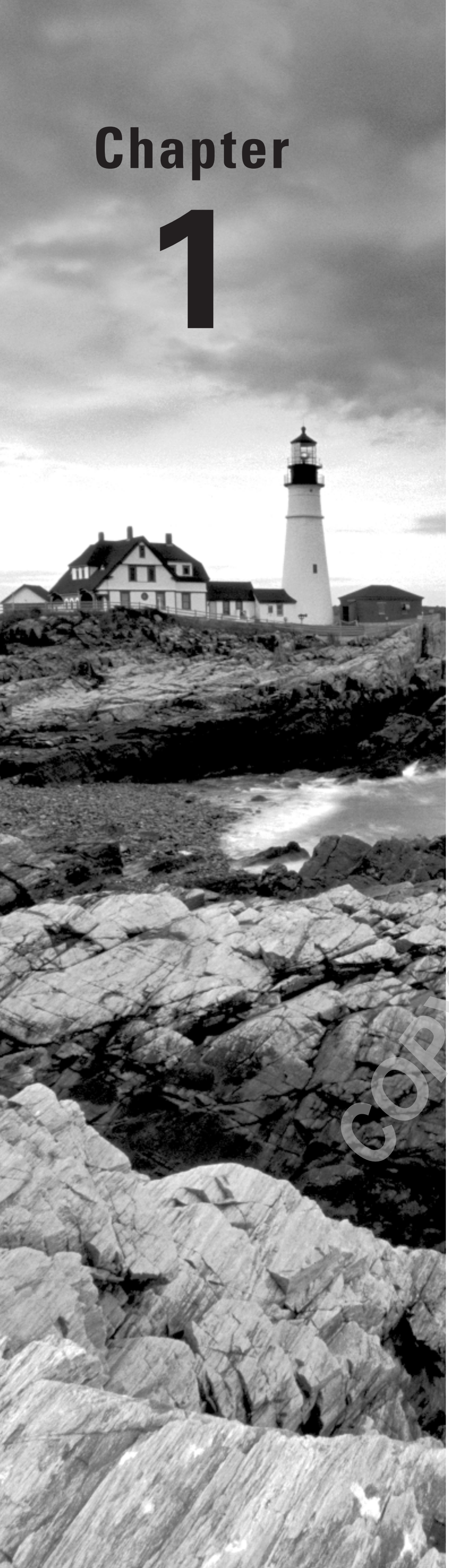

Chapter

1

Advanced Class Design

THE OCP EXAM TOPICS COVERED IN THIS CHAPTER INCLUDE THE FOLLOWING:

✓ **Java Class Design**

- Implement inheritance including visibility modifiers and composition
- Implement polymorphism
- Override hashCode, equals, and toString methods from Object class
- Develop code that uses the static keyword on initialize blocks, variables, methods, and classes

✓ **Advanced Java Class Design**

- Develop code that uses abstract classes and methods
- Develop code that uses final keyword
- Create inner classes including static inner class, local class, nested class, and anonymous inner class
- Use enumerated types including methods, and constructors in an enum type
- Develop code that declares, implements, and/or extends interface and use the `@Override` annotation

Congratulations! If you are reading this, you've probably passed the Java Programmer I OCA (Oracle Certified Associate) exam, and you are now ready to start your journey through the Java Programmer II OCP (Oracle Certified Professional) exam. Or perhaps you came here from an older version of the certification and are now upgrading.

The OCP builds upon the OCA. You are expected to know the material on the OCA when taking the OCP. Some objectives on the OCP are the same as those on the OCA, such as those concerning access modifiers, overloading, overriding, `abstract` classes, `static`, and `final`. Most are implied. For example, the OCP objectives don't mention `if` statements and loops. Clearly, you still need to know these. We will also point out differences in Java 8 to help those of you coming in from an older version of Java.

If you didn't score well on the OCA exam, or if it has been a while since you took it, we recommend reviewing the book you used to study for it. The OCP questions are a lot tougher. You really need to know the fundamentals well. If you've misplaced your review materials, feel free to check out our OCA book, *OCA: Oracle Certified Associate Java SE 8 Programmer I Study Guide* (Sybex, 2014).

This chapter includes a brief review of overlapping topics and then moves on to new material. You'll see how to use `instanceof`, implement `equals`/`hashCode`/`toString`, create enumerations, and create nested classes.

Reviewing OCA Concepts

In this section, we review the OCA objectives that are explicitly listed as being on the OCP. Since this is review, we will ask you questions followed by a brief reminder of the key points. These questions are harder than the ones on the OCA because they require you to reflect on a lot of what you learned at the same time.

Access Modifiers

First up on the review are the access modifiers `public`, `protected`, and `private` and default access. Imagine the following method exists. For now, just remember the instance variables it tries to access:

```
public static void main(String[] args) {
   BigCat cat = new BigCat();
   System.out.println(cat.name);
   System.out.println(cat.hasFur);
   System.out.println(cat.hasPaws);
   System.out.println(cat.id);
```

Now, suppose each of these classes has this `main` method that instantiates a `BigCat` and tries to print out all four variables. Which variables will be allowed in each case?

```
package cat;
public class BigCat {
   public String name = "cat";
   protected boolean hasFur = true;
   boolean hasPaws = true;
   private int id;
}

package cat.species;
public class Lynx extends BigCat { }

package cat;
public class CatAdmirer { }

package mouse;
public class Mouse { }
```

Think about it for a minute—no really. Pause and try to answer. Ready now? While this code compiles for `BigCat`, it doesn't in all of the classes.

The line with `cat.name` compiles in all four classes because any code can access `public` members. The line with `cat.id` compiles only in `BigCat` because only code in the same class can access `private` members. The line with `cat.hasPaws` compiles only in `BigCat` and `CatAdmirer` because only code in the same package can access code with default access.

Finally, the line with `cat.hasFur` also compiles only in `BigCat` and `CatAdmirer`. `protected` allows subclasses and code in the same package to access members. `Lynx` is a tricky one. Since the code is being accessed via the variable rather than by inheritance, it does not benefit from `protected`. However, if the code in `main` was `Lynx cat = new Lynx();`, `Lynx` would be able to access `cat.hasFur` using `protected` access because it would be seen as a subclass.

NOTE

Remember that there was a `default` keyword introduced in Java 8 for interfaces. That keyword is not an access modifier.

To review the rules for access modifiers at a glance, see Table 1.1.

TABLE 1.1 Access modifiers

Can access	**If that member is `private`?**	**If that member has default (package private) access?**	**If that member is `protected`?**	**If that member is `public`?**
Member in the same class	yes	yes	yes	yes
Member in another class in the same package	no	yes	yes	yes
Member in a superclass in a different package	no	no	yes	yes
Method/field in a class (that is not a superclass) in a different package	no	no	no	yes

Overloading and Overriding

Next we review the differences between overloading and overriding. Which method(s) in `BobcatKitten` overload or override the one in `Bobcat`?

```
1:    public class Bobcat {
2:       public void findDen() { }
3:    }
```

```
1:    public class BobcatKitten extends Bobcat {
2:       public void findDen() { }
3:       public void findDen(boolean b) { }
4:       public int findden() throws Exception { return 0; }
5:    }
```

The one on line 2 is an override because it has the same method signature. The one on line 3 is an overloaded method because it has the same method name but a different parameter list. The one on line 4 is not an override or overload because it has a different method name. Remember that Java is case sensitive.

To review, overloading and overriding happen only when the method name is the same. Further, *overriding* occurs only when the method signature is the same. The *method* *signature* is the method name and the parameter list. For *overloading*, the method parameters must vary by type and/or number.

When multiple overloaded methods are present, Java looks for the closest match first. It tries to find the following:

- Exact match by type
- Matching a superclass type
- Converting to a larger primitive type
- Converting to an autoboxed type
- Varargs

For overriding, the overridden method has a few rules:

- The access modifier must be the same or more accessible.
- The return type must be the same or a more restrictive type, also known as *covariant return types*.
- If any checked exceptions are thrown, only the same exceptions or subclasses of those exceptions are allowed to be thrown.

The methods must not be `static`. (If they are, the method is hidden and not overridden.)

Abstract Classes

Now we move on to reviewing `abstract` classes and methods. What are three ways that you can fill in the blank to make this code compile? Try to think of ways that use the `clean()` method rather than just putting a comment there.

```
abstract class Cat {
   _____________________
}
class Lion extends Cat {
   void clean() {}
}
```

Did you get three? One of them is a little tricky. The tricky one is that you could leave it blank. An `abstract` class is not required to have any methods in it, let alone any `abstract` ones. A second answer is the one that you probably thought of right away:

```
abstract void clean();
```

This one is the actual `abstract` method. It has the `abstract` keyword and a semicolon instead of a method body. A third answer is a default implementation:

```
void clean () {}
```

An `abstract` class may contain any number of methods including zero. The methods can be `abstract` or concrete. `Abstract` methods may not appear in a class that is not `abstract`. The first concrete subclass of an `abstract` class is required to implement all `abstract` methods that were not implemented by a superclass.

