OCA Java SE 8 Programmer I Exam Guide (Exams 1Z0-808)

by Kathy Sierra and Bert Bates

Chapter 1: Declarations and Access Control

CERTIFICATION OBJECTIVES

- Java Features and Benefits
- Identifiers and Keywords
- javac, java, main(), and Imports
- Declare Classes and Interfaces
- Declare Class Members
- Declare Constructors and Arrays
- Create static Class Members
- Use enums
- Two-Minute Drill
- 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 a while.

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. Following is a list of a few useful terms for this object-oriented (OO) language:

- **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 that 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's logic is stored and 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 (and Oracle) programmers 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 OO languages is the concept of *inheritance*, which allows code defined in one class or interface 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 the subclass is also free to *override* superclass methods to define more specific behavior. For example, a `Car` *super*class could define general methods common to all automobiles, but a `Ferrari` *sub*class could override the `accelerate()` method that was already defined in the `Car` class.

Interfaces

A powerful companion to inheritance is the use of interfaces. Interfaces are *usually* like a 100 percent abstract superclass that defines the methods a subclass must support, but not *how* they must be supported. In other words, for example, 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. Note: As of Java 8, interfaces can now include concrete, inheritable methods. We will talk much more about this when we dive into OO in the next chapter.

CERTIFICATION OBJECTIVE: FEATURES AND BENEFITS OF JAVA (OCA OBJECTIVE 1.5)

1.5 Compare and contrast the features and components of Java such as: platform independence, object orientation, encapsulation, etc.

Perhaps a great topic to start with, on our official coverage of the OCA 8, is to discuss the various benefits that Java provides to programmers. Java is now over 20 years old (wow!) and remains one of the most in-demand programming languages in the world. Somewhat confusingly there is a similarly named language, "JavaScript" (an implementation of the ECMA standard), which is also a very popular language. Java and JavaScript have some aspects in common, but they are not to be confused; they are quite distinct. Let's look at some of the benefits that Java provides to programmers and compare them (when appropriate) to some of Java's competitors. A caveat here, many of these benefits are based on extremely complex topics. These descriptions are by no means definitive, but they're sufficient for the exam:

- **Object oriented** As software systems get larger, they get more difficult to test and enhance. For the last several decades, object-oriented (OO) programming has been the dominant software design approach for large systems, because well-designed OO systems remain testable and enhanceable, even as they grow into huge applications with millions of lines of code. OO design also offers a natural way to think about how the components in a system should be constructed and how they should interact. The classes, objects, system state, and behaviors in well-designed OO systems are easy to map conceptually to their counterparts in the real world.
- **Encapsulation** Encapsulation is a key concept in OO programming. Encapsulation allows a software component to hide its data from other components, protecting the data from being updated without the component's approval or knowledge. Java makes encapsulation far easier to achieve than in non-OO languages.
- **Memory management** Unlike some of its competitors (C and C++), Java provides automatic memory management. In languages that don't provide automatic memory management, keeping track of memory through pointers is quite complex. Further, tracking down bugs related to memory management (often called *memory leaks*) is a common, error-prone, and time-consuming process.
- **Huge library** Java has an enormous library of prewritten, well-tested, and well-supported code. This code is easy to include in your Java applications and is well documented via the Java API. Throughout this book we will explore some of the most used (and most useful) members of Java's standard core library.
- **Secure by design** When compiled Java code is executed, it runs inside the Java Virtual Machine (JVM). The JVM provides a secure "sandbox" for your Java code to run in, and the JVM makes sure that nefarious programmers cannot write Java code that will cause trouble on other people's machines when it runs.
- **Write once, run anywhere (cross-platform execution)** One of the goals (largely, but not perfectly achieved) of Java is that much of the Java code you write can run on many platforms, ranging from tiny Internet-of-Things (IoT) devices, to phones, to laptop computers, to large servers. Another common phrase for this ability to run on many devices is *cross-platform*.
- **Strongly typed** A strongly typed language usually requires the programmer to explicitly declare the types of the data and objects being used in the program. Strong typing allows the Java compiler to catch many potential programming errors before your code even compiles. At the other end of the spectrum are dynamically typed languages. Dynamically typed languages can be less verbose, faster to code initially, and are often preferred in environments where small teams and rapid prototyping are the norm. But strongly typed languages like Java come into their own in large software shops with many teams of programmers and the need for more reliable, testable, production-quality code.
- **Multithreaded** Java provides built-in language features and APIs that allow programs to use many operating-system processes (hence, many "cores") at the same time. As systems grow to handle more computationally intensive problems and larger data sets, the ability to use all of a computer's core processors becomes essential. Multithreaded programming is never simple, but Java provides a rich toolkit to make it as easy as possible.
- **Distributed computing** Another way to tackle big programming problems is to distribute the workload across many machines. The Java API provides several ways to simplify tasks related to distributed computing. One such example is *serialization*, a process in which a Java object is converted to a portable form. Serialized objects can be sent to other machines, deserialized, and then used as a normal Java object.

Again, we've just scratched the surface of these complex topics, but if you understand these brief descriptions, you should be prepared to handle any questions for this objective. So much for the theory, let's get into details…

CERTIFICATION OBJECTIVE: IDENTIFIERS AND KEYWORDS (OCA OBJECTIVES 1.2 AND 2.1)

1.2 Define the structure of a Java class.

2.1 Declare and initialize variables (including casting of primitive data types).

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 and identifiers, but *using* the things you declare is covered primarily in later chapters.

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

- **Legal identifiers** The rules the compiler uses to determine whether a name is legal.
- **Oracle's Java Code Conventions** Oracle'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 you use them.

Legal Identifiers

Technically, legal identifiers must be composed of only Unicode characters, numbers, currency symbols, and connecting characters (such as underscores). The exam doesn't dive into the details of which ranges of the Unicode character set 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 digit!
- 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 the Java keywords.
- 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				

Oracle's Java Code Conventions

Oracle 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. Oracle 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 if you start at java.oracle.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 Oracle certifications is that the exams are administered uniformly throughout the world. To achieve that, the code listings that you'll see in the real exam are often quite cramped and do not follow Oracle's code standards. 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 Oracle standard of four.

