Intermediate Java programming

Skill Level: Introductory

Roy W. Miller (roy@roywmiller.com)
Programmer
Independent

13 Jan 2005

The Java™ language provides most of what professional programmers expect to see in a language, and even in an object-oriented language. Beyond the basics, though, the Java language provides some useful tools for creating sophisticated programs. This tutorial will introduce you to some of these more advanced Java language features commonly seen on typical Java technology development projects.

Section 1. Before you start

About this tutorial

This tutorial introduces you to capabilities of the Java language that are more sophisticated than those covered in the "Introduction to Java programming" tutorial (see Resources for a link to this and other material referenced in this tutorial). In order to get to the most out of this tutorial, you should have completed that introductory tutorial, or be familiar with the concepts covered in it.

The Java language provides a vast set of tools that can help a programmer accomplish almost any task. In this tutorial, we'll cover a few of the more advanced tools that are commonly used on Java development projects, including the following:

- Inheritance and abstraction
- Interfaces
- Nested classes
- Regular expressions
- Collections

- Dates
- I/O

Prerequisites

The content of this tutorial is geared toward moderately experienced Java programmers who might not be familiar with some of the more involved features of the language. It assumes a general knowledge of downloading and installing software, and a general knowledge of object-oriented programming (OOP) with the Java language. You might not use the more advanced Java language features we'll talk about here in every application -- in fact, you probably shouldn't -- but it's wise for a professional programmer to know about them and be able to use them when appropriate.

In addition to familiarity with the concepts covered in the "Introduction to Java programming" tutorial see Resources), you'll need to have the following installed to run the examples or sample code in this tutorial:

- JDK 1.4.2 or higher (5.0 recommended)
- The Eclipse IDE

All code examples in this tutorial have been tested with JDK 5.0 on the Windows XP platform, but should work without modification using JDK 1.4.x. You can download source code for the tutorial from Resources. It's contained in intermediate.jar, which you can import into your Eclipse workspace.

The sample JAR file does not necessarily contain the code for every example in this tutorial in its final form. Instead, it contains the core of what we'll cover, minus some of the iterative modifications that we'll apply to the code as we progress through the article. Modifying the core to explore the language features we'll cover in this tutorial is left as an exercise for you.

Section 2. Inheritance and abstraction

What's inheritance?

Classes in Java code exist in hierarchies. Classes above a given class in a hierarchy are *superclasses* of that class. That particular class is a *subclass* of every class higher up. A subclass *inherits* from its superclasses. The class `Object` is at the top of every class's hierarchy. In other words, every class is a subclass of (and inherits from) `Object`.

For example, suppose we have an `Adult` class that looks like this:

```
public class Adult {
        protected int age = 0;
        protected String firstname = "firstname";
        protected String lastname = "lastname";
        protected String gender = "MALE";
        protected int progress = 0;

        public Adult() { }
        public void move() {
                System.out.println("Moved.");
        }
        public void talk() {
                System.out.println("Spoke.");
        }
}
```

Our `Adult` class implicitly inherits from `Object`. That's assumed for any class, so you don't have to type `extends Object` in the class definition. But what does it mean to say that our class inherits from its superclass(es)? It simply means that `Adult` has access to the exposed variables and methods in its superclasses. In this case, it means that `Adult` can see and use the following from any of its superclasses (we have only one at the moment):

- `public` methods and variables
- `protected` methods and variables
- *Package protected* methods and variables (that is, those without an access specifier), if the superclass is in the same package as `Adult`

Constructors are special. They aren't full-fledged OO members, so they aren't inherited.

If a subclass *overrides* a method or variable from the superclass -- if the subclass implements a member with the same name, in other words -- that *hides* the superclass's member. To be accurate, overriding a variable hides it, and overriding a method simply overrides it, but the effect is the same: The overridden member is essentially hidden. You can still get to the members of the superclass by using the `super` keyword:

```
super.hiddenMemberName
```

In the case of `Adult`, all it inherits at the moment are the methods on `Object` (`toString()`, for example). So, the following code snippet is perfectly acceptable:

```
Adult anAdult = new Adult();
anAdult.toString();
```

The `toString()` method doesn't exist explicitly on `Adult`, but `Adult` inherits it.

There are "gotchas" here that you should keep in mind. First, it's very easy to give variables and methods in a subclass the same name as variables and methods in that class's superclass, then get confused when you can't call an inherited method. Remember, when you give a method the same name in a subclass as one that already exists in a superclass, you've hidden it. Second, constructors aren't inherited, *but they are called.* There is an implicit call to the superclass constructor in any subclass constructor you write, and it's the first thing that the subclass constructor does. You must live with this; there's nothing you can do to change it. For example, our `Adult` constructor actually looks like this at runtime, even though we didn't type anything in the body:

```
public Adult() {
        super();
}
```

That line in the constructor body calls the no-argument constructor on the superclass. In this case, that's the constructor of `Object`.

Defining a class hierarchy

Suppose we have another class called `Baby`. It looks like this:

```
public class Baby {
        protected int age = 0;
        protected String firstname = "firstname";
        protected String lastname = "lastname";
        protected String gender = "MALE";
        protected int progress = 0;

        public Baby() {
        }
        public void move() {
                System.out.println("Moved.");
        }
        public void talk() {
                System.out.println("Spoke.");
        }
}
```

Our `Adult` and `Baby` classes look very similar. In fact, they're almost identical. That kind of code duplication makes maintaining code more painful than it needs to be. We can create a superclass, move all the common elements up to that class, and remove the code duplication. Our superclass could be called `Person`, and it might look like this:

```
public class Person {
        protected int age = 0;
        protected String firstname = "firstname";
        protected String lastname = "lastname";
        protected String gender = "MALE";
        protected int progress = 0;
        public Person() {
        }
        public void move() {
                System.out.println("Moved.");
```

```
        }
        public void talk() {
                System.out.println("Spoke.");
        }
}
```

Now we can have `Adult` and `Baby` subclass `Person`, which makes those two classes pretty simple at the moment:

```
public class Adult {
        public Adult() {
        }
}
public class Baby {
        public Baby() {
        }
}
```

One we have this hierarchy, we can refer to an instance of each subclass as an instance of any of its superclasses in the hierarchy. For example:

```
Adult anAdult = new Adult();
System.out.println("anAdult is an Object: " + (Adult instanceof Object));
System.out.println("anAdult is a Person: " + (Adult instanceof Person));
System.out.println("anAdult is anAdult: " + (Adult instanceof Adult));
System.out.println("anAdult is a Baby: " + (Adult instanceof Baby));
```

This code will give us three `true` results and one `false` result. You can also cast an object to any type higher up in its hierarchy, like so:

```
Adult anAdult = new Adult();
Person aPerson = (Person) anAdult;
aPerson.move();
```

This code will compile without problems. We can cast an `Adult` to type `Person`, then call a `Person` method on it.

Because we have this hierarchy, the code on our subclasses is simpler. But do you see a problem here? Now all `Adults` and all `Babys` (excuse the bad plural) will talk and move in the same way. There's only one implementation of each behavior. That's not what we want, because grownups don't speak or move like babies. We could override `move()` and `talk()` on the subclasses, but then we've got essentially useless "standard" behavior defined on our superclass. What we really want is a way to force each of our subclasses to move and talk in its own particular way. That's what *abstract classes* are for.

Abstraction

In an OO context, *abstraction* refers to the act of generalizing data and behavior to a type higher up the hierarchy than the current class. When you move variables or methods from a subclass to a superclass, you're *abstracting* those members.

Those are general terms, and they apply in the Java language. But the language also adds the concepts of *abstract classes* and *abstract methods.* An abstract class is a class that can't be instantiated. For example, you might create a class called `Animal`. It makes no sense to instantiate such a class: In practice, you'd only want to create instances of a *concrete class* like `Dog`. But all `Animals` have some things in common, such as the ability to make noise. Saying that an `Animal` makes noise doesn't tell you much. The noise it makes depends on the kind of animal it is. How do you model that? You define the common stuff on the abstract class, and you force subclasses to implement concrete behavior specific to their types.

You can have both abstract and concrete classes in your hierarchies.

Using abstraction

Our `Person` class contains some method behavior that we don't know we need yet. Let's remove it and force subclasses to implement that behavior polymorphically. We can do that by defining the methods on `Person` to be abstract. Then our subclasses will have to implement those methods.

```
public abstract class Person {
        ...
        abstract void move();
        abstract void talk();
}

public class Adult extends Person {
        public Adult() {
        }
        public void move() {
                System.out.println("Walked.");
        }
        public void talk() {
                System.out.println("Spoke.");
        }
}

public class Baby extends Person {
        public Baby() {
        }
        public void move() {
                System.out.println("Crawled.");
        }
        public void talk() {
                System.out.println("Gurgled.");
        }
}
```

What have we done in this listing?

- We changed `Person` to make the methods abstract, forcing subclasses to implement them.
- We made `Adult` subclass `Person`, and implemented the methods.
- We made `Baby` subclass `Person`, and implemented the methods.

When you declare a method to be `abstract`, you require subclasses to implement

the method, or to be abstract themselves and pass along the implementation responsibility to their subclasses. You can implement some methods on an abstract class, and force subclasses to implement others. That's up to you. Simply declare the ones you don't want to implement as `abstract`, and don't provide a method body. If a subclass fails to implement an abstract method from a superclass, the compiler will complain.

