java language gives you operators and control statements that you can use to make decisions in your code. Most often, a decision in code starts with a *Boolean expression*— that is, one that evaluates to either `true` or `false`. Such expressions use *relational operators*, which compare one operand to another, and *conditional operators*.

Table 1 lists the relational and conditional operators of the Java language.

Table 1. Relational and conditional operators

Operator	Usage	Returns true if...
`>`	`a > b`	`a` is greater than `b`
`>=`	`a >= b`	`a` is greater than or equal to `b`
`<`	`a < b`	`a` is less than `b`

`<=`	`a <= b`	`a` is less than or equal to `b`
`==`	`a == b`	`a` is equal to `b`
`!=`	`a != b`	`a` is not equal to `b`
`&&`	`a && b`	`a` and `b` are both true, conditionally evaluates `b` (if `a` is false, `b` is not evaluated)
`||`	`a || b`	`a` or `b` is true, conditionally evaluates `b` (if `a` is true, `b` is not evaluated)
`!`	`!a`	`a` is false
`&`	`a & b`	`a` and `b` are both true, always evaluates `b`
`|`	`a | b`	`a` or `b` is true, always evaluates `b`
`^`	`a ^ b`	`a` and `b` are different

The if statement

Now that you have a bunch of operators, it's time to use them. This code shows what happens when you add some logic to the `Person` object's `getHeight()` accessor:

```
public int getHeight() {
  int ret = height;
  // If locale of the computer this code is running on is U.S.,
  if (Locale.getDefault().equals(Locale.US))
    ret /= 2.54;// convert from cm to inches
  return ret;
}
```

If the current locale is in the United States (where the metric system isn't in use), it might make sense to convert the internal value of `height` (in centimeters) to inches. This (somewhat contrived) example illustrates the use of the `if` statement, which evaluates a Boolean expression inside parentheses. If that expression evaluates to `true`, the program executes the next statement.

In this case, you only need to execute one statement if the `Locale` of the computer the code is running on is `Locale.US`. If you need to execute more than one statement, you can use curly braces to form a *compound statement*. A compound statement groups many statements into one — and compound statements can also contain other compound statements.

Variable scope

Every variable in a Java application has *scope*, or localized namespace, where you can access it by name within the code. Outside that space the variable is *out of scope*, and you get a compile error if you try to access it. Scope levels in the Java language are defined by where a variable is declared, as shown in Listing 1.

Listing 1. Variable scope

```
public class SomeClass {
  private String someClassVariable;
  public void someMethod(String someParameter) {
    String someLocalVariable = "Hello";

    if (true) {
      String someOtherLocalVariable = "Howdy";
    }
    someClassVariable = someParameter; // legal
    someLocalVariable = someClassVariable; // also legal
    someOtherLocalVariable = someLocalVariable;// Variable out of scope!
  }
  public void someOtherMethod() {
    someLocalVariable = "Hello there";// That variable is out of scope!

  }
}
```

Within `SomeClass`, `someClassVariable` is accessible by all instance (that is, nonstatic) methods. Within `someMethod`, `someParameter` is visible, but outside of that method it isn't, and the same is true for `someLocalVariable`. Within the `if` block, `someOtherLocalVariable` is declared, and outside of that `if` block it's out of scope. For this reason, we say that Java has *block scope*, because blocks (delimited by `{` and `}`) define the scope boundaries.

Scope has many rules, but Listing 1 shows the most common ones. Take a few minutes to familiarize yourself with them.

The else statement

Sometimes in a program's control flow, you want to take action only if a particular expression fails to evaluate to `true`. That's when `else` comes in handy:

```
public int getHeight() {
  int ret;
  if (gender.equals("MALE"))
    ret = height + 2;
  else {
    ret = height;
    Logger.getLogger("Person").info("Being honest about height...");
  }
  return ret;
}
```

The `else` statement works the same way as `if`, in that the program executes only the next statement that it encounters. In this case, two statements are grouped into a compound statement (notice the curly braces), which the program then executes.

You can also use `else` to perform an additional `if` check:

```
if (conditional) {
  // Block 1
} else if (conditional2) {
  // Block 2
} else if (conditional3) {
  // Block 3
} else {
  // Block 4
} // End
```

If *`conditional`* evaluates to `true`, *`Block 1`* is executed and the program jumps to the next statement after the final curly brace (which is indicated by `// End`). If *`conditional`* does **not** evaluate to `true`, then *`conditional2`* is evaluated. If *`conditional2`* is true, then *`Block 2`* is executed, and the program jumps to the next statement after the final curly brace. If *`conditional2`* is not true, then the program moves on to *`conditional3`*, and so on. Only if all three conditionals fail is *`Block 4`* executed.

The ternary operator

The Java language provides a handy operator for doing simple `if` / `else` statement checks. This operator's syntax is:

```
(conditional) ? statementIfTrue : statementIfFalse;
```

If *`conditional`* evaluates to `true`, *`statementIfTrue`* is executed; otherwise, *`statementIfFalse`* is executed. Compound statements are not allowed for either statement.

The ternary operator comes in handy when you know that you need to execute one statement as the result of the conditional evaluating to `true`, and another if it doesn't. Ternary operators are most often used to initialize a variable (such as a return value), like so:

```
public int getHeight() {
  return (gender.equals("MALE")) ? (height + 2) : height;
}
```

The parentheses following the question mark aren't strictly required, but they do make the code more readable.

For further exploration

The Java Tutorials: Operators

The Java Tutorials: Control Flow Statements

What is the Java ?: operator called and what does it do?

Java — Decision Making

IBM Code: Java journeys

Previous: Strings and operators

Next: Loops

Unit 9: Loops

Iterate over code or execute it repeatedly

J Steven Perry

September 14, 2016

Learn the structure, syntax, and programming paradigm of the Java platform and language. Start by mastering the essentials of object-oriented programming on the Java platform, and progress incrementally to the more-sophisticated syntax and libraries that you need to develop complex, real-world Java applications.

Before you begin

This unit is part of the "Intro to Java programming" learning path. Although the concepts discussed in the individual units are standalone in nature, the hands-on component builds as you progress through the units, and I recommend that you review the prerequisites, setup, and unit details before proceeding.

Unit objectives

- Understand the purpose of a loop
- Use correct syntax for `for`, `while`. and `do`...`while` statements
- Know how to break out of a loop, or skip a loop interation and then continue

What is a loop?

Sometimes you want your code to do the same thing over and over again until the job is done. You can use a *loop* for this purpose. A loop is a programming construct that executes repeatedly while a specific condition (or set of conditions) is met. For instance, you might ask a program to read all records until the end of a data file, or to process each element of an array in turn. (You'll learn about arrays in the next unit.)

Three loop constructs make it possible to iterate over code or execute it more than once:

- `for` loops
- `while` loops
- `do`...`while` loops

for loops

The basic loop construct in the Java language is the `for` statement. You can use a `for` statement to iterate over a range of values to determine how many times to execute a loop. The abstract syntax for a `for` loop is:

```
for (initialization; loopWhileTrue; executeAtBottomOfEachLoop) {
  statementsToExecute
}
```

At the *beginning* of the loop, the initialization statement is executed (multiple initialization statements can be separated by commas). Provided that *`loopWhileTrue`* (a Java conditional expression that must evaluate to either `true` or `false`) is true, the loop executes. At the *bottom* of the loop, *`executeAtBottomOfEachLoop`* executes.

For example, if you wanted the code in the `main()` method in Listing 1 to execute three times, you can use a `for` loop

Listing 1. A `for` loop

```
public static void main(String[] args) {
  Logger l = Logger.getLogger(Person.class.getName());
  for (int aa = 0; aa < 3; aa++)
    Person p = new Person("Joe Q Author", 42, 173, 82, "Brown", "MALE");
    l.info("Loop executing iteration# " + aa);
    l.info("Name: " + p.getName());
    l.info("Age:" + p.getAge());
    l.info("Height (cm):" + p.getHeight());
    l.info("Weight (kg):" + p.getWeight());
    l.info("Eye Color:" + p.getEyeColor());
    l.info("Gender:" + p.getGender());
  }
}
```

The local variable `aa` is initialized to zero at the beginning of Listing 1. This statement executes only once, when the loop is initialized. The loop then continues three times, and each time `aa` is incremented by one.

You'll see in the next unit that an alternative `for` loop syntax is available for looping over constructs that implement the `Iterable` interface (such as arrays and other Java utility classes). For now, just note the use of the `for` loop syntax in Listing 1.

while loops

The syntax for a `while` loop is:

```
while (condition) {
  statementsToExecute
}
```

As you might suspect, if *condition* evaluates to `true`, the loop executes. At the top of each iteration (that is, before any statements execute), the condition is evaluated. If the condition evaluates to `true`, the loop executes. So it's possible that a `while` loop will never execute if its conditional expression is not true at least once.

Look again at the `for` loop in Listing 1. For comparison, Listing 2 uses a `while` loop to obtain the same result.

Listing 2. A `while` loop

```
public static void main(String[] args) {
  Logger l = Logger.getLogger(Person.class.getName());
  int aa = 0;
  while (aa < 3) {
    Person p = new Person("Joe Q Author", 42, 173, 82, "Brown", "MALE");
    l.info("Loop executing iteration# " + aa);
    l.info("Name: " + p.getName());
    l.info("Age:" + p.getAge());
    l.info("Height (cm):" + p.getHeight());
    l.info("Weight (kg):" + p.getWeight());
    l.info("Eye Color:" + p.getEyeColor());
    l.info("Gender:" + p.getGender());
    aa++;
  }
}
```

As you can see, a `while` loop requires a bit more housekeeping than a `for` loop. You must initialize the `aa` variable and also remember to increment it at the bottom of the loop.

do...while loops

If you want a loop that always executes once and *then* checks its conditional expression, you can use a `do...while` loop, as shown in Listing 3.

Listing 3. A `do...while` loop

```
int aa = 0;
do {
  Person p = new Person("Joe Q Author", 42, 173, 82, "Brown", "MALE");
  l.info("Loop executing iteration# " + aa);
  l.info("Name: " + p.getName());
  l.info("Age:" + p.getAge());
  l.info("Height (cm):" + p.getHeight());
  l.info("Weight (kg):" + p.getWeight());
  l.info("Eye Color:" + p.getEyeColor());
  l.info("Gender:" + p.getGender());
  aa++;
} while (aa < 3);
```

The conditional expression (`aa < 3`) is not checked until the end of the loop.

Loop termination

At times, you need to bail out of — or *terminate*— a loop before the conditional expression evaluates to `false`. This situation can occur if you're searching an array of `String`s for a particular value, and once you find it, you don't care about the other elements of the array. For the times when you want to bail, the Java language provides the `break` statement, shown in Listing 4.

Listing 4. A `break` statement

```
public static void main(String[] args) {
  Logger l = Logger.getLogger(Person.class.getName());
  int aa = 0;
  while (aa < 3) {
    if (aa == 1)
      break;
    Person p = new Person("Joe Q Author", 42, 173, 82, "Brown", "MALE");
    l.info("Loop executing iteration# " + aa);
    l.info("Name: " + p.getName());
    l.info("Age:" + p.getAge());
    l.info("Height (cm):" + p.getHeight());
    l.info("Weight (kg):" + p.getWeight());
    l.info("Eye Color:" + p.getEyeColor());
    l.info("Gender:" + p.getGender());
    aa++;
  }
}
```

The `break` statement takes you to the next executable statement outside of the loop in which it's located.

Loop continuation

In the (simplistic) example in Listing 4, you want to execute the loop only once and then bail. You can also skip a single iteration of a loop but continue executing the loop. For that purpose, you need the `continue` statement, shown in Listing 5.

Listing 5. A `continue` statement

```
public static void main(String[] args) {
  Logger l = Logger.getLogger(Person.class.getName());
  int aa = 0;
  while (aa < 3) {
    aa++;
    if (aa == 2)
      continue;
    Person p = new Person("Joe Q Author", 42, 173, 82, "Brown", "MALE");
    l.info("Loop executing iteration# " + aa);
    l.info("Name: " + p.getName());
    l.info("Age:" + p.getAge());
    l.info("Height (cm):" + p.getHeight());
    l.info("Weight (kg):" + p.getWeight());
    l.info("Eye Color:" + p.getEyeColor());
    l.info("Gender:" +
    p.getGender());
  }
}
```

In Listing 5, you skip the second iteration of a loop but continue to the third. `continue` comes in handy when you are, say, processing records and come across a record you don't want to process. You can skip that record and move on to the next one.

Test your understanding

1. In Unit 7, you saw the `concat()` method used to illustrate method chaining in the implementation of `getName()`. Implement the `getName()` method using string concatenation (with the `+` operator). Write a JUnit test to ensure that the result is as you expect.

2. Which operator(s) are binary operators?
   a. `++`
   b. `+`
   c. `%`
   d. `--`
   e. `+=`
   f. a, b, and d
   g. b, c, and e
   h. None of the above
3. *String concatenation* means combining the contents of two or more strings to make a single, larger string.
   a. True
   b. False
4. Which of the following means "if a is less than or equal to b"?
   a. `if (a lt b)`
   b. `if (b < a)`
   c. `if (b +< a)`
   d. `if (a <= b)`
   e. None of the above
5. What is the value of variable `c` after execution of the following block? Explain your answer.

```
public int method1() {
   int a = 10;
   int b = 7;
   int c = 0;

   if (b >= a)
      c++;
      b = -47;
   if (b > c)
      c = b;

   return c;
}
```

6. Will the following code compile? Explain your answer.

```
public int method2() {
   int a = 0;

   if (a > 0) {
      int b = a;
   }

   int c = b;

   return c;
}
```

7. What is the value of variable `c` after execution of the following block? Explain your answer.

```
public boolean method3() {
   boolean a = true;
   boolean b = false;
   boolean c;

   if (a = b)
      c = false;
   else
      c = a;

   return c;
}
```

8. A `for` loop *always* executes at least once.
   a. True
   b. False
9. The condition that determines whether or not a `do...while` loop continues is checked at the **bottom** of the loop.
   a. True
   b. False
10. Write a method that contains a `for` loop that starts at 3, loops six times, and outputs the iteration number (1-based), along with the current loop variable using a JDK `Logger`. Use Listing 1 as an example of how to use `Logger`. Verify the results using your `Logger` output.
11. Repeat the exercise in Question 10 with a different method name, but instead use a `while` loop. Verify the results using your `Logger` output.
12. Repeat the exercise in Question 10 using a `do...while` loop. Verify the results using your `Logger` output.
13. Write a method that contains a `for` loop that starts at zero (0), loops ten (10) times, and outputs the iteration number (1-based), along with the current loop variable using a JDK `Logger` class. Skip iterations 4, 5, and 9. Verify the results using your `Logger` output.

Check your answers.

For further exploration

The Java Tutorials: The while and do-while Statements

The Java Tutorials: The for Statement

Java — Loop Control

IBM Code: Java journeys

Previous: Conditional operators and control statements
Next: Java Collections

Unit 10: Java Collections

Create and manage collections of objects

J Steven Perry September 14, 2016

Learn the structure, syntax, and programming paradigm of the Java platform and language. Start by mastering the essentials of object-oriented programming on the Java platform, and progress incrementally to the more-sophisticated syntax and libraries that you need to develop complex, real-world Java applications.

Before you begin

This unit is part of the "Intro to Java programming" learning path. Although the concepts discussed in the individual units are standalone in nature, the hands-on component builds as you progress through the units, and I recommend that you review the prerequisites, setup, and unit details before proceeding.

Unit objectives

- Understand the purpose of the Java Collections Framework
- Know how to declare and use Java arrays, lists, sets, and maps
- Understand boxing and unboxing
- Know how to make a collection iterable

The Java Collections Framework

Most real-world applications deal with collections of things like files, variables, records from files, or database result sets. The Java language has a sophisticated Collections Framework that you can use to create and manage collections of objects of various types. This unit introduces you to the most commonly used collection classes and gets you started with using them.

Arrays

Note: The square brackets in this unit's code examples are part of the required syntax for Java arrays, **not** indicators of optional elements.

Most programming languages include the concept of an *array* to hold a collection of things, and the Java language is no exception. An array is basically a collection of elements of the same type.

You can declare an array in one of two ways:

- Create the array with a certain size, which is fixed for the life of the array.
- Create the array with a certain set of initial values. The size of this set determines the size of the array — it's exactly large enough to hold all of those values, and its size is fixed for the life of the array.

Declaring an array

In general, you declare an array like this:

```
new elementType [arraySize]
```

You can create an integer array of elements in two ways. This statement creates an array that has space for five elements but is empty:

```
// creates an empty array of 5 elements:
int[] integers = new int[5];
```

This statement creates the array and initializes it all at once:

```
// creates an array of 5 elements with values:
int[] integers = new int[] { 1, 2, 3, 4, 5 };
```

or

```
// creates an array of 5 elements with values (without the new operator):
int[] integers = { 1, 2, 3, 4, 5 };
```

The initial values go between the curly braces and are separated by commas.

Another way to create an array is to create it and then code a loop to initialize it:

```
int[] integers = new int[5];
for (int aa = 0; aa < integers.length; aa++) {
  integers[aa] = aa+1;
}
```

The preceding code declares an integer array of five elements. If you try to put more than five elements in the array, the Java runtime will throw an *exception*. You'll learn about exceptions and how to handle them in Unit 14.

Loading an array

To load the array, you loop through the integers from 1 through the length of the array (which you get by calling `.length` on the array — more about that in a minute). In this case, you stop when you hit 5.

Once the array is loaded, you can access it as before:

```
Logger l = Logger.getLogger("Test");
for (int aa = 0; aa < integers.length; aa++) {
  l.info("This little integer's value is: " + integers[aa]);
}
```

This syntax also works, and (because it's simpler to work with) I use it throughout this unit:

```
Logger l = Logger.getLogger("Test");
for (int i : integers) {
  l.info("This little integer's value is: " + i);
}
```

The element index

Think of an array as a series of buckets, and into each bucket goes an element of a certain type. Access to each bucket is gained via an element *index*:

```
element = arrayName [elementIndex];
```

To access an element, you need the reference to the array (its name) and the index that contains the element that you want.

The length attribute

Every array has a `length` attribute, which has `public` visibility, that you can use to find out how many elements can fit in the array. To access this attribute, use the array reference, a dot (.), and the word length, like this:

```
int arraySize = arrayName.length;
```

Arrays in the Java language are *zero-based*. That is, for any array, the first element in the array is always at `arrayName[0]`, and the last is at `arrayName[arrayName.length - 1]`.

An array of objects

You've seen how arrays can hold primitive types, but it's worth mentioning that they can also hold objects. Creating an array of `java.lang.Integer` objects isn't much different from creating an array of primitive types and, again, you can do it in two ways:

```
// creates an empty array of 5 elements:
Integer[] integers = new Integer[5];
```

```
// creates an array of 5 elements with values:
Integer[] integers = new Integer[] {
Integer.valueOf(1),
Integer.valueOf(2),
Integer.valueOf(3),
Integer.valueOf(4),
Integer.valueOf(5)
};
```

Boxing and unboxing

Every primitive type in the Java language has a JDK counterpart class, as shown in Table 1.

Table 1. Primitives and JDK counterparts

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 (you've already seen this in calls to `Logger`), the primitive type is autoboxed, and wrapper types automatically have `toString()` invoked on them. Pretty handy.

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 learn about interfaces Unit 17.) You'll work here with its most commonly used implementation, `ArrayList`. You can make the declaration in two ways. The first uses the explicit syntax:

```
List<String> listOfStrings = new ArrayList<String>();
```

The second way uses the "diamond" operator (introduced in JDK 7):

```
List<String> listOfStrings = new ArrayList<>();
```

Notice that the type of the object in the `ArrayList` instantiation isn't specified. This is the case because the type of the class on the right side of the expression must match that of the left side. Throughout the remainder of this learning path, I use both types, because you're likely to see both usages in practice.

