IBM

developerWorks®

Introduction to Java programming, Part 2: Constructs for real-world applications

More-advanced Java language features

J Steven Perry

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

Continue familiarizing yourself with object-oriented programming on the Java™ platform. This second half of the *Introduction to Java programming* tutorial introduces the more-sophisticated syntax and libraries you need to develop complex, real-world Java applications. Topics covered include exception handling, inheritance and abstraction, regular expressions, generics, Java I/O, and Java serialization.

View more content in this series

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

About this tutorial

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

This second half of the *Introduction to Java programming* tutorial introduces capabilities of the Java language that are more sophisticated than those covered in Part 1.

Objectives

The Java language is mature and sophisticated enough to help you accomplish nearly any programming task. This tutorial introduces you to features of the Java language that you need to handle complex programming scenarios, including:

- Exception handling
- Inheritance and abstraction
- Interfaces

- Nested classes
- Regular expressions
- Generics
- `enum` types
- I/O
- Serialization

Prerequisites

The content of this tutorial is geared toward programmers new to the Java language who are unfamiliar with its more-sophisticated features. The tutorial assumes that you have worked through "*Introduction to Java programming*, Part 1: Java language basics" to:

- Gain an understanding of the basics of OOP on the Java platform
- Set up the development environment for the tutorial examples
- Begin the programming project that you continue developing in Part 2

System requirements

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

- JDK 8 from Oracle
- Eclipse IDE for Java Developers

Download and installation instructions for both are included in Part 1.

The recommended system configuration is:

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

Next steps with objects

Part 1 of this tutorial left off with a `Person` class that was reasonably useful, but not as useful as it could be. Here, you begin learning about techniques to enhance a class such as `Person`, starting with the following techniques:

- Overloading methods
- Overriding methods
- Comparing one object with another
- Using class variables and class methods

You'll start enhancing `Person` by *overloading* one of its methods.

Overloading methods

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

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

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

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

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

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

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

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

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

Overriding methods

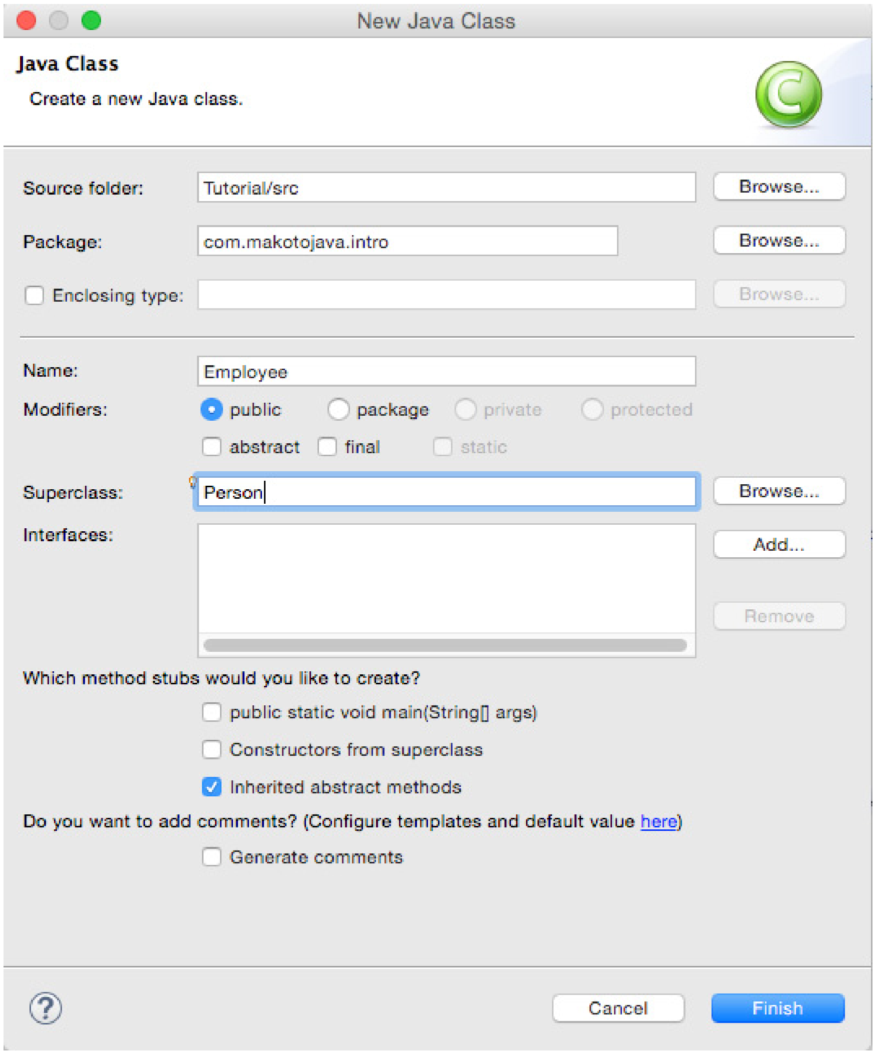
When a subclass provides its own implementation of a method that's defined on one of its parent classes, that's called *method overriding*. To see how method overriding is useful, you need to do some work on an `Employee` class. Once you have that class set up, I show you where method overriding comes in handy.

Employee: A subclass of Person

Recall from Part 1 of this tutorial that an `Employee` class might be a subclass (or *child*) of `Person` that has additional attributes such as taxpayer identification number, employee number, hire date, and salary.

To declare the `Employee` class, right-click the `com.makotojava.intro` package in Eclipse. Click **New > Class...** To open the New Java Class dialog box, shown in Figure 1.

Figure 1. New Java Class dialog box



New Java Class

Java Class

Create a new Java class.

Source folder:	Tutorial/src	Browse...
Package:	com.makotojava.intro	Browse...
☐ Enclosing type:		Browse...
Name:	Employee	
Modifiers:	● public ○ package ○ private ○ protected	
	☐ abstract ☐ final ☐ static	
Superclass:	Person	Browse...
Interfaces:		Add... / Remove

Which method stubs would you like to create?