Now that both `Adult` and `Baby` subclass `Person`, we can refer to an instance of either class as being of type `Person`.

Refactoring to abstract behavior

We now have `Person`, `Adult`, and `Baby` in our hierarchy. Suppose we wanted to make the two subclasses more realistic by changing their `move()` methods, like this:

```
public class Adult extends Person {
        ...
        public void move() {
                this.progress++;
        }
         ...
}

public class Baby extends Person {
        ...
        public void move() {
                this.progress++;
        }
         ...
}
```

Now each class updates its instance variable to reflect some progress being made whenever we call `move()`. But notice that the behavior is the same again. It makes sense to refactor the code to remove this code duplication. The most likely refactoring is to move `move()` to `Person`.

Yes, we're adding the method implementation back to the superclass we just took it out of. This is a simplistic example, so this back-and-forth might seem wasteful. But what we just experienced is a common occurrence when you write Java code. You often see classes and methods change as your system grows, and sometimes you end up with code duplication that you can refactor to superclasses. You might even do that, decide it was a mistake, and put the behavior back down in the subclasses. You simply can't know the right place for all behavior at the beginning of the development process. You learn the right place for behavior as you go.

Let's refactor our classes to put `move()` back on the superclass:

```
public abstract class Person {
        ...
        public void move() {
                this.progress++;
        }
        public abstract void talk();
}
```

```
public class Adult extends Person {
        public Adult() {
        }
        public void talk() {
                System.out.println("Spoke.");
        }
}

public class Baby extends Person {
        public Baby() {
        }
        public void talk() {
                System.out.println("Gurgled.");
                }
}
```

Now our subclasses implement their different versions of `talk()`, but share the same behavior for `move()`.

When to abstract ... and when not to

Deciding when to abstract (or create a hierarchy) is a hotly debated topic in OO circles, especially among Java language programmers. Certainly there are few right and wrong answers about how to structure class hierarchies. This is an area where conscientious and skilled practitioners can (and often do) disagree. That said, there are some good rules of thumb to follow with regard to hierarchies.

First, don't abstract first. Wait until the code tells you that you should. It is almost always better to refactor your way to an abstraction than to assume that you need it at the outset. Don't assume that you need a hierarchy. Many Java programmers overuse hierarchies.

Second, resist the use of abstract classes when you can. They're not bad; they're just restrictive. We used an abstract class to force our subclasses to implement certain behavior. Would an interface (which we'll discuss in Interfaces) be a better idea? Quite possibly. Your code will be easier to understand if it isn't made up of complex hierarchies with a mix of overridden and implemented methods. You might have a method defined three or four classes up the chain. By the time you use it in a sub-sub-sub-subclass, you might have to hunt to discover what that method does. That can make debugging frustrating.

Third, *do* use a hierarchy and/or abstract classes when it's smart to do so. There are many coding patterns that make use of the Java language abstract method and abstract class concepts, such as the Gang of Four Template Method pattern (see Resources).

Fourth, understand the price you pay when you use a hierarchy prematurely. It really can lead you down the wrong path quickly, because having the classes there, named as they are, with the methods they have, makes it very easy to assume all of that *should* be as it is. Maybe that hierarchy made sense when you created it, but it might not make sense anymore. Inertia can make it resistant to change.

In a nutshell, be smart about using hierarchies. Experience will help you be smarter,

but it won't make you all-wise. Remember to refactor.

Section 3. Interfaces

What's an interface?

The Java language includes the concept of an *interface,* which is simply a named set of publicly available behaviors and/or constant data elements for which an implementer of that interface must provide code. It doesn't specify behavior details. In essence (and to the Java compiler), an interface defines a new data type, and it's one of the more powerful features of the language.

Other classes *implement* the interface, which means that they can use any constants in that interface by name, and that they must specify behavior for the method definitions in the interface.

Any class in any hierarchy can implement a particular interface. That means that otherwise unrelated classes can implement the same interface.

Defining interfaces

Defining an interface is straightforward:

```
public interface interfaceName {
        final constantType
                    constantName = constantValue;
        ...
        returnValueType
                    methodName( arguments );
        ...
}
```

An interface declaration looks very similar to a class declaration, except that you use the `interface` keyword. You can name the interface anything you want, as long as the name is valid, but by convention interface names look like class names. You can include constants, method declarations, or both in an interface.

Constants defined in an interface look like constants defined in classes. The `public` and `static` keywords are assumed for constants defined in an interface, so you don't have to type them. (`final` is assumed as well, but most programmers type it out anyway).

Methods defined in an interface look different (generally speaking) from methods defined in classes, because methods in an interface have no implementation. They end with semicolon after the method declaration, and they don't include a body. Any

implementer of the interface is responsible for supplying the body of the method. The `public` and `abstract` keywords are assumed for methods, so you don't have to type them.

You can define hierarchies of interfaces just as you can define hierarchies of classes. You do this with the `extends` keyword, like so:

```
public interface interfaceName extends superinterfaceName, ... {
        interface body...
}
```

A class can be a subclass of only one superclass, but an interface can extend as many other interfaces as you want. Just list them after `extends`, separated by commas.

Here's an example of an interface:

```
public interface Human {
        final String GENDER_MALE = "MALE";
        final String GENDER_FEMALE = "FEMALE";
        void move();
        void talk();
}
```

Implementing interfaces

To use an interface, you simply *implement* it, which means providing behavior for the methods defined in the interface. You do that with the `implements` keyword:

```
public class className extends superclassName implements
   interfaceName, ... {
        class body
}
```

By convention, the `extends` clause (if there is one) comes first, followed by the `implements` clause. You can implement more than one interface by listing the interface names separated by commas.

For example, we could have our `Person` class implement the `Human` interface (saying "implements `Human`" means the same thing) like so:

```
public abstract class Person implements Human {
        protected int age = 0;
        protected String firstname = "firstname";
        protected String lastname = "lastname";
        protected String gender = Human.GENDER_MALE;
        protected int progress = 0;
        public void move() {
                this.progress++;
        }
}
```

When we implement the interface, we provide behavior for the methods. We have to implement those methods with signatures that match the ones in the interface, with the addition of the `public` access modifier. But we've implemented only `move()` on `Person`. Don't we have to implement `talk()`? No, because `Person` is an abstract class, and the `abstract` keyword is assumed for methods in an interface. That means any abstract class implementing the interface can implement what it wants, and ignore the rest. If it does not implement one or more methods, it passes that responsibility on to its subclasses. In our `Person` class, we chose to implement `move()` and not `talk()`, but we could have chosen to implement neither.

The instance variables in our class aren't defined in the interface. Some helpful constants are, and we can reference them by name in any class that implements the interface, as we did when we initialized `gender`. It's also quite common to see interfaces that contain only constants. If that's the case, you don't have to implement the interface to use those constants. Simply import the interface (if the interface and the implementing class are in the same package, you don't even have to do that) and reference the constants like this:

```
interfaceName.constantName
```

Using interfaces

An interface defines a new reference data type. That means that you can refer to an interface anywhere you would refer to a class, such as when you cast, as illustrated by the following code snippet from a `main()` method you can add to `Adult`:

```
public static void main(String[] args) {
        ...
        Adult anAdult = new Adult();
        anAdult.talk();
        Human aHuman = (Human) anAdult;
        aHuman.talk();
}
```

Both calls to `talk()` will display `Spoke.` on the console. Why? Because an `Adult` is a `Human` once it implements that interface. You can cast an `Adult` as a `Human`, then call methods defined by the interface, just as you can cast `anAdult` to `Person` and call `Person` methods on it.

`Baby` also implements `Human`. An `Adult` is not a `Baby`, and a `Baby` is not an `Adult`, but both can be described as type `Human` (or as type `Person` in our hierarchy). Consider this code somewhere in our system:

```
public static void main(String[] args) {
        ...
        Human aHuman = getHuman();
        aHuman.move();
}
```

Is the `Human` an `Adult` or a `Baby`? We don't have to care. As long as whatever we get back from `getPerson()` is of type `Human`, we can call `move()` on it and expect it to respond accordingly. We don't even have to care if the classes implementing the interface are in the same hierarchy.

Why interfaces?

There are three primary reasons for using interfaces:

- To create convenient or descriptive namespaces.
- To relate classes in different hierarchies.
- To hide underlying type details from your code.

When you create an interface to collect related constants, that interface gives you a descriptive name to use to refer to those constants. For example, you could have an interface named `Language` to store constant string names for languages. Then you could refer to those language names as `Language.ENGLISH` and the like. This can make your code easier to read.

The Java language supports single inheritance only. In other words, a class can only be a subclass of a single superclass. That's rather limiting sometimes. With interfaces, you can relate classes in different hierarchies. That's a powerful feature of the language. In essence, an interface simply specifies a set of behaviors that all implementors of the interface must support. It's possible that the only relationship that will exist between classes implementing the interface is that they share those behaviors that the interface defines. For example, suppose we had an interface called `Mover`:

```
public interface Mover {
        void move();
}
```

Now suppose that `Person` extended that interface. That means that any class implementing `Person` would also be a `Mover`. `Adult` and `Baby` would qualify. But so would `Cat`, or `Vehicle`. It's reasonable to assume that `Mountain` would not. Any class that implemented `Mover` would have the `move()` behavior. A `Mountain` instance wouldn't have it.