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 unit, you'll look more at the rules governing these types of variable assignments.

Formal type

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

- 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));
```

The `add()` method adds the element to the end of the `List`.

To ask the `List` how big it is, call `size()`:

```
List<Integer> listOfIntegers = new ArrayList<>();

listOfIntegers.add(Integer.valueOf(238));
Logger l = Logger.getLogger("Test");
l.info("Current List size: " + listOfIntegers.size());
```

To retrieve an item from the `List`, call `get()` and pass it the index of the item you want:

```
List<Integer> listOfIntegers = new ArrayList<>();
listOfIntegers.add(Integer.valueOf(238));
Logger l = Logger.getLogger("Test");
l.info("Item at index 0 is: " listOfIntegers.get(0));
```

In a real-world application, a `List` would contain records, or business objects, and you'd possibly want to look over them all as part of your processing. How do you do that in a generic fashion? Answer: You want to *iterate* over the collection, which you can do because `List` implements the `java.lang.Iterable` interface.

Iterable

If a collection implements `java.lang.Iterable`, it's called an *iterable collection*. You can start at one end and walk through the collection item-by-item until you run out of items.

In the Loops unit, I briefly mentioned the special syntax for iterating over collections that implement the `Iterable` interface. Here it is again in more detail:

```
for (objectType varName : collectionReference) {
  // Start using objectType (via varName) right away...
}
```

The preceding code is abstract; here's a more realistic example:

```
List<Integer> listOfIntegers = obtainSomehow();
Logger l = Logger.getLogger("Test");
for (Integer i : listOfIntegers) {
  l.info("Integer value is : " + i);
}
```

That little code snippet does the same thing as this longer one:

```
List<Integer> listOfIntegers = obtainSomehow();
Logger l = Logger.getLogger("Test");
for (int aa = 0; aa < listOfIntegers.size(); aa++) {
  Integer i = listOfIntegers.get(aa);
  l.info("Integer value is : " + i);
}
```

The first snippet uses shorthand syntax: It has no `index` variable (`aa` in this case) to initialize, and no call to the `List` 's `get()` method.

Because `List` extends `java.util.Collection`, which implements `Iterable`, you can use the shorthand syntax to iterate over any `List`.

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.

Keep this behavior in mind when iterating over a `Set`, like so:

```
Set<Integer> setOfIntegers = new HashSet();
setOfIntegers.add(Integer.valueOf(10));
setOfIntegers.add(Integer.valueOf(20));
setOfIntegers.add(Integer.valueOf(30));
setOfIntegers.add(Integer.valueOf(40));
setOfIntegers.add(Integer.valueOf(50));
Logger l = Logger.getLogger("Test");
for (Integer i : setOfIntegers) {
  l.info("Integer value is : " + i);
}
```

Chances are that the objects print out in a different order from the order you added them in, because a `Set` guarantees uniqueness, not order. You can see this result if you paste the preceding code into the `main()` method of your `Person` class and run it.

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 `Map`s 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;
}
```

In this example, `Map` contains `Integer` s, keyed by a `String`, which happens to be their `String` representation. To retrieve a particular `Integer` value, you need its `String` representation:

```
mapOfIntegers = createMapOfIntegers();
Integer oneHundred68 = mapOfIntegers.get("168");
```

Using Set with Map

On occasion, you might find yourself with a reference to a `Map`, and you want to walk over its entire set of contents. In this case, you need a `Set` of the keys to the `Map`:

```
Set<String> keys = mapOfIntegers.keySet();
Logger l = Logger.getLogger("Test");
for (String key : keys) {
  Integer  value = mapOfIntegers.get(key);
  l.info("Value keyed by '" + key + "' is '" + value + "'");
}
```

Note that the `toString()` method of the `Integer` retrieved from the `Map` is automatically called when used in the `Logger` call. `Map` returns a `Set` of its keys because the `Map` is keyed, and each key is unique. Uniqueness (not order) is the distinguishing characteristic of a `Set` (which might explain why there's no `keyList()` method).

Test your understanding

1. Which of these statements is true about Java arrays?
    a. Once created, the size of the array is fixed (that is, cannot be changed).
    b. To obtain the size of the array, use the `.length` property.
    c. Arrays are indexed, meaning each element of the array is obtained using a unique integer number.
    d. Arrays can contain primitive types as well as Java objects.
    e. All of the above.
2. Write a program with a method called `intInit()` that initializes an array of `int` to the values 1, 2, 3, 5, 7, 11, 13, 17, 19, 23, 27, 29 and then returns that array to the caller. Call your Java class `Unit10`, and use that class for all of the remaining coding exercises in this quiz.
3. Write a JUnit test case to test the `intInit()` method that verifies that the elements of your `int[]` are exactly as specified in Question 3. Use the element-access syntax (for example, `array[0]` to obtain the first element, and so on).
4. Enhance the `intInit()` method you wrote for Question 2 to print out the elements of the `int[]` using the shorthand `for` loop syntax.
5. Can you spot the error in the following code?

```
public void question5() {
  int[] intArray = new int[4];

  intArray[0] = 1;
  intArray[1] = 2;
  intArray[2] = Integer.valueOf(3);
  intArray[3] = Integer.MAX_VALUE;

}
```

    a. The `intArray` has too many values assigned to it. An exception will occur at runtime.
    b. Storing `Integer` objects and `int` in the same array is not allowed.
    c. The code contains no errors. Through auto-unboxing, the assignment of `Integer` object to the `int` array is achieved.
    d. None of the above.
6. Add a method to your `Unit10` class called `problem6()` that creates a `List` of the following:
    - `32`
    - `This is a string`
    - `Integer.valueOf(238)`
    - `-410`

- `null`

Write a JUnit test case to verify that your `List` was created in the correct order and contains the element values that you specified.

7. Which of these interfaces allows only one instance of a particular value?
   a. `List`
   b. `Map`
   c. `Set`
   d. All of the above
8. Which of these interfaces allows you to store a key/value pair using a unique key?
   a. `List`
   b. `Map`
   c. `Set`
   d. All of the above
9. Which of these interfaces is guaranteed to preserve the order of the items it contains?
   a. `List`
   b. `Map`
   c. `Set`
   d. All of the above

Check your answers.

For further exploration

> **Java Streams**
> The `java.util.stream` package makes it easy to run functional-style queries on collections, arrays, and other data sets. Explore the Streams library in this five-part series from developerWorks by Oracle's Java Language Architect, Brian Goetz.

5 things you didn't know about ... the Java Collections API, Part 1

5 things you didn't know about ... the Java Collections API, Part 2

The Java Tutorials: Collections

Collections framework overview

Java Collections cheat sheet from Rebel Labs

The Java Tutorials: Arrays

IBM Code: Java journeys

Previous: Loops
Next: Archiving Java code

Unit 11: Archiving Java code

Package your applications, and import other developers' code

J Steven Perry

September 14, 2016

Learn the structure, syntax, and programming paradigm of the Java platform and language. Start by mastering the essentials of object-oriented programming on the Java platform, and progress incrementally to the more-sophisticated syntax and libraries that you need to develop complex, real-world Java applications.

Before you begin

This unit is part of the "Intro to Java programming" learning path. Although the concepts discussed in the individual units are standalone in nature, the hands-on component builds as you progress through the units, and I recommend that you review the prerequisites, setup, and unit details before proceeding.

Unit objectives

- Know how to create a JAR file in Eclipse
- Be able to import third-party code into your Java applications

Creating JARs

Now that you've learned a bit about writing Java applications, you might be wondering how to package them up so that other developers can use them, or how to import other developers' code into your applications. This unit shows you how.

The JDK ships with a tool called JAR, which stands for Java Archive. You use this tool to create JAR files. After you package your code into a JAR file, other developers can drop the JAR file into their projects and configure their projects to use your code.

Creating a JAR file in Eclipse is easy. Watch as I take you through the steps.

To view this video, **Creating a JAR file in Eclipse** , please access the online version of the article. If this article is in the developerWorks archives, the video is no longer accessible.

Using third-party applications

The JDK is comprehensive, but it doesn't do everything you need for writing great Java code. As you grow more comfortable with writing Java applications, you might want to use more and more third-party applications to support your code. The Java open source community provides many libraries to help shore up these gaps.

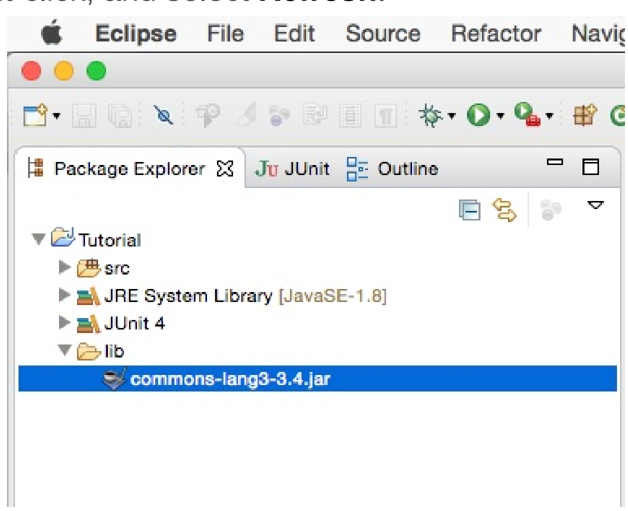
Suppose, for example, that you want to use Apache Commons Lang, a JDK replacement library for manipulating the core Java classes. The classes provided by Commons Lang help you manipulate arrays, create random numbers, and perform string manipulation.

Let's assume you've already downloaded Commons Lang, which is stored in a JAR file. To use the classes, your first step is to create a lib directory in your project and drop the JAR file into it:

1. Right-click the Intro root folder in the Eclipse Project Explorer view.
2. Click **New > Folder** and call the folder `lib`.
3. Click **Finish**.

The new folder shows up at the same level as src. Now copy the Commons Lang JAR file into your new lib directory. For this example, the file is called commons-lang3-3.4.jar. (It's common in naming a JAR file to include the version number, in this case 3.4.)

Now all you need to do is tell Eclipse to include the classes in the commons-lang3-3.4.jar file into your project:

1. In Package Explorer, select the lib folder, right-click, and select **Refresh**.
2. 

Verify that the JAR shows up in the lib folder:
3. Right-click commons-lang3-3.4 and choose **Build Path > Add to Build Path**.

After Eclipse processes the code (that is, the class files) in the JAR file, they're available to reference (import) from your Java code. Notice in Project Explorer that you have a new folder called Referenced Libraries that contains the commons-lang3-3.4.jar file.

For further exploration

JAR files revealed

5 things you didn't know about ... JARs

The Java Tutorials: Packaging Programs in JAR files

The Java Tutorials: Viewing the contents of a JAR file

Learning Java, 4th edition by Patrick Niemeyer and Daniel Leuck (see the "JAR files" section in Chapter 3).

IBM Code: Java journeys

Previous: Java Collections
Next: Writing good Java code

Unit 12: Writing good Java code

Strive to write clean, easily maintainable Java code

J Steven Perry September 14, 2016

Learn the structure, syntax, and programming paradigm of the Java platform and language. Start by mastering the essentials of object-oriented programming on the Java platform, and progress incrementally to the more-sophisticated syntax and libraries that you need to develop complex, real-world Java applications.

Before you begin

This unit is part of the "Intro to Java programming" learning path. Although the concepts discussed in the individual units are standalone in nature, the hands-on component builds as you progress through the units, and I recommend that you review the prerequisites, setup, and unit details before proceeding.

Unit objectives

- Follow best practices for class sizes, method sizes, and method names
- Understand the importance of refactoring
- Gain consistency in coding style and use of comments
- Use built-in logging

Best coding practices

You're now about halfway through this learning path. You have enough Java syntax under your belt to write basic Java programs. Before you continue on to more-advanced topics, this is a good moment to learn a few best coding practices. Read on for some essential pointers that can help you write cleaner, more maintainable Java code.

Keep classes small

So far you've created a few classes. After generating getter/setter pairs for even the small number (by the standards of a real-world Java class) of attributes, the `Person` class has 150 lines of code. At that size, `Person` is a small class. It's not uncommon (and it's unfortunate) to see classes with 50 or 100 methods and a thousand lines or more of source. Some classes might be that large out of necessity, but most likely they need to be *refactored*. Refactoring is changing the design of existing code without changing its results. I recommend that you follow this best practice.

In general, a class represents a conceptual entity in your application, and a class's size should reflect only the functionality to do whatever that entity needs to do. Keep your classes tightly focused to do a small number of things and do them well.

Keep only the methods that you need. If you need several helper methods that do essentially the same thing but take different parameters (such as the `printAudit()` method), that's a fine choice. But be sure to limit the list of methods to what you need, and no more.

Name methods carefully

A good coding pattern when it comes to method names is the *intention-revealing* method-names pattern. This pattern is easiest to understand with a simple example. Which of the following method names is easier to decipher at a glance?

- `a()`
- `computeInterest()`

The answer should be obvious, yet for some reason, programmers have a tendency to give methods (and variables, for that matter) small, abbreviated names. Certainly, a ridiculously long name can be inconvenient, but a name that conveys what a method does needn't be ridiculously long. Six months after you write a bunch of code, you might not remember what you meant to do with a method called `compInt()`, but it's obvious that a method called `computeInterest()`, well, probably computes interest.

Keep methods small

Small methods are as preferable as small classes, for similar reasons. One idiom I try to follow is to keep the size of a method to **one page** as I look at it on my screen. This practice makes my application classes more maintainable.

In the footsteps of Fowler
The best book in the industry (in my opinion, and I'm not alone) is *Refactoring: Improving the Design of Existing Code* by Martin Fowler et al. This book is even fun to read. The authors talk about "code smells" that beg for refactoring, and they go into great detail about the various techniques for fixing them.

If a method grows beyond one page, I refactor it. Eclipse has a wonderful set of refactoring tools. Usually, a long method contains subgroups of functionality bunched together. Take this functionality and move it to another method (naming it accordingly) and pass in parameters as needed.

Limit each method to a single job. I've found that a method doing only one thing well doesn't usually take more than about 30 lines of code.

Refactoring and the ability to write test-first code are the most important skills for new programmers to learn. If everybody were good at both, it would revolutionize the industry. If you become good at both, you will ultimately produce cleaner code and more-functional applications than many of your peers.

Use comments

Please, use comments. The people who follow along behind you (or even you, yourself, six months down the road) will thank you. You might have heard the old adage *Well-written code is self-documenting, so who needs comments?* I'll give you two reasons why I believe this adage is false:

- Most code is not well written.
- Try as we might, our code probably isn't as well written as we'd like to think.

So, comment your code. Period.

Use a consistent style

Coding style is a matter of personal preference, but I advise you to use standard Java syntax for braces:

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

Don't use this style:

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

Or this one:

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

Why? Well, it's standard, so most code you run across (as in, code you didn't write but might be paid to maintain) will most likely be written that way. Eclipse **does** allow you to define code styles and format your code any way you like. But, being new to Java, you probably don't have a style yet. So I suggest you adopt the Java standard from the start.

Use built-in logging

Before Java 1.4 introduced built-in logging, the canonical way to find out what your program was doing was to make a system call like this one:

```
public void someMethod() {
  // Do some stuff...
  // Now tell all about it
  System.out.println("Telling you all about it:");
  // Etc...
}
```

The Java language's built-in logging facility (refer back to Unit 5: Your first Java class) is a better alternative. I **never** use `System.out.println()` in my code, and I suggest you don't use it either. Another alternative is the commonly used log4j replacement library, part of the Apache umbrella project.

For further exploration

Java - Basic Syntax

Google Java Style Guide

Java Coding Guidelines

Speaking the Java language without an accent

Refactoring: Improving the Design of Existing Code by Martin Fowler et al.

Effective Java by Joshua Bloch

IBM Code: Java journeys

Previous: Archiving Java code
Next: Next steps with objects

Unit 13: Next steps with objects

Add versatility to your classes and methods

J Steven Perry

September 14, 2016

Learn the structure, syntax, and programming paradigm of the Java platform and language. Start by mastering the essentials of object-oriented programming on the Java platform, and progress incrementally to the more-sophisticated syntax and libraries that you need to develop complex, real-world Java applications.

Before you begin

This unit is part of the "Intro to Java programming" learning path. Although the concepts discussed in the individual units are standalone in nature, the hands-on component builds as you progress through the units, and I recommend that you review the prerequisites, setup, and unit details before proceeding.

Unit objectives

- Learn about method overloading and overriding
- Be able to compare one object with another
- Understand how and when to use class variable and methods

Overloading methods

It's time to return to the `Person` class. `Person` is reasonably useful at this point, but not as useful as it could be. You'll start enhancing `Person` by *overloading* one of its methods.

When you create two methods with the same name but with different argument lists (that is, different numbers or types of parameters), you have an *overloaded* method. At runtime, the JRE decides which variation of your overloaded method to call, based on the arguments that were passed to it.

Suppose that `Person` needs a couple of methods to print an audit of its current state. I call both of those methods `printAudit()`. Paste the overloaded method in Listing 1 into the Eclipse editor view in the `Person` class:

Listing 1. `printAudit()`: An overloaded method

```
public void printAudit(StringBuilder buffer) {
   buffer.append("Name=");
   buffer.append(getName());
   buffer.append(",");
   buffer.append("Age=");
   buffer.append(getAge());
   buffer.append(",");
   buffer.append("Height=");
   buffer.append(getHeight());
   buffer.append(",");
   buffer.append("Weight=");
   buffer.append(getWeight());
   buffer.append(",");
   buffer.append("EyeColor=");
   buffer.append(getEyeColor());
   buffer.append(",");
   buffer.append("Gender=");
   buffer.append(getGender());
}

public void printAudit(Logger l) {
   StringBuilder sb = new StringBuilder();
   printAudit(sb);
   l.info(sb.toString());
}
```

You have two overloaded versions of `printAudit()`, and one even uses the other. By providing two versions, you give the caller a choice of how to print an audit of the class. Depending on the parameters that are passed, the Java runtime calls the correct method.

Remember **two important rules** when you use overloaded methods:

- You can't overload a method just by changing its return type.
- You can't have two same-named methods with the same parameter list.

If you violate these rules, the compiler gives you an error.

Overriding methods

When a subclass provides its own implementation of a method that's defined on one of its parent classes, that's called *method overriding*. To see how method overriding is useful, you need to do some work on an `Employee` class. Watch the following video to see how to set up the `Employee` class and perform method overriding in that class. After you watch, I'll briefly recap those steps while giving you a closer look at the code.

The video also includes a demo of overriding the `equals()` method and autogenerating `equals()` and `hashCode()`, which I cover in detail in this unit's "Comparing objects" section.

To view this video, **Next steps with objects** , please access the online version of the article. If this article is in the developerWorks archives, the video is no longer accessible.

Employee: A subclass of Person

Recall from Unit 3: Object-oriented concepts and principles that an `Employee` class could be a subclass (or *child*) of `Person` that has additional attributes such as taxpayer identification number, employee number, hire date, and salary. You'll declare the `Employee` class now and add some of those attributes in a moment.

To declare the `Employee` class, right-click the `com.makotojava.intro` package in Eclipse. Click **New > Class...** and, in the New Java Class dialog box, enter `Employee` as the name of the class and `Person` as its superclass. Click **Finish**, and you can see the `Employee` class code in an edit window.

You don't explicitly need to declare a constructor, but go ahead and implement both constructors anyway. With the `Employee` class edit window having the focus, go to **Source > Generate Constructors from Superclass...**. In the Generate Constructors from Superclass dialog box, select both constructors and click **OK**. You now have an `Employee` class like the one in Listing 2.