☐ public static void main(String[] args)

☐ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

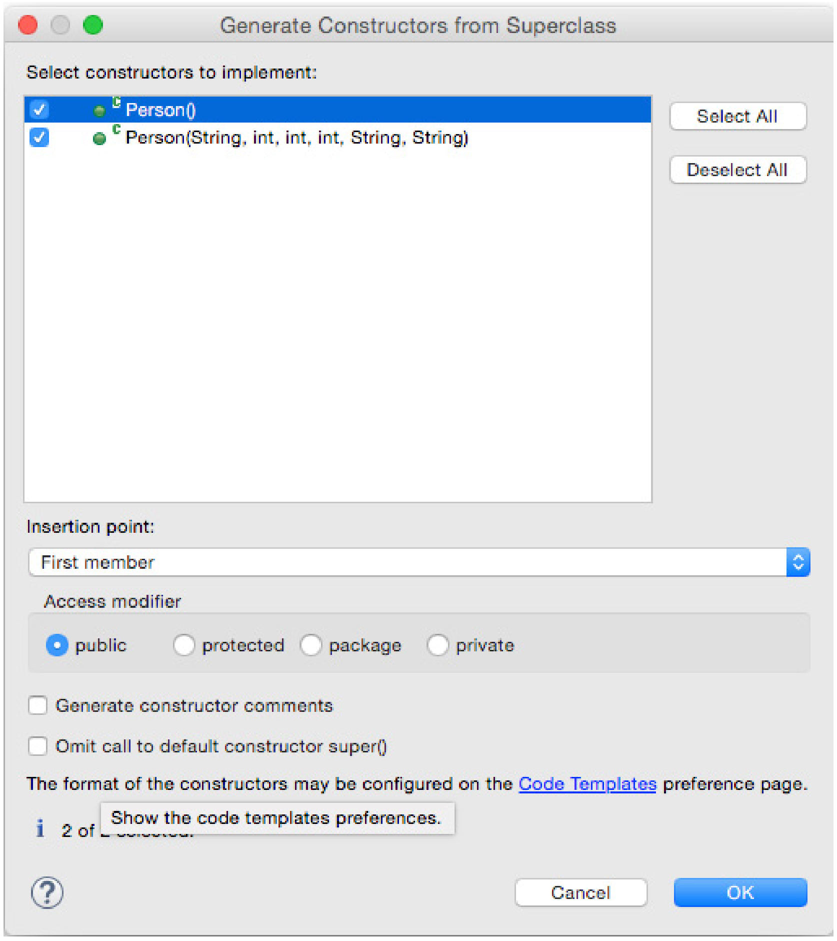
☐ Generate comments

Enter `Employee` as the name of the class and `Person` as its superclass. Click **Finish**, and you can see the `Employee` class code in an edit window.

You don't explicitly need to declare a constructor, but go ahead and implement both constructors anyway. With the `Employee` class edit window having the focus, go to **Source > Generate**

Constructors from Superclass.... In the Generate Constructors from Superclass dialog box (see Figure 2), select both constructors and click **OK**.

Figure 2. Generate Constructors from Superclass dialog box


Generate Constructors from Superclass

Select constructors to implement:

[✓] Person()
[✓] Person(String, int, int, int, String, String)

Insertion point: First member

Access modifier: [● public] ○ protected ○ package ○ private

[ ] Generate constructor comments
[ ] Omit call to default constructor super()

The format of the constructors may be configured on the Code Templates preference page.

Show the code templates preferences.

2 of 2 selected.

Eclipse generates the constructors for you. You now have an `Employee` class like the one in Listing 2.

Listing 2. The `Employee` class

```
package com.makotojava.intro;

public class Employee extends Person {

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

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

}
```