Last, but not least, using interfaces lets you ignore the details of specific types when you want to. Recall our example of calling `getPerson()`. We didn't care what underlying type we got back; we just wanted it to be something we could call `move()` on.

All of these are good reasons to use interfaces. Using one simply because you can is not.

Section 4. Nested classes

What's a nested class?

As its name suggests, a *nested class* in the Java language is a class declared within another class. Here's a simple example:

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

Typically, good programmers define nested classes when the nested class only makes sense within the context of the enclosing class. Some common examples include the following:

- Event handlers within a UI class
- Helper classes for UI components within those components
- Adapter classes to convert the innards of one class to some other form for users of the class

You can define a nested class as `public`, `private`, or `protected`. You also can define a nested class as `final` (to prevent it from being changed), `abstract` (meaning that it can't be instantiated), or `static`.

When you create a `static` class within another class, you're creating what is appropriately called a nested class. A nested class is defined within another class, but can exist outside an instance of the enclosing class. If your nested class isn't `static`, it can exist only within an instance of the enclosing class, and is more appropriately called an *inner class.* In other words, all inner classes are nested classes, but not all nested classes are inner classes. The vast majority of the nested classes you will encounter in your career will be inner classes, rather than simply nested ones.

Any nested class has access to all of the members of the enclosing class, even if they're declared `private`.

Defining nested classes

You define a nested class just as you define a non-nested class, but you do it within an enclosing class. For a somewhat contrived example, let's define a `Wallet` class inside `Adult`. While in real life you could have a `Wallet` apart from an `Adult`, it

wouldn't be all that useful, and it makes good sense that every `Adult` has a `Wallet` (or at least something to hold money, and `MoneyContainer` sounds a little odd). It also makes sense that `Wallet` wouldn't exist on `Person`, because a `Baby` doesn't have one, and all subclasses of `Person` would inherit it if it existed up there.

Our `Wallet` will be quite simple, since it only serves to illustrate the definition of a nested class:

```
protected class Wallet {
        protected ArrayList bills = new ArrayList();

        protected void addBill(int aBill) {
                bills.add(new Integer(aBill));
        }

        protected int getMoneyTotal() {
                int total = 0;
                for (Iterator i = bills.iterator(); i.hasNext(); ) {
                        Integer wrappedBill = (Integer) i.next();
                        int bill = wrappedBill.intValue();
                        total += bill;
                }
                return total;
        }
}
```

We'll define that class inside `Adult`, like this:

```
public class Adult extends Person {
        protected Wallet wallet = new Wallet();
        public Adult() {
        }
        public void talk() {
                System.out.println("Spoke.");
        }
        public void acceptMoney(int aBill) {
                this.wallet.addBill(aBill);
        }
        public int moneyTotal() {
                return this.wallet.getMoneyTotal();
        }
        protected class Wallet {
                ...
        }
}
```

Notice that we added `acceptMoney()` to let an `Adult` accept more money. (Feel free to expand the example to force your `Adult` to give some up, which is a more common event in real life.)

Once we have our nested class and our new `acceptMoney()` method, we can use them like this:

```
Adult anAdult = new Adult();
anAdult.acceptMoney(5);
System.out.println("I have this much money: " + anAdult.moneyTotal());
```

Executing this code should report that an`Adult` has a money total of 5.

Simplistic event handling

The Java language defines an event handling approach, with associated classes, that allows you to create and handle your own events. But event handling can be much simpler than that. All you really need is some logic to generate an "event" (which really doesn't have to be an event class at all), and some logic to listen for that event and then respond appropriately. For example, suppose that whenever a `Person` moves, our system generates (or *fires*) a `MoveEvent`, which we can choose to handle or not. This will require several changes to our system. We have to:

- Create an "application" class to launch our system and illustrate using the anonymous inner class.
- Create a `MotionListener` that our application can implement, and then handle the event in the listener.
- Add a `List` of listeners to `Adult`.
- Add an `addMotionListener()` method to `Adult` to register a listener.
- Add a `fireMoveEvent()` method to `Adult` so that it can tell listeners when to handle the event.
- Add code to our application to create an `Adult` and register itself as a handler.

This is all straightforward. Here's our `Adult` with the new stuff:

```
public class Adult extends Person {
        protected Wallet wallet = new Wallet();
        protected ArrayList listeners = new ArrayList();
        public Adult() {
        }
        public void move() {
          super.move(); fireMoveEvent();
        }
        ...
        public void addMotionListener(MotionListener aListener) {
          listeners.add(aListener);
        }
        protected void fireMoveEvent() {
          Iterator iterator = listeners.iterator();
          while(iterator.hasNext()) {
            MotionListener listener = (MotionListener) iterator.next();
            listener.handleMove(this);
        }
        }
        protected class Wallet {
                ...
        }
}
```

Notice that we now override `move()`, call `move()` on `Person` first, then call `fireMoveEvent()` to tell listeners to respond. We also added `addMotionListener()` to add a `MotionListener` to a running list of listeners. Here's what a `MotionListener` looks like:

```
public interface MotionListener {
        public void handleMove(Adult eventSource);
}
```

All that's left is to create our application class:

```
public class CommunityApplication implements MotionListener {
        public void handleMove(Adult eventSource) {
                System.out.println("This Adult moved: \n" + eventSource.toString());
        }
        public static void main(String[] args) {
                CommunityApplication application = new CommunityApplication();
                Adult anAdult = new Adult();
                anAdult.addMotionListener(application);
                anAdult.move();
        }
}
```

This class implements the `MotionListener` interface, which means that it implements `handleMove()`. All we do here is print a message to illustrate what happens when an event is fired.

Anonymous inner classes

Anonymous inner classes allow you to define a class in place, without naming it, to provide some context-specific behavior. It's a common approach for event handlers in user interfaces, which is a topic beyond the scope of this tutorial. But we can use an anonymous inner class even in our simplistic event-handling example.

You can convert the example from the previous page to use an anonymous inner class by changing the call to `addMotionListener()` in `CommunityApplication.main()` like so:

```
anAdult.addMotionListener(new MotionListener() {
  public void handleMove(Adult eventSource) {
    System.out.println("This Adult moved: \n" + eventSource.toString());
  }
});
```

Rather than having `CommunityApplication` implement `MotionListener`, we declared an unnamed (and thus anonymous) inner class of type `MotionListener`, and gave it an implementation of `handleMove()`. The fact that `MotionListener` is an interface, not a class, doesn't matter. Either is acceptable.

This code produces exactly the same result as the previous version, but it uses a more common and expected approach. You will almost always see event handlers implemented with anonymous inner classes.

Using nested classes

Nested classes can be very useful. They also can cause pain.

Use a nested class when it would make little sense to define the class outside of an enclosing class. In our example, we could have defined `Wallet` outside `Adult` without feeling too badly about it. But imagine something like a `Personality` class. Do you ever have one outside a `Person` instance? No, so it makes perfect sense to define it as a nested class. A good rule of thumb is that you should define a class as non-nested until it's obvious that it should be nested, then refactor to nest it.

Anonymous inner classes are the standard approach for event handlers, so use them for that purpose. In other cases, be very careful with them. Unless anonymous inner classes are small, focused, and familiar, they obfuscate code. They can also make debugging more difficult, although the Eclipse IDE helps minimize that pain. Generally, try not to use anonymous inner classes for anything but event handlers.

Section 5. Regular expressions

What is a regular expression?

A *regular expression* is essentially a pattern to describe a set of strings that share that pattern. For example, here's a set of strings that have some things in common:

- a string
- a longer string
- a much longer string

Each of these strings begins with "a" and ends with "string." The Java Regular Expression API helps you figure that out, and do interesting things with the results.

The Java language Regular Expression (or *regex*) API is quite similar to the regex facilities available in the Perl language. If you're a Perl programmer, you should feel right at home, at least with the Java language regex pattern syntax. If you're not used to regex, however, it can certainly look a bit weird. Don't worry: it's not as complicated as it seems.

The regex API

The Java language's regex capability has three core classes that you'll use almost all the time:

- `Pattern`, which describes a string pattern.

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

The best way to learn about regex is by example, so in this section we'll create a simple one in `CommunityApplication.main()`. Before we do, however, it's important to understand some regex pattern syntax. We'll discuss that in more detail in the next panel.

Pattern syntax

A regex *pattern* describes the structure of the string that the expression will try to find in an input string. This is where regular expressions can look a bit strange. Once you understand the syntax, though, it becomes less difficult to decipher.

Here are some of the most common pattern constructs you can use in pattern strings:

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

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

Matching

Armed with our new understanding of patterns, here's a simple example of code that

uses the classes in the Java Regular Expression API:

```
Pattern pattern = Pattern.compile("a.*string");
Matcher matcher = pattern.matcher("a string");

boolean didMatch = matcher.matches();
System.out.println(didMatch);

int patternStartIndex = matcher.start();
System.out.println(patternStartIndex);

int patternEndIndex = matcher.end();
System.out.println(patternEndIndex);
```

First, we create a `Pattern`. We do that by calling `compile()`, a static method on `Pattern`, with a string literal representing the pattern we want to match. That literal uses regex pattern syntax, which we can understand now. In this example, the English translation of the pattern is: "Find a string of the form 'a', followed by zero or more characters, following by 'string'".

