JAVA™ 2

SUN™ CERTIFIED PROGRAMMER & DEVELOPER

2

Declarations and Access Control

CERTIFICATION OBJECTIVES

- Declarations and Modifiers
- Declaration Rules
- Interface Implementation

✓ Two-Minute Drill

Q&A Self Test

We're on a roll. We've covered the fundamentals of keywords, primitives, arrays, and variables. Now it's time to drill deeper into rules for declaring classes, methods, and variables. We'll tackle access modifiers, abstract method implementation, interface implementation, and what you can and can't return from a method. Chapter 2 includes the topics asked most often on the exam, so you really need a solid grasp of this chapter's content. Grab your caffeine and let's get started.

CERTIFICATION OBJECTIVE

Declarations and Modifiers (Exam Objective 1.2)

Declare classes, nested classes, methods, instance variables, static variables, and automatic (method local) variables making appropriate use of all permitted modifiers (such as `public`, `final`, `static`, `abstract`, *and so forth). State the significance of each of these modifiers both singly and in combination, and state the effect of package relationships on declared items qualified by these modifiers.*

When you write code in Java, you're writing classes. Within those classes, as you know, are variables and methods (plus a few other things). *How* you declare your classes, methods, and variables dramatically affects your code's behavior. For example, a `public` method can be accessed from code running anywhere in your application. Mark that method `private`, though, and it vanishes from everyone's radar (except the class in which it was declared). For this objective, we'll study the ways in which you can modify (or not) a class, method, or variable declaration. You'll find that we cover modifiers in an extreme level of detail, and though we know you're already familiar with them, we're starting from the very beginning. Most Java programmers *think* they know how all the modifiers work, but on closer study often find out that they don't (at least not to the degree needed for the exam). Subtle distinctions are everywhere, so you need to be absolutely certain you're *completely* solid on everything in this objective before taking the exam.

Class Declarations and Modifiers

We'll start this objective by looking at how to declare and modify a class. Although nested (often called *inner*) classes are on the exam, we'll save nested class declarations for Chapter 8. You're going to love that chapter. No, *really*. Seriously. No kidding around.

Before we dig into class declarations, let's do a quick review of the rules:

- There can be only one `public` class per source code file.
- The name of the file must match the name of the `public` class.
- If the class is part of a package, the package statement must be the first line in the source code file.
- If there are import statements, they must go between the package statement and the class declaration. If there isn't a package statement, then the import statement(s) must be the first line(s) in the source code file. If there are no package or import statements, the class declaration must be the first line in the source code file. (Comments don't count; they can appear anywhere in the source code file.)
- Import and package statements apply to all classes within a source code file.

The following code is a bare-bones class declaration:

```
class MyClass { }
```

This code compiles just fine, but you can also add modifiers before the class declaration. Modifiers fall into two categories:

- Access modifiers: `public`, `protected`, `private`
- Nonaccess modifiers (including `strictfp`, `final`, and `abstract`)

We'll look at access modifiers first, so you'll learn how to restrict or allow access to a class you create. Access control in Java is a little tricky because there are *four* access *controls* (levels of access) but only *three* access *modifiers*. The fourth access control level (called *default* or *package* access) is what you get when you don't use any of the

three access modifiers. In other words, every class, method, and instance variable you declare has an access control, whether you explicitly type one or not. Although all four access controls (which means all three modifiers) work for most method and variable declarations, a class can be declared with only public or default access; the other two access control levels don't make sense for a class, as you'll see.

on the job

Java is a package-centric language; the developers assumed that for good organization and name scoping, you would put all your classes into packages. They were right, and you should. Imagine this nightmare: three different programmers, in the same company but working on different parts of a project, write a class named Utilities. If those three Utilities classes have not been declared in any explicit package, and are in the classpath, you won't have any way to tell the compiler or JVM which of the three you're trying to reference. Sun recommends that developers use reverse domain names, appended with division and/or project names. For example, if your domain name is geeksanonymous.com, and you're working on the client code for the TwelvePointOSteps program, you would name your package something like com.geeksanonymous.steps.client. That would essentially change the name of your class to `com.geeksanonymous.steps.client.Utilities`***. You might still have name collisions within your company, if you don't come up with your own naming schemes, but you're guaranteed not to collide with classes developed outside your company (assuming they follow Sun's naming convention, and if they don't, well, Really Bad Things could happen).***

Class Access

What does it mean to access a class? When we say code from one class (class A) has access to another class (class B), it means class A can do one of three things:

- Create an instance of class B
- Extend class B (in other words, become a subclass of class B)
- Access certain methods and variables within class B, depending on the access control of those methods and variables.

In effect, access means *visibility*. If class A can't see class B, the access level of the methods and variables *within* class B won't matter; class A won't have any way to access those methods and variables.

Default Access A class with *default* access has no modifier preceding it in the declaration. In other words, it's the access control you get when you don't type a modifier in the class declaration. Think of default access as package-level access, because *a class with default access can be seen only by classes within the same package*. For example, if class A and class B are in different packages, and class A has default access, class B won't be able to create an instance of class A, or even declare a variable or return type of class A. In fact, class B has to pretend that class A doesn't even exist, or the compiler will complain. Look at the following source file:

```
package cert;
class Beverage {
 }
```

Now look at the second source file:

```
package exam.stuff;
import cert.Beverage;
class Tea extends Beverage {
 }
```

As you can see, the superclass (Beverage) is in a different package from the subclass (Tea). The import statement at the top of the Tea file is trying (fingers crossed) to import the Beverage class. The Beverage file compiles fine, but watch what happens when we try to compile the Tea file:

```
 >javac Tea.java
Tea.java:1: Can't access class cert.Beverage. Class or
interface must be public, in same package, or an accessible member
class.
import cert.Beverage;
..
```

Tea won't compile because its superclass, Beverage, has default access and is in a different package. You can do one of two things to make this work. You could put both classes in the same package, or declare Beverage as `public`, as the next section describes.

exam Watch

When you see a question with complex logic, be sure to look at the access modifiers first. That way, if you spot an access violation (for example, a class in package A trying to access a default class in package B), you'll know the code won't compile so you don't have to bother working through the logic. It's not as if, you know, you don't have anything better to do with your time while taking the exam. Just choose the "Compilation fails" answer and zoom on to the next question.

Public Access A class declaration with the `public` keyword gives all classes from all packages access to the public class. In other words, *all* classes in the Java Universe (JU) (you'll be tested on this acronym) have access to a public class. Don't forget, though, that if a public class you're trying to use is in a different package from the class you're writing, you'll still need to import the public class. (Just kidding about the JU acronym. We just made that up to keep you on your toes.)

In the example from the preceding section, we may not want to place the subclass in the same package as the superclass. To make the code work, we need to add the keyword `public` in front of the superclass (Beverage) declaration, as follows:

```
package cert;
public class Beverage {
 }
```

This changes the Beverage class so it will be visible to all classes in all packages. The class can now be instantiated from all other classes, and any class is now free to subclass (extend from) it—*unless*, that is, the class is also marked with the nonaccess modifier `final`. Read on.

Other (Nonaccess) Class Modifiers

You can modify a class declaration using the keyword `final`, `abstract`, or `strictfp`. These modifiers are in addition to whatever access control is on the class, so you could, for example, declare a class as both `public` *and* `final`. But you can't *always* mix nonabstract modifiers. You're free to use `strictfp` in combination with `abstract` or `final`, but you must never, ever, *ever* mark a class as both `final` *and* `abstract`. You'll see why in the next two sections.

You won't need to know how `strictfp` works, so we're focusing only on modifying a class as `final` or `abstract`. For the exam, you need to know only that `strictfp` is a keyword and can be used to modify a class or a method, but never a variable. Marking a class as `strictfp` means that any method code in the

class will conform to the IEEE754 standard rules for floating points. Without that modifier, floating points used in the methods might behave in a platform-dependent way. If you don't declare a class as `strictfp`, you can still get `strictfp` behavior on a method-by-method basis, by declaring a *method* as `strictfp`. If you don't know the IEEE754 standard, now's not the time to learn it. You have, as we say, *bigger fish to fry*.

Final Classes When used in a class declaration, the `final` keyword means the class can't be subclassed. In other words, no other class can ever *extend* (inherit from) a `final` class, and any attempts to do so will give you a compiler error.

So why would you ever mark a class `final`? After all, doesn't that violate the whole OO notion of inheritance? You should make a final class *only* if you need an absolute guarantee that *none* of the methods in that class will ever be overridden. If you're deeply dependent on the implementations of certain methods, then using `final` gives you the security that nobody can change the implementation out from under you.

You'll notice many classes in the Java core libraries are final. For example, the String class cannot be subclassed. Imagine the havoc if you couldn't guarantee how a String object would work on any given system your application is running on! If programmers were free to extend the String class (and thus substitute their new String subclass instances where `java.lang.String` instances are expected), civilization—as we know it—could collapse. So use `final` for safety, but *only* when you're certain that your final class has indeed said all that ever needs to be said in its methods. Marking a class final means, in essence, your class can't ever be improved upon, or even specialized, by another programmer.

Another benefit of having nonfinal classes is this scenario: imagine you find a problem with a method in a class you're using, but you don't have the source code. So you can't modify the source to improve the method, but you *can* extend the class and override the method in your new subclass, and substitute the subclass everywhere the original superclass is expected. If the class is final, though, then you're stuck.

Let's modify our Beverage example by placing the keyword `final` in the declaration:

```
package cert;
public final class Beverage{
   public void importantMethod() {
   }
}
```

Now, if we try to compile the Tea subclass:

```
package exam.stuff;
import cert.Beverage;
class Tea extends Beverage {
 }
```

We get the following error:

```
>javac Tea.java
Tea.java:3: Can't subclass final classes: class
cert.Beverage class Tea extends Beverage{
1 error
```

In practice, you'll almost never make a final class. A final class obliterates a key benefit of OO—extensibility. So unless you have a serious safety or security issue, assume that some day another programmer will need to extend your class. If you don't, the next programmer forced to maintain your code will hunt you down and <insert really scary thing>.

Abstract Classes An *abstract class* can never be instantiated. Its sole purpose, mission in life, *raison d'être*, is to be extended (subclassed). Why make a class if you can't make objects out of it? Because the class might be just too, well, *abstract*. For example, imagine you have a class Car that has generic methods common to all vehicles. But you don't want anyone actually *creating* a generic, abstract Car object. How would they initialize its state? What color would it be? How many seats? Horsepower? All-wheel drive? Or more importantly, *how would it behave*? In other words, how would the methods be implemented?

No, you need programmers to instantiate *actual* car types such as SubaruOutback, BMWBoxster, and the like, and we'll bet the Boxster owner will tell you his car does things the Subaru can do "only in its dreams!" Take a look at the following abstract class:

```
abstract class Car {
   private double price;
   private Color carColor;
   private String model;
   private String year;
   public abstract void goFast();
   public abstract void goUpHill();
```

```
    public abstract void impressNeighbors();
    // Additional, important, and serious code goes here
}
```

The preceding code will compile fine. However, if you try to instantiate a Car in another body of code, you'll get a compiler error:

```
AnotherClass.java:7: class Car is an abstract
class. It can't be instantiated.
      Car x = new Car();
1 error
```

Notice that the methods marked `abstract` end in a semicolon rather than curly braces.

exam Watch

***Look for questions with a method declaration that ends with a semicolon, rather than curly braces. If the method is in a class—as opposed to an interface—then both the method and the class** must **be marked** `abstract`**. You might get a question that asks how you could fix a code sample that includes a method ending in a semicolon, but without an** `abstract` **modifier on the class or method. In that case, you could either mark the method and class** `abstract`**, or remove the** `abstract` **modifier from the method. Oh, and if you change a method from abstract to nonabstract,** don't forget to change the semicolon at the end of the method declaration into a curly brace pair!*

We'll look at abstract *methods* in more detail later in this objective, but always remember that *if even a single method is abstract, the whole class must be declared* `abstract`. One abstract method spoils the whole bunch. You can, however, put nonabstract methods in an abstract class. For example, you might have methods with implementations that shouldn't change from car type to car type, such as `getColor()` or `setPrice()`. By putting nonabstract methods in an abstract class, you give all concrete subclasses (concrete just means *not abstract*) inherited method implementations. The good news there is that concrete subclasses get to inherit functionality, and need to implement only the methods that define subclass-specific behavior.

(By the way, if you think we misused *raison d'être*, for gosh sakes don't send an email. We're rather pleased with ourselves, and let's see *you* work it into a programmer certification book.)

on the job

Coding with abstract class types (including interfaces, discussed later in this chapter) let's you take advantage of polymorphism, and gives you the greatest degree of flexibility and extensibility. You'll learn more about polymorphism in Chapter 5.

exam watch

***You can't mark a class as both abstract and final. They have nearly opposite meanings. An abstract class** must **be subclassed, whereas a final class** must not **be subclassed. If you see this combination of `abstract` and `final` modifiers, used for a class or method declaration,** the code will not compile.*

EXERCISE 2-1

Creating an Abstract Superclass and Concrete Subclass

The following exercise will test your knowledge of public, default, final, and abstract classes. Create an abstract superclass named Fruit and a concrete subclass named Apple. The superclass should belong to a package called *food* and the subclass can belong to the default package (meaning it isn't put into a package explicitly). Make the superclass `public` and give the subclass default access.

1. Create the superclass as follows:

```
package food;
public abstract class Fruit{ /* any code you want */}
```

2. Create the subclass in a separate file as follows:

```
import food.Fruit;
class Apple extends Fruit{ /* any code you want */}
```

3. Create a directory called *food* off the directory in your class path setting.
4. Attempt to compile the two files. If you want to use the Apple class, make sure you place the Fruit.class file in the food subdirectory.

Method and Variable Declarations and Modifiers

We've looked at what it means to use a modifier in a *class* declaration, and now we'll look at what it means to modify a *method* or *variable* declaration.

Methods and instance (nonlocal) variables are collectively known as *members*. You can modify a member with both access and nonaccess modifiers, and you have more modifiers to choose from (and combine) than when you're declaring a class.

Member Access

Because method and variable members are usually given access control in exactly the same way, we'll cover both in this section.

Whereas a class can use just two of the four access control levels (`default` or `public`), members can use all four:

- public
- protected
- *default*
- private

Default protection is what you get when you don't type an access modifier in the member declaration. The default and protected access control types have almost identical behavior, except for one difference that will be mentioned later.

exam Watch

It's crucial that you know access control inside and out for the exam. There will be quite a few questions with access control playing a role. Some questions test several concepts of access control at the same time, so not knowing one small part of access control could blow an entire question.

What does it mean for code in one class to have access to a *member* of another class? For now, ignore any differences between methods and variables. If class A has access to a member of class B, it means that class B's member is *visible* to class A. When a class does *not* have access to another member, the compiler will slap you for trying to access something that you're not even supposed to know exists!

You need to understand two different access issues:

- Whether method code in one class can access a member of another class
- Whether a subclass can inherit a member of its superclass

The first type of access is when a method in one class tries to access a method or a variable of another class, using the dot operator (.) to invoke a method or retrieve a variable. For example,

```
class Zoo {
   public String coolMethod() {
      return "Wow  baby";
  }
}

class Moo {
   public void useAZoo() {
      Zoo z = new Zoo();
      // If the preceding line compiles Moo has access
      // to the Zoo class
      // But… does it have access to the coolMethod()?

      System.out.println("A Zoo says, " + z.coolMethod());
     // The preceding line works because Moo can access the
     // public method
   }
}
```

The second type of access revolves around which, if any, members of a superclass a subclass can access *through inheritance*. We're not looking at whether the subclass can, say, invoke a method on an instance of the superclass (which would just be an example of the first type of access). Instead, we're looking at whether the subclass *inherits* a member of its superclass. Remember, if a subclass inherits a member, it's exactly as if the subclass actually declared the member itself. In other words, if a subclass *inherits* a member, the subclass *has* the member.

