Introduction to Java programming, Part 1: Java language basics

Object-oriented programming on the Java platform

J Steven Perry

August 24, 2017
(First published July 19, 2010)

Get an introduction to the structure, syntax, and programming paradigm of the Java™ language and platform in this two-part tutorial. Learn the Java syntax that youre most likely to encounter professionally and Java programming idioms you can use to build robust, maintainable Java applications. In Part 1, master the essentials of object-oriented programming on the Java platform, including fundamental Java syntax. Get started with creating Java objects and adding behavior to them, and conclude with a summary of Java coding best practices, covering ample ground in-between.

View more content in this series

Find out what to expect from this tutorial and how to get the most out of it.

About this tutorial

The two-part *Introduction to Java programming* tutorial is meant for software developers who are new to Java technology. Work through both parts to get up and running with object-oriented programming (OOP) and real-world application development using the Java language and platform.

This first part is a step-by-step introduction to OOP using the Java language. The tutorial begins with an overview of the Java platform and language, followed by instructions for setting up a development environment consisting of a Java Development Kit (JDK) and the Eclipse IDE. After you're introduced to your development environment's components, you begin learning basic Java syntax hands-on.

Part 2 covers more-advanced language features, including regular expressions, generics, I/O, and serialization. Programming examples in Part 2 build on the `Person` object that you begin developing in Part 1.

Objectives

When you finish Part 1, you'll be familiar with basic Java language syntax and able to write simple Java programs. Follow up with "Introduction to Java programming, Part 2: Constructs for real-world applications" to build on this foundation.

Prerequisites

This tutorial is for software developers who are not yet experienced with Java code or the Java platform. The tutorial includes an overview of OOP concepts.

System requirements

To complete the exercises in this tutorial, you will install and set up a development environment consisting of:

- JDK 8 from Oracle
- Eclipse IDE for Java Developers

Download and installation instructions for both are included in the tutorial.

The recommended system configuration is:

- 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.

Java platform overview

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

The Java language

Get to know the Java APIs

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

Like any programming language, the Java language has its own structure, syntax rules, and programming paradigm. The Java language's programming paradigm is based on the concept of OOP, which the language's features support.

The Java language is a C-language derivative, so its syntax rules look much like C's. For example, code blocks are modularized into methods and delimited by braces (`{` and `}`), and variables are declared before they are used.

Structurally, the Java language starts with *packages*. A package is the Java language's namespace mechanism. Within packages are classes, and within classes are methods, variables, constants, and more. You learn about the parts of the Java language in this tutorial.

The Java compiler

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

The JVM

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

The garbage collector

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

The Java Development Kit

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

The Java Runtime Environment

The Java Runtime Environment (JRE; also known as the Java runtime) includes the JVM, code libraries, and components that are necessary for running programs that are written in the Java language. The JRE is available for multiple platforms. You can freely redistribute the JRE with your applications, according to the terms of the JRE license, to give the application's users a platform on which to run your software. The JRE is included in the JDK.

Setting up your Java development environment

In this section, you'll download and install the JDK and the current release of the Eclipse IDE, and you'll set up your Eclipse development environment.

If you already have the JDK and Eclipse IDE installed, you might want to skip to the "Getting started with Eclipse" section or to the one after that, "Object-oriented programming concepts."

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, most developers appreciate the additional functionality, task management, and visual interface of an IDE.

Eclipse is a popular open source IDE for Java development. 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. If you download one of the Eclipse bundles, it will come with the JDK already.

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.

Install Eclipse

To download and install Eclipse, follow these steps:

1. Browse to the Eclipse packages downloads page.
2. Click **Eclipse IDE for Java Developers**.
3. Under Download Links on the right side, choose your platform (the site might already have sniffed out your OS type).
4. Click the mirror you want to download from; then, save the file to your hard drive.
5. When the download finishes, open the file and run the installation program, accepting the defaults.

