1

Declarations, Access Control, and Enums

CERTIFICATION OBJECTIVES

- Declare Classes and Interfaces
- Declare Class Members
- Declare Constructors and Arrays
- Create static Class Members
- Use enums

✓ Two-Minute Drill

Q&A Self Test

We assume that most of our readers have earned their OCA 8 Java certification and are reading this book in pursuit of the OCP 8 Java certification. If that's you, congratulations on earning your OCA 8! If you're NOT pursuing an Oracle Java certification, we'd like to think you might still find this book helpful. Oracle's certification exams are well regarded in the industry, and understanding the concepts in this book will make you a better Java programmer. But make no mistake, this book is REALLY focused on helping you pass the OCP 8.

Java Class Design and Object Orientation: A Refresher

If you compare the exam objectives of the OCA 8 exam and the OCP 8 exam, you'll notice some overlapping concepts. Specifically, many of the objectives in OCA 8 section 6 (Working with Methods and Encapsulation) and OCA 8 section 7 (Working with Inheritance) overlap heavily with the objectives in OCP 8 section 1 (Java Class Design) and OCP 8 section 2 (Advanced Java Class Design). This chapter is focused on most of those areas of conceptual overlap:

- 1.2 Inheritance, visibility, and composition (HAS-A)
- 1.6 The static keyword and init blocks
- 2.1 Abstract classes

- 2.2 The final keyword
- 2.4 Enums
- 2.5 Interfaces

You could consider this chapter a bit of a refresher from the OCA 8 exam, and if you feel really solid on the objectives listed above, you could consider skipping this chapter.

CERTIFICATION OBJECTIVE

Define Classes and Interfaces (OCP Objectives 1.2, 2.1, and 2.2)

1.2 Implement inheritance including visibility modifiers and composition.
2.1 Develop code that uses abstract classes and methods.
2.2 Develop code that uses the final keyword.

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 will be covered in Chapter 7. 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, constructor, 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.

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 or use the fully qualified name.

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 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 classpath 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 (OCP Objective 2.5)

2.5 Develop code that declares, implements, and/or extends interfaces and use the @Override annotation.

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 Java 8's new interface-related features—default and static methods—we will discuss interfaces from a traditional perspective, which is, again, defining a contract for *what* a class can do.