Next, we call `matcher()` on our `Pattern`. That call creates a `Matcher` instance. When that happens, the `Matcher` searches the string we passed in for matches against the pattern string we created the `Pattern` with. As you know, every Java language string is an indexed collection of characters, starting with 0 and ending with the string length minus one. The `Matcher` parses the string, starting at 0, and looks for matches against the pattern.

After that process completes, the `Matcher` contains lots of information about matches found (or not found) in our input string. We can access that information by calling various methods on our `Matcher`:

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

In our simple example, there is a single match starting at 0 and ending at 7. Thus, the call to `matches()` returns `true`, the call to `start()` returns 0, and the call to `end()` returns 8. If there were more in our string than the characters in the pattern we searched for, we could use `lookingAt()` instead of `matches()`. `lookingAt()` searches for substring matches for our pattern. For example, consider the following string:

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

We could search it for `a.*string` and get a match if we used `lookingAt()`. If we used `matches()` instead, it would return `false`, because there's more to the string than just what's in the pattern.

Complex patterns

Simple searches are easy with the regex classes, but much greater sophistication is possible.

You might be familiar with a *wiki,* a Web-based system that lets users modify pages to "grow" a site. Wikis, whether written in the Java language or not, are based almost entirely on regular expressions. Their content is based on string input from users, which is parsed and formatted by regular expressions. One of the most prominent features of wikis is that any user can create a link to another topic in the wiki by entering a *wiki word,* which is typically a series of concatenated words, each of which begins with an uppercase letter, like this:

```
MyWikiWord
```

Assume the following string:

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

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

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

Here's some code to search for wiki words:

```
String input = "Here is a WikiWord followed by AnotherWikiWord, then SomeWikiWord.";
Pattern pattern = Pattern.compile("[A-Z][a-z]*([A-Z][a-z]*)+");
Matcher matcher = pattern.matcher(input);

while (matcher.find()) {
        System.out.println("Found this wiki word: " + matcher.group());
}
```

You should see the three wiki words in your console.

Replacing

Searching for matches is useful, but we also can manipulate the string once we find a match. We can do that by replacing matches with something else, just as you might search for some text in a word processing program and replace it with something else. There are some methods on `Matcher` to help us:

- `replaceAll()`, which replaces all matches with a string we specify.
- `replaceFirst()`, which replaces only the first match with a string we specify.

Using these methods is straightforward:

```
String input = "Here is a WikiWord followed by AnotherWikiWord, then SomeWikiWord.";
Pattern pattern = Pattern.compile("[A-Z][a-z]*([A-Z][a-z]*)+");
Matcher matcher = pattern.matcher(input);
System.out.println("Before: " + input);

String result = matcher.replaceAll("replacement");
System.out.println("After: " + result);
```

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

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

If we had used `replaceFirst()`, we would've seen this:

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

Groups

We also can get a little fancier. When you search for matches against a regex pattern, you can get information about what you found. We already saw some of that with the `start()` and `end()` methods on `Matcher`. But we also can reference matches via *capturing groups.* In each pattern, you typically create groups by enclosing parts of the pattern in parentheses. Groups are numbered from left to right, starting with 1 (group 0 represents the entire match). Here is some code that replaces each wiki word with a string that "wraps" the word:

```
String input = "Here is a WikiWord followed by AnotherWikiWord, then SomeWikiWord.";
Pattern pattern = Pattern.compile("[A-Z][a-z]*([A-Z][a-z]*)+");
Matcher matcher = pattern.matcher(input);
System.out.println("Before: " + input);

String result = matcher.replaceAll("blah$0blah");
System.out.println("After: " + result);
```

Running this code should produce this result:

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

In this code, we referenced the entire match by including `$0` in the replacement string. Any portion of a replacement string that takes the form `$<some int>` refers to the group identified by the integer (so `$1` refers to group 1, and so on). In other

words, `$0` is equivalent to this:

```
matcher.group(0);
```

We could've accomplished the same replacement goal by using some other methods, rather than calling `replaceAll()`:

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

We get these results again:

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

A simple example

Our `Person` hierarchy doesn't offer us many opportunities for handling strings, but we can create a simple example that lets us use some of the regex skills we've learned.

Let's add a `listen()` method:

```
public void listen(String conversation) {
        Pattern pattern = Pattern.compile(".*my name is (.*).");
        Matcher matcher = pattern.matcher(conversation);

        if (matcher.lookingAt())
                System.out.println("Hello, " + matcher.group(1) + "!");
        else
                System.out.println("I didn't understand.");
}
```

This method lets us address some `conversation` to an `Adult`. If that string is of a particular form, our `Adult` can respond with a nice salutation. If not, it can say that it doesn't understand.

The `listen()` method checks the incoming string to see if it matches a certain pattern: one or more characters, followed by "my name is ", followed by one or more characters, followed by a period. We use `lookingAt()` to search a substring of the input for a match. If we find one, we construct a salutation string by grabbing what comes after "my name is", which we assume will be the name (that's what group 1 will contain). If we don't find one, we reply that we don't understand. Obviously, our `Adult` isn't much of a conversationalist at the moment.

This is an almost trivial example of the Java language's regex capabilities, but it illustrates how they can be used.

Clarifying expressions

Regular expressions can be cryptic. It's easy to get frustrated with code that looks very much like Sanskrit. Naming things well and building expressions can help.

For example, here's our pattern for a wiki word:

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

Now that you understand regex syntax, you should be able to read that without too much work, but our code would be much easier to understand if we declared a constant to hold the pattern string. We could name it something like `WIKI_WORD`. Our `listen()` method would start like this:

```
public void listen(String conversation) {
        Pattern pattern = Pattern.compile(WIKI_WORD);
        Matcher matcher = pattern.matcher(conversation);
        ...
}
```

Another trick that can help is to define constants for the parts of patterns, then build up more complex patterns as assemblies of named parts. Generally speaking, the more complicated the pattern, the more difficult it is to decipher, and the more prone to error it is. You'll find that there's no real way to debug regular expressions other than by trial and error. Make life simpler by naming patterns and pattern components.

Section 6. Collections

Introduction

The Java Collections Framework is large. In the Introduction to Java programming tutorial, I talked about the `ArrayList` class, but that's only scratching the surface. There are many classes and interfaces in the framework. We'll cover some more, although not all of them, here.

Collection interfaces and classes

The Java Collections Framework is based on concrete implementations of several

interfaces that define types of collections:

- The `List` interface defines a navigable collection of `Object` elements.
- The `Set` interface defines a collection with no duplicate elements.
- The `Map` interface defines a collection of key-value pairs.

We'll talk about several concrete implementations in this tutorial. This isn't an exhaustive list, but you're likely to see the following frequently on Java language development projects:

Interface	Implementation(s)
List	`ArrayList`, `Vector`
Set	`HashSet`, `TreeSet`
Map	`HashMap`

All of the interfaces in the framework except `Map` are subinterfaces of the `Collection` interface, which defines the most general structure of a collection. Each collection consists of *elements.* As implementors of subinterfaces of `Collection`, all collections share some (intuitive) behavior:

- Methods that describe the size of the collection (such as `size()` and `isEmpty()`)
- Methods that describe the contents of the collection (such as `contains()` and `containsAll()`)
- Methods that support manipulation of the collection's contents (such as `add()`, `remove()`, and `clear()`)
- Methods that let you convert a collection to an array (such as `toArray()`)
- A method that lets you get an iterator on the array (`iterator()`)

We'll talk about some of these in this section. Along the way, we'll discuss what an *iterator* is, and how to use it.

Note that `Maps` are special. They aren't really collections at all. However, they behave very much like collections, so we'll talk about them in this section as well.

List implementations

Older versions of the JDK contained a class called `Vector`. It's still there in newer versions, but you should use it only when you need a *synchronized* collection -- that is, one that is thread-safe. (Threading is beyond the scope of this article; we'll briefly discuss the concept in Summary.) In other cases, you should use the `ArrayList` class. You can still use `Vector`, but it imposes some overhead you often don't need.

An `ArrayList` is what it sounds like: an ordered list of elements. We already saw how to create one, and how to add elements to it, in the introductory tutorial. When we created a `Wallet` nested class in this tutorial, we incorporated an `ArrayList` to hold an `Adult`'s bills:

```
protected class Wallet {
        protected ArrayList bills = new ArrayList();

        protected void addBill(int aBill) {
                bills.add(new Integer(aBill));
        }

        protected int getMoneyTotal() {
                int total = 0;
                for (Iterator i = bills.iterator(); i.hasNext(); ) {
                        Integer wrappedBill = (Integer) i.next();
                        int bill = wrappedBill.intValue();
                        total += bill;
                }
                return total;
        }
}
```

The `getMoneyTotal()` method uses an *iterator* to march through the list of bills and total their values. An `Iterator` is similar to an `Enumeration` in older versions of the Java language. When you get an iterator on a collection (by calling `iterator()`), that iterator lets you *traverse* the collection using a couple of important methods, illustrated in the code above:

- `hasNext()` tells you if there is another element in the collection.
- `next()` gives you that next element.