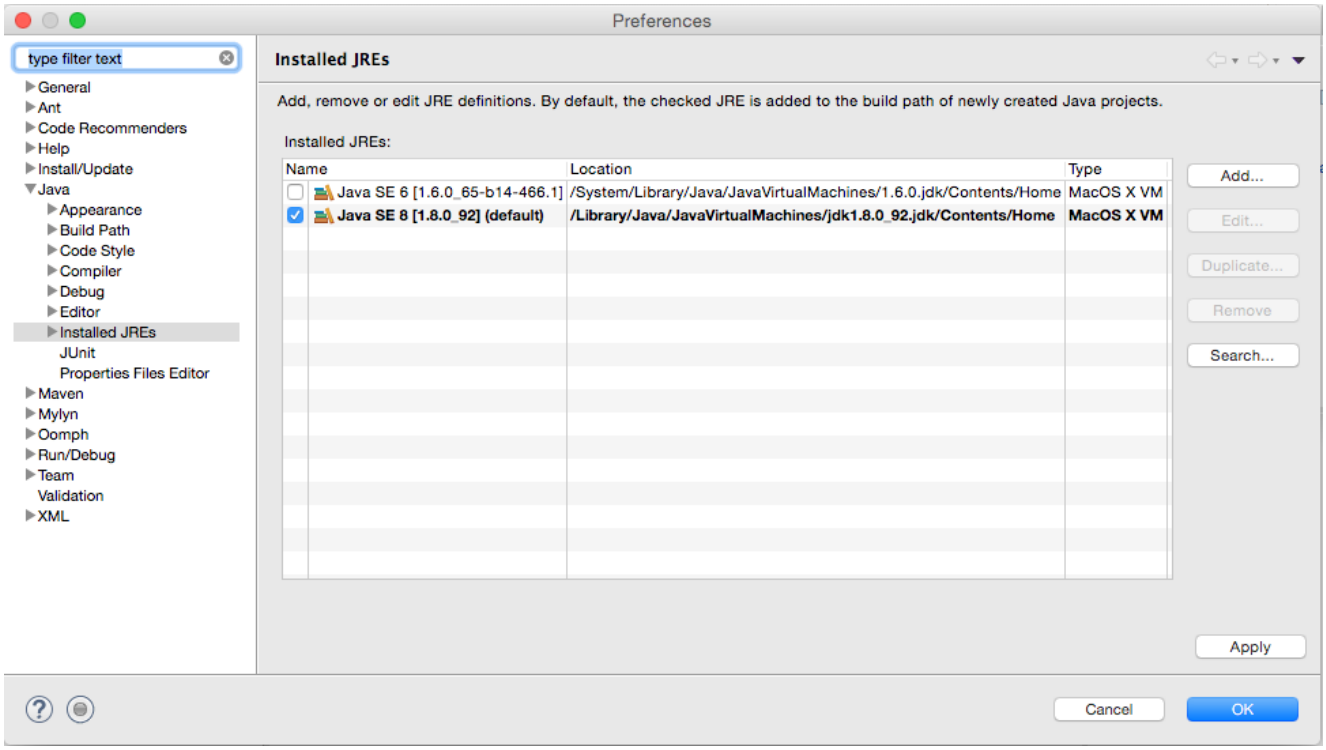
Set up Eclipse

The Eclipse IDE sits atop the JDK as a useful abstraction, but it still needs to access the JDK and its various tools. Before you can use Eclipse to write Java code, you must tell it where the JDK is located.

To set up your Eclipse development environment:

1. Launch Eclipse from your local hard disk. (In my case, the location is /Users/sperry/eclipse/java-neon.)
2. When asked which workspace you want to open, choose the default.
3. Close the Welcome to Eclipse window. (The welcome window is displayed each time you enter a new workspace. You can disable this behavior by deselecting the "Always show Welcome at start up" check box.)
4. Select **Preferences > Java > Installed JREs**. Figure 1 shows this selection highlighted in the Eclipse setup window for the JRE.

Figure 1. Configuring the JDK that Eclipse uses



5. Make sure that Eclipse points to the JRE that you downloaded with the JDK. If Eclipse does not automatically detect the JDK that you installed, click **Add...**, and in the next dialog box, click **Standard VM** and then click **Next**.
6. Specify the JDK's home directory (such as C:\home\jdk1.8.0_92 on Windows), and then click **Finish**.
7. Confirm that the JDK that you want to use is selected and click **OK**.

Eclipse is now set up and ready for you to create projects, and compile and run Java code. The next section familiarizes you with Eclipse.

Getting started with Eclipse

Eclipse is more than an IDE; it's an entire development ecosystem. This section is a brief hands-on introduction to using Eclipse for Java development.

The Eclipse development environment

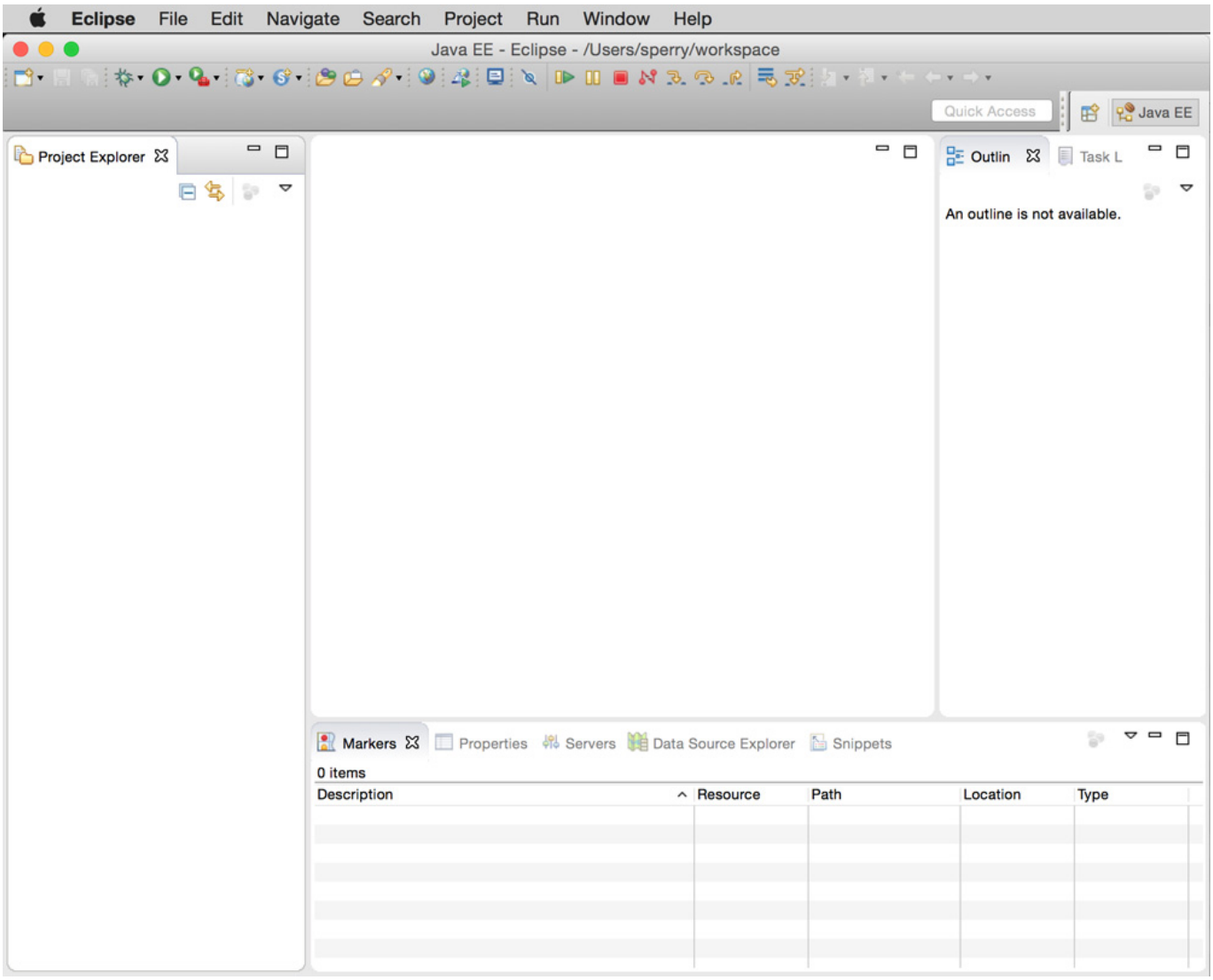
The Eclipse development environment has four main components:

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

Figure 2. Eclipse Java perspective



The Java perspective contains the tools that you need to begin writing Java applications. Each tabbed window shown in Figure 2 is a view for the Java perspective. Package Explorer and Outline are two particularly useful views.

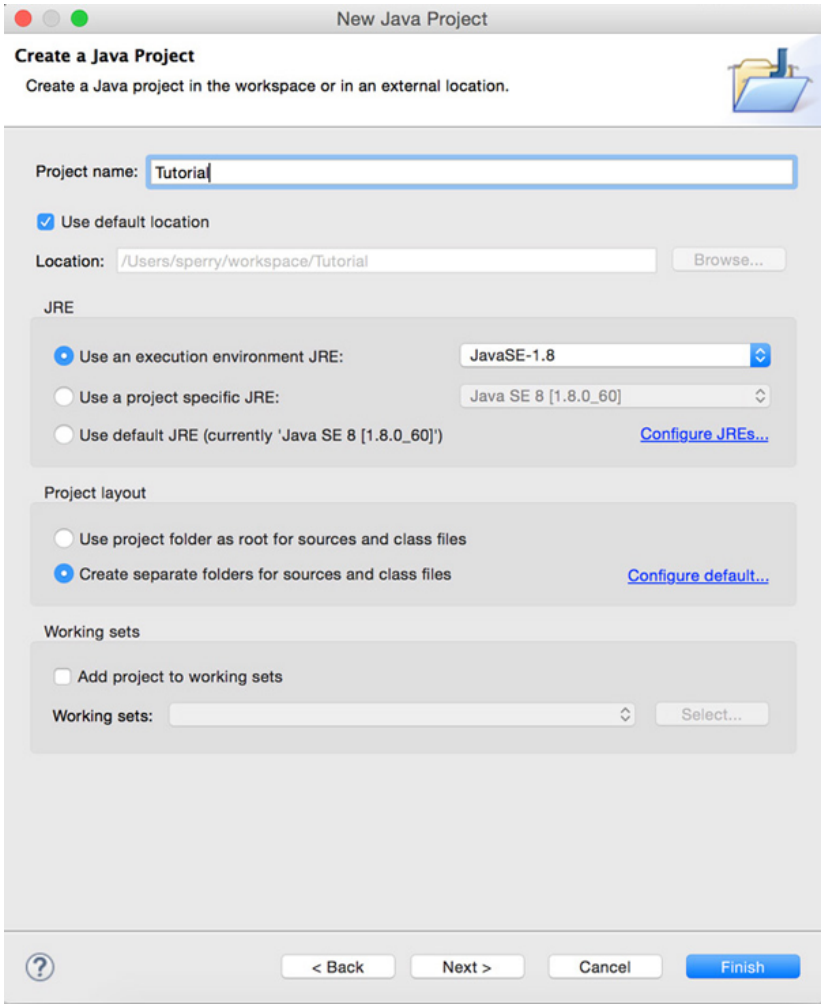
The Eclipse environment is highly configurable. Each view is dockable, so you can move it around in the Java perspective and place it where you want it. For now, though, stick with the default perspective and view setup.

Create a project

Follow these steps to create a new Java project:

1. Click **File > New > Java Project...** to start the New Java Project wizard, shown in Figure 3.

Figure 3. New Java Project wizard



2. Enter `Tutorial` as the project name and use the workspace location that you opened when you opened Eclipse.
3. Verify the JDK that you're using.
4. Click **Finish** to accept the project setup and create the project.

You have 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 this tutorial's next section — is essential.

Object-oriented programming concepts and principles

The Java language is (mostly) object oriented. This section is an introduction to OOP language concepts, using structured programming as a point of contrast.

What is an object?

Object-oriented languages follow a different programming pattern from structured programming languages like C and COBOL. The structured-programming paradigm is highly data oriented: You

have data structures, and then program instructions act on that data. Object-oriented languages such as the Java language combine data and program instructions into *objects*.

An object is a self-contained entity that contains attributes and behavior, and nothing more. Instead of having a data structure with fields (attributes) and passing that structure around to all of the program logic that acts on it (behavior), in an object-oriented language, data and program logic are combined. This combination can occur at vastly different levels of granularity, from fine-grained objects such as a `Number`, to coarse-grained objects, such as a `FundsTransfer` service in a large banking application.

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

This 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.

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 tutorial. You'll explore encapsulation and inheritance in more depth in subsequent sections.