Listing 2. The `Employee` class

```
package com.makotojava.intro;

public class Employee extends Person {

  public Employee() {
    super();
    // TODO Auto-generated constructor stub
  }

  public Employee(String name, int age, int height, int weight,
  String eyeColor, String gender) {
    super(name, age, height, weight, eyeColor, gender);
    // TODO Auto-generated constructor stub
  }

}
```

Employee as a child of Person

`Employee` inherits the attributes and behavior of its parent, `Person`. Add some attributes of `Employee`'s own, as shown in lines 7 through 9 of Listing 3.

Listing 3. The `Employee` class with `Person`'s attributes

```
package com.makotojava.intro;

import java.math.BigDecimal;

public class Employee extends Person {

  private String taxpayerIdentificationNumber;
  private String employeeNumber;
  private BigDecimal salary;

  public Employee() {
    super();
  }
  public String getTaxpayerIdentificationNumber() {
    return taxpayerIdentificationNumber;
  }
  public void setTaxpayerIdentificationNumber(String taxpayerIdentificationNumber) {
    this.taxpayerIdentificationNumber = taxpayerIdentificationNumber;
  }
```

```
  // Other getter/setters...
}
```

Don't forget to generate getters and setters for the new attributes, as you did for `Person` in Unit 5: Your first Java class.

Overriding the printAudit() method

Now you'll override the `printAudit()` method (see Listing 1) that you used to format the current state of a `Person` instance. `Employee` inherits that behavior from `Person`. If you instantiate `Employee`, set its attributes, and call either of the overloads of `printAudit()`, the call succeeds. However, the audit that's produced doesn't fully represent an `Employee`. The `printAudit()` method can't format the attributes specific to an `Employee`, because `Person` doesn't know about them.

The solution is to override the overload of `printAudit()` that takes a `StringBuilder` as a parameter and add code to print the attributes specific to `Employee`.

With `Employee` open in the editor window or selected in the Project Explorer view, go to **Source > Override/Implement Methods...**. In the Override/Implement Methods dialog box, select the `StringBuilder` overload of `printAudit()` and click **OK**. Eclipse generates the method stub for you, and then you can fill in the rest, like so:

```
@Override
public void printAudit(StringBuilder buffer) {
  // Call the superclass version of this method first to get its attribute values
  super.printAudit(buffer);

  // Now format this instance's values
  buffer.append("TaxpayerIdentificationNumber=");
  buffer.append(getTaxpayerIdentificationNumber());
  buffer.append(","); buffer.append("EmployeeNumber=");
  buffer.append(getEmployeeNumber());
  buffer.append(","); buffer.append("Salary=");
  buffer.append(getSalary().setScale(2).toPlainString());
}
```

Notice the call to `super.printAudit()`. What you're doing here is asking the (`Person`) superclass to exhibit its behavior for `printAudit()`, and then you augment it with `Employee`-type `printAudit()` behavior.

The call to `super.printAudit()` doesn't need to be first; it just seemed like a good idea to print those attributes first. In fact, you don't need to call `super.printAudit()` at all. If you don't call it, you must format the attributes from `Person` yourself in the `Employee.printAudit()` method, or they won't be included in the audit output.

Comparing objects

The Java language provides two ways to compare objects:

- The `==` operator
- The `equals()` method

Comparing objects with ==

The `==` syntax compares objects for equality such that `a == b` returns `true` only if `a` and `b` have the same value. For objects, this will be the case if the two refer to the *same object instance*. For primitives, if the *values are identical*.

Suppose you generate a JUnit test for `Employee` (which you saw how to do in Unit 5: Your first Java class. The JUnit test is shown in Listing 4.

Listing 4. Comparing objects with `==`

```
public class EmployeeTest {
  @Test
  public void test() {
    int int1 = 1;
    int int2 = 1;
    Logger l = Logger.getLogger(EmployeeTest.class.getName());

    l.info("Q: int1 == int2?           A: " + (int1 == int2));
    Integer integer1 = Integer.valueOf(int1);
    Integer integer2 = Integer.valueOf(int2);
    l.info("Q: Integer1 == Integer2?   A: " + (integer1 == integer2));
    integer1 = new Integer(int1);
    integer2 = new Integer(int2);
    l.info("Q: Integer1 == Integer2?   A: " + (integer1 == integer2));
    Employee employee1 = new Employee();
    Employee employee2 = new Employee();
    l.info("Q: Employee1 == Employee2? A: " + (employee1 == employee2));
  }
}
```

Run the Listing 4 code inside Eclipse (select `Employee` in the Project Explorer view, then choose **Run As > JUnit Test**) to generate the following output:

```
Sep 18, 2015 5:09:56 PM com.makotojava.intro.EmployeeTest test
INFO: Q: int1 == int2?           A: true
Sep 18, 2015 5:09:56 PM com.makotojava.intro.EmployeeTest test
INFO: Q: Integer1 == Integer2?   A: true
Sep 18, 2015 5:09:56 PM com.makotojava.intro.EmployeeTest test
INFO: Q: Integer1 == Integer2?   A: false
Sep 18, 2015 5:09:56 PM com.makotojava.intro.EmployeeTest test
INFO: Q: Employee1 == Employee2? A: false
```

In the first case in Listing 4, the values of the primitives are the same, so the `==` operator returns `true`. In the second case, the `Integer` objects refer to the same instance, so again `==` returns `true`. In the third case, even though the `Integer` objects wrap the same value, `==` returns `false` because `integer1` and `integer2` refer to different objects. Think of `==` as a test for "same object instance."

Comparing objects with equals()

`equals()` is a method that every Java language object gets for free, because it's defined as an instance method of `java.lang.Object` (which every Java object inherits from).

You call `equals()` like this:

```
a.equals(b);
```

This statement invokes the `equals()` method of object `a`, passing to it a reference to object `b`. By default, a Java program would simply check to see if the two objects are the same by using the `==` syntax. Because `equals()` is a method, however, it can be overridden. Compare the JUnit test case in Listing 4 to the one in Listing 5 (which I've called `anotherTest()`), which uses `equals()` to compare the two objects.

Listing 5. Comparing objects with `equals()`

```
@Test
public void anotherTest() {
  Logger l = Logger.getLogger(Employee.class.getName());
  Integer integer1 = Integer.valueOf(1);
  Integer integer2 = Integer.valueOf(1);
  l.info("Q: integer1 == integer2 ? A: " + (integer1 == integer2));
  l.info("Q: integer1.equals(integer2) ? A: " + integer1.equals(integer2));
  integer1 = new Integer(integer1);
  integer2 = new Integer(integer2);
  l.info("Q: integer1 == integer2 ? A: " + (integer1 == integer2));
  l.info("Q: integer1.equals(integer2) ? A: " + integer1.equals(integer2));
  Employee employee1 = new Employee();
  Employee employee2 = new Employee();
  l.info("Q: employee1 == employee2 ? A: " + (employee1 == employee2));
  l.info("Q: employee1.equals(employee2) ? A : " + employee1.equals(employee2));
}
```

Running the Listing 5 code produces this output:

```
Sep 19, 2015 10:11:57 AM com.makotojava.intro.EmployeeTest anotherTest
INFO: Q: integer1 == integer2 ? A: true
Sep 19, 2015 10:11:57 AM com.makotojava.intro.EmployeeTest anotherTest
INFO: Q: integer1.equals(integer2) ? A: true
Sep 19, 2015 10:11:57 AM com.makotojava.intro.EmployeeTest anotherTest
INFO: Q: integer1 == integer2 ? A: false
Sep 19, 2015 10:11:57 AM com.makotojava.intro.EmployeeTest anotherTest
INFO: Q: integer1.equals(integer2) ? A: true
Sep 19, 2015 10:11:57 AM com.makotojava.intro.EmployeeTest anotherTest
INFO: Q: employee1 == employee2 ? A: false
Sep 19, 2015 10:11:57 AM com.makotojava.intro.EmployeeTest anotherTest
INFO: Q: employee1.equals(employee2) ? A : false
```

A note about comparing Integers

In Listing 5, it should be no surprise that the `equals()` method of `Integer` returns `true` if `==` returns `true`. But notice what happens in the second case, where you create separate objects that both wrap the value `1`: `==` returns `false` because `integer1` and `integer2` refer to different objects; but `equals()` returns `true`.

The writers of the JDK decided that for `Integer`, the meaning of `equals()` would be different from the default (which, as you recall, is to compare the object references to see if they refer to the same object). For `Integer`, `equals()` returns `true` in cases in which the underlying (boxed) `int` value is the same.

For `Employee`, you didn't override `equals()`, so the default behavior (of using `==`) returns what you'd expect, because `employee1` and `employee2` refer to different objects.

For any object you write, then, you can define what `equals()` means as is appropriate for the application you're writing.

Overriding equals()

You can define what `equals()` means to your application's objects by overriding the default behavior of `Object.equals()`— and you can do this in Eclipse. With `Employee` having the focus in the IDE's source window, select **Source > Override/Implement Methods**.You want to implement the `Object.equals()` superclass method. So, find `Object` in the list of methods to override or implement, select the `equals(Object)` method, and click **OK**. Eclipse generates the correct code and places it in your source file.

It makes sense that the two `Employee` objects are equal if the states of those objects are equal. That is, they're equal if their values — name, and age — are the same.

Autogenerating equals()

Eclipse can generate an `equals()` method for you based on the instance variables (attributes) that you define for a class. Because `Employee` is a subclass of `Person`, you first generate `equals()` for `Person`. In the Eclipse Project Explorer view, right-click `Person` and choose **Generate hashCode() and equals()**. In the dialog box that opens, click **Select All** to include all of the attributes in the `hashCode()` and `equals()` methods, and click **OK**. Eclipse generates an `equals()` method that looks like the one in Listing 6.

Listing 6. An `equals()` method generated by Eclipse

```
@Override
public boolean equals(Object obj) {
  if (this == obj)
    return true;
  if (obj == null)
    return false;
  if (getClass() != obj.getClass())
    return false;
  Person other = (Person) obj;
  if (age != other.age)
    return false;
  if (eyeColor == null) {
    if (other.eyeColor != null)
      return false;
  } else if (!eyeColor.equals(other.eyeColor))
    return false;
  if (gender == null) {
    if (other.gender != null)
      return false;
  } else if (!gender.equals(other.gender))
    return false;
  if (height != other.height)
    return false;
  if (name == null) {
    if (other.name != null)
      return false;
  } else if (!name.equals(other.name))
    return false;
  if (weight != other.weight)
    return false;
  return true;
```

```
}
```

The `equals()` method generated by Eclipse looks complicated, but what it does is simple: If the object passed in is the same object as the one in Listing 6, `equals()` returns `true`. If the object passed in is null (meaning missing), it returns `false`.

Next, the method checks to see if the `Class` objects are the same (meaning that the passed-in object must be a `Person` object). If they are the same, each attribute value of the object passed in is checked to see if it matches value-for-value with the state of the given `Person` instance. If the attribute values are null, the `equals()` checks as many as it can, and if those match, the objects are considered equal. You might not want this behavior for every program, but it works for most purposes.

Exercises

Now, work through a couple of guided exercises to do even more with `Person` and `Employee` in Eclipse.

Exercise 1: Generate an equals() for Employee

Try following the steps in "Autogenerating `equals()`" to generate an `equals()` for `Employee`. Once you have your generated `equals()`, add the following JUnit test case (which I've called `yetAnotherTest()`) to it:

```
@Test
public void yetAnotherTest() {
  Logger l = Logger.getLogger(Employee.class.getName());
  Employee employee1 = new Employee();
  employee1.setName("J Smith");
  Employee employee2 = new Employee();
  employee2.setName("J Smith");
  l.info("Q: employee1 == employee2?      A: " + (employee1 == employee2));
  l.info("Q: employee1.equals(employee2)? A: " + employee1.equals(employee2));
}
```

If you run the code, you should see the following output:

```
Sep 19, 2015 11:27:23 AM com.makotojava.intro.EmployeeTest yetAnotherTest
INFO: Q: employee1 == employee2?      A: false
Sep 19, 2015 11:27:23 AM com.makotojava.intro.EmployeeTest yetAnotherTest
INFO: Q: employee1.equals(employee2)? A: true
```

In this case, a match on `Name` alone was enough to convince `equals()` that the two objects are equal. Try adding more attributes to this example and see what you get.

Exercise 2: Override toString()

Remember the `printAudit()` method from the beginning of this unit? If you thought it was working a little too hard, you were right. Formatting the state of an object into a `String` is such a common pattern that the designers of the Java language built it into `Object` itself, in a method called (no surprise) `toString()`. The default implementation of `toString()` isn't especially useful, but every object has one. In this exercise, you override `toString()` to make it a little more useful.

If you suspect that Eclipse can generate a `toString()` method for you, you're correct. Go back into your Project Explorer and right-click the `Person` class, then choose **Source > Generate toString()...**. In the dialog box, select all attributes and click **OK**. Now do the same thing for `Employee`. The code generated by Eclipse for `Employee` is shown in Listing 7.

Listing 7. A `toString()` method generated by Eclipse

```
@Override
public String toString() {
  return "Employee [taxpayerIdentificationNumber=" + taxpayerIdentificationNumber + ",
      employeeNumber=" + employeeNumber + ", salary=" + salary + "]";
}
```

The code that Eclipse generates for `toString` doesn't include the superclass's `toString()` (`Employee`'s superclass being `Person`). You can fix that situation quickly, using Eclipse, with this override:

```
@Override
public String toString() {
  return super.toString() + "Employee [taxpayerIdentificationNumber=" + taxpayerIdentificationNumber +
    ", employeeNumber=" + employeeNumber + ", salary=" + salary + "]";
}
```

The addition of `toString()` makes `printAudit()` much simpler:

```
@Override
  public void printAudit(StringBuilder buffer) {
  buffer.append(toString());
}
```

`toString()` now does the heavy lifting of formatting the object's current state, and you simply stuff what it returns into the `StringBuilder` and return.

I recommend always implementing `toString()` in your classes, if only for support purposes. It's virtually inevitable that at some point, you'll want to see what an object's state is while your application is running, and `toString()` is a great hook for doing that.

Class members

Every object instance has variables and methods, and for each one, the exact behavior is different, because it's based on the state of the object instance. The variables and methods that you have on `Person` and `Employee` are *instance* variables and methods. To use them, you must either instantiate the class you need or have a reference to the instance.

Classes can also have *class* variables and methods — known as *class members*. You declare class variables with the `static` keyword. The differences between class variables and instance variables are:

- Every instance of a class shares a single copy of a class variable.
- You can call class methods on the class itself, without having an instance.
- Class methods can access only class variables.

- Instance methods can access class variables, but class methods can't access instance variables.

When does it make sense to add class variables and methods? The best rule of thumb is to do so rarely, so that you don't overuse them. That said, it's a good idea to use class variables and methods:

- To declare constants that any instance of the class can use (and whose value is fixed at development time)
- On a class with utility methods that don't ever need an instance of the class (such as `Logger.getLogger()`)

Class variables

To create a class variable, use the `static` keyword when you declare it:

```
accessSpecifier static variableName [= initialValue];
```

Note: The square brackets here indicate that their contents are optional. The brackets are not part of the declaration syntax.

The JRE creates space in memory to store each of a class's *instance* variables for every instance of that class. In contrast, the JRE creates only a single copy of each *class* variable, regardless of the number of instances. It does so the first time the class is loaded (that is, the first time it encounters the class in a program). All instances of the class share that single copy of the variable. That makes class variables a good choice for constants that all instances should be able to use.

For example, you declared the `Gender` attribute of `Person` to be a `String`, but you didn't put any constraints on it. Listing 8 shows a common use of class variables.

Listing 8. Using class variables

```
public class Person {
  //. . .
  public static final String GENDER_MALE = "MALE";
  public static final String GENDER_FEMALE = "FEMALE";

  // . . .
  public static void main(String[] args) {
  Person p = new Person("Joe Q Author", 42, 173, 82, "Brown", GENDER_MALE);
    // . . .
  }
  //. . .
}
```

Declaring constants

Typically, constants are:

- Named in all uppercase

- Named as multiple words, separated by underscores
- Declared `final` (so that their values cannot be modified)
- Declared with a `public` access specifier (so that they can be accessed by other classes that need to reference their values by name)

In Listing 8, to use the constant for `MALE` in the `Person` constructor call, you would simply reference its name. To use a constant outside of the class, you would preface it with the name of the class where it was declared:

```
String genderValue = Person.GENDER_MALE;
```

Class methods

You've already called the static `Logger.getLogger()` method several times — whenever you retrieved a `Logger` instance to write output to the console. Notice, though, that to do so you didn't need an instance of `Logger`. Instead, you referenced the `Logger` class, which is the syntax for making a *class method* call. As with class variables, the `static` keyword identifies `Logger` (in this example) as a class method. Class methods are also sometimes called *static methods* for this reason.

Now you can combine what you learned about static variables and methods to create a static method on `Employee`. You declare a `private static final` variable to hold a `Logger`, which all instances share, and which is accessible by calling `getLogger()` on the `Employee` class. Listing 9 shows how.

Listing 9. Creating a class (or static) method

```
public class Employee extends Person {
  private static final Logger logger = Logger.getLogger(Employee.class.getName());
  //. . .
  public static Logger getLogger() {
    return logger;
  }

}
```

Two important things are happening in Listing 9:

- The `Logger` instance is declared with `private` access, so no class outside `Employee` can access the reference directly.
- The `Logger` is initialized when the class is loaded — because you use the Java initializer syntax to give it a value.

To retrieve the `Employee` class's `Logger` object, you make the following call:

```
Logger employeeLogger = Employee.getLogger();
```

For further exploration

The Java Tutorials: Overriding and Hiding Methods

Java - Overriding

The Java Tutorials: Defining Methods

What is the difference between `==` vs `equals()` in Java?

Java - `toString()` method

How to use the `toString()` method in Java?

IBM Code: Java journeys

Previous: Writing good Java code
Next: Exceptions

Unit 14: Exceptions

Use built-in Java platform mechanisms to handle errors in your code

J Steven Perry

September 14, 2016

Learn the structure, syntax, and programming paradigm of the Java platform and language. Start by mastering the essentials of object-oriented programming on the Java platform, and progress incrementally to the more-sophisticated syntax and libraries that you need to develop complex, real-world Java applications.

Before you begin

This unit is part of the "Intro to Java programming" learning path. Although the concepts discussed in the individual units are standalone in nature, the hands-on component builds as you progress through the units, and I recommend that you review the prerequisites, setup, and unit details before proceeding.

Unit objectives

- Learn exception-handling basics
- Understand the exception hierarchy and how to use multiple `catch` blocks

Exception-handling basics

No program ever works 100 percent of the time, and the designers of the Java language knew this. The Java platform has built-in mechanisms for handling situations in which your code doesn't work exactly as planned.

An *exception* is an event that occurs during program execution that disrupts the normal flow of the program's instructions. Exception handling is an essential technique of Java programming. You wrap your code in a `try` block (which means "try this and let me know if it causes an exception") and use it to `catch` various types of exceptions.

To get started with exception handling, take a look at the code in Listing 1.

Listing 1. Do you see the error?

```
@Test
public void yetAnotherTest() {
  Logger l = Logger.getLogger(Employee.class.getName());
//    Employee employee1 = new Employee();
  Employee employee1 = null;
  employee1.setName("J Smith");
  Employee employee2 = new Employee();
  employee2.setName("J Smith");
  l.info("Q: employee1 == employee2?      A: " + (employee1 == employee2));
  l.info("Q: employee1.equals(employee2)? A: " + employee1.equals(employee2));
}
```

Notice that the `Employee` reference is set to `null`. Run this code and you get the following output:

```
java.lang.NullPointerException
  at com.makotojava.intro.EmployeeTest.yetAnotherTest(EmployeeTest.java:49)
  .
  .
  .
```

This output is telling you that you're trying to reference an object through a `null` reference (pointer), which is a pretty serious development error. (You probably noticed that Eclipse warns you of the potential error with the message: `Null pointer access: The variable employee1 can only be null at this location`. Eclipse warns you about many potential development mistakes — yet another advantage of using an IDE for Java development.)