Employee as a child of Person

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

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

```
package com.makotojava.intro;

import java.math.BigDecimal;

public class Employee extends Person {

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

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

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

Don't forget to generate getters and setters for the new attributes, as you did for in the "Your first Java class" section in Part 1.

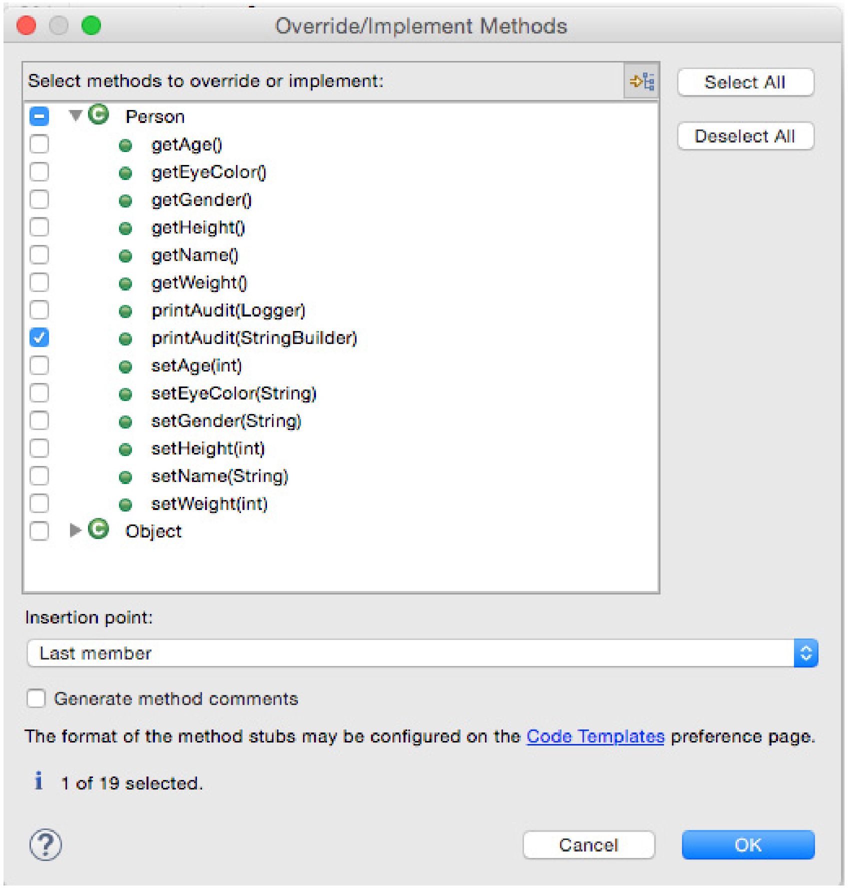
Overriding the printAudit() method

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

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

With `Employee` open in the editor window or selected in the Project Explorer view, go to **Source > Override/Implement Methods...**. In the Override/Implement Methods dialog box, shown in Figure 3, select the `StringBuilder` overload of `printAudit()` and click **OK**.

Figure 3. Override/Implement Methods dialog box


Override/Implement Methods

Select methods to override or implement:

- [-] Person
  - [ ] getAge()
  - [ ] getEyeColor()
  - [ ] getGender()
  - [ ] getHeight()
  - [ ] getName()
  - [ ] getWeight()
  - [ ] printAudit(Logger)
  - [✓] printAudit(StringBuilder)
  - [ ] setAge(int)
  - [ ] setEyeColor(String)
  - [ ] setGender(String)
  - [ ] setHeight(int)
  - [ ] setName(String)
  - [ ] setWeight(int)
- [ ] Object

Insertion point: Last member

[ ] Generate method comments

The format of the method stubs may be configured on the Code Templates preference page.

1 of 19 selected.

Eclipse generates the method stub for you, and then you can fill in the rest, like so:

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

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

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

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

Comparing objects

The Java language provides two ways to compare objects:

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

Comparing objects with ==

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

Suppose you generate a JUnit test for `Employee` (which you saw how to do in the "Your first Java class" section in Part 1. The JUnit test is shown in Listing 4.

Listing 4. Comparing objects with `==`

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

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

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

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

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

Comparing objects with equals()

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

You call `equals()` like this:

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

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

Listing 5. Comparing objects with `equals()`

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

Running the Listing 5 code produces this output:

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

A note about comparing Integers

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

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

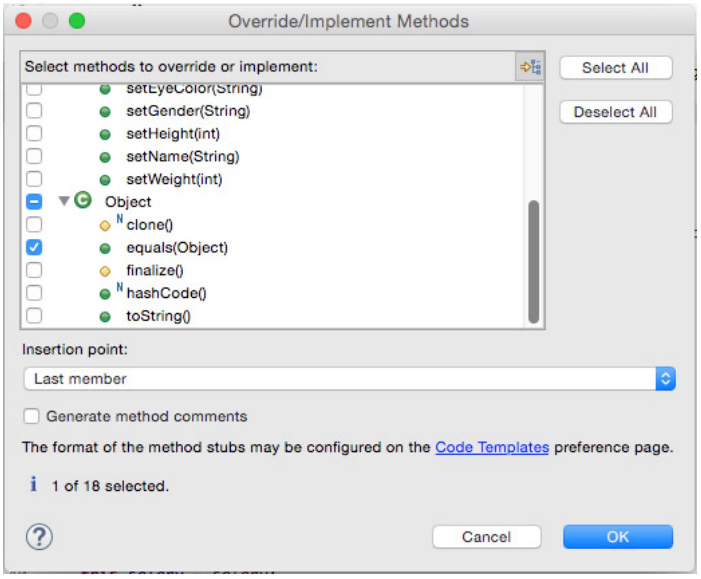
For `Employee`, you didn't override `equals()`, so the default behavior (of using `==`) returns what you'd expect, because `employee1` and `employee2` refer to different objects.

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

Overriding equals()

You can define what `equals()` means to your application's objects by overriding the default behavior of `Object.equals()`— and you can do this in Eclipse. With `Employee` having the focus in the IDE's source window, select **Source > Override/Implement Methods** to open the dialog box shown in Figure 4.

Figure 4. Override/Implement Methods dialog box


Override/Implement Methods

Select methods to override or implement:

- [ ] setEyeColor(String)
- [ ] setGender(String)
- [ ] setHeight(int)
- [ ] setName(String)
- [ ] setWeight(int)
- Object
  - [ ] clone()
  - [✓] equals(Object)
  - [ ] finalize()
  - [ ] hashCode()
  - [ ] toString()

Insertion point: Last member

[ ] Generate method comments

The format of the method stubs may be configured on the Code Templates preference page.

1 of 18 selected.

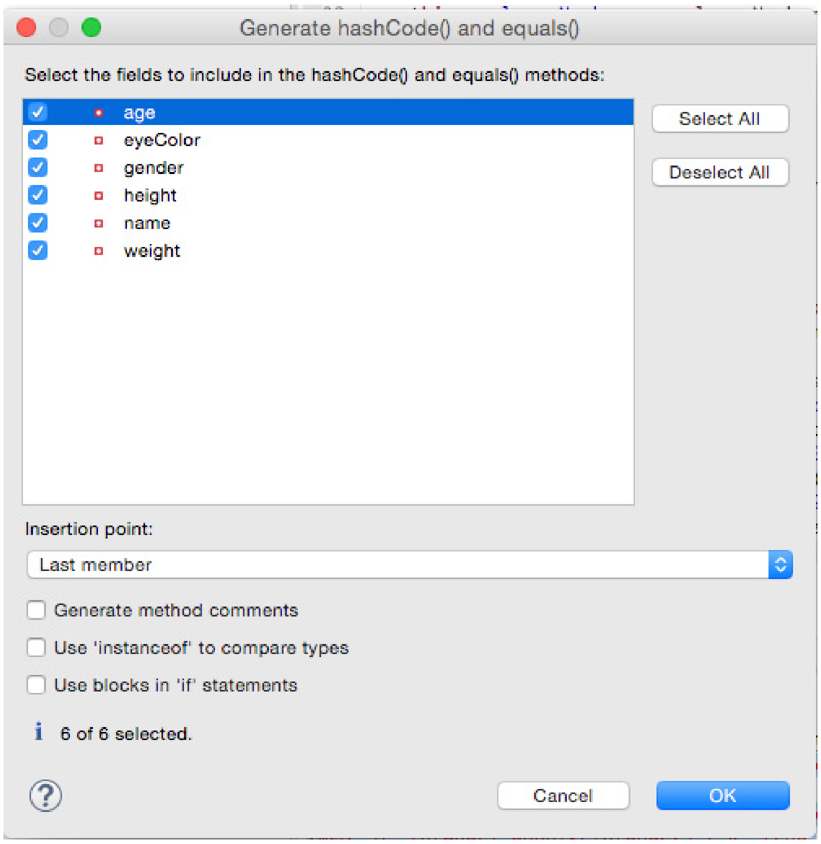
You want to implement the `Object.equals()` superclass method. So, find `Object` in the list of methods to override or implement, select the `equals(Object)` method, and click **OK**. Eclipse generates the correct code and places it in your source file.

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

Autogenerating equals()

Eclipse can generate an `equals()` method for you based on the instance variables (attributes) that you define for a class. Because `Employee` is a subclass of `Person`, you first generate `equals()` for `Person`. In the Eclipse Project Explorer view, right-click `Person` and choose **Generate hashCode() and equals()**. In the dialog box that opens (see Figure 5), click **Select All** to include all of the attributes in the `hashCode()` and `equals()` methods, and click **OK**.

Figure 5. Generate hashCode() and equals() dialog box


Generate hashCode() and equals()

Select the fields to include in the hashCode() and equals() methods:

✓ age
✓ eyeColor
✓ gender
✓ height
✓ name
✓ weight

Insertion point: Last member

☐ Generate method comments
☐ Use 'instanceof' to compare types
☐ Use blocks in 'if' statements

6 of 6 selected.

Eclipse generates an `equals()` method that looks like the one in Listing 6.

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

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

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

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

Exercises

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

Exercise 1: Generate an equals() for Employee

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

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

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

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

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

Exercise 2: Override toString()

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

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

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

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

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

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

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

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

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

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

Class members

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

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

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

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

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

Class variables

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

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

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

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

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

Listing 8. Using class variables

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

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

Declaring constants

Typically, constants are:

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

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

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

Class methods

If you've been following along since Part 1, you've already called the static `Logger.getLogger()` method several times — whenever you retrieved a `Logger` instance to write output to the console. Notice, though, that to do so you didn't need an instance of `Logger`. Instead, you referenced the `Logger` class, which is the syntax for making a *class method* call. As with class variables, the `static` keyword identifies `Logger` (in this example) as a class method. Class methods are also sometimes called *static methods* for this reason.

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

Listing 9. Creating a class (or static) method

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

  //. . .
  public static Logger getLogger() {
    return logger;
  }

}
```

Two important things are happening in Listing 9:

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

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

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

Exceptions

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

Exception-handling basics

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

To get started with exception handling, take a look at the code in Listing 10.

Listing 10. Do you see the error?

