Introduction to Java programming

Skill Level: Introductory

Roy W. Miller (roy@roywmiller.com)
IBM

19 Nov 2004

The Java language, and the ever-growing Java platform, have revolutionized programming. The goal of this tutorial is to introduce you to the Java syntax you're most likely to encounter professionally, and to expose you to idioms that will help you avoid trouble. Follow along with Java professional Roy Miller as he guides you through the essentials of Java programming, including the OOP paradigm and how it applies to Java programming; Java language syntax and use; creating objects and adding behavior, working with collections, handling errors; and tips for writing better code.

Section 1. About this tutorial

What's this tutorial about?

This tutorial introduces you to object-oriented programming (OOP) with the Java language. The Java platform is a vast topic, so we won't cover it all here, but we'll cover enough to get you started. A follow-up tutorial will offer further information and guidance in your Java programming endeavors.

The Java language certainly has its friends and foes, but its impact on the software development industry is undeniable. On the plus side, the Java language gives programmers less rope to hang themselves with than C++. It removes some of the more onerous programming tasks, like explicit memory management, and allows programmers to focus on business logic. On the negative side, according to OO purists, the Java language has too many non-OO remnants to make it a good tool. Regardless of your position, however, knowing how to use the language as a tool when it's the right tool for the job is a wise career choice.

Should I take this tutorial?

The content of this tutorial is geared toward beginning Java programmers who might not be familiar with OOP concepts, or with the Java platform specifically. It assumes a general knowledge of downloading and installing software, and a general knowledge of programming and data structures (like arrays), but doesn't assume more than a cursory familiarity with OOP.

This tutorial will guide you through setting up the Java platform on your machine and installing and working with Eclipse, a free integrated development environment (IDE), to write Java code. From that point on, you'll learn the underpinings of Java programming, including the OOP paradigm and how it applies to Java programming; Java language syntax and use; creating objects and adding behavior, working with collections, handling errors; and tips for writing better code. By the end of the tutorial, you'll be a Java programmer -- a beginning Java programmer, but a Java programmer nonetheless.

Software requirements

To run the examples or sample code in this tutorial, you'll need to have the Java 2 Platform, Standard Edition (J2SE) , version 1.4.2 or higher, and the Eclipse IDE installed on your machine. Don't worry if you don't have these packages installed yet -- we'll show you how to do that in Getting started . All code examples in this tutorial have been tested with J2SE 1.4.2 running on Windows XP. One of the beautiful things about the Eclipse platform, however, is that it runs on almost any OS you're likely to use, including Windows 98/ME/2000/XP, Linux, Solaris, AIX, HP-UX, and even Mac OS X.

Section 2. Getting started

Installation instructions

In this next few sections, I'll walk you through each of the steps for downloading and installing the Java 2 Platform Standard Edition (J2SE), version 1.4.2, and the Eclipse IDE. The former lets you compile and run Java programs. The latter gives you a powerful and user-friendly way to write code in the Java language. If you already have the Java SDK and Eclipse installed, feel free to skip to A brief Eclipse tour or to the next section, OOP concepts , if you're comfortable jumping right in.

Install the Java SDK

The original intent of the Java language was to let programmers write a single program that would run on any platform, an idea encapsulated by the catch-phrase "Write Once, Run Anywhere" (WORA). In reality, it's not quite that simple, but it's

becoming easier. The various components of Java technology support that effort. The Java platform comes in three editions, Standard, Enterprise, and Mobile (the latter two for enterprise mobile device development). We'll be working with J2SE, which includes all of the core Java libraries. All you need to do is download it and install it.

To download the J2SE software development kit (SDK), follow these steps:

1. Open a browser and go to the Java Technology home page. In the top middle of the page, you'll see links for various Java technology subject areas. Select **J2SE (Core/Desktop)**.
2. In the list of current J2SE releases, click on **J2SE 1.4.2**.
3. In the left navigation bar of the resulting page, click on **Downloads**.
4. There are several downloads on this page. Find and click the **Download J2SE SDK** link.
5. Accept the conditions of the license and click **Continue**.
6. You'll see a list of downloads by platform. Chose the appropriate download for whatever platform you're using.
7. Save the file to your hard drive.
8. When the download is complete, run the install program to install the SDK on your hard drive, preferably in a well-named folder at the root of the drive.

That's it! You now have a Java environment on your machine. The next step is to install an integrated development environment (IDE).

Install Eclipse

An integrated development environment (IDE) hides lots of the mundane technical details of working with the Java language, so you can focus on writing and running code. The JDK you just installed includes several command-line tools that would let you compile and run Java programs without an IDE, but using those tools quickly becomes painful for all but the simplest programs. Using an IDE hides the details, gives you powerful tools to help you program faster and better, and is simply a more pleasant way to program.

It is no longer necessary to pay for an excellent IDE. The Eclipse IDE is an open source project and is yours to download for free. Eclipse stores and tracks your Java code in readable files stored on your file system. (You can use Eclipse to work with code in a CVS repository as well.) The good news is that Eclipse lets you deal with files if you want to, but hides the file details if you'd rather deal only with various Java constructs like classes (which we'll discuss in detail later).

Downloading and installing Eclipse is simple. Follow these steps:

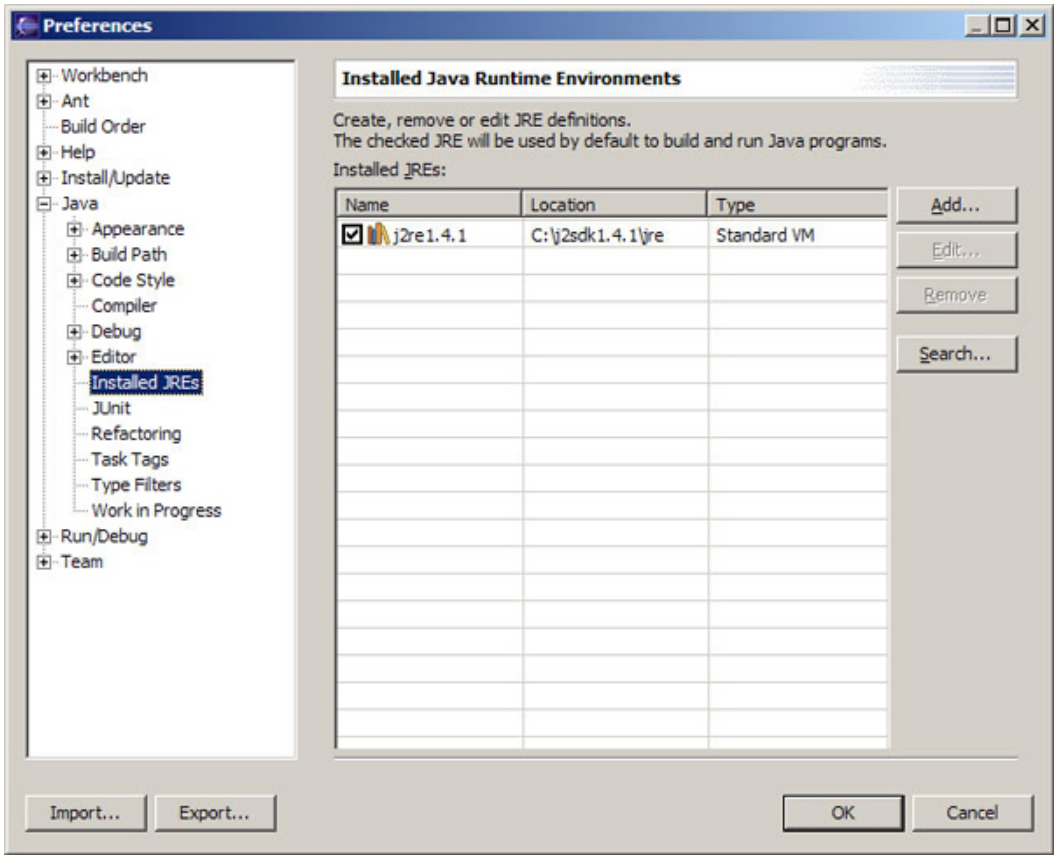
1. Open a browser and go to the Eclipse Web site.
2. Click the **Downloads** link on the left side of the page.
3. Click the **Main Eclipse Download Site** link to go to the Eclipse project downloads page.
4. You'll see a list of build types and names. Click the **3.0** link.
5. In the middle of the page, you'll see a list of Eclipse SDKs by platform; choose the one appropriate for your system.
6. Save the file to your hard drive.
7. When the download is complete, run the install program and install Eclipse on your hard drive, preferably in a well-named folder at the root of the drive.

All that's left now is to set up the IDE.

Set up Eclipse

To use Eclipse to write Java code, you must tell Eclipse where Java is located on your machine. Follow these steps:

1. Launch Eclipse by double-clicking on eclipse.exe, or the equivalent executable for your platform.
2. When the Welcome screen appears, click the **Go To The Workbench** link. This takes you to what's known as the Resource perspective (more on this later).
3. Click **Window>Preferences>Installed JREs,** which allows you to specify where your Java environment is installed on your machine (see Figure 1).
   Figure 1. Eclipse preferences



4. Odds are good that Eclipse will find an installed Java Runtime Environment (JRE), but you should explicitly point to the one you installed in Install the Java SDK. You can do that in the Preferences dialog. If Eclipse lists an existing JRE, click on it and press **Edit**; otherwise click **Add.**

5. Specify the path to the JRE folder of the JDK you installed in Install the Java SDK.

6. Click **OK**.

Eclipse is now set up to compile and run Java code. In the next section, we'll take a brief tour of the Eclipse environment to familiarize you with the tool.

A brief Eclipse tour

Working with Eclipse is a large topic and mostly out of the scope of this tutorial. See the Resources for links to more info on Eclipse. Here, we'll cover here just enough to get you familiar with how Eclipse works, and how you can use it for Java development.

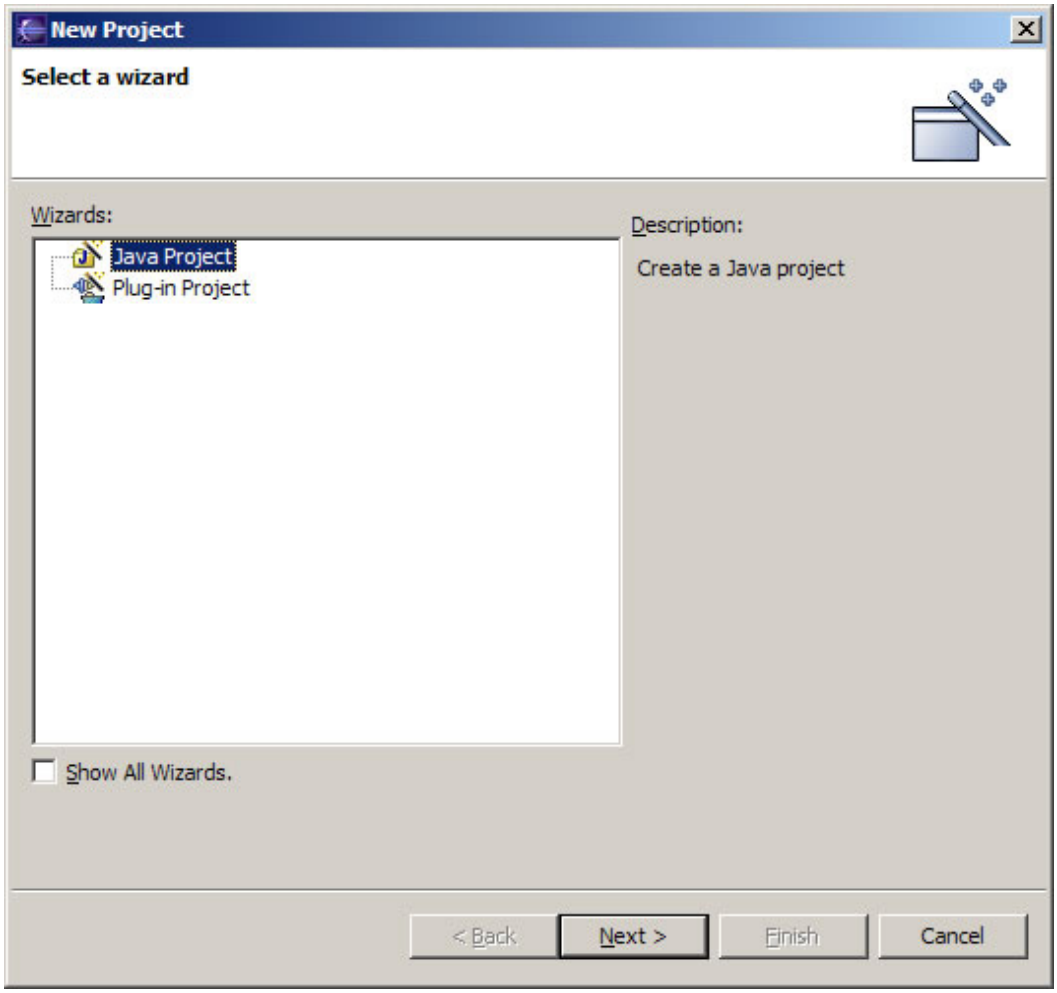
Assuming that you still have Eclipse up and running, you left off looking at the Resource perspective. Eclipse offers a set of *perspectives* on the code you write.

The Resource perspective shows a view of your file system within the Eclipse workspace you're using. A *workspace* holds all the files related to Eclipse development. Right now, there's nothing in your workspace that you really care about.

In general, Eclipse has *perspectives* that contain *views.* In the Resource perspective, you'll see a Navigator view, an Outline view, etc. You can drag and drop all of these views to position them wherever you want. It's an almost infinitely configurable environment. For now, though, the default arrangement is good enough. But what we're looking at won't let us do what we want to do. The first step to writing Java code in Eclipse is to create a Java project. This isn't a Java language construct; it's simply an Eclipse construct that helps you organize your Java code. Follow these steps to create a Java project:

1. Click **File>New>Project** to display the New Project wizard (see Figure 2). This is actually a "wizard wizard" -- in other words, it's a wizard that lets you choose which wizard to use (the New Project wizard, the New File wizard, and so on).

Figure 2. New project wizard



2. Make sure the Java Project wizard is selected and click **Next**.

3. Enter whatever project name you want ("Intro" might work well), leave all the defaults selected, and click **Finish**.