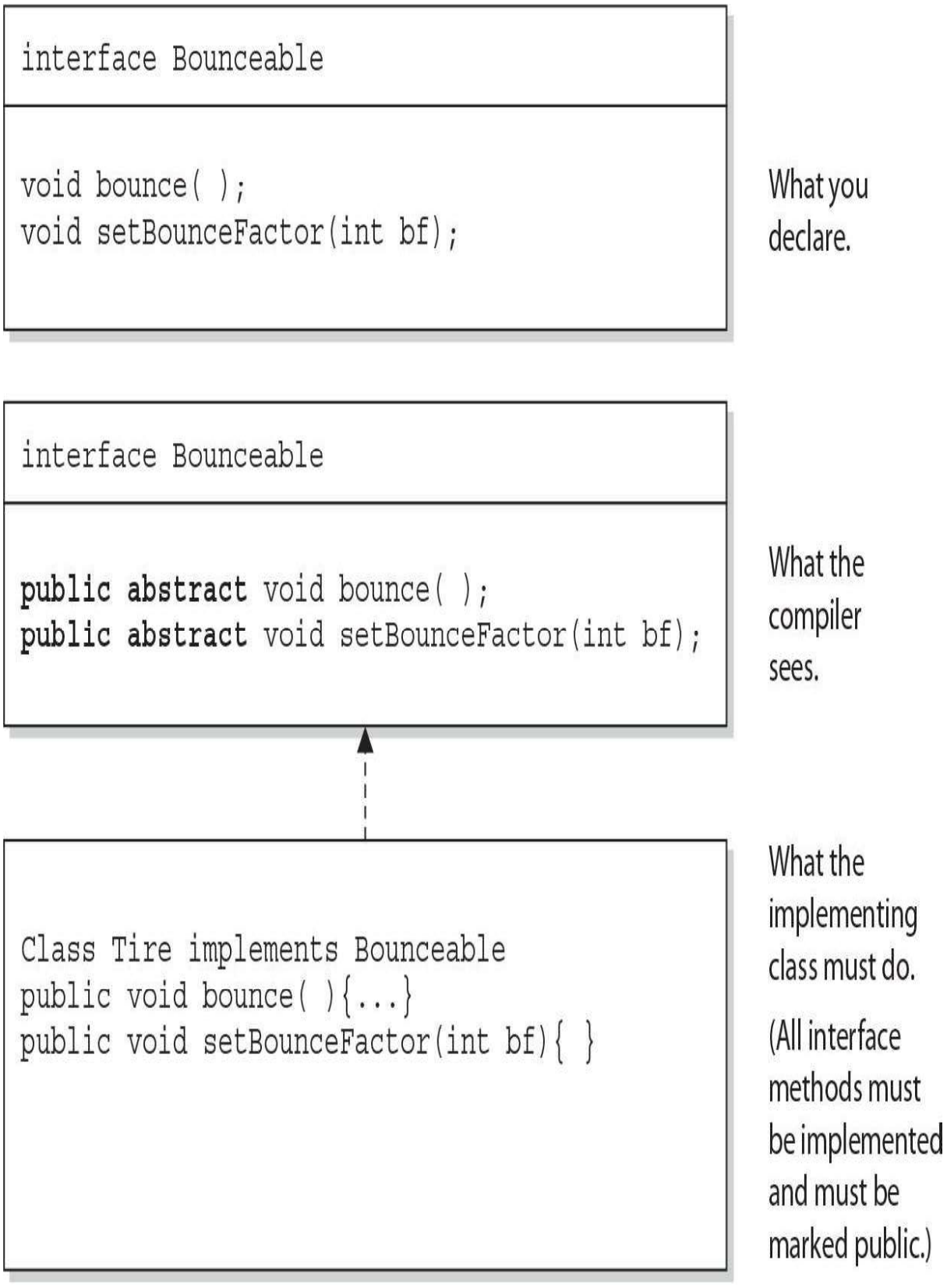
An interface is a contract. You could write an interface `Bounceable`, for example, that says in effect, "This is the `Bounceable` interface. Any concrete class type that implements this interface must agree to write the code for the `bounce()` and `setBounceFactor()` methods."

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 and 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` whereas `Tire` extends only `java.lang.Object`. But by making both `Ball` and `Tire` implement `Bounceable`, you're saying that `Ball` and `Tire` can be treated as "Things that can bounce," which in Java translates to "Things on which you can invoke the `bounce()` and `setBounceFactor()` methods." Figure 1-1 illustrates the relationship between interfaces and classes.

FIGURE 1-1

The relationship between interfaces and classes



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

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

But 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:

- All interface methods are implicitly `public`. Unless declared as `default` or `static`, they are also implicitly `abstract`. In other words, you do not need to actually type the `public` or `abstract` modifiers in the method declaration, but the method is still always `public` and `abstract`.
- All variables defined in an interface must be `public`, `static`, and `final`—in other words, interfaces can declare only constants, not instance variables.
- Interface methods 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, because all interface methods are implicitly `public` and `abstract`. Given that rule, you can see 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 the next chapter.) These concrete methods are called `default` methods. Later in the book (mostly in Chapter 2), 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 later in the chapter.

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
```

Later, we'll return to our discussion of default methods and static methods for interfaces.

CERTIFICATION OBJECTIVE

Declare Class Members (OCP Objectives 1.2, 1.6, 2.1, and 2.2)

1.2 Implement inheritance including visibility modifiers and composition.
1.6 Develop code that uses static keyword on initialize blocks, variables, methods, and classes.
2.1 Develop code that uses abstract classes and methods.
2.2 Develop code that uses the final keyword.

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 in this section of this chapter.

It's crucial that you know access control inside and outside for the exam. There will be quite a few questions in which 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
  }
}
```