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

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

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

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

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

Using try, catch, and finally

Listing 11 shows the buggy code from Listing 10 cleaned up with the standard code blocks for exception handling: `try`, `catch`, and `finally`.

Listing 11. Catching an exception

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

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

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

In Listing 11, the program recovers from the error and then prints out the exception's message:

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

The exception hierarchy

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

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

When a program causes an exception, you say that it *throws* the exception. A checked exception is declared to the compiler by any method with the `throws` keyword in its method signature. Next is a comma-separated list of exceptions that the method could potentially throw during its execution.

If your code calls a method that specifies that it throws one or more types of exceptions, you must handle it somehow, or add a `throws` to your method signature to pass that exception type along.

When an exception occurs, the Java runtime searches for an exception handler somewhere up the stack. If it doesn't find one by the time it reaches the top of the stack, it halts the program abruptly, as you saw in Listing 10.

Multiple catch blocks

You can have multiple `catch` blocks, but they must be structured in a particular way. If any exceptions are subclasses of other exceptions, the child classes are placed ahead of the parent classes in the order of the `catch` blocks. Listing 12 shows an example of different exception types structured in their correct hierarchical sequence.

Listing 12. Exception hierarchy example

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

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

try-with-resources blocks

The code in Listing 12 must declare a variable to hold the `bufferedReader` reference, and then in the `finally` must close the `BufferedReader`.

Alternative, more-compact syntax (available as of JDK 7) automatically closes resources when the `try` block goes out of scope. Listing 13 shows this newer syntax.

Listing 13. Resource-management syntax

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

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

Building Java applications

In this section, you continue building up `Person` as a Java application. Along the way, you can get a better idea of how an object, or collection of objects, evolves into an application.

The application entry point

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

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

Creating a driver class

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

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

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

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

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

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

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

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

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

Inheritance

You've encountered examples of inheritance a few times already in this tutorial. This section reviews some of Part 1's material on inheritance and explains in more detail how inheritance works — including the inheritance hierarchy, constructors and inheritance, and inheritance abstraction.

How inheritance works

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

For example, suppose you have a `Person` class that looks like the one in Listing 14.

Listing 14. Public `Person` class

```
public class Person {

  public static final String STATE_DELIMITER = "~";

  public Person() {
    // Default constructor
  }

  public enum Gender {
    MALE,
    FEMALE,
    UNKNOWN
  }

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


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

The `Person` class in Listing 14 implicitly inherits from `Object`. Because inheriting from `Object` is assumed for every class, you don't need to type `extends Object` for every class you define. But what does it mean to say that a class inherits from its superclass? It simply means that `Person` has access to the exposed variables and methods in its superclasses. In this case, `Person` can see and use `Object`'s public and protected methods and variables.

Defining a class hierarchy

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

```
public class Employee extends Person {

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

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

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

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

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

Single versus multiple inheritance

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

Constructors and inheritance

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

Constructor basics

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

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

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

```
public class Person {
}
```

Invoking a superclass constructor

To invoke a superclass constructor other than the default constructor, you must do so explicitly. For example, suppose `Person` has a constructor that takes just the name of the `Person` object being created. From `Employee`'s default constructor, you could invoke the `Person` constructor shown in Listing 15.

Listing 15. Initializing a new `Employee`

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

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

You would probably never want to initialize a new `Employee` object this way, however. Until you get more comfortable with object-oriented concepts, and Java syntax in general, it's a good idea to implement superclass constructors in subclasses only if you are sure you'll need them. Listing 16 defines a constructor in `Employee` that looks like the one in `Person` so that they match up. This approach is much less confusing from a maintenance standpoint.

Listing 16. Invoking a superclass

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

Declaring a constructor

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

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

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

No-arg constructors

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

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

  public Employee() {
  }
}
```

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

How constructors invoke constructors

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

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

}
```

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

Constructor access levels

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

Table 1. Constructor access rules

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

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

Inheritance and abstraction

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

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

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

Abstract classes and methods

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

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

Using abstraction

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

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

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

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

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

The power of abstraction

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

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

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

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

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

When (not) to abstract: Two rules

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

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

Assignments: Classes

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

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

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

Now consider this example:

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

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

Interfaces

In this section, you begin learning about interfaces and start using them in your Java code.

Interfaces: What are they good for?

As you know from the previous section, abstract methods, by design, specify a *contract*— through the method name, parameter(s), and return type — but provide no reusable code. Abstract methods — defined on abstract classes — are useful when the way the behavior is implemented is likely to change from the way it's implemented in one subclass of the abstract class to another.

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

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

Defining an interface

The syntax for defining an interface is straightforward:

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

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

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

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

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

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

```
public interface BonusEligible {
}
```

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

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

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

Implementing interfaces

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

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

Suppose you implement the `StockOptionRecipient` interface on the `Manager` class, as shown in Listing 17:

Listing 17. Implementing an interface

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

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

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

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

Generating interfaces in Eclipse

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

Using interfaces

An interface defines a new *reference* data type, which you can use to refer to an interface anywhere you would refer to a class. This ability includes when you declare a reference variable, or cast from one type to another, as shown in Listing 18.

Listing 18. Assigning a new `Manager` instance to a `StockOptionEligible` reference

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

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

Assignments: Interfaces

You can assign a reference from a class that implements an interface to a variable of an interface type, but certain rules apply. From Listing 18, you can see that assigning a `Manager` instance to a `StockOptionEligible` variable reference is valid. The reason is that the `Manager` class implements that interface. However, the following assignment would not be valid:

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

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

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

Nested classes

In this section, you learn about nested classes and where and how to use them.

