1

Declarations and Access Control

CERTIFICATION OBJECTIVES

- Declare Classes & Interfaces
- Develop Interfaces & Abstract Classes
- Use Primitives, Arrays, Enums, & Legal Identifiers
- Use Static Methods, JavaBeans Naming, & Var-Args

✓ Two-Minute Drill

Q&A Self Test

We assume that because you're planning on becoming certified, you already know the basics of Java. If you're completely new to the language, this chapter—and the rest of the book—will be confusing; so be sure you know at least the basics of the language before diving into this book. That said, we're starting with a brief, high-level refresher to put you back in the Java mood, in case you've been away for awhile.

Java Refresher

A Java program is mostly a collection of *objects* talking to other objects by invoking each other's *methods*. Every object is of a certain *type*, and that type is defined by a *class* or an *interface*. Most Java programs use a collection of objects of many different types.

- **Class** A template that describes the kinds of state and behavior that objects of its type support.
- **Object** At runtime, when the Java Virtual Machine (JVM) encounters the `new` keyword, it will use the appropriate class to make an object which is an instance of that class. That object will have its own state, and access to all of the behaviors defined by its class.
- **State (instance variables)** Each object (instance of a class) will have its own unique set of instance variables as defined in the class. Collectively, the values assigned to an object's instance variables make up the object's state.
- **Behavior (methods)** When a programmer creates a class, she creates methods for that class. Methods are where the class' logic is stored. Methods are where the real work gets done. They are where algorithms get executed, and data gets manipulated.

Identifiers and Keywords

All the Java components we just talked about—classes, variables, and methods—need names. In Java these names are called *identifiers*, and, as you might expect, there are rules for what constitutes a legal Java identifier. Beyond what's *legal*,

though, Java programmers (and Sun) have created *conventions* for naming methods, variables, and classes.

Like all programming languages, Java has a set of built-in *keywords*. These keywords must *not* be used as identifiers. Later in this chapter we'll review the details of these naming rules, conventions, and the Java keywords.

Inheritance

Central to Java and other object-oriented languages is the concept of *inheritance*, which allows code defined in one class to be reused in other classes. In Java, you can define a general (more abstract) superclass, and then extend it with more specific subclasses. The superclass knows nothing of the classes that inherit from it, but all of the subclasses that inherit from the superclass must explicitly declare the inheritance relationship. A subclass that inherits from a superclass is automatically given accessible instance variables and methods defined by the superclass, but is also free to override superclass methods to define more specific behavior.

For example, a Car *super*class class could define general methods common to all automobiles, but a Ferrari *sub*class could override the `accelerate()` method.

Interfaces

A powerful companion to inheritance is the use of interfaces. Interfaces are like a 100-percent abstract superclass that defines the methods a subclass must support, but not *how* they must be supported. In other words, an `Animal` interface might declare that all `Animal` implementation classes have an `eat()` method, but the `Animal` interface doesn't supply any logic for the `eat()` method. That means it's up to the classes that implement the Animal interface to define the actual code for how that particular Animal type behaves when its `eat()` method is invoked.

Finding Other Classes

As we'll see later in the book, it's a good idea to make your classes *cohesive*. That means that every class should have a focused set of responsibilities. For instance, if you were creating a zoo simulation program, you'd want to represent aardvarks with one class, and zoo visitors with a different class. In addition, you might have a Zookeeper class, and a Popcorn vendor class. The point is that you don't want a class that has both Aardvark *and* Popcorn behaviors (more on that in Chapter 2).

Even a simple Java program uses objects from many different classes: some that *you* created, and some built by others (such as Sun's Java API classes). Java organizes classes into *packages*, and uses *import* statements to give programmers a consistent

way to manage naming of, and access to, classes they need. The exam covers a *lot* of concepts related to packages and class access; we'll explore the details in this—and later—chapters.

CERTIFICATION OBJECTIVE

Identifiers & JavaBeans (Objectives 1.3 and 1.4)

1.3 Develop code that declares, initializes, and uses primitives, arrays, enums, and objects as static, instance, and local variables. Also, use legal identifiers for variable names.

1.4 Develop code that declares both static and non-static methods, and—if appropriate—use method names that adhere to the JavaBeans naming standards. Also develop code that declares and uses a variable-length argument list.

Remember that when we list one or more Certification Objectives in the book, as we just did, it means that the following section covers at least some part of that objective. Some objectives will be covered in several different chapters, so you'll see the same objective in more than one place in the book. For example, this section covers declarations, identifiers, and JavaBeans naming, but *using* the things you declare is covered primarily in later chapters.

So, we'll start with Java identifiers. The three aspects of Java identifiers that we cover here are

- **Legal Identifiers** The rules the compiler uses to determine whether a name is legal.
- **Sun's Java Code Conventions** Sun's recommendations for naming classes, variables, and methods. We typically adhere to these standards throughout the book, except when we're trying to show you how a tricky exam question might be coded. You won't be asked questions about the Java Code Conventions, but we strongly recommend that programmers use them.
- **JavaBeans Naming Standards** The naming requirements of the JavaBeans specification. You don't need to study the JavaBeans spec for the exam, but you do need to know a few basic JavaBeans naming rules we cover in this chapter.

Legal Identifiers

Technically, legal identifiers must be composed of only Unicode characters, numbers, currency symbols, and connecting characters (like underscores). The exam doesn't dive into the details of which ranges of the Unicode character set are considered to qualify as letters and digits. So, for example, you won't need to know that Tibetan digits range from `\u0420` to `\u0f29`. Here are the rules you *do* need to know:

- Identifiers must start with a letter, a currency character ($), or a connecting character such as the underscore (_). Identifiers cannot start with a number!
- After the first character, identifiers can contain any combination of letters, currency characters, connecting characters, or numbers.
- In practice, there is no limit to the number of characters an identifier can contain.
- You can't use a Java keyword as an identifier. Table 1-1 lists all of the Java keywords including one new one for 5.0, `enum`.
- Identifiers in Java are case-sensitive; `foo` and `FOO` are two different identifiers.

Examples of legal and illegal identifiers follow, first some legal identifiers:

```
int _a;
int $c;
int ______2_w;
int _$;
int this_is_a_very_detailed_name_for_an_identifier;
```

The following are illegal (it's your job to recognize why):

```
int :b;
int -d;
int e#;
int .f;
int 7g;
```

TABLE 1-1 Complete List of Java Keywords (`assert` added in 1.4, `enum` added in 1.5)

abstract	boolean	break	byte	case	catch
char	class	const	continue	default	do
double	else	extends	final	finally	float
for	goto	if	implements	import	instanceof
int	interface	long	native	new	package
private	protected	public	return	short	static
strictfp	super	switch	synchronized	this	throw
throws	transient	try	void	volatile	while
assert	enum				

Sun's Java Code Conventions

Sun estimates that over the lifetime of a standard piece of code, 20 percent of the effort will go into the original creation and testing of the code, and 80 percent of the effort will go into the subsequent maintenance and enhancement of the code. Agreeing on, and coding to, a set of code standards helps to reduce the effort involved in testing, maintaining, and enhancing any piece of code. Sun has created a set of coding standards for Java, and published those standards in a document cleverly titled "Java Code Conventions," which you can find at `java.sun.com`. It's a great document, short and easy to read and we recommend it highly.

That said, you'll find that many of the questions in the exam don't follow the code conventions, because of the limitations in the test engine that is used to deliver the exam internationally. One of the great things about the Sun certifications is that the exams are administered uniformly throughout the world. In order to achieve that, the code listings that you'll see in the real exam are often quite cramped, and do not follow Sun's code standards. In order to toughen you up for the exam, we'll often present code listings that have a similarly cramped look and feel, often indenting our code only two spaces as opposed to the Sun standard of four.

We'll also jam our curly braces together unnaturally, and sometimes put several statements on the same line…ouch! For example:

```
1. class Wombat implements Runnable {
2.   private int i;
3.   public synchronized void run() {
4.     if (i%5 != 0) { i++; }
5.     for(int x=0; x<5; x++, i++)
```

```
 6.       { if (x > 1) Thread.yield(); }
 7.     System.out.print(i + " ");
 8.   }
 9.   public static void main(String[] args) {
10.     Wombat n = new Wombat();
11.     for(int x=100; x>0; --x) { new Thread(n).start(); }
12. } }
```

Consider yourself forewarned—you'll see lots of code listings, mock questions, and real exam questions that are this sick and twisted. Nobody wants you to write your code like this. Not your employer, not your coworkers, not us, not Sun, and not the exam creation team! Code like this was created only so that complex concepts could be tested within a universal testing tool. The one standard that *is* followed as much as possible in the real exam are the naming standards. Here are the naming standards that Sun recommends, and that we use in the exam and in most of the book:

- **Classes and interfaces** The first letter should be capitalized, and if several words are linked together to form the name, the first letter of the inner words should be uppercase (a format that's sometimes called "camelCase"). For classes, the names should typically be nouns. For example:

  ```
  Dog
  Account
  PrintWriter
  ```

  For interfaces, the names should typically be adjectives like

  ```
  Runnable
  Serializable
  ```

- **Methods** The first letter should be lowercase, and then normal camelCase rules should be used. In addition, the names should typically be verb-noun pairs. For example:

  ```
  getBalance
  doCalculation
  setCustomerName
  ```

- **Variables** Like methods, the camelCase format should be used, starting with a lowercase letter. Sun recommends short, meaningful names, which sounds good to us. Some examples:

```
buttonWidth
accountBalance
myString
```

- **Constants** Java constants are created by marking variables `static` and `final`. They should be named using uppercase letters with underscore characters as separators:

```
MIN_HEIGHT
```

JavaBeans Standards

The JavaBeans spec is intended to help Java developers create Java components that can be easily used by other Java developers in a visual Integrated Development Environment (IDE) tool (like Eclipse or NetBeans). As a Java programmer, you want to be able to use components from the Java API, but it would be great if you could also buy the Java component you want from "Beans 'R Us," that software company down the street. And once you've found the components, you'd like to be able to access them through a development tool in such a way that you don't have to write all your code from scratch. By using naming rules, the JavaBeans spec helps guarantee that tools can recognize and use components built by different developers. The JavaBeans API is quite involved, but you'll need to study only a few basics for the exam.