Notice that we said three ways. There are plenty of other ways. For example, you could have the `clean()` method throw a `RuntimeException`.

Static and *Final*

Next on the review list are the `static` and `final` modifiers. To which lines in the following code could you independently add `static` and/or `final` without introducing a compiler error?

```
1:    abstract class Cat {
2:       String name = "The Cat";
3:       void clean() { }
4:    }
5:    class Lion extends Cat {
6:       void clean() { }
7:   }
```

Both `static` and `final` can be added to line 2. This allows the variable to be accessed as `Cat.name` and prevents it from being changed. `static` cannot be added to line 3 or 6 independently because the subclass overrides it. It could be added to both, but then you wouldn't be inheriting the method. The `final` keyword cannot be added to line 3 because the subclass method would no longer be able to override it. `final` can be added to line 6 since there are no subclasses of `Lion`.

To review, `final` prevents a variable from changing or a method from being overridden. `static` makes a variable shared at the class level and uses the class name to refer to a method.

`static` and `final` are allowed to be added on the class level too. You will see `static` classes in the section on nested classes at the end of this chapter, so don't worry if you didn't pick up on those. Using `final` on a class means that it cannot be subclassed. As with methods, a class cannot be both `abstract` and `final`. In the Java core classes, `String` is `final`.

Imports

Oracle no longer lists packages and imports in the objectives for the OCP 8 exam. They do include visibility modifiers, which means that you still need to understand packages and imports. So let's review. How many different ways can you think of to write imports that will make this code compile?

```
public class ListHelper {
   public List <String> copyAndSortList(List <String> original) {
      List <String> list = new ArrayList <String>(original);
      sort(list);
      return list;
   }
}
```

The key is to note that this question really has two parts. One thing to figure out is how to get `sort(list)` to compile. Since `sort()` is a `static` method on `Collections`, you definitely need a `static` import. Either of these will do it:

```
import static java.util.Collections.sort;
import static java.util.Collections.*;
```

The other part of the question is to note that `List` and ArrayList are both referenced. These are regular classes and need regular imports. One option is to use a wildcard:

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

The other option is to list them out:

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

There are other imports you can add, but they have redundancy or are unnecessary. For example, you could `import java.lang.*`. However, this package is always imported whether you specify it or not.

Using *instanceof*

Now we move on to the new topics. On the OCA, you learned about many operators including `<` and `==`. Now it is time to learn another: `instanceof`.

In `a instanceof B`, the expression returns `true` if the reference to which `a` points is an instance of class `B`, a subclass of `B` (directly or indirectly), or a class that implements the `B` interface (directly or indirectly).

Let's see how this works. You have three classes with which to work:

```
class HeavyAnimal { }
class Hippo extends HeavyAnimal { }
class Elephant extends HeavyAnimal { }
```

You see that `Hippo` is a subclass of `HeavyAnimal` but not `Elephant`. Remember that the exam starts with line numbers other than 1 when showing a code snippet. This is to tell you that you can assume the correct code comes before what you see. You can assume any missing code is correct and all imports are present.

```
12:   HeavyAnimal hippo = new Hippo();
13:   boolean b1 = hippo instanceof Hippo;          // true
14:   boolean b2 = hippo instanceof HeavyAnimal;    // true
15:   boolean b3 = hippo instanceof Elephant;       // false
```

On line 13, you see that hippo is an instance of itself. We'd certainly hope so! Line 14 returns true because hippo is an instance of its superclass. Line 15 returns false because hippo is not an Elephant. The variable reference is HeavyAnimal, so there could be an Elephant in there. At runtime, Java knows that the variable is in fact pointing to a Hippo.

All Java classes inherit from Object, which means that x instanceof Object is usually true, except for one case where it is false. If the literal null or a variable reference pointing to null is used to check instanceof, the result is false. null is not an Object. For example:

```
26:   HeavyAnimal hippo = new Hippo();
27:   boolean b4 = hippo instanceof Object;      // true
28:   Hippo nullHippo = null;
29:   boolean b5 = nullHippo instanceof Object;  // false
```

Line 27 returns true because Hippo extends from Object indirectly as do all classes. Line 29 returns false because the nullHippo variable reference points to null and null is not a Hippo. This next one is interesting:

```
30:   Hippo anotherHippo = new Hippo();
31:   boolean b5 = anotherHippo instanceof Elephant; // DOES NOT COMPILE
```

Line 31 is a tricky one. The compiler knows that there is no possible way for a Hippo variable reference to be an Elephant, since Hippo doesn't extend Elephant directly or indirectly.

The compilation check only applies when instanceof is called on a class. When checking whether an object is an instanceof an interface, Java waits until runtime to do the check. The reason is that a subclass could implement that interface and the compiler wouldn't know it. There is no way for Hippo to be a subclass of Elephant.

For example, suppose that you have an interface Mother and Hippo does not implement it:

```
public interface Mother {}
class Hippo extends HeavyAnimal { }
```

This code compiles:

```
42:   HeavyAnimal hippo = new Hippo();
43:   boolean b6 = hippo instanceof Mother;
```

It so happens that Hippo does not implement Mother. The compiler allows the statement because there could later be a class such as this:

```
class MotherHippo extends Hippo implements Mother { }
```

The compiler knows an interface could be added, so the instanceof statement could be true for some subclasses, whereas there is no possible way to turn a Hippo into an Elephant.

The instanceof operator is commonly used to determine if an instance is a subclass of a particular object before applying an explicit cast. For example, consider a method that takes as input an Animal reference and performs an operation based on that animal's type:

```
public void feedAnimal(Animal animal) {
   if(animal instanceof Cow) {
      ((Cow)animal).addHay();
   } else if(animal instanceof Bird) {
      ((Bird)animal).addSeed();
   } else if(animal instanceof Lion) {
       ((Lion)animal).addMeat();
   } else {
      throw new RuntimeException("Unsupported animal");
   } }
```

In this example, you needed to know if the animal was an instance of each subclass before applying the cast and calling the appropriate method. For example, a Bird or Lion probably will not have an addHay() method, a Cow or Lion probably will not have an addSeed() method, and so on. The else throwing an exception is common. It allows the code to fail when an unexpected Animal is passed in. This is a good thing. It tells the programmer to fix the code rather than quietly letting the new animal go hungry.

This is not a good way to write code. instanceof and the practice of casting with if statements is extremely rare outside of the exam. It is mostly used when writing a library that will be used by many others. On the exam, you need to understand how instanceof works though.

Understanding Virtual Method Invocation

You just saw a poor way of feeding some animals. A better way would be to make each Animal know how to feed itself. Granted this won't work in the real world, but there could be a sign in each animal habitat or the like.

```
abstract class Animal {
   public abstract void feed(); }
}
class Cow extends Animal {
   public void feed() { addHay(); }
   private void addHay() { }
}
```

```
class Bird extends Animal {
   public void feed() { addSeed(); }
   private void addSeed() { }
}
class Lion extends Animal {
   public void feed() { addMeat(); }
   private void addMeat() { }
}
```

The `Animal` class is `abstract`, and it requires that any concrete `Animal` subclass have a `feed()` method. The three subclasses that we defined have a one-line `feed()` method that delegates to the class-specific method. A `Bird` still gets seed, a `Cow` still gets hay, and so forth. Now the method to feed the animals is really easy. We just call `feed()` and the proper subclass's version is run.

This approach has a huge advantage. The `feedAnimal()` method doesn't need to change when we add a new `Animal` subclass. We could have methods to feed the animals all over the code. Maybe the animals get fed at different times on different days. No matter. `feed()` still gets called to do the work.

```
public void feedAnimal(Animal animal) {
   animal.feed();
}
```

We've just relied on *virtual method invocation*. We actually saw virtual methods on the OCA. They are just regular non-static methods. Java looks for an overridden method rather than necessarily using the one in the class that the compiler says we have. The only thing new about virtual methods on the OCP is that Oracle now *calls* them virtual methods in the objectives. You can simply think of them as methods.

In the above example, we have an `Animal` instance, but Java didn't call `feed` on the `Animal` class. Instead Java looked at the actual type of `animal` at runtime and called `feed` on that.