We'll also jam our curly braces together unnaturally, and we'll 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 Oracle, 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 only standards that *are* followed as much as possible in the real exam are the naming standards. Here are the naming standards that Oracle 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. Here are some examples:

  ```
  Dog
  Account
  PrintWriter
  ```

  For interfaces, the names should typically be adjectives, like these:

  ```
  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, but starting with a lowercase letter. Oracle 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
  ```

CERTIFICATION OBJECTIVE: DEFINE CLASSES (OCA OBJECTIVES 1.2, 1.3, 1.4, 6.4, AND 7.5)

1.2 Define the structure of a Java class.

1.3 Create executable Java applications with a main method; run a Java program from the command line; including console output. (sic)

1.4 Import other Java packages to make them accessible in your code.

6.4 Apply access modifiers.

7.5 Use abstract classes and interfaces.

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

Using the javac and java Commands

In this book, we're going to talk about invoking the `javac` and `java` commands about 1000 times. Although in the **real world** you'll probably use an integrated development environment (IDE) most of the time, you could see a few questions on the exam that use the command line instead, so we're going to review the basics. (By the way, we did NOT use an IDE while writing this book. We still have a slight preference for the command line while studying for the exam; all IDEs do their best to be "helpful," and sometimes they'll fix your problems without telling you. That's nice on the job, but maybe not so great when you're studying for a certification exam!)

Compiling with javac

The `javac` command is used to invoke Java's compiler. You can specify many options when running `javac`. For example, there are options to generate debugging information or compiler warnings. Here's the structural overview for `javac`:

```
javac [options] [source files]
```

There are additional command-line options called `@argfiles`, but they're rarely used, and you won't need to study them for the exam. Both the `[options]` and the `[source files]` are optional parts of the command, and both allow multiple entries. The following are legal `javac` commands:

```
javac -help
javac -version Foo.java Bar.java
```

The first invocation doesn't compile any files, but prints a summary of valid options. The second invocation passes the compiler an option (`-version`, which prints the version of the compiler you're using) and passes the compiler two `.java` files to compile (`Foo.java` and `Bar.java`). Whenever you specify multiple options and/or files, they should be separated by spaces.

Launching Applications with java

The `java` command is used to invoke the Java Virtual Machine (JVM). Here's the basic structure of the command:

```
java [options] class [args]
```

The `[options]` and `[args]` parts of the `java` command are optional, and they can both have multiple values. You must specify exactly one class file to execute, and the `java` command assumes you're talking about a `.class` file, so you don't specify the `.class` extension on the command line. Here's an example:

```
java -showversion MyClass x 1
```

This command can be interpreted as "Show me the version of the JVM being used, and then launch the file named `MyClass.class` and send it two String *arguments* whose values are `x` and `1`." Let's look at the following code:

```
public class MyClass {
  public static void main(String[] args) {
    System.out.println(args[0] + " " + args[1]);
  }
}
```

It's compiled and then invoked as follows:

```
java MyClass x 1
```

The output will be

```
x 1
```

We'll be getting into arrays in depth later, but for now it's enough to know that args—like all arrays—use a zero-based index. In other words, the first command-line argument is assigned to `args[0]`, the second argument is assigned to `args[1]`, and so on.

Using public static void main(String[ ] args)

The use of the `main()` method is implied in most of the questions on the exam, and on the OCA exam it is specifically covered. For the .0001 percent of you who don't know, `main()` is the method that the JVM uses to start execution of a Java program.

First off, it's important for you to know that naming a method `main()` doesn't give it the superpowers we normally associate with `main()`. As far as the compiler and the JVM are concerned, the **only** version of `main()` with superpowers is the `main()` with this signature:

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

Other versions of `main()` with other signatures are perfectly legal, but they're treated as normal methods. There is some flexibility in the declaration of the "special" `main()` method (the one used to start a Java application): the order of its modifiers can be altered a little; the `String` array doesn't have to be named `args`; and it can be declared using var-args syntax. The following are all legal declarations for the "special" `main()`:

```
static public void main(String[] args)
public static void main(String... x)
static public void main(String bang_a_gong[])
```

For the OCA 8 exam, the only other thing that's important for you to know is that `main()` **can be overloaded**. We'll cover overloading in detail in the next chapter.

Import Statements and the Java API

There are a gazillion Java classes in the world. The Java API has thousands of classes, and the Java community has written the rest. We'll go out on a limb and contend that all Java programmers everywhere use a combination of classes they wrote and classes that other programmers wrote. Suppose we created the following:

```
public class ArrayList {
  public static void main(String[] args) {
    System.out.println("fake ArrayList class");
 }
}
```

This is a perfectly legal class, but as it turns out, one of the most commonly used classes in the Java API is also named `ArrayList`, or so it seems…. The API version's actual name is `java.util.ArrayList`. That's its *fully qualified name*. The use of fully qualified names is what helps Java developers make sure that two versions of a class like `ArrayList` don't get confused. So now let's say that I want to use the `ArrayList` class from the API:

```
public class MyClass {
  public static void main(String[] args) {
    java.util.ArrayList<String> a =
      new java.util.ArrayList<String>();
  }
}
```

(First off, trust us on the `<String>` syntax; we'll get to that later.) Although this is legal, it's also a LOT of keystrokes. Since we programmers are basically lazy (there, we said it), we like to use other people's classes a LOT, AND we hate to type. If we had a large program, we might end up using `ArrayLists` many times.

`import` statements to the rescue! Instead of the preceding code, our class could look like this:

```
import java.util.ArrayList;
public class MyClass {
  public static void main(String[] args) {
    ArrayList<String> a = new ArrayList<String>();
}}
```

We can interpret the `import` statement as saying, "In the Java API there is a package called `'util'`, and in that package is a class called `'ArrayList'`. Whenever you see the word `'ArrayList'` in this class, it's just shorthand for: `'java.util.ArrayList'`." (Note: Lots more on packages to come!) If you're a C programmer, you might think that the `import` statement is similar to an `#include`. Not really. All a Java `import` statement does is save you some typing. That's it.

As we just implied, a package typically has many classes. The `import` statement offers yet another keystroke-saving capability. Let's say you wanted to use a few different classes from the `java.util` package: `ArrayList` and `TreeSet`. You can add a wildcard character (`*`) to your `import` statement that means, "If you see a reference to a class you're not sure of, you can look through the entire package for that class," like so:

```
import java.util.*;
public class MyClass {
  public static void main(String[] args) {
    ArrayList<String> a = new ArrayList<String>();
    TreeSet<String> t = new TreeSet<String>();
  }
}
```

When the compiler and the JVM see this code, they'll know to look through `java.util` for `ArrayList` and `TreeSet`. For the exam, the last thing you'll need to remember about using `import` statements in your classes is that you're free to mix and match. It's okay to say this:

```
ArrayList<String> a = new ArrayList<String>();
java.util.ArrayList<String> a2 = new java.util.ArrayList<String>();
```

Static Import Statements

Dear Reader, We really struggled with where to include this discussion of static imports. From a learning perspective, this is probably not the ideal location, but from a reference perspective, we thought it made sense. As you're learning the material for the first time, you might be confused by some of the ideas in this section. If that's the case, we apologize. Put a sticky note on this page and circle back around after you're finished with Chapter 3. On the other hand, once you're past the learning stage and you're using this book as a reference, we think putting this section here will be quite useful. Now, on to static imports.