4. At this point, Eclipse should ask you if it should switch to the Java perspective. Click **No**.

You just created a Java project called Intro, which you should see in the Navigator view in the upper-left corner of the screen. We didn't switch to the Java perspective after creating the project because there's a better perspective to use for our current purposes. Click the **Open Perspective** button on the tab in the upper-right corner of the window, then select the Java Browsing perspective. This perspective shows you what you need to see to create Java programs easily. As we create Java code, we'll walk through a few more Eclipse features so you can learn how to create, modify, and manage your code. Before we do that, though, we have to cover some basic object-oriented programming concepts, which we'll do in the next section. Right now, we'll wrap up this section by taking a look at some online Java documentation.

The Java API online

The Java *application programming interface* (API) is vast, so learning how to find things is important. The Java platform is big enough to give you almost any tool you need as a programmer. Learning to exploit the possibilities can take just as much effort as learning the mechanics of the language.

If you go to Sun's Java documentation page (see Resources for the link), you'll see a link there for API documentation for each version of the SDK. Follow the one for version 1.4.2 to see what the documentation is like.

You'll see three frames in your browser:

- A list of built-in packages at the upper left
- A list of all classes at the lower left
- Details of what you selected on the right

Every class in the SDK is there. Select the class `HashMap`. One the right you'll see a description of the class. At the top you'll see the name and package it's in, its class hierarchy, its implemented *interfaces* (which are outside the scope of this tutorial), and any direct subclasses it may have. After that, you'll find a description of the class. Sometimes this description includes example usage, related links, style recommendations, etc. After the description, you'll see a list of constructors, then a list of all of the methods on the class, then all inherited methods, then detailed descriptions of all methods. It's very complete, and there's an exhaustive index at the top and bottom of the right-hand frame.

Many of the terms in the previous paragraph (like *package*) are new to you at this

point. Don't worry. We'll cover all of them in detail. For now, the important thing to know is that Java language documentation is available to you online.

Section 3. OOP concepts

What is an object?

Java is what's known as an *object-oriented* (OO) language, with which you can do object-oriented programming (OOP). This is very different from procedural programming, and can be a little strange for most non-OO programmers. The first step is to understand what an *object* is, because that's what OOP is based on.

An *object* is a self-contained bunch of code that knows about itself and can tell other objects about itself if they ask it questions that it understands. An object has *data members* (variables) and *methods,* which are the questions it knows how to answer (even though they may not be worded as questions). The set of methods that an object knows how to respond to is its *interface.* Some methods are open to the public, meaning that another object can call (or invoke) them. That set of methods is known as the object's *public interface.*

When one object invokes a method on an another object, that's known as *sending a message,* or a *message send.* That phrase is certainly OO terminology, but most often in the Java world people tend to say, "Call this method" rather than, "Send this message." In the next section, we'll look at a conceptual example that should make this more clear.

Conceptual object example

Suppose we have a Person object. Each Person has a name, an age, a race, and a gender. Each Person also knows how to speak and walk. One Person can ask another Person how old it is, or could tell another Person to start (or stop) walking. In programming terms, you would create a Person object and give it some variables (like name and age). If you created a second Person object, it could ask the first how old it is, or tell it to start walking. It would do those things by calling methods on the first Person object. When we start writing code in the Java language, you'll see how the language implements the concept of an object.

Generally, the concept of an object remains the same across the Java language and other OO languages, though it gets implemented differently from language to language. The concepts are universal. Because that's true, OO programmers, regardless of the language they're programming in, tend to speak differently from procedural programmers. Procedural programmers often talk about functions and modules. OO programmers talk about objects, and they often talk about those

objects using personal pronouns. It's not uncommon to hear one OO programmer say to another, "This Supervisor object here says to the Employee object, 'Give me your ID,' because he needs it to assign tasks to the Employee."

Procedural programmers might think this way of talking is strange, but it's perfectly natural for OO programmers. In their programming world, everything's an object (with some notable exceptions in the Java language), and programs are objects interacting (or "talking") with each other.

Fundamental OO principles

The concept of an object is critical for OOP, of course, as is the idea of objects communicating with messages. But there are three other foundational principles you need to understand.

You can remember the three fundamental OO principles with the acronym PIE:

- **P** olymorphism
- **I** nheritance
- **E** ncapsulation

Those are fancy names, but the concepts aren't really all that difficult to understand. In the next few sections, we'll talk about each in more detail, in reverse order.

Encapsulation

Remember, an object is a self-contained thing that contains data elements and actions it can perform on those data elements. This is an implementation of a principle known as *information hiding.* The idea is that an object knows about itself. If another object wants facts about the first object, it has to ask. In OOP terms, it has to send the object a message to ask for its age. In Java terms, it has to call a method on the object that will return the age.

Encapsulation ensures that each object is distinct, and that programs are conversations among objects. The Java language lets a programmer violate this principle, but it's almost always a bad idea to do so.

Inheritance

When you were born, biologically speaking, you were a combination of the DNA of your parents. You aren't exactly like either one of them, but you're similar to both. OO has the same principle for objects. Think about a Person object again. Recall that each person has a race. Not all People are the same race, but are they similar to one another nonetheless? Sure! They aren't Horses, or Chimps, or Whales. They're People. All People have certain things in common that make them distinct

from other kinds of animals. But they're also slightly different from each other. Is a Baby the same as an Adult? Nope. They move and speak and differently. But a Baby is certainly a Person.

In OO terms, Person and Baby are *classes* of things in the same *hierarchy,* and (most likely) Baby *inherits* characteristics and behavior from its *parent.* We could say that a particular Baby is a type of Person, or that Baby *inherits from* Person. It doesn't work the other way around -- a Person isn't necessarily a Baby. Each Baby object is an *instance* of the Baby class, and when we create a Baby object, we *instantiate* it. Think of a class as the template for instances of that class. Generally, what an object can do depends on what type of object it is -- or, in other words, what class it's an instance of. Both a Baby and, say, an Adult are types of Person, but one can have a job and one can't.

In Java terms, Person is a *superclass* of Baby and Adult, which are *subclasses* of Person. Another related concept is the idea of *abstraction.* A Person is at a higher level of abstraction than a Baby or an Adult. Both are types of Person, but they're slightly different. Still, all Person objects have some things in common (like a name and an age). Can you instantiate a Person? Not really. You either have a Baby or an Adult. Person is what, in Java terms, you would call a *abstract class.* You can't have an instance of Person directly. You'll have a Baby or an Adult, both of which are types of Person, but realistic ones. Abstract classes are beyond the scope of this tutorial, so this is the last we'll talk about them.

Now, think again about what it means for a Baby to "speak." We'll consider the implications in the next panel.

Polymorphism

Does a Baby "speak" like an Adult? Of course not. A Baby makes noise, but it's not necessarily recognizable words like an Adult uses. So, if I instantiate a Baby object (saying "instantiate a Baby" means the same thing -- the word "object" is assumed) and tell it to speak, it might coo or gurgle. One would hope that an Adult would be coherent.

In the humanity hierarchy, we have Person at the top, with Baby and Adult beneath it, as subclasses. All People can speak, so Baby and Adult can, too, but they do it differently. A Baby gurgles and makes simple sounds. An Adult says words. That's what *polymorphism* is: Objects doing things their own way.

How the Java language is (and is not) OO

As we'll see, the Java language lets you create first-class objects, but not everything in the language is an object. That's quite different from some OO languages like Smalltalk. Smalltalk is purely OO, meaning that everything in it is an object. The Java language is a mixture of objects and non-objects. It also lets one object know about the guts of another, if you as a programmer allow that to happen. That violates the principle of encapsulation.

However, the Java language also gives every OO programmer the tools necessary to follow all of the OO rules and produce very good OO code. But doing so requires discipline. The language doesn't force you to do the right thing.

While many object purists rightfully dispute whether the Java language is OO or not, this isn't really a productive argument to have. The Java platform is here to stay. Learn how to do OOP as well as possible with Java code, and leave the purity arguments to others. The Java language will let you write clear, relatively concise, and maintainable programs, which is good enough in my book for most professional situations.

Section 4. The Java language under the covers

How the Java platform works

When you write code in the Java language, like many other languages, you write *source code,* then you compile it; the compiler checks your code against the syntax rules of the language. The Java platform adds another step beyond that, though. When you compile Java code, you end up with *bytecodes.* The Java virtual machine (JVM) then interprets those bytecodes at runtime -- that is, when you tell Java to run a program.

In file terms, when you write code, you create a .java file. When you compile that file, the Java compiler creates a .class file, which contains bytecodes. The JVM reads and interprets that file at runtime, and how it does so is based the platform you're running on. To run on different platforms, you have to compile your source code against libraries specific to that platform. As you might imagine, the promise of Write Once, Run Anywhere ends up being Write Once, Test Anywhere. There are subtle (or not-so-subtle) platform differences that might make your code behave differently on different platforms.

Garbage collection

When you create Java objects, the JRE automatically allocates memory space for that object from the *heap,* which is the big pool of memory available on your machine. The runtime then keeps track of that object for you. When your program isn't using it anymore, the JRE gets rid of it. You don't have to worry about it.

If you've written any software in the C++ language, which is (arguably) OO as well, you know that as a programmer you have to allocate and deallocate memory for your objects explicitly by using functions called `malloc()` and `free()`. That is onerous for programmers. It's also dangerous, because it allows *memory leaks* to sneak into your programs. A memory leak is nothing more than your program

gobbling up memory at an alarming rate, which stresses the processor of the machine it's running on. The Java platform keeps you from having to worry about this at all, because it has what's known as *garbage collection.*

The Java garbage collector is a background process that gets rid of objects that aren't being used anymore, rather than forcing you to do that explicitly. Computers are good at keeping track of thousands of things, and at allocating resources. The Java platform lets the computer do that. It keeps a running count of references to every object in memory. When the count hits zero, the garbage collector reclaims the memory used by that object. You can manually invoke the garbage collector, but I've never had to do so in my career. It usually handles itself, and certainly will for every coding example in this tutorial.

IDEs versus command-line tools

As we noted before, the Java platform comes with command-line tools that let you compile (`javac`) and run (`java`) Java programs. So why use an IDE like Eclipse? Simply because using the command-line tools can be a pain in the neck for programs of any complexity. They're there for you if you need them, but using an IDE is most often the wiser choice.

The primary reason this is true is that an IDE manages files and paths for you, and has wizards that assist you when you want to change your runtime environment. When I want to compile a Java program with the `javac` command line tool, I have to worry about setting the `CLASSPATH` environment variable ahead of time so that the JRE can know where my classes are, or I have to set that variable at compile time. In an IDE like Eclipse, all I have to do is tell Eclipse where to find my JRE. If my code uses classes that I didn't write, all I have to do is tell Eclipse what libraries my project references and where to find them. That is far simpler than using the command line to type in heinously long statements that specify the classpath.

If you want or need to use the command-line tools, you can find out more about how to use them at Sun's Java Technology Web site (see Resources).

Section 5. OOP with Java technology

Introduction

Java technology covers a lot, but the language itself isn't huge. Covering it in English, however, is no small task. This section of the tutorial will not cover the language exhaustively. Instead, it will cover what you need to know to get started, and what you're most likely to encounter as a beginning programmer. Other tutorials (see Resources for pointers to some of them) cover various aspects of the language,

additional helpful libraries from Sun and other sources, and even IDEs.

We'll cover enough here with narrative and code examples that you should be able to start writing Java programs and learning about how to do OOP well in the Java environment. From there, it's just a matter of practice and learning.

Most introductory tutorials read like language specification reference books. First you see all of the syntax rules, then you see some usage examples, then you cover more advanced topics, like objects. We won't take that route here. The reason is that the primary cause of poor OO code written in the Java language is that beginning programmers aren't immersed in objects from the start. Objects tend to be treated as an add-on, or ancillary topic. Instead, we'll weave the Java syntactical learning throughout the Java OO learning. That way, you'll end up with a cohesive picture of how to use the language in an OO context.

Structure of a Java object

Remember, an object is an *encapsulated* thing that knows facts about itself and can do things when asked appropriately. Every language has rules about how to define an object. In the Java language, objects generally look like the following listing, although they might not have all of these parts:

```
package  packageName;


import  packageNameToImport;


 accessSpecifier class  ClassName {
         accessSpecifier
                     dataType
                     variableName [=  initialValue ];
        ...

         accessSpecifier ClassName(  arguments ) {
                 constructor statement(s)
        }

         accessSpecifier
                     returnValueDataType
                     methodName (  arguments ) {
                 statement(s)
        }
}
```

There are a few new concepts here, which we'll discuss in the next few panels.

Packages

The *package declaration* comes first when you define a class:

```
package  packageName;
```

Every Java object exists in a `package`. If you don't explicitly say which one it belongs to, the Java language puts it in the *default package.* A `package` is simply a

set of objects, all of which (typically) are related in some way. `package` s refer to a file path on your file system. Package names use dot notation to translate that file path into something the Java platform understands. Each piece of the `package` name is called a *node.*

For example, in the `package` named `java.util.ArrayList`, `java` is a node, `util` is a node, and `ArrayList` is a node. The last node refers the file `ArrayList.java`.

Import statements

The *import* statements come next when you define a class:

```
import  packageNameToImport;

...
```

When your object makes use of objects in other `package` s, the Java compiler needs to know where to find them. An *import* statement tells the compiler where to find the classes you use. For example, if I wanted to use the `ArrayList` class from the `java.util` package, I would import it this way:

```
import java.util.ArrayList;
```

Each `import` ends with a semicolon, as do most statements in the Java language. You can have as many `import` s as you need to tell Java where to find all the classes you use. For example, if I wanted to use the `ArrayList` class from the `java.util` package, and the `BigInteger` class from the `java.math` package, I would import them like this:

```
import java.util.ArrayList;
import java.math.BigInteger;
```

If you import more than one class from the same package, you can use a shortcut to say that you want to import all classes from that package. For example, if I wanted to use `ArrayList` and `HashMap`, both from the `java.util` package, I would import them like this:

```
import java.util.*;
```

You need an *import* statement for every unique `package` you import from.