As we discussed before, you must cast to the correct type when you extract elements from a collection using `next()`.

An `Iterator` gives us some additional capability, however. We can remove elements from our `ArrayList` by calling `remove()` (or `removeAll()`, or `clear()`), but we also can use the `Iterator` to do that. Let's add a simplistic method to `Adult` called `spendMoney()`:

```
public void spendMoney(int aBill) {
        this.wallet.removeBill(aBill);
}
```

This method calls `removeBill()` on `Wallet`:

```
protected void removeBill(int aBill) {
        Iterator iterator = bills.iterator();
        while (iterator.hasNext()) {
                Integer bill = (Integer) iterator.next();
                if (bill.intValue() == aBill)
                        iterator.remove();
        }
}
```

We get an `Iterator` on the `bills` `ArrayList`, and traverse the list to find a match on the bill value passed in (`aBill`). If we find one, we call `remove()` on the iterator to remove that bill. Simple, but not as simple as it could be. The following code does the same job and is much easier to read:

```
protected void removeBill(int aBill) {
        bills.remove(new Integer(aBill));
}
```

You probably won't call `remove()` on an `Iterator` very often, but it's nice to have that tool if you need it.

At the moment, we can remove only a single bill at a time from our `Wallet`. It would be nicer to use the power of a `List` to help us remove multiple bills at once, like this:

```
public void spendMoney(List bills) {
        this.wallet.removeBills(bills);
}
```

We need to add `removeBills()` to our `wallet` to make this work. Let's try this:

```
protected void removeBills(List billsToRemove) {
        this.bills.removeAll(bills);
}
```

This is the most straightforward implementation we can use. We call `removeAll()` on our `List` of bills, passing in a `Collection`. That method then removes all the elements from the list that are contained in the `Collection`. Try running this code:

```
List someBills = new ArrayList();
someBills.add(new Integer(1));
someBills.add(new Integer(2));

Adult anAdult = new Adult();
anAdult.acceptMoney(1);
anAdult.acceptMoney(1);
anAdult.acceptMoney(2);

List billsToRemove = new ArrayList();
billsToRemove.add(new Integer(1));
billsToRemove.add(new Integer(2));

anAdult.spendMoney(someBills);
System.out.println(anAdult.wallet.bills);
```

The results aren't what we want. We end up with no bills in our wallet. Why? Because `removeAll()` removes all matches. In other words, any and all matches in our wallet for something in the `List` we pass to the method are removed. The bills we passed in contained 1 and 2. Our wallet contains two 1s and a single 2. When `removeAll()` looks for matches on the 1 element, it finds two matches and removes them both. That's not what we want! We need to change our code in `removeBills()` to account for this:

```
protected void removeBills(List billsToRemove) {
        Iterator iterator = billsToRemove.iterator();
        while (iterator.hasNext()) {
                this.bills.remove(iterator.next());
        }
}
```

This code removes single matches only, rather than all matches. Remember to be careful with `removeAll()`.

Set implementations

There are two commonly used `Set` implementations:

- `HashSet`, which does not guarantee the order of iteration.
- `TreeSet`, which does.

The Java language documentation suggests that you'll end up using the first implementation most of the time. In general, if you need to make sure the elements in your `Set` are in a certain order when you traverse it with an iterator, use the second implementation. Otherwise, the first will do. The ordering of elements in a `TreeSet` (which, by the way, implements the `SortedSet` interface) is called the *natural ordering;* this means that, most of the time, you should be able to order the elements based on an `equals()` comparison.

Suppose that each `Adult` has a set of nicknames. We really don't care what order they're in, but duplicates wouldn't make much sense. We could use a `HashSet` to hold them. First, we add an instance variable:

```
protected Set nicknames = new HashSet();
```

Then we add a method to add nicknames to the `Set`:

```
public void addNickname(String aNickname) {
        nicknames.add(aNickname);
}
```

Now try running this code:

```
Adult anAdult = new Adult();
anAdult.addNickname("Bobby");
anAdult.addNickname("Bob");
anAdult.addNickname("Bobby");
System.out.println(anAdult.nicknames);
```

You'll see only a single occurrence of `Bobby` in the console.

Map implementations

A `Map` is a collection of key-value pairs. It cannot contain duplicate keys. Each key must map to a single value, but that value can be of any type. You can think of a map as a named `List`. Imagine a `List` where each element has a name you can use to extract that element directly. The key can be anything of type `Object`, as can the value. Once again, that means you can't store primitives directly in a `Map` (do you hate primitives yet?). Instead, you have to use the primitive wrapper classes to store the values.

Although this is a financially risky strategy, we're going to give each `Adult` a set of admittedly simplistic credit cards. Each will have a name and a balance (initially 0). First, we add an instance variable:

```
protected Map creditCards = new HashMap();
```

Then we add a method to add a credit card to the `Map`:

```
public void addCreditCard(String aCardName) {
        creditCards.put(aCardName, new Double(0));
}
```

The interface of the `Map` is different from the interfaces of other collections. You call `put()` with a key and a value to add an entry. You call `get()` with key to extract a value, which we'll do next in a method that will show us the balance for a card:

```
public double getBalanceFor(String cardName) {
        Double balance = (Double) creditCards.get(cardName);
        return balance.doubleValue();
}
```

All that's left is to add the `charge()` method, which allows us to add to our balance:

```
public void charge(String cardName, double amount) {
        Double balance = (Double) creditCards.get(cardName);
        double primitiveBalance = balance.doubleValue();
        primitiveBalance += amount;
        balance = new Double(primitiveBalance);

        creditCards.put(cardName, balance);
}
```

Now try running this code, which should show you `19.95` in the console:

```
Adult anAdult = new Adult();
anAdult.addCreditCard("Visa");
anAdult.addCreditCard("MasterCard");

anAdult.charge("Visa", 19.95);
adAdult.showBalanceFor("Visa");
```

A typical credit card has a name, an account number, a credit limit, and a balance. Each entry in a `Map` can have only a single key and a single value. Our simplistic credit cards fit, because they had only a name and a current balance. We could get more sophisticated by creating a class called `CreditCard`, with instance variables for all of the characteristics of a card, then store instances of that class as the values for entries in our `Map`.

There are some other interesting aspects of the `Map` interface to cover before we go (this is not an exhaustive list):

Method	Behavior
`containsKey()`	Answers whether or not the `Map` contains the given key.
`containsValue()`	Answers whether or not the `Map` contains the given value.
`keySet()`	Returns a `Set` of the keys.
`values()`	Returns a `Set` of the values.
`entrySet()`	Returns a `Set` of key-value pairs, defined as instances of `Map.Entrys`.
`remove()`	Lets you remove the value for a given key.
`isEmpty()`	Answers whether or not the `Map` is empty (that is, has no keys).

Some of these methods, such as `isEmpty()` are simply convenient, but some are vital. For example, the only way to iterate over the elements in a `Map` is via one of the related sets (of keys, of values, or of key-value pairs).

The Collections class

When you're using the Java Collections Framework, you need to be aware of what's available in the `Collections` class. This class contains a host of static methods that support collection manipulation. We won't cover them all here, because you can read the API yourself, but we'll cover two that show up very frequently in Java code:

- `copy()`
- `sort()`

The first method lets you copy the contents of one collection to another, like this:

```
List source = new ArrayList();
source.add("one");
source.add("two");
List target = new ArrayList();
target.add("three");
target.add("four");
```

```
Collections.copy(target, source);
System.out.println(target);
```

This code copies `source` into `target`. The target has to be the same size as the source, so you can't copy a `List` into an empty `List`.

The `sort()` method sorts elements into their natural order. All elements have to implement the `Comparable` interface so that they are *mutually comparable.* Built-in classes like `String` do already. So, for a set of strings, we can sort them in lexicographically ascending order with this code:

```
List strings = new ArrayList();
strings.add("one");
strings.add("two");
strings.add("three");
strings.add("four");

Collections.sort(strings);
System.out.println(strings);
```

You'll get `[four, one, three, two]` in the console. But how can you sort classes you create? We can do this for our `Adult`. First, we make the class mutually comparable:

```
public class Adult extends Person implements Comparable {
        ...
}
```

Then we override `compareTo()` to compare two `Adult` instances. We'll keep the comparison simplistic for our example, so it's less work:

```
public int compareTo(Object other) {
        final int LESS_THAN = -1;
        final int EQUAL = 0;
        final int GREATER_THAN = 1;

        Adult otherAdult = (Adult) other;
        if ( this == otherAdult ) return EQUAL;

        int comparison = this.firstname.compareTo(otherAdult.firstname);
        if (comparison != EQUAL) return comparison;

        comparison = this.lastname.compareTo(otherAdult.lastname);
        if (comparison != EQUAL) return comparison;

        return EQUAL;
}
```