Sometimes classes will contain *static members*. (We'll talk more about static class members later, but since we're on the topic of imports we thought we'd toss in static imports now.) Static class members can exist in the classes you write and in a lot of the classes in the Java API.

As we said earlier, ultimately the only value `import` statements have is that they save typing and they can make your code easier to read. In Java 5 (a long time ago), the `import` statement was enhanced to provide even greater keystroke-reduction capabilities, although some would argue that this comes at the expense of readability. This feature is known as *static imports*. Static imports can be used when you want to "save typing" while using a class's static members. (You can use this feature on classes in the API and on your own classes.) Here's a "before and after" example using a few static class members provided by a commonly used class in the Java API, `java.lang.Integer`. This example also uses a static member that you've used a thousand times, probably without ever giving it much thought; the `out` field in the `System` class.

Before static imports:

```
public class TestStatic {
  public static void main(String[] args) {
    System.out.println(Integer.MAX_VALUE);
    System.out.println(Integer.toHexString(42));
  }
}
```

After static imports:

```
import static java.lang.System.out;             // 1
import static java.lang.Integer.*;              // 2
public class TestStaticImport {
  public static void main(String[] args) {
    out.println(MAX_VALUE);                     // 3
    out.println(toHexString(42));               // 4
  }
}
```

Both classes produce the same output:

```
2147483647
2a
```

Let's look at what's happening in the code that's using the static import feature:

1. Even though the feature is commonly called "static import," the syntax MUST be `import static` followed by the fully qualified name of the `static` member you want to import, or a wildcard. In this case, we're doing a static import on the `System` class `out` object.
2. In this case, we might want to use several of the `static` members of the `java.lang.Integer` class. This static import statement uses the wildcard to say, "I want to do static imports of ALL the `static` members in this class."
3. Now we're finally seeing the *benefit* of the static import feature! We didn't have to type the `System` in `System.out.println`! Wow! Second, we didn't have to type the `Integer` in `Integer.MAX_VALUE`. So in this line of code we were able to use a shortcut for a `static` method AND a constant.

4. Finally, we do one more shortcut, this time for a method in the `Integer` class.

We've been a little sarcastic about this feature, but we're not the only ones. We're not convinced that saving a few keystrokes is worth possibly making the code a little harder to read, but enough developers requested it that it was added to the language.

Here are a couple of rules for using static imports:

- You must say `import static`; you can't say `static import`.
- Watch out for ambiguously named `static` members. For instance, if you do a static import for both the `Integer` class and the `Long` class, referring to `MAX_VALUE` will cause a compiler error, because both `Integer` and `Long` have a `MAX_VALUE` constant and Java won't know which `MAX_VALUE` you're referring to.
- You can do a static import on `static` object references, constants (remember they're `static` and `final`), and `static` methods.

Exam Watch

As you've seen, when using `import` and `import static` statements, sometimes you can use the wildcard character * to do some simple searching for you. (You can search within a package or within a class.) As you saw earlier, if you want to "search through the `java.util` package for class names," you can say this:

```
import java.util.*;                  // ok to search the java.util package
```

In a similar vein, if you want to "search through the `java.lang.Integer` class for static members," you can say this:

```
import static java.lang.Integer.*;   // ok to search the
                                     // java.lang.Integer class
```

But you can't create broader searches. For instance, you CANNOT use an `import` to search through the entire Java API:

```
import java.*;    // Legal, but this WILL NOT search across packages.
```

Class Declarations and Modifiers

The class declarations we'll discuss in this section are limited to top-level classes. In addition to top-level classes, Java provides for another category of class known as *nested classes* or *inner classes*. Inner classes are included on the OCP exam, but not the OCA exam. When you become an OCP candidate, you're going to love learning about inner classes. 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. In general, modifiers fall into two categories:

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

We'll look at access modifiers first, so you'll learn how to restrict or allow access to a class you create. Access control in Java is a little tricky because there are four access *controls* (levels of access) but only three access *modifiers*. The fourth access control level (called *default* or *package* access) is what you get when you don't use any of the three access modifiers. In other words, *every* class, method, and instance variable you declare has an access *control*, whether you explicitly type one or not. Although all four access *controls* (which means all three *modifiers*) work for most method and variable declarations, a class can be declared with only `public` or *default* access; the other two access control levels don't make sense for a class, as you'll see.

on the job Java is a package-centric language; the developers assumed that for good organization and name scoping, you would put all your classes into packages. They were right, and you should. Imagine this nightmare: Three different programmers, in the same company but working on different parts of a project, write a class named `Utilities`. If those three `Utilities` classes have not been declared in any explicit package and are in the classpath, you won't have any way to tell the compiler or JVM which of the three you're trying to reference. Oracle 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 Oracle'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* this:

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

(Note: For various reasons, the error messages we show throughout this book might not match the error messages you get. Don't worry, the real point is to understand when you're apt to get an error of some sort.)

`Tea` won't compile because its superclass, `Beverage`, has default access and is in a different package. You can do one of two things to make this work. You could put both classes in the same package, or 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 strictly 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 they 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 result in a compiler error.

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

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

There's a benefit to having nonfinal classes in this scenario: Imagine that 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, 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 let's try to compile the `Tea` subclass:

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

We get an error—something like this:

```
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. Unless you have a serious safety or security issue, assume that someday 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 that 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: USE INTERFACES (OCA OBJECTIVE 7.5)

7.6 Use abstract classes and interfaces.

Declaring an Interface

In general, 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.

Note: As of Java 8, you can now also describe the *how*, but you usually won't. Until we get to the new interface-related features of Java 8—`default` and `static` methods—we will discuss interfaces from a traditional perspective, which is again, defining a contract for *what* a class can do.

An interface is a contract. You could write an interface `Bounceable`, for example, that says in effect, "This is the `Bounceable` interface. Any concrete 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.

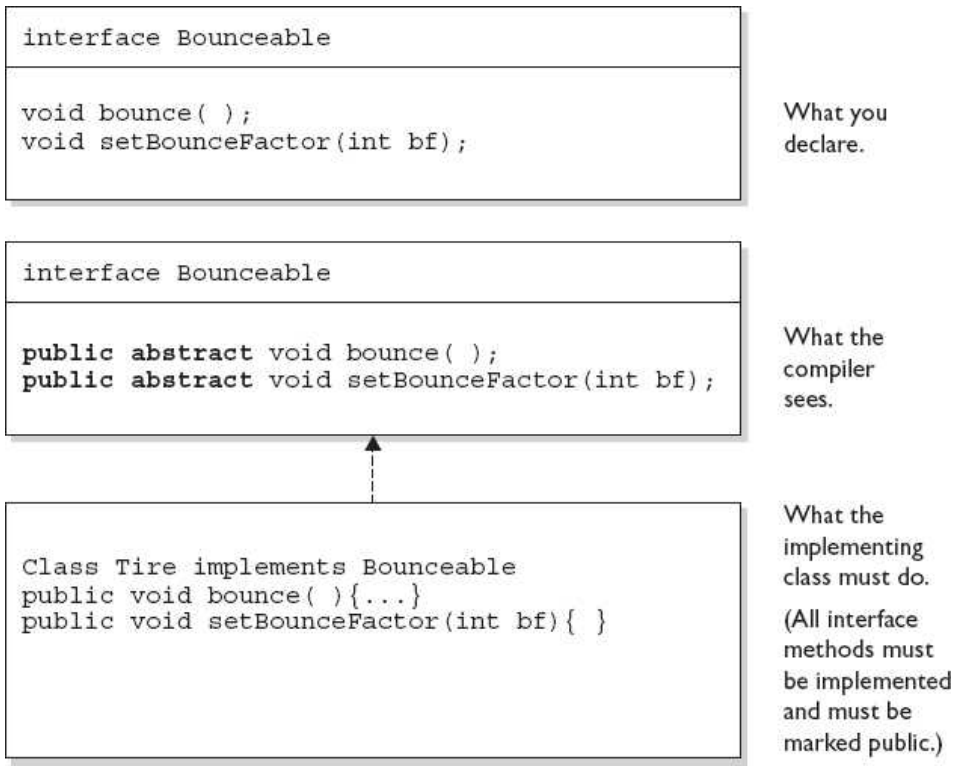

Figure 1-1: The relationship between interfaces and classes

Think of a traditional 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 although an `abstract` class can define both `abstract` and nonabstract methods, an interface *generally* has 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:

- Interface methods are implicitly `public` and `abstract`, unless declared as `default` or `static`. 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 cannot be marked `final`, `strictfp`, or `native`. (More on these modifiers later in the chapter.)
- 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 not declared `default` or `static` 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
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 actually have to 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.

Declaring `default` Interface Methods

As of Java 8, interfaces can include inheritable* methods with concrete implementations. (*The strict definition of "inheritance" has gotten a little fuzzy with Java 8; we'll talk more about inheritance in Chapter 2.) These concrete methods are called `default` methods. In the next chapter we'll talk a lot about the various OO-related rules that are impacted because of `default` methods. For now we'll just cover the simple declaration rules:

- `default` methods are declared by using the `default` keyword. The `default` keyword can be used only with interface method signatures, not class method signatures.
- `default` methods are `public` by definition, and the `public` modifier is optional.
- `default` methods **cannot** be marked as `private`, `protected`, `static`, `final`, or `abstract`.
- `default` methods must have a concrete method body.

Here are some examples of legal and illegal `default` methods:

```
interface TestDefault {
  default int m1(){return 1;} // legal
  public default void m2(){;} // legal
  static default void m3(){;} // illegal: default cannot be marked static
  default void m4();          // illegal: default must have a method body
}
```

Declaring `static` Interface Methods

As of Java 8, interfaces can include `static` methods with concrete implementations. As with interface `default` methods, there are OO implications that we'll discuss in Chapter 2. For now, we'll focus on the basics of declaring and using `static` interface methods:

- `static` interface methods are declared by using the `static` keyword.
- `static` interface methods are public by default, and the `public` modifier is optional.
- `static` interface methods cannot be marked as `private`, `protected`, `final`, or `abstract`.
- `static` interface methods must have a concrete method body.
- When invoking a `static` interface method, the method's type (interface name) MUST be included in the invocation.

Here are some examples of legal and illegal static interface methods and their use:

```
interface StaticIface {
  static int m1(){ return 42; }     // legal
  public static void m2(){ ; }      // legal
  // final static void m3(){ ; }    // illegal: final not allowed
  // abstract static void m4(){ ; } // illegal: abstract not allowed
  // static void m5();              // illegal: needs a method body
}

public class TestSIF implements StaticIface {
  public static void main(String[] args) {
    System.out.println(StaticIface.m1());   // legal: m1()'s type
                                            // must be included
    new TestSIF().go();
    // System.out.println(m1());     // illegal: reference to interface
                                     // is  required
  }
  void go() {
    System.out.println(StaticIface.m1()); // also legal from an instance
  }
}
```

which produces this output:

```
42
42
```

As we said earlier, we'll return to our discussion of `default` methods and `static` methods for interfaces in Chapter 2.

CERTIFICATION OBJECTIVE: DECLARE CLASS MEMBERS (OCA OBJECTIVES 2.1, 2.2, 2.3, 4.1, 4.2, 6.2, 6.3, AND 6.4)

2.1 Declare and initialize variables (including casting of primitive data types).

2.2 Differentiate between object reference variables and primitive variables.

2.3 Know how to read or write to object fields.

4.1 Declare, instantiate, initialize, and use a one-dimensional array.

4.2 Declare, instantiate, initialize, and use multidimensional array. (sic)

6.2 Apply the static keyword to methods and fields.

6.3 Create and overload constructors; including impact on default constructors. (sic)

6.4 Apply access modifiers.

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

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

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 we will mention later.

Note: As of Java 8, the word `default` can ALSO be used to declare certain methods in interfaces. When used in an interface's method declaration, `default` has a different meaning than what we are describing for the rest of this chapter.

It's crucial that you know access control inside and outside for the exam. There will be quite a few questions where access control plays a role. Some questions test several concepts of access control at the same time, so not knowing one small part of access control could mean you 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 occurs 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. Here's an example:

```
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.

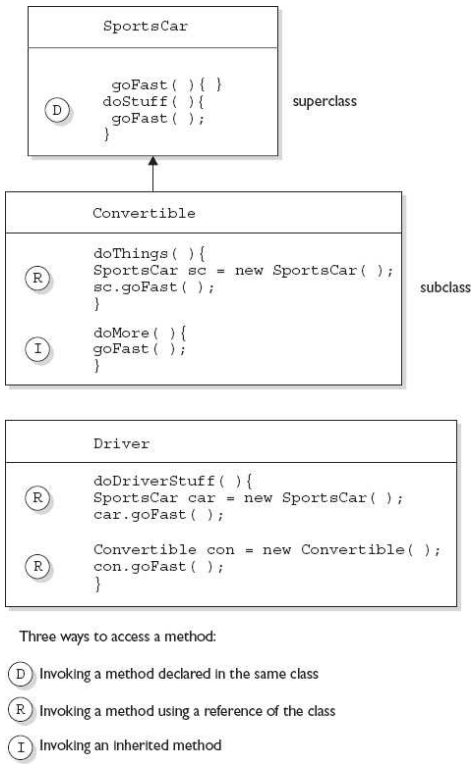

Figure 1-2: Comparison of inheritance vs. dot operator for member access

Much of access control (both types) centers on whether the two classes involved are in the same or different packages. Don't forget, though, that 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 visible either, even if the member is declared `public`. Once you've confirmed that the class is visible, then it makes sense to look at access levels on individual members.