Declaring a class

The *class declaration* comes next when you define a class:

```
                    accessSpecifier class  ClassName {
         accessSpecifier
                    dataType
                    variableName [=  initialValue ];
        ...

         accessSpecifier ClassName(  arguments ) {
                 constructor statement(s)
```

```
        }

         accessSpecifier
                    returnValueDataType
                    methodName (  arguments ) {
                statement(s)
        }
}
```

You define a Java object as a `class`. Think of a `class` as a template for an object, sort of like a cookie cutter. The `class` defines the *type* of object you can create with it. You can stamp out as many objects of that type as you want. When you do that, you've created an *instance* of the class -- or, to put it another way, you've *instantiated* an object. (**Note:** The word *object* is commonly used interchangeably to refer both to a class and to an instance of a class.)

The *access specifier* for the class can have several values, but most of the time it's `public`, and that's all we'll talk about in this tutorial. You can name a class whatever you'd like, but by convention class names start with a capital letter, and each subsequent word in the name starts with a capital letter also.

Classes have two types of *members: variables* (or *data members*) and *methods.* All of the members of a class are defined within the class's *body,* which exists between a single set of curly braces (or *curlies*) for the class.

Variables

The values of the variables of a class are what distinguish each instance of the class, which is why they're frequently called *instance variables.* A variable has an *access specifier,* a *data type,* a *name,* and (optionally) an *initial value.* Here is a list of the access specifiers and what they mean:

- `public`: Any object in any package can see the variable.
- `protected`: Any instance of the class, subclasses in the same package, and any non-subclass in the same package can see the variable. Subclasses in other packages can't see it.
- `private`: No object other than a specific instance of this class can see the variable, not even a subclass.
- No specifier (or *package protected*): Only classes within the same package as the class containing the variable can see it.

If you try to access a variable that is inaccessible to you, the compiler will tell you that the variable is not visible to you. The circumstances under which you should use each specifier is a judgment call, and we'll revisit it later.

Methods

The *methods* of a class define what it can do. There are two flavors of method in the

Java language:

- Constructors
- Other methods

Both have access specifiers (which dictate which other objects can use them) and bodies (between the curlies), and both contain one or more statements. Beyond that, their form and function are quite different. We'll cover each in turn in the next two panels.

Constructors

Constructors let you specify how to instantiate a class. You declare a constructor like this:

```
                    accessSpecifier ClassName(  arguments ) {
        constructor statement(s)
}
```

You get a *default constructor* (that takes no arguments) for free with every class you create. You don't even have to define it. Constructors look different from other methods in that they don't have a return value data type. That's because the return value data type is the class itself. You invoke a constructor in your code like this:

```
ClassName variableHoldingAnInstanceOfClassName = new ClassName(  arguments );
```

When you call a constructor, you use the `new` keyword. Constructors can take parameters or not (the default constructor doesn't). In the strict sense, constructors aren't methods or members of a class. They're a special animal in the Java language. In practice, though, they look and act like methods much of the time, and many people lump the two together. Just remember that they're special.

Non-constructor methods

Non-constructor methods in the Java language are what you use most often. You declare them like this:

```
                    accessSpecifier
                    returnValueDataType
                    methodName (  arguments ) {
        statement(s)
}
```

Every method has a return type, but not every method returns something. If the method returns nothing, you use the keyword `void` as the return type. You can name a method anything you want, as long as it's a valid identifier (it can't start with a period (`.`), for example), but by convention, method names:

- Are alphabetic

- Start with a lowercase letter
- Start subsequent words with uppercase letters

You invoke (or *call*) a method like this:

```
returnType variableForReturnValue = instanceOfSomeClass.methodName(parameter1, parameter2, ...);
```

Here, we're calling `methodName( )` on the `instanceOfSomeClass` object, and passing in some arguments. The distinction between *parameters* and *arguments* doesn't come up much, but they are different. A method takes parameters. When you pass *specific* values to a method as you call it, those values are arguments for the call.

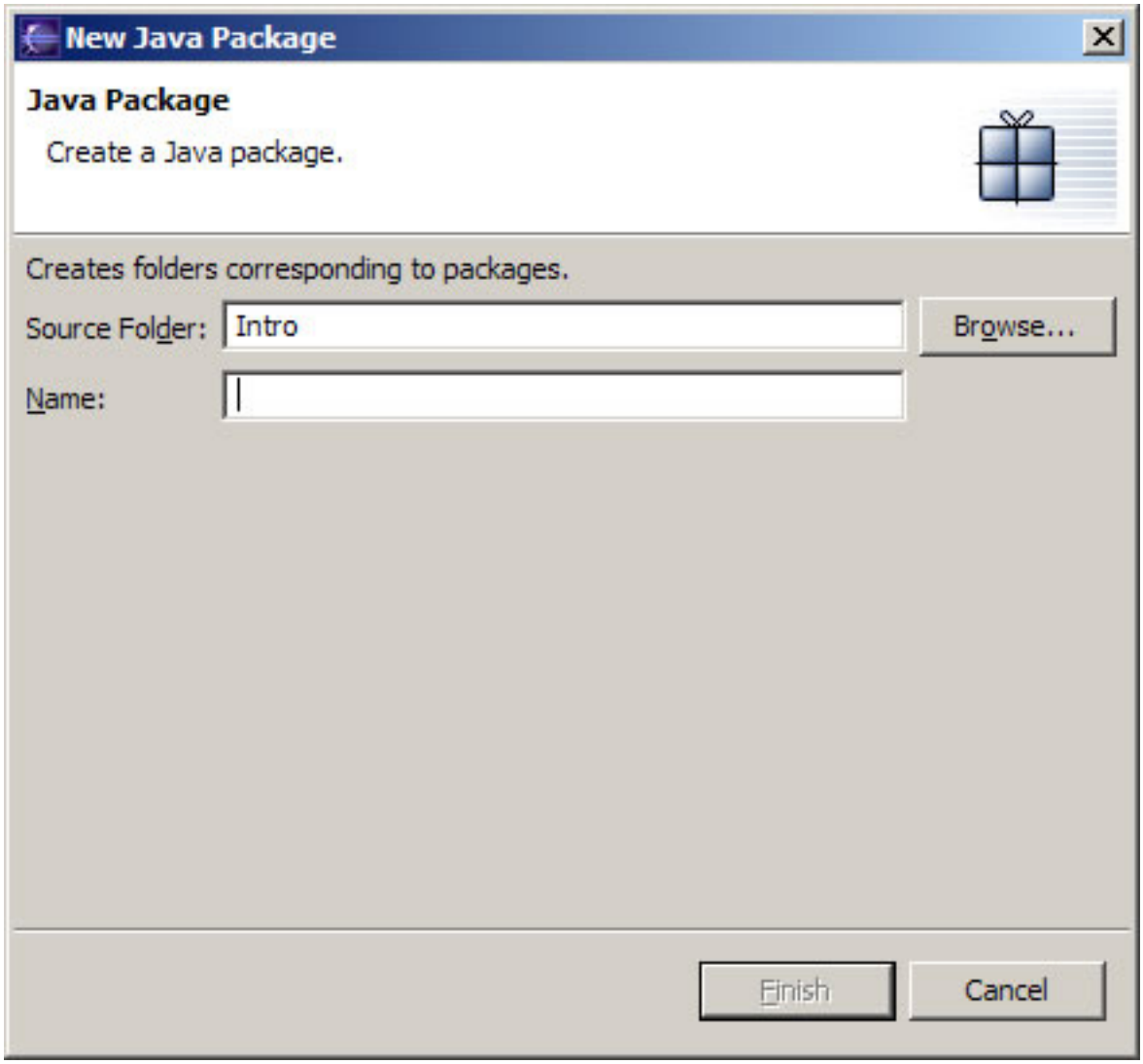
Section 6. Your first Java object

Creating a package

If you're not there already, get to the Java Browsing perspective in Eclipse. We'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.

Rather than using the default package, let's create one specifically for the Intro project. Click **File>New>Package.** That should bring up the Package wizard (see Figure 3).

Figure 3. Package wizard



Type **intro.core** as the package name and click **Finish**. You should see the following package in the Packages view in the workspace:

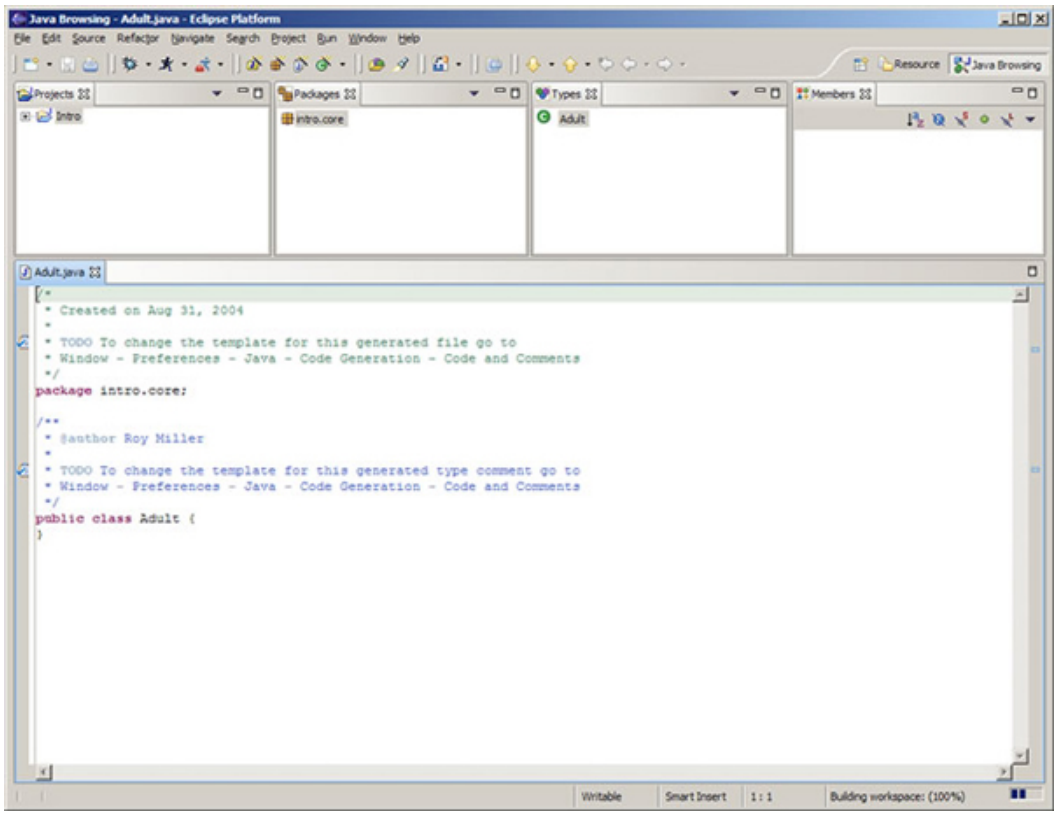
```
intro.core
```

Notice that the icon to the left of the package is ghosted -- that is, it looks like a grayed-out version of the package icon. That's a common Eclipse user interface convention for empty items. Your package doesn't have any Java classes in it yet, so its icon is ghosted.

Declaring the class

You can create a Java class in Eclipse with **File>New,** but we'll use the toolbar instead. Look above the Packages view to see the creation tools for projects, packages, and classes. Click the New Java Class tool (the green "C") to display the New Java Class wizard. Enter **Adult** as the class name, and accept all defaults by clicking **Finish**. Now you should see a few changes:

- The `Adult` class appears in the Classes view, to the right of the Packages view (see Figure 4).
  Figure 4. Workspace



- The `intro.core` package icon is no longer ghosted.
- An editor for `Adult.java` shows up below the views.

At this point, the class looks like this:

```
package intro.core;

public class Adult {
}
```

Eclipse generated a shell or template for the class for you, and included the `package` statement at the top. The class body is empty for now. We simply have to add things to flesh it out. You can configure templates for new classes, methods, and so on in the Preferences wizard you used before (**Window>Preferences**). You can configure code templates in the preferences path **Java>Code Style>Code Templates.** In fact, to simplify things for displaying code from this point on, I'm going to remove all comments from the templates -- meaning any lines preceded by `// comments` , or wrapped in `/* comments */`, or wrapped in `/** comments */`. From now on, you won't see any comments in code, unless we're specifically discussing their use, which we'll do in the next panel.

Before we move on, however, let's demonstrate a way that the Eclipse IDE makes your life easier. In the editor, change the word **class** to **clas** and wait a few seconds.

Notice that Eclipse underlines it with a wavy red line. If you mouse over the underlined item, Eclipse pops up an information window to tell you that you've got a syntax error. Eclipse helps you out by continuously compiling your code and alerting you unobtrusively if there's a problem. If you were using the command line tool `javac`, you'd have to compile the code and wait to see the errors. That can slow your development to a crawl. Eclipse removes that pain.

Comments

Like most every language, the Java language supports comments, which are simply statements that the compiler ignores when it checks syntax compliance. Java has several varieties of comments:

```
//  Single-line comment. The compiler ignores any text after the forward slashes.
/*  Multi-line comment. The compiler ignores any text between the asterisks. */
/**  javadoc comment. Compiler ignores text between asterisks, and the javadoc tool uses it. */
```

The last one is the most interesting. In a nutshell, the `javadoc` tool that comes with the Java SDK distribution you installed and can help you generate HTML documentation for your code. You can generate documentation for your own classes that looks very similar to what you see in the Java API documentation that we looked at in The Java API online. Once you've commented your code appropriately, you can run `javadoc` from the command line. You can find instructions for doing that, and all available information about `javadoc`, at the Java Technology Web site (see Resources).

Reserved words

There's one more item to cover before we start writing code that the compiler will check. Java has some words that are off-limits for you when naming variables. Here's the list:

`abstract`	`boolean`	`break`	`byte`
`case`	`catch`	`char`	`class`
`const`	`continue`	`char`	`class`
`default`	`do`	`double`	`else`
`extend`	`false`	`final`	`finally`
`float`	`for`	`goto`	`if`
`implements`	`import`	`int`	`instanceof`
`interface`	`long`	`int`	`native`
`new`	`null`	`package`	`private`
`protected`	`public`	`package`	`private`
`static`	`strictfp`	`super`	`switch`
`synchronized`	`short`	`super`	`this`
`throw`	`throws`	`true`	`try`

`transient`	`return`	`void`	`volatile`
`while`	`assert`	`true`	`false`
`null`			

It's not a very long list, but Eclipse makes reserved words bold as you type, so you don't have to remember them anyway. All but the last three in the list are Java *keywords.* The last three are *reserved words*. The difference doesn't matter for our purposes; you can't use either kind.

Now, on to some real code.