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 the "Writing good Java code" section.

Getting started with the Java language

It would be impossible to introduce the entire Java language syntax in a single tutorial. The remainder of Part 1 focuses on the basics of the language, leaving you with enough knowledge and practice to write simple programs. OOP is all about objects, so this section starts with two topics specifically related to how the Java language handles them: reserved words and the structure of a Java object.

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.

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 section, 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 tutorial, 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 section and continues throughout the tutorial.

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 this tutorial, 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.

Your first Java class

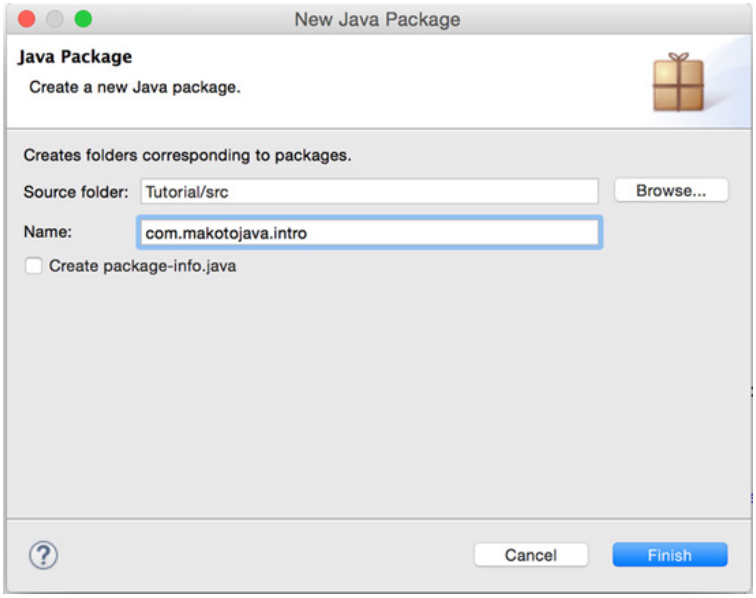
It's time to pull together what you've learned in the previous sections and start writing some code. This section walks you through declaring a class and adding variables and methods to it using the Eclipse Package Explorer. You learn how to use the `Logger` class to keep an eye on your application's behavior, and also how to use a `main()` method as a **test harness**.

Creating a package

If you're not already there, get to the Package Explorer view (in the Java perspective) in Eclipse through **Window > Perspective > Open Perspective**. You're going to get set up to create your first Java class. The first step is to create a place for the class to live. Packages are namespace constructs, and they also conveniently map directly to the file system's directory structure.

Rather than use the default package (almost always a bad idea), you create one specifically for the code you are writing. Click **File > New > Package** to start the Java Package wizard, shown in Figure 4.

Figure 4. The Eclipse Java Package wizard



Type `com.makotojava.intro` into the Name text box and click **Finish**. You can see the new package created in the Package Explorer.