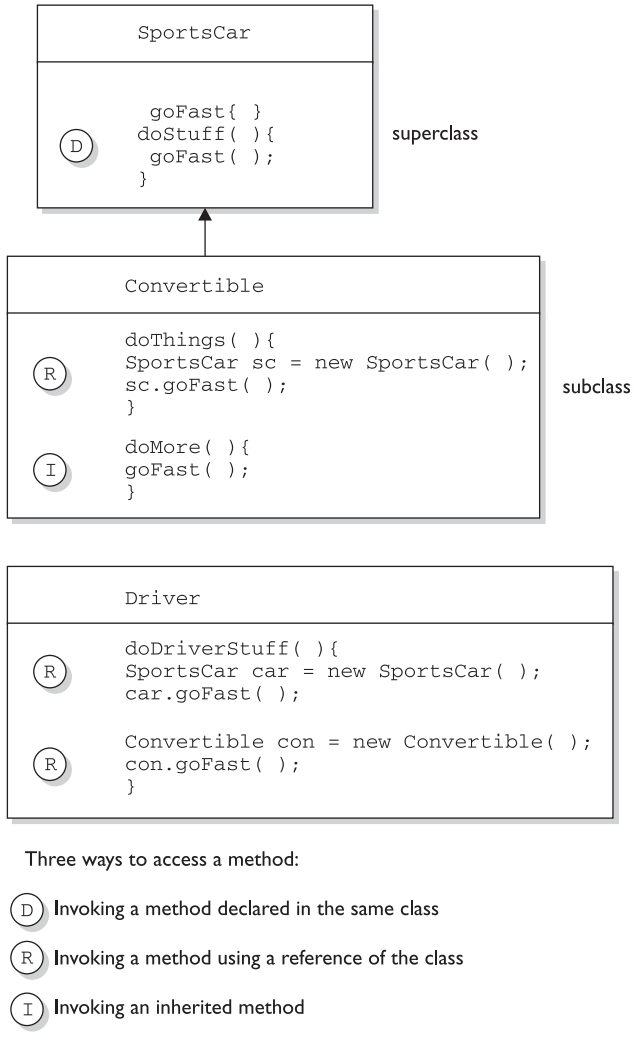
```
class Zoo {
   public String coolMethod() {
      return "Wow  baby";
  }
}

class Moo extends Zoo {
   public void useMyCoolMethod() {
      // Does an instance of Moo inherit the coolMethod()?
     System.out.println("Moo says, " + this.coolMethod());
     // The preceding line works because Moo can inherit the public method

     // Can an instance of Moo invoke coolMethod() on an instance of Zoo?
     Zoo z = new Zoo();
     System.out.println("Zoo says, " + z.coolMethod());
     // coolMethod() is public, so Moo can invoke it on a Zoo reference

   }
}
```

Figure 2-1 compares the effect of access modifiers on whether a class can inherit a member of another class, or access a member of another class using a reference of an instance of that class.

Much of access control (both types) centers on whether the two classes involved are in the same or different packages. Don't forget, though, if class A *itself* can't be accessed by class B, then no *members* within class A can be accessed by class B.

FIGURE 2-1

Comparison of inheritance vs. dot operator for member access



exam Watch

You need to know the effect of different combinations of class and member access (such as a default class with a public variable). To figure this out, first look at the access level of the class. If the class itself will not be visible to another class, then none of the members will be either, even if the member is declared `public`. Once you've confirmed that the class is visible, then it makes sense to look at access levels on individual members.

Public Members When a method or variable member is declared `public`, it means all other classes, regardless of the package they belong to, can access the member (assuming the class itself is visible). Look at the following source file:

```
package book;
import cert.*;  // Import all classes in the cert package
class Goo {
   public static void main(String [] args) {
      Sludge o = new Sludge();
      o.testIt();
   }
}
```

Now look at the second file:

```
package cert;
public class Sludge {
   public void testIt() {
      System.out.println("sludge");
   }
}
```

As you can see, Goo and Sludge are in different packages. However, Goo can invoke the method in Sludge without problems because both the Sludge class and its `testIt()` method are marked `public`.

For a subclass, if a member of its superclass is declared `public`, the subclass inherits that member *regardless of whether both classes are in the same package*. Read the following code:

```
package cert;
public class Roo {
   public String doRooThings() {
    // imagine the fun code that goes here
   }
}
```

The Roo class declares the `doRooThings()` member as `public`. So if we make a subclass of Roo, any code in that Roo subclass can call its own inherited `doRooThings()` method.

```
package notcert; //Not the package Roo is in
import cert.Roo;
class  Cloo extends Roo {
   public void testCloo() {
      System.out.println(doRooThings());
   }
}
```

Notice in the preceding code that the `doRooThings()` method is invoked without having to preface it with a reference. Remember, if you see a method invoked (or a variable accessed) without the dot operator (.), it means the method or variable belongs to the class where you see that code. It also means that the method or variable is implicitly being accessed using the `this` reference. So in the preceding code, the call to `doRooThings()` in the Cloo class could also have been written as *`this`*`.doRooThings()`. The reference `this` always refers to the currently executing object—in other words, the object running the code where you see the `this` reference. Because the `this` reference is implicit, you don't need to preface your member access code with it, but it won't hurt. Some programmers include it to make the code easier to read for new (or non) java programmers.

Besides being able to invoke the `doRooThings()` method on itself, code from some *other* class can call `doRooThings()` on a Cloo instance, as in the following:

```
class Toon {
   public static void main (String [] args) {
      Cloo c = new Cloo();
      System.out.println(c.doRooThings()); //No problem; method is public
   }
}
```

Private Members Members marked `private` can't be accessed by code in any class other than the class in which the private member was declared. Let's make a small change to the Roo class from an earlier example.

```
package cert;
public class Roo {
   private String doRooThings() {
    // imagine the fun code that goes here, but only the Roo class knows
   }
}
```

The `doRooThings()` method is now private, so no other class can use it. If we try to invoke the method from any other class, we'll run into trouble.

```
package notcert;
import cert.Roo;
class  UseARoo {
   public void testIt() {
      Roo r = new Roo(); //So far so good; class Roo is still public
      System.out.println(r.doRooThings()); //Compiler error!
   }
}
```

If we try to compile the `UseARoo` class, we get the following compiler error:

```
%javac Balloon.java
Balloon.java:6: No method matching doRooThings() found in class
cert.Roo.
      r.doRooThings();
1 error
```

It's as if the method `doRooThings()` doesn't exist, and as far as any code outside of the Roo class is concerned, it's true. *A private member is invisible to any code outside the member's own class.*

What about a subclass that tries to *inherit* a private member of its superclass? When a member is declared private, a subclass can't inherit it. For the exam, you need to recognize that a subclass can't see, use, or even *think about* the private members of its superclass. You can, however, declare a matching method in the subclass. But regardless of how it looks, it is *not* an overriding method! It is simply a method that happens to have the same name as a private method (which you're not supposed to know about) in the superclass. The rules of overriding do not apply, so you can make this newly-declared-but-just-happens-to-match method declare new exceptions, or change the return type, or anything else you want to do with it.

```
package cert;
public class Roo {
   private String doRooThings() {
    // imagine the fun code that goes here, but no other class will know
   }
}
```

The `doRooThings()` method is now off limits to all subclasses, even those in the same package as the superclass.

```
package cert;  //Cloo and Roo are in the same package
class  Cloo extends Roo {  //Still OK, superclass Roo is public
   public void testCloo() {
      System.out.println(doRooThings()); //Compiler error!
   }
}
```

If we try to compile the subclass Cloo, the compiler is delighted to spit out the following error:

```
%javac Cloo.java
Cloo.java:4: Undefined method: doRooThings()
      System.out.println(doRooThings());
1 error
```

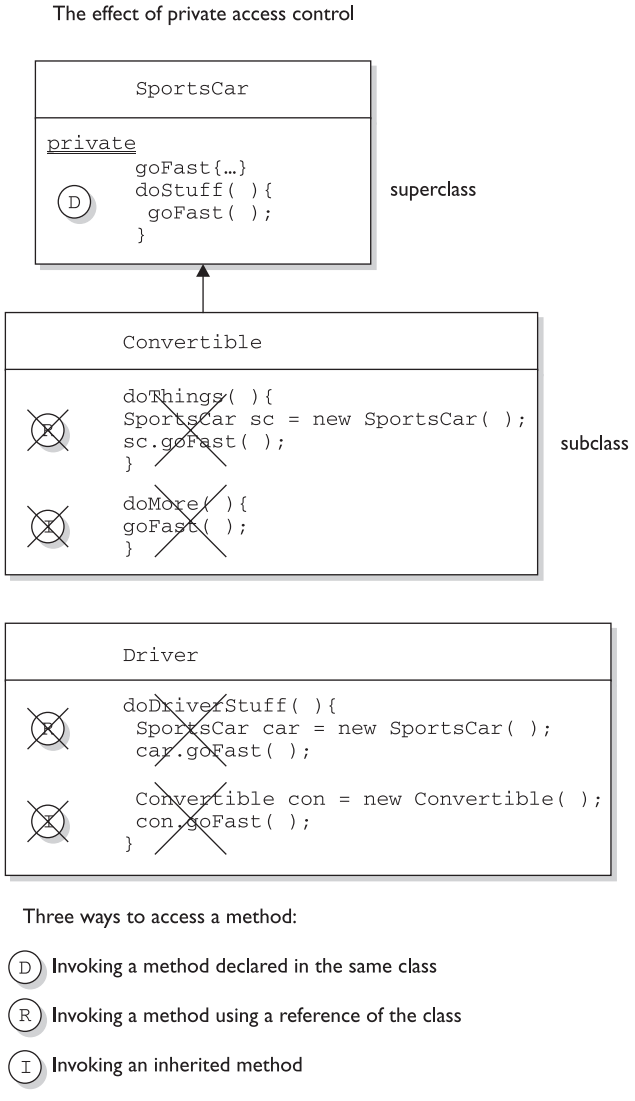
Although you're allowed to mark instance variables as `public`, in practice it's nearly always best to keep all variables `private` or `protected`. If variables need to be changed, set, or read, programmers should use public accessor methods, so that code in any other class has to ask to get or set a variable (by going through a method), rather than access it directly. Accessor methods should usually take the form `get<propertyName>` and `set<propertyName>`, and provide a place to check and/or validate before returning or modifying a value. Without this protection, the weight variable of a Cat object, for example, could be set to a negative number if the offending code goes straight to the public variable as in `someCat.weight = -20`. But an accessor method, `setWeight(int wt)`, could check for an inappropriate number. (OK, wild speculation, but we're guessing a negative weight might be inappropriate for a cat. And no wisecracks from you cat haters.) Chapter 5 will discuss this data protection (encapsulation) in more detail.

Can a private method be overridden by a subclass? That's an interesting question, but the answer is *technically* no. Since the subclass, as we've seen, cannot *inherit* a private method, it therefore cannot *override* the method—overriding *depends* on inheritance. We'll cover the implications of this in more detail a little later in this section as well as in Chapter 5, but for now just remember that a method marked `private` cannot be overridden. Figure 2-2 illustrates the effects of the public and private access modifiers on classes from the same or different packages.

Protected and Default Members The protected and default access control levels are almost identical, but with one critical difference. A default member may

FIGURE 2-2

The effects of public and private access



be accessed only if the class accessing the member belongs to the same package, whereas a protected member can be accessed (through inheritance) by a subclass even if the subclass is in a different package. Take a look at the following two classes:

```
package certification;
public class OtherClass {
```

```
   void testIt() {   // No modifier means method has default access
      System.out.println("OtherClass");
   }
}
```

In another source code file you have the following:

```
package somethingElse;
import certification.OtherClass;
class AccessClass {
   static public void main(String [] args) {
      OtherClass o = new OtherClass();
      o.testIt();
   }
}
```

As you can see, the `testIt()` method in the second file has default (think: *package-level*) access. Notice also that class OtherClass is in a different package from the AccessClass. Will AccessClass be able to use the method `testIt()`? Will it cause a compiler error? Will Daniel ever marry Francesca? Stay tuned.

```
%javac AccessClass.java
AccessClass.java:5: No method matching testIt() found in class
certification.OtherClass.
      o.testIt();
1 error
```

From the preceding results, you can see that AccessClass can't use the OtherClass method `testIt()` because `testIt()` has default access, and AccessClass is not in the same package as OtherClass. So AccessClass can't see it, the compiler complains, and we have no idea who Daniel and Francesca are.

Default and protected behavior differ *only* when we talk about subclasses. This difference is not often used in actual practice, but that doesn't mean it won't be on the exam! Let's look at the distinctions between protected and default access.

If the `protected` keyword is used to define a member, any subclass of the class declaring the member can access it. It doesn't matter if the superclass and subclass are in different packages, the protected superclass member is still visible to the subclass (although visible only in a very specific way as we'll see a little later). This is in contrast to the default behavior, which doesn't allow a subclass to access a superclass member unless the subclass is in the same package as the superclass.

Whereas default access doesn't extend any *special* consideration to subclasses (you're either in the package or you're not), the `protected` modifier respects the parent-child relationship, even when the child class moves away (and joins a new package). So, when you think of default access, think package restriction. *No exceptions.* But when you think `protected`, think *package + kids.* A class with a protected member is marking that member as having package-level access for all classes, but with a special exception for subclasses outside the package.

But what does it mean for a subclass-outside-the-package to have access (visibility) to a superclass (parent) member? It means the subclass *inherits* the member. It does not, however, mean the subclass-outside-the-package can access the member using a reference to an instance of the superclass. In other words, protected = inheritance. Protected does *not* mean that the subclass can treat the protected superclass member as though it were public. So if the subclass-outside-the-package gets a reference to the superclass (by, for example, creating an instance of the superclass somewhere in the subclass' code), the subclass *cannot* use the dot operator on the superclass reference to access the protected member. To a subclass-outside-the-package, a protected member might as well be default (or even private), when the subclass is using a reference to the superclass. *The subclass can* only *see the protected member through inheritance.*

Are you confused? So are we. Hang in there and it will all become clear with the next batch of code examples. (And don't worry; we're not *actually* confused. We're just trying to make you feel better if *you* are. You know, like it's *OK* for you to feel as though nothing makes sense, and that it isn't your fault. *Or is it?* <insert evil laugh>)

Let's take a look at a protected instance variable (remember, an instance variable is a member) of a superclass.

```
package certification;
public class Parent {
   protected int x = 9; // protected access
}
```

The preceding code declares the variable *x* as `protected`. This makes the variable accessible to all other classes in the certification package, as well as inheritable by any subclasses outside the package. Now let's create a subclass in a different package, and attempt to use the variable *x* (that the subclass inherits).

```
package other; // Different package
import certification.Parent;
class Child extends Parent {
```

```
   public void testIt() {
      System.out.println("x is " + x); // No problem; Child inherits x
   }
}
```

The preceding code compiles fine. Notice, though, that the Child class is accessing the protected variable *through inheritance*. Remember, anytime we talk about a subclass having *access* to a superclass member, we could be talking about the subclass *inheriting* the member, not simply accessing the member through a reference to an instance of the superclass (the way any other nonsubclass would access it). Watch what happens if the subclass Child (outside the superclass' package) tries to access a protected variable using a Parent class reference.

```
package other;
import certification.Parent;
class Child extends Parent {
   public void testIt() {
      System.out.println("x is " + x); // No problem; Child inherits x
      Parent p = new Parent(); // Can we access x using the p reference?
      System.out.println("X in parent is " + p.x); // Compiler error!
   }
}
```

The compiler is more than happy to show us the problem:

```
%javac -d . other/Child.java
other/Child.java:9: x has protected access in certification.Parent
System.out.println("X in parent is " + p.x);
                                         ^
1 error
```

So far we've established that a protected member has essentially package-level or *default* access to all classes except for subclasses. We've seen that subclasses outside the package can inherit a protected member. Finally, we've seen that subclasses outside the package can't use a superclass reference to access a protected member. For a subclass outside the package, *the protected member can be accessed only through inheritance*.

But there's still one more issue we haven't looked at…what does a protected member look like to *other* classes trying to use the subclass-outside-the-package to get to the subclass' inherited protected superclass member? For example, using our previous Parent/Child classes, what happens if some other class—Neighbor, say—in the same package as the Child (subclass), has a reference to a Child instance and

wants to access the member variable *x*? In other words, how does that protected member behave once the subclass has inherited it? Does it maintain its protected status, such that classes in the Child's package can see it?