Notice how this technique is called virtual *method* invocation. Instance variables don't work this way. In this example, the `Animal` class refers to `name`. It uses the one in the superclass and not the subclass.

```
abstract class Animal {
   String name = "???";
   public void printName() {
      System.out.println(name);
   }
}
class Lion extends Animal {
   String name = "Leo";
}
```

```
public class PlayWithAnimal {
   public static void main(String... args) {
      Animal animal = new Lion();
      animal.printName();
   }
}
```

This outputs `???`. The `name` declared in `Lion` would only be used if `name` was referred to from `Lion` (or a subclass of `Lion`.) But no matter how you call `printName()`, it will use the `Animal`'s name, not the `Lion`'s name.

Aside from the formal sounding name, there isn't anything new here. Let's try one more example to make sure that the exam can't trick you. What does the following print?

```
abstract class Animal {
   public void careFor() {
      play();
   }
   public void play() {
      System.out.println("pet animal");
   } }
class Lion extends Animal {
   public void play() {
      System.out.println("toss in meat");
   } }
public class PlayWithAnimal {
   public static void main(String... args) {
      Animal animal = new Lion();
      animal.careFor();
   } }
```

The correct answer is `toss in meat`. The `main` method creates a new `Lion` and calls `careFor`. Since only the `Animal` superclass has a `careFor` method, it executes. That method calls `play`. Java looks for overridden methods, and it sees that `Lion` implements `play`. Even though the call is from the `Animal` class, Java still looks at subclasses, which is good because you don't want to pet a `Lion`!

Annotating Overridden Methods

You already know how to override a method. Java provides a way to indicate explicitly in the code that a method is being overridden. In Java, when you see code that begins with an `@` symbol, it is an annotation. An *annotation* is extra information about the program, and it is a type of *metadata*. It can be used by the compiler or even at runtime.

The @Override annotation is used to express that you, the programmer, intend for this method to override one in a superclass or implement one from an interface. You don't traditionally think of implementing an interface as overriding, but it actually is an override. It so happens that the method being overridden is an abstract one.

The following example shows this annotation in use:

```
1:    class Bobcat {
2:      public void findDen() { }
3:    }
4:    class BobcatMother extends Bobcat {
5:       @Override
6:       public void findDen() { }
7:    }
```

Line 5 tells the compiler that the method on line 6 is intended to override another method. Java ignores whitespace, which means that lines 5 and 6 could be merged into one:

```
6:    @Override public void findDen(boolean b) { }
```

This is helpful because the compiler now has enough information to tell you when you've messed up. Imagine if you wrote

```
1:    class Bobcat {
2:      public void findDen() { }
3:    }
4:    class BobcatMother extends Bobcat {
5:       @Override
6:       public void findDen(boolean b) { } // DOES NOT COMPILE
7:    }
```

Line 5 still tells Java the method that line 6 is intended to override another method. However, the method on line 6 overloads the method rather than overriding it. Java recognizes that this is a broken promise and gives it a compiler error.

It is useful to have the compiler tell you that you are not actually overriding when you think that you are. The problem could be a typo. Or it could be that the superclass or interface changed without your knowledge. Either way, it is useful information to know so that you can fix the code. It is a great idea to get in the habit of using @Override in order to avoid accidentally overloading a method.

@Override is allowed only when referencing a method. Just as there is no such thing as overriding a field, the annotation cannot be used on a field either.

Much of the time, you will not see @Override used on the exam when a method is being overridden. The exam is testing whether you can recognize an overridden method. However, when you see @Override show up on the exam, you must check carefully that the method is doing one of three things:

- Implementing a method from an interface
- Overriding a superclass method of a class shown in the example
- Overriding a method declared in `Object`, such as `hashCode`, `equals`, or `toString`

To be fair, the third one is a special case of the second. It is less obvious. Since the methods aren't declared on the page in front of you, we mention it specifically. Pay attention to the signatures of these three methods in the next sections so that you know the method signatures well and can spot where they are overridden.

Coding *equals*, *hashCode*, and *toString*

All classes in Java inherit from `java.lang.Object`, either directly or indirectly, which means that all classes inherit any methods defined in `Object`. Three of these methods are common for subclasses to override with a custom implementation. First, we will look at `toString()`. Then we will talk about `equals()` and `hashCode()`. Finally, we will discuss how `equals()` and `hashCode()` relate.

toString

When studying for the OCA, we learned that Java automatically calls the `toString()` method if you try to print out an object. We also learned that some classes supply a human-readable implementation of `toString()` and others do not. When running the following example, we see one of each:

```
public static void main(String[] args) {
   System.out.println(new ArrayList());      // []
   System.out.println(new String[0]);        // [Ljava.lang.String;@65cc892e
}
```

`ArrayList` provided an implementation of `toString()` that listed the contents of the `ArrayList`, in this case, an empty `ArrayList`. (If you want to be technical about it, a superclass of `ArrayList` implemented `toString()` and `ArrayList` inherited that one instead of the one in `Object`, whereas the array used the default implementation from `Object`.) You don't need to know that for the exam, though.

Clearly, providing nice human-readable output is going to make things easier for developers working with your code. They can simply print out your object and understand what it represents. Luckily, it is easy to override `toString()` and provide your own implementation.

Let's start with a nice, simple example:

```
public class Hippo {
   private String name;
   private double weight;
   public Hippo(String name, double weight) {
      this.name = name;
      this.weight = weight;
   }
   @Override
   public String toString() {
      return name;
   }
   public static void main(String[] args) {
      Hippo h1 = new Hippo("Harry", 3100);
      System.out.println(h1);     // Harry
   } }
```

Now when we run this code, it prints `Harry`. Granted that we have only one `Hippo`, so it isn't hard to keep track of this! But when the zoo later gets a whole family of hippos, it will be easier to remember who is who.

When you implement the `toString()` method, you can provide as much or as little information as you would like. In this example, we use all of the instance variables in the object:

```
public String toString() {
   return "Name: " + name + ", Weight: " + weight;
}
```

Real World Scenario

The Easy Way to Write `toString()` Methods

Once you've written a `toString()` method, it starts to get boring to write more—especially if you want to include a lot of instance variables. Luckily, there is an open source library that takes care of it for you. Apache Commons Lang (`http://commons.apache.org/proper/commons-lang/`) provides some methods that you might wish were in core Java.

This is all you have to write to have Apache Commons return all of the instance variables in a `String`:

```
public String toString() {
   return ToStringBuilder.reflectionToString(this);
}
```

Calling our `Hippo` test class with this `toString()` method outputs something like `toString.Hippo@12da89a7[name=Harry,weight=3100.0]`. You might be wondering what this reflection thing is that is mentioned in the method name. *Reflection* is a technique used in Java to look at information about the class at runtime. This lets the ToStringBuilder class determine what are all of the instance variables and to construct a String with each.

When testing your code, there is a benefit to not having information in toString() that isn't useful to your caller (12da89a7). Apache Commons accounts for this as well. You can write

```
 @Override public String toString() {
   return ToStringBuilder.reflectionToString(this,
      ToStringStyle.SHORT_PREFIX_STYLE);
}
```

This time our Hippo test class outputs Hippo[name=Harry,weight=3100.0]. There are a few other styles that support letting you choose to omit the class names or the instance variable names.

equals

Remember that Java uses == to compare primitives and for checking if two variables refer to the same object. Checking if two objects are equivalent uses the equals() method, or at least it does if the developer implementing the method overrides equals(). In this example, you can see that only one of the two classes provides a custom implementation of equals():

```
String s1 = new String("lion");
String s2 = new String("lion");
System.out.println(s1.equals(s2));                 // true
StringBuilder sb1 = new StringBuilder("lion");
StringBuilder sb2 = new StringBuilder("lion");
System.out.println(sb1.equals(sb2));               // false
```

String does have an equals() method. It checks that the values are the same. StringBuilder uses the implementation of equals() provided by Object, which simply checks if the two objects being referred to are the same.

There is more to writing your own equals() method than there was to writing toString(). Suppose the zoo gives every lion a unique identification number. The following Lion class implements equals() to say that any two Lion objects with the same ID are the same Lion:

```
public class Lion {
   private int idNumber;
3:      private int age;
4:      private String name;
5:      public Lion(int idNumber, int age, String name) {
6:        this.idNumber = idNumber;
7:        this.age = age;
8:        this.name = name;
9:     }
10:    @Override public boolean equals(Object obj) {
11:       if ( !(obj instanceof Lion)) return false;
12:       Lion otherLion = (Lion) obj;
13:       return this.idNumber == otherLion.idNumber;
14:    }
15: }
```

First, pay attention to the method signature on line 10. It takes an `Object` as the method parameter rather than a `Lion`. Line 11 checks whether a cast would be allowed. You get to use the new `instanceof` operator that you just learned! There is no way that a `Lion` is going to be equal to a `String`. The method needs to return `false` when this occurs. If you get to line 12, a cast is OK. Then line 13 checks whether the two objects have the same identification number.