Any number less than 0 means "less than," but -1 is a good value to use. Likewise, 1 is convenient for "greater than." As you can see, 0 means "equal." Comparing two objects in this way is obviously a manual process. You have to march through the instance variables and compare each. In this case, we compared the first name and last name, and effectively sorted by last name. But you should be able to see why our example is simplistic. Each `Adult` has more than just a first name and a last

name. If we wanted to do a deeper comparison, we would have to compare the `Wallets` of each `Adult` to see if they were equal, which probably means that we would have to implement `compareTo()` on `Wallet` and the rest. Also, to be correct about this, whenever you override `compareTo()`, you need to be sure the comparison is consistent with `equals()`. We don't implement `equals()`, so we didn't worry about being consistent with it, but we could have. In fact, I've seen code that includes a line like the following, before returning `EQUAL`:

```
assert this.equals(otherAdult) : "compareTo inconsistent with equals.";
```

The other approach to comparing objects is to extract the algorithm in `compareTo()` into a object of type `Comparator`, then call `Collections.sort()` with the collection to be sorted and the `Comparator`, like this:

```
public class AdultComparator implements Comparator {

        public int compare(Object object1, Object object2) {
                final int LESS_THAN = -1;
                final int EQUAL = 0;
                final int GREATER_THAN = 1;

                if ((object1 == null) ;amp;amp (object2 == null))
                        return EQUAL;
                if (object1 == null)
                        return LESS_THAN;
                if (object2 == null)
                        return GREATER_THAN;

                Adult adult1 = (Adult) object1;
                Adult adult2 = (Adult) object2;
                if (adult1 == adult2)
                        return EQUAL;

                int comparison = adult1.firstname.compareTo(adult2.firstname);
                if (comparison != EQUAL)
                        return comparison;

                comparison = adult1.lastname.compareTo(adult2.lastname);
                if (comparison != EQUAL)
                        return comparison;

                return EQUAL;
        }
}

public class CommunityApplication {

        public static void main(String[] args) {
                Adult adult1 = new Adult();
                adult1.setFirstname("Bob");
                adult1.setLastname("Smith");

                Adult adult2 = new Adult();
                adult2.setFirstname("Al");
                adult2.setLastname("Jones");

                List adults = new ArrayList();
                adults.add(adult1);
                adults.add(adult2);

                Collections.sort(adults, new AdultComparator());
                System.out.println(adults);
        }
}
```

You should see "Al Jones" and "Bob Smith," in that order, in your console window.

There are some good reasons to use the second approach. The technical ones are beyond the scope of this tutorial. From the perspective of good OOD, however, it might be a good idea to separate the comparison code into another object, rather than giving each `Adult` the ability to compare itself to another. However, since this is really what `equals()` does, even though the result is boolean, there are good arguments for either approach.

Using collections

When should you use a particular type of collection? That's a judgment call on your part, and it's because of calls like this that you hope to be paid very well as a programmer.

Despite what many professionals may believe, there are very few hard and fast rules about which classes to use in any given situation. In my personal experience, in the vast majority of the times I've used collections, an `ArrayList` or a `HashMap` (remember, a `Map` isn't a true collection) did the trick. That will probably be the same in your experience. Here are some rules of thumb, some more obvious than others:

- When you think you need a collection, start with a `List`, then let the code tell you if you need another type.
- If you need a unique grouping of things, use a `Set`.
- If the iteration order is important when traversing a collection, use the `Tree...` flavors of collections, where available.
- Avoid using `Vector`, unless you need its synchronization capability.
- Don't worry about optimization until (and unless) performance becomes an issue.

Collections are one of the most powerful aspects of the Java language. Don't be afraid to use them, but watch out for the "gotchas." For example, there is a convenient way to convert from an `Array` to an `ArrayList`:

```
Adult adult1 = new Adult();
Adult adult2 = new Adult();
Adult adult3 = new Adult();

List immutableList = Arrays.asList(new Object[] { adult1, adult2, adult3 });
immutableList.add(new Adult());
```

This code throws an `UnsupportedOperationException`, because the `List` returned by `Arrays.asList()` is immutable. You cannot add a new element to an immutable `List`. Keep your eyes open.

Section 7. Dates

Introduction

The Java language gives you lots of tools for handling dates. Some of them are much more frustrating than the tools available in other languages. That said, with the tools that the Java language provides, there is almost nothing you can't do to create dates and format them exactly how you want.

Creating dates

When the Java language was young, it contained a class called `Date` that was quite helpful for creating and manipulating dates. Unfortunately, that class did not support internationalization very well, so Sun added two classes that aimed to help the situation:

- `Calendar`
- `DateFormat`

We'll talk about `Calendar` first, and leave `DateFormat` for later.

Creating a `Date` is still relatively simple:

```
Date aDate = new Date(System.currentTimeMillis());
```

Or we could use this code:

```
Date aDate = new Date();
```

This will give us a `Date` representing the exact date and time right now, in the current *locale* format. Internationalization is beyond the scope of this tutorial, but for now, simply know that the `Date` you get back is consistent with the geography of your local machine.

Now that we have an instance, what can we do with it? Very little, directly. We can compare one `Date` with another to see if the first is `before()` or `after()` the second. We also can essentially reset it to a new instant in time by calling `setTime()` with a `long` representing the number of milliseconds since midnight on January 1, 1970 (which is what `System.currentTimeMillis()` returns). Beyond that, we're limited.

Calendars

The `Date` class is now more confusing than useful, because most of its date processing behavior is deprecated. You used to be able to get and set parts of the `Date` (such as the year, the month, etc.). Now we're left having to use both `Date` and `Calendar` to get the job done. Once we have a `Date` instance, we can use a `Calendar` to get and set parts of it. For example:

```
Date aDate = new Date(System.currentTimeMillis());
Calendar calendar = GregorianCalendar.getInstance();
calendar.setTime(aDate);
```

Here we create a `GregorianCalendar` and set its time to the `Date` we created before. We could have accomplished the same goal by calling a different method on our `Calendar`:

```
Calendar calendar = GregorianCalendar.getInstance();
calendar.setTimeInMillis(System.currentTimeMillis());
```

Armed with a `Calendar`, we can now access and manipulate components of our `Date`. Getting and setting parts of the `Date` is a simple process. We simply call appropriate getters and setters on our `Calendar`, like this:

```
calendar.set(Calendar.MONTH, Calendar.JULY);
calendar.set(Calendar.DAY_OF_MONTH, 15);
calendar.set(Calendar.YEAR, 1978);
calendar.set(Calendar.HOUR, 2);
calendar.set(Calendar.MINUTE, 15);
calendar.set(Calendar.SECOND, 37);
System.out.println(calendar.getTime());
```

This will print the formatted output string for July 15, 1978 at 02:15:37 a.m. (there also are helper methods on `Calendar` that allow us to set some or almost all of those components simultaneously). Here we called `set()`, which takes two parameters:

- The *field* (or component) of the `Date` we want to set.
- The value for that field.

We can reference the fields with named constants in the `Calendar` class itself. In some cases, there is more than one name for the same field, as with `Calendar.DAY_OF_MONTH`, which can also be referenced with `Calendar.DATE`. The values are straightforward, except perhaps the ones for `Calendar.MONTH` and the one for `Calendar.HOUR`. Months in Java language dates are zero-based (that is, January is 0), which really makes it wise to use the named constants to set them, and can make it frustrating to display dates correctly. The hours run from 0 to 24.

Once we have an established `Date`, we can extract parts of it:

```
System.out.println("The YEAR is: " + calendar.get(Calendar.YEAR));
System.out.println("The MONTH is: " + calendar.get(Calendar.MONTH));
System.out.println("The DAY is: " + calendar.get(Calendar.DATE));
System.out.println("The HOUR is: " + calendar.get(Calendar.HOUR));
System.out.println("The MINUTE is: " + calendar.get(Calendar.MINUTE));
System.out.println("The SECOND is: " + calendar.get(Calendar.SECOND));
System.out.println("The AM_PM indicator is: " + calendar.get(Calendar.AM_PM));
```

Built-in date formatting

You used to be able format dates with `Date`. Now you have to use several other classes:

- `DateFormat`
- `SimpleDateFormat`
- `DateFormatSymbols`

We won't cover all the complexities of date formatting here. You can explore these classes on your own. But we will talk about the basics of using these tools.

The `DateFormat` class lets us create a locale-specific formatter, like this:

```
DateFormat dateFormatter = DateFormat.getDateInstance(DateFormat.DEFAULT);
Date aDate = new Date();
String formattedDate = dateFormatter.format(today);
```

This code creates a formatted date string with the default format for this locale. On my machine, it looks something like this:

```
Nov 11, 2005
```

This is the default style, but it's not all that is available to us. We can use any of several predefined styles. We also can call `DateFormat.getTimeInstance()` to format a time, or `DateFormat.getDateTimeInstance()` to format both a date and a time. Here is the output of the various styles, all for the U.S. locale:

Style	Date	Time	Date/Time
`DEFAULT`	Nov 11, 2005	7:44:56 PM	Nov 11, 2005 7:44:56 PM
`SHORT`	11/11/05	7:44 PM	11/11/05 7:44 PM
`MEDIUM`	Nov 11, 2005	7:44:56 PM	Nov 11, 2005 7:44:56 PM
`LONG`	November 11, 2005	7:44:56 PM EST	November 11, 2005 7:44:56 PM EST
`FULL`	Thursday, November 11, 2005	7:44:56 PM EST	Thursday, November 11, 2005 7:44:56 PM EST

Customized formatting

These predefined formats are fine in most cases, but you can also use `SimpleDateFormat` to define your own formats. Using `SimpleDateFormat` is straightforward:

- Instantiate a `SimpleDateFormat` with a format pattern string (and a locale, if you wish).
- Call `format()` on it with a `Date`.