No! Once the subclass-outside-the-package inherits the protected member, that member (as inherited by the subclass) becomes private to any code outside the subclass. So if class Neighbor instantiates a Child object, then even if class Neighbor is in the same package as class Child, class Neighbor won't have access to the Child's inherited (but protected) variable *x*. The bottom line: when a subclass-outside-the-package inherits a protected member, the member is essentially private inside the subclass, such that only the subclass' own code can access it. Figure 2-3 illustrates the effect of protected access on classes and subclasses in the same or different packages.

Whew! That wraps up `protected`, the most misunderstood modifier in Java. Again, it's used only in very special cases, but you can count on it showing up on the exam. Now that we've covered the `protected` modifier, we'll switch to default member access, a piece of cake compared to `protected`.

Let's start with the default behavior of a member in a superclass. We'll modify the Parent's member *x* to make it default.

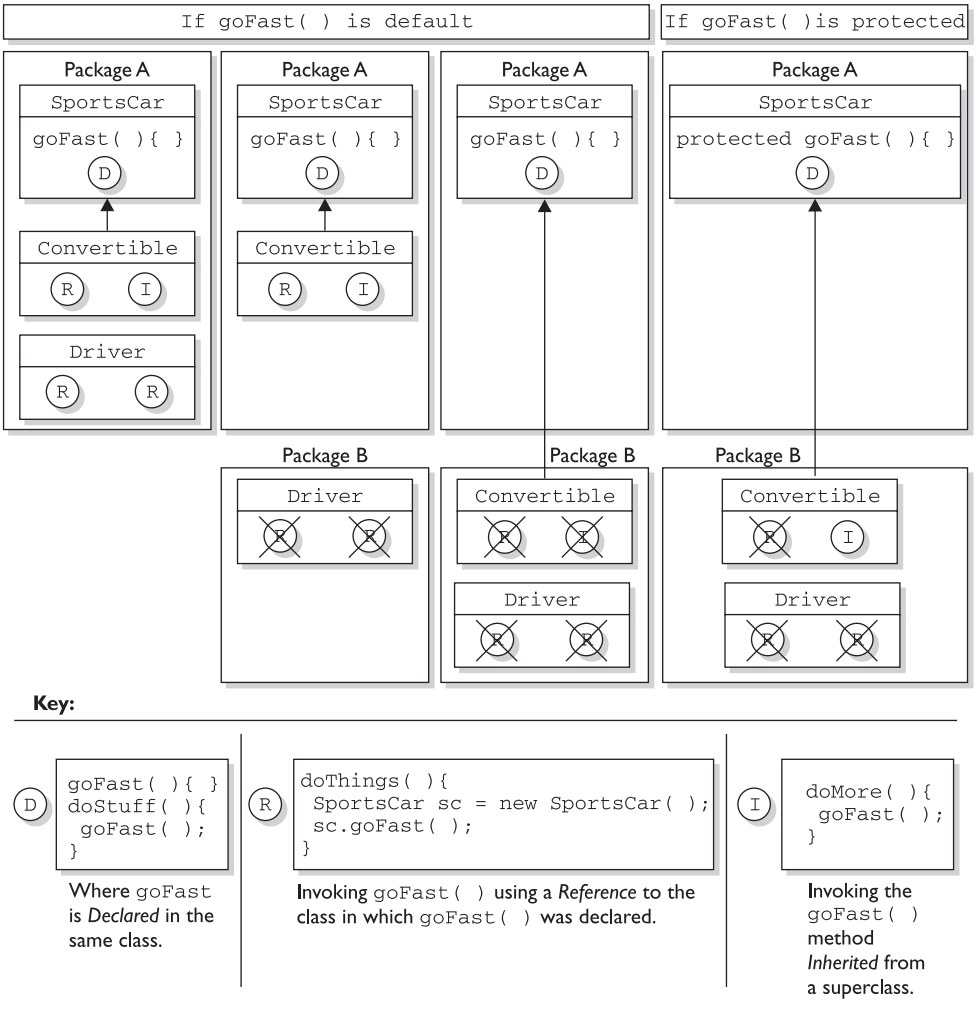
```
package certification;
public class Parent {
   int x = 9; // No access modifier, means default (package) access
}
```

Notice we didn't place an access modifier in front of the variable *x*. Remember that if you don't type an access modifier before a class or member declaration, the access control is default, which means *package level*. We'll now attempt to access the default member from the Child class that we saw earlier. When we compile the Child file, we get the following error:

```
%javac Child.java
Child.java:4: Undefined variable: x
      System.out.println("Variable x is " + x);
1 error
```

The compiler gives the same error as when a member is declared as `private`. The subclass Child (in a different package from the superclass Parent) can't see or

FIGURE 2-3 The effects of protected access



use the default superclass member *x*! Now, what about default access for two classes in the same package?

```
package certification;
public class Parent{
   int x = 9; // default access
}
```

And in the second class you have the following:

```
package certification;
class Child extends Parent{
   static public void main(String [] args) {
      Child sc = new Child();
      sc.testIt();
   }
   public void testIt() {
      System.out.println("Variable x is " + x); // No problem;
   }
}
```

The preceding source file compiles fine, and the class Child runs and displays the value of *x*. Just remember that default members are visible only to the subclasses that are in the same package as the superclass.

Local Variables and Access Modifiers Can access modifiers be applied to local variables? This one should be simple to remember: *NO!*

exam Watch

There is never a case where an access modifier can be applied to a local variable, so watch out for code like the following:

```
class Foo {
   void doStuff() {
      private int x = 7;
     this.doMore(x);
   }
}
```

You can be certain that any local variable declared with an access modifier will not compile. In fact, there is only one modifier that can ever be applied to local variables—`final`.

That about does it for our discussion on member *access* modifiers. Table 2-1 shows all the combinations of access and visibility; you really should spend some time with it. Next, we're going to dig into the other (nonaccess) modifiers that you can apply to member declarations.

TABLE 2-1 Determining Access to Class Members

Visibility	Public	Protected	*Default*	Private
From the same class	Yes	Yes	Yes	Yes
From any class in the same package	Yes	Yes	Yes	No
From any non-subclass class outside the package	Yes	No	No	No
From a subclass in the same package	Yes	Yes	Yes	No
From a subclass outside the same package	Yes	Yes	No	No

Nonaccess Member Modifiers

We've discussed member *access*, which refers to whether or not code from one class can invoke a method (or access an instance variable) from another class. That still leaves a boatload of *other* modifiers you can use on member declarations. Two you're already familiar with—`final` and `abstract`—because we applied them to class declarations earlier in this chapter. But we still have to take a quick look at `transient`, `synchronized`, `native`, `strictfp`, and then a long look at the Big One—`static`. We'll look first at modifiers applied to *methods*, followed by a look at modifiers applied to *instance variables*. We'll wrap up this objective with a look at how `static` works when applied to variables and methods.

Final Methods The `final` keyword prevents a method from being overridden in a subclass, and is often used to enforce the API functionality of a method. For example, the Thread class has a method called `isAlive()` that checks whether a thread is still active. If you extend the Thread class, though, there is really no way that you can correctly implement this method yourself (it uses native code, for one thing), so the designers have made it final. Just as you can't subclass the String class (because we need to be able to trust in the behavior of a String object), you can't override many of the methods in the core class libraries. This can't-be-overridden restriction provides for safety and security, but you should use it with great caution. Preventing a subclass from overriding a method stifles many of the benefits of OO including extensibility through polymorphism.

A typical final method declaration looks like this:

```
class SuperClass{
   public final void showSample() {
      System.out.println("One thing.");
   }
}
```

It's legal to extend SuperClass, since the class itself isn't marked `final`, but we can't override the final method `showSample()`, as the following code attempts to do:

```
class SubClass extends SuperClass{
   public void showSample() { // Try to override the final superclass method
      System.out.println("Another thing.");
   }
}
```

Attempting to compile the preceding code gives us the following:

```
%javac FinalTest.java
FinalTest.java:5: The method void showSample() declared in class
SubClass cannot override the final method of the same signature
declared in class SuperClass. Final methods cannot be overridden.
     public void showSample() { }
1 error
```

Final Arguments Method arguments are the variable declarations that appear in between the parentheses in a method declaration. A typical method declaration with multiple arguments looks like this:

```
public Record getRecord(int fileNumber, int recordNumber) {}
```

Method arguments are essentially the same as local variables. In the preceding example, the variables *fileNumber* and *recordNumber* will both follow all the rules applied to local variables. This means they can also have the modifier `final`:

```
public Record getRecord(int fileNumber, final int recordNumber) {}
```

In this example, the variable *recordNumber* is declared as `final`, which of course means it can't be modified within the method. In this case, "modified" means reassigning a new value to the variable. In other words, a final argument must keep the same value that the parameter had when it was passed into the method.

Abstract Methods An *abstract method* is a method that's been declared (as `abstract`) but not implemented. In other words, the method contains no functional code. And if you recall from the previous section on abstract classes, an abstract method declaration doesn't even have curly braces for where the implementation code goes, but instead closes with a semicolon. You mark a method `abstract` when you want to force subclasses to provide the implementation. For example, if you write an abstract class Car with a method `goUpHill()`, you might want to force each subtype of car to define its own `goUpHill()` behavior, specific to that particular type of car. (If you've ever lived in the Rockies, you know that the differences in how cars go uphill (or *fail* to) is not, um, subtle.)

A typical abstract method declaration is as follows:

```
public abstract void showSample();
```

Notice that the abstract method ends with a semicolon instead of curly braces. It is illegal to have an abstract method in a class that is not declared abstract. Look at the following illegal class:

```
public class IllegalClass{
   public abstract void doIt();
}
```

The preceding class will produce the following error if you try to compile it:

```
%javac IllegalClass.java
IllegalClass.java:1: class IllegalClass must be declared abstract.
It does not define void doIt() from class IllegalClass.
public class IllegalClass{
1 error
```

You can, however, have an abstract class with no abstract methods. The following example will compile fine:

```
public abstract class LegalClass{
   void goodMethod() {
      // lots of real implementation code here
   }
 }
```

In the preceding example, `goodMethod()` is not abstract. Three different clues tell you it's not an abstract method:

- The method is not marked abstract.
- The method declaration includes curly braces, as opposed to ending in a semicolon.
- The method provides actual implementation code.

Any class that extends an abstract class *must* implement all abstract methods of the superclass. *Unless the subclass is also abstract.* The rule is

> *The first concrete subclass of an abstract class must implement all abstract methods of the superclass.*

Concrete just means nonabstract, so if you have an abstract class extending another abstract class, the abstract subclass doesn't need to provide implementations for the inherited abstract methods. Sooner or later, though, somebody's going to make a nonabstract subclass (in other words, a class that can be instantiated), and that subclass will have to implement all the abstract methods from up the inheritance tree. The following example demonstrates an inheritance tree with two abstract classes and one concrete class:

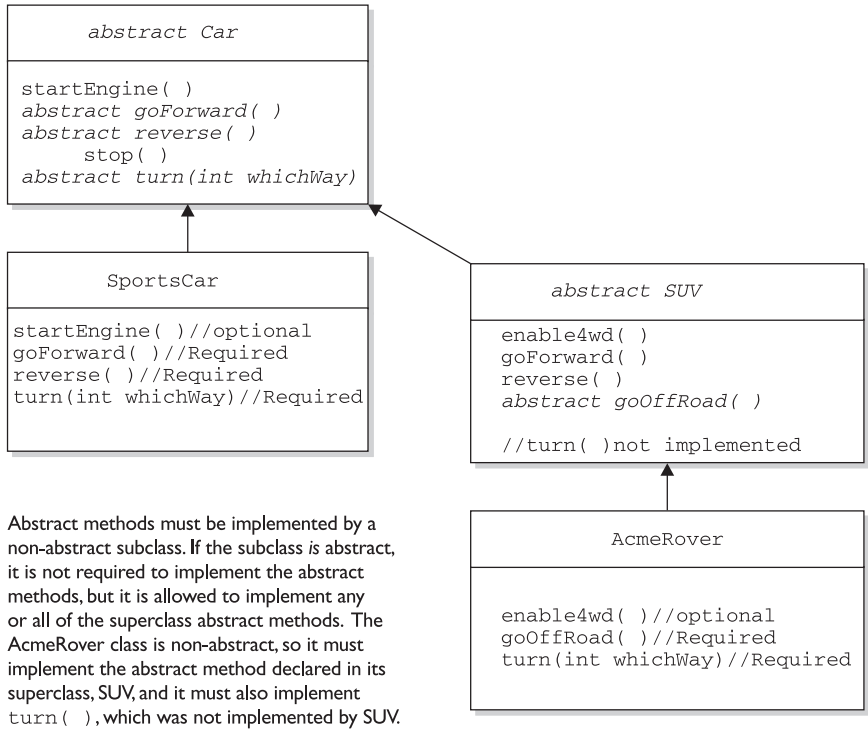
```
public abstract class Vehicle {
   private String type;
   public abstract void goUpHill();  // Abstract method
   public String getType() {
      return type;
   }    // Non-abstract method
}

public abstract class Car extends Vehicle {
   public abstract void goUpHill(); // Still abstract
   public void doCarThings() {
      // special car code goes here
   }
}

public class Mini extends Car {
   public void goUpHill() {
```

```
        // Mini-specific going uphill code
    }

}
```

So how many methods does class Mini have? Three. It inherits both the `getType()` and `doCarThings()` methods, because they're public and concrete (nonabstract). But because `goUpHill()` is abstract in the superclass Vehicle, and is never implemented in the Car class (so it remains abstract), it means class Mini—as the first concrete class below Vehicle—*must* implement the `goUpHill()` method. In other words, *class Mini can't pass the buck* (of abstract method implementation) to the next class down the inheritance tree, but class Car *can* since Car, like Vehicle, is abstract. Figure 2-4 illustrates the effects of the abstract modifier on concrete and abstract subclasses.

FIGURE 2-4

The effects of abstract on subclasses



Abstract methods must be implemented by a non-abstract subclass. If the subclass *is* abstract, it is not required to implement the abstract methods, but it is allowed to implement any or all of the superclass abstract methods. The AcmeRover class is non-abstract, so it must implement the abstract method declared in its superclass, SUV, and it must also implement `turn( )`, which was not implemented by SUV.

exam Watch

Look for concrete classes that don't provide method implementations for abstract methods of the superclass. For example, the following code won't compile:

```
public abstract class A {
    abstract void foo();
}
class B extends A {
    void foo(int I) {
    }
}
```

Class B won't compile because it doesn't implement the inherited abstract method `foo()`. Although the `foo(int I)` method in class B might appear to be an implementation of the superclass' abstract method, it is simply an overloaded method (a method using the same identifier, but different arguments), so it doesn't fulfill the requirements for implementing the superclass' abstract method. We'll look at the differences between overloading and overriding in detail in Chapter 5.

A method can never, ever, ever be marked as both `abstract` *and* `final`, or both `abstract` *and* `private`. Think about it—abstract methods *must* be implemented (which essentially means overridden by a subclass) whereas final and private methods cannot *ever* be overridden by a subclass. Or to phrase it another way, an `abstract` designation means *the superclass doesn't know anything* about how the subclasses should behave in that method, whereas a `final` designation means *the superclass knows everything* about how all subclasses (however far down the inheritance tree they may be) should behave in that method. The `abstract` and `final` modifiers are virtually opposites. Because `private` methods cannot even be *seen* by a subclass (let alone inherited) they too cannot be overridden, so they too cannot be marked `abstract`.