The `this.` syntax is not required. Line 12 could have been `return idNumber == otherLion.idNumber`. Many programmers explicitly code `this.` to be explicit about the object being referenced.

The Contract for `equals()` Methods

Since `equals()` is such a key method, Java provides a number of rules in the contract for the method. The exam expects you to recognize correct and incorrect `equals()` methods, but it will not ask you to name which property is broken. That said, it is helpful to have seen it at least once.

The `equals()` method implements an equivalence relation on non-null object references:

- *It is reflexive*: For any non-null reference value `x`, `x.equals(x)` should return `true`.
- *It is symmetric*: For any non-null reference values `x` and `y`, `x.equals(y)` should return `true` if and only if `y.equals(x)` returns `true`.
- *It is transitive*: For any non-null reference values `x`, `y`, and `z`, if `x.equals(y)` returns `true` and `y.equals(z)` returns `true`, then `x.equals(z)` should return `true`.
- *It is consistent*: For any non-null reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return `true` or consistently return `false`, provided no information used in equals comparisons on the objects is modified.
- For any non-null reference value `x`, `x.equals(null)` should return `false`.

Much of this is common sense. The definition of equality doesn't change at random, and the same objects can't be equal "sometimes." The most interesting rule is the last one. It should be obvious that an object and `null` aren't equal. The key is that `equals()` needs to return `false` when this occurs rather than throw a `NullPointerException`.

For practice, can you see what is wrong with this `equals()` method?

```
public boolean equals(Lion obj) {
   if (obj == null) return false;
   return this.idNumber == obj.idNumber;
}
```

There is actually nothing wrong with this method. It is a perfectly good method. However, it does not override `equals()` from `Object`. It overloads that method, which is probably not what was intended.

Real World Scenario

The Easy Way to Write `equals()` Methods

Like `toString()`, you can use Apache Commons Lang to do a lot of the work for you. If you want all of the instance variables to be checked, your `equals()` method can be one line:

```
public boolean equals(Object obj) {
   return EqualsBuilder.reflectionEquals(this, obj);
}
```

This is nice. However, for `equals()`, it is common to look at just one or two instance variables rather than all of them.

```
public boolean equals(Object obj) {
   if ( !(obj instanceof LionEqualsBuilder)) return false;
   Lion other = (Lion) obj;
   return new EqualsBuilder().appendSuper(super.equals(obj))
      .append(idNumber, other.idNumber)
      .append(name, other.name)
      .isEquals();
}
```

Not quite as elegant, right? You have to remember to handle the `null` and `instanceof` guard conditions first. It is still better than having to code the whole thing by hand, though. Comparing the `idNumber` is easy because you can call `==`. Comparing the name means checking that either both names are `null` or the names are the same. If either name is `null`, you need to return `false`. This logic is a bit messy if you write it out by hand.

hashCode

Whenever you override `equals()`, you are also expected to override `hashCode()`. The hash code is used when storing the object as a key in a map. You will see this in Chapter 3, "Generics and Collections."

A *hash code* is a number that puts instances of a class into a finite number of categories. Imagine that I gave you a deck of cards, and I told you that I was going to ask you for specific cards and I want to get the right card back quickly. You have as long as you want to prepare, but I'm in a big hurry when I start asking for cards. You might make 13 piles of cards: All of the aces in one pile, all the twos in another pile, and so forth. That way, when I ask for the five of hearts, you can just pull the right card out of the four cards in the pile with fives. It is certainly faster than going through the whole deck of 52 cards! You could even make 52 piles if you had enough space on the table.

The following is the code that goes with our little story. Cards are equal if they have the same rank and suit. They go in the same pile (hash code) if they have the same rank.

```
public class Card {
   private String rank;
   private String suit;
   public Card(String r, String s) {
      if (r == null || s == null)
         throw new IllegalArgumentException();
      rank = r;
      suit = s;
   }
   public boolean equals(Object obj) {
      if ( !(obj instanceof Card)) return false;
      Card c = (Card) obj;
      return rank.equals(c.rank) && suit.equals(c.suit);
   }
   public int hashCode() {
      return rank.hashCode();
   }
}
```

In the constructor, you make sure that neither instance variable is `null`. This check allows `equals()` to be simpler because you don't have to worry about `null` there. The `hashCode()` method is quite simple. It asks the *`rank`* for its hash code and uses that.

That's all well and good. But what do you do if you have a primitive and need the hash code? The hash code is just a number. On the exam, you can just use a primitive number as is or divide to get a smaller `int`. Remember that all of the instance variables don't need to be used in a `hashCode()` method. It is common not to include `boolean` and `char` variables in the hash code.

The official JavaDoc contract for `hashCode()` is harder to read than it needs to be. The three points in the contract boil down to these:

- Within the same program, the result of `hashCode()` must not change. This means that you shouldn't include variables that change in figuring out the hash code. In our hippo example, including the name is fine. Including the weight is not because hippos change weight regularly.
- If `equals()` returns `true` when called with two objects, calling `hashCode()` on each of those objects must return the same result. This means `hashCode()` can use a subset of the variables that `equals()` uses. You saw this in the `card` example. We used only one of the variables to determine the hash code.
- If `equals()` returns `false` when called with two objects, calling `hashCode()` on each of those objects does not have to return a different result. This means `hashCode()` results do not need to be unique when called on unequal objects.

Going back to our `Lion`, which has three instance variables and only used `idNumber` in the `equals()` method, which of these do you think are legal `hashCode()` methods?

```
16:   public int hashCode() { return idNumber; }
17:   public int hashCode() { return 6; }
18:   public long hashcode() { return idNumber; }
19:   public int hashCode() { return idNumber * 7 + age; }
```

Line 16 is what you would expect the `hashCode()` method to be. Line 17 is also legal. It isn't particularly efficient. It is like putting the deck of cards in one giant pile. But it is legal. Line 18 is not an override of `hashCode()`. It uses a lowercase `c`, which makes it a different method. If it were an override, it wouldn't compile because the return type is wrong. Line 19 is not legal because it uses more variables than `equals()`.

Real World Scenario

The Easy Way to Write `hashCode()` Methods

You probably thought that this was going to be about the Apache Commons Lang class for hash code. There is one, but it isn't the easiest way to write hash code.

It is easier to code your own. Just pick the key fields that identify your object (and don't change during the program) and combine them:

```
public int hashCode() {
   return keyField + 7 * otherKeyField.hashCode();
}
```

It is common to multiply by a prime number when combining multiple fields in the hash code. This makes the hash code more unique, which helps when distributing objects into buckets.

Working with *Enums*

In programming, it is common to have a type that can only have a finite set of values. An *enumeration* is like a fixed set of constants. In Java, an *enum* is a class that represents an enumeration. It is much better than a bunch of constants because it provides type-safe checking. With numeric constants, you can pass an invalid value and not find out until runtime. With `enums`, it is impossible to create an invalid `enum` type without introducing a compiler error.

Enumerations show up whenever you have a set of items whose types are known at compile time. Common examples are the days of the week, months of the year, the planets in the solar system, or the cards in a deck. Well, maybe not the planets in a solar system, given that Pluto had its planetary status revoked.

To create an `enum`, use the `enum` keyword instead of the `class` keyword. Then list all of the valid types for that `enum`.

```
public enum Season {
   WINTER, SPRING, SUMMER, FALL
}
```

Since an `enum` is like a set of constants, use the uppercase letter convention that you used for constants.

Behind the scenes, an `enum` is a type of class that mainly contains `static` members. It also includes some helper methods like `name()` that you will see shortly. Using an `enum` is easy:

```
Season s = Season.SUMMER;
System.out.println(Season.SUMMER);                   // SUMMER
System.out.println(s == Season.SUMMER);              // true
```

As you can see, `enums` print the name of the `enum` when `toString()` is called. They are also comparable using `==` because they are like `static final` constants.

An enum provides a method to get an array of all of the values. You can use this like any normal array, including in a loop:

```
for(Season season: Season.values()) {
   System.out.println(season.name() + " " + season.ordinal());
}
```

The output shows that each enum value has a corresponding int value in the order in which they are declared. The int value will remain the same during your program, but the program is easier to read if you stick to the human-readable enum value.

```
WINTER 0
SPRING 1
SUMMER 2
FALL 3
```

You can't compare an int and enum value directly anyway. Remember that an enum is a type and *not* an int.

```
if ( Season.SUMMER == 2) {} // DOES NOT COMPILE
```

You can also create an enum from a String. This is helpful when working with older code. The String passed in must match exactly, though.

```
Season s1 = Season.valueOf("SUMMER");    // SUMMER
Season s2 = Season.valueOf("summer");    // exception
```

The first statement works and assigns the proper enum value to s1. The second statement encounters a problem. There is no enum value with the lowercase name "summer." Java throws up its hands in defeat and throws an IllegalArgumentException.

```
Exception in thread "main" java.lang.IllegalArgumentException: No enum constant
enums.Season.summer
```

Another thing that you can't do is extend an enum.

```
public enum ExtendedSeason extends Season { } // DOES NOT COMPILE
```

The values in an enum are all that are allowed. You cannot add more at runtime by extending the enum.