First, JavaBeans are Java classes that have *properties*. For our purposes, think of properties as `private` instance variables. Since they're `private`, the only way they can be accessed from outside of their class is through *methods* in the class. The methods that change a property's value are called *setter* methods, and the methods that retrieve a property's value are called *getter* methods. The JavaBean naming rules that you'll need to know for the exam are the following:

JavaBean Property Naming Rules

- If the property is not a boolean, the getter method's prefix must be *get*. For example, `getSize()` is a valid JavaBeans getter name for a property named "size." Keep in mind that you do not need to have a variable named *size*

(although some IDEs expect it). The name of the property is *inferred* from the getters and setters, not through any variables in your class. What you return from `getSize()` is up to you.

- If the property is a boolean, the getter method's prefix is either `get` or `is`. For example, `getStopped()` or `isStopped()` are both valid JavaBeans names for a boolean property.
- The setter method's prefix must be *set*. For example, `setSize()` is the valid JavaBean name for a property named *size*.
- To complete the name of a getter or setter method, change the first letter of the property name to uppercase, and then append it to the appropriate prefix (`get`, `is`, or `set`).
- Setter method signatures must be marked `public`, with a `void` return type and an argument that represents the property type.
- Getter method signatures must be marked `public`, take no arguments, and have a return type that matches the argument type of the setter method for that property.

Second, the JavaBean spec supports *events*, which allow components to notify each other when something happens. The event model is often used in GUI applications when an event like a mouse click is multicast to many other objects that may have things to do when the mouse click occurs. The objects that receive the information that an event occurred are called *listeners*. For the exam, you need to know that the methods that are used to add or remove listeners from an event must also follow JavaBean naming standards:

JavaBean Listener Naming Rules

- Listener method names used to "register" a listener with an event source must use the prefix `add`, followed by the listener type. For example, `addActionListener()` is a valid name for a method that an event source will have to allow others to register for Action events.
- Listener method names used to remove ("unregister") a listener must use the prefix `remove`, followed by the listener type (using the same rules as the registration add method).
- The type of listener to be added or removed must be passed as the argument to the method.

Examples of valid JavaBean method signatures are

```
public void setMyValue(int v)
public int getMyValue()
public boolean isMyStatus()
public void addMyListener(MyListener m)
public void removeMyListener(MyListener m)
```

Examples of *invalid* JavaBean method signatures are

```
void setCustomerName(String s)         // must be public
public void modifyMyValue(int v)       // can't use 'modify'
public void addXListener(MyListener m) // listener type mismatch
```

exam Watch

The objective says you have to know legal identifiers only for variable names, but the rules are the same for ALL Java components. So remember that a legal identifier for a variable is also a legal identifier for a method or a class. However, you need to distinguish between legal identifiers and naming conventions, such as the JavaBeans standards, that indicate how a Java component should be named. In other words, you must be able to recognize that an identifier is legal even if it doesn't conform to naming standards. If the exam question is asking about naming conventions—not just whether an identifier will compile—JavaBeans will be mentioned explicitly.

CERTIFICATION OBJECTIVE

Declare Classes (Exam Objective 1.1)

1.1 Develop code that declares classes (including abstract and all forms of nested classes), interfaces, and enums, and includes the appropriate use of package and import statements (including static imports).

When you write code in Java, you're writing classes or interfaces. 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 declare and modify (or not) a class. 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 section's objectives before taking the exam.

Source File Declaration Rules

Before we dig into class declarations, let's do a quick review of the rules associated with declaring classes, `import` statements, and `package` statements in a source file:

- There can be only one `public` class per source code file.
- Comments can appear at the beginning or end of any line in the source code file; they are independent of any of the positioning rules discussed here.
- If there *is* a `public` class in a file, the name of the file must match the name of the `public` class. For example, a class declared as `public class Dog { }` must be in a source code file named `Dog.java`.
- If the class is part of a package, the `package` statement must be the first line in the source code file, before any `import` statements that may be present.
- If there are `import` statements, they must go *between* the `package` statement (if there is one) 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.
- `import` and `package` statements apply to *all* classes within a source code file. In other words, there's no way to declare multiple classes in a file and have them in different packages, or use different imports.
- A file can have more than one nonpublic class.

- Files with no public classes can have a name that does not match any of the classes in the file.

In Chapter 10 we'll go into a lot more detail about the rules involved with declaring and using imports, packages, and a feature new to Java 5, static imports.

Class Declarations and Modifiers

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. 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`.
- Non-access 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! 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 when we try to compile the Tea file we get something like:

```
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. Apart from using fully qualified class names, which we'll cover in Chapter 10, you can do one of two things to make this work. You could put both classes in the same package, or you could declare Beverage as `public`, as the next section describes.

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 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) 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.

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 nonaccess modifiers. You're free to use `strictfp` in combination with `final`, for example, 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 IEEE 754 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 IEEE 754 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 object-oriented (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.

A 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 an error something like

```
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). (Note, however, that you can compile and execute an `abstract` class, as long as you don't try

to make an instance of it.) 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 `BMWBoxster` and `SubaruOutback`. 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 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 something like this:

```
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.

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 change the semicolon to code (like a curly brace pair). Remember, 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* earlier, don't send an e-mail. We'd like to see *you* work it into a programmer certification book.)

Coding with abstract class types (including interfaces, discussed later in this chapter) lets you take advantage of polymorphism, and gives you the greatest degree of flexibility and extensibility. You'll learn more about polymorphism in Chapter 2.

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 1-1

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.

CERTIFICATION OBJECTIVE

Declare Interfaces (Exam Objectives 1.1 and 1.2)

1.1 Develop code that declares classes (including abstract and all forms of nested classes), interfaces, and enums, and includes the appropriate use of package and import statements (including static imports).

1.2 Develop code that declares an interface. Develop code that implements or extends one or more interfaces. Develop code that declares an abstract class. Develop code that extends an abstract class.

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."

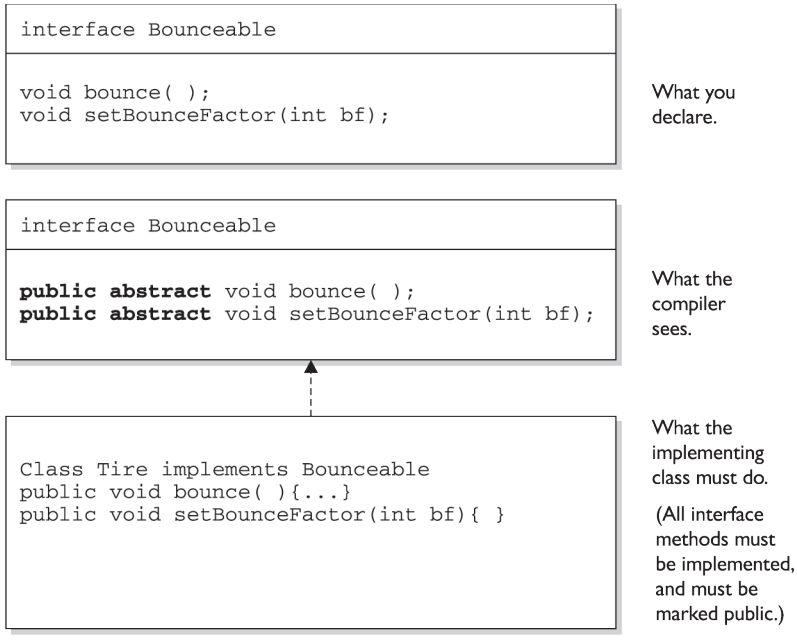
By defining an interface for Bounceable, any class that wants to be treated as a Bounceable thing can simply implement the `Bounceable` interface and provide code for the interface's two methods.

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 1-1 illustrates the relationship between interfaces and classes.

FIGURE 1-1

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 following form:

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

But while an `abstract` class can define both `abstract` and non-`abstract` methods, an interface can have only `abstract` methods. Another way interfaces differ from `abstract` classes is that interfaces have very little flexibility in how the methods and variables defined in the interface are declared. These rules are strict:

- All interface methods are implicitly `public` and `abstract`. In other words, 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.

- Interface methods must not be `static`.
- Because interface methods are abstract, they cannot be marked `final`, `strictfp`, or `native`. (More on these modifiers later.)
- 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 2 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.

Look for interface methods declared with any combination of `public`, `abstract`, or no modifiers. For example, the following five method declarations, if declared within their own interfaces, 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
                        // together, and abstract is implied
static void bounce();   // interfaces define instance methods
private void bounce();  // interface methods are always public
protected void bounce();    // (same as above)
```

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.