Where to use nested classes

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

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

  }
}
```

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

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

Scope in nested classes

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

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

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

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

Public nested classes

Because it's `private`, only `Manager` can create an instance of `DirectReports`. But suppose you wanted to give an external entity the ability to create instances of `DirectReports`. In this case, it seems like you could give the `DirectReports` class `public` scope, and then any external code could create `DirectReports` instances, as shown in Listing 19.

Listing 19. Creating `DirectReports` instances: First attempt

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

The code in Listing 19 doesn't work, and you're probably wondering why. The problem (and also its solution) lies with the way `DirectReports` is defined within `Manager`, and with the rules of scope.

The rules of scope, revisited

If you had a member variable of `Manager`, you'd expect the compiler to require you to have a reference to a `Manager` object before you could reference it, right? Well, the same applies to `DirectReports`, at least as it's defined in Listing 19.

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

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

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

Static inner classes

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

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

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

Anonymous inner classes

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

Listing 20 builds Listing 17, adding a default method for handling `Employee` types that are not `StockOptionEligible`. The listing starts with a method in `HumanResourcesApplication` to process the stock options, followed by a JUnit test to drive the method.

Listing 20. Handling `Employee` types that are not `StockOptionEligible`

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

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

In the Listing 20 example, I provide implementations of two interfaces that use anonymous inner classes. First are two separate implementations of `StockOptionEligible`— one for `Employee`s and one for `Person`s (to obey the interface). Then comes an implementation of `StockOptionProcessingCallback` that's used to handle processing of stock options for the `Manager` instances.

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

Regular expressions

A *regular expression* is essentially a pattern to describe a set of strings that share that pattern. If you're a Perl programmer, you should feel right at home with the regular expression (regex) pattern syntax in the Java language. If you're not used to regular expressions syntax, however, it can look weird. This section gets you started with using regular expressions in your Java programs.

The Regular Expressions API

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

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

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

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

- `Pattern` describes a string pattern.
- `Matcher` tests a string to see if it matches the pattern.

- `PatternSyntaxException` tells you that something wasn't acceptable about the pattern that you tried to define.

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

Regex pattern syntax

A *regex pattern* describes the structure of the string that the expression tries to find in an input string. The pattern syntax can look strange to the uninitiated, but once you understand it, you'll find it easier to decipher. Table 2 lists some of the most common regex constructs that you use in pattern strings.

Table 2. Common regex constructs

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

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

Pattern matching

Armed with the pattern syntax in Table 2, you can work through the simple example in Listing 21, using the classes in the Java Regular Expressions API.

Listing 21. Pattern matching with regex

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

First, Listing 21 creates a `Pattern` class by calling `compile()`— a static method on `Pattern`— with a string literal representing the pattern you want to match. That literal uses the regex pattern syntax. In this example, the English translation of the pattern is:

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

Methods for matching

Next, Listing 21 calls `matcher()` on `Pattern`. That call creates a `Matcher` instance. The `Matcher` then searches the string you passed in for matches against the pattern string you used when you created the `Pattern`.

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

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

Listing 21 finds a single match starting at 0 and ending at 7. Thus, the call to `matches()` returns `true`, the call to `start()` returns `0`, and the call to `end()` returns `8`.

lookingAt() versus matches()

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

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

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

Complex patterns in regex

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

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

```
MyWikiWord
```

Suppose a user inputs the following string:

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

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

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

And here's code to search for wiki words:

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

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

Replacing strings

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

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

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

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

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

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

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

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

Matching and manipulating groups

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

In each pattern, you typically create groups by enclosing parts of the pattern in parentheses. Groups are numbered from left to right, starting with 1 (group 0 represents the entire match). The code in Listing 22 replaces each wiki word with a string that "wraps" the word.

Listing 22. Matching groups

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

Run the Listing 22 code, and you get the following console output:

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

Listing 22 references the entire match by including `$0` in the replacement string. Any portion of a replacement string of the form `$`*`int`* refers to the group identified by the integer (so `$1` refers to group 1, and so on). In other words, `$0` is equivalent to `matcher.group(0);`.

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

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

And you'd get the same result:

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

Generics

The introduction of generics in JDK 5.0 (released in 2004) marked a huge leap forward for the Java language. If you've used C++ templates, you'll find that generics in the Java language are similar but not exactly the same. If you haven't used C++ templates, don't worry: This section offers a high-level introduction to generics in the Java language.

What are generics?

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

Generics in action

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

Listing 23 shows how `java.util.ArrayList` is instantiated.

Listing 23. Instantiating `ArrayList`

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

As you can see, the `ArrayList` is heterogeneous: It contains two `String` types and one `Integer` type. Before JDK 5.0, the Java language had nothing to constrain this behavior, which caused many coding mistakes. In Listing 23, for example, everything is looking good so far. But what about accessing the elements of the `ArrayList`, which Listing 24 tries to do?

Listing 24. Attempt to access elements in `ArrayList`

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

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

With generics, you can specify the type of item that went in the `ArrayList`. Listing 25 shows how, and what happens if you try and add an object of the wrong type (line 3).

Listing 25. A second attempt, using generics

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

Iterating with generics

Generics enhance the Java language with special syntax for dealing with entities, such as `List`s, that you commonly want to step through element by element. If you want to iterate through `ArrayList`, for instance, you could rewrite the code from Listing 25 like so:

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

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

Parameterized classes

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

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

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

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

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

A parameterized List

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

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

The `E`, which stands for Element, is the "thing" I mentioned earlier. The `concreteListClass` is the class from the JDK that you're instantiating. The JDK includes several `List<E>` implementations, but you use `ArrayList<E>`. Another way you might see a generic class discussed is `Class<T>`, where `T` stands for Type. When you see `E` in Java code, it's usually referring to a collection of some kind. And when you see `T`, it's denoting a parameterized class.

So, to create an `ArrayList` of, say, `java.lang.Integer`, you do this:

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

SimpleList: A parameterized class

Now suppose you want to create your own parameterized class called `SimpleList`, with three methods:

- `add()` adds an element to the end of the `SimpleList`.
- `size()` returns the current number of elements in the `SimpleList`.
- `clear()` completely clears the contents of the `SimpleList`.

Listing 26 shows the syntax to parameterize `SimpleList`.

Listing 26. Parameterizing `SimpleList`

```
package com.makotojava.intro;
import java.util.ArrayList;
import java.util.List;
public class SimpleList<E> {
  private List<E> backingStore;
  public SimpleList() {
    backingStore = new ArrayList<E>();
  }
  public E add(E e) {
    if (backingStore.add(e))
    return e;
    else
    return null;
  }
  public int size() {
    return backingStore.size();
  }
  public void clear() {
    backingStore.clear();
  }
}
```

`SimpleList` can be parameterized with any `Object` subclass. To create and use a `SimpleList` of, say, `java.math.BigDecimal` objects, you might do this:

```
package com.makotojava.intro;
import java.math.BigDecimal;
import java.util.logging.Logger;
import org.junit.Test;
public class SimpleListTest {
  @Test
  public void testAdd() {
    Logger log = Logger.getLogger(SimpleListTest.class.getName());

    SimpleList<BigDecimal> sl = new SimpleList<>();
    sl.add(BigDecimal.ONE);
    log.info("SimpleList size is : " + sl.size());
    sl.add(BigDecimal.ZERO);
    log.info("SimpleList size is : " + sl.size());
    sl.clear();
    log.info("SimpleList size is : " + sl.size());
  }
}
```

And you would get this output:

```
Sep 20, 2015 10:24:33 AM com.makotojava.intro.SimpleListTest testAdd
INFO: SimpleList size is: 1 Sep 20, 2015 10:24:33 AM com.makotojava.intro.SimpleListTest testAdd
INFO: SimpleList size is: 2 Sep 20,
2015 10:24:33 AM com.makotojava.intro.SimpleListTest testAdd
INFO: SimpleList size is: 0
```

Parameterized methods

At times, you might not want to parameterize your entire class, but only one or two methods. In this case, you create a *generic method*. Consider the example in Listing 27, where the method `formatArray` is used to create a string representation of the contents of an array.

Listing 27. A generic method