Abstract methods also cannot be marked as `synchronized`, `strictfp`, or `native`, all of which are modifiers describing something about the *implementation* of a method. Because abstract methods define the signature, access, and return type, but can say nothing about implementation, be watching for any of the following illegal method declarations:

```
abstract synchronized void foo();
abstract strictfp void foof();
abstract native void poof();
```

The preceding declarations will deliver you a nice compiler error message similar to

```
MyClass.java:18: illegal combination of modifiers: abstract and synchronized
   abstract synchronized void foo();
                              ^
MyClass.java:19: illegal combination of modifiers: abstract and strictfp
   abstract strictfp void foof();
                          ^
MyClass.java:20: illegal combination of modifiers: abstract and native
   abstract native void poof();
                        ^
```

Finally, you need to know that the `abstract` modifier can never be combined with the `static` modifier. We'll cover static methods later in this objective, but for now just remember that the following would be illegal:

```
abstract static void doStuff();
```

And it would give you an error that should be familiar by now:

```
MyClass.java:2: illegal combination of modifiers: abstract and static
  abstract static void doStuff();
                       ^
```

Synchronized Methods The `synchronized` keyword indicates that a method can be accessed by only one thread at a time. We'll discuss this nearly to death in Chapter 9, but for now all we're concerned with is knowing that *the `synchronized` modifier can be applied only to methods*—not variables, not classes, just methods. A typical synchronized declaration looks like this:

```
public synchronized Record retrieveUserInfo(int id) { }
```

You should also know that the `synchronized` modifier can be matched with any of the four access control levels (which means it can be paired with any of the three access modifier keywords). And you can also combine `synchronized` with `final`, but *never with* `abstract`. Synchronization is an implementation issue; only the programmer can decide whether a method needs to be marked as synchronized. If you declare a method like the following,

```
abstract synchronized void doStuff();
```

you'll get a compiler error similar to this:

```
MyClass.java:2: illegal combination of modifiers: abstract and synchronized
abstract synchronized void doStuff();
                           ^
```

Native Methods The `native` modifier indicates that a method is implemented in a platform-dependent language, such as C. You don't need to know how to use native methods for the exam, other than knowing that `native` is a modifier (thus a reserved keyword), `native` can *never* be combined with `abstract`, and `native` can be applied only to methods—not classes, not variables, just methods.

Strictfp Methods We looked earlier at using `strictfp` as a class modifier, but even if you don't declare a *class* as `strictfp`, you can still declare an individual *method* as `strictfp`. Remember, `strictfp` forces floating points (and any floating-point operations) to adhere to the IEE754 standard. With `strictfp`, you can predict how your floating points will behave regardless of the underlying platform the JVM is running on. The downside is that if the underlying platform is capable of supporting greater precision, a `strictfp` method won't be able to take advantage of it.

You'll need to have the IEEE754 standard pretty much memorized—that is, if you need something to help you fall asleep. For the exam, however, you don't need to know anything about `strictfp` other than what it's used for, that it can modify a class or *nonabstract* method declaration, and that *a variable can never be declared* `strictfp`.

Variable Declarations

We've already discussed variable *access*, which refers to the ability of code in one class to access a variable in another class. In this section we'll look at the other keywords that apply to variable declarations, but first we'll do a quick review of the difference between *instance* and *local* variables.

Instance Variables *Instance variables* are defined inside the class, but outside of any method, and are only initialized when the class is instantiated. Instance variables are the *fields* that belong to each unique object. For example, the following code defines fields (instance variables) for the name, title, and manager for employee objects:

```
class Employee {
   //  define fields (instance variables) for employee instances
   private String name;
   private String title,
   private String manager;
   // other code goes here including access methods for private fields
}
```

The preceding Employee class says that each employee instance will know its own name, title, and manager. In other words, each instance can have its own unique values for those three fields. If you see the term "field," "instance variable," "property," or "attribute," they mean virtually the same thing. (There actually *are* subtle but occasionally important distinctions between the terms, but those distinctions aren't used on the exam.)

For the exam, you need to know that instance variables

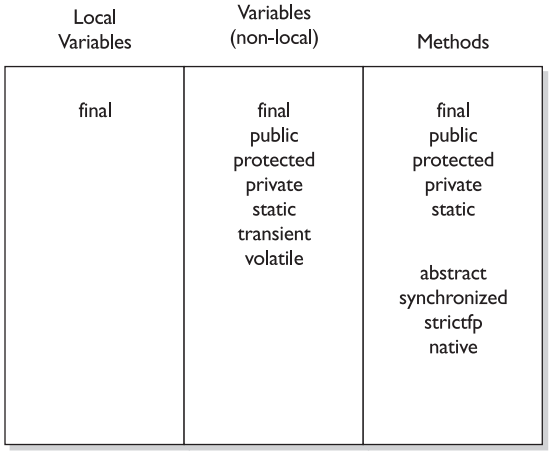
- Can use any of the four access levels (which means they can be marked with any of the three access modifiers)
- Can be marked `final`
- Can be marked `transient`
- Cannot be marked `abstract`
- Cannot be marked `synchronized`
- Cannot be marked `strictfp`
- Cannot be marked `native`

We've already covered the effects of applying access control to instance variables (it works the same way as it does for member methods). A little later in this chapter we'll look at what it means to apply the `final` or `transient` modifier to an instance variable. First, though, we'll take a quick look at the difference between instance and local variables. Figure 2-5 compares the way in which modifiers can be applied to methods vs. variables.

Local (Automatic/Stack/Method) Variables *Local variables* are variables *declared* within a method. That means the variable is not just *initialized* within the method, but also *declared within the method*. Just as the local variable starts its life inside the method, it's also destroyed when the method has completed. *Local*

FIGURE 2-5

Comparison of modifiers on variables vs. methods



variables are always on the stack, not the heap. Although the *value* of the variable might be passed into, say, another method that then stores the *value* in an instance variable, the variable itself lives only within the scope of the method.

Just don't forget that while the local variable is on the stack, if the variable is an object reference *the object itself will still be created on the heap.* There is no such thing as a stack *object*, only a stack *variable*. You'll often hear programmers use the phrase, "local object," but what they really mean is, "locally declared reference variable." So if you hear a programmer use that expression, you'll know that he's just too lazy to phrase it in a technically precise way. You can tell him we said that—unless he's really really big and knows where we live.

Local variable declarations can't use most of the modifiers that can be applied to instance variables, such as `public` (or the other access modifiers), `transient`, `volatile`, `abstract`, or `static`, but as we saw earlier, local variables *can* be marked `final`. And if you remember Chapter 1 (which we know you do, since it is, in fact, *unforgettable*), before a local variable can be used, it must be initialized with a value.

```
class TestServer {
   public void logIn() {
      int count = 10;
   }
}
```

Typically, you'll initialize a local variable in the same line in which you declare it, although you might still need to reinitialize it later in the method. The key is to remember that a local variable *must* be initialized before you try to *use* it. The compiler will reject any code that tries to use a local variable that hasn't been assigned a value, because—unlike instance variables—*local variables don't get default values.*

A local variable can't be referenced in any code outside the method in which it's declared. In the preceding code example, it would be impossible to refer to the variable *count* anywhere else in the class except within the scope of the method `logIn()`. Again, that's not to say that the *value* of *count* can't be passed out of the method to take on a new life. But the *variable* holding that value, *count*, can't be accessed once the method is complete, as the following illegal code demonstrates:

```
class TestServer {
   public void logIn() {
      int count = 10;
   }
   public void doSomething(int i) {
      count = i;  // Won't compile! Can't access count outside method login()
   }
}
```

It *is* possible to declare a local variable with the same name as an instance variable. That's known as shadowing, and the following code demonstrates this in action:

```
class TestServer {
   int count = 9;  // Declare an instance variable named count
   public void logIn() {
      int count = 10;  // Declare a local variable named count
      System.out.println("local variable count is " + count);
   }
   public void count() {
      System.out.println("instance variable count is " + count);
   }
   public static void main(String[] args) {
      new TestServer().logIn();
      new TestServer().count();
   }
}
```

The preceding code produces the following output:

```
local variable count is 10
instance variable count is 9
```

Why on earth (or the planet of your choice) would you want to do that? Normally, you won't. But one of the more common reasons is to name an argument with the same name as the instance variable to which the parameter will be assigned. The following (but wrong) code is trying to set an instance variable's value using a parameter:

```
class Foo {
   int size = 27;
   public void setSize(int size) {
      size = size;  // ??? which size equals which size???
   }
}
```

So you've decided that—for overall readability—you want to give the argument the same name as the instance variable its value is destined for, but how do you resolve the naming collision? Use the keyword `this`. The keyword `this` always always always refers to the object currently running. The following code shows this in action:

```
class Foo {
   int size = 27;
   public void setSize(int size) {
      this.size = size;  // this.size means the current object's
      // instance variable for size
   }
}
```

Final Variables Declaring a variable with the `final` keyword makes it impossible to reinitialize that variable once it has been initialized with an *explicit* value (notice we said *explicit* rather than *default*). For primitives, this means that once the variable is assigned a value, the value can't be altered. For example, if you assign 10 to the `int` variable *x*, then *x* is going to *stay* 10, forever. So that's straightforward for primitives, but what does it mean to have a final *object reference* variable? A reference variable marked `final` can't ever be reassigned to refer to a different object. The data *within* the object, however, *can* be modified, but the reference variable cannot be changed. In other words, you can use a final reference to *modify the object* it refers to, but you *can't modify the reference* variable to make it refer to a *different* object. Burn this in: *there are no final objects*, only final *references*.

You might need to remind yourself what the *value* of a reference variable actually *is*. A reference variable's value—in other words, the bit pattern the variable holds—*is not an object*. Just as the value of a primitive variable is the bit pattern representing the primitive (for example, the bits representing the integer value 2), the value of a reference variable is a bit pattern representing, well, a *reference*. We're not using "traditional" pointers in Java, but you can still *think* of it as a pointer (not necessarily a pointer to an object, but a pointer to a pointer to…). A reference variable holds bits that represent, in a platform-dependent format, *a way to get to an object*. That's really all we care about, and all we're even allowed to *know* about reference variables in Java, unless you happen to be one of the developers of a JVM.

Final instance variables don't have to be explicitly initialized in the same line in which they're declared, but the compiler will make sure that the final variable has a value by the time the constructor has completed. Don't count on the default value for final variables, though, because a final variable—even if it's an instance variable—won't be given one. The rule is: if you declare a final instance variable, you're obligated to give it an explicit value, and you must do so by the time the constructor completes. Look at the following code:

```
class FinalTest{
   final int x; // Will not work unless x is assigned in the constructor
   public void showFinal() {
      System.out.println("Final x = " + x);
   }
}
```

Attempting to compile the preceding code gives us the following:

```
%javac FinalTest.java
FinalTest.java:2: Blank final variable 'x' may not have been
initialized. It must be assigned a value in an initializer, or in
every constructor.
   final int x;
1 error
```

If you declare an instance variable as `final`, but don't give it an explicit value at the time you declare it, the variable is considered a *blank* `final`. The final instance variable can stay *blank* only until the constructor completes.

```
class FinalTest{
   final int x; // Will work because it's initialized in the constructor
```

```
   public FinalTest() {
      x = 28;  // Whew! The compiler is relieved that we took care of it
      System.out.println("Final x = " + x);
   }
}
```

So now we've seen that you need to assign a value to a final variable, but *then* what? As we mentioned earlier, *you can't change a final variable once it's been initialized!* Let's look at declaring an object reference variable as `final`:

```
import java.util.Date;
class TestClass {
   final Date d = new Date();
   public void showSample() {
      d.setYear(2001); //Altering Date object, not d variable, so it's OK
   }
}
```

In the `showSample()` method in the preceding class, the *year* of the Date instance is modified by invoking `setYear()` on the final reference variable *d*. That's perfectly legal, and the class compiles fine, because an instance can have its data modified even though the reference to it is declared `final`. But now let's see what happens when we try to assign a new object to the final reference variable *d*, after *d* has been initialized.

```
import java.util.Date;
class FinalTest {
   final Date d = new Date(); // Initialize d
   public void showSample() {
      d.setYear(2001);
      d = new Date(); // Won't work! Can't change the value of d
   }
}
```

Code within the `showSample()` method tries to reassign a new object to *d*. If we try to compile the preceding class, we're treated to this error:

```
%javac FinalTest.java
FinalTest.java:6: Can't assign a value to a final variable: d
          d = new Date();
1 error
```

exam Watch

Look for code that tries to reassign a final variable, but don't expect it to be obvious. For example, *a variable declared in an interface is always implicitly final, whether you declare it that way or not!* ***So you might see code similar to the following:***

```
interface Foo {
    Integer x = new Integer(5); // x is implicitly final
}
class FooImpl implements Foo {
    void doStuff() {
       x = new Integer(5); // Big Trouble! Can't assign new object to x
    }
}
```

The reference variable x is `final`***. No matter what. You're allowed to explicitly declare it as*** `final` ***if you like, but it doesn't matter to the compiler whether you do or not. It simply*** *is* ***final, just because it's an interface variable, and they are always implicitly*** `public static final`***. We'll look at interface variables again later in this chapter, but for now just remember that a final variable can't be reassigned, and that in the case of interface variables, they're final even if they don't say it out loud. The exam expects you to spot any attempt to violate this rule.***

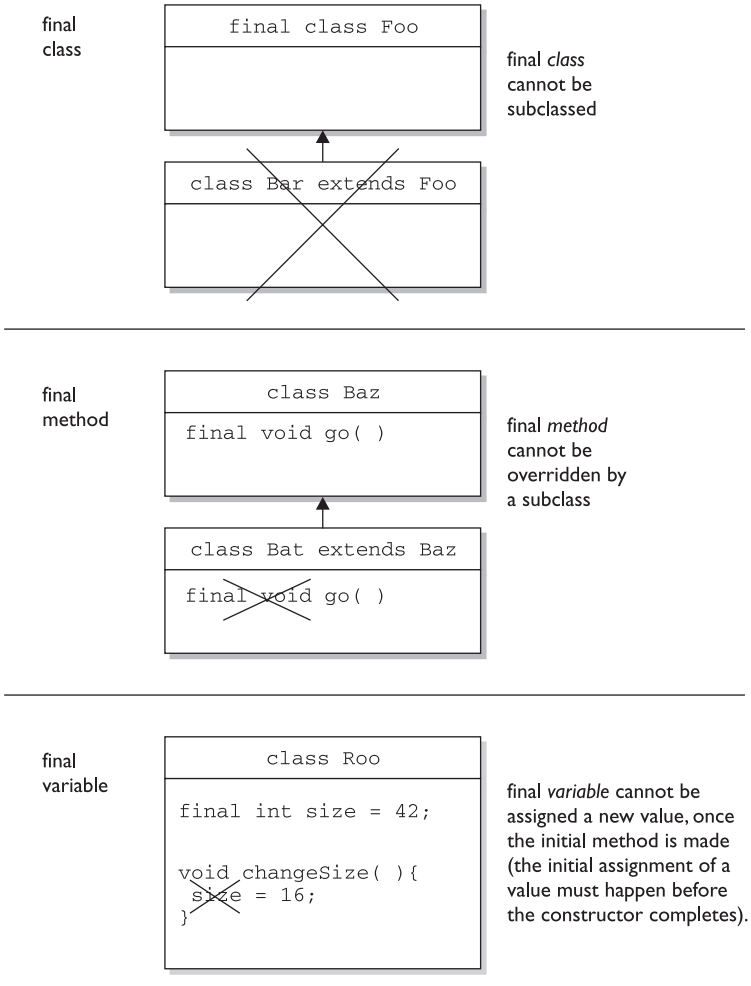
We've now covered how the `final` modifier can be applied to classes, methods, and variables. Figure 2-6 highlights the key points and differences of the various applications of `final`.

Transient Variables If you mark an instance variable as `transient`, you're telling the JVM to skip (ignore) this variable when you attempt to serialize the object declaring it. Serialization is one of the coolest features of Java; it lets you save (sometimes called "flatten") an object by writing its state (in other words, the *value of its instance variables*) to a special type of IO stream. With serialization you can save an object to a file, or even ship it over a wire for reinflating (deserializing) at the other end, in another JVM. For the exam, you aren't required to know how serialization works, but you need to know that `transient` *can be applied only to instance variables.*

Don't be surprised, though, if serialization shows up in some future version of the exam. Regardless of its relevance for the exam, serialization is one of the most

FIGURE 2-6

The effect of `final` on variables, methods, and classes



powerful aspects of Java and is worth your learning all about it. Most advanced uses of Java—RMI, EJB, and Jini, for example—depend on it. OK, we'll step off the serialization soapbox now, and resume our exam prep already in progress.