Adding variables

As I've said before, an `Adult` instance knows its name, age, race, and gender. We can add those pieces of data to our `Adult` class by declaring them as variables. Then every instance of the `Adult` class will have them. Most likely, each `Adult` will have different values for those variables. That's why each object's variables are often known as *instance variables* -- they're what distinguish each instance of a class. Let's add them, using `protected` as the access specifier for each:

```
package intro.core;

public class Adult {
        protected int age;
        protected String name;
        protected String race;
        protected String gender;
}
```

Now every `Adult` instance will contain those pieces of data. Notice here that each line of code ends with a semicolon. The Java language requires it. Notice also that each variable does indeed have a data type. We've got one integer and three string variables. Data types for variables can be of two flavors:

- Primitive data types
- Objects (either user-defined or internal to the Java language), also known as *reference variables*

Primitive data types

There are nine primitive data types you're likely to see on a regular basis:

Type	Size	Default value	Example
`boolean`	N/A	`false`	`true`
`byte`	8 bits	`0`	`2`
`char`	16 bits	`'u/0000'`	`'a'`
`short`	16 bits	`0`	`12`

`int`	32 bits	`0`	`123`
`long`	64 bits	`0`	`9999999`
`float`	32 bits with a decimal point	`0.0`	`123.45`
`double`	64 bits with a decimal point	`0.0`	`999999999.99999999`

We used an `int` for `age` because we don't need any decimal values, and an integer is big enough to hold any realistic human age. We used a `String` for the other three variables, because they aren't numeric. `String` is a class in the `java.lang` package, which you can access in your Java code automatically anytime you want (we'll talk about this more in Strings). You can also declare variables to be of user-defined types, such as `Adult`.

We defined each variable on a separate line, but we didn't have to. When you have two or more variables of the same type, you can define them on a single line, separated by commas, like this:

```
                    accessSpecifier dataType variableName1,
variableName2,
variableName3,...
```

If we wanted to initialize these variables when we declared them, we would just add initialization after each variable name:

```
                    accessSpecifier dataType variableName1 =  initialValue,
variableName2 =  initialValue, ...
```

Now our class knows about itself, and we can prove it, which we'll do next.

The main() method

There's a special method that you can include in any class so that the JRE can execute code. It's called `main()`. Each class can have only one `main()` method. Of course, not every class will have one, but because `Adult` is the only class we have at the moment, we'll add a `main()` method to it so that we can instantiate an `Adult` and inspect its instance variables:

```
package intro.core;

public class Adult {
        protected int age;
        protected String name;
        protected String race;
        protected String gender;

        public static void main(String[] args) {
                Adult myAdult = new Adult();

                System.out.println("Name: " + myAdult.name);
                System.out.println("Age: " + myAdult.age);
                System.out.println("Race: " + myAdult.race);
                System.out.println("Gender: " + myAdult.gender);
        }
```

```
}
```

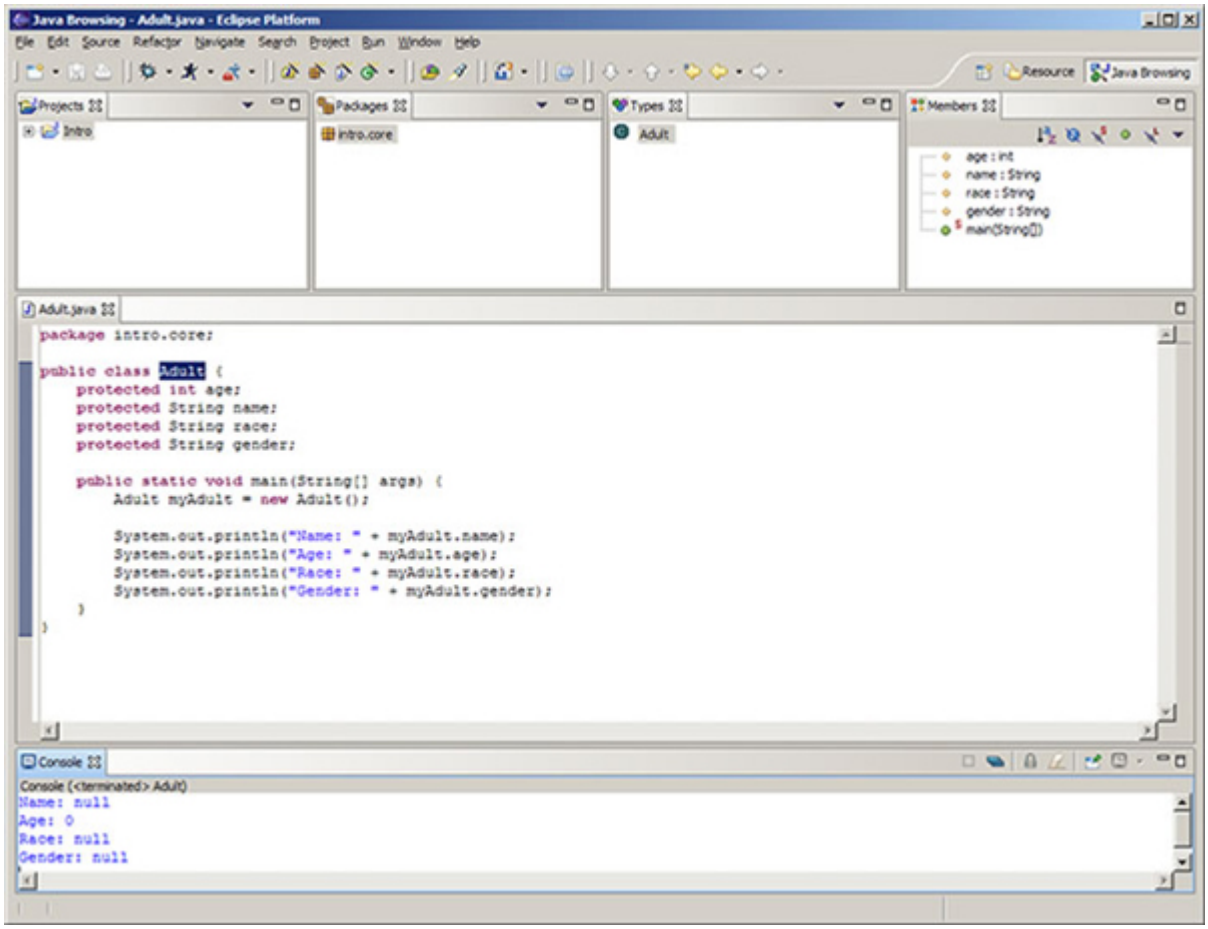
In the body of `main()`, we instantiate an `Adult`, then print out the values of its instance variables. Look at that first line. This is a case where OO purists get upset with the Java language. They say that `new` ought to be a method on `Adult`, and that you should call it thusly: `Adult.new()`. I certainly see their point, but the Java language doesn't work that way, and it's one reason purists can rightfully claim that it isn't purely OO. Look at that first line again. Remember that every Java class has a default constructor, which is what we're using here.

After we instantiate our `Adult`, we store it in a *local variable* called `myAdult`. Then we print out the values of its instance variables. In almost any language, you can print stuff out to the console. The Java language is no exception. The way you do it in Java code is to call the `println()` method on the `out` stream of the `System` object. Don't worry about understanding all of the details of the process for the moment. Just know that we're using a helpful method call to print things out. For each call, we pass a string literal and concatenate the value of an instance variable on `myAdult`. We'll revisit those method details later.

Executing code in Eclipse

To execute this code, you have a small amount of work to do in Eclipse. Click the `Adult` class in the Types view and click the "running man" icon on the toolbar. You should see the Run dialog, which allows you to create a launch configuration for your program. Select **Java Application** as the type of configuration you want to create, then click **New**. Eclipse will specify "Adult" as the default name for the configuration, which is fine. Click **Run** to see the results. Eclipse will display a Console view below your code editor, and it should look something like Figure 5.

Figure 5. Results of a run



Notice that the variables contained their default values. By default, any instance variable of a user-defined or built-in type contains `null`. It's almost always a good idea to initialize variables explicitly, though, especially objects, so you can be sure of the values that they have in them. Go back and initialize those values to:

s

Variable	Value
`name`	`"Bob"`
`age`	`25`
`race`	`"inuit"`
`gender`	`"male"`

Rerun the code by clicking on the running man icon again. You should see the new values on the console.

Now let's make our `Adult` capable of telling other objects about its data.

Section 7. Adding behavior

Accessors

Looking inside our `Adult` object by just referencing variables directly was convenient, but it's usually not a good idea for one object to dig into the internals of another like that. It violates that principle of encapsulation that we talked about before, and it allows one object to monkey around with another's internal state. A wiser choice is for one object to be able to tell other objects the values of its instance variables when asked. You use *accessors* (or *access methods*) to do that.

Accessors are methods like any other methods, but they typically adhere to a specific naming convention. To provide the value of an instance variable to another object, create a method named `getVariableName()`. Likewise, to allow other objects to set instance variables on your object, create a method named `setVariableName()`.

In the Java community, these access methods are commonly called *getters* and *setters*, as their names begin with `get` and `set`. These are pretty much the simplest methods you'll ever see, so they're good candidates to illustrate basic method concepts. You should know that *accessor* is a general term for a method that obtains information about an object. Not all accessors follow the naming convention for getters and setters, as we'll see later.

Here are some common characteristics of getters and setters:

- The access specifier for getters and setters is typically `public`.
- Getters typically don't take any parameters.
- Setters typically take parameters, and often take just one, which is a new value for the instance variable they set.
- The return type for a getter is typically the same as the return type of the instance variable it reports the value of.
- The return type for setters is typically `void`, meaning that they return nothing (they just set the value of an instance variable).

Declaring accessors

We could add accessors for the `age` instance variable on `Adult` that look like this:

```
public int getAge() {
        return age;
}
public void setAge(int anAge) {
        age = anAge;
}
```

The `getAge()` method responds with the value of the `age` variable by using the `return` keyword. Methods that don't return anything have an implicit `return void;` statement at the end. In that getter, we referred to `age` by using its name.

We also could've said `return this.age;`. The `this` variable refers to the current object. It's implicit when you refer to an instance variable directly. Some OO programmers from the Smalltalk world prefer to use `this` whenever they refer to an instance variable, just as they would always use the `self` keyword when coding in Smalltalk. I rather like that myself, but the Java language doesn't require it, and it does add extra stuff on the screen, so the examples in this tutorial won't use `this` unless the code would be unclear without it.

Calling methods

Now that we have accessors, we should replace the direct `age` access in our `main()` method with method calls. `main()` should now look like this:

```
public static void main(String[] args) {
        Adult myAdult = new Adult();
        System.out.println("Name: " + myAdult.name);
        System.out.println("Age: " + myAdult.getAge());
        System.out.println("Race: " + myAdult.race);
        System.out.println("Gender: " + myAdult.gender);
}
```

If you run this again, the results should be the same. Notice that calling a method on an object is easy. Use this form:

```
instanceName.methodName()
```

If the method takes no parameters (like our getter), you still have to include the parentheses after the name of the method when you call it. If the method takes parameters (like our setter), enclose them in the parentheses, separated by commas if there are more than one.

A word about the setter before we move on: It takes an `int` parameter named `anAge`. It then assigns that parameter's value to the instance variable `age`. We could've named the parameter anything we wanted. The name is unimportant, but when you refer to that parameter within the method, you have to use the name you gave it.

Before we go, let's try using the setter. Add this line to `main()` immediately after we instantiate an `Adult`:

```
myAdult.setAge(35);
```

Now rerun the code. Your results should show an age of 35. Here's what went on behind the scenes:

- We passed the *value* of an integer to the method as a parameter.

- The JRE allocated memory for the parameter and named it `anAge`.

Non-accessor methods

Accessors are helpful, but we want our `Adult` objects to be able to do things other than share their data, so we need to add other methods. We want our `Adult`s to speak, so let's start there. The `speak()` method could look like this:

```
public String speak() {
        return "hello";
}
```

By now, the syntax should be familiar. The method returns a string literal. Let's use it, and clean up `main()` while we're at it. Change the first call to `println()` to read:

```
System.out.println(myAdult.speak());
```

Rerun the code. You should see `hello` on the console.

Strings

We've used several variables of type `String` so far, but we haven't discussed them. Handling strings in C is labor-intensive, because they're null-terminated arrays of 8-bit characters that you have to manipulate. In the Java language, strings are first-class objects of type `String`, with methods that help you manipulate them. 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'`.

You've already seen how to instantiate a `String` object and set its value, but there are several other ways to do that. Here are a couple ways to create a `String` instance with a value of "hello":

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

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

You can do many things with `String`s, and the class has a large number of helpful methods. Without even using a method, we've already done something interesting with `String`s by *concatenating* a couple, which means combining them, one after the other:

```
System.out.println("Name: " + myAdult.getName());
```

Instead of using `+`, we could have called `concat()` on a `String` to join it with another:

```
System.out.println("Name: ".concat(myAdult.getName()));
```

This code might look a little strange, so let's walk through it briefly, left to right:

- `System` is a built-in object that lets you interact with various things in the system environment (including some capabilities of the Java platform itself).
- `out` is a *class variable* on `System`, meaning that it's accessible without having an instance of `System`. It represents the console.
- `println()` is a method on `out` that takes a `String` parameter, prints it to the console, and follows it with a newline character to start a new line.
- `"Name: "` is a string literal. The Java platform treats this literal as an instance of `String`, so we can call methods on it directly.
- `concat()` is an instance method on `String` that takes a `String` parameter and concatenates it to the `String` you called the method on.
- `myAdult` is our Adult instance.
- `getName()` is the accessor for the `name` instance variable.

So, the JRE gets the name of our `Adult`, calls `concat()` with it, and appends "Bob" to "Name: ".

In Eclipse, you can see the methods available on any object by placing your insertion point after the period following the variable containing an instance, then pressing Ctrl-Spacebar. That will pop up a list of methods on the object to the left of the period. You can scroll through the list (it wraps) with the arrow keys on your keyboard, highlight the one you want, and press Enter to select it. For example, to see all of the methods available on `String` objects, put your insertion point after the period following "Name: " and press Ctrl-Spacebar.