Now that we've covered the basics, we look at using enums in switch statements and how to add extra functionality to enums.

Using *Enums* in *Switch* Statements

Enums may be used in switch statements. Pay attention to the case value in this code:

```
Season summer = Season.SUMMER;
switch (summer) {
   case WINTER:
      System.out.println("Get out the sled!");
      break;
   case SUMMER:
      System.out.println("Time for the pool!");
      break;
   default:
      System.out.println("Is it summer yet?");
}
```

The code prints "Time for the pool!" since it matches SUMMER. Notice that we just typed the value of the enum rather than writing Season.WINTER. The reason is that Java already knows that the only possible matches can be enum values. Java treats the enum type as implied. In fact, if you were to type case Season.WINTER, it would not compile. Keep in mind that an enum type is not an int. The following code does not compile:

```
switch (summer) {
   case 0:     // DOES NOT COMPILE
      System.out.println("Get out the sled!");
      break;
}
```

You can't compare an int with an enum. Pay special attention when working with enums that they are used only as enums.

Adding Constructors, Fields, and Methods

Enums can have more in them than just values. It is common to give state to each one. Our zoo wants to keep track of traffic patterns for which seasons get the most visitors.

```
1:    public enum Season {
2:       WINTER("Low"), SPRING("Medium"), SUMMER("High"), FALL("Medium");
3:       private String expectedVisitors;
4:       private Season(String expectedVisitors) {
5:          this.expectedVisitors = expectedVisitors;
6:       }
7:       public void printExpectedVisitors() {
8:          System.out.println(expectedVisitors);
9:       } ]
```

There are a few things to notice here. On line 2, we have a semicolon. This is required if there is anything in the enum besides the values.

NOTE

This is such a subtle detail that we are going to say it again: remember that the semicolon at the end of the enum values is optional only if the only thing in the enum is that list of values.

Lines 3–9 are regular Java code. We have an instance variable, a constructor, and a method. The constructor is `private` because it can only be called from within the `enum`. The code will not compile with a `public` constructor.

Calling this new method is easy:

```
Season.SUMMER.printExpectedVisitors();
```

Notice how we don't appear to call the constructor. We just say that we want the `enum` value. The first time that we ask for any of the `enum` values, Java constructs all of the `enum` values. After that, Java just returns the already-constructed `enum` values. Given that explanation, you can see why this code calls the constructor only once:

```
public enum OnlyOne {
   ONCE(true);
   private OnlyOne(boolean b) {
      System.out.println("constructing");
   }
   public static void main(String[] args) {
      OnlyOne firstCall = OnlyOne.ONCE;    // prints constructing
      OnlyOne secondCall = OnlyOne.ONCE;   // doesn't print anything
    } }
```

This technique of a constructor and state allows you to combine logic with the benefit of a list of values. Sometimes, you want to do more. For example, our zoo has different seasonal hours. It is cold and gets dark early in the winter. We could keep track of the hours through instance variables, or we can let each `enum` value manage hours itself:

```
public enum Season {
   WINTER {
      public void printHours() { System.out.println("9am-3pm"); }
   }, SPRING {
      public void printHours() { System.out.println("9am-5pm"); }
   }, SUMMER {
      public void printHours() { System.out.println("9am-7pm"); }
   }, FALL {
      public void printHours() { System.out.println("9am-5pm"); }
   };
   public abstract void printHours();
}
```

What's going on here? It looks like we created an abstract class and a bunch of tiny subclasses. In a way we did. The enum itself has an abstract method. This means that each and every enum value is required to implement this method. If we forget one, we get a compiler error.

If we don't want each and every enum value to have a method, we can create a default implementation and override it only for the special cases:

```
public enum Season3 {
   WINTER {
      public void printHours() { System.out.println("short hours"); }
   }, SUMMER {
      public void printHours() { System.out.println("long hours"); }
   }, SPRING, FALL;
   public void printHours() { System.out.println("default hours"); }
}
```

This one looks better. We only code the special cases and let the others use the enum-provided implementation. Notice how we still have the semicolon after FALL. This is needed when we have anything other than just the values. In this case, we have a default method implementation.

Just because an enum can have lots of methods, doesn't mean that it should. Try to keep your enums simple. If your enum is more than a page or two, it is way too long. Most enums are just a handful of lines. The main reason they get long is that when you start with a one- or two-line method and then declare it for each of your dozen enum types, it grows long. When they get too long or too complex, it makes the enum hard to read.

Creating Nested Classes

A *nested class* is a class that is defined within another class. A nested class that is not static is called an *inner class*. There are four of types of nested classes:

- A member inner class is a class defined at the same level as instance variables. It is not static. Often, this is just referred to as an inner class without explicitly saying the type.
- A local inner class is defined within a method.
- An anonymous inner class is a special case of a local inner class that does not have a name.
- A static nested class is a static class that is defined at the same level as static variables.

There are a few benefits of using inner classes. They can encapsulate helper classes by restricting them to the containing class. They can make it easy to create a class that will be used in only one place. They can make the code easier to read. They can also make the code harder to read when used improperly. Unfortunately, the exam tests these edge cases where programmers wouldn't actually use a nested class. This section covers all four types of nested classes.

Member Inner Classes

A *member inner class* is defined at the member level of a class (the same level as the methods, instance variables, and constructors). Member inner classes have the following properties:

- Can be declared public, private, or protected or use default access
- Can extend any class and implement interfaces
- Can be abstract or final
- Cannot declare static fields or methods
- Can access members of the outer class including `private` members

The last property is actually pretty cool. It means that the inner class can access the outer class without doing anything special. Ready for a complicated way to print "Hi" three times?

```
1:   public class Outer {
2:      private String greeting = "Hi";
3:
4:      protected class Inner {
5:         public int repeat = 3;
6:         public void go() {
7:         for (int i = 0; i < repeat; i++)
8:            System.out.println(greeting);
9:     }
10:  }
11:
12:  public void callInner() {
13:     Inner inner = new Inner();
14:     inner.go();
15:  }
16:  public static void main(String[] args) {
17:     Outer outer = new Outer();
18:     outer.callInner();
19: } }
```

A member inner class declaration looks just like a stand-alone class declaration except that it happens to be located inside another class—oh, and that it can use the instance variables declared in the outer class. Line 8 shows that the inner class just refers to `greeting` as if it were available. This works because it is in fact available. Even though the variable is `private`, it is within that same class.

Since a member inner class is not static, it has to be used with an instance of the outer class. Line 13 shows that an instance of the outer class can instantiate `Inner` normally. This works because `callInner()` is an instance method on `Outer`. Both `Inner` and `callInner()` are members of `Outer`. Since they are peers, they just write the name.

There is another way to instantiate `Inner` that looks odd at first. OK, well maybe not just at first. This syntax isn't used often enough to get used to it:

```
20:   public static void main(String[] args) {
21:      Outer outer = new Outer();
22:      Inner inner = outer.new Inner();   // create the inner class
23:      inner.go();
24: }
```

Let's take a closer look at line 22. We need an instance of `Outer` in order to create `Inner`. We can't just call `new Inner()` because Java won't know with which instance of `Outer` it is associated. Java solves this by calling `new` as if it were a method on the *`outer`* variable.

`.class` Files for Inner Classes

Compiling the `Outer.java` class with which we have been working creates two class files. `Outer.class` you should be expecting. For the inner class, the compiler creates `Outer$Inner.class`. You don't need to know this syntax for the exam. We mention it so that you aren't surprised to see files with `$` appearing in your directories. You do need to understand that multiple class files are created.