Public Members

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

Look at the following source file:

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

Now look at the second file:

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

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

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

```
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.

Notice in the following code that the `doRooThings()` method is invoked without having to preface it with a reference:

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

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:

```
package notcert;
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, this is 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 do anything else you want it to do.

```
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
```

Can a `private` method be overridden by a subclass? That's an interesting question, but the answer is no. Because 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.

Protected and Default Members

Note: Just a reminder, in the next several sections, when we use the word "default," we're talking about access control. We're NOT talking about the new kind of Java 8 interface method that can be declared `default`.

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");
  }
}
```

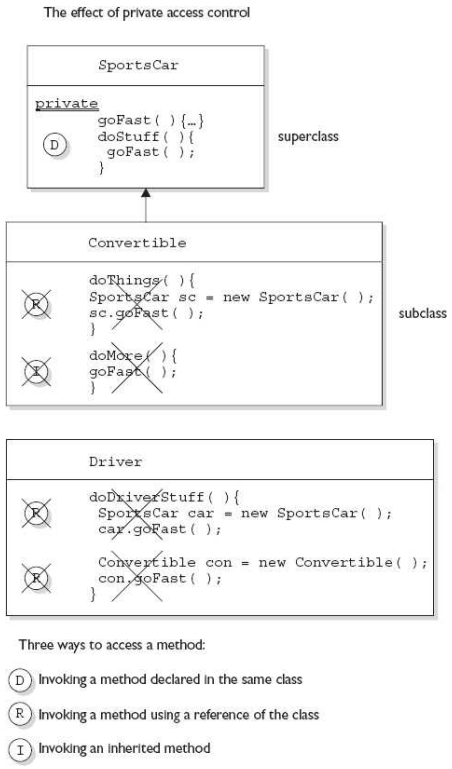

Figure 1-3: Effects of `public` and `private` access

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's 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? Hang in there and it will all become clearer with the next batch of code examples.

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 that 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's package) tries to access a `protected` variable using a `Parent` class reference:

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

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

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

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

But there's still one more issue we haven't looked at: What does a `protected` member look like to other classes trying to use the subclass-outside-the-package to get to the subclass's 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`. 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
}
```

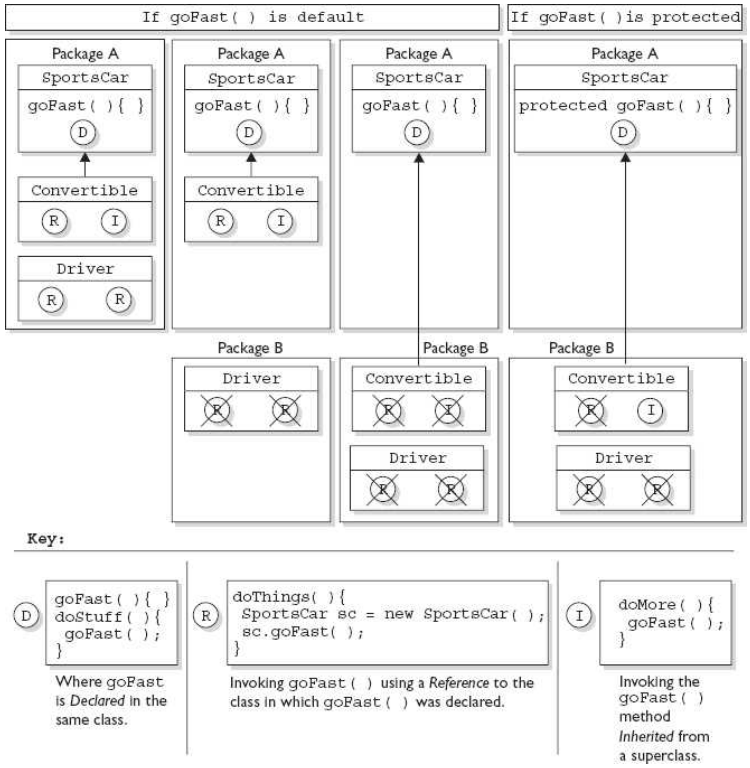

Figure 1-4: Effects of `protected` access

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

When we try to compile the `Child.java` file, we get an error 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 nonsubclass 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`, much later in the chapter.

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 recNumber) {}
```

Method arguments are essentially the same as local variables. In the preceding example, the variables `fileNumber` and `recNumber` 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 `recNumber` 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` parameter must keep the same value as the argument 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 **might** provide actual implementation code inside the curly braces.

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.

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) { }
}
```

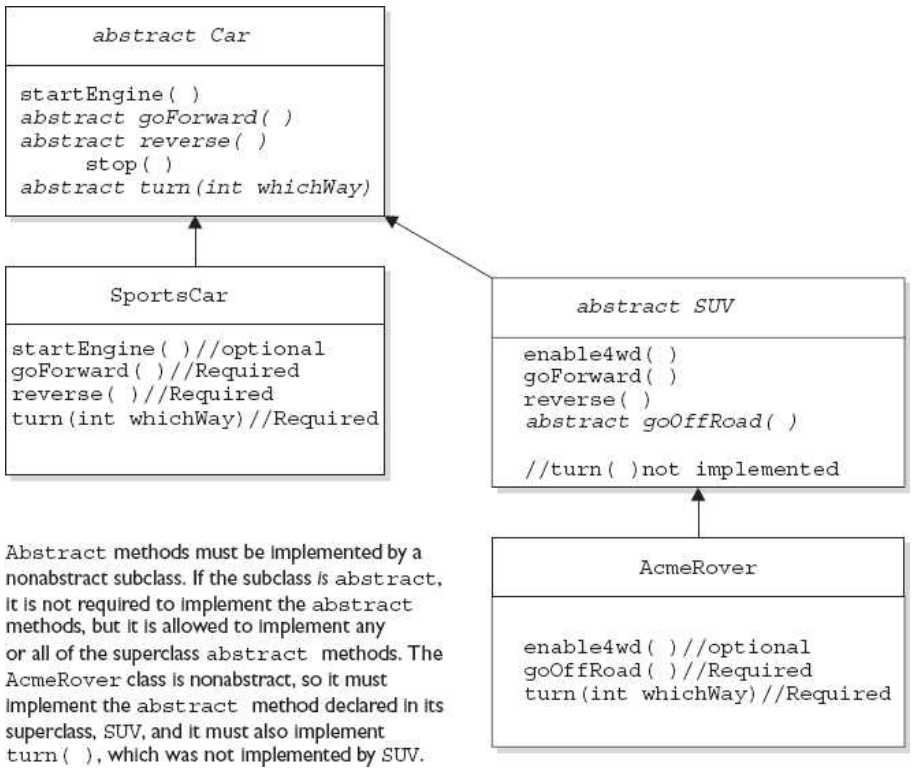

Figure 1-5: The effects of the `abstract` modifier on concrete and `abstract` subclasses