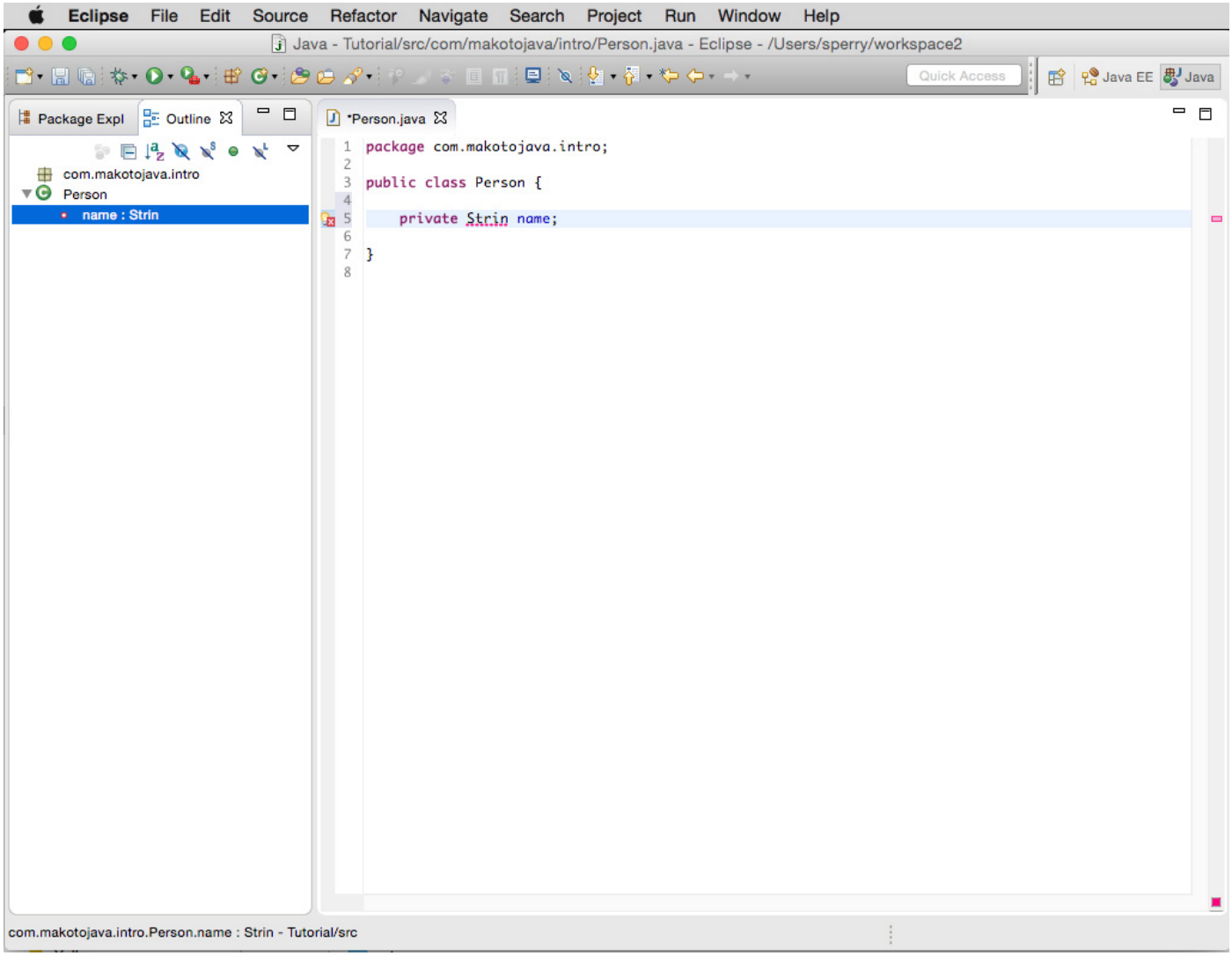
Declaring the class

You can create a class from the Package Explorer in more than one way, but the easiest way is to right-click the package you just created and choose **New > Class...**. The New Class dialog box opens.

In the Name text box, type `Person` and then click **Finish**.

The new class is displayed in your edit window. I recommend closing a few of the views (Problems, Javadoc, and others) that open by default in the Java Perspective the first time you open it to make it easier to see your source code. (Eclipse remembers that you don't want to see those views the next time you open Eclipse and go to the Java perspective.) Figure 5 shows a workspace with the essential views open.

Figure 5. A well-ordered workspace



Eclipse generates a shell class for you and includes the `package` statement at the top. You just need to flesh out the class now. You can configure how Eclipse generates new classes through **Window > Preferences > Java > Code Style > Code Templates**. For simplicity, go with Eclipse's out-of-the-box code generation.

In Figure 5, notice the asterisk (*) next to the new source-code file name, indicating that I've made a modification. And notice that the code is unsaved. Next, notice that I made a mistake when declaring the `Name` attribute: I declared `Name`'s type to be `Strin`. The compiler could not find a reference to such a class and flagged it as a compile error (that's the wavy red line underneath `Strin`). Of course, I can fix my mistake by adding a `g` to the end of `Strin`. This is a small demonstration of the power of using an IDE instead of command-line tools for software development. Go ahead and correct the error by changing the type to `String`.

Adding class variables

In Listing 3, you began to flesh out the `Person` class, but I didn't explain much of the syntax. Now, I'll formally define how to add class variables.

Recall that a variable has an *`accessSpecifier`*, a *`dataType`*, a *`variableName`*, and, optionally, an *`initialValue`*. Earlier, you looked briefly at how to define the *`accessSpecifier`* and *`variableName`*. Now, you see the *`dataType`* that a variable can have.

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

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

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

Built-in logging

Before going further into coding, you need to know how your programs tell you what they are doing.

The Java platform includes the `java.util.logging` package, a built-in logging mechanism for gathering program information in a readable form. Loggers are named entities that you create through a static method call to the `Logger` class:

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

When calling the `getLogger()` method, you pass it a `String`. For now, just 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 are making a `Logger` call inside of a static method, reference the name of the class you're inside of:

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

In this example, the code you're inside of is the `Person` class, so you reference a special literal called `class` that retrieves the `Class` object (more on this later) and gets its `Name` attribute.

This tutorial's "Writing good Java code" section includes a tip on how *not* to do logging.

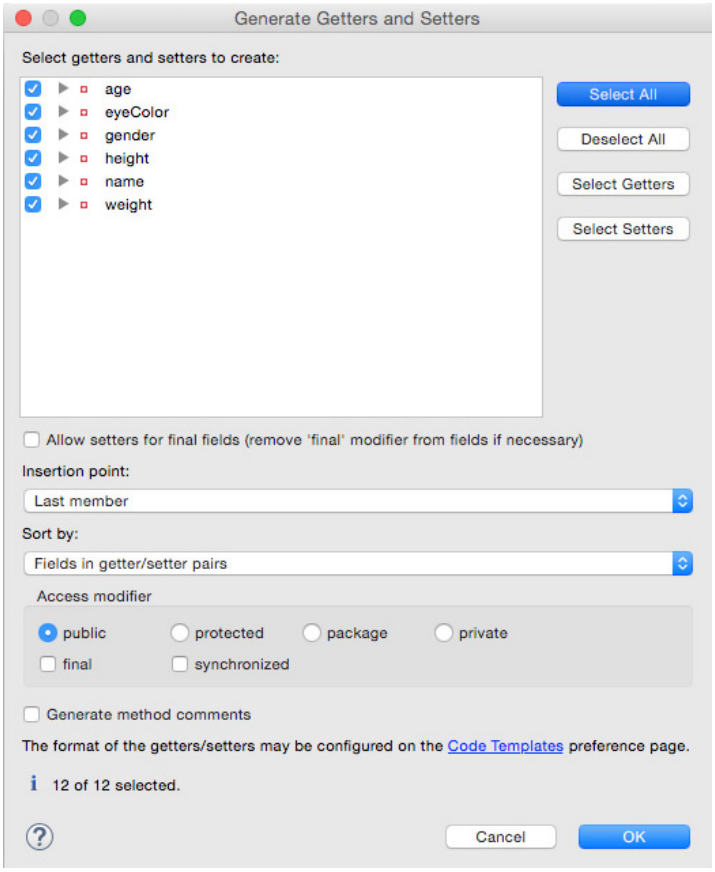
Before we get into the meat of testing, first go into the Eclipse source-code editor for `Person` and add this code just after `public class Person {` from Listing 3 so that it looks like this:

```
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;
}
```

Eclipse has a handy code generator to generate getters and setters (among other things). To try out the code generator, put your mouse caret on the `Person` class definition (that is, on the word `Person` in the class definition) and click **Source > Generate Getters and Setters...**. When the dialog box opens, click **Select All**, as shown in Figure 6.