Inner classes can have the same variable names as outer classes. There is a special way of calling `this` to say which class you want to access. You also aren't limited to just one inner class. Please never do this in code you write. Here is how to nest multiple classes and access a variable with the same name in each:

```
public class A {
   private int x = 10;
   class B {
     private int x = 20;
     class C {
        private int x = 30;
        public void allTheX() {
           System.out.println(x);            // 30
           System.out.println(this.x);       // 30
           System.out.println(B.this.x);     // 20
           System.out.println(A.this.x);    // 10
12:        } } }
13:      public static void main(String[] args) {
14:         A a = new A();
15:         A.B b = a.new B();
16:         A.B.C c = b.new C();
17:         c.allTheX();
18:   }}
```

Yes, this code makes us cringe too. It has two nested classes. Line 14 instantiates the outermost one. Line 15 uses the awkward syntax to instantiate a `B`. Notice the type is `A.B`. We could have written `B` as the type because that is available at the member level of `B`. Java knows where to look for it. On line 16, we instantiate a `C`. This time, the `A.B.C` type is necessary to specify. `C` is too deep for Java to know where to look. Then line 17 calls a method on `c`.

Lines 8 and 9 are the type of code that we are used to seeing. They refer to the instance variable on the current class—the one declared on line 6 to be precise. Line 10 uses `this` in a special way. We still want an instance variable. But this time we want the one on the `B` class, which is the variable on line 4. Line 11 does the same thing for class `A`, getting the variable from line 2.

Private Interfaces

This following code looks weird but is legal:

```
public class CaseOfThePrivateInterface {
    private interface Secret {
      public void shh();
    }
    class DontTell implements Secret {
       public void shh() { }
    } }
```

The rule that all methods in an interface are `public` still applies. A class that implements the interface must define that method as `public`.

The interface itself does not have to be `public`, though. Just like any inner class, an inner interface can be `private`. This means that the interface can only be referred to within the current outer class.

Local Inner Classes

A *local inner class* is a nested class defined within a method. Like local variables, a local inner class declaration does not exist until the method is invoked, and it goes out of scope when the method returns. This means that you can create instances only from within the method. Those instances can still be returned from the method. This is just how local variables work. Local inner classes have the following properties:

- They do not have an access specifier.
- They cannot be declared `static` and cannot declare `static` fields or methods.
- They have access to all fields and methods of the enclosing class.
- They do not have access to local variables of a method unless those variables are `final` or effectively final. More on this shortly.

Ready for an example? Here's a complicated way to multiply two numbers:

```
1:   public class Outer {
2:      private int length = 5;
3:      public void calculate() {
4:         final int width = 20;
5:         class Inner {
6:            public void multiply() {
7:               System.out.println(length * width);
8:            }
9:         }
10:        Inner inner = new Inner();
11:        inner.multiply();
12:     }
13:     public static void main(String[] args) {
14:        Outer outer = new Outer();
15:        outer.calculate();
16:     }
17: }
```

Lines 5 through 9 are the local inner class. That class's scope ends on line 12 where the method ends. Line 7 refers to an instance variable and a final local variable, so both variable references are allowed from within the local inner class.

Earlier, we made the statement that local variable references are allowed if they are `final` or effectively final. Let's talk about that now. The compiler is generating a class file from your inner class. A separate class has no way to refer to local variables. If the local variable is `final`, Java can handle it by passing it to the constructor of the inner class or by storing it in the class file. If it weren't effectively final, these tricks wouldn't work because the value could change after the copy was made. Up until Java 7, the programmer actually had to type the `final` keyword. In Java 8, the "effectively final" concept was introduced. If the code could still compile with the keyword `final` inserted before the local variable, the variable is effectively final.

Remember that the "effectively final" concept was introduced in Java 8. If you are looking at older mock exam questions online, some of the answers about local variables and inner classes might be different.

Which of the variables do you think are effectively final in this code?

```
34:   public void isItFinal() {
35:      int one = 20;
36:      int two = one;
37:      two++;
38:      int three;
39:      if ( one == 4) three = 3;
40:      else three = 4;
41:      int four = 4;
42:      class Inner { }
43:      four = 5;
44: }
```

`one` is effectively final. It is only set in the line in which it is declared. `two` is not effectively final. The value is changed on line 37 after it is declared. `three` is effectively final because it is assigned only once. This assignment may happen in either branch of the `if` statement, but it can happen in only one of them. `four` is not effectively final. Even though the assignment happens after the inner class, it is not allowed.

Anonymous Inner Classes

An *anonymous inner class* is a local inner class that does not have a name. It is declared and instantiated all in one statement using the `new` keyword. Anonymous inner classes are required to extend an existing class or implement an existing interface. They are useful when you have a short implementation that will not be used anywhere else. Here's an example:

```
1:    public class AnonInner {
2:       abstract class SaleTodayOnly {
3:          abstract int dollarsOff();
4:       }
5:       public int admission(int basePrice) {
6:          SaleTodayOnly sale = new SaleTodayOnly() {
7:             int dollarsOff() { return 3; }
8:          };
9:          return basePrice - sale.dollarsOff();
10:  } }
```

Lines 2 through 4 define an `abstract` class. Lines 6 through 8 define the inner class. Notice how this inner class does not have a name. The code says to instantiate a new `SaleTodayOnly` object. But wait. `SaleTodayOnly` is `abstract`. This is OK because we provide the class body right there—anonymously.

Pay special attention to the semicolon on line 8. We are declaring a local variable on these lines. Local variable declarations are required to end with semicolons, just like other Java statements—even if they are long and happen to contain an anonymous inner class.

Now we convert this same example to implement an interface instead of extending an abstract class:

```
1:    public class AnonInner {
2:       interface SaleTodayOnly {
3:          int dollarsOff();
4:       }
5:       public int admission(int basePrice) {
6:          SaleTodayOnly sale = new SaleTodayOnly() {
7:             public int dollarsOff() { return 3; }
8:          };
9:          return basePrice - sale.dollarsOff();
10:   } }
```

The most interesting thing here is how little has changed. Lines 2 through 4 declare an interface instead of an abstract class. Line 7 is public instead of using default access since interfaces require public methods. And that is it. The anonymous inner class is the same whether you implement an interface or extend a class! Java figures out which one you want automatically.

But what if we want to implement both an interface and extend a class? You can't with an anonymous inner class, unless the class to extend is java.lang.Object. Object is a special class, so it doesn't count in the rule. Remember that an anonymous inner class is just an unnamed local inner class. You can write a local inner class and give it a name if you have this problem. Then you can extend a class and implement as many interfaces as you like. If your code is this complex, a local inner class probably isn't the most readable option anyway.

There is one more thing that you can do with anonymous inner classes. You can define them right where they are needed, even if that is an argument to another method:

```
1:    public class AnonInner {
2:       interface SaleTodayOnly {
3:          int dollarsOff();
4:       }
5:       public int pay() {
6:          return admission(5, new SaleTodayOnly() {
7:             public int dollarsOff() { return 3;    }
8:          });
9:       }
10:      public int admission(int basePrice, SaleTodayOnly sale) {
```

```
11:        return basePrice - sale.dollarsOff();
12:    }}
```

Lines 6 through 8 are the anonymous inner class. We don't even store it in a local variable. Instead, we pass it directly to the method that needs it. Reading this style of code does take some getting used to. But it is a concise way to create a class that you will use only once.

Before you get too attached to anonymous inner classes, know that you'll see a shorter way of coding them in Chapter 4, "Functional Programming."

Real World Scenario

Inner Classes as Event Handlers

Writing graphical user interface code isn't on the exam. Nonetheless, it is a very common use of inner classes, so we'll give you a taste of it here:

```
JButton button = new JButton("red");
button.addActionListener(new ActionListener() {

      public void actionPerformed(ActionEvent e) {
         // handle the button click
      }
});
```

This technique gives the event handler access to the instance variables in the class with which it goes. It works well for simple event handling.

You should be aware that inner classes go against some fundamental concepts, such as reuse of classes and high cohesion (discussed in the next chapter). Therefore, make sure that inner classes make sense before you use them in your code.

Static Nested Classes

The final type of nested class is not an inner class. A *static nested class* is a static class defined at the member level. It can be instantiated without an object of the enclosing class, so it can't access the instance variables without an explicit object of the enclosing class. For example, `new OuterClass().var` allows access to the instance variable `var`.

In other words, it is like a regular class except for the following:

- The nesting creates a namespace because the enclosing class name must be used to refer to it.
- It can be made `private` or use one of the other access modifiers to encapsulate it.
- The enclosing class can refer to the fields and methods of the `static` nested class.

```
1:   public class Enclosing {
2:      static class Nested {
3:         private int price = 6;
4:      }
5:      public static void main(String[] args) {
6:         Nested nested = new Nested();
7:         System.out.println(nested.price);
8:   } }
```

Line 6 instantiates the nested class. Since the class is `static`, you do not need an instance of `Enclosing` in order to use it. You are allowed to access `private` instance variables, which is shown on line 7.