Using strings

For now, let's make use of concatenation in our `Adult` class. To this point, we've got a `name` instance variable. It would be nice to have a `firstname` and `lastname`, then concatenate them when somebody asks an `Adult` for its name. No problem! Add the following method:

```
public String getName() {
        return firstname + " " + lastname;
}
```

Eclipse should be showing squiggly red lines in the method because those instance variables don't exist yet, which means that the code won't compile. Now replace the existing `name` instance variable with these two (with defaults that make more sense):

```
protected String firstname = "firstname";
protected String lastname = "lastname";
```

Next, change the first `println()` call to look like this:

```
System.out.println("Name: " + myAdult.getName());
```

Now we have a fancier getter for our name variables. It concatenates them nicely to create a full name for our `Adult`. Alternatively, we could've made `getName()` look like this:

```
public String getName() {
        return firstname.concat(" ").concat(lastname);
}
```

This code does the same thing, but it illustrates the explicit use of a method on `String`, and it also illustrates chaining method calls. When we call `concat()` on `firstname` with a string literal (a space), it returns a new `String` that's a combination of the two. We then immediately call `concat()` on *that* one to join the first name and space with `lastname`. That gives us a nicely formatted full name.

Arithmetic and assignment operators

Our `Adult` can speak, but it can't move. Let's add some behavior for walking.

First, let's add an instance variable to keep track of the number of steps each `Adult` takes:

```
public int progress = 0;
```

Now let's add a method called `walk()`:

```
public String walk(int steps) {
        progress = progress + steps;
        return "Just took " + steps + " steps";
}
```

Our method takes an integer parameter for the number of steps to take, updates `progress` to reflect that number of steps, then reports some results. It also would be wise to add a getter for `progress`, but not a setter. Why? Well, allowing another object to teleport us forward to some total number of steps probably isn't smart. If another object wants to tell us to walk, it can call `walk()`. That's a judgment call, certainly, and this is a somewhat trivial example. On a real project, those sorts of design decisions have to be made all the time, and frequently can't be made in advance, no matter what serious OO design (OOD) gurus say.

In our method, we updated `progress` by adding `steps` to it. We stored the result in `progress` again. We used the most basic *assignment operator,* =, to store the result. We used the + *arithmetic operator* to add the two. There are other ways to accomplish the same goal. This code would do the same thing:

```
public String walk(int steps) {
        progress += steps;
        return "Just took " + steps + " steps";
}
```

Using the += assignment operator is a little less clumsy than the first approach we used. It keeps us from having to reference the `progress` variable twice. But it does the same thing: it adds `steps` to `progress` and stores the result in `progress`.

The table below is a list and brief description of the most commonly seen Java arithmetic and assignment operators (note that some of the arithmetic operators are *binary,* having two operands, and some are *unary,* having one operand).

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 and a
`-`	`-a`	Arithmetically negates a
`*`	`a * b`	Multiples a and b
`/`	`a / b`	Divides a by b
`%`	`a % b`	Returns the remainder of dividing a by b (in other words, this is 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 incrementing
`--`	`--a`	Decrements a by 1; computes the value of a after incrementing
`+=`	`a += b`	Same as `a = a + b`
`-=`	`a -= b`	Same as `a = a - b`
`*=`	`a *= b`	Same as `a = a * b`
`%=`	`a %= b`	Same as `a = a % b`

We've also already seen some other things that are called *operators* in the Java language. For instance, there's `.` (period), which qualifies names of packages and calls methods; `( params )`, which delimits a comma-separated list of parameters to a method; and `new`, which, when followed by a constructor name, instantiates an object. We'll see a few more in the next section.

Section 8. Conditional execution

Conditional execution introduction

Code that simply runs from the first statement to the last without changing direction really can't accomplish much. To be useful, your programs need to make decisions, and perhaps act differently in different situations. As with any useful language, the Java language gives you tools to do that, in the form of various statements and operators. This section will cover the bulk of those available to you when you write Java code.

Relational and conditional operators

The Java language gives you some operators and some flow control statements to let you make decisions in your code. Most often, a decision in code starts with a boolean expression (one that evaluates to true or false). Those expressions use *relational operators,* which compare one operand or expression to another, and *conditional operators.* Here's a list:

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 true, there's no need to evaluate b)
`||`	`a || b`	a or b is true, conditionally evaluates b (if a is true, there's no need to evaluate b)
`!`	`! 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 (true if a is true and b is false, or vice versa, but not if both are true or both are false)

Conditional execution with if

Now we have to use those operators. Let's add some simple logic to our `walk()` method:

```
public String walk(int steps) {
        if (steps > 100)
                return "I can't walk that far at once";

        progress = progress + steps;
        return "Just took " + steps + " steps.";
}
```

Now the logic in the method checks to see how big `steps` is. If it's too big, the method returns immediately and says it's too far. Each method can return only once. But aren't there two here? Yes, but only one will be executed. The Java `if` condition uses the following form:

```
if (  boolean expression ) {
         statements to execute if true...
} [else {
         statements to execute if false...
}]
```

The curlies aren't required if there's only a single statement after the `if` and/or the `else`, which is why our code didn't use them. You don't have to have an `else` clause, and we don't. We could have put the remaining code in the method in an `else`, but the effect would've been the same, and it would've added what's known as *unnecessary syntactical sugar,* which decreases readability of the code.

Variable scope

Every variable in a Java applications has *scope*, or characteristics that define where you can access that variable with just its name. If the variable is *in scope,* you can interact with it by name. If it's *out of scope*, you can't.

There are several scope levels in the Java language, defined by where a variable is declared (**Note:** None of these are official, as far as I know, but they're typical names programmers use):

```
public class SomeClass {

         member variable scope

        public void someMethod(  parameters ) {
                 method parameter scope

                local variable declaration(s)
                 local scope

                someStatementWithACodeBlock {
                        block scope
                }
        }
}
```

The scope of a variable extends to the end of the section (or block) of code in which it was declared. For example, in our `walk()` method, we referred to the `steps` parameter by its simple name, because it was in scope. Outside that method, referring to `steps` would produce a compile error. Code can also refer to variables declared in wider scope than that code. For example, we were allowed to refer to the `progress` instance variable inside the `walk()` method.

Other forms of if

We could make the conditional test fancier by using another form of the `if` statement:

```
if (  boolean expression ) {
         statements to execute if true...
} else if (  boolean expression ) {
         statements to execute if false...
} else if (  boolean expression ) {
         statements to execute if false...
} else {
         default statements to execute...
}
```

Maybe our method could look like this:

```
if (steps > 100)
        return "I can't walk that far at once";
else if (steps > 50)
        return "That's almost too far";
else {
        progress = progress + steps;
        return "Just took " + steps + " steps.";
}
```

There's also a shorthand version of `if` that looks a little odd, but accomplishes the same goal, although it doesn't allow multiple statements in either the `if` part or the `else` part. It uses the `?:` *ternary operator.* (A ternary operator is one that handles three operands.) We could rewrite our simple `if` condition this way:

```
return (steps > 100) ? "I can't walk that far at once" : "Just took " + steps + " steps.";
```

That wouldn't accomplish our goal, however, because when `steps` is less than 100, we want to return a message and we want to update `progress`. So in this case, using the shortcut `?:` operator wasn't an option because you couldn't execute multiple statements with it.

The switch statement

`if` is only one of the statements that lets you test conditions in your code. Another one you're likely to encounter is the `switch` statement. It evaluates an integer expression, then executes one or more statements based on the value of that expression. Its syntax is typically as follows:

```
switch ( integer expression ) {
        case 1:
                 statement(s)
                [break;]
        case 2:
                 statement(s)
                [break;]
        case n:
                 statement(s)
                [break;]
        [default:
                 statement(s)
                break;]
}
```

The JRE evaluates the *integer expression,* picks the case that applies, then executes the statements for that case. The final statement of each case except the last one is `break;`. That "breaks out" of the `switch` statement, and control continues to the next line of code after it. Technically, none of the `break;` statements are required. The last one is particularly unnecessary, because control would fall out of the statement anyway. But it's good practice to include them. If you don't include `break;` in each case, program execution will fall through to the next case, and so on, until it encounters a `break;` or reaches the end of the statement. The `default` case is what executes if the integer value doesn't trigger any of the cases. It's optional.

In essence, a `switch` statement really is an `if-else if` statement with an integer condition; if your condition is based on a single integer value, you could use either kind of statement. Could we rewrite our `if` condition in our `walk()` method as a `switch`? No, because we checked a boolean expression there (`steps > 100`). That's not allowed in a `switch`.

A switch example

Here's a trivial example of using `switch` (it's a rather classic example):

```
int month = 3;
switch (month) {
        case 1: System.out.println("January"); break;
        case 2: System.out.println("February"); break;
        case 3: System.out.println("March"); break;
        case 4: System.out.println("April"); break;
        case 5: System.out.println("May"); break;
        case 6: System.out.println("June"); break;
        case 7: System.out.println("July"); break;
        case 8: System.out.println("August"); break;
        case 9: System.out.println("September"); break;
        case 10: System.out.println("October"); break;
        case 11: System.out.println("November"); break;
        case 12: System.out.println("December"); break;
        default:  System.out.println("That's not a valid month number."); break;
}
```

`month` is an integer variable representing a month number. Because it's an integer, a `switch` is valid. For each valid case, we print the name of the month, then break out of the statement. The default case handles numbers outside the valid range for months.

Finally, here's an example of how having cases fall through can be a great little trick:

```
int month = 3;
switch (month) {
  case 2:
  case 3:
  case 9: System.out.println("My family has someone with a birthday in this month."); break;
  case 1:
  case 4:
  case 5:
  case 6:
  case 7:
  case 8:
  case 10:
  case 11:
  case 12: System.out.println("Nobody in my family has a birthday in this month."); break;
        default: System.out.println("That's not a valid month number."); break;
```

```
}
```

Here we see that cases 2, 3, and 9 get the same treatment, and the remaining cases get another one. Note that the cases don't have to be in order, and that having cases fall through was what we needed in this situation.

Play it again

At the beginning of this tutorial, we were doing everything unconditionally, which is fine for a while, but had its limits. Similarly, sometimes you want your code to do the same thing over and over again until the job's done. For example, suppose we wanted to have our `Adult` say more than just a single "hello." It's relatively easy to do that with Java code (although not as easy as it is with a scripting language like Groovy). The Java language gives you several ways to *iterate* over code, or execute it more than once:

- `for` statements
- `do` statements
- `while` statements

These are commonly known as *loops* (for example, a "`for` loop"), because they iterate over blocks of code until you tell them to stop. In the next few sections, we'll talk about each one briefly, and use them to make our `speak()` method a little more chatty.

for loops

The most basic looping construct in the Java language is the `for` statement, which lets you iterate over a range of values to determine how many times to execute the loop. Its most commonly used syntax looks like this:

```
for ( initialization; termination; increment ) {
        statement(s)
}
```

The *initialization* expression establishes where the loop starts. The *termination* expression establishes where it stops. The *increment* expression determines by how much the initialization variable is incremented each time through the loop. Each time through, the loop executes the statements in the block, which is the set of statements between the curlies (though remember that any code block in Java code is between curly braces, not just code in a `for` loop).

The syntax and capabilities of `for` are different in Java 5.0, so check out John Zukowski's column on new and interesting features in the recently released version of the language (see Resources for a link).

Using for loops

Let's change our `speak()` method to say "hello" three times, using a `for` loop. While we're at it, we'll learn about a built-in Java class that makes assembling strings a snap:

```
public String speak() {
        StringBuffer speech = new StringBuffer();
        for (int i = 0; i < 3; i++) {
                speech.append("hello");
        }
        return speech.toString();
}
```

The `StringBuffer` class in the `java.lang` package lets you manipulate strings easily, and is perfect for the job of appending strings together (which is the same thing as concatenating them). We simply instantiate one, then call `append()` each time we want to add something to the speech each `Adult` makes. The `for` loop is where the real business goes on. In the parentheses of the loop, we declared an integer variable `i` to server as our *loop counter* (the letters *i, j,* and *k* are very common as loop counters, but you can name the variable whatever you want). The next expression says we'll keep looping until that variable reaches a value less than three. The expression after that says that we'll increment our counter by one each time through the loop (remember the ++ operator?). Each time through the loop, we'll call `append()` on `speech` and stick another "hello" on the end.

Now, replace our old `speak()` method with this one, remove all of the `println` statements from `main()`, and then add one that calls `speak()` on `Adult`. When you're done, the class ought to look like this:

```
package intro.core;

public class Adult {
        protected int age = 25;
        protected String firstname = "firstname";
        protected String lastname = "lastname";
        protected String race = "inuit";
        protected String gender = "male";
        protected int progress = 0;

        public static void main(String[] args) {
                Adult myAdult = new Adult();
                System.out.println(myAdult.speak());
        }
        public int getAge() {
                return age;
        }
        public void setAge(int anAge) {
                age = anAge;
        }
        public String getName() {
                return firstname.concat(" ").concat(lastname);
        }
        public String speak() {
                StringBuffer speech = new StringBuffer();
                for (int i = 0; i < 3; i++) {
                        speech.append("hello");
                }
                return speech.toString();
        }
        public String walk(int steps) {
                if (steps > 100)
```

```
                        return "I can't walk that far at once";
                else if (steps > 50)
                        return "That's almost too far";
                else {
                        progress = progress + steps;
                        return "Just took " + steps + " steps.";
                }
        }
}
```

When you run it, you should get `hellohellohello` on the console. But using `for` is only one way to get this job done. The Java language gives you two other alternatives, which we'll look at next.

while loops

Let's try `while` first. The following version of `speak()` produces the same results as the version we saw on the previous panel:

```
public String speak() {
        StringBuffer speech = new StringBuffer();
        int i = 0;
        while (i < 3) {
                speech.append("hello");
                i++;
        }
        return speech.toString();
}
```

The basic syntax of `while` loops looks like this:

```
while ( boolean expression ) {
         statement(s)
}
```

A `while` loop executes code in its block until its expression returns `false`. How do you control the loop? You have to make sure the expression becomes false at some point; otherwise, you have an infinite loop. In our case, we declared a local variable called *i* outside the loop, initialized it to 0, then tested its value in our loop expression. Each time through the loop, we increment *i*. When it's no longer less than three, the loop will finish and we'll return the `String` stored in our buffer.

Here we see how `for` is rather convenient. In the `for` version, we declared and initialized our control variable, tested its value, and incremented it in one line of code. The `while` version required more housekeeping. If we forgot to increment our counter, we'd have an infinite loop. If we didn't initialize it, the compiler would complain. But the `while` version can be very useful if you have a complicated boolean expression to test (enclosing that in the `for` loop one-liner can make it tough to read).