Fortunately, you can use `try` and `catch` blocks (along with a little help from `finally`) to catch the error.

Using try, catch, and finally

Listing 2 shows the buggy code from Listing 1 cleaned up with the standard code blocks for exception handling: `try`, `catch`, and `finally`.

Listing 2. Catching an exception

```
@Test
public void yetAnotherTest() {
  Logger l = Logger.getLogger(Employee.class.getName());

  //    Employee employee1 = new Employee();
  try {
    Employee employee1 = null;
    employee1.setName("J Smith");
    Employee employee2 = new Employee();
    employee2.setName("J Smith");
    l.info("Q: employee1 == employee2?      A: " + (employee1 == employee2));
    l.info("Q: employee1.equals(employee2)? A: " + employee1.equals(employee2));
  } catch (Exception e) {
    l.severe("Caught exception: " + e.getMessage());
  } finally {
    // Always executes
  }
}
```

Together, the `try`, `catch`, and `finally` blocks form a net for catching exceptions. First, the `try` statement wraps code that might throw an exception. In that case, execution drops immediately to the `catch` block, or *exception handler*. When all the trying and catching is done, execution continues to the `finally` block, whether or not an exception occurred. When you catch an exception, you can try to recover gracefully from it, or you can exit the program (or method).

In Listing 2, the program recovers from the error and then prints out the exception's message:

```
Sep 19, 2015 2:01:22 PM com.makotojava.intro.EmployeeTest yetAnotherTest
SEVERE: Caught exception: null
```

The exception hierarchy

The Java language incorporates an entire exception hierarchy consisting of many types of exceptions grouped into two major categories:

- **Checked exceptions** are checked by the compiler (meaning the compiler makes sure that they get handled somewhere in your code). In general, these are direct subclasses of `java.lang.Exception`.
- **Unchecked exceptions** (also called *runtime exceptions*) are not checked by the compiler. These are subclasses of `java.lang.RuntimeException`.

When a program causes an exception, you say that it *throws* the exception. A checked exception is declared to the compiler by any method with the `throws` keyword in its method signature. Next is a comma-separated list of exceptions that the method could potentially throw during its execution. If your code calls a method that specifies that it throws one or more types of exceptions, you must handle it somehow, or add a `throws` to your method signature to pass that exception type along.

When an exception occurs, the Java runtime searches for an exception handler somewhere up the stack. If it doesn't find one by the time it reaches the top of the stack, it halts the program abruptly, as you saw in Listing 1.

Multiple catch blocks

You can have multiple `catch` blocks, but they must be structured in a particular way. If any exceptions are subclasses of other exceptions, the child classes are placed ahead of the parent classes in the order of the `catch` blocks. Listing 3 shows an example of different exception types structured in their correct hierarchical sequence.

Listing 3. Exception hierarchy example

```
@Test
public void exceptionTest() {
  Logger l = Logger.getLogger(Employee.class.getName());
  File file = new File("file.txt");
  BufferedReader bufferedReader = null;
  try {
    bufferedReader = new BufferedReader(new FileReader(file));
    String line = bufferedReader.readLine();
    while (line != null) {
      // Read the file
    }
  } catch (FileNotFoundException e) {
    l.severe(e.getMessage());
  } catch (IOException e) {
    l.severe(e.getMessage());
```

```
  } catch (Exception e) {
    l.severe(e.getMessage());
  } finally {
    // Close the reader
  }
}
```

In this example, the `FileNotFoundException` is a child class of `IOException`, so it must be placed ahead of the `IOException catch` block. And `IOException` is a child class of `Exception`, so it must be placed ahead of the `Exception catch` block.

try-with-resources blocks

The code in Listing 3 must declare a variable to hold the `bufferedReader` reference, and then in the `finally` must close the `BufferedReader`.

Alternative, more-compact syntax (available as of JDK 7) automatically closes resources when the `try` block goes out of scope. Listing 4 shows this newer syntax.

Listing 4. Resource-management syntax

```
@Test
public void exceptionTestTryWithResources() {
  Logger l = Logger.getLogger(Employee.class.getName());
  File file = new File("file.txt");
    try (BufferedReader bufferedReader = new BufferedReader(new FileReader(file))) {
      String line = bufferedReader.readLine();
      while (line != null) {
// Read the file
      }
    } catch (Exception e) {
      l.severe(e.getMessage());
    }
}
```

Essentially, you assign resource variables after `try` inside parentheses, and when the `try` block goes out of scope, those resources are automatically closed. The resources must implement the `java.lang.AutoCloseable` interface; if you try to use this syntax on a resource class that doesn't, Eclipse warns you.

Test your understanding

1. Which of these statements is true regarding exceptions?
    a. An exception is an abnormal flow of execution in a Java program.
    b. A checked exception is not checked by the compiler, so you must check for the exception yourself.
    c. You must specify all runtime exceptions on your method signature by using the `throws` keyword.
    d. None of the above.
    e. All of the above.
2. The `finally` block executes every time, whether or not an exception was thrown.
    a. True
    b. False
3. Will the following code compile? Explain your answer.

```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.logging.Logger;

public class Test1 {

   private static final Logger l = Logger.getLogger(Test1.class.getName());

   public void hierarchyExample() {
      File file = new File("file.txt");

      BufferedReader bufferedReader = null;

      try {
         bufferedReader = new BufferedReader(new FileReader(file));
         String line = bufferedReader.readLine();
         while (line != null) {
            // Read the line
         }
      } catch (IOException e) {
         l.severe(e.getMessage());
      } catch (FileNotFoundException e) {

         l.severe(e.getMessage());
      } catch (Exception e) {
         l.severe(e.getMessage());
      }
   }

}
```

4. A class used in a try-with-resources block must implement which of these interfaces?
   a. `java.lang.Iterable`
   b. `java.lang.AutoClosable`
   c. Both `java.lang.Iterable` and `java.io.Closable`
   d. Both `java.lang.AutoCloseable` and `java.lang.Exception`
   e. None of the above.
5. An unchecked exception does not have to be specified to the compiler, and may be thrown at runtime.
   a. True
   b. False
6. Which of these `Exception` subclasses are unchecked exceptions? Use the resources in the "For further exploration" section, along with the JDK Javadoc, for help with your answer.
   a. `java.lang.IllegalArgumentExeption`
   b. `java.sql.SQLException`
   c. `java.time.DateTimeException`
   d. `java.lang.IllegalArgumentException`
   e. All of the above.
7. What special considerations must you use if a checked exception is thrown during the execution of a method you've written?
   a. Ensure that the exception is declared using the `throws` keyword in your method's signature, or catch and handle the exception in your method.
   b. None. Checked exceptions are only checked by the runtime, not the compiler.

   c. Be sure to use a `finally` block in all affected code.
   d. There's no such thing as a checked exception.
   e. None of the above.
8. What tells the compiler that an exception is an unchecked exception?
   a. There's no way for the compiler to know if an exception is unchecked, which makes exceptions such a difficult topic in Java.
   b. An unchecked exception is a direct subclass of `java.lang.Exception`.
   c. There's no such thing as an unchecked exception.
   d. The compiler does not check for unchecked exceptions.
   e. None of the above.

Check your answers.

For further exploration

The Java Tutorials: Exceptions

Java - Exceptions

The Java Tutorials: Unchecked Exceptions — The Controversy

Java: checked vs unchecked exception explanation

IBM Code: Java journeys

Previous: Next steps with objects
Next: Building Java applications

Unit 15: Building Java applications

Create an application from a collection of objects

J Steven Perry

September 14, 2016

Learn the structure, syntax, and programming paradigm of the Java platform and language. Start by mastering the essentials of object-oriented programming on the Java platform, and progress incrementally to the more-sophisticated syntax and libraries that you need to develop complex, real-world Java applications.

Before you begin

This unit is part of the "Intro to Java programming" learning path. Although the concepts discussed in the individual units are standalone in nature, the hands-on component builds as you progress through the units, and I recommend that you review the prerequisites, setup, and unit details before proceeding.

Unit objectives

- Learn how an object, or collection of objects, becomes an application
- Use Eclipse to create a driver class

The application entry point

All Java applications need an entry point where the Java runtime knows to start executing code. That entry point is the `main()` method. Domain objects — that is, objects (`Person` and `Employee`, for example) that are part of your application's *business domain*— typically don't have `main()` methods, but at least one class in every application must.

As you know, `Person` and its `Employee` subclass are conceptually part of a human-resources application. Now you'll add a new class to the application to give it an entry point.

Creating a driver class

The purpose of a *driver class* (as its name implies) is to "drive" an application. Notice that this simple driver for a human-resources application contains a `main()` method:

```
package com.makotojava.intro;
public class HumanResourcesApplication {
  public static void main(String[] args) {
  }
}
```

Now, create a driver class in Eclipse using the same procedure you used to create `Person` and `Employee`. Name the class `HumanResourcesApplication`, being sure to select the option to add a `main()` method to the class. Eclipse will generate the class for you.

Next, add some code to your new `main()` method so that it looks like this:

```
package com.makotojava.intro;
import java.util.logging.Logger;

public class HumanResourcesApplication {
  private static final Logger log = Logger.getLogger(HumanResourcesApplication.class.getName());
  public static void main(String[] args) {
    Employee e = new Employee();
    e.setName("J Smith");
    e.setEmployeeNumber("0001");
    e.setTaxpayerIdentificationNumber("123-45-6789");
    e.setSalary(BigDecimal.valueOf(45000.0));
    e.printAudit(log);
  }
}
```

Finally, launch the `HumanResourcesApplication` class and watch it run. You should see this output:

```
Sep 19, 2015 7:59:37 PM com.makotojava.intro.Person printAudit
INFO: Name=J Smith,Age=0,Height=0,Weight=0,EyeColor=null,Gender=null
 TaxpayerIdentificationNumber=123-45-6789,EmployeeNumber=0001,Salary=45000.00
```

That's all there is to creating a simple Java application. In the next unit, you begin looking at some of the syntax and libraries that can help you develop more-complex applications.

For further exploration

The Java Tutorials: A Closer Look at the "Hello, World!" Application

Why is the Java main method static?

Why main method is declared public static in Java

IBM Code: Java journeys

Previous: Exceptions
Next: Inheritance

developerWorks®

Unit 16: Inheritance

Enhance code reuse by deriving classes from other classes

J Steven Perry

September 14, 2016

Learn the structure, syntax, and programming paradigm of the Java platform and language. Start by mastering the essentials of object-oriented programming on the Java platform, and progress incrementally to the more-sophisticated syntax and libraries that you need to develop complex, real-world Java applications.

Before you begin

This unit is part of the "Intro to Java programming" learning path. Although the concepts discussed in the individual units are standalone in nature, the hands-on component builds as you progress through the units, and I recommend that you review the prerequisites, setup, and unit details before proceeding.

Unit objectives

- Understand the concept of a class hierarchy
- Know the various ways to implement a constructor
- Learn when and why to use abstract classes and methods
- Know how to assign a reference from one class to a variable of a type belonging to another class.

How inheritance works

Classes in Java code exist in hierarchies. Classes above a given class in a hierarchy are *superclasses* of that class. That particular class is a *subclass* of every class higher up the hierarchy. A subclass inherits from its superclasses. The `java.lang.Object` class is at the top of the class hierarchy — so every Java class is a subclass of, and inherits from, `Object`.

For example, suppose you have a `Person` class that looks like the one in Listing 1.

Listing 1. Public `Person` class

```
public class Person {

  public static final String STATE_DELIMITER = "~";

  public Person() {
    // Default constructor
```

```
  }

  public enum Gender {
    MALE,
    FEMALE,
    UNKNOWN
  }

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


  private String name;
  private int age;
  private int height;
  private int weight;
  private String eyeColor;
  private Gender gender;
```

The `Person` class in Listing 1 implicitly inherits from `Object`. Because inheriting from `Object` is assumed for every class, you don't need to type `extends Object` for every class you define. But what does it mean to say that a class inherits from its superclass? It simply means that `Person` has access to the exposed variables and methods in its superclasses. In this case, `Person` can see and use `Object`'s public and protected methods and variables.

Defining a class hierarchy

Now suppose you have an `Employee` class that inherits from `Person`. `Employee`'s class definition would look something like this:

```
public class Employee extends Person {

  private String taxpayerIdentificationNumber;
  private String employeeNumber;
  private BigDecimal salary;
  // . . .
}
```

The `Employee` inheritance relationship to all of its superclasses (its *inheritance graph*) implies that `Employee` has access to all public and protected variables and methods in `Person` (because `Employee` directly extends `Person`), as well as those in `Object` (because `Employee` actually extends `Object`, too, though indirectly). However, because `Employee` and `Person` are in the same package, `Employee` also has access to the *package-private* (sometimes called *friendly*) variables and methods in `Person`.

To go one step deeper into the class hierarchy, you could create a third class that extends `Employee`:

```
public class Manager extends Employee {
  // . . .
}
```

In the Java language, any class can have at most one direct superclass, but a class can have any number of subclasses. That's the most important thing to remember about inheritance hierarchy in the Java language.

Single versus multiple inheritance

Languages like C++ support the concept of *multiple inheritance*: At any point in the hierarchy, a class can directly inherit from one or more classes. The Java language supports only *single inheritance*— meaning you can only use the `extends` keyword with a single class. So the class hierarchy for any Java class always consists of a straight line all the way up to `java.lang.Object`. However, as you'll learn in Unit 17: Interfaces, the Java language supports implementing multiple interfaces in a single class, giving you a workaround of sorts to single inheritance.

Constructors and inheritance

Constructors aren't full-fledged object-oriented members, so they aren't inherited; instead, you must explicitly implement them in subclasses. Before I go into that topic, I'll review some basic rules about how constructors are defined and invoked.

Constructor basics

Remember that a constructor always has the same name as the class it's used to construct, and it has no return type. For example:

```
public class Person {
  public Person() {
  }
}
```

Every class has at least one constructor, and if you don't explicitly define a constructor for your class, the compiler generates one for you, called the *default constructor*. The preceding class definition and this one are identical in how they function:

```
public class Person {
}
```

Invoking a superclass constructor

To invoke a superclass constructor other than the default constructor, you must do so explicitly. For example, suppose `Person` has a constructor that takes just the name of the `Person` object being created. From `Employee`'s default constructor, you could invoke the `Person` constructor shown in Listing 2:

Listing 2. Initializing a new `Employee`

```
public class Person {
  private String name;
  public Person() {
  }
  public Person(String name) {
    this.name = name;
  }
}

// Meanwhile, in Employee.java
public class Employee extends Person {
  public Employee() {
    super("Elmer J Fudd");
  }
}
```

You would probably never want to initialize a new `Employee` object this way, however. Until you get more comfortable with object-oriented concepts, and Java syntax in general, it's a good idea to implement superclass constructors in subclasses only if you are sure you'll need them. Listing 3 defines a constructor in `Employee` that looks like the one in `Person` so that they match up. This approach is much less confusing from a maintenance standpoint.

Listing 3. Invoking a superclass

```
public class Person {
  private String name;
  public Person(String name) {
    this.name = name;
  }
}
// Meanwhile, in Employee.java
public class Employee extends Person {
  public Employee(String name) {
    super(name);
  }
}
```

Declaring a constructor

The first thing a constructor does is invoke the default constructor of its immediate superclass, unless you — on the first line of code in the constructor — invoke a different constructor. For example, the following two declarations are functionally identical:

```
public class Person {
  public Person() {
  }
}
// Meanwhile, in Employee.java
public class Employee extends Person {
  public Employee() {
  }
}
```

```
public class Person {
  public Person() {
  }
}
// Meanwhile, in Employee.java
public class Employee extends Person {
  public Employee() {
  super();
  }
}
```

No-arg constructors

If you provide an alternate constructor, you must explicitly provide the default constructor; otherwise it is unavailable. For example, the following code gives you a compile error:

```
public class Person {
  private String name;
  public Person(String name) {
    this.name = name;
  }
}
// Meanwhile, in Employee.java
public class Employee extends Person {

  public Employee() {
  }
}
```

The `Person` class in this example has no default constructor, because it provides an alternate constructor without explicitly including the default constructor.

How constructors invoke constructors

A constructor can invoke another constructor in the same class via the `this` keyword, along with an argument list. Like `super()`, the `this()` call must be the first line in the constructor, as in this example:

```
public class Person {
  private String name;
  public Person() {
    this("Some reasonable default?");
  }
  public Person(String name) {
    this.name = name;
  }

}
```

You see this idiom frequently. One constructor delegates to another, passing in a default value if that constructor is invoked. This technique is also a great way to add a new constructor to a class while minimizing impact on code that already uses an older constructor.

Constructor access levels

Constructors can have any access level you want, and certain rules of visibility apply. Table 1 summarizes the rules of constructor access.

Table 1. Constructor access rules

Constructor access modifier	Description
`public`	Constructor can be invoked by any class.
`protected`	Constructor can be invoked by an class in the same package or any subclass.
No modifier (*package-private*)	Constructor can be invoked by any class in the same package.
`private`	Constructor can be invoked only by the class in which the constructor is defined.

You might be able to think of use cases in which constructors would be declared `protected` or even package-private, but how is a `private` constructor useful? I've used private constructors when I didn't want to allow direct creation of an object through the `new` keyword when implementing, say, the Factory pattern. In that case, I'd use a static method to create instances of the class, and that method — being included in the class — would be allowed to invoke the private constructor.

Inheritance and abstraction

If a subclass overrides a method from a superclass, the method is essentially hidden because calling it through a reference to the subclass invokes the subclass's version of the method, not the superclass's version. However, the superclass method is still accessible. The subclass can invoke the superclass method by prefacing the name of the method with the `super` keyword (and unlike with the constructor rules, this can be done from any line in the subclass method, or even in a different method altogether). By default, a Java program calls the subclass method if it's invoked through a reference to the subclass.

This capability also applies to variables, provided the caller has access to the variable (that is, the variable is visible to the code trying to access it). This detail can cause you no end of grief as you gain proficiency in Java programming. Eclipse provides ample warnings — for example, that you're hiding a variable from a superclass, or that a method call won't call what you think it will.

In an OOP context, *abstraction* refers to generalizing data and behavior to a type higher up the inheritance hierarchy than the current class. When you move variables or methods from a subclass to a superclass, you say you are *abstracting* those members. The main reason for abstracting is to reuse common code by pushing it as far up the hierarchy as possible. Having common code in one place makes it easier to maintain.

Abstract classes and methods

At times, you want to create classes that only serve as abstractions and do not necessarily ever need to be instantiated. Such classes are called *abstract classes*. By the same token, sometimes certain methods need to be implemented differently for each subclass that implements the superclass. Such methods are *abstract methods*. Here are some basic rules for abstract classes and methods:

- Any class can be declared `abstract`.

- Abstract classes cannot be instantiated.
- An abstract method cannot contain a method body.
- Any class with an abstract method must be declared `abstract`.

Using abstraction

Suppose you don't want to allow the `Employee` class to be instantiated directly. You simply declare it using the `abstract` keyword, and you're done:

```
public abstract class Employee extends Person {
  // etc.
}
```

If you try to run this code, you get a compile error:

```
public void someMethodSomwhere() {
  Employee p = new Employee();// compile error!!
}
```

The compiler is complaining that `Employee` is abstract and can't be instantiated.

The power of abstraction

Suppose that you need a method to examine the state of an `Employee` object and make sure that it's valid. This need would seem to be common to all `Employee` objects, but it would have zero potential for reuse because it would behave differently among all potential subclasses. In that case, you declare the `validate()` method `abstract` (forcing all subclasses to implement it):

```
public abstract class Employee extends Person {
  public abstract boolean validate();
}
```

Every direct subclass of `Employee` (such as `Manager`) is now required to implement the `validate()` method. However, once a subclass implements the `validate()` method, none of its subclasses need to implement it.