Figure 6. Eclipse generating getters and setters



For the insertion point, choose **Last member** and click **OK**.

Now, add a constructor to `Person` by typing the code from Listing 5 into your source window just below the top part of the class definition (the line immediately beneath `public class Person ()`).

Listing 5. `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 is not 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

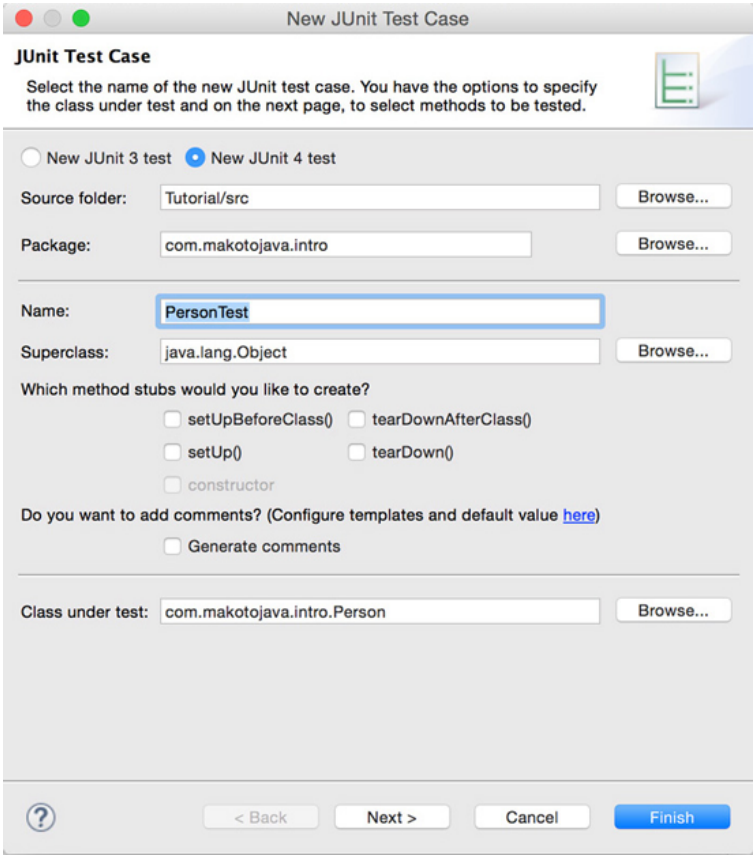
libraries such as JUnit, but using `main()` as your test harness can be a quick-and-dirty way to create a test harness.

Generate a JUnit test case

Now you generate a JUnit test case where you instantiate a `Person`, using the constructor in Listing 5, 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 are set to the correct attributes).

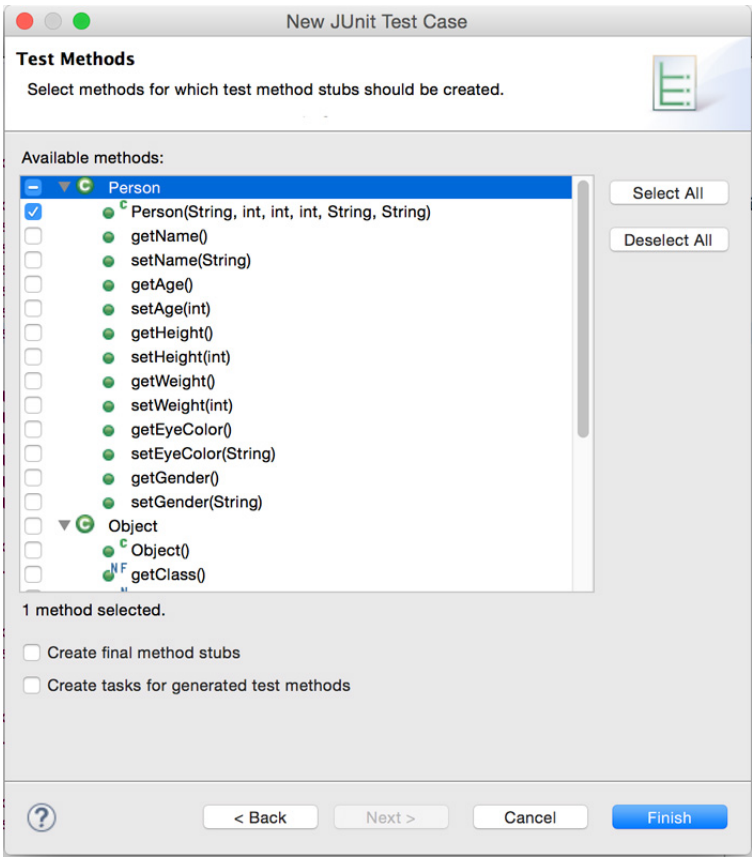
In the Package Explorer, right-click your `Person` class and then click **New > JUnit Test Case**. The first page of the New JUnit Test Case wizard opens, as shown in Figure 7.

Figure 7. Creating a JUnit test case



Accept the defaults by clicking **Next**. You see the Test Methods dialog box, shown in Figure 8.