Now we've seen `for` and `while` loops, but the next section will illustrate that there's a third way.

do loops

The following code will do exactly the same thing as the last two loops we've looked at:

```
public String speak() {
        StringBuffer speech = new StringBuffer();
        int i = 0;
        do {
                speech.append("hello");
                i++;
        } while (i < 3) ;
        return speech.toString();
}
```

The basic syntax of `do` loops looks like this:

```
do {
         statement(s)
} while ( boolean expression ) ;
```

A `do` loop is virtually the same as a `while` loop, except it tests its boolean expression *after* each execution of the loop block. With a `while` loop, what happens if the expression evaluates to `false` the first time it's tested? The loop won't execute even once. With a `do` loop, you're guaranteed that the loop with execute at least once. The distinction can come in handy at times.

Before we leave loops, let's cover two *branching statements* that can be helpful. We've already seen `break` when we talked about `switch` statements. It has a similar effect in loops: It stops the loop. The `continue` statement, on the other hand, stops the current iteration of the loop and moves on to the next one. Here's a trivial example:

```
for (int i = 0; i < 3; i++) {
        if (i < 2) {
                System.out.println("Haven't hit 2 yet...");
                continue;
        }

        if (i == 2) {
                System.out.println("Hit 2...");
                break;
        }
}
```

If you put that code in your `main()` method and run it, you'll get output like this:

```
Haven't hit 2 yet...
Haven't hit 2 yet...
Hit 2...
```

The first two times through the loop, `i` is less than 2, so we print "Haven't hit 2 yet...", then `continue`, which goes to the next iteration of the loop. When `i` is 2, the block of code for the first `if` doesn't execute. We fall through to the second `if`, print out "Hit 2...", then `break` out of the loop.

In the next section, we'll increase the richness of the behavior we can add by talking about how to handle *collections* of things.

Section 9. Collections

Collections introduction

Most real-world software applications deal with collections of things (files, variables, lines of files, etc.). Often, OO programs deal with collections of objects. The Java language has a sophisticated Collections Framework that allows you to create and manage collections of objects of various types. This framework could justify a tutorial all its own, so we can't cover it all here. Instead, we'll cover one very commonly used collection, and some techniques for using it. Those techniques apply to most collections available in the Java language.

Arrays

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 nothing more than a collection of *elements* of the same type.

There are two ways to declare an array:

- Create it with a certain size, which is fixed forever.
- Create it with a certain set of initial values. The size of this set determines the size of the array -- it will be exactly large enough to hold all of those values. Again, this size is fixed forever.

In general, you declare an array like this:

```
new  elementType [ arraySize ]
```

To create an integer array of five elements, you would do one of these two things:

```
int[] integers = new int[5];
int[] integers = new int[] { 1, 2, 3, 4, 5 };
```

The first statement creates an empty array of five elements. The second is a shortcut for initializing an array. It lets you specify a comma-separated list of initial values between curlies. Notice that we didn't include a size in the square brackets -- the number of items in our initialization block dictated a size of five elements. That's easier than creating the array and then coding a loop to put values in it, like so:

```
int[] integers = new int[5];
for (int i = 1; i <= integers.length; i++) {
        integers[i] = i;
        System.out.print(integers[i] + " ");
}
```

This code also declares the integer array of five elements. If you try to put more than

five elements in the array, you'll run into problem when you run the code. To load the array, we loop through the integers from 1 to the length of the array, which we discovered by accessing `length()` on our array. Each time through the loop, we put an integer into the array. When we hit 5, we stop.

Once our array is loaded, we can access the elements in the array with a similar loop:

```
for (int i = 0; i < integers.length; i++) {
        System.out.print(integers[i] + " ");
}
```

Think of an array as a series of buckets. Each element in the array sits in one of the buckets, which was assigned an *index* number when you created the array. You access the element at a particular bucket by writing:

```
arrayName [ elementIndex ]
```

The indices of an array are zero-based, meaning that the first one is at 0. So now our loop makes good sense. We start this loop with 0 because arrays are zero-based, and we loop through each element of the array, printing out the value at each index.

What is a collection?

Arrays are nice, but working with them is a little awkward. Loading them takes effort, and once you declare one, you can only load things of the array's type into it, and only as many items as the array can accommodate. Arrays certainly don't feel very OO. In fact, the primary reason the Java language has arrays at all is to serve as a holdover from pre-OO programming days. They're ubiquitous in software, so not having them in the language would make it tough to exist in the real world, especially when you have to interact with other systems that use them. But the Java language gives you many more tools for handling collections of things. Those tools are very OO.

The concept of a collection isn't that difficult to understand. When you need a fixed number of elements of the same type, you can use an array. When you need elements of various types, or a dynamically changing number of them, you use a Java Collection.

ArrayList

In this tutorial, we'll talk about only one type of collection, called an `ArrayList`. In the process, you'll learn another reason many OO purists have a bone to pick with the Java language.

In order to use an `ArrayList` in your code, you have to add an import statement for it in your class:

```
import java.util.ArrayList;
```

You declare an empty `ArrayList` like so:

```
ArrayList referenceVariableName = new ArrayList();
```

Adding and removing things from lists is straightforward. There are multiple methods to do it, but the two most common are as follows:

```
someArrayList.add(someObject);
Object removedObject = someArrayList.remove(someObject);
```

Boxing and unboxing primitives

Java Collections hold objects, not primitives. Arrays can hold both, but they aren't as OO as we'd like, most of the time. If you want to store anything that's a subtype of `Object` in the list, you simply call one of the various methods on `ArrayList` to get the job done. The simplest is:

```
referenceVariableName.add(someObject);
```

That appends the object you add to the end of the list. So far, so good. But what happens when you want to add a primitive to the list? You can't do that directly. Instead, you have to *wrap* your primitives in objects. There's one wrapper class per primitive type:

- `Boolean` for `boolean` s
- `Byte` for `byte` s
- `Character` for `char` s
- `Integer` for `int` s
- `Short` for `short` s
- `Long` for `long` s
- `Float` for `float` s
- `Double` for `double` s

For example, to put an `int` primitive into an `ArrayList`, we would have to use code like the following:

```
Integer boxedInt = new Integer(1);
someArrayList.add(boxedInt);
```

Wrapping a primitive in a wrapper instance is also known as *boxing* the primitive. To get the primitive back out, you have to *unbox* it. There are lots of helpful methods on the wrapper classes, but the fact that you have to have them at all really irks most programmers because it requires lots of extra work to use primitives with Collections. Java 5.0 reduces that pain by supporting *autoboxing/unboxing*.

Using collections

Most adults in the real world carry some money around with them. Let's assume every `Adult` has a wallet that holds his or her money. For this tutorial, we'll assume that:

- Money is represented by bills only
- The face value of the bill (as an integer) identifies each bill
- All money in the wallet is in U.S. dollars
- Each `Adult` starts its programmed "life" with no money

Remember our array of integers? Let's make an `ArrayList` instead. Add an import for `ArrayList`, then add `ArrayList` to your `Adult` class at the end of the list of other instance variables:

```
protected ArrayList wallet = new ArrayList();
```

We created the `ArrayList` and initialized it to an empty list, because our `Adult` has to earn every dollar. We can add some `wallet` accessors also:

```
public ArrayList getWallet() {
        return wallet;
}
public void setWallet(ArrayList aWallet) {
        wallet = aWallet;
}
```

Which accessors to provide is a judgment call, but in this case we went with the typical ones. There's no reason why we couldn't call `setWallet()` something like `resetWallet()`, or even `goBankrupt()`, because we are resetting it to an empty `ArrayList`. Should another object be able to reset our `wallet` with a new one? Again, a judgment call. That's what OOD is all about!

Now we're all set up to add some methods that let us interact with our `wallet`:

```
public void addMoney(int bill) {
        Integer boxedBill = new Integer(bill);
        wallet.add(boxedBill);
}
public void spendMoney(int bill) {
        Integer boxedBill = new Integer(bill);
        boolean haveThatBill = wallet.contains(boxedBill);

        if(haveThatBill) {
                wallet.remove(boxedBill);
        } else {
                System.out.println("I don't have that bill.");
        }
}
```

We'll look at these in more detail in the next few panels.

Interacting with collections

The `addMoney()` method lets us insert a bill into our wallet. Recall that our "bills" are simply integers. To add them to our collection, we wrap an `int` in an `Integer`.

The `spendMoney()` method does the boxing dance again to check for the bill in our `wallet` by calling `contains()`. If we have that bill, we call `remove()` to take it out. If we don't, we say so.

Let's use these methods in `main()`. Replace its current contents with the following:

```
public static void main(String[] args) {
        Adult myAdult = new Adult();

        myAdult.addMoney(5);
        myAdult.addMoney(1);
        myAdult.addMoney(10);

        StringBuffer bills = new StringBuffer();
        Iterator iterator = myAdult.getWallet().iterator();
        while (iterator.hasNext()) {
                Integer boxedInteger = (Integer) iterator.next();
                bills.append(boxedInteger);
        }
        System.out.println(bills.toString());
}
```

Our `main()` method combines a lot of things that we know about at this point. First, we call `addMoney()` a few times to put money in our `wallet`. Then we loop through the contents of our `wallet` to print out what's in there. We use a `while` loop to do that, but we have to do some extra work. We have to:

- Get an `Iterator` for the list, which lets us access elements in the list
- Call `hasNext()` on the `Iterator` as the boolean expression for our loop to see if we still have elements to handle
- Call `next()` on the `Iterator` to get the next element each time through the loop
- *Typecast* (or *cast*) the returned object to the type we know is in the list (an `Integer`, in this case)

That's the standard idiom for looping through a collection in the Java language. Alternatively, we could call `toArray()` on it and get an array back, which we could then loop through using `for` like we did a while back. The more OO way to do it is to exploit the power of the Java Collections framework.

The only new concept here is the idea of *casting*. What's that? As we already know, objects in the Java language have a type, or class. If you look at the signature of the `next()` method, you'll see that it returns an `Object`, not a particular subclass of `Object`. All objects in the world of Java programming are subclasses of `Object`, but the Java language needs to know what specific type an object is in order for you to be able to call methods specific to the type you want to deal with. If you don't cast, you're limited to the methods available on `Object`, which is a pretty small list. In this particular example, we didn't need to call any methods on the `Integer` s we got out of our list, but if we did, we'd have to cast first.

Section 10. Enhancing your object

Enhancing your object introduction

Right now, our `Adult` is reasonably useful, but not quite as useful as it could be. In this section, we'll enhance the object to make it easier to use and more helpful. This will involve:

- Creating some helpful constructors
- *Overloading* some methods to create a more convenient public interface
- Adding code to support comparing `Adult`s
- Adding code to make it easier to debug code that uses `Adult`s

Along the way, we'll learn about some *refactoring* techniques, and see how to handle things that might go wrong when we run our code.

Creating constructors

We've talked about constructors before. You might remember that every object in your Java code gets a default, no-argument constructor for free. You don't have to define it, and you won't see it in your code. In fact, we've been taking advantage of that in our `Adult` class. You don't see a constructor there.

In practice, however, it's wise to define your own constructors. When you do that, you can be absolutely sure that someone examining your class knows how to construct it in the way that you intended it to be constructed. So let's define our own no-argument constructor. Recall the basic structure of a constructor:

```
                    accessSpecifier ClassName(  arguments ) {
         constructor statement(s)
}
```

Defining a no-argument constructor for `Adult` is simple:

```
public Adult {
}
```

We're done. Our no-argument constructor does nothing, really, except create an `Adult`. Now when we call `new` to create an `Adult`, we'll be using our no-argument constructor instead of the default constructor. But what if we want the constructor to do something? In the case of `Adult`, it would be very convenient to be able to pass in a first name and a last name as `String`s, and have the constructor set our instance variables to those initial values. That's equally simple to do:

```
public Adult(String aFirstname, String aLastname) {
        firstname = aFirstname;
```

```
        lastname = aLastname;
}
```

This constructor takes two arguments and sets our instance variables from them. We now have two constructors. We really don't need the first constructor, but there's nothing wrong with keeping it. It gives users of this class options. They can create an `Adult` with a default name, or they can create one with a specific name that they provide.

What we just did, even though you probably didn't know it, was *overload* a method. We'll discuss this concept in more detail in the next panel.

Overloading methods

When you create two methods with the same name, but with a different number (or different types) of parameters, you have *overloaded* that method. This is one of the powerful things about objects. The Java language runtime will determine which version of the method to call, based on what you pass in. In the case of our constructors, if we don't pass in any arguments, the JRE will use the no-argument constructor. If we pass in two `String` s, the runtime will use the version that takes two `String` parameters. If we pass in arguments of a different type (or a single `String`), the runtime will complain that there's no constructor available that takes those types.

You can overload any method, not just constructors, which makes it easy to create a convenient interface for users of your classes. Let's try it by adding another version of our `addMoney()` method. At the moment, that method takes a single `int`. That's fine, but what if we want to add $100 to an `Adult` 's bankroll? We would have to call the method as many times as necessary to add the particular set of bills that total $100. That's very inconvenient. It would be much nicer to be able to pass in an array of `int` s representing a set of bills. So let's overload the method to accept an array parameter. Here's the method we have now:

```
public void addMoney(int bill) {
        Integer boxedBill = new Integer(bill);
        wallet.add(boxedBill);
}
```

Here's the overloaded version:

```
public void addMoney(int[] bills) {
        for (int i = 0; i < bills.length; i++) {
                int bill = bills[i];
                Integer boxedBill = new Integer(bill);
                wallet.add(boxedBill);
        }
}
```

This method looks very similar to our other `addMoney()` method, but this takes an array parameter. Let's try using it by changing our `main()` method on `Adult` to look like this:

```
public static void main(String[] args) {
        Adult myAdult = new Adult();

        myAdult.addMoney(new int[] { 1, 5, 10 });
        System.out.println(myAdult);
```

```
}
```

When we run the code, we can see that our `Adult` has a `wallet` with $16 in it now. That's a much nicer interface. But we're not done. Remember, we're professional programmers, and we want to keep our code clean. Do you see any code duplication in our two methods? The two lines in the first version show up verbatim in the second version. If we wanted to change what we do when we add money, we'd have to change code in two places, which is less than ideal. If we added another version of the method to take an `ArrayList` as parameter instead of an array, we'd have to change code in three places. That quickly becomes unacceptable. Instead, we can *refactor* the code to remove the duplication. On the next section, we'll do a *refactoring* called Extract Method to accomplish the job.