For example, suppose you have an `Executive` object that extends `Manager`. This definition would be valid:

```
public class Executive extends Manager {
  public Executive() {
  }
}
```

When (not) to abstract: Two rules

As a first rule of thumb, don't abstract in your initial design. Using abstract classes early in the design forces you down a path that could restrict your application. You can always refactor common behavior (which is the entire point of having abstract classes) further up the inheritance graph — and it's almost always better to refactor after you've discovered that you do need to. Eclipse has wonderful support for refactoring.

Second, as powerful as abstract classes are, resist using them. Unless your superclasses contain much common behavior and aren't meaningful on their own, let them remain nonabstract. Deep inheritance graphs can make code maintenance difficult. Consider the trade-off between classes that are too large and maintainable code.

Assignments: Classes

You can assign a reference from one class to a variable of a type belonging to another class, but certain rules apply. Take a look at this example:

```
Manager m = new Manager();
Employee e = new Employee();
Person p = m; // okay
p = e; // still okay
Employee e2 = e; // yep, okay
e = m; // still okay
e2 = p; // wrong!
```

The destination variable must be of a supertype of the class belonging to the source reference, or else the compiler gives you an error. Whatever is on the right side of the assignment must be a subclass or the same class as the thing on the left. To put it another way: a subclass is more specific in purpose than its superclass, so think of a subclass as being **narrower** than its superclass. And a superclass, being more general, is **wider** than its subclass. The rule is this, you may never make an assignment that will **narrow** the reference.

Now consider this example:

```
Manager m = new Manager();
Manager m2 = new Manager();
m = m2; // Not narrower, so okay
Person p = m; // Widens, so okay
Employee e = m; // Also widens
Employee e = p; // Narrows, so not okay!
```

Although an `Employee` is a `Person`, it's most definitely not a `Manager`, and the compiler enforces this distinction.

Test your understanding

Refer to the following code listing for questions 2, 3, and 5.

```
package com.makotojava.intro;

public abstract class Person {

   private String name;

   String friendlyVariable;

   protected Person(String name) {
      this.name = name;
      this.friendlyVariable = name;
   }

   protected abstract boolean validate();
```

```
   public String getName() {
      return name;
   }

   public void setName(String name) {
      this.name = name;
   }

}

package com.makotojava.intro2;

public class Employee extends Person {

   private String title;

   public String getTitle() {
      return title;
   }

   public void setTitle(String title) {
      this.title = title;
   }

   public static void main(String[] args) {
      Employee e = new Employee();

      e.getName();
   }

}
```

1. A Java class can directly subclass (via `extends`) as many other classes as necessary. Explain your answer.
   a. True
   b. False
2. Refer to the preceding code listing for this question. `Employee` is a subclass of `Person`. Explain your answer.
   a. True
   b. False
3. Refer to the preceding code listing for this question. `Employee` can freely access the `friendlyVariable` variable of `Person` since it is a subclass of `Person`. Explain your answer.
   a. True
   b. False
4. `Employee` is not required to implement the `validate()` method of `Person`. Explain your answer.
   a. True
   b. False
5. Refer to the preceding code listing for this question. Correct the `Employee` code so that it will compile.
6. Suppose that class `A` doesn't declare a constructor. If class `B` extends `A`, it's not required to implement any constructor. Explain your answer.
   a. True
   b. False
7. Constructors are inherited just like any other methods. Explain your answer.
   a. True

    b. False
8. How might a class with two constructors invoke one from the other?
    a. By using the `super()` keyword
    b. By using the `this` keyword to represent the constructor. The compiler will generate the rest.
    c. Java classes are not allowed to have more than one constructor.
    d. By default, when two or more constructors are in the same class, the compiler generates method stubs that automatically call each other.
    e. By using the `this` keyword to represent the called constructor on the first line of code of the calling constructor, and passing the correct arguments
    f. None of the above.
9. How does one go about instantiating an `abstract` class?
    a. By invoking its public constructor.
    b. It is not possible to instantiate an abstract class.
    c. There is no such thing as an abstract class.
    d. None of the above.
10. What is the main purpose of abstraction through the use of generic superclasses?
    a. Abstraction facilitates code reuse.
    b. Abstraction in Java does not work the same as it does in C++ or other languages.
    c. A superclass that implements abstraction cannot be instantiated.
    d. Abstraction provides a way to make the code run faster.

Check your answers.

For further exploration

Default constructors and inheritance in Java

The Java Tutorials: Providing Constructors for Your Classes

The Java Tutorials: Abstract Methods and Classes

Java - Abstraction

Java - Object & Classes

IBM Code: Java journeys

Previous: Building Java applications
Next: Interfaces

Unit 17: Interfaces

Create a named set of behaviors

J Steven Perry

September 14, 2016

Learn the structure, syntax, and programming paradigm of the Java platform and language. Start by mastering the essentials of object-oriented programming on the Java platform, and progress incrementally to the more-sophisticated syntax and libraries that you need to develop complex, real-world Java applications.

Before you begin

This unit is part of the "Intro to Java programming" learning path. Although the concepts discussed in the individual units are standalone in nature, the hands-on component builds as you progress through the units, and I recommend that you review the prerequisites, setup, and unit details before proceeding.

Unit objectives

- Understand the purpose of an interface
- Learn the syntax for defining an interface
- Know how to implement an interface
- Learn the syntax for using an interface and assigning an interface reference to a variable

Interfaces in action

The following short video addresses all of this unit's learning objectives. I show you what an interface is, how to define and implement an interface, and how the reference assignment rules work. After you watch, you can get a closer look at the code, and a recap of the concepts, by reading this unit's text.

To view this video, **Interfaces** , please access the online version of the article. If this article is in the developerWorks archives, the video is no longer accessible.

Interfaces: What are they good for?

As you know from the previous unit, abstract methods, by design, specify a *contract*— through the method name, parameter(s), and return type — but provide no reusable code. Abstract methods

— defined on abstract classes — are useful when the way the behavior is implemented is likely to change from the way it's implemented in one subclass of the abstract class to another.

When you see a set of common behaviors in your application (think `java.util.List`) that can be grouped together and named, but for which two or more implementations exist, you might consider defining that behavior with an *interface*— and that's why the Java language provides this feature. However, this fairly advanced feature is easily abused, obfuscated, and twisted into the most heinous shapes (as I've witnessed first-hand), so use interfaces with caution.

It might be helpful to think about interfaces this way: They are like abstract classes that contain **only** abstract methods; they define **only** the contract but none of the implementation.

Defining an interface

The syntax for defining an interface is straightforward:

```
public interface InterfaceName {
    returnType methodName(argumentList);
  }
```

An interface declaration looks like a class declaration, except that you use the `interface` keyword. You can name the interface anything you want to (subject to language rules), but by convention, interface names look like class names.

Methods defined in an interface have no method body. The implementer of the interface is responsible for providing the method body (as with abstract methods).

You define hierarchies of interfaces, as you do for classes, except that a single class can implement as many interfaces as you want it to. Remember, a class can extend only one class. If one class extends another and implements an interface or interfaces, you list the interfaces after the extended class, like this:

```
public class Manager extends Employee implements BonusEligible, StockOptionRecipient {
  // And so on
}
```

An interface doesn't need to have any body at all. The following definition, for example, is perfectly acceptable:

```
public interface BonusEligible {
}
```

Generally speaking, such interfaces are called *marker interfaces*, because they mark a class as implementing that interface but offer no special explicit behavior.

Once you know all that, actually defining an interface is easy:

```
public interface StockOptionRecipient {
  void processStockOptions(int numberOfOptions, BigDecimal price);
}
```

Implementing interfaces

To define an interface on your class, you must *implement* the interface, which means that you provide a method body that provides the behavior to fulfill the interface's contract. You use the `implements` keyword to implement an interface:

```
public class ClassName extends SuperclassName implements InterfaceName {
  // Class Body
}
```

Suppose you implement the `StockOptionRecipient` interface on the `Manager` class, as shown in Listing 1:

Listing 1. Implementing an interface

```
public class Manager extends Employee implements StockOptionRecipient {
  public Manager() {
  }
  public void processStockOptions (int numberOfOptions, BigDecimal price) {
    log.info("I can't believe I got " + number + " options at $" +
    price.toPlainString() + "!");
  }
}
```

When you implement the interface, you provide behavior for the method or methods on the interface. You must implement the methods with signatures that match the ones on the interface, with the addition of the `public` access modifier.

An abstract class can declare that it implements a particular interface, but you're not required to implement all of the methods on that interface. Abstract classes aren't required to provide implementations for all of the methods they claim to implement. However, the first concrete class (that is, the first one that can be instantiated) must implement all methods that the hierarchy doesn't implement.

Note: Subclasses of a concrete class that implements an interface do not need to provide their own implementation of that interface (because the methods on the interface have been implemented by the superclass).

Generating interfaces in Eclipse

Eclipse can easily generate the correct method signature for you if you decide that one of your classes should implement an interface. Just change the class signature to implement the interface. Eclipse puts a red squiggly line under the class, flagging it to be in error because the class doesn't provide the methods on the interface. Click the class name, press Ctrl + 1, and Eclipse suggests "quick fixes" for you. Of these, choose **Add Unimplemented Methods**, and Eclipse generates the methods for you, placing them at the bottom of the source file.

Using interfaces

An interface defines a new *reference* data type, which you can use to refer to an interface anywhere you would refer to a class. This ability includes when you declare a reference variable, or cast from one type to another, as shown in Listing 2.

Listing 2. Assigning a new `Manager` instance to a `StockOptionEligible` reference

```
package com.makotojava.intro;
import java.math.BigDecimal;
import org.junit.Test;
public class ManagerTest {
  @Test
  public void testCalculateAndAwardStockOptions() {
    StockOptionEligible soe = new Manager();// perfectly valid
    calculateAndAwardStockOptions(soe);
    calculateAndAwardStockOptions(new Manager());// works too
    }
    public static void calculateAndAwardStockOptions(StockOptionEligible soe) {
    BigDecimal reallyCheapPrice = BigDecimal.valueOf(0.01);
    int numberOfOptions = 10000;
    soe.awardStockOptions(numberOfOptions, reallyCheapPrice);
  }
}
```

As you can see, it's valid to assign a new `Manager` instance to a `StockOptionEligible` reference, and to pass a new `Manager` instance to a method that expects a `StockOptionEligible` reference.

Assignments: Interfaces

You can assign a reference from a class that implements an interface to a variable of an interface type, but certain rules apply. From Listing 2, you can see that assigning a `Manager` instance to a `StockOptionEligible` variable reference is valid. The reason is that the `Manager` class implements that interface. However, the following assignment would not be valid:

```
 Manager m = new Manager();
  StockOptionEligible soe = m; //okay
  Employee e = soe; // Wrong!
```

Because `Employee` is a supertype of `Manager`, this code might at first seem okay, but it's not. Why not? Because `Manager` implements the `StockOptionEligible` interface, whereas `Employee` does not.

Assignments such as these follow the rules of assignment that you saw in Unit 16: Inheritance. And as with classes, you can only assign an interface reference to a variable of the same type or a superinterface type.

For further exploration

Is there more to an interface than having the correct methods

Java - Interfaces

The Java Tutorials: Interfaces

IBM Code: Java journeys

Previous: Inheritance

Next: Nested classes

Unit 18: Nested classes

Define tightly coupled classes

J Steven Perry

September 14, 2016

Learn the structure, syntax, and programming paradigm of the Java platform and language. Start by mastering the essentials of object-oriented programming on the Java platform, and progress incrementally to the more-sophisticated syntax and libraries that you need to develop complex, real-world Java applications.

Before you begin

This unit is part of the "Intro to Java programming" learning path. Although the concepts discussed in the individual units are standalone in nature, the hands-on component builds as you progress through the units, and I recommend that you review the prerequisites, setup, and unit details before proceeding.

Unit objectives

- Know how to define nested classes and when it's appropriate to use them
- Understand the side effects of using nested classes
- Comprehend the special use of `new` operator with nested classes
- Know how and when to use static inner classes and anonymous inner classes

Where to use nested classes

As its name suggests, a *nested class* (or *inner class*) is a class defined within another class:

```
public class EnclosingClass {
  . . .
  public class NestedClass {
  . . .

  }
}
```

Like member variables and methods, Java classes can also be defined at any scope including `public`, `private`, or `protected`. Nested classes can be useful when you want to handle internal processing within your class in an object-oriented fashion but limit the functionality to the class where you need it.

Typically, you use a nested class when you need a class that's tightly coupled with the class in which it's defined. A nested class has access to the private data within its enclosing class, but this structure carries with it side effects that aren't obvious when you start working with nested classes.

Scope in nested classes

Because a nested class has scope, it's bound by the rules of scope. For example, a member variable can only be accessed through an instance of the class (an object). The same is true of a nested class.

Suppose you have the following relationship between a `Manager` and a nested class called `DirectReports`, which is a collection of the `Employee`s that report to that `Manager`:

```
public class Manager extends Employee {
  private DirectReports directReports;
  public Manager() {
    this.directReports = new DirectReports();
  }
  . . .
  private class DirectReports {
  . . .
  }
}
```

Just as each `Manager` object represents a unique human being, the `DirectReports` object represents a collection of actual people (employees) who report to a manager. `DirectReports` differ from one `Manager` to another. In this case, it makes sense that I would only reference the `DirectReports` nested class in the context of its enclosing instance of `Manager`, so I've made it `private`.

Public nested classes

Because it's `private`, only `Manager` can create an instance of `DirectReports`. But suppose you wanted to give an external entity the ability to create instances of `DirectReports`. In this case, it seems like you could give the `DirectReports` class `public` scope, and then any external code could create `DirectReports` instances, as shown in Listing 1.

Listing 1. Creating `DirectReports` instances: First attempt

```
public class Manager extends Employee {
  public Manager() {
  }
  . . .
  public class DirectReports {
  . . .
  }
}
//
public static void main(String[] args) {
  Manager.DirectReports dr = new Manager.DirectReports();// This won't work!
}
```

The code in Listing 1 doesn't work, and you're probably wondering why. The problem (and also its solution) lies with the way `DirectReports` is defined within `Manager`, and with the rules of scope.

The rules of scope, revisited

If you had a member variable of `Manager`, you'd expect the compiler to require you to have a reference to a `Manager` object before you could reference it, right? Well, the same applies to `DirectReports`, at least as it's defined in Listing 1.

To create an instance of a public nested class, you use a special version of the `new` operator. Combined with a reference to an enclosing instance of an outer class, `new` gives you a way you to create an instance of the nested class:

```
public class Manager extends Employee {
  public Manager() {
  }
  . . .
  public class DirectReports {
  . . .
  }
  }
// Meanwhile, in another method somewhere...
public static void main(String[] args) {
  Manager manager = new Manager();
  Manager.DirectReports dr = manager.new DirectReports();
}
```

Note on line 12 that the syntax calls for a reference to the enclosing instance, plus a dot and the `new` keyword, followed by the class you want to create.

Static inner classes

At times, you want to create a class that's tightly coupled (conceptually) to a class, but where the rules of scope are somewhat relaxed, not requiring a reference to an enclosing instance. That's where *static* inner classes come into play. One common example is to implement a `Comparator`, which is used to compare two instances of the same class, usually for the purpose of ordering (or sorting) the classes:

```
public class Manager extends Employee {
  . . .
  public static class ManagerComparator implements Comparator<Manager> {
  . . .
  }
  }
// Meanwhile, in another method somewhere...
public static void main(String[] args) {
  Manager.ManagerComparator mc = new Manager.ManagerComparator();
  . . .
}
```

In this case, you don't need an enclosing instance. Static inner classes act like their regular Java class counterparts, and you should use them only when you need to couple a class tightly with its definition. Clearly, in the case of a utility class like `ManagerComparator`, creating an external class is unnecessary and potentially clutters up your code base. Defining such classes as static inner classes is the way to go.

Anonymous inner classes

With the Java language, you can implement abstract classes and interfaces pretty much anywhere, even in the middle of a method if necessary, and even without providing a name for the class. This capability is basically a compiler trick, but there are times when anonymous inner classes are handy to have.

Listing 2 builds on the example in Listing 1 in Unit 17: Interfaces, adding a default method for handling `Employee` types that are not `StockOptionEligible`. The listing starts with a method in `HumanResourcesApplication` to process the stock options, followed by a JUnit test to drive the method.

Listing 2. Handling `Employee` types that are not `StockOptionEligible`

```
// From HumanResourcesApplication.java
public void handleStockOptions(final Person person, StockOptionProcessingCallback callback) {
  if (person instanceof StockOptionEligible) {
    // Eligible Person, invoke the callback straight up
    callback.process((StockOptionEligible)person);
  } else if (person instanceof Employee) {
    // Not eligible, but still an Employee. Let's cobble up a
    /// anonymous inner class implementation for this
    callback.process(new StockOptionEligible() {
      @Override
      public void awardStockOptions(int number, BigDecimal price) {
        // This employee is not eligible
        log.warning("It would be nice to award " + number + " of shares at $" +
            price.setScale(2, RoundingMode.HALF_UP).toPlainString() +
            ", but unfortunately, Employee " + person.getName() +
            " is not eligible for Stock Options!");
      }
    });
  } else {
    callback.process(new StockOptionEligible() {
      @Override
      public void awardStockOptions(int number, BigDecimal price) {
        log.severe("Cannot consider awarding " + number + " of shares at $" +
            price.setScale(2, RoundingMode.HALF_UP).toPlainString() +
            ", because " + person.getName() +
            " does not even work here!");
      }
    });
  }
}
// JUnit test to drive it (in HumanResourcesApplicationTest.java):
@Test
public void testHandleStockOptions() {
  List<Person> people = HumanResourcesApplication.createPeople();

  StockOptionProcessingCallback callback = new StockOptionProcessingCallback() {
    @Override
    public void process(StockOptionEligible stockOptionEligible) {
      BigDecimal reallyCheapPrice = BigDecimal.valueOf(0.01);
      int numberOfOptions = 10000;
      stockOptionEligible.awardStockOptions(numberOfOptions, reallyCheapPrice);
    }
  };
  for (Person person : people) {
    classUnderTest.handleStockOptions(person, callback);
```

```
  }
}
```

In the Listing 2 example, I provide implementations of two interfaces that use anonymous inner classes. First are two separate implementations of `StockOptionEligible`— one for `Employee`s and one for `Person`s (to obey the interface). Then comes an implementation of `StockOptionProcessingCallback` that's used to handle processing of stock options for the `Manager` instances.

It might take time to grasp the concept of anonymous inner classes, but they're super handy. I use them all the time in my Java code. And as you progress as a Java developer, I believe you will too.

Test your understanding

1. What is a nested class?
   a. A class that provides some utility to other classes in the application
   b. A class that is defined within another class
   c. A class where one or more interface references are passed into its constructor
   d. None of the above
2. Why might you use a nested class?
   a. When you need to define a class that is tightly coupled (functionally) with another class
   b. When you want to write a class that makes use of a large number of interfaces from the JDK
   c. When one class needs to access to another class private data, but you have exceeded the maximum number of allowable classes for your application
   d. For a class has more than 20 methods defined on it and should be refactored
3. Can you spot the error in the following code? Choose the best answer, explain your choice, and provide the correct code.

```
package com.makotojava.intro.quiz;

import java.util.logging.Logger;

public class Outer {

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

   public void setInner(Inner inner) {
      this.inner = inner;
   }

   private Inner inner;

   public Inner getInner() {

      return inner;
   }

   private class Inner {
   }

   public static void main(String[] args) {
      Outer outer = new Outer();
      Inner inner = new Outer.Inner();
      outer.setInner(inner);
      log.info("Outer/Inner: " + outer.hashCode() + "/" + outer.getInner().hashCode());
```

```
   }

}
```

a. The class name `Outer` is not a legal Java class name.
b. The class name `Inner` is confusing.
c. The main method defined in class `Outer` has the wrong method signature.
d. The `log.info()` call in main() has too many parameters.
e. The class body for class `Inner` is empty.
f. None of the above.