```
public class MyClass {
// Other possible stuff... ignore...
  public <E> String formatArray(E[] arrayToFormat) {
    StringBuilder sb = new StringBuilder();

    int index = 0;
    for (E element : arrayToFormat) {
      sb.append("Element ");
      sb.append(index++);
      sb.append(" => ");
      sb.append(element);
      sb.append('\n');
    }

    return sb.toString();
  }
// More possible stuff... ignore...
}
```

Rather than parameterize `MyClass`, you make generic just the one method you want to use create a consistent string representation that works for any element type.

In practice, you'll find yourself using parameterized classes and interfaces far more often then methods, but now you know that the capability is available if you need it.

enum types

In JDK 5.0, a new data type was added to the Java language, called `enum` (not to be confused with `java.util.Enumeration`). The `enum` type represents a set of constant objects that are all related to a particular concept, each of which represents a different constant value in that set. Before `enum` was introduced into the language, you would have defined a set of constant values for a concept (say, gender) like so:

```
public class Person {
  public static final String MALE = "male";
  public static final String FEMALE = "female";
  public static final String OTHER = "other";
}
```

Any code that needed to reference that constant value would have been written something like this:

```
public void myMethod() {
  //. . .
  String genderMale = Person.MALE;
  //. . .
}
```

Defining constants with enum

Using the `enum` type makes defining constants much more formal — and more powerful. Here's the `enum` definition for `Gender`:

```
public enum Gender {
  MALE,
  FEMALE,
  OTHER
}
```

This example only scratches the surface of what you can do with `enum`s. In fact, `enum`s are much like classes, so they can have constructors, attributes, and methods:

```
package com.makotojava.intro;

public enum Gender {
  MALE("male"),
  FEMALE("female"),
  OTHER("other");

  private String displayName;
  private Gender(String displayName) {
    this.displayName = displayName;
  }

  public String getDisplayName() {
    return this.displayName;
  }
}
```

One difference between a class and an `enum` is that an `enum`'s constructor must be declared `private`, and it cannot extend (or inherit from) other `enum`s. However, an `enum`**can** implement an interface.

An enum implementing an interface

Suppose you define an interface, `Displayable`:

```
package com.makotojava.intro;
public interface Displayable {
  public String getDisplayName();
}
```

Your `Gender enum` could implement this interface (and any other `enum` that needed to produce a friendly display name), like so:

```
package com.makotojava.intro;

public enum Gender implements Displayable {
  MALE("male"),
  FEMALE("female"),
  OTHER("other");

  private String displayName;
  private Gender(String displayName) {
    this.displayName = displayName;
  }
  @Override
  public String getDisplayName() {
    return this.displayName;
  }
}
```

I/O

This section is an overview of the `java.io` package. You learn to use some of its tools to collect and manipulate data from various sources.

Working with external data

More often than not, the data you use in your Java programs comes from an external data source, such as a database, direct byte transfer over a socket, or file storage. Most of the Java tools for collecting and manipulating external data are in the `java.io` package.

Files

Of all the data sources available to your Java applications, files are the most common and often the most convenient. If you want to read a file in your application, you must use *streams* that parse its incoming bytes into Java language types.

`java.io.File` is a class that defines a resource on your file system and represents that resource in an abstract way. Creating a `File` object is easy:

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

File f2 = new File("/home/steve/testFile.txt");
```

The `File` constructor takes the name of the file it creates. The first call creates a file called temp.txt in the specified directory. The second call creates a file in a specific location on my Linux system. You can pass any `String` to the constructor of `File`, provided that it's a valid file name for your operating system, whether or not the file that it references even exists.

This code asks the newly created `File` object if the file exists:

```
File f2 = new File("/home/steve/testFile.txt");
if (f2.exists()) {
  // File exists. Process it...
} else {
  // File doesn't exist. Create it...
  f2.createNewFile();
}
```

`java.io.File` has some other handy methods that you can use to:

- Delete files
- Create directories (by passing a directory name as the argument to `File`'s constructor)
- Determine if a resource is a file, directory, or symbolic link
- More

The main action of Java I/O is in writing to and reading from data sources, which is where streams come in.

Using streams in Java I/O

You can access files on the file system by using streams. At the lowest level, streams enable a program to receive bytes from a source or to send output to a destination. Some streams handle all kinds of 16-bit characters (`Reader` and `Writer` types). Others handle only 8-bit bytes (`InputStream` and `OutputStream` types). Within these hierarchies are several flavors of streams, all found in the `java.io` package.

Byte streams read (`InputStream` and subclasses) and write (`OutputStream` and subclasses) 8-bit bytes. In other words, a byte stream can be considered a more raw type of stream. Here's a summary of two common byte streams and their usage:

- `FileInputStream` / `FileOutputStream`: Reads bytes from a file, writes bytes to a file
- `ByteArrayInputStream` / `ByteArrayOutputStream`: Reads bytes from an in-memory array, writes bytes to an in-memory array

Character streams

Character streams read (`Reader` and its subclasses) and write (`Writer` and its subclasses) 16-bit characters. Here's a selected listing of character streams and their usage:

- `StringReader` / `StringWriter`: Read and write characters to and from `String`s in memory.
- `InputStreamReader` / `InputStreamWriter` (and subclasses `FileReader` / `FileWriter`): Act as a bridge between byte streams and character streams. The `Reader` flavors read bytes from a byte stream and convert them to characters. The `Writer` flavors convert characters to bytes to put them on byte streams.
- BufferedReader / BufferedWriter: Buffer data while reading or writing another stream, making read and write operations more efficient.

Rather than try to cover streams in their entirety, I'll focus here on the recommended streams for reading and writing files. In most cases, these are character streams.

Reading from a File

You can read from a `File` in several ways. Arguably the simplest approach is to:

1. Create an `InputStreamReader` on the `File` you want to read from.
2. Call `read()` to read one character at a time until you reach the end of the file.

Listing 28 is an example in reading from a `File`:

Listing 28. Reading from a `File`

```
public List<Employee> readFromDisk(String filename) {
  final String METHOD_NAME = "readFromDisk(String filename)";
  List<Employee> ret = new ArrayList<>();
  File file = new File(filename);
  try (InputStreamReader reader = new InputStreamReader(new FileInputStream(file))) {
    StringBuilder sb = new StringBuilder();
    int numberOfEmployees = 0;
    int character = reader.read();
    while (character != -1) {
        sb.append((char)character);
        character = reader.read();
    }
    log.info("Read file: \n" + sb.toString());
    int index = 0;
    while (index < sb.length()-1) {
      StringBuilder line = new StringBuilder();
      while ((char)sb.charAt(index) != '\n') {
        line.append(sb.charAt(index++));
      }
      StringTokenizer strtok = new StringTokenizer(line.toString(), Person.STATE_DELIMITER);
      Employee employee = new Employee();
      employee.setState(strtok);
      log.info("Read Employee: " + employee.toString());
      ret.add(employee);
      numberOfEmployees++;
      index++;
    }
    log.info("Read " + numberOfEmployees + " employees from disk.");
  } catch (FileNotFoundException e) {
    log.logp(Level.SEVERE, SOURCE_CLASS, METHOD_NAME, "Cannot find file " +
       file.getName() + ", message = " + e.getLocalizedMessage(), e);
  } catch (IOException e) {
    log.logp(Level.SEVERE, SOURCE_CLASS, METHOD_NAME, "IOException occurred,
       message = " + e.getLocalizedMessage(), e);
  }
  return ret;
}
```