The result is a formatted date string. Here's an example:

```
Date aDate = new Date();
SimpleDateFormat formatter = new SimpleDateFormat("MM/dd/yyyy");
String formattedDate = formatter.format(today);
System.out.println(formattedDate);
```

When you run this code, you'll get something like the following (it will reflect the date that's current when you run the code, of course):

```
11/05/2005
```

The quoted string in the example above follows the pattern syntax rules for date formatting patterns. Java.sun.com has some excellent summaries of those rules (see Resources). Here are some helpful rules of thumb:

- You can specify patterns for dates and times.
- Some of the pattern syntax isn't intuitive (for example, `mm` defines a two-digit minute pattern; to get an abbreviated month, you use `MM`).
- You can include text literals in your patterns by placing them in single quotes (for example., using `"''on' MM/dd/yyyy"` above produces `on 11/05/2005`).
- The number of characters in a text component of a pattern dictates whether its abbreviated or long form will be used (`"MM"` yields `11`, but `"MMM"` yields `Nov`, and `"MMMM"` yields `November`).
- The number of characters in a numeric component of a pattern dictates the minimum number of digits.

If the standard symbols of `SimpleDateFormat` still don't meet your custom formatting needs, you can use `DateFormatSymbols` to customize the symbols for any component of a `Date` or time. For example, we could implement a unique set of abbreviations for months of the year, like this (using the same `SimpleDateFormat` as before):

```
DateFormatSymbols symbols = new DateFormatSymbols();
String[] oddMonthAbbreviations = new String[] {
        "Ja","Fe","Mh","Ap","My","Jn","Jy","Au","Se","Oc","No","De" };
symbols.setShortMonths(oddMonthAbbreviations);

formatter = new SimpleDateFormat("MMM dd, yyyy", symbols);
formattedDate = formatter.format(now);
System.out.println(formattedDate);
```

This code calls a different constructor on `SimpleDateFormat`, one that takes a pattern string and a `DateFormatSymbols` that defines the abbreviations used when a short month appears in a pattern. When we format the date with these symbols, the result looks something like this for the `Date` we saw above:

```
No 15, 2005
```

The customization capabilities of `SimpleDateFormat` and `DateFormatSymbols` should be enough to create any format you need.

Manipulating dates

You can go forward and backward in time by incrementing and decrementing dates, or parts of them. Two methods let you do this:

- `add()`
- `roll()`

The first lets you add some amount (or subtract by adding a negative amount) of time to a particular field of a `Date`. Doing that will adjust all other fields of the `Date` accordingly based on the addition to a particular field. For example, assume we begin with November 15, 2005 and increment the day field by 20. We could use code like this:

```
Calendar calendar = GregorianCalendar.getInstance();
calendar.set(Calendar.MONTH, 10);
calendar.set(Calendar.DAY_OF_MONTH, 15);
calendar.set(Calendar.YEAR, 2005);
formatter = new SimpleDateFormat("MMM dd, yyyy");
System.out.println("Before: " + formatter.format(calendar.getTime()));

calendar.add(Calendar.DAY_OF_MONTH, 20);
System.out.println("After: " + formatter.format(calendar.getTime()));
```

The result looks something like this:

```
Before: Nov 15, 2005
After: Dec 05, 2005
```

Rather simple. But what does it mean to *roll* a `Date`? It means you are incrementing or decrementing a particular date/time field by a given amount, without affecting

other fields. For example, we could roll our date from November to December like this:

```
Calendar calendar = GregorianCalendar.getInstance();
calendar.set(Calendar.MONTH, 10);
calendar.set(Calendar.DAY_OF_MONTH, 15);
calendar.set(Calendar.YEAR, 2005);
formatter = new SimpleDateFormat("MMM dd, yyyy");
System.out.println("Before: " + formatter.format(calendar.getTime()));
calendar.roll(Calendar.MONTH, true);
System.out.println("After: " + formatter.format(calendar.getTime()));
```

Notice that the month is rolled up (or incremented) by 1. There are two forms of `roll()`:

- `roll(int field, boolean up)`
- `roll(int field, int amount)`

We used the first. To decrement a field using this form, you pass `false` as the second argument. The second form of the method lets you specify the increment or decrement amount. If a rolling action would create an invalid date value (for example, 09/31/2005), these methods adjust the other fields accordingly, based on valid maximum and minimum values for dates, hours, etc. You can roll forward with positive values, and backward with negative ones.

It's fine to try to predict what your rolling actions will do, and you can certainly do so, but more often than not, trial and error is the best method. Sometimes you'll guess right, but sometimes you'll have to experiment to see what produces the correct results.

Using Dates

Everybody has a birthday. Let's add one to our `Person` class. First, we add an instance variable to `Person`:

```
protected Date birthdate = new Date();
```

Next, we add accessors for the variable:

```
public Date getBirthdate() {
        return birthdate;
}
public void setBirthdate(Date birthday) {
        this.birthdate = birthday;
}
```

Next, we'll remove the `age` instance variable, because we'll now calculate it. We also remove the `setAge()` accessor, because `age` will now be a derived value. We replace the body of `getAge()` with the following code:

```
public int getAge() {
        Calendar calendar = GregorianCalendar.getInstance();
        calendar.setTime(new Date());
        int currentYear = calendar.get(Calendar.YEAR);

        calendar.setTime(birthdate);
        int birthYear = calendar.get(Calendar.YEAR);

        return currentYear - birthYear;
}
```

In this method, we now calculate the value of `age` based on the year of the `Person`'s `birthdate` and the year of today's date.

Now we can try it out, with this code:

```
Calendar calendar = GregorianCalendar.getInstance();
calendar.setTime(new Date());
calendar.set(Calendar.YEAR, 1971);
calendar.set(Calendar.MONTH, 2);
calendar.set(Calendar.DAY_OF_MONTH, 23);

Adult anAdult = new Adult();
anAdult.setBirthdate(calendar.getTime());
System.out.println(anAdult);
```

We set `birthdate` on an `Adult` to March 23, 1971. If we run this code in January 2005, we should get this output:

```
An Adult with:
Age: 33
Name: firstname lastname
Gender: MALE
Progress: 0
```

There are a few other housekeeping details that I leave as an exercise for you:

- Update `compareTo()` on `Adult` to reflect the presence of a new instance variable.
- Had we implemented it, we would have to update `equals()` on `Adult` to reflect the presence of a new instance variable.
- Had we implemented `equals()`, we would have implemented `hashCode()` as well, and we would have to update `hashCode()` to reflect the presence of a new instance variable.

Section 8. I/O

Introduction

The data that a Java language program uses has to come from somewhere. More often than not, it comes from some external data source. There are many different kinds of data sources, including databases, direct byte transfer over a socket, and files. The Java language gives you lots of tools with which you can get information from external sources. These tools are mostly in the `java.io` package.

Of all the data sources available, files are the most common, and often the most convenient. Knowing how to use the available Java language APIs to interact with files is a fundamental programmer skill.

In general, the Java language gives you a wrapper class (`File`) for a file in your OS. To read that file, you have to use *streams* that parse the incoming bytes into Java language types. In this section, we will talk about all of the objects you'll typically use to read files.

Files

The `File` class defines a resource on your filesystem. It's a pain in the neck, especially for testing, but it's the reality Java programmers have to deal with.

Here's how you instantiate a `File`:

```
File aFile = new File("temp.txt");
```

This creates a `File` with the *path* `temp.txt` in the current directory. We can create a `File` with any path string we want, as long as it's valid. Note that having this `File` object doesn't mean that the underlying file actually exists on the filesystem in the expected location. Our object merely represents an actual file that may or may not be there. If the underlying file doesn't exist, we won't know there's a problem until we try to read from or write to it. That's a bit unpleasant, but it makes sense. For example, we can ask our `File` if it exists:

```
aFile.exists();
```

If it doesn't, we can create it:

```
aFile.createNewFile();
```

Using other methods on `File`, we also can delete files, create directories, determine whether a file system resource is a directory or a file, etc. The real action occurs, though, when we write to and read from the file. To do that, we need to understand a little bit about streams.

Streams

We can access files on the filesystem using *streams.* At the lowest level, streams allow a program to receive bytes from a source and/or to send output to a destination. Some streams handle all kinds of 16-bit characters (types `Reader` and `Writer`). Others handle only 8-bit bytes (types `InputStream` and `OutputStream`). Within these hierarchies are several flavors of streams (all found in the `java.io` package). At the highest level of abstraction, there are *character streams* and *byte streams.*

Byte streams read (`InputStream` and subclasses) and write (`OutputStream` and subclasses) 8-bit bytes. In other words, byte streams could be considered a more raw type of stream. As a result, it's easy to understand why the Java.sun.com tutorial on essential Java language classes (see Resources) says that byte streams are typically used for binary data, such as images. Here's a selected list of byte streams:

Streams	Usage
`FileInputStream` `FileOutputStream`	Read bytes from a file, and write bytes to a file.
`ByteArrayInputStream` `ByteArrayOutputStream`	Read bytes from an in-memory array, and write bytes to an in-memory array.