Volatile Variables The `volatile` modifier tells the JVM that a thread accessing the variable must always reconcile its own private copy of the variable with the master copy in memory. *Say what?* Don't worry about it. For the exam, all you need to know about `volatile` is that, as with `transient`, it can be applied only to instance variables. Make no mistake, the idea of multiple threads accessing

an instance variable is scary stuff, and very important for any Java programmer to understand. But as you'll see in Chapter 9, you'll probably use synchronization, rather than the `volatile` modifier, to make your data thread-safe.

The `volatile` modifier may also be applied to project managers.

Static Variables and Methods

The `static` modifier has such a profound impact on the behavior of a method or variable that we're treating it as a concept entirely separate from the other modifiers. To understand the way a static member works, we'll look first at a reason for using one. Imagine you've got a utility class with a method that always runs the same way; its sole function is to return, say, a random number. It wouldn't matter which instance of the class performed the method—it would always behave exactly the same way. In other words, the method's behavior has no dependency on the state (instance variable values) of an object. So why, then, do you need an object when the method will never be instance-specific? Why not just ask *the class itself* to run the method?

Let's imagine another scenario: suppose you want to keep a running count of all instances instantiated from a particular class. Where do you actually keep that variable? It won't work to keep it as an instance variable within the class whose instances you're tracking, because the count will just be initialized back to a default value with each new instance. The answer to both the utility-method-always-runs-the-same scenario and the keep-a-running-total-of-instances scenario is to use the `static` modifier. Variables and methods marked `static` belong to the *class*, rather than to any particular instance. In fact, you can use a static method or variable *without having any instances of that class at all*. You need only have the class available to be able to invoke a static method or access a static variable. Static variables, too, can be accessed without having an instance of a class. But if there are instances, a static variable of a class will be shared by *all* instances of that class; there is only one copy.

The following code declares and uses a static counter variable:

```
class Frog {
   static int frogCount = 0;  // Declare and initialize static variable
   public Frog() {
      frogCount += 1;  // Modify the value in the constructor
   }
   public static void main (String [] args) {
      new Frog();
      new Frog();
```

```
        new Frog();
        System.out.println("Frog count is now " + frogCount);
    }
}
```

In the preceding code, the static *frogCount* variable is set to zero when the Frog class is first loaded by the JVM, *before any Frog instances are created!* (By the way, you don't actually *need* to initialize a static variable to zero; static variables get the same default values instance variables get.) Whenever a Frog instance is created, the Frog constructor runs and increments the static *frogCount* variable. When this code executes, three Frog instances are created in `main()`, and the result is

```
Frog count is now 3
```

Now imagine what would happen if *frogCount* were an instance variable (in other words, *nonstatic*):

```
class Frog {
   int frogCount = 0;  // Declare and initialize instance variable
   public Frog() {
      frogCount += 1;  // Modify the value in the constructor
   }
   public static void main (String [] args) {
      new Frog();
      new Frog();
      new Frog();
      System.out.println("Frog count is now " + frogCount);
   }
}
```

When this code executes, it should still create three Frog instances in `main()`, but the result is…a compiler error! We can never get this code to run because it won't even compile.

```
Frog.java:11: non-static variable frogCount cannot be referenced
from a static context
    System.out.println("Frog count is " + frogCount);
                                         ^
    1 error
```

The JVM doesn't know *which* Frog object's *frogCount* you're trying to access. The problem is that `main()` is itself a static method, and thus isn't running against any particular *instance* of the class, rather just on the class itself. A static method can't access a nonstatic (instance) variable, because *there is no instance!* That's not to say there aren't instances of the class alive on the heap, but rather that even if there are,

the static method doesn't know anything about them. The same applies to instance methods; a static method can't directly invoke a nonstatic method. Think static = class, nonstatic = instance. Making the method called by the JVM (`main()`) a static method means the JVM doesn't have to create an instance of your class just to start running code.

exam Watch

One of the mistakes most often made by new Java programmers is attempting to access an instance variable (which means nonstatic variable) from the static `main()` method (which doesn't know anything about any instances, so it can't access the variable). The following code is an example of illegal access of a nonstatic variable from a static method:

```
class Foo {
   int x = 3;
   public static void main (String [] args) {
     System.out.println("x is " + x);
   }
}
```

Understand that this code will never compile, because you can't access a nonstatic (instance) variable from a static method. Just think of the compiler saying, "Hey, I have no idea which Foo object's x variable you're trying to print!" Remember, it's the class running the `main()` method, not an instance of the class. Of course, the tricky part for the exam is that the question won't look as obvious as the preceding code. The problem you're being tested for—accessing a nonstatic variable from a static method—will be buried in code that might appear to be testing something else. For example, the code above would be more likely to appear as

```
class Foo {
   int x = 3;
   float y = 4.3f;
   public static void main (String [] args) {
     for (int z = x; z < ++x; z--, y = y + z) {
       // complicated looping and branching code
     }
}
```

So while you're off trying to follow the logic, the real issue is that x and y can't be used within `main()`, because x and y are instance, not static, variables! The same applies for accessing nonstatic methods from a static method. The rule is, a static method of a class can't access a nonstatic (instance) member—method or variable—of its own class.

Accessing Static Methods and Variables

Since you don't need to have an instance in order to invoke a static method or access a static variable, then how *do* you invoke or use a static member? What's the syntax? We know that with a regular old instance method, you use the dot operator on a reference to an instance:

```
class Frog {
   int frogSize = 0;
   public int getFrogSize() {
      return frogSize;
   }
   public Frog(int s) {
      frogSize = s;
   }
   public static void main (String [] args) {
      Frog f = new Frog(25);
      System.out.println(f.getFrogSize()); // Access instance method using f
   }
}
```

In the preceding code, we instantiate a Frog, assign it to the reference variable *f*, and then use that *f* reference to invoke a method *on the Frog instance we just created*. In other words, the `getFrogSize()` method is being invoked on a specific Frog object on the heap.

But this approach (using a reference to an object) isn't appropriate for accessing a static method, because there might not be any instances of the class at all! So, the way we access a static method (or static variable) is to use the dot operator *on the class name*, as opposed to on a reference to an instance, as follows:

```
class Frog {
   static int frogCount = 0;  // Declare and initialize static variable
   public Frog() {
      frogCount += 1;  // Modify the value in the constructor
   }
}

class TestFrog {
   public static void main (String [] args) {
     new Frog();
     new Frog();
     new Frog();
     System.out.print("frogCount:"+Frog.frogCount); //Access static variable
   }
 }
```

But just to make it really confusing, the Java language also allows you to use an object reference *variable* to access a static member:

```
Frog f = new Frog();
int frogs = f.getFrogCount(); // Access static method getFrogCount using f
```

In the preceding code, we instantiate a Frog, assign the new Frog object to the reference variable *f*, and then use the *f* reference to invoke a static method! But even though we are using a specific Frog instance to access the static method, the rules haven't changed. This is merely a syntax trick to let you *use* an object reference *variable* (but not the object it refers to) to get to a static method or variable, but the static member is still unaware of the particular instance used to invoke the static member. In the Frog example, the compiler knows that the reference variable *f* is of type Frog, and so the Frog class static method is run with no awareness or concern for the Frog instance at the other end of the *f* reference. In other words, the compiler cares only that reference variable *f* is declared as type Frog. Figure 2-7 illustrates the effects of the static modifier on methods and variables.

Another point to remember is that *static methods can't be overridden*! This doesn't mean they can't be redefined in a subclass, as we'll see a little later when we look at overriding in more detail, but redefining and overriding aren't the same thing.

Things you can mark as `static`:

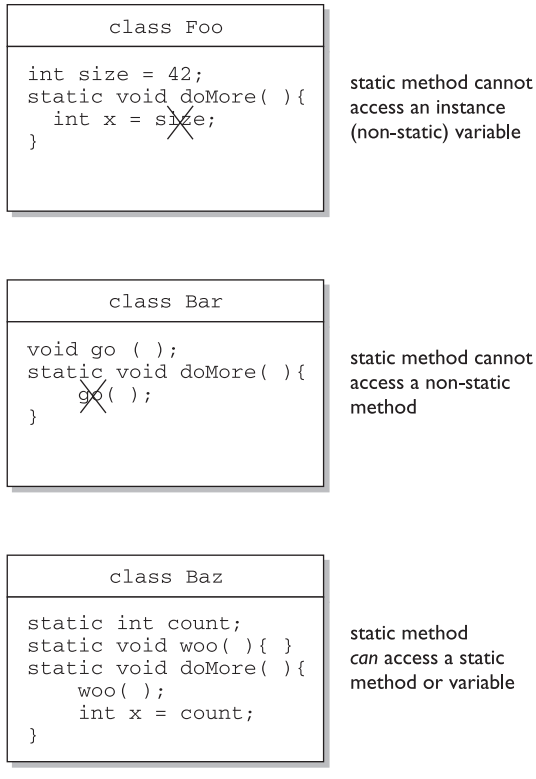
- Methods
- Variables
- Top-level nested classes (we'll look at nested classes in Chapter 8)

Things you *can't* mark as `static`:

- Constructors (makes no sense; a constructor is used only to create instances)
- Classes
- Interfaces
- Inner classes (unless you want them to be top-level nested classes; we'll explore this in Chapter 8)
- Inner class methods and instance variables
- Local variables

FIGURE 2-7

The effects of static on methods and variables



CERTIFICATION OBJECTIVE

Declaration Rules (Exam Objective 4.1)

Identify correctly constructed source files, package declarations, import statements, class declarations (of all forms, including nested classes), interface declarations, method declarations (including the `main()` *method that is used to start execution of a class), variable declarations, and identifiers.*

The previous objective, 1.2, covered the fundamentals of declarations including modifiers applied to classes, methods, and variables. In this objective, we'll look at how those fundamentals must be applied in a few specific situations. We're not covering all of Objective 4.1 in this section, however. Inner classes won't be discussed here because they're already in Chapter 8, the chapter on inner classes (what are the

odds?), and we'll hold off on interfaces until we get to Objective 4.2, the section immediately following this one.

We promise that this section will be much shorter than the previous one. We promise that we'll introduce very little new information. We promise you'll win friends and influence people with your declaration prowess. We promise to stop making promises.

Source Files, Package Declarations, and Import Statements

It's been awhile since we looked at source declaration rules (about 30+ pages ago), so let's do a quick review of the rules *again*:

- There can be only one public class per source code file.
- The name of the file must match the name of the public class.
- If the class is part of a package, the package statement must be the first line in the source code file.
- Import and package statements apply to *all* classes within a source code file.
- If there are import statements, they must go between the package statement and the class declaration. If there isn't a package statement, the import statement(s) must be the first line(s) in the source code file. If there are no package or import statements, the class declaration must be the first line in the source code file. (Comments don't count; they can appear anywhere in the source code file.)

Source File Structure

We know that you know all this, so we'll just focus on the kinds of import and package issues you might see on the exam. The following legal (albeit pretty useless) code declares a class `Foo`, in package `com.geeksanonymous`:

```
package com.geeksanonymous;  // Notice the semicolon
class Foo { }
```

There can be only one package statement per source code file, so the following would not be legal:

```
package com.geeksanonymous;
package com.wickedlysmart;  // Illegal! Only one package declaration allowed
class Foo { }
```

If class Foo adds any import statements, they must be below the package declaration and above the class declaration, as follows:

```
package  com.geeksanonymous;
import java.util.*; // Wildcard package import
import com.wickedlysmart.Foo;  // Explicit class import
class Bob { }
```

If class Foo has no package declaration, the import statements must be above the class declaration, as follows:

```
import java.util.*; // Wildcard package import
import com.wickedlysmart.Foo;  // Explicit class import
class Bob { }
```

You can have only one public class per source code file. You can put as many classes in a source code file as you like, but only one (or none) can be public. The file name should match the name of the public class, but if no public class is in the file, you can name it whatever you like. The following source code file, with two public classes, would be illegal:

```
package com.geeksanonymous;
public class Foo { }
public class Bat { }
```

But the following is fine:

```
package com.geeksanonymous;
class Foo { }
public class Bat { }
```

The order in which the classes appear makes no difference; as long as the package and import statements appear before the first class (and in the correct order), the class order doesn't matter.

on the Job

You should group classes into a single source code file only when those classes should only be used together as one component. Typically, you'll keep each class in a separate file, with the file name matching the class name (a requirement if the class is public; optional, but good practice, if the class has default access). Putting multiple classes into a single source code file makes it much harder to locate the source for a particular class, and makes the source code less reusable.

Keep in mind that *package and import declarations apply to* all *classes in a source file!* For the exam, you'll need to recognize that the package declaration at the top of a code example means that all classes in that file are in the same package.

exam Watch

The exam uses a line numbering scheme that indicates whether the code in the question is a snippet (a partial code sample taken from a larger file), or a complete file. If the line numbers start at 1, you're looking at a complete file. If the numbers start at some arbitrary (but always greater than 1) number, you're looking at only a fragment of code rather than the complete source code file. For example, the following indicates a complete file:

```
1. package fluffy;
2. class Bunny {
3.    public void hop() { }
4.     }
```

whereas the following indicates a snippet:

```
 9. public void hop() {
10.  System.out.println("hopping");
11. }
```

Using Import Statements

Import statements come in two flavors—*wildcard* import and *explicit class* import. Before we look at both in more detail, say it with me again, "Java is not C." An *import* statement is not an *include*! Import statements are little more than a way for you to save keystrokes when you're typing your code. When you put a class in a package (through the package declaration), you essentially give the class a longer name, which we call the *fully qualified name*. The fully qualified name of a class, as opposed to *just* the class name, is like talking about the difference between your full name (say, Albert James Bates IV) and your first name (Albert).

For example, if class Foo is in a package `com.geeksanonymous`, the Foo class is still named Foo, but it also has a fully qualified name of `com.geeksanonymous.Foo`. As we looked at earlier, package organization helps prevent name collisions—in case other programmers build a class named Foo, for example. But if a programmer from WickedlySmart builds a Foo class, its fully qualified name will be `com.wickedlysmart.Foo` (or possibly even `com.wickedlysmart.projectx.Foo`), while a programmer from GeeksAnonymous gives her Foo class the fully qualified name of

`com.geeksanonymous.Foo`. Once you put Foo in a package, if you refer to the Foo class in some other code, the compiler needs to know *which* Foo you're talking about.

OK, so given that there might be more than one Foo floating around, and that even within a single application you might want to use, say, two different Foo classes, you need a way to distinguish between them. Otherwise, the compiler would never know what you meant if you typed the following:

```
class Bar {
   void doStuff() {
      Foo f = new Foo(); // Here you want the WickedlySmart version
   }                     // But how will the compiler know?
}
```

To eliminate the confusion, you're required to do one of two things to help the compiler:

1. Use an `import` statement,

```
import com.wickedlysmart.Foo;
class Bar {
   void doStuff() {
      Foo f = new Foo(); // Now the compiler knows which one to use
   }
}
```

or

2. Use the fully qualified name throughout your code:

```
class Bar {
   void doStuff() {
      com.wickedlysmart.Foo f = new com.wickedlysmart.Foo() // No doubts
   }
}
```

OK, we don't know about *you*, but we'd prefer the one with less typing. The `import` statement is almost always the way to go. You need to recognize that either option is legal, however. And using *both together* is legal as well. It's not a problem, for example, to do the following:

```
import com.wickedlysmart.Foo;  // Import class Foo
class Bar {
  void doStuff() {
     com.wickedlysmart.Foo f = new com.wickedlysmart.Foo() //OK; not needed
   }
}
```

exam Watch

You might see questions that appear to be asking about classes and packages in the core Java API that you haven't studied, because you didn't think they were part of the exam objectives. For example, if you see a question like

```
class Foo extends java.rmi.UnicastRemoteObject {
  /// more code
}
```

don't panic! You're not actually being tested on your RMI knowledge, but rather a language and/or syntax issue. If you see code that references a class you're not familiar with, you can assume you're being tested on the way in which the code is *structured,* ***as opposed to what the class actually*** *does.* ***In the preceding code example, the question might really be about whether you need an import statement if you use the fully qualified name in your code (the answer is no, by the way).***

When do you use wildcard package imports vs. explicit class imports? Most of the time the compiler is just as happy with either, so the choice is more a matter of style and/or convenience. The tradeoffs usually come down to readability vs. typing.