Writing to a File

As with reading from a `File`, you have several ways to write to a `File`. Once again, I go with the simplest approach:

1. Create a `FileOutputStream` on the `File` you want to write to.
2. Call `write()` to write the character sequence.

Listing 29 is an example of writing to a `File`:

Listing 29. Writing to a `File`

```
public boolean saveToDisk(String filename, List<Employee> employees) {
  final String METHOD_NAME = "saveToDisk(String filename, List<Employee> employees)";

  boolean ret = false;
  File file = new File(filename);
  try (OutputStreamWriter writer = new OutputStreamWriter(new FileOutputStream(file))) {
    log.info("Writing " + employees.size() + " employees to disk (as String)...");
    for (Employee employee : employees) {
      writer.write(employee.getState()+"\n");
    }
    ret = true;
    log.info("Done.");
  } catch (FileNotFoundException e) {
    log.logp(Level.SEVERE, SOURCE_CLASS, METHOD_NAME, "Cannot find file " +
       file.getName() + ", message = " + e.getLocalizedMessage(), e);
  } catch (IOException e) {
    log.logp(Level.SEVERE, SOURCE_CLASS, METHOD_NAME, "IOException occurred,
       message = " + e.getLocalizedMessage(), e);
  }
  return ret;
}
```

Buffering streams

Reading and writing character streams one character at a time isn't efficient, so in most cases you probably want to use buffered I/O instead. To read from a file using buffered I/O, the code looks just like Listing 28, except that you wrap the `InputStreamReader` in a `BufferedReader`, as shown in Listing 30.

Listing 30. Reading from a `File` with buffered I/O

```
public List<Employee> readFromDiskBuffered(String filename) {
  final String METHOD_NAME = "readFromDisk(String filename)";
  List<Employee> ret = new ArrayList<>();
  File file = new File(filename);
  try (BufferedReader reader = new BufferedReader(new InputStreamReader(new FileInputStream(file)))) {
    String line = reader.readLine();
    int numberOfEmployees = 0;
    while (line != null) {
      StringTokenizer strtok = new StringTokenizer(line, Person.STATE_DELIMITER);
      Employee employee = new Employee();
      employee.setState(strtok);
      log.info("Read Employee: " + employee.toString());
      ret.add(employee);
      numberOfEmployees++;
      // Read next line

      line = reader.readLine();
    }
    log.info("Read " + numberOfEmployees + " employees from disk.");
  } catch (FileNotFoundException e) {
    log.logp(Level.SEVERE, SOURCE_CLASS, METHOD_NAME, "Cannot find file " +
       file.getName() + ", message = " + e.getLocalizedMessage(), e);
  } catch (IOException e) {
    log.logp(Level.SEVERE, SOURCE_CLASS, METHOD_NAME, "IOException occurred,
       message = " + e.getLocalizedMessage(), e);
  }
  return ret;
}
```

Writing to a file using buffered I/O is the same: You wrap the `OutputStreamWriter` in a `BufferedWriter`, as shown in Listing 31.

Listing 31. Writing to a `File` with buffered I/O

```
public boolean saveToDiskBuffered(String filename, List<Employee> employees) {
  final String METHOD_NAME = "saveToDisk(String filename, List<Employee> employees)";

  boolean ret = false;
  File file = new File(filename);
  try (BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(new FileOutputStream(file)))) {
    log.info("Writing " + employees.size() + " employees to disk (as String)...");
    for (Employee employee : employees) {
      writer.write(employee.getState()+"\n");
    }
    ret = true;
    log.info("Done.");
  } catch (FileNotFoundException e) {
    log.logp(Level.SEVERE, SOURCE_CLASS, METHOD_NAME, "Cannot find file " +
       file.getName() + ", message = " + e.getLocalizedMessage(), e);
  } catch (IOException e) {
    log.logp(Level.SEVERE, SOURCE_CLASS, METHOD_NAME, "IOException occurred,
       message = " + e.getLocalizedMessage(), e);
  }
  return ret;
}
```

Java serialization

Java serialization is another one the Java platform's essential libraries. Serialization is primarily used for object persistence and object remoting, two use cases where you need to be able to take a snapshot of the state of an object and then reconstitute later. This section gives you a taste of the Java Serialization API and shows how to use it in your programs.

What is object serialization?

Serialization is a process whereby the state of an object and its metadata (such as the object's class name and the names of its attributes) are stored in a special binary format. Putting the object into this format —*serializing* it — preserves all the information necessary to reconstitute (or *deserialize*) the object whenever you need to do so.

The two primary use cases for object serialization are:

- *Object persistence*— storing the object's state in a permanent persistence mechanism such as a database
- *Object remoting*— sending the object to another computer or system

java.io.Serializable

The first step in making serialization work is to enable your objects to use the mechanism. Every object you want to be serializable must implement an interface called `java.io.Serializable`:

```
import java.io.Serializable;
public class Person implements Serializable {
  // etc...
}
```

In this example, the `Serializable` interface marks the objects of the `Person` class — and every subclass of `Person`— to the runtime as `serializable`.

Any attributes of an object that are not serializable cause the Java runtime to throw a `NotSerializableException` if it tries to serialize your object. You can manage this behavior by using the `transient` keyword to tell the runtime not to try to serialize certain attributes. In that case, you are responsible for making sure that the attributes are restored (if necessary) so that your object functions correctly.

Serializing an object

Now, try an example that combines what you learned about Java I/O with what you're learning now about serialization.

Suppose you create and populate a `List` of `Employee` objects and then want to serialize that `List` to an `OutputStream`, in this case to a file. That process is shown in Listing 32.

Listing 32. Serializing an object

```
public class HumanResourcesApplication {
  private static final Logger log = Logger.getLogger(HumanResourcesApplication.class.getName());
  private static final String SOURCE_CLASS = HumanResourcesApplication.class.getName();

  public static List<Employee> createEmployees() {
    List<Employee> ret = new ArrayList<Employee>();
    Employee e = new Employee("Jon Smith", 45, 175, 75, "BLUE", Gender.MALE,
       "123-45-9999", "0001", BigDecimal.valueOf(100000.0));
    ret.add(e);
    //
    e = new Employee("Jon Jones", 40, 185, 85, "BROWN", Gender.MALE, "223-45-9999",
       "0002", BigDecimal.valueOf(110000.0));
    ret.add(e);
    //
    e = new Employee("Mary Smith", 35, 155, 55, "GREEN", Gender.FEMALE, "323-45-9999",
       "0003", BigDecimal.valueOf(120000.0));
    ret.add(e);
    //
    e = new Employee("Chris Johnson", 38, 165, 65, "HAZEL", Gender.UNKNOWN,
       "423-45-9999", "0004", BigDecimal.valueOf(90000.0));
    ret.add(e);
    // Return list of Employees
    return ret;
  }

  public boolean serializeToDisk(String filename, List<Employee> employees) {
    final String METHOD_NAME = "serializeToDisk(String filename, List<Employee> employees)";

    boolean ret = false;// default: failed
    File file = new File(filename);
    try (ObjectOutputStream outputStream = new ObjectOutputStream(new FileOutputStream(file))) {
      log.info("Writing " + employees.size() + " employees to disk (using Serializable)...");
      outputStream.writeObject(employees);
      ret = true;
      log.info("Done.");

    } catch (IOException e) {
      log.logp(Level.SEVERE, SOURCE_CLASS, METHOD_NAME, "Cannot find file " +
       file.getName() + ", message = " + e.getLocalizedMessage(), e);
    }
    return ret;
  }
```

The first step is to create the objects, which is done in `createEmployees()` using the specialized constructor of `Employee` to set some attribute values. Next, you create an `OutputStream`— in this case, a `FileOutputStream`— and then call `writeObject()` on that stream. `writeObject()` is a method that uses Java serialization to serialize an object to the stream.