Importing a `static` Nested Class

Importing a `static` nested class is interesting. You can import it using a regular import:

```
package bird;
public class Toucan {
   public static class Beak {}
}
package watcher;
import bird.Toucan.Beak;    // regular import ok
public class BirdWatcher {
   Beak beak;
}
```

And since it is `static`, alternatively you can use a `static` import:

```
import static bird.Toucan.Beak;
```

Either one will compile. Surprising, isn't it? Java treats the `static` nested class as if it were a namespace.

To review the four types of nested classes, make sure that you know the information in Table 1.2.

TABLE 1.2 Types of nested classes

	Member inner class	**Local inner class**	**Anonymous inner class**	`static` **nested class**
Access modifiers allowed	`public`, `protected`, `private`, or default access	None. Already local to method.	None. Already local to statement.	`public`, `protected`, `private`, or default access
Can extend any class and any number of interfaces	Yes	Yes	No—must have exactly one superclass or one interface	Yes
Can be `abstract`	Yes	Yes	N/A—because no class definition	Yes
Can be `final`	Yes	Yes	N/A—because no class definition	Yes
Can access instance members of enclosing class	Yes	Yes	Yes	No (not directly; requires an instance of the enclosing class)
Can access local variables of enclosing class	No	Yes—if `final` or effectively final	Yes—if `final` or effectively final	No
Can declare `static` methods	No	No	No	Yes

Summary

The `instanceof` keyword compares an object to a class or interface type. It also looks at subclasses and subinterfaces. `x instanceof Object` returns `true` unless `x` is `null`. If the compiler can determine that there is no way for `instanceof` to return `true`, it will generate a compiler error. Virtual method invocation means that Java will look at subclasses when finding the right method to call. This is true, even from within a method in the superclass.

The methods `toString()`, `equals()`, and `hashCode()` are implemented in `Objects` that classes can override to change their behavior. `toString()` is used to provide a human-readable representation of the object. `equals()` is used to specify which instance variables should be considered for equality. `equals()` is required to return `false` when the object passed in is `null` or is of the wrong type. `hashCode()` is used to provide a grouping in some collections. `hashCode()` is required to return the same number when called with objects that are `equals()`.

The `enum` keyword is short for enumerated values or a list of values. `Enums` can be used in `switch` statements. They are not `int` values and cannot be compared to `int` values. In a `switch`, the `enum` value is placed in the case. `Enums` are allowed to have instance variables, constructors, and methods. `Enums` can also have value-specific methods. The `enum` itself declares that method as well. It can be `abstract`, in which case all `enum` values must provide an implementation. Alternatively, it can be concrete, in which case `enum` values can choose whether they want to override the default implementation.

There are four types of nested classes. Member inner classes require an instance of the outer class to use. They can access `private` members of that outer class. Local inner classes are classes defined within a method. They can also access `private` members of the outer class. Local inner classes can also access `final` or effectively final local variables. Anonymous inner classes are a special type of local inner class that does not have a name. Anonymous inner classes are required to extend exactly one class by name or implement exactly one `interface`. Static nested classes can exist without an instance of the outer class.

This chapter also contained a review of access modifiers, overloading, overriding, abstract classes, static, final, and imports. It also introduced the optional `@Override` annotation for overridden methods or methods implemented from an interface.

Exam Essentials

Be able to identify the output of code using `instanceof`. `instanceof` checks if the left operand is the same class or interface (or a subclass) as the right operand. If the left operand is `null`, the result is `false`. If the two operands are not in the same class hierarchy, the code will not compile.

Recognize correct and incorrect implementations of equals(), hashCode(), and toString(). `public boolean equals(Object obj)` returns `false` when called with `null` or a class of the wrong type. `public int hashCode()` returns a number calculated with all or some of the instance variables used in `equals()`. `public String toString()` returns any `String`.

Be able to create enum classes. `enums` have a list of values. If that is all that is in the `enum`, the semicolon after the values is optional. `Enums` can have instance variables, constructors, and methods. The constructors are required to be private or package private. Methods are allowed to be on the `enum` top level or in the individual `enum` values. If the `enum` declares an `abstract` method, each `enum` value must implement it.

Identify and use nested classes. A member inner class is instantiated with code such as `outer.new Inner();`. Local inner classes are scoped to the end of the current block of code and not allowed to have `static` members. Anonymous inner classes are limited to extending a class or implementing one `interface`. A semicolon must end the statement creating an anonymous inner class. Static nested classes cannot access the enclosing class instance variables.

Know how to use imports and static imports. Classes can be imported by class name or wildcard. Wildcards do not look at subdirectories. In the event of a conflict, class name imports take precedence. Static imports import static members. They are written as `import static`, not static import. Make sure that they are importing static methods or variables rather than class names.

Understand the rules for method overriding and overloading. The Java compiler allows methods to be overridden in subclasses if certain rules are followed: a method must have the same signature, be at least as accessible as the parent method, must not declare any new or broader exceptions, and must use covariant return types. Methods are overloaded if they have the same method name but a different argument list. An overridden method may use the optional `@Override` annotation.

Review Questions

1. What is the result of the following code?

```
public class Employee {
   public int employeeId;
   public String firstName, lastName;
   public int yearStarted;
   @Override public int hashCode() {
      return employeeId;
   }
   public boolean equals(Employee e) {
      return this.employeeId == e.employeeId;
   }
   public static void main(String[] args) {
      Employee one = new Employee();
      one.employeeId = 101;
      Employee two = new Employee();
      two.employeeId = 101;
      if (one.equals(two)) System.out.println("Success");
      else System.out.println("Failure");
  } }
```

A. `Success`
B. `Failure`
C. The `hashCode()` method fails to compile.
D. The `equals()` method fails to compile.
E. Another line of code fails to compile.
F. A runtime exception is thrown.

2. What is the result of compiling the following class?

```
public class Book {
   private int ISBN;
   private String title, author;
   private int pageCount;
   public int hashCode() {
      return ISBN;
   }
   @Override public boolean equals(Object obj) {
     if ( !(obj instanceof Book)) {
         return false;
      }
      Book other = (Book) obj;
      return this.ISBN == other.ISBN;
   }
// imagine getters and setters are here
}
```

A. The code compiles.

B. The code does not compile because `hashCode()` is incorrect.

C. The code does not compile because `equals()` does not override the parent method correctly.

D. The code does not compile because `equals()` tries to refer to a private field.

E. The code does not compile because the `ClassCastException` is not handled or declared.

F. The code does not compile for another reason.

3. What is the result of the following code?

```
String s1 = "Canada";
String s2 = new String(s1);
if(s1 == s2) System.out.println("s1 == s2");
if(s1.equals(s2)) System.out.println("s1.equals(s2)");
```

A. There is no output.

B. `s1 == s2`

C. `s1.equals(s2)`

D. Both B and C.

E. The code does not compile.

F. The code throws a runtime exception.

4. What is true about the following code? You may assume `city` and `mascot` are never `null`.

```
public class BaseballTeam {
   private String city, mascot;
   private int numberOfPlayers;
   public boolean equals(Object obj) {
      if ( !(obj instanceof BaseballTeam))
         return false;
      BaseballTeam other = (BaseballTeam) obj;
      return (city.equals(other.city) && mascot.equals(other.mascot));
   }
   public int hashCode() {
      return numberOfPlayers;
   }
// imagine getters and setters are here
}
```

A. The class does not compile.
B. The class compiles but has an improper `equals()` method.
C. The class compiles but has an improper `hashCode()` method.
D. The class compiles and has proper `equals()` and `hashCode()` methods.

5. Which of the following statements are true, assuming `a` and `b` are `String` objects? (Choose all that apply.)

A. If `a.equals(b)` is `true`, `a.hashCode() == b.hashCode()` is always `true`.
B. If `a.equals(b)` is `true`, `a.hashCode() == b.hashCode()` is sometimes but not always `true`.
C. If `a.equals(b)` is `false`, `a.hashCode() == b.hashCode()` can never be `true`.
D. If `a.equals(b)` is `false`, `a.hashCode() == b.hashCode()` can sometimes be `true`.

6. What is the result of the following code?

```
public class FlavorsEnum {
   enum Flavors {
      VANILLA, CHOCOLATE, STRAWBERRY
   }
   public static void main(String[] args) {
      System.out.println(Flavors.CHOCOLATE.ordinal());
   }
}
```

A. `0`
B. `1`
C. `9`
D. `CHOCOLATE`
E. The code does not compile due to a missing semicolon.
F. The code does not compile for a different reason.