Figure 8. Select methods for the wizard to generate test cases



In this dialog box, you select the method or methods that you want the wizard to build tests for. In this case, select just the constructor, as shown in Figure 8. Click **Finish**, and Eclipse generates the JUnit test case.

Next, open `PersonTest`, go into the `testPerson()` method, and make it look like Listing 6.

Listing 6. The `testPerson()` method

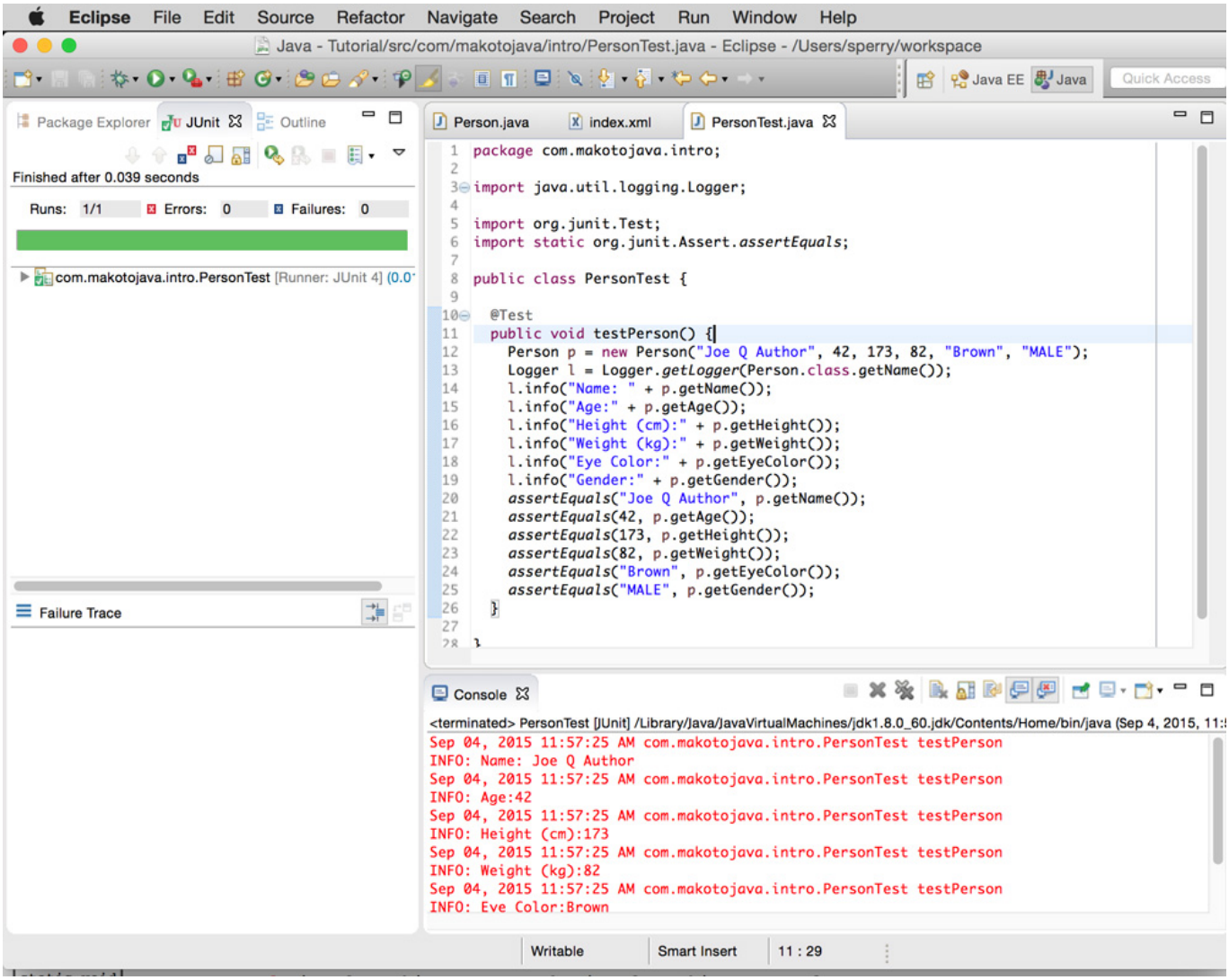
```
@Test
public void testPerson() {
  Person p = new Person("Joe Q Author", 42, 173, 82, "Brown", "MALE");
  Logger l = Logger.getLogger(Person.class.getName());
  l.info("Name: " + p.getName());
  l.info("Age:" + p.getAge());
  l.info("Height (cm):" + p.getHeight());
  l.info("Weight (kg):" + p.getWeight());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

Don't worry about the `Logger` class for now. Just enter the code as you see it in Listing 6. You're now ready to run your first Java program (and JUnit test case).

Running your unit test in Eclipse

In Eclipse, right-click PersonTest.java in the Package Explore and select **Run As** > **JUnit Test**. Figure 9 shows what happens.

Figure 9. See `Person` run



The Console view opens automatically to show `Logger` output, and the JUnit view indicates that the test ran without errors.

Adding behavior to a Java class

`Person` is looking good so far, but it can use some additional behavior to make it more interesting. Creating behavior means adding methods. This section looks more closely at *accessor methods*—namely, the getters and setters you've already seen in action.

Accessor methods

The getters and setters that you saw in action at the end of the preceding section 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 in Listing 6, 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.

Strings and operators

The tutorial has so far introduced several variables of type `String`, but without much explanation. You learn more about strings in this section, and also find out when and how to use 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*.

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 the "Your first Java class" section 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 2 summarizes the Java language's arithmetic operators:

Table 2. 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 2, 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 section.

Conditional operators and control statements

In this section, you learn about the various statements and operators you can use to tell your Java programs how you want them to act based on different input.

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 3 lists the relational and conditional operators of the Java language.

Table 3. 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 7.

Listing 7. 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 7 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.

Loops

In addition to being able to apply conditions to your programs and see different outcomes based on various `if`/`then` scenarios, you sometimes want your code to do the same thing over and over again until the job is done. In this section, learn about constructs used to iterate over code or execute it more than once.

What is a loop?

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 section.)

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 8 to execute three times, you can use a `for` loop.

Listing 8. 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 8. 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 section 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 8.

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 8. For comparison, Listing 9 uses a `while` loop to obtain the same result.

Listing 9. 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 10.

Listing 10. 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 11.

Listing 11. 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 11, 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 12.

Listing 12. 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 12, 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.

Java Collections

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 section introduces you to the most commonly used collection classes and gets you started with using them.

Arrays

Note: The square brackets in this section'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 Part 2.

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 section:

```
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 4.

Table 4. 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 in Part 2.) 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 tutorial, 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 section, 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 section, 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).

Archiving Java code

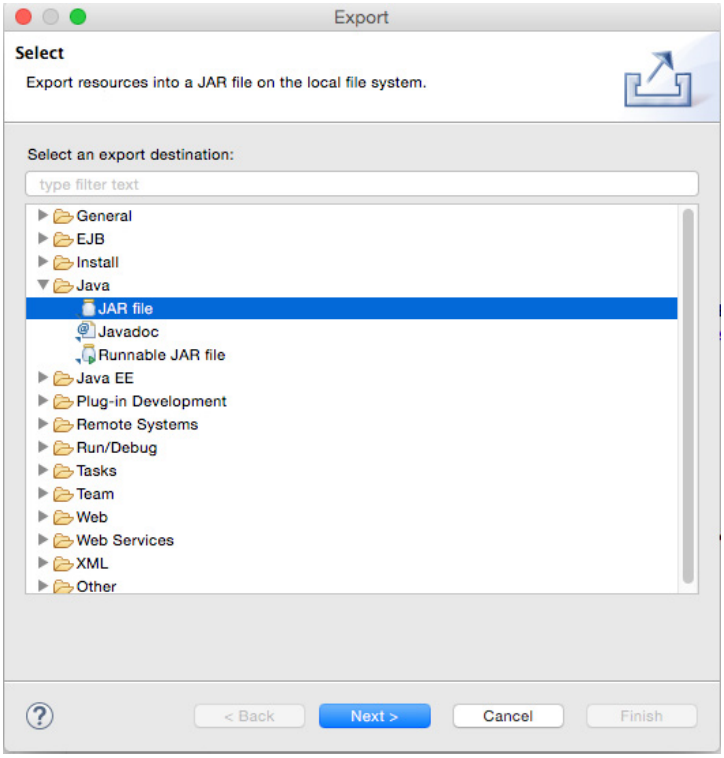
Now that you've learned a bit about writing Java applications, you might be wondering how to package them up so that other developers can use them, or how to import other developers' code into your applications. This section shows you how.

JARs

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. In your workspace, right-click the `com.makotojava.intro` package and click **File > Export**. You see the dialog box shown in Figure 10. Choose **Java > JAR file** and click **Next**.

Figure 10. Export dialog box



When the next dialog box opens, browse to the location where you want to store your JAR file and name the file whatever you like. The .jar extension is the default, which I recommend using. Click **Finish**.

You see your JAR file in the location you selected. You can use the classes in it from your code if you put the JAR in your build path in Eclipse. Doing that is easy, too, as you see next.

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.

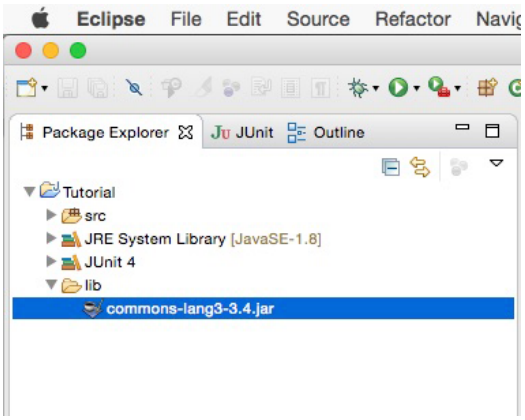
Let's assume you've already downloaded Commons Lang, which is stored in a JAR file. To use the classes, your first step is to create a lib directory in your project and drop the JAR file into it:

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.

Writing good Java code

You've got enough Java syntax under your belt to write basic Java programs, which means that the first half of this tutorial is about to conclude. This final section lays out a few best practices 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 the "Your first Java class" section) 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.

Conclusion to Part 1

In this tutorial, you learned about object-oriented programming, discovered Java syntax that you can use to create useful objects, and familiarized yourself with an IDE that helps you control your development environment. You know how to create and run Java objects that can do a good

number of things, including doing different things based on different input. You also know how to JAR up your applications for other developers to use in their programs, and you've got some basic best Java programming practices under your belt.

What's next

In the second half of this tutorial, you begin learning about some of the more advanced constructs of Java programming, although the overall discussion is still introductory in scope. Java programming topics covered in that tutorial include:

- Exception handling
- Inheritance and abstraction
- Interfaces
- Nested classes
- Regular expressions
- Generics
- Enum types
- I/O
- Serialization

Read "*Introduction to Java programming*, Part 2: Constructs for real-world applications."

Related topics

- developerWorks Java development
- IBM developer kits
- OO Design Process: The object primer
- 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
- 5 things you didn't know about ... JARs
- Speaking the Java language without an accent
- IBM Code: Java journeys