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.

You need to remember one key rule 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: **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 code (assume two separate files):

```
interface Foo {
  int BAR = 42;
  void go();
}

class Zap implements Foo {
  public void go() {
     BAR = 27;
  }
}
```

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. So the BAR = 27 assignment will not compile.

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,
                           // 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;   // 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.***

CERTIFICATION OBJECTIVE

Declare Class Members (Objectives 1.3 and 1.4)

1.3 Develop code that declares, initializes, and uses primitives, arrays, enums, and objects as static, instance, and local variables. Also, use legal identifiers for variable names.

1.4 Develop code that declares both static and non-static methods, and—if appropriate—use method names that adhere to the JavaBeans naming standards. Also develop code that declares and uses a variable-length argument list.

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.

Access Modifiers

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.

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 1-2 compares a class inheriting a member of another class, and accessing a member of another class using a reference of an instance of that class.

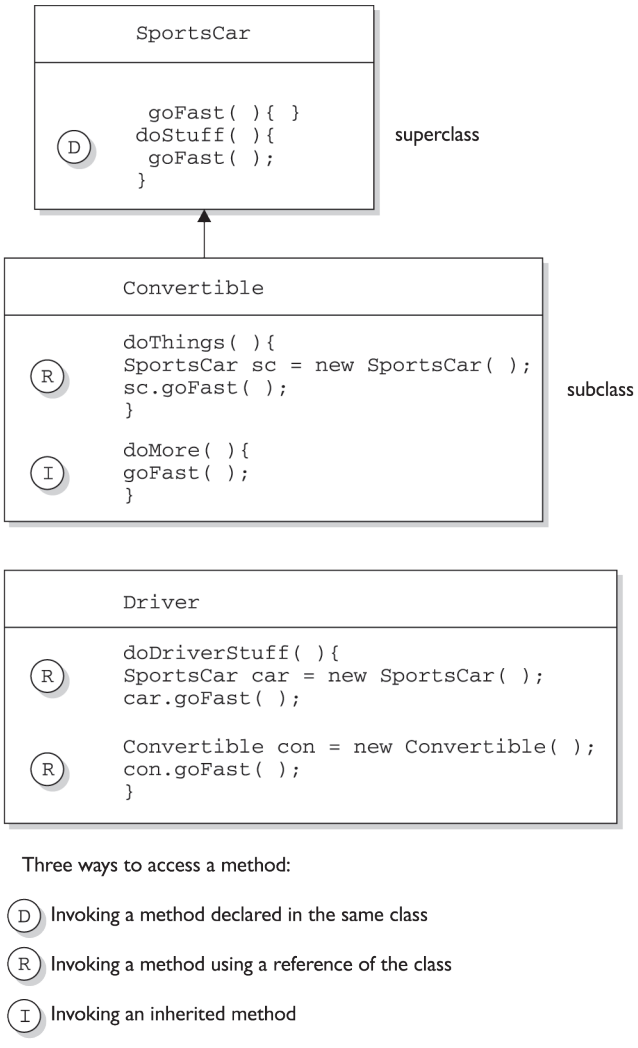
Much of access control (both types) centers on whether the two classes involved are in the same or different packages. Don't forget, though, if class A *itself* can't be accessed by class B, then no members within class A can be accessed by class B.

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).

FIGURE 1-2 Comparison of inheritance vs. dot operator for member access.



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:

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

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
    return "fun";
  }
}
```

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 public
    System.out.println(r.doRooThings()); //Compiler error!
  }
}
```

If we try to compile UseARoo, we get a compiler error something like this:

```
cannot find symbol
symbol  : method doRooThings()
```

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
    return "fun";
   }
}
```

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 an error something like this:

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

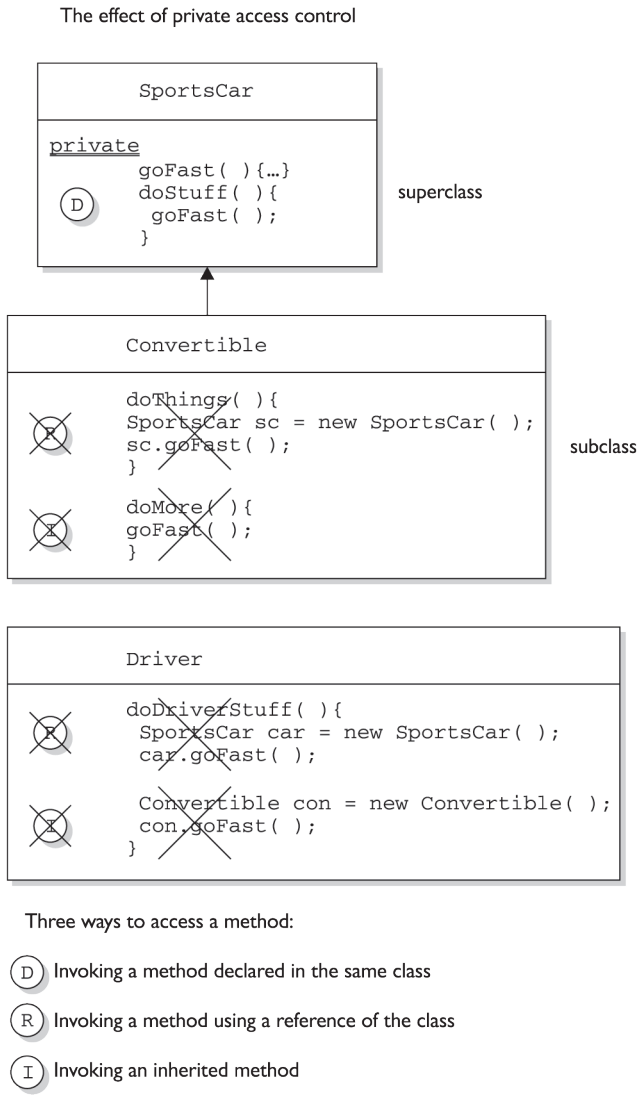
Although you're allowed to mark instance variables as `public`***, in practice it's nearly always best to keep all variables*** `private` ***or*** `protected`***. If variables need to be changed, set, or read, programmers should use public accessor methods, so that code in any other class has to ask to get or set a variable (by going through a method), rather than access it directly. JavaBean-compliant accessor methods take the form get<propertyName> or, for booleans, is<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. Or not.) Chapter 2 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 2, but for now just remember that a method marked `private` cannot be overridden. Figure 1-3 illustrates the effects of the `public` and `private` modifiers on classes from the same or different packages.

FIGURE 1-3

Effects of `public` and `private` access



Protected and Default Members

The `protected` and default access control levels are almost identical, but with one critical difference. A *default* member may 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 first 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.