7. What is the result of the following code? (Choose all that apply.)

```
public class IceCream {
   enum Flavors {
      VANILLA, CHOCOLATE, STRAWBERRY
   }
   public static void main(String[] args) {
      Flavors f = Flavors.STRAWBERRY;
      switch (f) {
         case 0: System.out.println("vanilla");
         case 1: System.out.println("chocolate");
         case 2: System.out.println("strawberry");
              break;
         default: System.out.println("missing flavor");
     } } }
```

A. `vanilla`

B. `chocolate`

C. `strawberry`

D. `missing flavor`

E. The code does not compile.

F. An exception is thrown.

8. What is the result of the following code?

```
1:    public class Outer {
2:        private int x = 5;
3:        protected class Inner {
4:            public static int x = 10;
5:            public void go() { System.out.println(x); }
6:        }
7:        public static void main(String[] args) {
8:            Outer out = new Outer();
9:            Outer.Inner in = out.new Inner();
10:           in.go();
11:   } }
```

A. The output is `5`.

B. The output is `10`.

C. Line 4 generates a compiler error.

D. Line 8 generates a compiler error.

E. Line 9 generates a compiler error.

F. An exception is thrown.

9. What is the result of the following code?

```
1:    public class Outer {
2:    private int x = 24;
3:    public int getX() {
4:       String message = "x is ";
5:       class Inner {
6:          private int x = Outer.this.x;
7:          public void printX() {
8:             System.out.println(message + x);
9:          }
10:      }
11:      Inner in = new Inner();
12:      in.printX();
13:      return x;
14:   }
15:   public static void main(String[] args) {
16:      new Outer().getX();
17:   } }
```

A. `x` is `0`.

B. `x` is 24.

C. Line 6 generates a compiler error.

D. Line 8 generates a compiler error.

E. Line 11 generates a compiler error.

F. An exception is thrown.

10. The following code appears in a file named `Book.java`. What is the result of compiling the source file?

```
1:   public class Book {
2:      private int pageNumber;
3:      private class BookReader {
4:         public int getPage() {
5:             return pageNumber;
6:     } } }
```

A. The code compiles successfully, and one bytecode file is generated: `Book.class`.

B. The code compiles successfully, and two bytecode files are generated: `Book.class` and `BookReader.class`.

C. The code compiles successfully, and two bytecode files are generated: `Book.class` and `Book$BookReader.class`.

D. A compiler error occurs on line 3.

E. A compiler error occurs on line 5.

11. Which of the following statements can be inserted to make `FootballGame` compile?

```
package my.sports;
public class Football {
   public static final int TEAM_SIZE = 11;
}
package my.apps;
// INSERT CODE HERE
public class FootballGame {
   public int getTeamSize() { return TEAM_SIZE; }
}
```

A. `import my.sports.Football;`

B. `import static my.sports.*;`

C. `import static my.sports.Football;`

D. `import static my.sports.Football.*;`

E. `static import my.sports.*;`

F. `static import my.sports.Football;`

G. `static import my.sports.Football.*;`

12. What is the result of the following code?

```
public class Browsers {
   static class Browser {
      public void go() {
         System.out.println("Inside Browser");
      }
   }
   static class Firefox extends Browser {
      public void go() {
         System.out.println("Inside Firefox");
      }
   }
   static class IE extends Browser {
      @Override public void go() {
         System.out.println("Inside IE");
      }
   }
   public static void main(String[] args) {
      Browser b = new Firefox();
      IE e = (IE) b;
      e.go();
   }
}
```

A. `Inside Browser`
B. `Inside Firefox`
C. `Inside IE`
D. The code does not compile.
E. A runtime exception is thrown.

13. Which is a true statement about the following code?

```
public class IsItFurry {
   static interface Mammal { }
   static class Furry implements Mammal { }
   static class Chipmunk extends Furry { }
   public static void main(String[] args) {
      Chipmunk c = new Chipmunk();
      Mammal m = c;
      Furry f = c;
      int result = 0;
      if (c instanceof Mammal) result += 1;
      if (c instanceof Furry) result += 2;
      if (null instanceof Chipmunk) result += 4;
      System.out.println(result);
  } }
```

A. The output is `0`.
B. The output is `3`.
C. The output is `7`.
D. `c instanceof Mammal` does not compile.
E. `c instanceof Furry` does not compile.
F. `null instanceof Chipmunk` does not compile.

14. Which is a true statement about the following code? (Choose all that apply.)

```
import java.util. *;
public class IsItFurry {
   static class Chipmunk { }
   public static void main(String[] args) {
      Chipmunk c = new Chipmunk();
      ArrayList <Chipmunk> l = new ArrayList<>();
      Runnable r = new Thread();
      int result = 0;
      if (c instanceof Chipmunk) result += 1;
      if (l instanceof Chipmunk) result += 2;
      if (r instanceof Chipmunk) result += 4;
      System.out.println(result);
   } }
```

A. The code compiles, and the output is `0`.

B. The code compiles, and the output is `3`.

C. The code compiles, and the output is `7`.

D. `c instanceof Chipmunk` does not compile.

E. `l instanceof Chipmunk` does not compile.

F. `r instanceof Chipmunk` does not compile.

15. Which of the following statements are true about the `equals()` method? (Choose all that apply.)

A. If `equals(null)` is called, the method should throw an exception.

B. If `equals(null)` is called, the method should return `false`.

C. If `equals(null)` is called, the method should return `true`.

D. If `equals()` is passed the wrong type, the method should throw an exception.

E. If `equals()` is passed the wrong type, the method should return `false`.

F. If `equals()` is passed the wrong type, the method should return `true`.

16. Which of the following can be inserted in `main`?

```
public class Outer {
   class Inner { }

   public static void main(String[] args) {
      // INSERT CODE HERE
   } }
```

A. `Inner in = new Inner();`

B. `Inner in = Outer.new Inner();`

C. `Outer.Inner in = new Outer.Inner();`

D. `Outer.Inner in = new Outer().Inner();`

E. `Outer.Inner in = new Outer().new Inner();`

F. `Outer.Inner in = Outer.new Inner();`

17. What is the result of the following code? (Choose all that apply.)

```
1:    public enum AnimalClasses {
2:       MAMMAL(true), FISH(Boolean.FALSE), BIRD(false),
3:    REPTILE(false), AMPHIBIAN(false), INVERTEBRATE(false)
4:        boolean hasHair;
5:        public AnimalClasses(boolean hasHair) {
6:           this.hasHair = hasHair;
7:        }
8:        public boolean hasHair() {
9:           return hasHair;
10:       }
11:       public void giveWig() {
12:          hasHair = true;
13:       } }
```

A. Compiler error on line 2.
B. Compiler error on line 3.
C. Compiler error on line 5.
D. Compiler error on line 8.
E. Compiler error on line 12.
F. Compiler error on another line.
G. The code compiles successfully.

18. What is the result of the following code? (Choose all that apply.)

```
public class Swimmer {
   enum AnimalClasses {
      MAMMAL, FISH {
         public boolean hasFins() { return true; }},
      BIRD, REPTILE, AMPHIBIAN, INVERTEBRATE;
      public abstract boolean hasFins();
   }
   public static void main(String[] args) {
      System.out.println(AnimalClasses.FISH);
      System.out.println(AnimalClasses.FISH.ordinal());
      System.out.println(AnimalClasses.FISH.hasFins());
      System.out.println(AnimalClasses.BIRD.hasFins());
   }
}
```

A. `fish`
B. `FISH`
C. `0`
D. `1`

E. `false`

F. `true`

G. The code does not compile.

19. Which of the following can be inserted to override the superclass method? (Choose all that apply.)

```
public class LearnToWalk {
   public void toddle() {}
   class BabyRhino extends LearnToWalk {
      // INSERT CODE HERE
   }
}
```

A. `public void toddle() {}`

B. `public void Toddle() {}`

C. `public final void toddle() {}`

D. `public static void toddle() {}`

E. `public void toddle() throws Exception {}`

F. `public void toddle(boolean fall) {}`

20. What is the result of the following code?

```
public class FourLegged {
   String walk = "walk,";
   static class BabyRhino extends FourLegged {
      String walk = "toddle,";
   }
   public static void main(String[] args) {
      FourLegged f = new BabyRhino();
      BabyRhino b = new BabyRhino();
      System.out.println(f.walk);
      System.out.println(b.walk);
   } }
```

A. `toddle,toddle,`

B. `toddle,walk,`

C. `walk,toddle,`

D. `walk,walk,`

E. The code does not compile.

F. A runtime exception is thrown.

21. Which of the following could be inserted to fill in the blank? (Choose all that apply.)

```
public interface Otter {
   default void play() { }
}
class RiverOtter implements Otter {
  _____________________________
}
```

A. `@Override public boolean equals(Object o) { return false; }`
B. `@Override public boolean equals(Otter o) { return false; }`
C. `@Override public int hashCode() { return 42; }`
D. `@Override public long hashCode() { return 42; }`
E. `@Override public void play() { }`
F. `@Override void play() { }`