Refactoring while enhancing

Refactoring is the process of changing the structure of existing code without changing its function. Your application should produce the same output after the refactoring process, but your code should be cleaner, clearer, and less duplicative. It's convenient to refactor before adding a feature (to make it easier to add, or make it more obvious where to add it), and after adding a feature (to clean up any made make while adding it). In this case, we added a new method and we now see some duplicated code. Time to refactor!

First, we need to create a method that captures the two lines of duplicated code. We'll call it `addToWallet()`:

```
protected void addToWallet(int bill) {
        Integer boxedBill = new Integer(bill);
        wallet.add(boxedBill);
}
```

We made this method `protected` because it's really our own internal helper method, not part of the public interface of our class. Now let's replace those lines of code in our methods with a call to this new method:

```
public void addMoney(int bill) {
        addToWallet(bill);
}
```

Here's the overloaded version:

```
public void addMoney(int[] bills) {
        for (int i = 0; i < bills.length; i++) {
                int bill = bills[i];
                addToWallet(bill);
        }
}
```

If you rerun the code, you should see the same results. That kind of refactoring should get to be a habit, and Eclipse makes it easier for you by including many automated refactorings. Going into detail about them is beyond the scope of this tutorial, but you can experiment with them. If we had selected those two lines of duplicated code in, say, the first version of `addMoney()`, we could have right-clicked on the selected code and selected **Refactor>Extract Method.** Then Eclipse would

have walked us through the refactoring. This is one of the most powerful features of the IDE.

Class members

The variables and methods we have on `Adult` are instance variables and methods. Recall that every instance will have those.

Classes themselves can also have variables and methods. Those are collectively called *class members,* and you declare them with the `static` keyword. The differences between class members and the instance varieties are:

- Every instance of a class shares a single copy of a class variable
- You can call class methods on the class itself, without having an instance
- Instance methods can access class variables, but class methods cannot access instance variables
- Class methods can access only class variables

When does it make sense to add class variables or methods? The best rule of thumb is to do so rarely, so that you don't overuse them. Some typical uses are:

- To declare constants that any instance of the class can use
- To track "counters" of instances of the class
- On a class with utility methods that don't ever need an instance to be helpful (such as the `Collections.sort()` method)

Class variables

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

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

The JRE creates a copy of a class's instance variables for every instance of that class. It creates only a single copy of each class variable, regardless of the number of instances, the first time it encounters the class in a program. All instances will share (and potentially modify) that single copy. That makes class variables a good choice for *constants* that all instances should be able to use.

For example, we've been using integers to describe "bills" in the `wallet` of `Adult`. That's perfectly acceptable, but it might be nice to name the integer values so that we can easily see what the numbers represent when we read the code. Let's declare a few constants to do that, in the same place we declare the instance variables for our class:

```
protected static final int ONE_DOLLAR_BILL = 1;
protected static final int FIVE_DOLLAR_BILL = 5;
protected static final int TEN_DOLLAR_BILL = 10;
protected static final int TWENTY_DOLLAR_BILL = 20;
protected static final int FIFTY_DOLLAR_BILL = 30;
protected static final int ONE_HUNDRED_DOLLAR_BILL = 40;
```

By convention, class constants have names in all uppercase letters, with words separated by underscores. We used `static` to declare them as class variables, and we added the `final` keyword to make sure that they can't be changed by any instance (that is, to make them constants). Now we can change `main()` to add some money to our `Adult` using these new named constants:

```
public static void main(String[] args) {
        Adult myAdult = new Adult();
        myAdult.addMoney(new int[] { Adult.ONE_DOLLAR_BILL, Adult.FIVE_DOLLAR_BILL });
        System.out.println(myAdult);
}
```

Reading this code makes it obvious what we're adding to our `wallet`.

Class methods

As we've already seen, we call an instance method like this:

```
variableWithInstance.methodName();
```

We called the method on a named variable that holds an instance of a class. When you call a class method, you call it like this:

```
ClassName.methodName();
```

We didn't need an instance to call this method. We called it on the class itself. The `main()` method we've been using is a class method. Look at its signature. Notice that it is declared as `public static`. We've seen that access specifier before. The `static` keyword identifies this as a class method, which is the reason those methods are sometimes called *static methods.* We didn't need to have an instance of `Adult` to call `main()`.

We can create class methods for `Adult` if we want to, although there's really no reason to do so in this case. To illustrate how, though, let's add a trivial class method:

```
public static void doSomething() {
        System.out.println("Did something");
}
```

Comment out the current lines of code in `main()` and add this line:

```
Adult.doSomething();
Adult myAdult = new Adult();
myAdult.doSomething();
```

When you run that code, you should see the appropriate message in the console twice. The first call to `doSomething()` is the typical way to call a class method. You can also call them through an instance of a class, as in the third line of code. But

that's not really good form. Eclipse tells you that by underlining that line with a wavy yellow line and suggesting that you should access that method in a "static way," meaning on the class, not an instance.

Comparing objects with ==

There are two ways to compare objects in the Java language:

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

The first, and most basic, compares objects for *object equality.* In other words, this statement:

```
a == b
```

will return `true` if and only if `a` and `b` refer to exactly the same instance of a class (that is, the same object). The exception is for primitives. When you compare two primitives with ==, the Java language runtime compares their values (remember, they aren't true objects anyway). Try this experiment in `main()`, and view the results in the console:

```
int int1 = 1;
int int2 = 1;
Integer integer1 = new Integer(1);
Integer integer2 = new Integer(1);
Adult adult1 = new Adult();
Adult adult2 = new Adult();

System.out.println(int1 == int2);
System.out.println(integer1 == integer2);
integer2 = integer1;
System.out.println(integer1 == integer2);
System.out.println(adult1 == adult2);
```

The first comparison returns `true`, because we're comparing primitives with the same value. The second returns `false`, because the two variables don't refer to the same object instance. The third returns `true`, because the two variables now refer to the same instance. Trying it with our class, we also get `false` because `adult1` and `adult2` don't refer to same instance.

Comparing objects with equals()

You call `equals()` on an object like this:

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

The `equals()` method is on type `Object`, which is the parent of every class in the Java language. That means that any class you create will inherit the basic `equals()` behavior from `Object`. That basic behavior is no different from the == operator. In other words, by default, these two statements both use == and return `false`:

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

Look at the `spendMoney()` method on `Adult` again. What goes on behind the scenes when we call `contains()` on our `wallet`? The Java language uses the == operator to compare the objects in the list to the one we asked for. If it finds a match, the method returns `true`; otherwise it returns `false`. Because we're comparing primitives, it can find a match based on the values of integers (remember that == compares primitives based on their value).

That's great for primitives, but what if want to compare the contents of objects? The == operator won't do it. To compare the contents of objects, we have to *override* the `equals()` method on the class a is an instance of. That means that you create a method with exactly the same signature as the method in one of your superclasses, but you implement the method differently. If you do that, you can compare the contents of two objects to see if they're equal, rather than just checking to see if two variables refer to the same instance.

Try this experiment in `main()`, and view the results in the console:

```
Adult adult1 = new Adult();
Adult adult2 = new Adult();

System.out.println(adult1 == adult2);
System.out.println(adult1.equals(adult2));

Integer integer1 = new Integer(1);
Integer integer2 = new Integer(1);

System.out.println(integer1 == integer2);
System.out.println(integer1.equals(integer2));
```

The first comparison returns `false`, because `adult1` and `adult2` refer to different instances of `Adult`. The second returns `false` as well, because the default implementation of `equals()` simply compares the two variables to see if they refer to the same instance. But that default behavior of `equals()` isn't usually what we want. We'd like to compare the *contents* of two `Adult` s to see if they look the same. We can override `equals()` to do that. As you see from the final two comparisons in the example above, the `Integer` class overrides the method such that == returns `false`, but `equals()` compares the wrapped `int` values for equality. We'll do something similar for `Adult` in the next panel.

Overriding equals()

Overriding `equals()` to compare objects actually requires us to override two methods:

```
public boolean equals(Object other) {
        if (this == other)
                return true;

        if ( !(other instanceof Adult) )
                return false;

        Adult otherAdult = (Adult)other;
        if (this.getAge() == otherAdult.getAge() &&
                this.getName().equals(otherAdult.getName()) &&
                this.getRace().equals(otherAdult.getRace()) &&
```

```
                this.getGender().equals(otherAdult.getGender()) &&
                this.getProgress() == otherAdult.getProgress() &&
                this.getWallet().equals(otherAdult.getWallet()))
                return true;
        else
                return false;
}

public int hashCode() {
        return firstname.hashCode() + lastname.hashCode();
}
```

We override `equals()` in the following way, which is a typical Java idiom:

- If the object we're going to compare is the same object as this one, the two obviously are equal, so that we return `true`
- We check to be sure the object we're going to compare is an instance of `Adult` (if it's not, the two objects obviously aren't the same)
- We cast the incoming object to an `Adult` so we can call familiar methods on it
- We compare the pieces of the two `Adult`s that should be the same if the two objects are "equal" (for whatever definition of equal we're using)
- If any of those things aren't equal, we return `false`; otherwise we return `true`

Note that we can compare the `age` of each with ==, because it's a primitive value. We use `equals()` to compare `String`s, because that class overrides `equals()` to compare the contents of the `String`s (if we used ==, we'd get `false` every time, because two `String`s aren't ever the same object). We did the same for `ArrayList`, because it overrides `equals()` to check that two lists have the same elements in the same order, which is good enough for our simple example.

Whenever you override `equals()`, you should override `hashCode()` also. The reasons why are beyond the scope of this tutorial, but for now, just know that the Java language uses the value returned by that method to put instances of your class in collections that use a hashing algorithm to space objects (such as `HashMap`). The only hard and fast rules for what the method returns (other than that it must return an integer) are that `hashCode()` must return:

- The same value for the same object every time
- Equal values for equal objects

Generally, returning the hashcode values for some or all of an object's instance variables is a good enough way to compute a hashcode. Another option is to convert the variables to `String`s, combine them, then return the hashcode for the resulting `String`. Yet another option is to multiply one or more of the numeric variables by some constant to go for further uniqueness, but that's often overkill.

Overriding toString()

The `Object` class has a `toString()` method, which every class you create will inherit. It returns a `String` representation of your object, and is very helpful for debugging. To see what the default implementation of `toString()` does, try this experiment in `main()`:

```
public static void main(String[] args) {
        Adult myAdult = new Adult();

        myAdult.addMoney(1);
        myAdult.addmoney(5);

        System.out.println(myAdult);
}
```

The results we'll get in the console look like this:

```
intro.core.Adult@b108475c
```

The `println()` method calls `toString()` on objects passed to it. Because we haven't overridden `toString()` yet, we get the default output, which is an object ID. Every object has one, but it doesn't tell you very much about the object. It would be better if we overrode `toString()` to give us a nicely formatted picture of the contents of our `Adult`:

```
public String toString() {
        StringBuffer buffer = new StringBuffer();

        buffer.append("And Adult with: " + "\n");
        buffer.append("Age: " + age + "\n");
        buffer.append("Name: " + getName() + "\n");
        buffer.append("Race: " + getRace() + "\n");
        buffer.append("Gender: " + getGender() + "\n");
        buffer.append("Progress: " + getProgress() + "\n");
        buffer.append("Wallet: " + getWallet());

        return buffer.toString();
}
```

We create a `StringBuffer` to build a `String` representation of our object, then return the `String`. When you rerun, the console should show some nice output like this:

```
An Adult with:
Age: 25
Name: firstname lastname
Race: inuit
Gender: male
Progress: 0
Wallet: [1, 5]
```

That's much more convenient and helpful than a cryptic object ID.

Exceptions

It would be nice if nothing ever went wrong in our code, but that's unlikely. Sometimes things don't go as we'd like, and sometimes the problem is worse than just producing unwanted results. When that happens, the JRE *throws an exception.* The language includes some special statements that let you *catch* the exception and handle it appropriately. Here is the general format for those statements:

```
try {
        statement(s)
} catch ( exceptionType
                    name ) {
        statement(s)
} finally {
        statement(s)
}
```

The `try` statement wraps code that might throw an exception. If it does, execution drops immediately to the `catch` block, also known as an *exception handler.* When all the trying and catching is done, execution continues to the `finally` block, regardless of whether or not there was an exception thrown. When you catch an exception, you can attempt to recover from it, or you can exit the program (or method) gracefully.

Handling exceptions

Try (no pun intended) this experiment in `main()`:

```
public static void main(String[] args) {
        Adult myAdult = new Adult();

        myAdult.addMoney(1);
        String wontWork = (String) myAdult.getWallet().get(0);
}
```

When we run this, we'll get an exception. The console will show something like this:

```
java.lang.ClassCastException
        at intro.core.Adult.main(Adult.java:19)
Exception in thread "main"
```

This *stack trace* reports the type of exception and the line number where it occurred. Remember that we have to cast when we remove an `Object` from a Collection. We have a collection of `Integer`s, but we tried to get the first one with `get(0)` (where 0 refers to the index of the first element in the list, because the list is zero-based, just like an array) and cast it to type `String`. The Java language runtime complains. At the moment, the program just dies. Let's make that it shut down more gracefully by handling the exception:

```
try {
        String wontWork = (String) myAdult.getWallet().get(0);
} catch (ClassCastException e) {
        System.out.println("You can't cast that way.");
}
```

Here we catch the exception and print a nice message. Alternatively, we could have done nothing in the `catch` block, and printed our nice message in the `finally` block, but that wasn't necessary. In some cases, the exception object (commonly, but not necessarily, named `e` or `ex`) can give you more information about the error, which can help you report better information or recover gracefully.

The exception hierarchy

The Java language incorporates an entire exception hierarchy, which means that

there are many types of exceptions. At the highest level, some exceptions are *checked* by the compiler, and some, called `RuntimeException` s, aren't. The language rule is that you must *catch or specify* your exceptions. If a method can throw a non- `RuntimeException`, the method either has to handle it, or has to specify that a calling method must. You do that with the `throws` expression in the method signature. For example:

```
protected void someMethod() throws IOException
```

In your code, if you call a method that specifies that it throws one or types of exceptions, you have to handle it somehow, or add a `throws` to your method signature to pass it along up the *call stack* to methods that called your code. In the event of an exception, the Java language runtime will search for an exception handler somewhere up the stack, if there isn't one where the exception was thrown. If it doesn't find one by the time it reaches the top of the stack, it halts the program abruptly.