```
No method matching testIt() found in class
certification.OtherClass.   o.testIt();
```

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. If the `protected` keyword is used to define a member, any subclass of the class declaring the member can access it *through inheritance*. 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 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 see the `protected` member only 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>)

Protected Details

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 *inside* 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, any time 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.Par-
ent
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, with the exception of subclasses of 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 and its subclasses can access it. Figure 1-4 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`.

Default Details

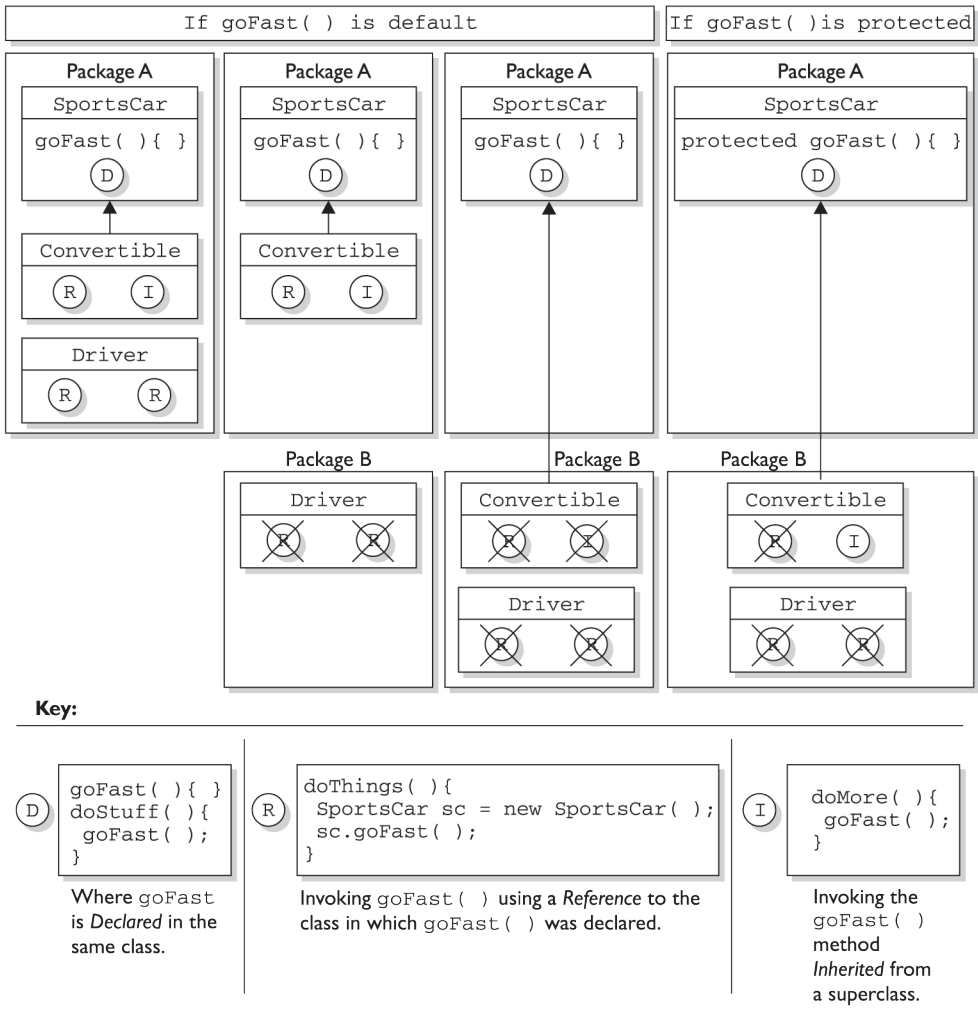
Let's start with the default behavior of a member in a superclass. We'll modify the Parent's member `x` to make it default.

```
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.

FIGURE 1-4

Effects of `protected` access



When we compile the child file, we get an error something like this:

```
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 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 to subclasses only if those subclasses are in the same package as the superclass.

Local Variables and Access Modifiers

Can access modifiers be applied to local variables? NO!

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 1-2 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 1-2 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 a subclass in the same package	Yes	Yes	Yes	No
From a subclass outside the same package	Yes	Yes, *through inheritance*	No	No
From any non-subclass class outside the package	Yes	No	No	No

Nonaccess Member Modifiers

We've discussed member access, which refers to whether 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 section 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* 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 something like this:

```
%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 recNumber) {}
```

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 earlier section "Abstract Classes," an abstract method declaration doesn't even have curly braces for where the implementation code goes, but instead closes with a semicolon. In other words, *it has no method body*. 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.

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

Notice that the abstract method ends with a semicolon instead of curly braces. **It is illegal to have even a single abstract method in a class that is not explicitly 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:

```
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. In other words, the method has a method body.
- 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 this:

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:

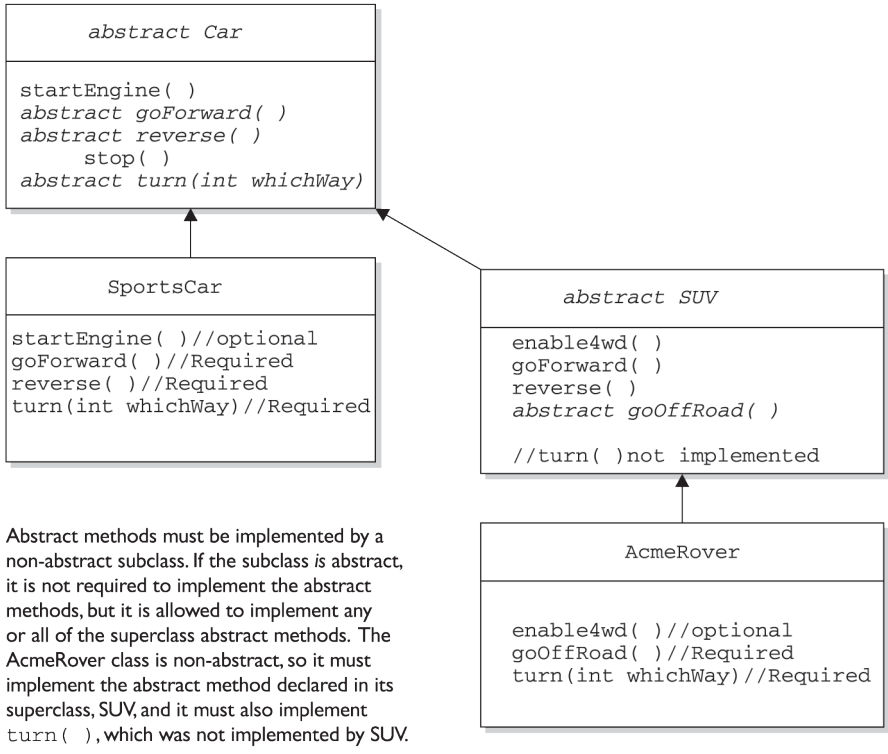
```
public abstract class Vehicle {
  private String type;
  public abstract void goUpHill();  // Abstract method
  public String getType() {         // Non-abstract method
    return type;
  }
}

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 1-5 illustrates the effects of the `abstract` modifier on concrete and `abstract` subclasses.

FIGURE 1-5 The effects of the `abstract` modifier on concrete and `abstract` subclasses



Look for concrete classes that don't provide method implementations for `abstract` methods of the superclass. 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 2.

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`.

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 11, 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).

Native Methods

The `native` modifier indicates that a method is implemented in platform-dependent code, often in 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) and that native can be applied only to *methods*—not classes, not variables, just methods. Note that a `native` method's body must be a semicolon (;) (like abstract methods), indicating that the implementation is omitted.

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 IEEE 754 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 want to study the IEEE 754 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 method declaration, and that a variable can never be declared `strictfp`.

Methods with Variable Argument Lists (var-args)

As of 5.0, Java allows you to create methods that can take a variable number of arguments. Depending on where you look, you might hear this capability referred to as "variable-length argument lists," "variable arguments," "var-args," "varargs," or our personal favorite (from the department of obfuscation), "variable arity parameter." They're all the same thing, and we'll use the term "var-args" from here on out.

As a bit of background, we'd like to clarify how we're going to use the terms "argument" and "parameter" throughout this book.

- **arguments** The things you specify between the parentheses when you're *invoking* a method:

```
doStuff("a", 2);  // invoking doStuff, so a & 2 are arguments
```

- **parameters** The things in the *method's signature* that indicate what the method must receive when it's invoked:

```
void doStuff(String s, int a) { } // we're expecting two
                                  // parameters: String and int
```

We'll cover using var-arg methods more in the next few chapters, for now let's review the declaration rules for var-args:

- **Var-arg type** When you declare a var-arg parameter, you must specify the type of the argument(s) this parameter of your method can receive. (This can be a primitive type or an object type.)
- **Basic syntax** To declare a method using a var-arg parameter, you follow the type with an ellipsis (...), a space, and then the name of the array that will hold the parameters received.
- **Other parameters** It's legal to have other parameters in a method that uses a var-arg.
- **Var-args limits** The var-arg must be the last parameter in the method's signature, and you can have only one var-arg in a method.

Let's look at some legal and illegal var-arg declarations:

Legal:

```
void doStuff(int... x) { }  // expects from 0 to many ints
                            // as parameters
void doStuff2(char c, int... x)  { }  // expects first a char,
                                      // then 0 to many ints
void doStuff3(Animal... animal) { }   // 0 to many Animals
```

Illegal:

```
void doStuff4(int x...) { }             // bad syntax
void doStuff5(int... x, char... y) { }  // too many var-args
void doStuff6(String... s, byte b) { }  // var-arg must be last
```

Constructor Declarations

In Java, objects are constructed. Every time you make a new object, at least one constructor is invoked. Every class has a constructor, although if you don't create one explicitly, the compiler will build one for you. There are tons of rules concerning

constructors, and we're saving our detailed discussion for Chapter 2. For now, let's focus on the basic declaration rules. Here's a simple example:

```
class Foo {
  protected Foo() { }          // this is Foo's constructor

  protected void Foo() { }     // this is a badly named,
                               // but legal, method
}
```

The first thing to notice is that constructors look an awful lot like methods. A key difference is that a constructor can't ever, ever, ever, have a return type…ever! Constructor declarations can however have all of the normal access modifiers, and they can take arguments (including var-args), just like methods. The other BIG RULE, to understand about constructors is that they must have the same name as the class in which they are declared. Constructors can't be marked `static` (they are after all associated with object instantiation), they can't be marked `final` or `abstract` (because they can't be overridden). Here are some legal and illegal constructor declarations:

```
class Foo2 {

  // legal constructors

  Foo2() { }
  private Foo2(byte b) { }
  Foo2(int x) { }
  Foo2(int x, int... y) { }

  // illegal constructors

  void Foo2() { }            // it's a method, not a constructor
  Foo() { }                  // not a method or a constructor
  Foo2(short s);             // looks like an abstract method
  static Foo2(float f) { }   // can't be static
  final Foo2(long x) { }     // can't be final
  abstract Foo2(char c) { }  // can't be abstract
  Foo2(int... x, int t) { }  // bad var-arg syntax
 }
```

Variable Declarations

There are two types of variables in Java:

- **Primitives** A primitive can be one of eight types: `char`, `boolean`, `byte`, `short`, `int`, `long`, `double`, or `float`. Once a primitive has been declared, its primitive type can never change, although in most cases its value can change.
- **Reference variables** A reference variable is used to refer to (or access) an object. A reference variable is declared to be of a specific type and that type can never be changed. A reference variable can be used to refer to any object of the declared type, or of a *subtype* of the declared type (a compatible type). We'll talk a lot more about using a reference variable to refer to a subtype in Chapter 2, when we discuss polymorphism.

Declaring Primitives and Primitive Ranges

Primitive variables can be declared as class variables (statics), instance variables, method parameters, or local variables. You can declare one or more primitives, of the same primitive type, in a single line. In Chapter 3 we will discuss the various ways in which they can be initialized, but for now we'll leave you with a few examples of primitive variable declarations:

```
byte b;
boolean myBooleanPrimitive;
int x, y, z;                  // declare three int primitives
```

On previous versions of the exam you needed to know how to calculate ranges for all the Java primitives. For the current exam, you can skip some of that detail, but it's still important to understand that for the integer types the sequence from small to big is `byte`, `short`, `int`, `long`, and that `doubles` are bigger than `floats`.

You will also need to know that the number types (both integer and floating-point types) are all signed, and how that affects their ranges. First, let's review the concepts.

All six number types in Java are made up of a certain number of 8-bit bytes, and are *signed*, meaning they can be negative or positive. The leftmost bit (the most significant digit) is used to represent the sign, where a 1 means negative and 0 means positive, as shown in Figure 1-6. The rest of the bits represent the value, using two's complement notation.

FIGURE 1-6 The Sign bit for a byte

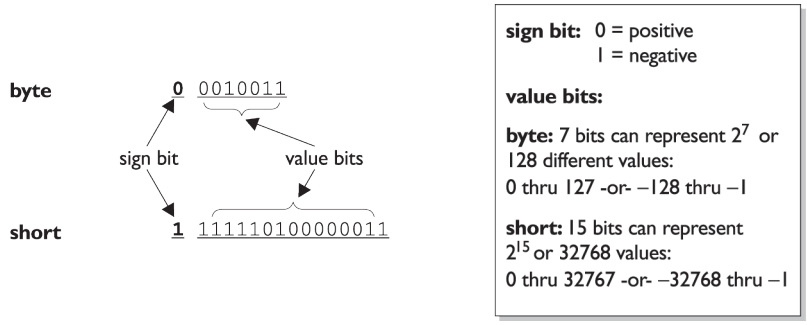


Table 1-3 shows the primitive types with their sizes and ranges. Figure 1-6 shows that with a byte, for example, there are 256 possible numbers (or 2^8). Half of these are negative, and half -1 are positive. The positive range is one less than the negative range because the number zero is stored as a positive binary number. We use the formula $-2^{(bits-1)}$ to calculate the negative range, and we use $2^{(bits-1)}-1$ for the positive range. Again, if you know the first two columns of this table, you'll be in good shape for the exam.

TABLE 1-3 Ranges of Numeric Primitives

Type	Bits	Bytes	Minimum Range	Maximum Range
`byte`	8	1	-2^7	2^7-1
`short`	16	2	-2^{15}	$2^{15}-1$
`int`	32	4	-2^{31}	$2^{31}-1$
`long`	64	8	-2^{63}	$2^{63}-1$
`float`	32	4	n/a	n/a
`double`	64	8	n/a	n/a

The range for floating-point numbers is complicated to determine, but luckily you don't need to know these for the exam (although you are expected to know that a double holds 64 bits and a float 32).

For `boolean` types there is not a range; a `boolean` can be only `true` or `false`. If someone asks you for the bit depth of a `boolean`, look them straight in the eye and say, "That's virtual-machine dependent." They'll be impressed.

The `char` type (a character) contains a single, 16-bit Unicode character. Although the extended ASCII set known as ISO Latin-1 needs only 8 bits (256 different characters), a larger range is needed to represent characters found in languages other than English. Unicode characters are actually represented by unsigned 16-bit integers, which means 2^{16} possible values, ranging from 0 to 65535 (2^{16})-1. You'll learn in Chapter 3 that because a `char` is really an integer type, it can be assigned to any number type large enough to hold 65535 (which means anything larger than a `short`. Although both `chars` and `shorts` are 16-bit types, remember that a `short` uses 1 bit to represent the sign, so fewer positive numbers are acceptable in a short).

Declaring Reference Variables

Reference variables can be declared as static variables, instance variables, method parameters, or local variables. You can declare one or more reference variables, of the same type, in a single line. In Chapter 3 we will discuss the various ways in which they can be initialized, but for now we'll leave you with a few examples of reference variable declarations:

```
Object o;
Dog myNewDogReferenceVariable;
String s1, s2, s3;               // declare three String vars.
```

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
}
```

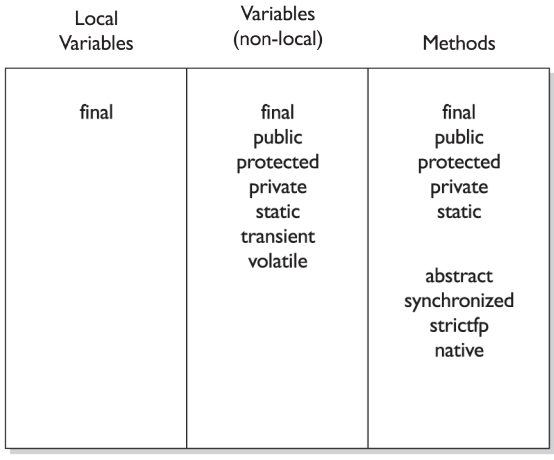
The preceding Employee class says that each employee instance will know its own name, title, and manager. In other words, each instance can have its own unique values for those three fields. If you see the term "field," "instance variable," "property," or "attribute," they mean virtually the same thing. (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`
- Cannot be marked `static`, because then they'd become class variables.

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 1-7 compares the way in which modifiers can be applied to methods vs. variables.

FIGURE 1-7 Comparison of modifiers on variables vs. methods



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 variables are always on the stack, not the heap. (We'll talk more about the stack and the heap in Chapter 3). 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 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 as you'll learn in Chapter 3 (but here's a preview), before a local variable can be *used*, it must be *initialized* with a value. For instance:

```
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. It's known as *shadowing*, as the following code demonstrates:

```
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 a parameter with the same name as the instance variable to which the parameter will be assigned.

The following (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 parameter 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, size. The size
                       // on the right is the parameter
    }
}
```

Array Declarations

In Java, arrays are objects that store multiple variables of the same type, or variables that are all subclasses of the same type. Arrays can hold either primitives or object

references, but the array itself will always be an object on the heap, even if the array is declared to hold primitive elements. In other words, there is no such thing as a primitive array, but you can make an array of primitives.

For the exam, you need to know three things:

- How to make an array reference variable (declare)
- How to make an array object (construct)
- How to populate the array with elements (initialize)

For this objective, you only need to know how to declare an array, we'll cover constructing and initializing arrays in Chapter 3.

on the job

Arrays are efficient, but many times you'll want to use one of the Collection types from java.util (including HashMap, ArrayList, and TreeSet). Collection classes offer more flexible ways to access an object (for insertion, deletion, reading, and so on) and unlike arrays, can expand or contract dynamically as you add or remove elements. There's a Collection type for a wide range of needs. Do you need a fast sort? A group of objects with no duplicates? A way to access a name-value pair? Chapter 7 covers them in more detail.

Arrays are declared by stating the type of elements the array will hold (an object or a primitive), followed by square brackets to either side of the identifier.

Declaring an Array of Primitives

```
int[] key;  // Square brackets before name (recommended)
int key []; // Square brackets after name (legal but less
            // readable)
```

Declaring an Array of Object References

```
Thread[] threads;  // Recommended
Thread threads []; // Legal but less readable
```

on the job

When declaring an array reference, you should always put the array brackets immediately after the declared type, rather than after the identifier (variable name). That way, anyone reading the code can easily tell that, for example, `key` ***is a reference to an*** `int` ***array object, and not an*** `int` ***primitive.***

We can also declare multidimensional arrays, which are in fact arrays of arrays. This can be done in the following manner:

```
String[][][] occupantName;
String[] ManagerName [];
```

The first example is a three-dimensional array (an array of arrays of arrays) and the second is a two-dimensional array. Notice in the second example we have one square bracket before the variable name and one after. This is perfectly legal to the compiler, proving once again that just because it's legal doesn't mean it's right.

exam Watch

It is never legal to include the size of the array in your declaration. Yes, we know you can do that in some other languages, which is why you might see a question or two that include code similar to the following:

```
int[5] scores;
```

The preceding code won't compile. Remember, the JVM doesn't allocate space until you actually instantiate the array object. That's when size matters.

In Chapter 3, we'll spend a lot of time discussing arrays, how to initialize and use them, and how to deal with multi-dimensional arrays…stay tuned!

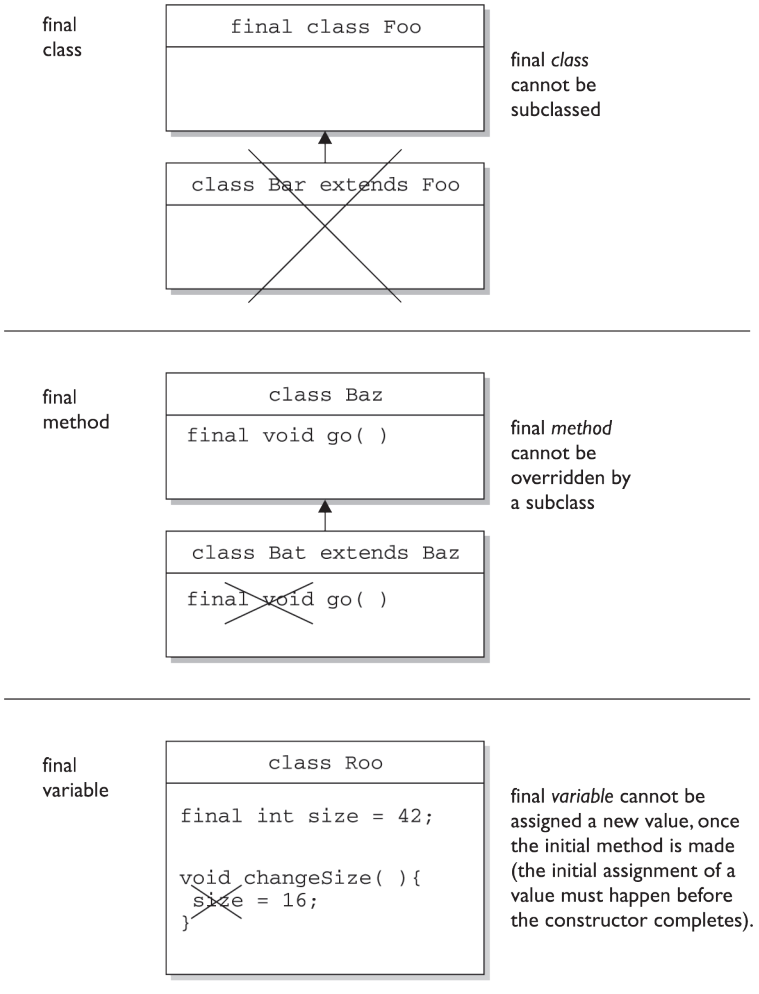
Final Variables

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 can be modified, but the reference variable cannot be changed. In other words, a `final` reference still allows you to modify the state of 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. We'll explain this in more detail in Chapter 3.

We've now covered how the `final` modifier can be applied to classes, methods, and variables. Figure 1-8 highlights the key points and differences of the various applications of `final`.

FIGURE 1-8 Effect of `final` on variables, methods, and classes



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 containing 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 I/O 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. Serialization has been added to the exam as of Java 5, and we'll cover it in great detail in Chapter 6.

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 11, 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 is used to create variables and methods that will exist independently of any instances created for the class. In other words, `static` members exist before you ever make a new instance of a class, and there will be only one copy of the `static` member regardless of the number of instances of that class. In other words, all instances of a given class share the same value for any given `static` variable. We'll cover `static` members in great detail in the next chapter.

Things you can mark as `static`:

- Methods
- Variables
- A class nested within another class, but not within a method (more on this in Chapter 8).
- Initialization blocks

Things you can't mark as `static`:

- Constructors (makes no sense; a constructor is used only to create instances)
- Classes (unless they are nested)
- Interfaces
- Method local inner classes (we'll explore this in Chapter 8)
- Inner class methods and instance variables
- Local variables

Declaring Enums

As of 5.0, Java lets you restrict a variable to having one of only a few pre-defined values—in other words, one value from an enumerated list. (The items in the enumerated list are called, surprisingly, `enums`.)

Using enums can help reduce the bugs in your code. For instance, in your coffee shop application you might want to restrict your size selections to BIG, HUGE, and OVERWHELMING. If you let an order for a LARGE or a GRANDE slip in, it might cause an error. Enums to the rescue. With the following simple declaration, you can guarantee that the compiler will stop you from assigning anything to a `CoffeeSize` except `BIG`, `HUGE`, or `OVERWHELMING`:

```
enum CoffeeSize { BIG, HUGE, OVERWHELMING };
```

From then on, the only way to get a `CoffeeSize` will be with a statement something like this:

```
CoffeeSize cs = CoffeeSize.BIG;
```

It's not required that `enum` constants be in all caps, but borrowing from the Sun code convention that constants are named in caps, it's a good idea.

The basic components of an `enum` are its constants (i.e., BIG, HUGE, and OVERWHELMING), although in a minute you'll see that there can be a lot more to an `enum`. Enums can be declared as their own separate class, or as a class member, however they must not be declared within a method!

Declaring an `enum` *outside* a class:

```
enum CoffeeSize { BIG, HUGE, OVERWHELMING }  // this cannot be
                                             // private or protected

class Coffee {
   CoffeeSize size;
}

public class CoffeeTest1 {
   public static void main(String[] args) {
      Coffee drink = new Coffee();
      drink.size = CoffeeSize.BIG;        // enum outside class
   }
}
```

The preceding code can be part of a single file. (Remember, the file must be named `CoffeeTest1.java` because that's the name of the `public` class in the file.) The key point to remember is that the `enum` can be declared with only the `public` or default modifier, just like a non-inner class. Here's an example of declaring an `enum` *inside* a class:

```
class Coffee2 {
  enum CoffeeSize {BIG, HUGE, OVERWHELMING }

  CoffeeSize size;
}

public class CoffeeTest2 {
  public static void main(String[] args) {
    Coffee2 drink = new Coffee2();
    drink.size = Coffee2.CoffeeSize.BIG;   // enclosing class
                                           // name required
  }
}
```

The key points to take away from these examples are that enums can be declared as their own class, or enclosed in another class, and that the syntax for accessing an enum's members depends on where the enum was declared.

The following is NOT legal:

```
public class CoffeeTest1 {
  public static void main(String[] args) {
    enum CoffeeSize { BIG, HUGE, OVERWHELMING } // WRONG! Cannot
                                                // declare enums
                                                // in methods
    Coffee drink = new Coffee();
    drink.size = CoffeeSize.BIG;
  }
}
```

To make it more confusing for you, the Java language designers made it optional to put a semicolon at the end of the `enum` declaration:

```
public class CoffeeTest1 {

  enum CoffeeSize { BIG, HUGE, OVERWHELMING }; // <--semicolon
                                               // is optional here
  public static void main(String[] args) {
    Coffee drink = new Coffee();
    drink.size = CoffeeSize.BIG;
  }
}
```

So what gets created when you make an `enum`? The most important thing to remember is that enums are not Strings or ints! Each of the enumerated CoffeeSize types are actually instances of CoffeeSize. In other words, BIG is of type CoffeeSize. Think of an `enum` as a kind of class, that looks something (but not exactly) like this:

```
// conceptual example of how you can think
// about enums

class CoffeeSize {
    public static final CoffeeSize BIG =
                            new CoffeeSize("BIG", 0);
    public static final CoffeeSize HUGE =
                            new CoffeeSize("HUGE", 1);
    public static final CoffeeSize OVERWHELMING =
                            new CoffeeSize("OVERWHELMING", 2);
```

```
public CoffeeSize(String enumName, int index) {
        // stuff here
     }
   public static void main(String[] args) {
     System.out.println(CoffeeSize.BIG);
   }
}
```

Notice how each of the enumerated values, BIG, HUGE, and OVERWHELMING, are instances of type CoffeeSize. They're represented as `static` and `final`, which in the Java world, is thought of as a constant. Also notice that each `enum` value knows its index or position…in other words, the order in which `enum` values are declared matters. You can think of the CoffeeSize enums as existing in an array of type CoffeeSize, and as you'll see in a later chapter, you can iterate through the values of an `enum` by invoking the `values()` method on any `enum` type. (Don't worry about that in this chapter.)

Declaring Constructors, Methods, and Variables in an enum

Because an `enum` really is a special kind of class, you can do more than just list the enumerated constant values. You can add constructors, instance variables, methods, and something really strange known as a *constant specific class body*. To understand why you might need more in your `enum`, think about this scenario: imagine you want to know the actual size, in ounces, that map to each of the three CoffeeSize constants. For example, you want to know that BIG is 8 ounces, HUGE is 10 ounces, and OVERWHELMING is a whopping 16 ounces.

You could make some kind of a lookup table, using some other data structure, but that would be a poor design and hard to maintain. The simplest way is to treat your `enum` values (BIG, HUGE, and OVERWHELMING), as objects that can each have their own instance variables. Then you can assign those values at the time the enums are initialized, by passing a value to the `enum` constructor. This takes a little explaining, but first look at the following code:

```
enum CoffeeSize {

     BIG(8), HUGE(10), OVERWHELMING(16);
     // the arguments after the enum value are "passed"
     // as values to the constructor

     CoffeeSize(int ounces) {
```

```
        this.ounces = ounces;  // assign the value to
                               // an instance variable
      }

     private int ounces;       // an instance variable each enum
                               // value has
     public int getOunces() {
       return ounces;
     }
}

class Coffee {
   CoffeeSize size;    // each instance of Coffee has-a
                       // CoffeeSize enum

   public static void main(String[] args) {
      Coffee drink1 = new Coffee();
      drink1.size = CoffeeSize.BIG;

      Coffee drink2 = new Coffee();
      drink2.size = CoffeeSize.OVERWHELMING;

      System.out.println(drink1.size.getOunces()); // prints 8
      System.out.println(drink2.size.getOunces()); // prints 16
   }
}
```

The key points to remember about `enum` constructors are

- You can NEVER invoke an `enum` constructor directly. The `enum` constructor is invoked automatically, with the arguments you define after the constant value. For example, `BIG(8)` invokes the CoffeeSize constructor that takes an `int`, passing the `int` literal 8 to the constructor. (Behind the scenes, of course, you can imagine that `BIG` is also passed to the constructor, but we don't have to know—or care—about the details.)
- You can define more than one argument to the constructor, and you can overload the `enum` constructors, just as you can overload a normal class constructor. We discuss constructors in much more detail in Chapter 2. To initialize a CoffeeType with both the number of ounces and, say, a lid type, you'd pass two arguments to the constructor as `BIG(8, "A")`, which means you have a constructor in CoffeeSize that takes both an `int` and a `String`.

And finally, you can define something really strange in an `enum` that looks like an anonymous inner class (which we talk about in Chapter 8). It's known as a constant specific class body, and you use it when you need a particular constant to override a method defined in the `enum`.

Imagine this scenario: you want enums to have two methods—one for ounces and one for lid code (a String). Now imagine that most coffee sizes use the same lid code, "B", but the OVERWHELMING size uses type "A". You can define a `getLidCode()` method in the CoffeeSize enum that returns "B", but then you need a way to override it for OVERWHELMING. You don't want to do some hard-to-maintain `if/then` code in the `getLidCode()` method, so the best approach might be to somehow have the OVERWHELMING constant override the `getLidCode()` method.

This looks strange, but you need to understand the basic declaration rules:

```
enum CoffeeSize {
     BIG(8),
     HUGE(10),
     OVERWHELMING(16) {    // start a code block that defines
                           // the "body" for this constant

       public String getLidCode() {   // override the method
                                      // defined in CoffeeSize
         return "A";
       }
     };    // <-- the semicolon is REQUIRED when you have a body

     CoffeeSize(int ounces) {
       this.ounces = ounces;
     }

    private int ounces;

    public int getOunces() {
      return ounces;
    }
    public String getLidCode() {  // this method is overridden
                                  // by the OVERWHELMING constant

      return "B";   // the default value we want to return for
                    // CoffeeSize constants
    }
}
```

CERTIFICATION SUMMARY

After absorbing the material in this chapter, you should be familiar with some of the nuances of the Java language. You may also be experiencing confusion around why you ever wanted to take this exam in the first place. That's normal at this point. If you hear yourself saying, "What was I thinking?" just lie down until it passes. We would like to tell you that it gets easier…that this was the toughest chapter and it's all downhill from here…

Let's briefly review what you'll need to know for the exam.

There will be many questions dealing with keywords indirectly, so be sure you can identify which are keywords and which aren't.

Although naming conventions like the use of camelCase won't be on the exam directly, you will need to understand the basics of JavaBeans naming, which uses camelCase.

You need to understand the rules associated with creating legal identifiers, and the rules associated with source code declarations, including the use of `package` and `import` statements.

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`, and `private`) define the access control of a class or member.

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.

We covered interface implementation. Remember 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.

You've also looked at the other modifiers including `static`, `final`, `abstract`, `synchronized`, and so on. You've learned how some modifiers can never be combined in a declaration, such as mixing `abstract` with either `final` or `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.

Remember that as of Java 5, methods can be declared with a var-arg parameter (which can take from zero to many arguments of the declared type), but that you can have only one var-arg per method, and it must be the method's last parameter.

Make sure you're familiar with the relative sizes of the numeric primitives. Remember that while the values of non-final variables can change, a reference variable's type can never change.

You also learned that arrays are objects that contain many variables of the same type. Arrays can also contain other arrays.

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.

Finally, we covered a feature new to Java 5, enums. An `enum` is a much safer and more flexible way to implement constants than was possible in earlier versions of Java. Because they are a special kind of class, enums can be declared very simply, or they can be quite complex—including such attributes as methods, variables, constructors, and a special type of inner class called a constant specific class body.

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.

For the exam, knowing what you can't do with the Java language is just as important as knowing what you can do. Give the sample questions a try! They're very similar to the difficulty and structure of the real exam questions, and should be an eye opener for how difficult the exam can be. Don't worry if you get a lot of them wrong. If you find a topic that you are weak in, spend more time reviewing and studying. Many programmers need two or three serious passes through a chapter (or an individual objective) before they can answer the questions confidently.

TWO-MINUTE DRILL

Remember that in this chapter, when we talk about classes, we're referring to non-inner classes, or *top-level* classes. We'll devote all of Chapter 8 to inner classes.

Identifiers (Objective 1.3)

- ❑ Identifiers can begin with a letter, an underscore, or a currency character.
- ❑ After the first character, identifiers can also include digits.
- ❑ Identifiers can be of any length.
- ❑ JavaBeans methods must be named using camelCase, and depending on the method's purpose, must start with `set`, `get`, `is`, `add`, or `remove`.

Declaration Rules (Objective 1.1)

- ❑ A source code file can have only one `public` class.
- ❑ If the source file contains a `public` class, the filename must match the `public` class name.
- ❑ A file can have only one `package` statement, but multiple `imports`.
- ❑ The `package` statement (if any) must be the first (non-comment) 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 (non-comment) 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.

Class Access Modifiers (Objective 1.1)

- ❑ 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.
- ❑ 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 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

Class Modifiers (Nonaccess) (Objective 1.2)

- 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 of its `abstract` methods.

Interface Implementation (Objective 1.2)

- 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.
- Interface methods 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 the interface's methods.
  - 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 (allowing for covariant returns) 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.
- 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.

Member Access Modifiers (Objectives 1.3 and 1.4)

- 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 other 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 when subclasses are involved:
  - Default members can be accessed only by 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 a superclass instance (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 not accessible to any other class in the subclass package, except for the subclass' own subclasses.

Local Variables (Objective 1.3)

- Local (method, automatic, or 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 (Objective 1.3)

- `final` methods cannot be overridden in a subclass.
- `abstract` methods are declared, with a signature, a return type, and an optional throws clause, but are not implemented.
- `abstract` methods end in a semicolon—no curly braces.
- Three ways to spot a non-abstract 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.
- The `synchronized` modifier applies only to methods and code blocks.
- `synchronized` methods can have any access control and can also be marked `final`.

- `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 `native` modifier applies only to methods.
- The `strictfp` modifier applies only to classes and methods.

Methods with var-args (Objective 1.4)

- As of Java 5, methods can declare a parameter that accepts from zero to many arguments, a so-called var-arg method.
- A var-arg parameter is declared with the syntax `type... name`; for instance: `doStuff(int... x) { }`
- A var-arg method can have only one var-arg parameter.
- In methods with normal parameters and a var-arg, the var-arg must come last.

Variable Declarations (Objective 1.3)

- Instance variables can
  - Have any access control
  - Be marked `final` or `transient`
- Instance variables can't be `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.

Array Declarations (Objective 1.3)

- Arrays can hold primitives or objects, but the array itself is always an object.
- When you declare an array, the brackets can be to the left or right of the variable name.
- It is never legal to include the size of an array in the declaration.
- An array of objects can hold any object that passes the IS-A (or instanceof) test for the declared type of the array. For example, if Horse extends Animal, then a Horse object can go into an Animal array.

Static Variables and Methods (Objective 1.4)

- They are not tied to any particular instance of a class.
- No classes instances are needed in order to use `static` members of the class.
- There is only one copy of a `static` variable / class and all instances share it.
- `static` methods do not have direct access to non-static members.

Enums (Objective 1.3)

- An `enum` specifies a list of constant values that can be assigned to a particular type.
- An `enum` is NOT a String or an int; an enum constant's type is the enum type. For example, WINTER, SPRING, SUMMER, and FALL are of the enum type Season.
- An `enum` can be declared outside or inside a class, but NOT in a method.
- An `enum` declared outside a class must NOT be marked `static`, `final`, `abstract`, `protected`, or `private`.
- Enums can contain constructors, methods, variables, and constant class bodies.
- `enum` constants can send arguments to the `enum` constructor, using the syntax BIG(8), where the int literal 8 is passed to the `enum` constructor.
- `enum` constructors can have arguments, and can be overloaded.
- `enum` constructors can NEVER be invoked directly in code. They are always called automatically when an `enum` is initialized.
- The semicolon at the end of an `enum` declaration is optional. These are legal:

```
enum Foo { ONE, TWO, THREE}
enum Foo { ONE, TWO, THREE};
```

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. Stay focused.

If you have a rough time with these at first, don't beat yourself up. Be positive. Repeat nice affirmations to yourself like, "I am smart enough to understand enums" and "OK, so that other guy knows enums better than I do, but I bet he can't <insert something you *are* good at> like me."

1. Given the following,

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

Which code fragments will compile? (Choose all that apply.)

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 (true); } }
```

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

2. Which declare a compilable `abstract` class? (Choose all that apply.)

A. `public abstract class Canine { public Bark speak(); }`

B. `public abstract class Canine { public Bark speak() { } }`

C. `public class Canine { public abstract Bark speak(); }`

D. `public class Canine abstract { public abstract Bark speak(); }`

3. Which is true? (Choose all that apply.)

A. "X extends Y" is correct if and only if X is a class and Y is an interface.

B. "X extends Y" is correct if and only if X is an interface and Y is a class.

C. "X extends Y" is correct if X and Y are either both classes or both interfaces.

D. "X extends Y" is correct for all combinations of X and Y being classes and/or interfaces.

4. Which are valid declarations? (Choose all that apply.)

A. `int $x;`
B. `int 123;`
C. `int _123;`
D. `int #dim;`
E. `int %percent;`
F. `int *divide;`
G. `int central_sales_region_Summer_2005_gross_sales;`

5. Which method names follow the JavaBeans standard? (Choose all that apply.)

A. `addSize`
B. `getCust`
C. `deleteRep`
D. `isColorado`
E. `putDimensions`

6. Given:

```
1. class Voop {
2.   public static void main(String [] args) {
3.     doStuff(1);
4.     doStuff(1,2);
5.   }
6.   // insert code here
7. }
```

Which, inserted independently at line 6, will compile? (Choose all that apply.)

A. `static void doStuff(int... doArgs) { }`
B. `static void doStuff(int[] doArgs) { }`
C. `static void doStuff(int doArgs...) { }`
D. `static void doStuff(int... doArgs, int y) { }`
E. `static void doStuff(int x, int... doArgs) { }`

7. Which are legal declarations? (Choose all that apply.)

A. `short x [];`
B. `short [] y;`
C. `short[5] x2;`
D. `short z2 [5];`
E. `short [] z [] [];`
F. `short [] y2 = [5];`

8. Given:

```
1. enum Animals {
2.   DOG("woof"), CAT("meow"), FISH("burble");
3.   String sound;
4.   Animals(String s) { sound = s; }
5. }
6. class TestEnum {
7.   static Animals a;
8.   public static void main(String[] args) {
9.     System.out.println(a.DOG.sound + " " + a.FISH.sound);
10.  }
11. }
```

What is the result?

A. `woof burble`

B. Multiple compilation errors

C. Compilation fails due to an error on line 2

D. Compilation fails due to an error on line 3

E. Compilation fails due to an error on line 4

F. Compilation fails due to an error on line 9

9. Given:

```
1. enum A { A }
2. class E2 {
3.   enum B { B }
4.   void C() {
5.     enum D { D }
6.   }
7. }
```

Which statements are true? (Choose all that apply.)

A. The code compiles.

B. If only line 1 is removed the code compiles.

C. If only line 3 is removed the code compiles.

D. If only line 5 is removed the code compiles.

E. If lines 1 and 3 are removed the code compiles

F. If lines 1, 3 and 5 are removed the code compiles.

SELF TEST ANSWERS

1. Given the following,

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

Which code fragments will compile? (Choose all that apply.)

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 (true); } }
```

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

Answer:

☑ **C** and **D** are correct. **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.

☒ **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`. (Objective 1.1)

2. Which declare a compilable `abstract` class? (Choose all that apply.)

A. `public abstract class Canine { public Bark speak(); }`

B. `public abstract class Canine { public Bark speak() { } }`

C. `public class Canine { public abstract Bark speak(); }`

D. `public class Canine abstract { public abstract Bark speak(); }`

Answer:

☑ **B** is correct. `abstract` classes don't have to have any `abstract` methods.

☒ **A** is incorrect because `abstract` methods must be marked as such. **C** is incorrect because you can't have an `abstract` method unless the class is `abstract`. **D** is incorrect because the keyword `abstract` must come before the classname. (Objective 1.1)

3. Which is true? (Choose all that apply.)

A. "X extends Y" is correct if and only if X is a class and Y is an interface.

B. "X extends Y" is correct if and only if X is an interface and Y is a class.

C. "X extends Y" is correct if X and Y are either both classes or both interfaces.

D. "X extends Y" is correct for all combinations of X and Y being classes and/or interfaces.

Answer:

☑ **C** is correct.

☒ **A** is incorrect because classes implement interfaces, they don't extend them.
B is incorrect because interfaces only "inherit from" other interfaces.
D is incorrect based on the preceding rules. (Objective 1.2)

4. Which are valid declarations? (Choose all that apply.)

A. `int $x;`

B. `int 123;`

C. `int _123;`

D. `int #dim;`

E. `int %percent;`

F. `int *divide;`

G. `int central_sales_region_Summer_2005_gross_sales;`

Answer:

☑ **A, C,** and **G** are legal identifiers.

☒ **B** is incorrect because an identifier can't start with a digit.
D, E, and **F** are incorrect because identifiers must start with $, _, or a letter. (Objective 1.3)

5. Which method names follow the JavaBeans standard? (Choose all that apply.)

A. `addSize`

B. `getCust`

C. `deleteRep`

D. `isColorado`

E. `putDimensions`

Answer:

☑ **B** and **D** use the valid prefixes '`get`' and '`is`'.

☒ **A, C,** and **E** are incorrect because '`add`', '`delete`' and '`put`' are not standard JavaBeans name prefixes. (Objective 1.4)

6. Given:

```
1. class Voop {
2.   public static void main(String[] args) {
3.     doStuff(1);
4.     doStuff(1,2);
5.   }
6.   // insert code here
7. }
```

Which, inserted independently at line 6, will compile? (Choose all that apply.)

A. `static void doStuff(int... doArgs) { }`

B. `static void doStuff(int[] doArgs) { }`

C. `static void doStuff(int doArgs...) { }`

D. `static void doStuff(int... doArgs, int y) { }`

E. `static void doStuff(int x, int... doArgs) { }`

Answer:

☑ **A** and E use valid var-args syntax.

☒ **B** and **C** are invalid var-arg syntax, and **D** is invalid because the var-arg must be the last of a method's arguments. (Objective 1.4)

7. Which are legal declarations? (Choose all that apply.)

A. `short x [];`

B. `short [] y;`

C. `short[5] x2;`

D. `short z2 [5];`

E. `short [] z [] [];`

F. `short [] y2 = [5];`

Answer:

☑ **A, B,** and **E** are correct array declarations; E is a three dimensional array.

☒ **C, D,** and **F** are incorrect, you can't include the size of your array in a declaration unless you also instantiate the array object. F uses invalid instantiation syntax. (Objective 1.3)

8. Given:

```
1. enum Animals {
2.   DOG("woof"), CAT("meow"), FISH("burble");
3.   String sound;
4.   Animals(String s) { sound = s; }
5. }
6. class TestEnum {
```

```
 7.   static Animals a;
 8.   public static void main(String [] args) {
 9.     System.out.println(a.DOG.sound + " " + a.FISH.sound);
10.   }
11. }
```

What is the result?

A. `woof burble`

B. Multiple compilation errors

C. Compilation fails due to an error on line 2

D. Compilation fails due to an error on line 3

E. Compilation fails due to an error on line 4

F. Compilation fails due to an error on line 9

Answer:

☑ **A** is correct; enums can have constructors and variables.

☒ **B, C, D, E,** and **F** are incorrect; these lines all use correct syntax.

9. Given:

```
1. enum A { A }
2. class E2 {
3.   enum B { B }
4.   void C() {
5.     enum D { D }
6.   }
7. }
```

Which statements are true? (Choose all that apply.)

A. The code compiles.

B. If only line 1 is removed the code compiles.

C. If only line 3 is removed the code compiles.

D. If only line 5 is removed the code compiles.

E. If lines 1 and 3 are removed the code compiles.

F. If lines 1, 3 and 5 are removed the code compiles.

Answer:

☑ **D** and **F** are correct. Line 5 is the only line that will not compile, because enums cannot be local to a method.

☒ **A, B, C** and **E** are incorrect based on the above.