If you use the wildcard import, other programmers reading your code will know that you're referencing classes from a particular package, but they won't be able to know how many classes—and what those classes are—from the package you've used unless they wade through the rest of the code! So the explicit class import helps folks reading your code (including you, if you're like most programmers and forget what you wrote a week after writing it) know exactly which classes you're using. On the other hand, if you're using, say, seven classes from a single package, it gets tedious to type each class in specifically. If we were forced at gunpoint to pick sides, we'd prefer the explicit class import, because of its, well, *explicitness*.

The one difference that might matter to you (but which you won't need to know for the exam) is that the order in which the compiler resolves imports is not simply top to bottom. Explicit imports are resolved first, then the classes from the current package, and last—the implicit imports.

exam Watch

Look for syntax errors on import statements. Can you spot what's wrong with the following code?

```
import java.util.Arraylist.*; // Wildcard import
import java.util; // Explicit class import
```

The first import looks like it should be a valid wildcard import, but ArrayList is a class, not a package, so it makes no sense (not to mention making the compiler cranky) to use the wildcard import on a single class. Pay attention to the syntax detail of the import statement, by looking at how the statement ends. If it ends with .*; (dot, asterisk, semicolon), then it must be a wildcard statement; therefore, the thing immediately preceding the .*; must be a package name, not a class name. Conversely, the second import looks like an explicit class import, but util is a package, not a class, so you can't end that statement with a semicolon.

Think about another dilemma for a moment: what happens if you have two classes with the same name, from two different packages, and you want to use *both* in the same source code? In that case, you have to use the fully qualified names in code. Even in the core class libraries you'll find more than one class using the same name. You'll find a List class, for example, in both `java.awt` and `java.util`. If you want to use both, you'll have to make it clear to the compiler.

Wildcard imports alone won't work properly since importing both *packages* still doesn't help the compiler figure out *which* version of the List class you want. The following code shows the problem of trying to use two classes of the same name (although different packages):

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

class TestImport {
   void doStuff() {
      List fromAWT = new List(); // How will the compiler know which to use?
      List fromUtil = new List(); // How will the compiler know which to use?
   }
}
```

The preceding code confuses the compiler (*never* a pretty thing), and you'll get a message similar to this:

```
TestImport.java:6: reference to List is ambiguous, both class
java.util.List in java.util and class java.awt.List in java.awt
match
```

```
List w = new List();
^
```

Formatting the Main() Method

When you want your code to actually run, you have to get the ball rolling with a `main()` method. The following rules apply to the `main()` method:

- It must be marked `static`.
- It must have a `void` return type.
- It must have a single String array argument.
- You can name the argument anything you want.
- It should be declared `public` (for the purposes of the exam, assume it *must* be public).

There's nothing special about the `main()` method; it's just another static method in your class. The only thing that makes it different from other methods is that it has the signature the JVM is looking for when you invoke Java as follows:

```
java MyClass
```

Typing that at the command line starts the JVM looking for the class file named `MyClass`, and when it finds it, it looks for the `main()` method—the one with a signature matching what the JVM is searching for. If it finds the matching method, you're good to go. If it doesn't, you get a runtime error like this:

```
Exception in thread "main" java.lang.NoSuchMethodError: main
```

The tricky thing about this error is that you can get it even when there *is* a `main()` method. The following code compiles fine, but still produces the previous `NoSuchMethodError` when you try to invoke this class from the command line:

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

Did you spot the problem? There *is* a `main()` method, but it isn't static. So when we say "the `main()` method," you need to know whether we mean "*a* method that

happens to be named `main()`" (which you're allowed to have) or "*the* Main() Method"—the one the JVM looks for.

Look for lots of subtle variations surrounding the `main()` method. You might see classes with a `main()` method similar to the preceding example, where the signature doesn't match what the JVM wants. You must know that** not having a proper `main()` method is a runtime error, not a compiler error! **So while you're completely free to have as many methods named `main()` as you like (or none at all), if no methods match the `main()` method the JVM looks for, then you won't be able to** run **the class by invoking Java using that class' name. You can still instantiate the class from other code (or invoke its static methods once the JVM is already running), it just can't be used to crank up a virtual machine and bootstrap your program. If the `main()` method doesn't look like this:

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

you won't be able to** run **the class. You actually do have a few** slight **variations you can make to the `main()` method. For example, the following is a perfectly legal, executable `main()` method:

```
static public void main (String whatever []) { }
```

In other words, you're allowed to name the String array argument whatever you like, and the `static` and `public` modifiers can be used in a different order. The most important point for the exam is to know that not having the "able-to-run" `main()` method is a runtime, rather than compiler, error. A class with a legal, nonstatic `main()` method, for example, will compile just fine, and other code is free to call that method. But when it comes time to use that class to invoke the JVM, that nonstatic `main()` method just won't cut it, and you'll get the runtime error.

We've covered everything we need for this objective except for interface declarations, which we'll look at next, and inner class declarations, which we'll look at in Chapter 8. The key points for this objective are the structure of a source code file (where to place the package, import, and class declarations) and the signature of the `main()` method (`public static void main (String [] args)`). Next, we're going to dive into the rules for declaring and implementing interfaces.

CERTIFICATION OBJECTIVE

Interface Implementation (Exam Objective 4.2)

Identify classes that correctly implement an interface where that interface is either java.lang.Runnable or a fully specified interface in the question.

So far in this chapter, we began with Objective 1.2—a look at how to use modifiers in class, method, and variable declarations. Next, for Objective 4.1, we covered the rules for structuring a source code file and declaring the `main()` method. In this objective, we'll focus on interface declarations and implementations.

exam Watch

You must know how to implement the `java.lang.Runnable` interface, without being shown the code in the question. In other words, you might be asked to choose from a list of six classes which one provides a correct implementation of `Runnable`. Be sure you memorize the signature of the one and only one `Runnable` interface method:

```
public void run() { }
```

For any other interface-related question not dealing with `Runnable`, if the specification of the interface matters, the interface code will appear in the question. A question, for example, might show you a complete interface and a complete class, and ask you to choose whether or not the class correctly implements the interface. But if the question is about `Runnable`, you won't be shown the interface. You're expected to have `Runnable` memorized!

Declaring an Interface

When you create an interface, you're defining a contract for *what* a class can do, without saying anything about *how* the class will do it. An interface is a contract. You could write an interface `Bounceable`, for example, that says in effect, "This is the Bounceable interface. Any class type that implements this interface must agree to write the code for the `bounce()` and `setBounceFactor()` methods."

By defining an interface for Bounceable, any class that wants to be treated as a *Bounceable thing* can simply implement the Bounceable interface and provide code for the interface's two methods.

Interfaces can be implemented by *any* class, *from any inheritance tree*. This lets you take radically different classes and give them a common characteristic. For example, you might want both a Ball and a Tire to have bounce behavior, but Ball and Tire don't share any inheritance relationship; Ball extends Toy while Tire extends only `java.lang.Object`. But by making both Ball and Tire implement Bounceable, you're saying that Ball and Tire can be treated as, "Things that can bounce," which in Java translates to "Things on which you can invoke the `bounce()` and `setBounceFactor()` methods." Figure 2-8 illustrates the relationship between interfaces and classes.

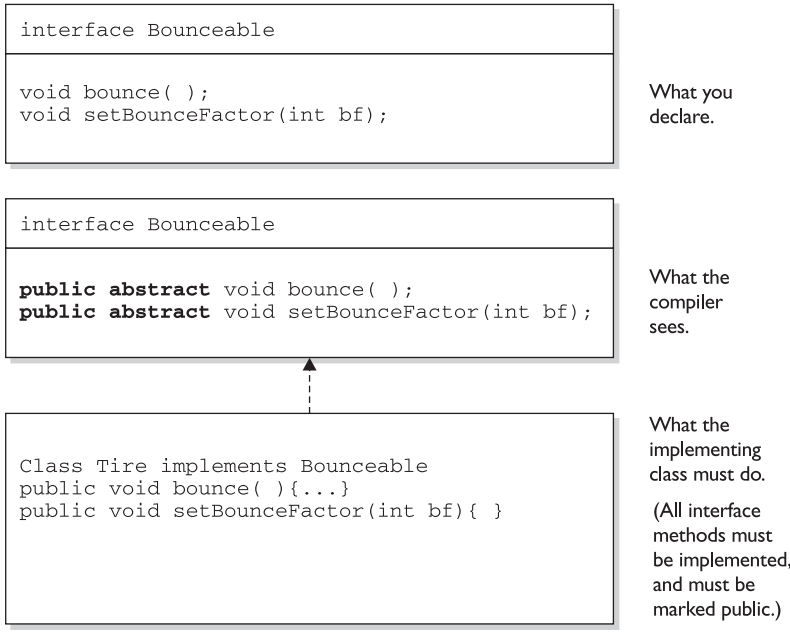
Think of an interface as a 100-percent abstract class. Like an abstract class, an interface defines abstract methods that take the form,

```
abstract void bounce();  // Ends with a semicolon rather than curly braces
```

But while an abstract class can define both abstract and nonabstract methods, an interface can have *only* abstract methods. Another place interfaces differ from

FIGURE 2-8

The relationship between interfaces and classes



abstract classes is that interfaces have very little flexibility in how the methods and variables defined in the interface are declared. The rules are strict, but simple:

- All interface methods are implicitly public and abstract.
- Interface methods must not be static.
- You do not need to actually *type* the `public` or `abstract` modifiers in the method declaration, but the method is still always public and abstract.
- All *variables* defined in an interface must be public, static, and final—in other words, interfaces can declare only constants, not instance variables.
- Because interface methods are abstract, they cannot be marked `final`, `native`, `strictfp`, or `synchronized`.
- An interface can *extend* one or more other interfaces.
- An interface cannot extend anything *but* another interface.
- An interface cannot *implement* another interface or class.
- An interface must be declared with the keyword `interface`.
- Interface types can be used polymorphically (see Chapter 5 for more details).

The following is a legal interface declaration:

```
public abstract interface Rollable { }
```

Typing in the `abstract` modifier is considered redundant; interfaces are implicitly abstract whether you type `abstract` or not. You just need to know that both of these declarations are legal, and functionally identical:

```
public abstract interface Rollable { }
public interface Rollable { }
```

The `public` modifier *is* required if you want the interface to have public rather than default access.

We've looked at the interface declaration but now we'll look closely at the *methods* within an interface:

```
public interface Bounceable {
    public abstract void bounce();
    public abstract void setBounceFactor(int bf);
}
```

Typing in the `public` and `abstract` modifiers on the methods is redundant, though, since all interface methods are implicitly public and abstract. Given that rule, you can see that the following code is exactly equivalent to the preceding interface:

```
public interface Bounceable {
      void bounce();  // No modifiers
      void setBounceFactor(int bf);  // No modifiers
}
```

You must remember that all interface methods are public and abstract *regardless of what you see in the interface definition.*

exam Watch

Look for interface methods declared with any combination of public, abstract, or no modifiers. For example, the following five method declarations, if declared within an interface, are legal *and identical!*

```
void bounce();
public void bounce();
abstract void bounce();
public abstract void bounce();
abstract public void bounce();
```

The following interface method declarations won't compile:

```
final void bounce(); // final and abstract can never be used
static void bounce(); // interfaces define instance methods
private void bounce(); // interface methods are always public
protected void bounce(); // (same as above)
synchronized void bounce(); // can't mix abstract and synchronized
native void bounce(); // can't mix abstract and native
strictfp void bounce(); // can't mix abstract and strictfp
```

Declaring Interface Constants

You're allowed to put constants in an interface. By doing so, you guarantee that any class implementing the interface will have access to the same constant. Imagine that a Bounceable thing works by using `int` values to represent gravity where the Bounceable thing is, its degree of bounciness (bounce-osity?), and so on. Now imagine that for a Bounceable thing, gravity is set such that a 1 is low, 2 is medium, 3 is high, and for bounciness, 4 is a little bouncy, 8 is very bouncy, and 12 is

extremely bouncy. Those numbers are tough to remember when you're trying to decide how to set the values ("let's see, was it 8 for high gravity and 3 for medium bounce? Or was it the other way around…"). Now let's say that you (the developer of Bounceable) decide that it would be much easier for programmers to remember names like *HIGH_GRAVITY*, *LOW_BOUNCE*, and *HIGH_BOUNCE* as opposed to knowing the exact *int* values corresponding to each of those. So, you know you want to define some constants so the programmer can just use the constant name rather than the *int* value. You need something like the following:

```
public final static int LOW_BOUNCE = 4;
public final static int HIGH_GRAVITY = 3;
… // and so on
```

That way, if a method takes the `int` values,

```
public void animateIt(int gravity, int bounceFactor) {  }
```

then the code that calls `animateIt()` can substitute the constants wherever the `int` values are expected, as follows:

```
animator.animateIt(LOW_GRAVITY, HIGH_BOUNCE);
```

So we've made a case for using constants with easy-to-remember names (as opposed to using nearly arbitrary numbers), but where do you *put* these constants so that all Bounceable things (things as in "things that implement the Bounceable interface") can substitute the `int` constant name everywhere one of the `int` values is needed? You could define them in some companion class called, for example, BounceableConstants. But why not just put them in the Bounceable interface? That way you can guarantee that all Bounceable things will always have access to the constants, without having to create another class. Look at the changes we've made to the `Bounceable` interface:

```
public interface Bounceable {
   int LOW_GRAVITY = 1;
   int MEDIUM_GRAVITY = 2;
   int HIGH_GRAVITY = 3;
   int LOW_BOUNCE = 4;
   int MEDIUM_BOUNCE = 8;
   int HIGH_BOUNCE = 12;

   void bounce();
```

```
    void setBounceFactor(int bounceFactor);
    void setGravity(int gravity);
}
```

By placing the constants right in the interface, *any class that implements the interface has direct access to the constants, just as if the class had inherited them.* For example, the following would be legal for a Bounceable implementation class:

```
class Ball implements Bounceable {
   // Lots of exciting code goes here
   public void bounce() {
      animator.animateIt(LOW_GRAVITY, HIGH_BOUNCE);  // MUCH easier this way
   }
   // Still more action-packed code goes here
}
```

You need to remember a few rules for interface constants. They must always be

- public
- static
- final

So that sounds simple, right? After all, interface constants are no different from any other publicly accessible constants, so they obviously must be declared `public`, `static`, and `final`. But before you breeze past the rest of this discussion, think about the implications. First, because interface constants are defined in an interface, they don't have to be *declared* as `public`, `static`, or `final`. They must *be* public, static, and final, but you don't have to actually *declare* them that way. Just as interface methods are always public and abstract whether you say so in the code or not, *any variable defined in an interface must be—and implicitly is—a public constant.* See if you can spot the problem with the following implementation of Bounceable:

```
class Check implements Bounceable {
   // Implementation code goes here
   public void adjustGravityFactors(int x) {
      if (x > LOW_GRAVITY) {
      LOW_GRAVITY = x;
      MEDIUM_GRAVITY = x + 1;
      HIGH-GRAVITY = x + 2;
      }
   }
}
```

You can't change the value of a constant! Once the value has been assigned, the value can never be modified. The assignment happens in the interface itself (where the constant is declared), so the implementing class can access it and use it, but as a read-only value.

exam Watch

Look for interface definitions that define constants, but without explicitly using the required modifiers. For example, the following are all identical:

```
public int x = 1; // Looks non-static and non-final, but isn't!
int x = 1; // Looks default, non-final, and non-static, but isn't!
static int x = 1; // Doesn't show final or public
final int x = 1; // Doesn't show static or public
public static int x = 1; // Doesn't show final
public final int x = 1; // Doesn't show static
static final int x = 1 // Doesn't show public
public static final int x = 1; // Exactly what you get implicitly
```

Any combination of the required (but implicit) modifiers is legal, as is using no modifiers at all! On the exam, you can expect to see questions you won't be able to answer correctly unless you know, for example, that an interface variable is final and can never be given a value by the implementing (or any other) class.

Implementing an Interface

When you implement an interface, you're agreeing to adhere to the contract defined in the interface. That means you're agreeing to provide legal implementations for every method defined in the interface, and that anyone who knows what the interface methods look like (not how they're *implemented*, but how they can be *called* and what they *return*) can rest assured that they can invoke those methods on an instance of your implementing class.

For example, if you create a class that implements the `Runnable` interface (so that your code can be executed by a specific thread), you *must* provide the `public void run()` method. Otherwise, the poor thread could be told to go execute your `Runnable` object's code and—surprise surprise—the thread then discovers the object has no `run()` method! (At which point, the thread would blow up and the JVM would crash in a spectacular yet horrible explosion.) Thankfully, Java prevents this meltdown from occurring by running a compiler check on any class that claims to implement an interface. If the class *says* it's implementing an interface, it darn well

better have an implementation for each method in the interface (with a few exceptions we'll look at in a moment).

We looked earlier at several examples of implementation classes, including the `Ball` class that implements `Bounceable`, but the following class would also compile legally:

```
public class Ball implements Bounceable {  // Keyword 'implements'
   public void bounce() { }
   public void setBounceFactor(int bf) { }
}
```

OK, we know what you're thinking: "This has got to be the worst implementation class in the history of implementation classes." It compiles, though. And runs. The interface contract guarantees that a class will have the method (in other words, others can call the method subject to access control), but it never guaranteed a good implementation—or even any actual implementation code in the body of the method. The compiler will never say to you, "Um, excuse me, but did you *really* mean to put *nothing* between those curly braces? *HELLO*. This *is* a method after all, so shouldn't it do something?"

Implementation classes must adhere to the same rules for method implementation as a class extending an abstract class. In order to be a legal implementation class, a nonabstract implementation class *must* do the following:

- Provide concrete (nonabstract) implementations for all methods from the declared interface.
- Follow all the rules for legal overrides (see Chapter 5 for details).
- Declare no checked exceptions on implementation methods other than those declared by the interface method, or subclasses of those declared by the interface method.
- Maintain the signature of the interface method, and maintain the same return type (but does not have to declare the exceptions declared in the interface method declaration).

But wait, there's more! An implementation class can itself be abstract! For example, the following is legal for a class Ball implementing the Bounceable interface:

```
abstract class Ball implements Bounceable { }
```

Notice anything missing? We never provided the implementation methods. And that's OK. If the implementation class is abstract, it can simply pass the buck to its first concrete subclass. For example, if class BeachBall extends Ball, and BeachBall is not abstract, then BeachBall will have to provide all the methods from Bounceable:

```
class BeachBall extends Ball {
  // Even though we don't say it in the class declaration above,
  //BeachBall implements Bounceable, since BeachBall's abstract
  //superclass (Ball) implements Bounceable

   public void bounce() {
      // interesting BeachBall-specific bounce code
   }
   public void setBounceFactor(int bf) {
      // clever BeachBall-specific code for setting a bounce factor
   }
   // if class Ball defined any abstract methods, they'll have to be
   // implemented here as well.
}
```

exam Watch

Look for methods that** claim **to implement an interface but don't provide the correct method implementations. Unless the implementing class is abstract, the implementing class must provide implementations for all methods defined in the interface.

Two more rules you need to know and then we can put this topic to sleep (or put *you* to sleep; we always get those two confused):

1. **A class can implement more than one interface.**

   It's perfectly legal to say, for example, the following:

   ```
   public class Ball implements Bounceable, Serializable,
   Runnable { … }
   ```

   You can extend only one class, but implement many. But remember that *subclassing* defines who and what you are, whereas *implementing* defines a role you can play or a hat you can wear, despite how different you might be from some other class implementing the same interface (but from a different inheritance tree). For example, a person *extends* `HumanBeing` (although for some, that's debatable). But a person may also *implement* `programmer`, `snowboarder`, `employee`, `parent`, or `personcrazyenoughtotakethisexam`.

2. **An interface can itself *extend* another interface, but never *implement* anything.**

   The following code is perfectly legal:

   ```
   public interface Bounceable extends Moveable { }
   ```

   What does that mean? The first concrete (nonabstract) implementation class of `Bounceable` must implement all the methods of Bounceable, plus all the methods of Moveable! The subinterface, as we call it, simply adds more requirements to the contract of the superinterface. You'll see this concept applied in many areas of Java, especially J2EE where you'll often have to build your own interface that extends one of the J2EE interfaces.

Hold on though, because here's where it gets strange. *An interface can extend more than one interface!* Think about that for a moment. You know that when we're talking about classes, the following is illegal:

```
public class Programmer extends Employee, Geek { } // Illegal!
```

A class is not allowed to extend multiple classes in Java. It that were allowed, it would be multiple inheritance, a potential nightmare in some scenarios (more on that in Chapter 5). An interface, however, is free to extend multiple interfaces.

```
interface Bounceable extends Moveable, Spherical {
   void bounce();
   void setBounceFactor(int bf);
}

interface Moveable {
   void moveIt();
}

interface Spherical {
   void doSphericalThing();
}
```

Ball is required to implement Bounceable, plus all methods from the interfaces that Bounceable extends (including any interfaces *those* interfaces extend and so on

until you reach the top of the stack—or is it *bottom* of the stack?—well, you know what we mean). So Ball would need to look like the following:

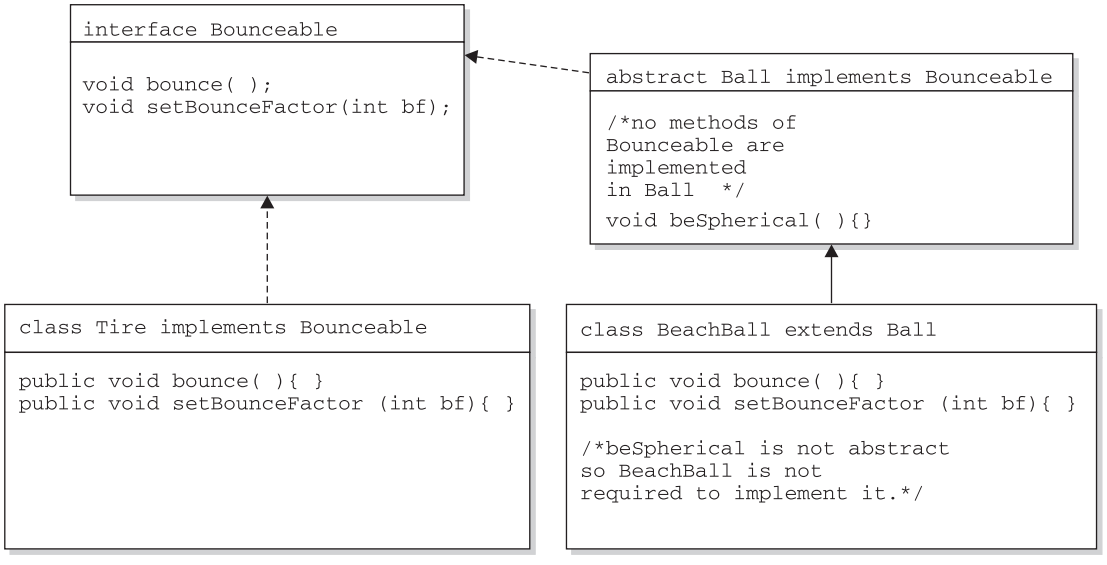
```
class Ball implements Bounceable {
   // Implement the methods from Bounceable
   public void bounce() { }
   public void setBounceFactor(int bf) { }

   // Implement the methods from Moveable
   public void moveIt() { }

   // Implement the methods from Spherical
   public void doSphericalThing() { }
}
```

If class Ball fails to implement any of the methods from Bounceable, Moveable, or Spherical, the compiler will jump up and down wildly, red in the face, until it does. *Unless*, that is, *class Ball is marked abstract*. In that case, Ball could choose to implement any, all, or none of the methods from any of the interfaces, thus leaving the rest of the implementations to a concrete subclass of Ball, as follows:

```
abstract class Ball implements Bounceable {
   public void bounce() { … }  // Define bounce behavior
   public void setBounceFactor(int bf) { … }
   // Don't implement the rest; leave it for a subclass
}

class SoccerBall extends Ball {
   // class SoccerBall must implement the interface methods that Ball didn't
   public void moveIt() { … }
   public void doSphericalThing() { … }
   // SoccerBall can choose to override the Bounceable methods
   // implemented by Ball
   public void bounce() { … }
}
```

Figure 2-9 compares the legal and illegal use of extends and implements, for both classes and interfaces.

FIGURE 2-9 Legal and illegal uses of extends and implements



Because BeachBall is the first concrete class to implement Bounceable, it must provide implementations for all methods of Bounceable, except those defined in the abstract class Ball. Because Ball did not provide implementations of Bounceable methods, BeachBall was required to implement all of them.

exam Watch

Look for illegal uses of extends and implements. The following shows examples of legal and illegal class and interface declarations:

```
class Foo { } // OK
class Bar implements Foo { } // No! Can't implement a class
interface Baz { } // OK
interface Fi { } // OK
interface Fee implements Baz { } // No! Interface can't implement an interface
interface Zee implements Foo { } // No! Interface can't implement a class
interface Zoo extends Foo { } // No! Interface can't extend a class
interface Boo extends Fi { } // OK. Interface can extend an interface
class Toon extends Foo, Button { } // No! Class can't extend multiple classes
class Zoom implements Fi, Fee { } // OK. class can implement multiple interfaces
interface Vroom extends Fi, Fee { } // OK. interface can extend multiple interfaces
```

Burn these in, and watch for abuses in the questions you get on the exam. Regardless of what the question** appears **to be testing, the** real **problem might be the class or interface declaration. Before you get caught up in, say, tracing a complex threading flow, check to see if the code will even compile. (Just that tip alone may be worth your putting us in your will!) (You'll be impressed by the effort the exam developers put into distracting you from the** real **problem.) (How did people manage to write anything before parentheses were (was?) invented?)

CERTIFICATION SUMMARY

You now have a good understanding of access control as it relates to classes, methods, and variables. You've looked at how access modifiers (`public`, `protected`, `private`) define the access control of a class or member. You've also looked at the other modifiers including `static`, `final`, `abstract`, `synchronized`, etc. You've learned how some modifiers can never be combined in a declaration, such as mixing `final` with `abstract` or `abstract` with `private`.

Keep in mind that there are no final objects in Java. A reference *variable* marked `final` can never be changed, but the *object* it refers to can be modified. You've seen that `final` applied to methods means a subclass can't override them, and when applied to a class, the final class can't be subclassed.

You learned that abstract classes can contain both abstract and nonabstract methods, but that if even a single method is marked `abstract`, the class must be marked abstract. Don't forget that a concrete (nonabstract) subclass of an abstract class must provide implementations for all the abstract methods of the superclass, but that an abstract class does not have to implement the abstract methods from its superclass. An abstract subclass can "pass the buck" to the first concrete subclass.

Remember what you've learned about static variables and methods, especially that static members are *per-class* as opposed to *per-instance*. Don't forget that a static method can't directly access an instance variable from the class it's in, because it doesn't have an explicit reference to any particular instance of the class.

You've also looked at source code declarations, including the use of package and import statements. Don't forget that you can have a `main()` method with any legal signature you like, but if it isn't `public static void main (String [] args)`, the JVM won't be able to invoke it to start your program running.

Finally, you covered interface implementation, including the requirement to implement `public void run()` for a class that implements Runnable. You also saw that interfaces can extend another interface (even multiple interfaces), and that any class that implements an interface must implement all methods from *all* the interfaces in the inheritance tree of the interface the class is implementing.

Before you hurl yourself at the practice test, spend some time with the following optimistically named "Two-Minute Drill." Come back to this particular drill often, as you work through this book and especially when you're doing that last-minute cramming. Because—and here's the advice you wished your mother had given you before you left for college—*it's not what you know, it's when you know it.*

TWO-MINUTE DRILL

Class Access Modifiers

- ❑ There are three access *modifiers*: `public`, `protected`, and `private`.
- ❑ There are four access *levels*: `public`, `protected`, default, and `private`.
- ❑ Classes can have only public or default access.
- ❑ Class visibility revolves around whether code in one class can:
  - ❑ Create an instance of another class
  - ❑ Extend (or subclass), another class
  - ❑ Access methods and variables of another class
- ❑ A class with default access can be seen only by classes within the same package.
- ❑ A class with public access can be seen by all classes from all packages.

Class Modifiers (nonaccess)

- ❑ Classes can also be modified with `final`, `abstract`, or `strictfp`.
- ❑ A class cannot be both final *and* abstract.
- ❑ A final class cannot be subclassed.
- ❑ An abstract class cannot be instantiated.
- ❑ A single abstract method in a class means the whole class must be abstract.
- ❑ An abstract class can have both abstract and nonabstract methods.
- ❑ The first concrete class to extend an abstract class must implement all abstract methods.

Member Access Modifiers

- Methods and instance (nonlocal) variables are known as "members."
- Members can use all four access levels: `public`, `protected`, default, `private`.
- Member access comes in two forms:
  - Code in one class can access a member of another class.
  - A subclass can inherit a member of its superclass.
- If a class cannot be accessed, its members cannot be accessed.
- Determine class visibility before determining member visibility.
- Public members can be accessed by all other classes, even in different packages.
- If a superclass member is public, the subclass inherits it—regardless of package.
- Members accessed without the dot operator (.) must belong to the same class.
- `this.` always refers to the currently executing object.
- `this.aMethod()` is the same as just invoking `aMethod()`.
- Private members can be accessed only by code in the same class.
- Private members are not visible to subclasses, so private members cannot be inherited.
- Default and protected members differ only in when subclasses are involved:
  - Default members can be accessed only by other classes in the same package.
  - Protected members can be accessed by other classes in the same package, plus subclasses regardless of package.
- Protected = package plus kids (kids meaning subclasses).
- For subclasses outside the package, the protected member can be accessed only through inheritance; a subclass outside the package cannot access a protected member by using a reference to an instance of the superclass (in other words, inheritance is the only mechanism for a subclass outside the package to access a protected member of its superclass).
- A protected member inherited by a subclass from another package is, in practice, private to all other classes (in other words, no other classes from

the subclass' package or any other package will have access to the protected member from the subclass).

Local Variables

- ❑ Local (method, automatic, stack) variable declarations cannot have access modifiers.
- ❑ `final` is the only modifier available to local variables.
- ❑ Local variables don't get default values, so they must be initialized before use.

Other Modifiers—Members

- ❑ Final methods cannot be overridden in a subclass.
- ❑ Abstract methods have been declared, with a signature and return type, but have not been implemented.
- ❑ Abstract methods end in a semicolon—no curly braces.
- ❑ Three ways to spot a nonabstract method:
  - ❑ The method is not marked abstract.
  - ❑ The method has curly braces.
  - ❑ The method has code between the curly braces.
- ❑ The first nonabstract (concrete) class to extend an abstract class must implement all of the abstract class' abstract methods.
- ❑ Abstract methods must be implemented by a subclass, so they must be inheritable. For that reason:
  - ❑ Abstract methods cannot be private.
  - ❑ Abstract methods cannot be final.
- ❑ The `synchronized` modifier applies only to methods.
- ❑ Synchronized methods can have any access control and can also be marked `final`.
- ❑ Synchronized methods cannot be abstract.
- ❑ The `native` modifier applies only to methods.
- ❑ The `strictfp` modifier applies only to classes and methods.

- Instance variables can
  - Have any access control
  - Be marked `final` or `transient`
- Instance variables cannot be declared `abstract`, `synchronized`, `native`, or `strictfp`.
- It is legal to declare a local variable with the same name as an instance variable; this is called "shadowing."
- Final variables have the following properties:
  - Final variables cannot be reinitialized once assigned a value.
  - Final reference variables cannot refer to a different object once the object has been assigned to the final variable.
  - Final reference variables must be initialized before the constructor completes.
- There is no such thing as a final object. An object reference marked `final` does not mean the object itself is immutable.
- The `transient` modifier applies only to instance variables.
- The `volatile` modifier applies only to instance variables.

Static variables and methods

- They are not tied to any particular instance of a class.
- An instance of a class does not need to exist in order to use static members of the class.
- There is only one copy of a static variable per class and all instances share it.
- Static variables get the same default values as instance variables.
- A static method (such as `main()`) cannot access a nonstatic (instance) variable.
- Static members are accessed using the class name:
  `ClassName.theStaticMethodName()`
- Static members can also be accessed using an instance reference variable,
  `someObj.theStaticMethodName()`
  but that's just a syntax trick; the static method won't know anything about the instance referred to by the variable used to invoke the method. The

compiler uses the *class type* of the reference variable to determine which static method to invoke.

- ❑ Static methods cannot be *overridden*, although they can be redeclared/redefined by a subclass. So although static methods can sometimes *appear* to be overridden, polymorphism will not apply (more on this in Chapter 5).

Declaration Rules

- ❑ A source code file can have only one public class.
- ❑ If the source file contains a public class, the file name should match the public class name.
- ❑ A file can have only one package statement, but can have multiple import statements.
- ❑ The package statement (if any) must be the first line in a source file.
- ❑ The import statements (if any) must come after the package and before the class declaration.
- ❑ If there is no package statement, import statements must be the first statements in the source file.
- ❑ Package and import statements apply to all classes in the file.
- ❑ A file can have more than one nonpublic class.
- ❑ Files with no public classes have no naming restrictions.
- ❑ In a file, classes can be listed in any order (there is no forward referencing problem).
- ❑ Import statements only provide a typing shortcut to a class' fully qualified name.
- ❑ Import statements cause no performance hits and do not increase the size of your code.
- ❑ If you use a class from a different package, but do not import the class, you must use the fully qualified name of the class everywhere the class is used in code.
- ❑ Import statements can coexist with fully qualified class names in a source file.
- ❑ Imports ending in `'.*;'` are importing all classes within a package.

- Imports ending in `';'` are importing a single class.
- You must use fully qualified names when you have different classes from different packages, with the same class name; an import statement will not be explicit enough.

Properties of main()

- It must be marked `static`.
- It must have a `void` return type.
- It must have a single String array argument; the name of the argument is flexible, but the convention is *args*.
- For the purposes of the exam, assume that *the* `main()` method must be public.
- Improper `main()` method declarations (or the lack of a `main()` method) cause a runtime error, not a compiler error.
- In the declaration of `main()`, the order of `public` and `static` can be switched, and *args* can be renamed.
- Other overloaded methods named `main()` can exist legally in the class, but if none of them match the expected signature for *the* `main()` method, then the JVM won't be able to use that class to start your application running.

java.lang.Runnable

- You must memorize the `java.lang.Runnable` interface; it has a single method you must implement: `public void run {}`.

Interface Implementation

- Interfaces are contracts for what a class can do, but they say nothing about the way in which the class must do it.
- Interfaces can be implemented by any class, from any inheritance tree.
- An interface is like a 100-percent abstract class, and is implicitly abstract whether you type the `abstract` modifier in the declaration or not.
- An interface can have only abstract methods, no concrete methods allowed.

- ❑ Interfaces are by default public and abstract—explicit declaration of these modifiers is optional.
- ❑ Interfaces can have constants, which are always implicitly `public`, `static`, and `final`.
- ❑ Interface constant declarations of `public`, `static`, and `final` are optional in any combination.
- ❑ A legal nonabstract implementing class has the following properties:
  - ❑ It provides concrete implementations for all methods from the interface.
  - ❑ It must follow all legal override rules for the methods it implements.
  - ❑ It must not declare any new *checked* exceptions for an implementation method.
  - ❑ It must not declare any checked exceptions that are broader than the exceptions declared in the interface method.
  - ❑ It may declare runtime exceptions on any interface method implementation regardless of the interface declaration.
  - ❑ It must maintain the exact signature and return type of the methods it implements (but does not have to declare the exceptions of the interface).
- ❑ A class implementing an interface can itself be abstract.
- ❑ An abstract implementing class does not have to implement the interface methods (but the first concrete subclass must).
- ❑ A class can extend only one class (no multiple inheritance), but it can implement many.
- ❑ Interfaces *can* extend one or more other interfaces.
- ❑ Interfaces *cannot* extend a class, or implement a class or interface.
- ❑ When taking the exam, verify that interface and class declarations are legal before verifying other code logic.

SELF TEST

The following questions will help you measure your understanding of the material presented in this chapter. Read all of the choices carefully, as there may be more than one correct answer. Choose all correct answers for each question.

Declarations and Modifiers (Sun Objective 1.2)

1. What is the most restrictive access modifier that will allow members of one class to have access to members of another class in the same package?

   A. `public`
   B. `abstract`
   C. `protected`
   D. `synchronized`
   E. default access

2. Given a method in a `public` class, what access modifier do you use to restrict access to that method to only the other members of the same class?

   A. `final`
   B. `static`
   C. `private`
   D. `protected`
   E. `volatile`
   F. default access

3. Given the following,