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's `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's `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—for top-level classes—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. When you are studying for your OCP 8, you'll study the `synchronized` keyword extensively, 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)

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 parameters." 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
```

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-arg 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), and 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`, and `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.

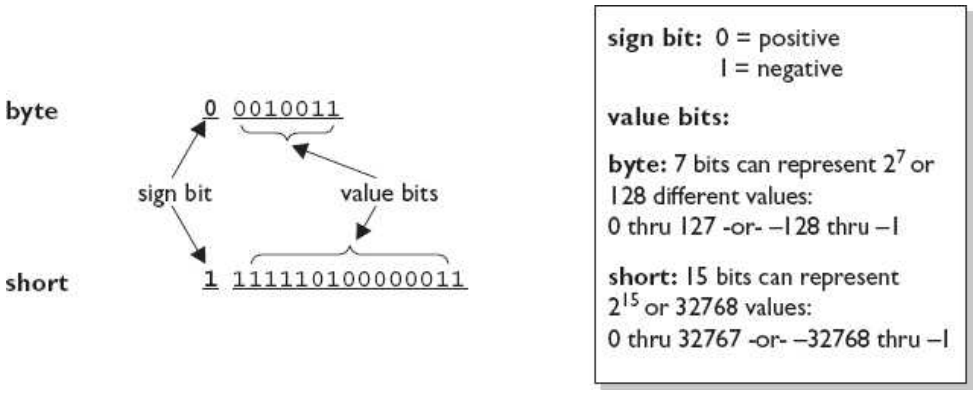

Figure 1-6: The sign bit for a byte

Table 1-3: 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

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 0 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.

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).

There is not a range of `boolean` values; 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 initialized only 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. 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 versus variables.

Local Variables	Variables (nonlocal)	Methods
final	final public protected private static transient volatile	final public protected private static abstract synchronized strictfp native

Figure 1-7: Comparison of modifiers on variables vs. methods

Local (Automatic/Stack/Method) Variables

A local variable is a variable 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 reassign 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 an 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 5.

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 are Collection types 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? Java provides a wide variety of Collection types to address these situations, but the only Collection type on the exam is `ArrayList`, and Chapter 5 discusses `ArrayList` 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, 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 6, we'll spend a lot of time discussing arrays, how to initialize and use them and how to deal with multidimensional arrays…stay tuned!

Final Variables

Declaring a variable with the `final` keyword makes it impossible to reassign 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 never 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`.

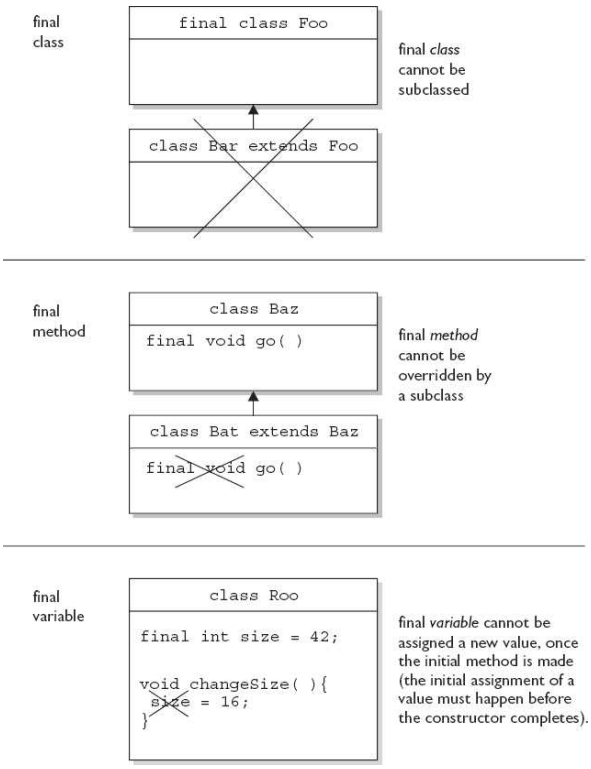

Figure 1-8: Effect of `final` on variables, methods, and classes

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. We were happy when serialization was added to the exam as of Java 5, but we're sad to say that as of Java 7, serialization is no longer on the exam.

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 it exists.

on the job The `volatile` modifier may also be applied to project managers!

Static Variables and Methods

Note: The discussion of `static` in this section DOES NOT include the new `static` interface method discussed earlier in this chapter. Don't you just love how the Java 8 folks reused important Java terms?

The `static` modifier is used to create variables and methods that will exist independently of any instances created for the class. All `static` members exist before you ever make a new instance of a class, and there will be only one copy of a `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 (not on the OCA 8 exam)
- 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 (unless they are nested)
- Method local inner classes (not on the OCA 8 exam)
- Inner class methods and instance variables (not on the OCA 8 exam)
- Local variables

CERTIFICATION OBJECTIVE: DECLARE AND USE ENUMS (OCA OBJECTIVE 1.2)

1.2 Define the structure of a Java class.

Note: During the creation of this book, Oracle adjusted some of the objectives for the OCA exam. We're not 100 percent sure that the topic of enums is included in the OCA exam, but we've decided that it's better to be safe than sorry, so we recommend that OCA candidates study this section. In any case, you're likely to encounter the use of enums in the Java code you read, so learning about them will pay off regardless.

Declaring enums

Java lets you restrict a variable to having one of only a few predefined 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, imagine you're creating a commercial-coffee-establishment application, and in your coffee shop application, you might want to restrict your `CoffeeSize` 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 Oracle code convention that constants are named in caps, it's a good idea.

The basic components of an `enum` are its constants (that is, `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!

Here's an example 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 (or, in general, enum classes can exist in their own file like `CoffeeSize.java`). But remember, in this case 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 an `enum` that isn't enclosed in a class 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 `enum`s 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 (when no other declarations for this `enum` follow):

```
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 `enum`s are not `String`s or `int`s! Each of the enumerated `CoffeeSize` values is actually an instance 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);

    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`, is an instance 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 enum`s 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, each of which can have its 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 {
    // 8, 10 & 16 are passed to the constructor
    BIG(8), HUGE(10), OVERWHELMING(16);
    CoffeeSize(int ounces) {    // constructor
      this.ounces = ounces;
    }

    private int ounces;         // an instance variable
    public int getOunces() {
      return ounces;
    }
}

class Coffee {
   CoffeeSize size;            // each instance of Coffee has an 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
      for(CoffeeSize cs: CoffeeSize.values())
         System.out.println(cs + " " + cs.getOunces());
   }
}
```

which produces:

```
8
BIG 8
HUGE 10
OVERWHELMING 16
```

Note: Every enum has a static method, values(), that returns an array of the enum's values in the order they're declared.

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 CoffeeSize 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. 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 more code follows

      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 asking, "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.

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 learned the basic syntax for the `java` and `javac` command-line programs.

You learned about when `main()` has superpowers and when it doesn't.

We covered the basics of `import` and `import static` statements. It's tempting to think that there's more to them than saving a bit of typing, but there isn't.

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.

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 nonfinal 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 `enums`. An `enum` is a safe and flexible way to implement constants. 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, in other words, *top-level* classes.

Java Features and Benefits (OCA Objective 1.5)

- While Java provides many benefits to programmers, for the exam you should remember that Java supports object-oriented programming in general, encapsulation, automatic memory management, a large API (library), built-in security features, multiplatform compatibility, strong typing, multithreading, and distributed computing.

Identifiers (OCA Objective 2.1)

- 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.

Executable Java Files and main() (OCA Objective 1.3)

- You can compile and execute Java programs using the command-line programs `javac` and `java`, respectively. Both programs support a variety of command-line options.
- The only versions of `main()` methods with special powers are those versions with method signatures equivalent to `public static void main(String[] args)`.
- `main()` can be overloaded.

Imports (OCA Objective 1.4)

- An `import` statement's only job is to save keystrokes.
- You can use an asterisk (`*`) to search through the contents of a single package.
- Although referred to as "static imports," the syntax is `import static`….
- You can import API classes and/or custom classes.

Source File Declaration Rules (OCA Objective 1.2)

- 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 it can have multiple `imports`.
- The `package` statement (if any) must be the first (noncomment) line in a source file.
- The `import` statements (if any) must come after the `package` statement (if any) and before the first class declaration.
- If there is no `package` statement, `import` statements must be the first (noncomment) 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 (OCA Objective 6.4)

- 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) (OCA Objectives 1.2, 7.1, and 7.5)

- 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 (OCA Objective 7.5)

- Usually, 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.
- Usually, an interface is like a 100 percent `abstract` class and is implicitly abstract whether or not you type the `abstract` modifier in the declaration.
- Usually interfaces have only `abstract` methods.
- Interface methods are by default `public` and usually `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.
- As of Java 8, interfaces can have concrete methods declared as either `default` or `static`.

Note: This section uses some concepts that we HAVE NOT yet covered. Don't panic: once you've read through all of the book, this section will make sense as a reference.

- 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 (OCA Objective 6.4)

- Methods and instance (nonlocal) variables are known as "members."
- Members can use all four access levels: `public`, `protected`, default, and `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 + 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's own subclasses.

Local Variables (OCA Objectives 2.1 and 6.4)

- 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 (OCA Objectives 7.1 and 7.5)

- `final` methods cannot be overridden in a subclass.
- `abstract` methods are declared with a signature, a return type, and an optional `throws` clause, but they are not implemented.
- `abstract` methods end in a semicolon—no curly braces.
- Three ways to spot a nonabstract method:
  - The method is not marked `abstract`.
  - The method has curly braces.
  - The method **MIGHT** have code between the curly braces.
- The first nonabstract (concrete) class to extend an `abstract` class must implement all of the `abstract` class's `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 (OCA Objective 1.2)

- 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.

Constructors (OCA Objectives 1.2, and 6.3)

- Constructors must have the same name as the class
- Constructors can have arguments, but they cannot have a return type.
- Constructors can use any access modifier (even `private`!).

Variable Declarations (OCA Objective 2.1)

- 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 reassigned once assigned a value.
  - `final` reference variables cannot refer to a different object once the object has been assigned to the `final` variable.
  - `final` 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 can't change.
- The `transient` modifier applies only to instance variables.
- The `volatile` modifier applies only to instance variables.

Array Declarations (OCA Objectives 4.1 and 4.2)

- 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 to the 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 (OCA Objective 6.2)

- They are not tied to any particular instance of a class.
- No class instances are needed in order to use `static` members of the class or interface.
- There is only one copy of a `static` variable/class, and all instances share it.

- `static` methods do not have direct access to nonstatic members.

enums (OCA Objective 1.2)

- An `enum` specifies a list of constant values assigned to a type.
- An `enum` is NOT a `String` or an `int`; an `enum` constant's type is the `enum` type. For example, `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`.
- `enum`s can contain constructors, methods, variables, and constant-specific 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};
    ```
  - `MyEnum.values()` returns an array of `MyEnum`'s values.

SELF TEST

The following questions will help you measure your understanding of the material presented in this chapter. Read all 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 `enum`s" and "OK, so that other guy knows `enum`s better than I do, but I bet he can't <insert something you *are* good at> like me."

1. Which are 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

2. Given: ?

```
class Rocket {
  private void blastOff() { System.out.print("bang "); }
}
public class Shuttle extends Rocket {
  public static void main(String[] args) {
    new Shuttle().go();
  }
  void go() {
    blastOff();
    // Rocket.blastOff();  // line A
  }
  private void blastOff() { System.out.print("sh-bang "); }
}
```

Which are true? (Choose all that apply.)

A. As the code stands, the output is `bang`

B. As the code stands, the output is `sh-bang`

C. As the code stands, compilation fails

D. If line A is uncommented, the output is `bang bang`

E. If line A is uncommented, the output is `sh-bang bang`

F. If line A is uncommented, compilation fails.
```

3. Given that the `for` loop's syntax is correct, and given: ?

```
import static java.lang.System.*;
class _ {
  static public void main(String[] __A_V_) {
    String $ = "";
    for(int x=0; ++x < __A_V_.length; ) // for loop
      $ += __A_V_[x];
    out.println($);
  }
}
```

And the command line:

```
java _ - A .
```

What is the result?

A. `-A`

B. A.

C. `-A.`

D. `_A.`

E. `_-A.`

F. Compilation fails

G. An exception is thrown at runtime

4. 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

5. Given two files: ?

```
1. package pkgA;
2. public class Foo {
3. int a = 5;
4. protected int b = 6;
5. public int c = 7;
6. }
```

```
3. package pkgB;
4. import pkgA.*;
5. public class Baz {
6.   public static void main(String[] args) {
7.     Foo f = new Foo();
8.     System.out.print(" " + f.a);
9.     System.out.print(" " + f.b);
```

```
10.     System.out.println(" " + f.c);
11.   }
12. }
```

What is the result? (Choose all that apply.)

A. `5 6 7`

B. 5 followed by an exception

C. Compilation fails with an error on line 7

D. Compilation fails with an error on line 8

E. Compilation fails with an error on line 9

F. Compilation fails with an error on line 10

6. Given: ?

```
1. public class Electronic implements Device
     { public void doIt() { } }
2.
3. abstract class Phone1 extends Electronic { }
4.
5. abstract class Phone2 extends Electronic
     { public void doIt(int x) { } }
6.
7. class Phone3 extends Electronic implements Device
     { public void doStuff() { } }
8.
9. interface Device { public void doIt(); }
```

What is the result? (Choose all that apply.)