4. Refer to the code listing in Question 3. Suppose you want a new class called `AnotherOuter` in the same package as `Outer` that could instantiate `Outer.Inner`. What change(s) would you need to make to the declaration of `Outer.Inner`?
   a. No changes are necessary; `Inner` can be instantiated as-is.
   b. No inner class can be instantiated outside its enclosing class.
   c. Change the visibility of `Inner` to `protected` or package-private.
   d. Add the `static` modifier to the `Inner` class declaration.
   e. None of the above.
5. Given the following skeleton, flesh out the `main()` method to:
   1. Instantiate the `Hello` class
   2. Call `Hello`s `talk()` method, passing it an instance of `HelloCallback` implemented as an anonymous inner class
   3. Modify the anonymous inner class implementation of `sayHello()`to use the `Logger` defined on `Hello` to output:
      - "This implementation says: "
      - The `whatToSay` string

```
import java.util.logging.Logger;

public class Hello {

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

   interface HelloCallback {
      void sayHello(String whatToSay);
   }

   public void talk(HelloCallback helloCallback) {
      helloCallback.sayHello("Hello, world (how original :/)...");
   }

   public static void main(String[] args) {
        // YOUR ANSWER GOES HERE
   }

}
```

6. Is it possible to write an inner class that can be instantiated by any class in your application (regardless of what package in which it resides) without an enclosing instance of the outer class? Explain your answer.
7. What's the difference between a nested class an an inner class?
   a. An inner class is one that is defined using private access only, whereas a nested class is declared static.

   b. The two terms are synonymous and can be used interchangeably.
   c. A nested class must reside inside the enclosing class and be declared before any of the enclosing class's variables.
   d. A nested class is defined in main(), whereas an inner class can be defined anywhere.
   e. There is no such thing as a nested class.
   f. None of the above.
8. Which of these statements is true regarding inner classes?
   a. An inner class can access any private data variables of its enclosing class unless it is declared `static`.
   b. An inner class must be declared `public` to be instantiated by any other class than its enclosing class.
   c. A static inner class is not allowed except under special circumstances.
   d. An inner class is completely invisible to its enclosing class.
   e. None of the above.

Check your answers

For further exploration

The Java Tutorials: Nested Classes

Java - Inner Classes

Java inner class and static nested class

When to use inner classes in Java for helper classes

What are the purposes of inner classes

IBM Code: Java journeys

Previous: Interfaces
Next: Regular expressions

Unit 19: Regular expressions

Describe and search for string patterns in your Java code

J Steven Perry

September 14, 2016

Learn the structure, syntax, and programming paradigm of the Java platform and language. Start by mastering the essentials of object-oriented programming on the Java platform, and progress incrementally to the more-sophisticated syntax and libraries that you need to develop complex, real-world Java applications.

Before you begin

This unit is part of the "Intro to Java programming" learning path. Although the concepts discussed in the individual units are standalone in nature, the hands-on component builds as you progress through the units, and I recommend that you review the prerequisites, setup, and unit details before proceeding.

Unit objectives

- Learn what the three core regex classes are and what they do
- Become familiar with regex pattern syntax
- Be able to perform both simple and more complex searches and replacements
- Know how to reference matches by capturing groups

The Regular Expressions API

A *regular expression* is essentially a pattern to describe a set of strings that share that pattern. This unit gets you started with using regular expressions in your Java programs.

Here's a set of strings that have a few things in common:

- A string
- A longer string
- A much longer string

Note that each of these strings begins with *A* and ends with *string*. The Java Regular Expressions API helps you pull out these elements, see the pattern among them, and do interesting things with the information you've gleaned.

The Regular Expressions API has three core classes that you use almost all the time:

- `Pattern` describes a string pattern.
- `Matcher` tests a string to see if it matches the pattern.
- `PatternSyntaxException` tells you that something wasn't acceptable about the pattern that you tried to define.

You'll begin working on a simple regular-expressions pattern that uses these classes shortly. But first, take a look at the regex pattern syntax.

Regex pattern syntax

A *regex pattern* describes the structure of the string that the expression tries to find in an input string. The pattern syntax can look strange to the uninitiated, but once you understand it, you'll find it easier to decipher. Table 1 lists some of the most common regex constructs that you use in pattern strings.

Table 1. Common regex constructs

Regex construct	What qualifies as a match
`.`	Any character
`?`	Zero (0) or one (1) of what came before
`*`	Zero (0) or more of what came before
`+`	One (1) or more of what came before
`[]`	A range of characters or digits
`^`	Negation of whatever follows (that is, "not *whatever*")
`\d`	Any digit (alternatively, `[0-9]`)
`\D`	Any nondigit (alternatively, `[^0-9]`)
`\s`	Any whitespace character (alternatively, `[\n\t\f\r]`)
`\S`	Any nonwhitespace character (alternatively, `[^\n\t\f\r]`)
`\w`	Any word character (alternatively, `[a-zA-Z_0-9]`)
`\W`	Any nonword character (alternatively, `[^\w]`)

The first few constructs are called *quantifiers*, because they quantify what comes before them. Constructs like `\d` are predefined character classes. Any character that doesn't have special meaning in a pattern is a literal and matches itself.

Pattern matching

Armed with the pattern syntax in Table 1, you can work through the simple example in Listing 1, using the classes in the Java Regular Expressions API.

Listing 1. Pattern matching with regex

```
Pattern pattern = Pattern.compile("[Aa].*string");
  Matcher matcher = pattern.matcher("A string");
  boolean didMatch = matcher.matches();
  Logger.getAnonymousLogger().info (didMatch);
  int patternStartIndex = matcher.start();
  Logger.getAnonymousLogger().info (patternStartIndex);
  int patternEndIndex = matcher.end();
  Logger.getAnonymousLogger().info (patternEndIndex);
```

First, Listing 1 creates a `Pattern` class by calling `compile()`— a static method on `Pattern`— with a string literal representing the pattern you want to match. That literal uses the regex pattern syntax. In this example, the English translation of the pattern is:

Find a string of the form `A` or `a` followed by zero or more characters, followed by `string`.

Methods for matching

Next, Listing 1 calls `matcher()` on `Pattern`. That call creates a `Matcher` instance. The `Matcher` then searches the string you passed in for matches against the pattern string you used when you created the `Pattern`.

Every Java language string is an indexed collection of characters, starting with 0 and ending with the string length minus one. The `Matcher` parses the string, starting at 0, and looks for matches against it. After that process is complete, the `Matcher` contains information about matches found (or not found) in the input string. You can access that information by calling various methods on `Matcher`:

- `matches()` tells you if the entire input sequence was an exact match for the pattern.
- `start()` tells you the index value in the string where the matched string starts.
- `end()` tells you the index value in the string where the matched string ends, plus one.

Listing 1 finds a single match starting at 0 and ending at 7. Thus, the call to `matches()` returns `true`, the call to `start()` returns `0`, and the call to `end()` returns `8`.

lookingAt() versus matches()

If your string had more elements than the number of characters in the pattern you searched for, you could use `lookingAt()` instead of `matches()`. The `lookingAt()` method searches for substring matches for a specified pattern. For example, consider the following string:

```
a string with more than just the pattern.
```

If you search this string for `a.*string`, you get a match if you use `lookingAt()`. But if you use `matches()`, it returns `false`, because there's more to the string than what's in the pattern.

Complex patterns in regex

Simple searches are easy with the regex classes, but you can also do highly sophisticated things with the Regular Expressions API.

Wikis are based almost entirely on regular expressions. Wiki content is based on string input from users, which is parsed and formatted using regular expressions. Any user can create a link to another topic in a wiki by entering a wiki word, which is typically a series of concatenated words, each of which begins with an uppercase letter, like this:

```
MyWikiWord
```

Suppose a user inputs the following string:

```
Here is a WikiWord followed by AnotherWikiWord, then YetAnotherWikiWord.
```

You could search for wiki words in this string with a regex pattern like this:

```
[A-Z][a-z]*([A-Z][a-z]*)+
```

And here's code to search for wiki words:

```
String input = "Here is a WikiWord followed by AnotherWikiWord, then SomeWikiWord.";
Pattern pattern = Pattern.compile("[A-Z][a-z]*([A-Z][a-z]*)+");
Matcher matcher = pattern.matcher(input);
while (matcher.find()) {
  Logger.getAnonymousLogger().info("Found this wiki word: " + matcher.group());
}
```

Run this code, and you can see the three wiki words in your console.

Replacing strings

Searching for matches is useful, but you also can manipulate strings after you find a match for them. You can do that by replacing matched strings with something else, just as you might search for text in a word-processing program and replace it with other text. `Matcher` has a couple of methods for replacing string elements:

- `replaceAll()` replaces all matches with a specified string.
- `replaceFirst()` replaces only the first match with a specified string.

Using `Matcher`'s `replace` methods is straightforward:

```
String input = "Here is a WikiWord followed by AnotherWikiWord, then SomeWikiWord.";
Pattern pattern = Pattern.compile("[A-Z][a-z]*([A-Z][a-z]*)+");
Matcher matcher = pattern.matcher(input);
Logger.getAnonymousLogger().info("Before: " + input);
String result = matcher.replaceAll("replacement");
Logger.getAnonymousLogger().info("After: " + result);
```

This code finds wiki words, as before. When the `Matcher` finds a match, it replaces the wiki word text with its replacement. When you run the code, you can see the following on your console:

```
Before: Here is WikiWord followed by AnotherWikiWord, then SomeWikiWord.
  After: Here is replacement followed by replacement, then replacement.
```

If you had used `replaceFirst()`, you would have seen this:

```
Before: Here is a WikiWord followed by AnotherWikiWord, then SomeWikiWord.
  After: Here is a replacement followed by AnotherWikiWord, then SomeWikiWord.
```

Matching and manipulating groups

When you search for matches against a regex pattern, you can get information about what you found. You've seen some of that capability with the `start()` and `end()` methods on `Matcher`. But it's also possible to reference matches by capturing *groups*.

In each pattern, you typically create groups by enclosing parts of the pattern in parentheses. Groups are numbered from left to right, starting with 1 (group 0 represents the entire match). The code in Listing 2 replaces each wiki word with a string that "wraps" the word:

Listing 2. Matching groups

```
String input = "Here is a WikiWord followed by AnotherWikiWord, then SomeWikiWord.";
Pattern pattern = Pattern.compile("[A-Z][a-z]*([A-Z][a-z]*)+");
Matcher matcher = pattern.matcher(input);
Logger.getAnonymousLogger().info("Before: " + input);
String result = matcher.replaceAll("blah$0blah");
Logger.getAnonymousLogger().info("After: " + result);
```

Run the Listing 2 code, and you get the following console output:

```
Before: Here is a WikiWord followed by AnotherWikiWord, then SomeWikiWord.
  After: Here is a blahWikiWordblah followed by blahAnotherWikiWordblah,
  then blahSomeWikiWordblah.
```

Listing 2 references the entire match by including `$0` in the replacement string. Any portion of a replacement string of the form `$`*`int`* refers to the group identified by the integer (so `$1` refers to group 1, and so on). In other words, `$0` is equivalent to `matcher.group(0);`.

You could accomplish the same replacement goal by using other methods. Rather than calling `replaceAll()`, you could do this:

```
StringBuffer buffer = new StringBuffer();
while (matcher.find()) {
  matcher.appendReplacement(buffer, "blah$0blah");
}
matcher.appendTail(buffer);
Logger.getAnonymousLogger().info("After: " + buffer.toString());
```

And you'd get the same result:

```
Before: Here is a WikiWord followed by AnotherWikiWord, then SomeWikiWord.
  After: Here is a blahWikiWordblah followed by blahAnotherWikiWordblah,
  then blahSomeWikiWordblah.
```

Test your understanding

1. Which statement best describes the `?` quantifier?
    a. Matches zero or more times
    b. Matches one or more times
    c. Matches the first occurrence and appends the match to the output group
    d. Matches once or not at all
    e. None of the above

2. Which statement best describes the `+` quantifier?
   a. Matches zero or more times
   b. Matches one or more times
   c. Matches the first occurrence and appends the match to the output group
   d. Matches once or not at all
   e. None of the above
3. Which statement best describes the `*` quantifier?
   a. Matches zero or more times
   b. Matches one or more times
   c. Matches the first occurrence and appends the match to the output group
   d. Matches once or not at all
   e. None of the above
4. **True or false:** The `Matcher` class is used to describe the input string to the `Pattern` class.
5. Which answer best describes an application of the following regular expression string: `[A-Z]?\d`
   1. Match any character A through Z one or more times, followed by a single optional digit.
   2. Match any character A through Z zero or one times, followed by a single optional digit.
   3. Match any character A through Z one or more times, followed by a single digit.
   4. Match any character A through Z zero or one times, followed by a single digit.
   5. None of the above.
6. Examine the following code and choose the response that best describes the matches (in order).

```
@Test
public void testFindMatches() {

   String input = "Do you run? Ran? No, bro, run! Bro, I ran and run.";

   String regex = "r[au]n";

   Pattern pattern = Pattern.compile(regex);
   Matcher matcher = pattern.matcher(input);

   int matchCount = 0;
   StringBuilder matchHolder = new StringBuilder();
   while (matcher.find()) {
   if (matchCount > 0)
   matchHolder.append(',');
   matchHolder.append(matcher.group());
   matchCount++;
   }

   System.out.println("Matches: " + matchHolder.toString());

}
```

   a. run,Ran,run,ran,run
   b. run,run,run,run
   c. run,ran,run,Ran,run
   d. run,run,ran,run
   e. The specified pattern does not match any part of the input string.

7. Programming exercise, part 1: Create a new class (call it `MyRegExMatcher`), and write a method called `matchesAll` that takes two parameters — a String called `regex`, and a String called `input`— and returns a `boolean`. For now, just write the method to return `false`.
8. Programming exercise, part 2: Create a JUnit test case that calls the method you wrote for Question 7. Your JUnit test will invoke the method with the simplest regular expression you can come up with that matches this input String: `The quick brown fox jumped over the lazy dogs`
Note: the regular expression may only contain quantifiers and must contain the letters l and x only (no other letters).
9. Programming exercise, part 3: Implement the method from Question 7 so that your test case passes (if your test case does not pass, your regular expression might be wrong). Return `true` if the entire input string matches the regular expression pattern, false otherwise. Hint: Use the `Pattern` class, and the `Matcher` class, as you saw in Listing 1.

Check your answers.

For further exploration

Java Regular Expressions API

Java - Regular Expressions

Regular Expression Test Page

The Java Tutorials: Regular Expressions

IBM Code: Java journeys

Previous: Nested classes
Next: Generics

Unit 20: Generics

Define classes with abstract type parameters

J Steven Perry September 14, 2016

Learn the structure, syntax, and programming paradigm of the Java platform and language. Start by mastering the essentials of object-oriented programming on the Java platform, and progress incrementally to the more-sophisticated syntax and libraries that you need to develop complex, real-world Java applications.

Before you begin

This unit is part of the "Intro to Java programming" learning path. Although the concepts discussed in the individual units are standalone in nature, the hands-on component builds as you progress through the units, and I recommend that you review the prerequisites, setup, and unit details before proceeding.

Unit objectives

- Understand the advantages of using generics
- Know when and how to parameterize classes or methods
- Learn how to iterate with generics
- Know how to use the `enum` data type

What are generics?

Build your skills
Learn how type erasure works to enable Java Generics in this step-by-step recipe and video demo.

JDK 5.0 (released in 2004) introduced *generic types* (*generics*) and the associated syntax into the Java language. Basically, some then-familiar JDK classes were replaced with their generic equivalents. Generics is a compiler mechanism whereby you can create (and use) types of things (such as classes or interfaces, and methods) in a generic fashion by harvesting the common code and *parameterizing* (or *templatizing*) the rest. This approach to programming is called *generic programming*.

Generics in action

To see what a difference generics makes, consider the example of a class that has been in the JDK for a long time: `java.util.ArrayList`, which is a `List` of `Object`s that's backed by an array.

Listing 1 shows how `java.util.ArrayList` is instantiated.

Listing 1. Instantiating `ArrayList`

```
ArrayList arrayList = new ArrayList();
arrayList.add("A String");
arrayList.add(new Integer(10));
arrayList.add("Another String");
// So far, so good.
```

As you can see, the `ArrayList` is heterogeneous: It contains two `String` types and one `Integer` type. Before JDK 5.0, the Java language had nothing to constrain this behavior, which caused many coding mistakes. In Listing 1, for example, everything is looking good so far. But what about accessing the elements of the `ArrayList`, which Listing 2 tries to do?

Listing 2. Attempt to access elements in `ArrayList`

```
ArrayList arrayList = new ArrayList();
arrayList.add("A String");
arrayList.add(new Integer(10));
arrayList.add("Another String");
// So far, so good.
processArrayList(arrayList);
// In some later part of the code...
private void processArrayList(ArrayList theList) {
  for (int aa = 0; aa < theList.size(); aa++) {
    // At some point, this will fail...
    String s = (String)theList.get(aa);
  }
}
```

Without prior knowledge of what's in the `ArrayList`, you must either check the element that you want to access to see if you can handle its type, or face a possible `ClassCastException`.

With generics, you can specify the type of item that went in the `ArrayList`. Listing 3 shows how, and what happens if you try and add an object of the wrong type (line 3).

Listing 3. A second attempt, using generics

```
ArrayList<String> arrayList = new ArrayList<>();
arrayList.add("A String");
arrayList.add(new Integer(10));// compiler error!
arrayList.add("Another String");
// So far, so good.
processArrayList(arrayList);
// In some later part of the code...
private void processArrayList(ArrayList<String> theList) {
  for (int aa = 0; aa < theList.size(); aa++) {
    // No cast necessary...
    String s = theList.get(aa);
  }
}
```

Iterating with generics

Generics enhance the Java language with special syntax for dealing with entities, such as `List`s, that you commonly want to step through element by element. If you want to iterate through `ArrayList`, for instance, you could rewrite the code from Listing 3 like so:

```
private void processArrayList(ArrayList<String> theList) {
  for (String s : theList) {
    String s = theList.get(aa);
  }
}
```

This syntax works for any type of object that is `Iterable` (that is, implements the `Iterable` interface).

Parameterized classes

Parameterized classes shine when it comes to collections, so that's the context for the following examples. Consider the `List` interface, which represents an ordered collection of objects. In the most common use case, you add items to the `List` and then access those items either by index or by iterating over the `List`.

If you're thinking about parameterizing a class, consider if the following criteria apply:

- A core class is at the center of some kind of wrapper: The "thing" at the center of the class might apply widely, and the features (attributes, for example) surrounding it are identical.
- The behavior is common: You do pretty much the same operations regardless of the "thing" at the center of the class.

Applying these two criteria, you can see that a collection fits the bill:

- The "thing" is the class of which the collection is composed.
- The operations (such as `add`, `remove`, `size`, and `clear`) are pretty much the same regardless of the object of which the collection is composed.

A parameterized List

In generics syntax, the code to create a `List` looks like this:

```
List<E> listReference = new concreteListClass<E>();
```

The `E`, which stands for Element, is the "thing" I mentioned earlier. The `concreteListClass` is the class from the JDK that you're instantiating. The JDK includes several `List<E>` implementations, but you use `ArrayList<E>`. Another way you might see a generic class discussed is `Class<T>`, where `T` stands for Type. When you see `E` in Java code, it's usually referring to a collection of some kind. And when you see `T`, it's denoting a parameterized class.