In this example, you are storing the `List` object (and its contained `Employee` objects) in a file, but this same technique is used for any type of serialization.

To drive the code in Listing 32, you could use a JUnit test, as shown here:

```
public class HumanResourcesApplicationTest {

  private HumanResourcesApplication classUnderTest;
  private List<Employee> testData;

  @Before
  public void setUp() {
    classUnderTest = new HumanResourcesApplication();
    testData = HumanResourcesApplication.createEmployees();
  }
  @Test
  public void testSerializeToDisk() {
    String filename = "employees-Junit-" + System.currentTimeMillis() + ".ser";
    boolean status = classUnderTest.serializeToDisk(filename, testData);
    assertTrue(status);
  }

}
```

Deserializing an object

The whole point of serializing an object is to be able to reconstitute, or deserialize, it. Listing 33 reads the file you've just serialized and deserializes its contents, thereby restoring the state of the `List` of `Employee` objects.

Listing 33. Deserializing objects

```
public class HumanResourcesApplication {

  private static final Logger log = Logger.getLogger(HumanResourcesApplication.class.getName());
  private static final String SOURCE_CLASS = HumanResourcesApplication.class.getName();

  @SuppressWarnings("unchecked")
  public List<Employee> deserializeFromDisk(String filename) {
    final String METHOD_NAME = "deserializeFromDisk(String filename)";

    List<Employee> ret = new ArrayList<>();
    File file = new File(filename);
    int numberOfEmployees = 0;
    try (ObjectInputStream inputStream = new ObjectInputStream(new FileInputStream(file))) {
      List<Employee> employees = (List<Employee>)inputStream.readObject();
      log.info("Deserialized List says it contains " + employees.size() +
         " objects...");
      for (Employee employee : employees) {
        log.info("Read Employee: " + employee.toString());
        numberOfEmployees++;
      }
      ret = employees;
      log.info("Read " + numberOfEmployees + " employees from disk.");
    } catch (FileNotFoundException e) {
      log.logp(Level.SEVERE, SOURCE_CLASS, METHOD_NAME, "Cannot find file " +
         file.getName() + ", message = " + e.getLocalizedMessage(), e);
    } catch (IOException e) {
      log.logp(Level.SEVERE, SOURCE_CLASS, METHOD_NAME, "IOException occurred,
       message = " + e.getLocalizedMessage(), e);
    } catch (ClassNotFoundException e) {
      log.logp(Level.SEVERE, SOURCE_CLASS, METHOD_NAME, "ClassNotFoundException,
         message = " + e.getLocalizedMessage(), e);
    }
    return ret;
  }

}
```

Again, to drive the code in Listing 33, you could use a JUnit test like this one:

```
public class HumanResourcesApplicationTest {

  private HumanResourcesApplication classUnderTest;

  private List<Employee> testData;

  @Before
  public void setUp() {
    classUnderTest = new HumanResourcesApplication();
  }

  @Test
  public void testDeserializeFromDisk() {
    String filename = "employees-Junit-" + System.currentTimeMillis() + ".ser";
    int expectedNumberOfObjects = testData.size();
    classUnderTest.serializeToDisk(filename, testData);
    List<Employee> employees = classUnderTest.deserializeFromDisk(filename);
    assertEquals(expectedNumberOfObjects, employees.size());
  }

}
```

For most application purposes, marking your objects as `serializable` is all you ever need to worry about when it comes to serialization. When you do need to serialize and deserialize your objects explicitly, you can use the technique shown in Listing 32 and Listing 33. But as your application objects evolve, and you add and remove attributes to and from them, serialization takes on a new layer of complexity.

serialVersionUID

In the early days of middleware and remote object communication, developers were largely responsible for controlling the "wire format" of their objects, which caused no end of headaches as technology began to evolve.

Suppose you added an attribute to an object, recompiled it, and redistributed the code to every computer in an application cluster. The object would be stored on a computer with one version of the serialization code but accessed by other computers that might have a different version of the code. When those computers tried to deserialize the object, bad things often happened.

Java serialization metadata — the information included in the binary serialization format — is sophisticated and solves many of the problems that plagued early middleware developers. But it can't solve every problem.

Java serialization uses a property called `serialVersionUID` to help you deal with different versions of objects in a serialization scenario. You don't need to declare this property on your objects; by default, the Java platform uses an algorithm that computes a value for it based on your class's attributes, its class name, and position in the local galactic cluster. Most of the time, that algorithm works fine. But if you add or remove an attribute, that dynamically generated value changes, and the Java runtime throws an `InvalidClassException`.

To avoid this outcome, get in the habit of explicitly declaring a `serialVersionUID`:

```
import java.io.Serializable;
  public class Person implements Serializable {
  private static final long serialVersionUID = 20100515;
  // etc...
  }
```

I recommend using a scheme of some kind for your `serialVersionUID` version number (I've used the current date in the preceding example). And you should declare `serialVersionUID` as `private static final` and of type `long`.

You might be wondering when to change this property. The short answer is that you should change it whenever you make an incompatible change to the class, which usually means you've added or removed an attribute. If you have one version of the object on one computer that has the attribute added or removed, and the object gets remoted to a computer with a version of the object where the attribute is either missing or expected, things can get weird. This is where the Java platform's built-in `serialVersionUID` check comes in handy.

As a rule of thumb, any time you add or remove features (meaning attributes or any other instance-level state variables) of a class, change its `serialVersionUID`. Better to get a `java.io.InvalidClassException` on the other end of the wire than an application bug that's caused by an incompatible class change.

Conclusion to Part 2

The *Introduction to Java programming* tutorial has covered a significant portion of the Java language, but the language is huge. A single tutorial can't possibly encompass it all.

As you continue learning about the Java language and platform, you probably will want to do further study into topics such as regular expressions, generics, and Java serialization. Eventually, you might also want to explore topics not covered in this introductory tutorial, such as concurrency and persistence.

Related topics

- developerWorks Java development
- IBM developer kits
- 5 things you didn't know about Java Object Serialization
- IBM Code: Java journeys