A. Compilation succeeds

B. Compilation fails with an error on line 1

C. Compilation fails with an error on line 3

D. Compilation fails with an error on line 5

E. Compilation fails with an error on line 7

F. Compilation fails with an error on line 9

7. Given: ?

```
4. class Announce {
5.   public static void main(String[] args) {
6.     for(int __x = 0; __x < 3; __x++) ;
7.     int #lb = 7;
8.     long [] x [5];
9.     Boolean []ba[];
10.  }
11. }
```

What is the result? (Choose all that apply.)

A. Compilation succeeds

B. Compilation fails with an error on line 6

C. Compilation fails with an error on line 7

D. Compilation fails with an error on line 8

E. Compilation fails with an error on line 9

8. Given: ?

```
3. public class TestDays {
4.   public enum Days { MON, TUE, WED };
5.   public static void main(String[] args) {
6.     for(Days d : Days.values() )
7.       ;
8.     Days [] d2 = Days.values();
9.     System.out.println(d2[2]);
```

```
10.   }
11. }
```

What is the result? (Choose all that apply.)

A. `TUE`

B. `WED`

C. The output is unpredictable

D. Compilation fails due to an error on line 4

E. Compilation fails due to an error on line 6

F. Compilation fails due to an error on line 8

G. Compilation fails due to an error on line 9

9. Given: ?

```
4. public class Frodo extends Hobbit {
5.   public static void main(String[] args) {
6.     int myGold = 7;
7.     System.out.println(countGold(myGold, 6));
8.   }
9. }
10. class Hobbit {
11.   int countGold(int x, int y) { return x + y; }
12. }
```

What is the result?

A. `13`

B. Compilation fails due to multiple errors

C. Compilation fails due to an error on line 6

D. Compilation fails due to an error on line 7

E. Compilation fails due to an error on line 11

10. Given: ?

```
interface Gadget {
  void doStuff();
}
abstract class Electronic {
  void getPower() { System.out.print("plug in "); }
}
public class Tablet extends Electronic implements Gadget {
  void doStuff() { System.out.print("show book "); }
  public static void main(String[] args) {
    new Tablet().getPower();
    new Tablet().doStuff();
  }
}
```

Which are true? (Choose all that apply.)

A. The class `Tablet` will NOT compile

B. The interface `Gadget` will NOT compile

C. The output will be `plug in show book`

D. The `abstract` class `Electronic` will NOT compile

E. The class `Tablet` CANNOT both extend and implement

11. Given that the Integer class is in the java.lang package and given: ?

```
1. // insert code here
2. class StatTest {
3.   public static void main(String[] args) {
4.     System.out.println(Integer.MAX_VALUE);
5.   }
```

```
6. }
```

Which, inserted independently at line 1, compiles? (Choose all that apply.)

A. `import static java.lang;`

B. `import static java.lang.Integer;`

C. `import static java.lang.Integer.*;`

D. `static import java.lang.Integer.*;`

E. `import static java.lang.Integer.MAX_VALUE;`

F. None of the above statements are valid import syntax

12. Given: ?

```
interface MyInterface {
  // insert code here
}
```

Which lines of code—inserted independently at `insert code here`—will compile? (Choose all that apply.)

A. `public static m1() {;}`

B. `default void m2() {;}`

C. `abstract int m3();`

D. `final short m4() {return 5;}`

E. `default long m5();`

F. `static void m6() {;}`

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

A. Java is a dynamically typed programming language

B. Java provides fine-grained control of memory through the use of pointers

C. Java provides programmers the ability to create objects that are well encapsulated

D. Java provides programmers the ability to send Java objects from one machine to another

E. Java is an implementation of the ECMA standard

F. Java's encapsulation capabilities provide its primary security mechanism

Answers

1. ☑ **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. (OCA Objective 7.5)

2. ☑ **B** and **F** are correct. Since `Rocket.blastOff()` is `private`, it can't be overridden, and it is invisible to class `Shuttle`.

☒ **A, C, D**, and **E** are incorrect based on the above. (OCA Objective 6.4)

3. ☑ **B** is correct. This question is using valid (but inappropriate and weird) identifiers, static imports, `main()`, and pre-incrementing logic. (Note: You might get a compiler warning when compiling this code.)

☒ **A, C, D, E, F**, and **G** are incorrect based on the above. (OCA Objective 1.2)

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

☒ **B, C, D, E**, and **F** are incorrect; these lines all use correct syntax. (OCA Objective 1.2)

5. ☑ **D** and **E** are correct. Variable a has default access, so it cannot be accessed from outside the package. Variable b has protected access in `pkgA`.

   ☒ **A, B, C**, and **F** are incorrect based on the above information. (OCA Objectives 1.4 and 6.5)

6. ☑ **A** is correct; all of these are legal declarations.

   ☒ **B, C, D, E**, and **F** are incorrect based on the above information. (OCA Objective 7.5)

7. ☑ **C** and **D** are correct. Variable names cannot begin with a #, and an array declaration can't include a size without an instantiation. The rest of the code is valid.

   ☒ **A, B**, and **E** are incorrect based on the above. (OCA Objective 2.1)

8. ☑ **B** is correct. Every `enum` comes with a `static values()` method that returns an array of the `enum`'s values in the order in which they are declared in the `enum`.

   ☒ **A, C, D, E, F**, and **G** are incorrect based on the above information. (OCP Objective 1.2)

9. ☑ **D** is correct. The `countGold()` method cannot be invoked from a static context.

   ☒ **A, B, C**, and **E** are incorrect based on the above information. (OCA Objective 6.2)

10. ☑ **A** is correct. By default, an interface's methods are `public` so the `Tablet.doStuff` method must be public, too. The rest of the code is valid.

    ☒ **B, C, D**, and **E** are incorrect based on the above. (OCA Objective 7.5)

11. ☑ **C** and **E** are correct syntax for static imports. Line 4 isn't making use of `static imports`, so the code will also compile with none of the imports.

    ☒ **A, B, D**, and **F** are incorrect based on the above. (OCA Objective 1.4)

12. ☑ **B, C**, and **F** are correct. As of Java 8, interfaces can have `default` and `static` methods.

    ☒ **A, D**, and **E** are incorrect. **A** has no return type; **D** cannot have a method body; and **E** needs a method body. (OCA Objective 7.5)

13. ☑ **C** and **D** are correct.

    ☒ **A** is incorrect because Java is a statically typed language. **B** is incorrect because it does not provide pointers. **E** is incorrect because JavaScript is an implementation of the ECMA standard, not Java. **F** is incorrect because the use of bytecode and the JVM provide Java's primary security mechanisms.