Character streams read (`Reader` and its subclasses) and write (`Writer` and its subclasses) 16-bit characters. Subclasses either read to or write from data *sinks,* or process bytes in transit. Here's a selected list of character streams:

Streams	Usage
`StringReader` `StringWriter`	These streams read and write characters to and from `Strings` in memory.

`InputStreamReader`
`InputStreamWriter` (and subclasses
`FileReader`
`FileWriter`)
	Bridge between byte streams and character streams. The `Reader` flavors read bytes from a byte stream and convert them to characters. The `Writer` flavors convert characters to bytes to put them on byte streams.

`BufferedReader` and `BufferedWriter`	Buffer data while reading or writing another stream, which allows read and write operations to be more efficient. You *wrap* another stream in a buffered stream.

Streams are a large topic, and we can't cover them in their entirety here. Instead, we'll focus on the recommended streams for reading and writing files. In most cases, these will be the character streams, but we'll use both character and byte streams to illustrate the possibilities.

Reading and writing files

There are several ways to read from and write to a `File`. Arguably the simplest approach goes like this:

- Create a `FileOutputStream` on the `File` to write to it.
- Create a `FileInputStream` on the `File` to read from it.
- Call `read()` to read from the `File` and `write()` to write to it.
- Close the streams, cleaning up if necessary.

The code might look something like this:

```
try {
        File source = new File("input.txt");
        File sink = new File("output.txt");

        FileInputStream in = new FileInputStream(source);
        FileOutputStream out = new FileOutputStream(sink);
        int c;

        while ((c = in.read()) != -1)
           out.write(c);

        in.close();
        out.close();
} catch (Exception e) {
        e.printStackTrace();
}
```

Here we create two `File` objects: a `FileInputStream` to read from the source file, and a `FileOutputStream` to write to the output `File`. (Note: This example was adapted from the Java.sun.com `CopyBytes.java` example; see Resources.) We then read in each byte of the input and write it to the output. Once done, we close the streams. It may seem wise to put the calls to `close()` in a `finally` block. However, the Java language compiler will still require that you catch the various exceptions that occur, which means yet another `catch` in your `finally`. Is it worth it? Maybe.

So now we have a basic approach for reading and writing. But the wiser choice, and in some respects the easier choice, is to use some other streams, which we'll discuss in the next panel.

Buffering streams

There are several ways to read from and write to a `File`, but the typical, and most convenient, approach goes like this:

- Create a `FileWriter` on the `File`.
- Wrap the `FileWriter` in a `BufferedWriter`.
- Call `write()` on the `BufferedWriter` as often as necessary to write the contents of the `File`, typically ending each line with a line termination

character (that is, \n).

- Call `flush()` on the `BufferedWriter` to empty it.
- Close the `BufferedWriter`, cleaning up if necessary.

The code might look something like this:

```
try {
        FileWriter writer = new FileWriter(aFile);
        BufferedWriter buffered = new BufferedWriter(writer);
        buffered.write("A line of text.\n");
        buffered.flush();
} catch (IOException e1) {
        e1.printStackTrace();
}
```

Here we create a `FileWriter` on `aFile`, then we wrap it in a `BufferedWriter`. Buffered writing is more efficient than simply writing bytes out one at a time. When we're done writing each line (which we manually terminate with \n), we call `flush()` on the `BufferedWriter`. If we didn't, we wouldn't see any data in the file, which defeats the purpose of all of this file writing effort.

Once we have data in the file, we can read it with some similarly straightforward code:

```
String line = null;
StringBuffer lines = new StringBuffer();
try {
        FileReader reader = new FileReader(aFile);
        BufferedReader bufferedReader = new BufferedReader(reader);
        while ( (line = bufferedReader.readLine()) != null) {
                lines.append(line);
                lines.append("\n");
        }
} catch (IOException e1) {
        e1.printStackTrace();
}
System.out.println(lines.toString());
```

We create a `FileReader`, then wrap it in a `BufferedReader`. That allows us to use the convenient `readLine()` method. We read each line until there are none left, appending each line to the end of our `StringBuffer`. When reading from a file, an `IOException` could occur, so we surround all of our file-reading logic with a `try`/`catch` block.

Section 9. Wrap-up

Summary

We've covered a significant portion of the Java language in the "Introduction to Java programming" tutorial (see Resources) and this tutorial, but the Java language is huge and a single, monolithic tutorial (or even several smaller ones) can't encompass it all. Here's a sampling of some areas we did not explore at all:

Topic	Brief description
Threads	Having a program that does only one thing at a time can be useful, but most sophisticated Java language programs have multiple *threads* of execution that run simultaneously. For example, printing or searching tasks might run in the background. The class `Thread` and related classes in `java.lang` can give you powerful and flexible threading capability in your programs. developerWorks hosts many good resources on threading in your Java code, but a good place to start would be with the "Introduction to Java threads" and "Concurrency in JDK 5.0" tutorials and this series of articles by Brian Goetz.
Reflection	One of the more powerful aspects of the Java language (and often one of the most mind-bending) is *reflection,* or the ability to see information about your code itself. The package `java.lang.reflect` includes classes like `Class` and `Method` that let you inspect the structure of your code. For example, you can find a method starting with `get`, then call it by calling `invoke()` on the `Method` object -- very powerful. Part 2 of the "Java programming dynamics" series by Dennis M. Sosnoski talks about using reflection.
NIO	Since JDK 1.4, the Java language has incorporated some more sophisticated I/O capabilities, based on a completely new API called *new I/O*, or *NIO* for short. The principal difference is that traditional Java language I/O is based on streams (as we've already discussed), while NIO is based on a concepts called *block I/O, channels,* and *buffers.* This *block I/O* tends to be more efficient than sending single bytes one at a time through a stream. The drawback is that NIO can be conceptually difficult. "Getting started with new I/O (NIO)" by Greg Travis is an excellent tutorial on the subject.
Sockets	Your Java language programs can communicate with virtually any program on an IP-enabled device. All you need to do is open a *socket* connection to an IP address and port on that device. The Java language Sockets API facilitates that. See the "Java sockets 101" tutorial by Roy Miller and Adam Williams for an introduction to the API. The "Using JSSE for secure socket communication" tutorial by Greg Travis shows you how to make your socket communication secure.

Swing	The Java language includes extensive support for GUI development, in the form of Swing. The Swing set of APIs includes classes for widgets and other elements to create full-featured user interfaces. There are several important developerWorks resources related to Swing, but a good place to start is the introductory article "The Java 2 user interface" by Matt Chapman. The Magic with Merlin column by John Zukowski addresses more recent changes and updates to Swing. John also hosts the Client-side Java programming discussion forum, so you can get assistance with Swing programming there, too. The "Migrate your Swing application to SWT" tutorial discusses how to migrate from Swing to IBM's SWT, a lighter-weight but still powerful alternative.
JNI	When your Java program needs to communicate with another program written, for example, in C, the Java language gives you a way to do that: The *Java Native Interface* (JNI). This API lets you translate Java language method calls into calls to C functions (for interacting with the OS and such). The developerWorks "Java programming with JNI" tutorial discusses the mechanics of JNI with Java, C, and C++ code.
RMI	The Java language Remote Method Invocation (RMI) API allows a Java language program in one process or on one machine to access another Java language program running in another process and/or on another machine. In other words, RMI supports *distributed method calls* between programs running in different Java VMs. Brad Rubin's "Java distributed objects: Using RMI and CORBA" tutorial gives a solid introduction to RMI, and discusses RMI and CORBA together. You should also check out this article by Edward Harned to learn why RMI is *not* an off-the-shelf application server.
Security	The Java language includes sophisticated security APIs to support authentication and authorization. How can you be sure that those who use your program have permission to do? How can you protect information from prying eyes? The security APIs can help. developerWorks offers an abundance of content related to Java security. Here are just a few: Java security discussion forum hosted by John Peck, a veteran Java programmer and security expert; "Java authorization internals" by Abhijit Belapurkar; and "Java security, Part 1: Crypto basics" and "Java security, Part 2: Authentication and authorization" both by Brad Rubin .

Resources

Learn

- Take Roy Miller's "Introduction to Java programming" for the basics of Java prgramming. (developerworks, November 2004)
- The java.sun.com Web site has links to all things Java programming. You can find every "official" Java language resource you need there, including the language specification and API documentation. You can also can find links to excellent tutorials on various aspects of the Java language, beyond the fundamental tutorial.
- Go to Sun's Java documentation page, for a link to API documentation for each of the SDK versions.
- Check out John Zukowski's excellent article on regex with the Java language here. This is just one article in his Magic with Merlin column.
- The Sun Java tutorial is an excellent resource. It's a gentle introduction to the language, but also covers many of the topics addressed in this tutorial. If nothing else, it's a good resource for examples and for links to other tutorials that go into more detail about various aspects of the language.
- The java.sun.com Web site has some excellent summaries of the date pattern rules here and more on the `CopyBytes.java` example here.
- The developerWorks 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

- Download the intermediate.jar that accompanies this tutorial.
- You can download Eclipse from the Eclipse Web site. Or, make it easy on yourself and download Eclipse bundled with the latest IBM Java runtime (available for Linux and Windows).

Discuss

- Participate in developerWorks blogs and get involved in the developerWorks community.

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 Java 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.