So, to create an `ArrayList` of, say, `java.lang.Integer`, you do this:

```
List<Integer> listOfIntegers = new ArrayList<Integer>();
```

SimpleList: A parameterized class

Now suppose you want to create your own parameterized class called `SimpleList`, with three methods:

- `add()` adds an element to the end of the `SimpleList`.
- `size()` returns the current number of elements in the `SimpleList`.
- `clear()` completely clears the contents of the `SimpleList`.

Listing 4 shows the syntax to parameterize `SimpleList`.

Listing 4. Parameterizing `SimpleList`

```
package com.makotojava.intro;
import java.util.ArrayList;
import java.util.List;
public class SimpleList<E> {
  private List<E> backingStore;
  public SimpleList() {
    backingStore = new ArrayList<E>();
  }
  public E add(E e) {
    if (backingStore.add(e))
    return e;
    else
    return null;
  }
  public int size() {
    return backingStore.size();
  }
  public void clear() {
    backingStore.clear();
  }
}
```

`SimpleList` can be parameterized with any `Object` subclass. To create and use a `SimpleList` of, say, `java.math.BigDecimal` objects, you might do this:

```
package com.makotojava.intro;
import java.math.BigDecimal;
import java.util.logging.Logger;
import org.junit.Test;
public class SimpleListTest {
  @Test
  public void testAdd() {
    Logger log = Logger.getLogger(SimpleListTest.class.getName());

    SimpleList<BigDecimal> sl = new SimpleList<>();
    sl.add(BigDecimal.ONE);
    log.info("SimpleList size is : " + sl.size());
    sl.add(BigDecimal.ZERO);
    log.info("SimpleList size is : " + sl.size());
    sl.clear();
    log.info("SimpleList size is : " + sl.size());
  }
}
```

And you would get this output:

```
Sep 20, 2015 10:24:33 AM com.makotojava.intro.SimpleListTest testAdd
INFO: SimpleList size is: 1 Sep 20, 2015 10:24:33 AM com.makotojava.intro.SimpleListTest testAdd
INFO: SimpleList size is: 2 Sep 20,
2015 10:24:33 AM com.makotojava.intro.SimpleListTest testAdd
INFO: SimpleList size is: 0
```

Parameterized methods

At times, you might not want to parameterize your entire class, but only one or two methods. In this case, you create a *generic method*. Consider the example in Listing 5, where the method `formatArray` is used to create a string representation of the contents of an array.

Listing 5. A generic method

```
public class MyClass {
// Other possible stuff... ignore...
  public <E> String formatArray(E[] arrayToFormat) {
    StringBuilder sb = new StringBuilder();

    int index = 0;
    for (E element : arrayToFormat) {
      sb.append("Element ");
      sb.append(index++);
      sb.append(" => ");
      sb.append(element);
      sb.append('\n');
    }

    return sb.toString();
  }
// More possible stuff... ignore...
}
```

Rather than parameterize `MyClass`, you make generic just the one method you want to use create a consistent string representation that works for any element type.

In practice, you'll find yourself using parameterized classes and interfaces far more often then methods, but now you know that the capability is available if you need it.

enum types

In JDK 5.0, a new data type was added to the Java language, called `enum` (not to be confused with `java.util.Enumeration`). The `enum` type represents a set of constant objects that are all related to a particular concept, each of which represents a different constant value in that set. Before `enum` was introduced into the language, you would have defined a set of constant values for a concept (say, gender) like so:

```
public class Person {
  public static final String MALE = "male";
  public static final String FEMALE = "female";
  public static final String OTHER = "other";
}
```

Any code that needed to reference that constant value would have been written something like this:

```
public void myMethod() {
  //. . .
  String genderMale = Person.MALE;
  //. . .
}
```

Defining constants with enum

Using the `enum` type makes defining constants much more formal — and more powerful. Here's the `enum` definition for `Gender`:

```
public enum Gender {
  MALE,
  FEMALE,
  OTHER
}
```

This example only scratches the surface of what you can do with `enum`s. In fact, `enum`s are much like classes, so they can have constructors, attributes, and methods:

```
package com.makotojava.intro;

public enum Gender {
  MALE("male"),
  FEMALE("female"),
  OTHER("other");

  private String displayName;
  private Gender(String displayName) {
    this.displayName = displayName;
  }

  public String getDisplayName() {
    return this.displayName;
  }
}
```

One difference between a class and an `enum` is that an `enum`'s constructor must be declared `private`, and it cannot extend (or inherit from) other `enum`s. However, an `enum`**can** implement an interface.

An enum implementing an interface

Suppose you define an interface, `Displayable`:

```
package com.makotojava.intro;
public interface Displayable {
  public String getDisplayName();
}
```

Your `Gender enum` could implement this interface (and any other `enum` that needed to produce a friendly display name), like so:

```
package com.makotojava.intro;

public enum Gender implements Displayable {
  MALE("male"),
  FEMALE("female"),
  OTHER("other");

  private String displayName;
  private Gender(String displayName) {
    this.displayName = displayName;
  }
  @Override
  public String getDisplayName() {
    return this.displayName;
  }
}
```

Test your understanding

1. Which statement best describes generics?
    a. Generics is built into the Java language to enable generic programming.
    b. Generics is a compiler mechanism that allows types to be specified as parameters, which results in code templates to help facilitate code reuse.
    c. Generics is a programming approach to creating reusable classes in the Java language, where you specify the object's class when instantiating it.
    d. All of the above.
2. Which of the following `for` loop statement snippets correctly uses the new generics compiler shorthand for accessing the following `String[]`?

    ```
    String[] stringArray = { "1", "2", "3", "Four" };
    ```

    a. `for (int a = 0; a < stringArray.length)`
    b. `for (int a = 0; a < stringArray.length; a++)`
    c. `for (String s : stringArray)`
    d. `for (stringArray[s] = a)`
    e. None of the above
3. The `enum` type is best described as:
    a. A new compiler shortcut to define constants, rather than declaring static final `String`s.
    b. A new data type that can have attributes, constructors, and even methods, specifically designed for defining static values that have a type that can be checked by the compiler.
    c. A new data type designed to creating numeric constants.
    d. Enums do not exist in Java.
    e. None of the above.
4. **True or false:** The Java `List` interface is an example of generics in the JDK.
5. **True or false:** Enums cannot implement an interface.
6. Define a new `enum` called `EyeColor` to represent the following eye colors:
    - Blue
    - Green
    - Brown
    - Gray
    - Gold
    - Black

Your enum should have a single constructor that takes a `String` description of the eye color. Include a method called `getDescription()` to retrieve the description.

7. Implement the `MyClass` from Listing 5, and create a JUnit test case (just to act as a test harness) to format a `String` array of four elements (whatever `String` values you like). Print the results using a JDK `Logger` class.

Check your answers.

For further exploration

Introduction to generic types in JDK 5.0

Java theory and practice: Generics gotchas

The Java - Generics

Java - Generics

IBM Code: Java journeys

Previous: Regular expressions
Next: I/O

Unit 21: I/O

Collect and manipulate external data in your Java progams

J Steven Perry

September 14, 2016

Learn the structure, syntax, and programming paradigm of the Java platform and language. Start by mastering the essentials of object-oriented programming on the Java platform, and progress incrementally to the more-sophisticated syntax and libraries that you need to develop complex, real-world Java applications.

Before you begin

This unit is part of the "Intro to Java programming" learning path. Although the concepts discussed in the individual units are standalone in nature, the hands-on component builds as you progress through the units, and I recommend that you review the prerequisites, setup, and unit details before proceeding.

Unit objectives

- Know the main uses of the `java.io.File` class
- Understand how to use byte streams and character streams
- Know how to read data from and write data to a `File`

Working with external data

More often than not, the data you use in your Java programs comes from an external data source, such as a database, direct byte transfer over a socket, or file storage. Most of the Java tools for collecting and manipulating external data are in the `java.io` package.

Files

Of all the data sources available to your Java applications, files are the most common and often the most convenient. If you want to read a file in your application, you must use *streams* that parse its incoming bytes into Java language types.

`java.io.File` is a class that defines a resource on your file system and represents that resource in an abstract way. Creating a `File` object is easy:

```
File f = new File("temp.txt");

File f2 = new File("/home/steve/testFile.txt");
```

The `File` constructor takes the name of the file it represents. The first call represents a file called temp.txt in the current directory. The second call represents a file in a specific location on my Linux system. You can pass any `String` to the constructor of `File`, provided that it's a valid file name for your operating system, whether or not the file that it references even exists.

This code asks the newly created `File` object if the file exists:

```
File f2 = new File("/home/steve/testFile.txt");
if (f2.exists()) {
  // File exists. Process it...
} else {
  // File doesn't exist. Create it...
  f2.createNewFile();
}
```

`java.io.File` has some other handy methods that you can use to:

- Delete files
- Create directories (by passing a directory name as the argument to `File`'s constructor)
- Determine if a resource is a file, directory, or symbolic link
- More

The main action of Java I/O is in writing to and reading from data sources, which is where streams come in.

Using streams in Java I/O

You can access files on the file system by using streams. At the lowest level, streams enable a program to receive bytes from a source or to send output to a destination. Some streams handle all kinds of 16-bit characters (`Reader` and `Writer` types). Others handle only 8-bit bytes (`InputStream` and `OutputStream` types). Within these hierarchies are several flavors of streams, all found in the `java.io` package.

Byte streams read (`InputStream` and subclasses) and write (`OutputStream` and subclasses) 8-bit bytes. In other words, a byte stream can be considered a more raw type of stream. Here's a summary of two common byte streams and their usage:

- `FileInputStream` / `FileOutputStream`: Reads bytes from a file, writes bytes to a file
- `ByteArrayInputStream` / `ByteArrayOutputStream`: Reads bytes from an in-memory array, writes bytes to an in-memory array

Character streams

Character streams read (`Reader` and its subclasses) and write (`Writer` and its subclasses) 16-bit characters. Here's a selected listing of character streams and their usage:

- `StringReader` / `StringWriter`: Read and write characters to and from `String`s in memory.
- `InputStreamReader` / `InputStreamWriter` (and subclasses `FileReader` / `FileWriter`): Act as a bridge between byte streams and character streams. The `Reader` flavors read bytes from a byte stream and convert them to characters. The `Writer` flavors convert characters to bytes to put them on byte streams.
- BufferedReader / BufferedWriter: Buffer data while reading or writing another stream, making read and write operations more efficient.

Rather than try to cover streams in their entirety, I'll focus here on the recommended streams for reading and writing files. In most cases, these are character streams.

Reading from a File

You can read from a `File` in several ways. Arguably the simplest approach is to:

1. Create an `InputStreamReader` on the `File` you want to read from.
2. Call `read()` to read one character at a time until you reach the end of the file.

Listing 1 is an example in reading from a `File`:

Listing 1. Reading from a `File`

```
public List<Employee> readFromDisk(String filename) {
  final String METHOD_NAME = "readFromDisk(String filename)";
  List<Employee> ret = new ArrayList<>();
  File file = new File(filename);
  try (InputStreamReader reader = new InputStreamReader(new FileInputStream(file))) {
    StringBuilder sb = new StringBuilder();
    int numberOfEmployees = 0;
    int character = reader.read();
    while (character != -1) {
        sb.append((char)character);
        character = reader.read();
    }
    log.info("Read file: \n" + sb.toString());
    int index = 0;
    while (index < sb.length()-1) {
      StringBuilder line = new StringBuilder();
      while ((char)sb.charAt(index) != '\n') {
        line.append(sb.charAt(index++));
      }
      StringTokenizer strtok = new StringTokenizer(line.toString(), Person.STATE_DELIMITER);
      Employee employee = new Employee();
      employee.setState(strtok);
      log.info("Read Employee: " + employee.toString());
      ret.add(employee);
      numberOfEmployees++;
      index++;
    }
    log.info("Read " + numberOfEmployees + " employees from disk.");
  } catch (FileNotFoundException e) {
    log.logp(Level.SEVERE, SOURCE_CLASS, METHOD_NAME, "Cannot find file " +
       file.getName() + ", message = " + e.getLocalizedMessage(), e);
  } catch (IOException e) {
    log.logp(Level.SEVERE, SOURCE_CLASS, METHOD_NAME, "IOException occurred,
       message = " + e.getLocalizedMessage(), e);
  }
  return ret;
}
```

Writing to a File

As with reading from a `File`, you have several ways to write to a `File`. Once again, I go with the simplest approach:

1. Create a `FileOutputStream` on the `File` you want to write to.
2. Call `write()` to write the character sequence.

Listing 2 is an example of writing to a `File`:

Listing 2. Writing to a `File`

```
public boolean saveToDisk(String filename, List<Employee> employees) {
  final String METHOD_NAME = "saveToDisk(String filename, List<Employee> employees)";

  boolean ret = false;
  File file = new File(filename);
  try (OutputStreamWriter writer = new OutputStreamWriter(new FileOutputStream(file))) {
    log.info("Writing " + employees.size() + " employees to disk (as String)...");
    for (Employee employee : employees) {
      writer.write(employee.getState()+"\n");
    }
    ret = true;
    log.info("Done.");
  } catch (FileNotFoundException e) {
    log.logp(Level.SEVERE, SOURCE_CLASS, METHOD_NAME, "Cannot find file " +
       file.getName() + ", message = " + e.getLocalizedMessage(), e);
  } catch (IOException e) {
    log.logp(Level.SEVERE, SOURCE_CLASS, METHOD_NAME, "IOException occurred,
       message = " + e.getLocalizedMessage(), e);
  }
  return ret;
}
```

Buffering streams

Reading and writing character streams one character at a time isn't efficient, so in most cases you probably want to use buffered I/O instead. To read from a file using buffered I/O, the code looks just like Listing 1, except that you wrap the `InputStreamReader` in a `BufferedReader`, as shown in Listing 3.

Listing 3. Reading from a `File` with buffered I/O

```
public List<Employee> readFromDiskBuffered(String filename) {
  final String METHOD_NAME = "readFromDisk(String filename)";
  List<Employee> ret = new ArrayList<>();
  File file = new File(filename);
  try (BufferedReader reader = new BufferedReader(new InputStreamReader(new FileInputStream(file)))) {
    String line = reader.readLine();
    int numberOfEmployees = 0;
    while (line != null) {
      StringTokenizer strtok = new StringTokenizer(line, Person.STATE_DELIMITER);
      Employee employee = new Employee();
      employee.setState(strtok);
      log.info("Read Employee: " + employee.toString());
      ret.add(employee);
      numberOfEmployees++;
      // Read next line
```

```
      line = reader.readLine();
    }
    log.info("Read " + numberOfEmployees + " employees from disk.");
  } catch (FileNotFoundException e) {
    log.logp(Level.SEVERE, SOURCE_CLASS, METHOD_NAME, "Cannot find file " +
       file.getName() + ", message = " + e.getLocalizedMessage(), e);
  } catch (IOException e) {
    log.logp(Level.SEVERE, SOURCE_CLASS, METHOD_NAME, "IOException occurred,
       message = " + e.getLocalizedMessage(), e);
  }
  return ret;
}
```

Writing to a file using buffered I/O is the same: You wrap the `OutputStreamWriter` in a `BufferedWriter`, as shown in Listing 4.

Listing 4. Writing to a `File` with buffered I/O

```
public boolean saveToDiskBuffered(String filename, List<Employee> employees) {
  final String METHOD_NAME = "saveToDisk(String filename, List<Employee> employees)";

  boolean ret = false;
  File file = new File(filename);
  try (BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(new FileOutputStream(file)))) {
    log.info("Writing " + employees.size() + " employees to disk (as String)...");
    for (Employee employee : employees) {
      writer.write(employee.getState()+"\n");
    }
    ret = true;
    log.info("Done.");
  } catch (FileNotFoundException e) {
    log.logp(Level.SEVERE, SOURCE_CLASS, METHOD_NAME, "Cannot find file " +
       file.getName() + ", message = " + e.getLocalizedMessage(), e);
  } catch (IOException e) {
    log.logp(Level.SEVERE, SOURCE_CLASS, METHOD_NAME, "IOException occurred,
       message = " + e.getLocalizedMessage(), e);
  }
  return ret;
}
```

Test your understanding

1. **True or false:** A file must exist on disk before you can create a `File` object to represent it.
2. Which statement best describes *character streams*?
   a. Character streams are 8-bit streams that are used to read and write from files into memory through `ByteArrayProcessor` interfaces.
   b. Character streams should not be used to read binary data.
   c. Character streams are mainly used to read and write text files.
   d. Character streams are 16-bit streams that are used to read and write data from files through `Reader` and `Writer` subclasses.
   e. None of the above
3. Which statement best describes why you might use a `BufferedReader`?
   a. Using a `BufferedReader` to wrap an `OutputStreamWriter` helps it process 16-bit character streams more efficiently by buffering the input and output.
   b. When using a `StreamReader` and `StreamWriter` together, you must be careful not to cross the streams.

   c. A `BufferedReader`, when acting as a wrapper for `InputStreamReader`, helps it process 16-bit character streams more efficiently by buffering the input.
   d. A `BufferedReader` should never be used to read from an input stream.
   e. None of the above
4. Create a file called lorem.txt that contains the first 250 words of *lorem ipsem* (you can use generator.lorem-ipsum.info or similar websites to generate this text). Save the file to the root directory of your Java project. Now write a class called `Unit21` with a method called `readFile()` to read the file, print out its contents using a JDK `Logger` instance, and return the `String` (containing the file's contents) to the caller.
   Write a JUnit test class called `Unit21Test` as a test harness.
   Issues to consider:
   - How will you specify the file to be read by `readFile()`?
   - How will you handle exceptions?
   - How will you make sure to close any file resources when you're finished?
5. Add a method to your `Unit21` class from Question 4 called `writeFile()`, which writes the file you read in to a new file called lorem2.txt. Add a new test method to `Unit21Test` as a test harness. **Note:** The same types of issues that you addressed in your solution to Question 4 also apply to output streams.
6. Augment your solution to Question 4 so that each line is no longer than `maxCharactersPerLine` characters long. (Don't worry about trying to preserve words when you reach the `maxCharactersPerLine`th character.) Write your JUnit test to specify 80 as the value for `maxCharactersPerLine`.
7. Augment your solution to Question 6 so that you do not truncate a word that occurs when you hit the `maxCharactersPerLine`th character. Instead, output that word (and any words that follow) on the next line. This is similar to what a word processor does. Write your JUnit test to specify 80 as the value for `maxCharactersPerLine`. **Hint:** it might be easier to process the current line one word at a time, and check to see if the current word causes the line to exceed `maxCharactersPerLine`, and if so put that word on the next line.

Check your answers.

For further exploration

The Java Tutorials: I/O Streams

Java I/O Tutorial

Java in a Nutshell, 6th ed. by Benjamin J. Evans and David Flanagan (see Chapter 10, "File Handling and I/O")

Reading a plain text file in Java

Reading a binary input stream into a single byte array in Java

IBM Code: Java journeys

Previous: Generics

Next: Java serialization

Unit 22: Java serialization

Store object state in binary format

J Steven Perry

September 14, 2016

Learn the structure, syntax, and programming paradigm of the Java platform and language. Start by mastering the essentials of object-oriented programming on the Java platform, and progress incrementally to the more-sophisticated syntax and libraries that you need to develop complex, real-world Java applications.

Before you begin

This unit is part of the "Intro to Java programming" learning path. Although the concepts discussed in the individual units are standalone in nature, the hands-on component builds as you progress through the units, and I recommend that you review the prerequisites, setup, and unit details before proceeding.

Unit objectives

- Learn what object serialization is and why you'd use it
- Know the syntax for making objects serializable, serializing objects, and deserializing objects
- Be able to deal with different versions of the same object in a serialization scenario

What is object serialization?

Serialization is a process whereby the state of an object and its metadata (such as the object's class name and the names of its attributes) are stored in a special binary format. Putting the object into this format —*serializing* it — preserves all the information necessary to reconstitute (or *deserialize*) the object whenever you need to do so.