The good news is that most IDEs (and Eclipse is certainly among them) will tell you if your code needs to catch an exception thrown by a method you call. Then you can decide what to do with it.

There's much more to exception handling, of course, but it's too much for this tutorial. Hopefully what we've covered here will help you know what to expect.

Section 11. Java applications

What applications are

We've seen an application already, albeit a very simple one. Our `Adult` class has had a `main()` method since the beginning. That was necessary because you need such a method to have the Java language runtime execute your code. Typically, though, your domain objects won't have `main()` methods. Java applications typically consist of:

- A single class with a `main()` method that starts things off
- A bunch of other classes that do the work

In order to illustrate how this works, we need to add another class to our application. That class will be the "driver," so to speak.

Creating a driver class

Our driver class can be very simple:

```
package intro.core;

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

Follow these steps to create the class and really make it "drive" our program:

- Create the class in Eclipse using the same New Java Class toolbar button we used to create `Adult` in Declaring the class.
- Name the class `CommunityApplication`, and make sure you've selected the option to add a `main()` method to the class. Eclipse generates the class for you, including `main()`.
- Delete the `main()` method from the `Adult` class.

All that's left to do is to put something in our new `main()`:

```
package intro.core;

public class CommunityApplication {
        public static void main(String[] args) {
                Adult myAdult = new Adult();
                System.out.println(myAdult.walk(10));
        }
}
```

Create a new launch configuration in Eclipse, just like we did for the `Adult` class in Executing code in Eclipse, and run it. You should see that our object walked 10 steps.

What we have now is a simple application that starts in `CommunityApplication.main()`, and uses our `Adult` domain object. Of course, applications can be more complicated than this, but the basic idea remains the same. It's not uncommon for Java applications to have hundreds of classes. Once a primary driver class kicks things off, the program runs by having classes collaborate with each other to get the job done. Following the execution of the program can be quite disorienting if you're used to procedural programs that start at the beginning and run to the end, but it gets easier to understand with practice.

JAR files

How do you package a Java application up so others can use it, or give them code they can use in their own programs (such as a library of helpful objects, or a framework)? You create a Java Archive (JAR) file that packages up your code so other programmers can include it in their Java Build Path in Eclipse, or on their classpath if they're using command-line tools. Once again, Eclipse makes life much easier for you. Creating a JAR file in Eclipse (and many other IDEs) is a snap:

1. In your workspace, right-click on the `intro.core` package and select **Export**

2. Select **JAR file** in the **Export** dialog, then click **Next**
3. Browse to the location you want to put your JAR file, and name the file whatever you'd like and give it a .jar file extension
4. Click **Finish**

You should see your JAR file in the location you specified. Once you have a JAR file (yours or from another source), you can use the classes in it in your code if you put it on your Java Build Path in Eclipse. Doing that is no sweat either. There's no code we need to add to our path at the moment, but let's follow the steps you would take to do so:

1. Right-click on the Intro project in your workspace, then select **Properties**
2. In the Properties dialog, select the Libraries tab
3. There you see buttons for **Add JARs...** and **Add External JARs...**, which you can use to put JARs on your Java Build Path

Once the code (that is, the class files) in a JAR fileis are on your Java Build Path, you can use those classes in your Java code without getting a compile error. If the JAR file includes source code, you can associate those source files with the class files on your path. Then you can get pop-up code help and even open the code and see it.

Section 12. Writing good Java code

Java code introduction

You now know a good bit about Java syntax, but that's not really what professional programming is about. What makes a "good" Java program?

There are probably as manu answers to that question as there are professional Java programmers. But I have a few suggestions that I believe most professional Java programmers would agree improve the quality of the Java code they deal with on a daily basis. In the interest of full disclosure, I should say that I am biased in favor of agile methods like Extreme Programming (XP), so many of my opinions about "good" code are informed by the agile community in general, and by XP in particular. Still, I think most experienced professional Java programmers would agree with the points I'll make in this section.

Keep classes small

We created a simple `Adult` class in this tutorial. Even after we moved the `main()` method to another class, `Adult` was left with over 100 lines of code. It has over 20 methods, and it really doesn't do that much, compared to many classes you're likely to see (and create) professionally. This is a *small* class. It's not uncommon to see classes with 50 to 100 methods. What makes that worse than having fewer? Maybe nothing. The point of methods is to have the ones you need. If you need several helper methods that do essentially the same thing but take different parameters (such as our `addMoney()` methods), that's a fine choice. Just be sure to limit the list of methods to what you need, and no more.

Typically, a class with a very large number of methods has some that don't belong there, because that gargantuan object is doing too much. In his book *Refactoring* (see Resources for a link), Martin Fowler calls this the Foreign Method code smell. If you have an object with 100 methods, you should think hard about whether or not that single object should really be multiple objects. Large classes usually stink in college. It's the same with Java code.

Keep methods small

Small methods are just as preferable as small classes, and for similar reasons.

One of the gripes lots of experienced OO programmers have with the Java language is that it gave the masses OO, but didn't teach them how to do it well. In other words, it gave them enough rope to hang themselves (though at least not as much as C++ gives them). A common place to see this is in a class with a `main()` method five miles long, or a single method named `doIt()`. Just because you *can* put all your code in a single method in your class doesn't mean you *should.* The Java language has more syntactic sugar than lots of other OO languages, so some verbosity is required, but don't overdo it.

Think about such extremely long methods for a moment. Scrolling through ten screens of code to figure out what's going on makes it tough to figure out what's going on! What does the method do? You'll need to get a big cup of coffee and study it for a couple hours to figure out. A small, even tiny, method is an easily digestible chunk of code. Runtime efficiency isn't the reason to have small methods. Readability is the real prize. That will make your code easier to maintain, and easier to change as you add features.

Limit each method to the performance of a single job.

Named methods well

The best coding pattern I've ever come across (and I've forgotten the source) is called *intention-revealing method names.* Which of the two following method names is easier to decipher at a glance?

- `a()`
- `computeCommission()`

The answer should be obvious. For some reason, programmers seem to have an aversion to long method names. Certainly a ridiculously long name can be inconvenient, but a name long enough to be clear usually isn't ridiculously long. I have no problem with a method name like `aReallyLongMethodNameThatIsAbsolutelyClear()`. At 3:00 a.m. when I'm trying to figure out why my program isn't working, if I come across a method named `a()`, I want to beat somebody up.

Spend a few extra minutes coming up with a very descriptive method name; if possible, consider naming methods in a such way that makes your code read more like English, even if that means adding additional helper methods to do the job. For example, consider adding a helper method to make this code more readable:

```
if (myAdult.getWallet().isEmpty()) {
        do something
}
```

The `isEmpty()` method on `ArrayList` is helpful by itself, but that boolean condition in our `if` statement could benefit from a method on `Adult` called `hasMoney()` that looks like this:

```
public boolean hasMoney() {
       return !getWallet().isEmpty();
}
```

Then our `if` reads more like English:

```
if (myAdult.hasMoney()) {
        do something
}
```

This technique is simple, and perhaps trivial in this case, but it's amazingly powerful as code gets more complex.

Keep the number of classes to a minimum

One of the guidelines for simple design in XP is to accomplish a goal with as few classes as necessary, but no fewer. If you need another class, certainly add it. If another class would make the code simpler or make it easier to express your intent, go ahead and add the class. But there's no reason to have classes just to have them. It's common to have fewer classes in the early days of a project than you end up with, of course, but it's typically easier to refactor your code into more classes than to combine them. If you have a class with a large number of methods, analyze whether or not there's another object trapped in there just waiting to get out. If there is, create a new object.

On almost all of my Java projects, nobody was afraid to create classes, but we were

also constantly trying to reduce the number of classes without making our intent less clear.

Keep the number of comments to a minimum

I used to write copious comments in my code. You could read them like a book. Then I got a little smarter.

Every computer science program, every programming book, and lots of programmers I know tell you to comment code. In some cases, comments are helpful. In many cases, they make code maintenance more difficult. Think about what you have to do when you change code. Is there a comment there? If so, you had better change the comment, or it will be horribly out of date, and over time might not even describe the code at all anymore. It almost doubles your maintenance time, in my experience.

My rule of thumb is this: If the code is so hard to read and understand that it needs a comment, I need to make it clear enough not to need one. It might be too long, or do too much. If so, I make it simpler. It might be too cryptic. If so, I add helper methods that make it clear. In fact, in three years of Java programming with members of the same team, I can count the number of comments I wrote on my fingers and toes. Make the code clear! If you need an overall picture of what the system, or what some particular component, does, write a brief document to describe it.

Verbose comments typically are harder to maintain, usually don't express your intent as well as a small, well-written method does, and become outdated quickly. Don't depend on comments for much at all.

Use a consistent style

Coding style is really a matter of what's necessary and acceptable in your environment. I don't even know of a style I would call "typical." It's often a matter of personal taste. For example, here's some code that makes me twitch until I change it:

```
public void myMethod()
{
        if (this.a == this.b)
        {
                statements
        }
}
```

Why does that bother me? Because I'm personally biased against coding styles that add lines of code that are, in my opinion, unnecessary. The Java compiler recognizes the following code just the same, and I save several lines:

```
public void myMethod() {
        if (this.a == this.b)
                statements
}
```

Neither way is "right" or "wrong." One is simply shorter than the other. So what happens when I code with somebody who likes the first form better? We talk about it, pick a style we're going to use, then stick to it. The only hard and fast style rule is to *be consistent.* If everybody working on a project uses a different style, reading code will become difficult. Pick a style and don't vary it.

Avoid switch

Some Java programmers love `switch` statements. I used to think they were nice, but then I realized that a `switch` is really just a bunch of `if`s, and that usually means that conditional logic shows up in more than one place in my code. That's code duplication, and it's a no-no. Why? Because having the same code in multiple places makes code harder to change. If I have the same `switch` in three places, and I want to change how I handle a particular case, I have to change three bits of code.

Now, what if you can refactor the code such that you have a single `switch` statement? Great! I don't believe there's anything evil about using it. In some cases, it's more clear than nested `if`s. But if you see it cropping up in lots of places, you've got a problem you should fix. An easy way to keep that from happening is to avoid `switch` unless it's the best tool for the job. In my experience, it rarely is.

Be public

I saved the most controversial recommendation for last. Take a deep breath.

I believe you should err on the side of making all of your methods `public`. Instance variables should be `protected`.

Of course, many professional programmers would shudder at the thought, because if everything is public, anybody can change it, perhaps in unauthorized ways. In a world where everything is publicly accessible, you have to depend on programmer discipline to ensure that people won't access something they shouldn't, when they shouldn't. But there's very little in programming life more frustrating than wanting to access a variable or method that's not visible to you. If you restrict access to things in your code that you assume others shouldn't access, you're assuming you're omniscient. That's a dangerous assumption most of the time.

This frustration crops up frequently when you're using other people's code. You can see a method that does exactly what you want to do, but it isn't publicly accessible. Sometimes there's a good reason for that, and it makes sense to limit accessibility. Sometimes, though, the only reason it's not `public` is that the folks who wrote the code thought, "Nobody ever will need to access this." Or maybe they thought, "Nobody should access this because..." and then they didn't have a solid reason. Many times, people use `private` because its available. Don't do that.

Make methods `public` and variables `protected` until you have a good reason to restrict access.

In the footsteps of Fowler

Now you know how to create good Java code, and how to keep it good.

The best book in the industry on this subject is *Refactoring* by Martin Fowler (see Resources for link). It's even fun to read. *Refactoring* means changing the design of existing code without changing its results. Fowler talks about "code smells" that beg for refactoring, and goes into great detail about the various techniques (or "refactorings") for fixing them. In my opinion, refactoring and the ability to write code test-first (see Resources) 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, it will be easier to get a job, because you will be able to produce better results than most of your peers.

Writing Java code is relatively simple. Writing *good* Java code is a craft. Focus on becoming a craftsman or craftswoman.

Section 13. Wrap-up

Summary

In this tutorial, you've learned about OOP, discovered Java syntax that lets you create useful objects, and got to know an IDE that helps you control your development environment. You can create objects that can do a good number of things, although certainly not everything you can imagine. But you can extend your learning in several ways, including perusing the Java API and exploring other capabilities of the Java language through other developerWorks tutorials. See Resources for links to all of those.

The Java language certainly isn't perfect; every language has its quirks, and every programmer has a favorite. The Java platform is, however, a good tool that can help you write very good professional programs that are in high demand.

Resources

Learn

- The official Java Technology home page has links to all things related to the Java platform. You can find every "official" Java language resource you need right here, including the language specification and API documentation.
- Learn more about the command-line tools that come with Java.
- Visit the Java documentation page for a link to API documentation for each of the SDK versions.
- John Zukowski's Taming Tiger column on *developerWorks* offers a look at that latest version of the J2SE platform.
- The `javadoc` home page covers all the ins and outs of `javadoc`, including how to use the command-line tool and how to write your own `Doclet`s that let you create custom formats for your documentation.
- The Sun Java tutorial is an excellent resource. It's a gentle introduction to the language, and contains much more material than this tutorial, including links to other tutorials that go into more detail about various aspects of the language.
- Refactoring, Martin Fowler et al (Addison-Wesley, 1999) is an excellent resource for improving existing code. >
- The New to Java technology page is a clearinghouse for *developerWorks* resources for beginning Java developers, including links to tutorials and certification resources.
- You'll find articles about every aspect of Java programming in the developerWorks Java technology zone.
- Also see the Java technology zone tutorials page for a complete listing of free Java-focused tutorials from *developerWorks* .

Get products and technologies

- You can download Eclipse from the Eclipse Web site.

About the author

Roy W. Miller
Roy Miller is an independent software development coach, programmer, and author. He began his career at Andersen Consulting (now Accenture), and most recently spent three years using the Java platform professionally at RoleModel Software, Inc., in Holly Springs, NC. He has developed software, managed teams, and coached other programmers at clients ranging from two-person start-ups to Fortune 50 companies. Roy is also a frequent contributor to *developerWorks*. For technical questions or comments about the content of this tutorial, contact Roy at roy@roywmiller.com.