```
1.   abstract class A {
2.     abstract short m1() ;
3.     short m2() { return (short) 420; }
4.   }
5.
6.   abstract class B extends A {
7.     // missing code ?
8.     short m1() { return (short) 42; }
9.   }
```

   which three of the following statements are true? (Choose three.)

A. The code will compile with no changes.
B. Class B must either make an `abstract` declaration of method `m2()` or implement method `m2()` to allow the code to compile.
C. It is legal, but not required, for class B to either make an abstract declaration of method `m2()` or implement method `m2()` for the code to compile.
D. As long as line 8 exists, class A must declare method `m1()` in some way.
E. If line 6 were replaced with `'class B extends A {'` the code would compile.
F. If class A was not abstract and method `m1()` on line 2 was implemented, the code would not compile.

4. Which two of the following are legal declarations for nonnested classes and interfaces? (Choose two.)

A. `final abstract class Test {}`
B. `public static interface Test {}`
C. `final public class Test {}`
D. `protected abstract class Test {}`
E. `protected interface Test {}`
F. `abstract public class Test {}`

5. How many of the following are legal method declarations?

```
1 - protected abstract void m1();
2 - static final void m1(){}
3 - transient private native void m1() {}
4 - synchronized public final void m1() {}
5 - private native void m1();
6 - static final synchronized protected void m1() {}
```

A. 1
B. 2
C. 3
D. 4
E. 5
F. All of them

6. Given the following,

```
1.  package testpkg.p1;
2.  public class ParentUtil {
3.    public int x = 420;
4.    protected int doStuff() { return x; }
5.  }

1.  package testpkg.p2;
2.  import testpkg.p1.ParentUtil;
3.  public class ChildUtil extends ParentUtil {
4.    public static void main(String [] args) {
5.      new ChildUtil().callStuff();
6.    }
7.    void callStuff() {
8.      System.out.print("this " + this.doStuff() );
9.      ParentUtil p = new ParentUtil();
10.     System.out.print(" parent " + p.doStuff() );
11.   }
12. }
```

which statement is true?

A. The code compiles and runs, with output `this 420 parent 420`.

B. If line 8 is removed, the code will compile and run.

C. If line 10 is removed, the code will compile and run.

D. Both lines 8 and 10 must be removed for the code to compile.

E. An exception is thrown at runtime.

Declaration Rules (Sun Objective 4.1)

7. Given the following,

```
1.   interface Count {
2.     short counter = 0;
3.     void countUp();
4.   }
5.   public class TestCount implements Count {
6.
7.     public static void main(String [] args) {
8.       TestCount t = new TestCount();
9.       t.countUp();
10.    }
```

```
11.    public void countUp() {
12.      for (int x = 6; x>counter; x--, ++counter) {
13.         System.out.print(" " + counter);
14.      }
15.    }
16.  }
```

what is the result?

A. 0 1 2

B. 1 2 3

C. 0 1 2 3

D. 1 2 3 4

E. Compilation fails

F. An exception is thrown at runtime

8. Given the following,

```
1.   import java.util.*;
2.   public class NewTreeSet2 extends NewTreeSet {
3.     public static void main(String [] args) {
4.       NewTreeSet2 t = new NewTreeSet2();
5.       t.count();
6.     }
7.   }
8.   protected class NewTreeSet {
9.     void count() {
10.      for (int x = 0; x < 7; x++,x++ ) {
11.        System.out.print(" " + x);
12.      }
13.    }
14.  }
```

what is the result?

A. 0 2 4

B. 0 2 4 6

C. Compilation fails at line 4

D. Compilation fails at line 5

E. Compilation fails at line 8

F. Compilation fails at line 10

9. Given the following,

```
1.
2.   public class NewTreeSet extends java.util.TreeSet{
3.     public static void main(String [] args) {
4.       java.util.TreeSet t = new java.util.TreeSet();
5.       t.clear();
6.     }
7.     public void clear() {
8.        TreeMap m = new TreeMap();
9.        m.clear();
10.    }
11.  }
```

which two statements, added independently at line 1, allow the code to compile? (Choose two.)

A. No statement is required

B. `import java.util.*;`

C. `import.java.util.Tree*;`

D. `import java.util.TreeSet;`

E. `import java.util.TreeMap;`

10. Which two are valid declarations within an interface? (Choose two.)

A. `public static short stop = 23;`

B. `protected short stop = 23;`

C. `transient short stop = 23;`

D. `final void madness(short stop);`

E. `public Boolean madness(long bow);`

F. `static char madness(double duty);`

11. Which of the following class level (*nonlocal*) variable declarations will not compile?

A. `protected int a;`

B. `transient int b = 3;`

C. `public static final int c;`

D. `volatile int d;`

E. `private synchronized int e;`

Interface Implementation (Sun Objective 4.2)

12. Given the following,

```
1.  interface DoMath {
2.     double getArea(int rad); }
3.
4.   interface MathPlus {
5.     double getVol(int b, int h); }
6.
7.
8.
```

which two code fragments inserted at lines 7 and 8 will compile? (Choose two.)

A. `class AllMath extends DoMath { public double getArea(int r); }`

B. `interface AllMath implements MathPlus { public double getVol(int x, int y); }`

C. `interface AllMath extends DoMath { public float getAvg(int h, int l); }`

D. `class AllMath implements MathPlus { public double getArea(int rad); }`

E. `abstract class AllMath implements DoMath, MathPlus { public double getArea(int rad) { return rad * rad * 3.14; } }`

13. Which three are valid method signatures in an interface? (Choose three.)

A. `private int getArea();`

B. `public float getVol(float x);`

C. `public void main(String [] args);`

D. `public static void main(String [] args);`

E. `boolean setFlag(Boolean [] test []);`

14. Which two statements are true for any concrete class implementing the `java.lang.Runnable` interface? (Choose two.)

A. You can extend the `Runnable` interface as long as you override the `public run()` method.

B. The class must contain a method called `run()` from which all code for that thread will be initiated.

C. The class must contain an empty `public void` method named `run()`.

D. The class must contain a `public void` method named `runnable()`.

E. The class definition must include the words `implements Threads` and contain a method called `run()`.

F. The mandatory method must be `public`, with a return type of `void`, must be called `run()`, and cannot take any arguments.

15. Given the following,

```
1.   interface Base {
2.     boolean m1 ();
3.     byte m2(short s);
4.   }
```

which two code fragments will compile? (Choose two.)

A. `interface Base2 implements Base {}`

B.
```
abstract class Class2 extends Base {
  public boolean m1() { return true; } }
```

C. `abstract class Class2 implements Base { }`

D.
```
abstract class Class2 implements Base {
  public boolean m1() { return (7 > 4); } }
```

E.
```
class Class2 implements Base {
  boolean m1() { return false; }
  byte m2(short s) { return 42; } }
```

SELF TEST ANSWERS

Declarations and Modifiers

1. ☑ **E.** `default access` is the "package oriented" access modifier.
   ☒ **A** and **C** are wrong because `public` and `protected` are less restrictive. **B** and **D** are wrong because `abstract` and `synchronized` are not access modifiers.

2. ☑ **C.** The `private` access modifier limits access to members of the same class.
   ☒ **A, B, D, E,** and **F** are wrong because `protected` and `default` are the wrong access modifiers, and `final, static`, and `volatile` are modifiers but not access modifiers.

3. ☑ **A, C,** and **E. A** and **C** are correct, because an `abstract` class does not need to implement any of its superclass' methods. **E** is correct because as it stands, it is a valid concrete extension of class A.
   ☒ **B** is wrong because an `abstract` class does not need to implement any of its superclass' methods. **D** is wrong because a class that extends another class is free to add new methods. **F** is wrong because it is legal to extend an `abstract` class from a concrete class.

4. ☑ **C, F.** Both are legal class declarations.
   ☒ **A** is wrong because a class cannot be `abstract` and `final`—there would be no way to use such a class. **B** is wrong because interfaces and classes cannot be marked as `static`. **D** and **E** are wrong because classes and interfaces cannot be marked as `protected`.

5. ☑ **E.** Statements 1, 2, 4, 5, and 6 are legal declarations.
   ☒ **A, B, C, D,** and **F** are incorrect because the only illegal declaration is 3; `transient` applies only to variable declarations, not to method declarations. As you can see from these other examples, method declarations can be very extensive.

6. ☑ **C.** The ParentUtil instance `p` cannot be used to access the `doStuff()` method. Because `doStuff()` has `protected` access, and the ChildUtil class is not in the same package as the ParentUtil class, `doStuff()` can be accessed only by instances of the ChildUtil class (a subclass of `ParentUtil`).
   ☒ **A, B, D,** and **E** are incorrect because of the access rules described previously.

Declaration Rules

7. ☑ **E.** The code will not compile because the variable `counter` is an interface variable that is by default `final static`. The compiler will complain at line 12 when the code attempts to

increment `counter`.

☒ **A**, **B**, **C**, and **D** are incorrect because of the explanation given above.

8. ☑ **E.** Nonnested classes cannot be marked `protected` (or `final` for that matter), so the compiler will fail at line 8.

☒ **A**, **B**, **C**, and **D** are incorrect because of the explanation given above.

9. ☑ **B** and **E.** TreeMap is the only class that must be imported. TreeSet does not need an import statement because it is described with a fully qualified name.

☒ **A** is incorrect because TreeMap must be imported. **C** is incorrect syntax for an import statement. **D** is incorrect because it will not import TreeMap, which is required.

10. ☑ **A** and **E** are valid interface declarations.

☒ **B** and **C** are incorrect because interface variables cannot be either `protected` or `transient`. **D** and **F** are incorrect because interface methods cannot be `final` or `static`.

11. ☑ **E** will not compile; the `synchronized` modifier applies only to methods.

☒ **A** and **B** will compile because `protected` and `transient` are legal variable modifiers. **C** will compile because when a variable is declared `final` it does not have to be initialized with a value at the same time. **D** will compile because `volatile` is a proper variable modifier.

Interface Implementation

12. ☑ **C** and **E. C** are **E** are correct because interfaces and abstract classes do not need to fully implement the interfaces they extend or implement (respectively).

☒ **A** is incorrect because a class cannot extend an interface. **B** is incorrect because an interface cannot implement anything. **D** is incorrect because the method being implemented is from the wrong interface.

13. ☑ **B**, **C**, and **E.** These are all valid interface method signatures.

☒ **A**, is incorrect because an interface method must be `public`; if it is not explicitly declared `public` it will be made `public` implicitly. **D** is incorrect because interface methods cannot be `static`.

14. ☑ **B** and **F.** When a thread's `run()` method completes, the thread will die. The `run()` method must be declared `public void` and not take any arguments.

☒ **A** is incorrect because classes can never extend interfaces. **C** is incorrect because the

`run()` method is typically not empty; if it were, the thread would do nothing. **D** is incorrect because the mandatory method is `run()`. **E** is incorrect because the class implements `Runnable`.

15. ☑ **C** and **D. C** is correct because an `abstract` class doesn't have to implement any or all of its interface's methods. **D** is correct because the method is correctly implemented ((7 > 4) is a `boolean`).

☒ **A** is incorrect because interfaces don't implement anything. **B** is incorrect because classes don't extend interfaces. **E** is incorrect because interface methods are implicitly `public`, so the methods being implemented must be `public`.