The two primary use cases for object serialization are:

- *Object persistence*— storing the object's state in a permanent persistence mechanism such as a database
- *Object remoting*— sending the object to another computer or system

java.io.Serializable

The first step in making serialization work is to enable your objects to use the mechanism. Every object you want to be serializable must implement an interface called `java.io.Serializable`:

```
import java.io.Serializable;
public class Person implements Serializable {
  // etc...
}
```

In this example, the `Serializable` interface marks the objects of the `Person` class — and every subclass of `Person`— to the runtime as `serializable`.

Any attributes of an object that are not serializable cause the Java runtime to throw a `NotSerializableException` if it tries to serialize your object. You can manage this behavior by using the `transient` keyword to tell the runtime not to try to serialize certain attributes. In that case, you are responsible for making sure that the attributes are restored (if necessary) so that your object functions correctly.

Serializing an object

Now, try an example that combines what you learned about Java I/O in Unit 21 with what you're learning now about serialization.

Suppose you create and populate a `List` of `Employee` objects and then want to serialize that `List` to an `OutputStream`, in this case to a file. That process is shown in Listing 1.

Listing 1. Serializing an object

```
public class HumanResourcesApplication {
  private static final Logger log = Logger.getLogger(HumanResourcesApplication.class.getName());
  private static final String SOURCE_CLASS = HumanResourcesApplication.class.getName();

  public static List<Employee> createEmployees() {
    List<Employee> ret = new ArrayList<Employee>();
    Employee e = new Employee("Jon Smith", 45, 175, 75, "BLUE", Gender.MALE,
       "123-45-9999", "0001", BigDecimal.valueOf(100000.0));
    ret.add(e);
    //
    e = new Employee("Jon Jones", 40, 185, 85, "BROWN", Gender.MALE, "223-45-9999",
       "0002", BigDecimal.valueOf(110000.0));
    ret.add(e);
    //
    e = new Employee("Mary Smith", 35, 155, 55, "GREEN", Gender.FEMALE, "323-45-9999",
       "0003", BigDecimal.valueOf(120000.0));
    ret.add(e);
    //
    e = new Employee("Chris Johnson", 38, 165, 65, "HAZEL", Gender.UNKNOWN,
       "423-45-9999", "0004", BigDecimal.valueOf(90000.0));
    ret.add(e);
    // Return list of Employees
    return ret;
  }

  public boolean serializeToDisk(String filename, List<Employee> employees) {
    final String METHOD_NAME = "serializeToDisk(String filename, List<Employee> employees)";

    boolean ret = false;// default: failed
    File file = new File(filename);
    try (ObjectOutputStream outputStream = new ObjectOutputStream(new FileOutputStream(file))) {
      log.info("Writing " + employees.size() + " employees to disk (using Serializable)...");
      outputStream.writeObject(employees);
      ret = true;
      log.info("Done.");
```

```
    } catch (IOException e) {
      log.logp(Level.SEVERE, SOURCE_CLASS, METHOD_NAME, "Cannot find file " +
       file.getName() + ", message = " + e.getLocalizedMessage(), e);
    }
    return ret;
  }
```

The first step is to create the objects, which is done in `createEmployees()` using the specialized constructor of `Employee` to set some attribute values. Next, you create an `OutputStream`— in this case, a `FileOutputStream`— and then call `writeObject()` on that stream. `writeObject()` is a method that uses Java serialization to serialize an object to the stream.

In this example, you are storing the `List` object (and its contained `Employee` objects) in a file, but this same technique is used for any type of serialization.

To drive the code in Listing 1, you could use a JUnit test, as shown here:

```
public class HumanResourcesApplicationTest {

  private HumanResourcesApplication classUnderTest;
  private List<Employee> testData;

  @Before
  public void setUp() {
    classUnderTest = new HumanResourcesApplication();
    testData = HumanResourcesApplication.createEmployees();
  }
  @Test
  public void testSerializeToDisk() {
    String filename = "employees-Junit-" + System.currentTimeMillis() + ".ser";
    boolean status = classUnderTest.serializeToDisk(filename, testData);
    assertTrue(status);
  }

}
```

Deserializing an object

The whole point of serializing an object is to be able to reconstitute, or deserialize, it. Listing 2 reads the file you've just serialized and deserializes its contents, thereby restoring the state of the `List` of `Employee` objects.

Listing 2. Deserializing objects

```
public class HumanResourcesApplication {

  private static final Logger log = Logger.getLogger(HumanResourcesApplication.class.getName());
  private static final String SOURCE_CLASS = HumanResourcesApplication.class.getName();

  @SuppressWarnings("unchecked")
  public List<Employee> deserializeFromDisk(String filename) {
    final String METHOD_NAME = "deserializeFromDisk(String filename)";

    List<Employee> ret = new ArrayList<>();
    File file = new File(filename);
    int numberOfEmployees = 0;
    try (ObjectInputStream inputStream = new ObjectInputStream(new FileInputStream(file))) {
      List<Employee> employees = (List<Employee>)inputStream.readObject();
      log.info("Deserialized List says it contains " + employees.size() +
         " objects...");
```

```
      for (Employee employee : employees) {
        log.info("Read Employee: " + employee.toString());
        numberOfEmployees++;
      }
      ret = employees;
      log.info("Read " + numberOfEmployees + " employees from disk.");
    } catch (FileNotFoundException e) {
      log.logp(Level.SEVERE, SOURCE_CLASS, METHOD_NAME, "Cannot find file " +
         file.getName() + ", message = " + e.getLocalizedMessage(), e);
    } catch (IOException e) {
      log.logp(Level.SEVERE, SOURCE_CLASS, METHOD_NAME, "IOException occurred,
       message = " + e.getLocalizedMessage(), e);
    } catch (ClassNotFoundException e) {
      log.logp(Level.SEVERE, SOURCE_CLASS, METHOD_NAME, "ClassNotFoundException,
         message = " + e.getLocalizedMessage(), e);
    }
    return ret;
  }

}
```

Again, to drive the code in Listing 2, you could use a JUnit test like this one:

```
public class HumanResourcesApplicationTest {

  private HumanResourcesApplication classUnderTest;

  private List<Employee> testData;

  @Before
  public void setUp() {
    classUnderTest = new HumanResourcesApplication();
  }

  @Test
  public void testDeserializeFromDisk() {
    String filename = "employees-Junit-" + System.currentTimeMillis() + ".ser";
    int expectedNumberOfObjects = testData.size();
    classUnderTest.serializeToDisk(filename, testData);
    List<Employee> employees = classUnderTest.deserializeFromDisk(filename);
    assertEquals(expectedNumberOfObjects, employees.size());
  }

}
```

For most application purposes, marking your objects as `serializable` is all you ever need to worry about when it comes to serialization. When you do need to serialize and deserialize your objects explicitly, you can use the technique shown in Listing 1 and Listing 2. But as your application objects evolve, and you add and remove attributes to and from them, serialization takes on a new layer of complexity.

serialVersionUID

In the early days of middleware and remote object communication, developers were largely responsible for controlling the "wire format" of their objects, which caused no end of headaches as technology began to evolve.

Suppose you added an attribute to an object, recompiled it, and redistributed the code to every computer in an application cluster. The object would be stored on a computer with one version of the serialization code but accessed by other computers that might have a different version of the code. When those computers tried to deserialize the object, bad things often happened.

Java serialization metadata — the information included in the binary serialization format — is sophisticated and solves many of the problems that plagued early middleware developers. But it can't solve every problem.

Java serialization uses a property called `serialVersionUID` to help you deal with different versions of objects in a serialization scenario. You don't need to declare this property on your objects; by default, the Java platform uses an algorithm that computes a value for it based on your class's attributes, its class name, and position in the local galactic cluster. Most of the time, that algorithm works fine. But if you add or remove an attribute, that dynamically generated value changes, and the Java runtime throws an `InvalidClassException`.

To avoid this outcome, get in the habit of explicitly declaring a `serialVersionUID`:

```
import java.io.Serializable;
  public class Person implements Serializable {
  private static final long serialVersionUID = 20100515;
  // etc...
  }
```

I recommend using a scheme of some kind for your `serialVersionUID` version number (I've used the current date in the preceding example). And you should declare `serialVersionUID` as `private static final` and of type `long`.

You might be wondering when to change this property. The short answer is that you should change it whenever you make an incompatible change to the class, which usually means you've added or removed an attribute. If you have one version of the object on one computer that has the attribute added or removed, and the object gets remoted to a computer with a version of the object where the attribute is either missing or expected, things can get weird. This is where the Java platform's built-in `serialVersionUID` check comes in handy.

As a rule of thumb, any time you add or remove features (meaning attributes or any other instance-level state variables) of a class, change its `serialVersionUID`. Better to get a `java.io.InvalidClassException` on the other end of the wire than an application bug that's caused by an incompatible class change.

Test your understanding

1. How do you make a class "serializable"?
    a. There's no special requirement for using Java serialization; it is built right into the Java language.
    b. Every class that needs to be serialized must, at a minimum, implement the `java.io.Serializable` interface.
    c. To use Java serialization, you must implement `hashCode()` and `equals()`.
    d. Attribute values must be declared in alphabetical order or else they cannot be properly serialized and deserialized.

   e. None of the above
2. What happens if you try to serialize an object for which no `serialVersionUID` is declared on its class?
   a. The Java serialization runtime will compute a default value for this field based on the declaration of the class, and serialization proceeds normally.
   b. You'll get a `NotSerializableException` when you try to serialize the object.
   c. The JVM will throw a `NoSerialVersionUIDFieldDeclared` exception when the class is loaded into memory.
   d. The object will be serialized, but it cannot be deserialized — as a security measure, in case an incompatible class change was made.
   e. None of the above
3. Choose the correct declaration of `serialVersionUID`:
   a. `public static final String serialVersionUID = "1";`
   b. `private long serialVersionUID = 1L;`
   c. `private static final long serialVersionUID = 12345L;`
   d. `private static final String serialVersionUID = "Sdflkjsdfgd0980980(DF)(*)90";`
   e. None of the above
4. Under what conditions would you want to regenerate `serialVersionUID` for one of your classes as a matter of best practice?
   a. `serialVersionUID` is generated automatically, so it never requires regenerating.
   b. If your class implements more than one method and three attributes, it's probably a good idea.
   c. If you make any change to the class that makes a previous instance of the class incompatible, you should regenerate `serialVersionUID` so as not to unwittingly introduce bugs into your code.
   d. If you make a change to the class, even if you are certain the change is compatible with any previous instance of the class, you should regenerate `serialVersionUID` just to be safe.
   e. None of the above
5. Suppose you need to add an attribute to your class that doesn't need to be serialized. How do you let the Java serialization runtime know?
   a. You don't need to. The serialization runtime will automatically detect and handle the unimportant attribute.
   b. You declare the attribute using the `ignoreSerial` keyword.
   c. You must write custom code to handle such attributes when they are deserialized.
   d. You declare the attribute using the `transient` keyword.
   e. None of the above
6. Create two classes:
   - `Container`
   - `Contained`

   Make both classes serializable. Each must contain a single read-only attribute: `name`, a `String`. Generate a getter for the attribute. `Container` has an additional read-only attribute: `contained`, which is a `Contained` instance. Generate a getter for this attribute. Initialize the

`name` attribute in each class's constructor. Initialize the `Contained` instance in `Container`'s constructor.
Write a JUnit test harness called `testContainer_problem6()` that creates an instance of `Container` (set the name to anything you like) and serializes it to a disk file called Container.ser.
Refresh your project and make sure the file shows up.

7. Write a separate JUnit test case called `testContainer_problem7()` to read in the serialized object from Question 6, and verify that the contents are as you expect.
8. Add a new `String` attribute to `Container` and set it to anything you like. Then rerun `testContainer_problem7()`**only** (it is important that you do **not** regenerate the Container.ser). What do you expect will happen? Explain your answer.

Check your answers.

For further exploration

5 things you didn't know about Java Object Serialization

The Java Tutorials: Serializable Objects

What is a serialVersionUID and why should I use it?

Understand the serialVersionUID

IBM Code: Java journeys

Previous: I/O
Next: Java in the cloud

Unit 23: Java in the cloud

Deploy Java applications to IBM Cloud

J Steven Perry

July 12, 2018

Learn the structure, syntax, and programming paradigm of the Java platform and language. Start by mastering the essentials of object-oriented programming on the Java platform, and progress incrementally to the more-sophisticated syntax and libraries that you need to develop complex, real-world Java applications.

Before you begin

This unit is part of the "Intro to Java programming" learning path. Although the concepts discussed in the individual units are standalone in nature, the hands-on component builds as you progress through the units, and I recommend that you review the prerequisites, setup, and unit details before proceeding.

Unit objectives

- Download, install, and set up the WebSphere® Liberty application server
- Install the Eclipse Tools for IBM Cloud
- Test the sample application locally from Eclipse
- Push the application to IBM Cloud from Eclipse
- Modify the app, retest it locally, and redeploy it to the cloud

Cloud development with Java

So far, you've only run the human resources (HR) application code on your own computer, from within Eclipse. But browser-based business applications are now common — and it's becoming increasingly convenient to deploy such apps in the cloud. No doubt you're curious where Java programming fits into the cloud computing picture. This unit aims to satisfy that curiosity by showing how to deploy a web version of the HR app in the cloud using IBM Cloud and WebSphere Liberty — all without needing to leave the comfort of Eclipse.

Java web programming rightly deserves its own specialized treatment, and you'll get only a small taste here. To make it easy for you to follow along hands-on, I've made the source code for a web version of HR app available on GitHub. (If you explore the project's src/com/makotojava/intro/ folder, you'll find `Person`, `Employee`, `Manager`, and all of the other Java classes that you've been working with.)

The steps that you'll perform follow a typical development cycle: Develop and test an app locally, push it to the cloud, modify and retest the app locally as needed, and redeploy it to the cloud.

Set up the tools, test the app locally, and push it to IBM Cloud

If you don't already have an IBM Cloud account, register now.

Next, pull the code from my GitHub project and import it into your Eclipse workspace. Then follow along with this video demo:

To view this video, **Java in the cloud** , please access the online version of the article. If this article is in the developerWorks archives, the video is no longer accessible.

Good work! You've deployed your application to the cloud.

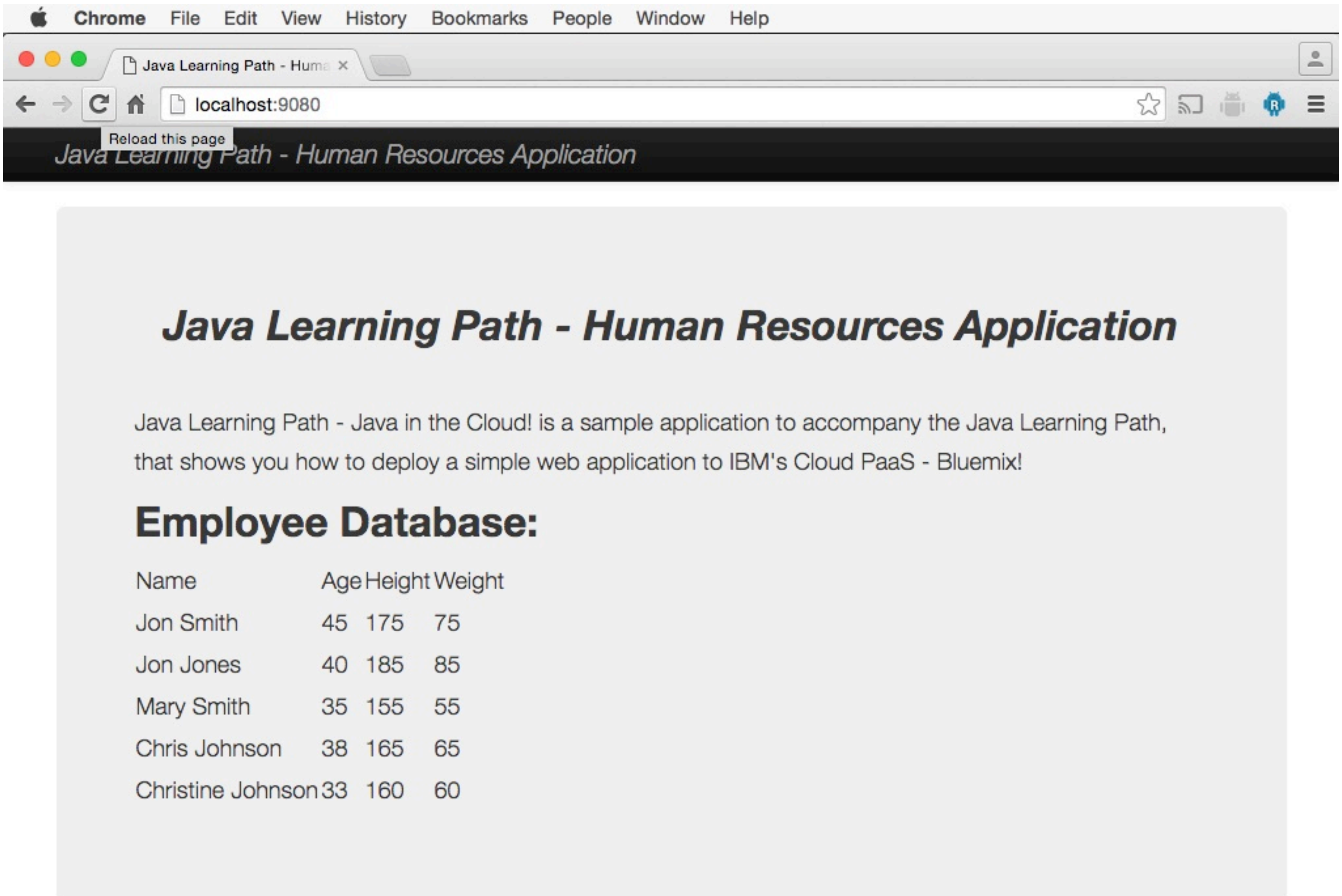
Modify the application and test it locally

Of course, as software developers, we know that deployed applications must be maintained through bug fixes and enhancements. You'll simulate that process now by making a small change to the HR app and then testing the change locally in your Liberty server before redeploying to the cloud.

In your Eclipse project, uncomment the following lines of code in the `createEmployees()` method in HumanResourcesApplication.java:

```
//
e = new Employee("Christine Johnson", 33, 160, 60, "BROWN", Gender.FEMALE, "424-45-9999", "0005",
 BigDecimal.valueOf(190000.0));
ret.add(e);
```

Now test the change locally: Start your local Liberty server, bring up your browser, and verify that a new row that includes **Christine Johnson** is displayed in the app:



Push your modifications to the cloud

Now that you've tested the change locally, it's time to push the modified app to the cloud, just as you would for a real-world application.

Stop your local Liberty server. Then follow the same deployment steps as before — except this time, all you need to do is specify the IBM Cloud server to deploy to (select the one you set up earlier, if it isn't automatically selected). Then click **OK**.

You might notice quite a bit more output this time as IBM Cloud updates your application. After the app is deployed, you'll see a message similar to this one (it might differ slightly):

```
[AUDIT   ] CWWKF0011I: The server defaultServer is ready to run a smarter planet.
[INFO    ] CWWKF0008I: Feature update completed in 23.875 seconds.
[Application Running Check] - Application appears to be running - javalearningpath.
```

Now go to your browser and access the application at the **Deployed URL** and verify that your change is in the cloud version.

IBM Cloud is a powerful platform, and I've only scratched the surface of its capabilities in this unit. I encourage you to check out the resources under For further exploration to learn more about IBM Cloud.

Fini

Kudos! You've completed the "Intro to Java programming" learning path.

For further exploration

Get started with IBM Cloud

IBM Cloud container tutorials

Tutorials on IBM Cloud services

IBM Code

Previous: Java serialization
Next: Learning path summary