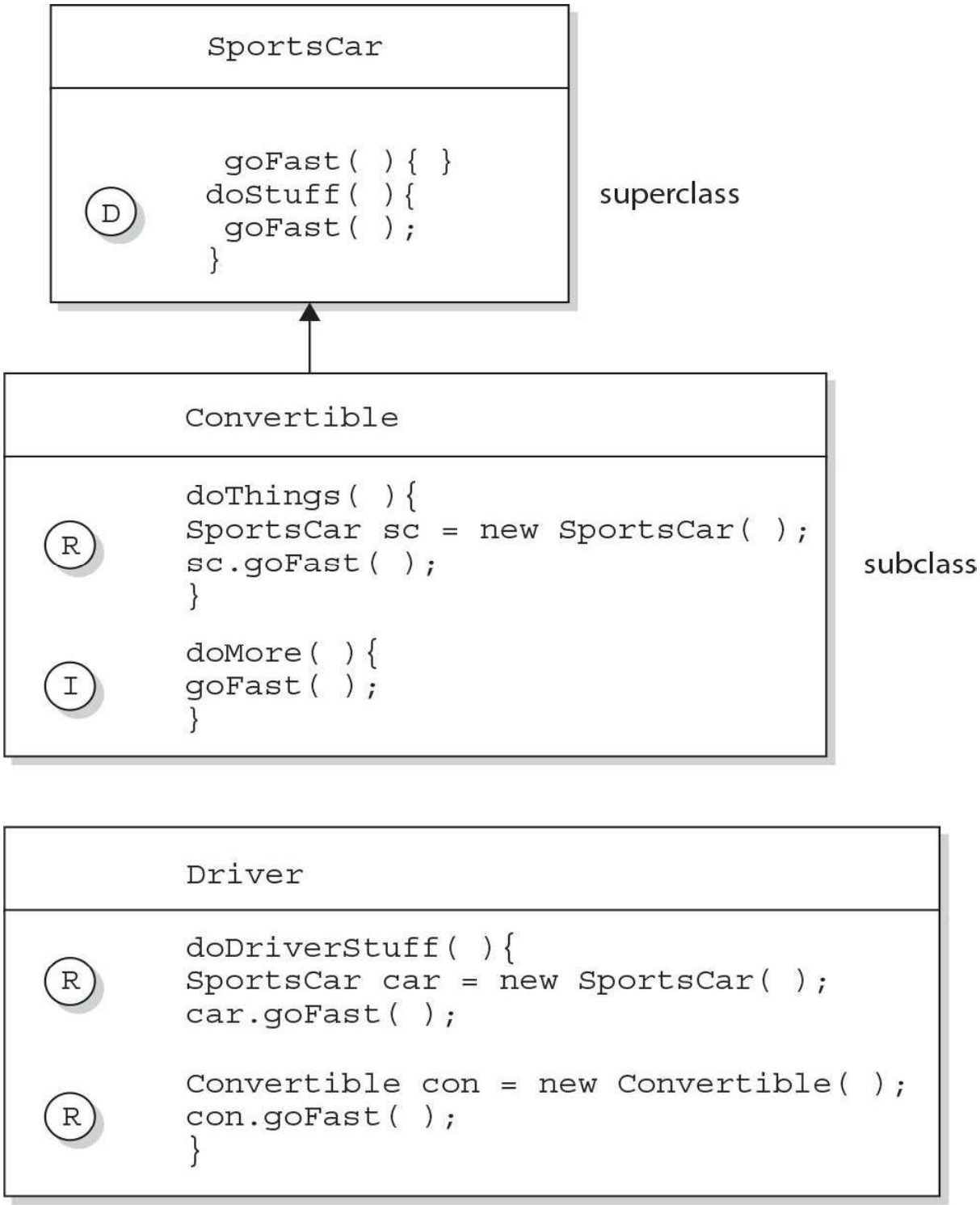
The second type of access revolves around which, if any, members of a superclass a subclass can access through inheritance. We're not looking at whether the subclass can, say, invoke a method on an instance of the superclass (which would just be an example of the first type of access). Instead, we're 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



Three ways to access a method:

(D) Invoking a method declared in the same class

(R) Invoking a method using a reference of the class

(I) Invoking an inherited method

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() { // do fun, secret stuff
      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:

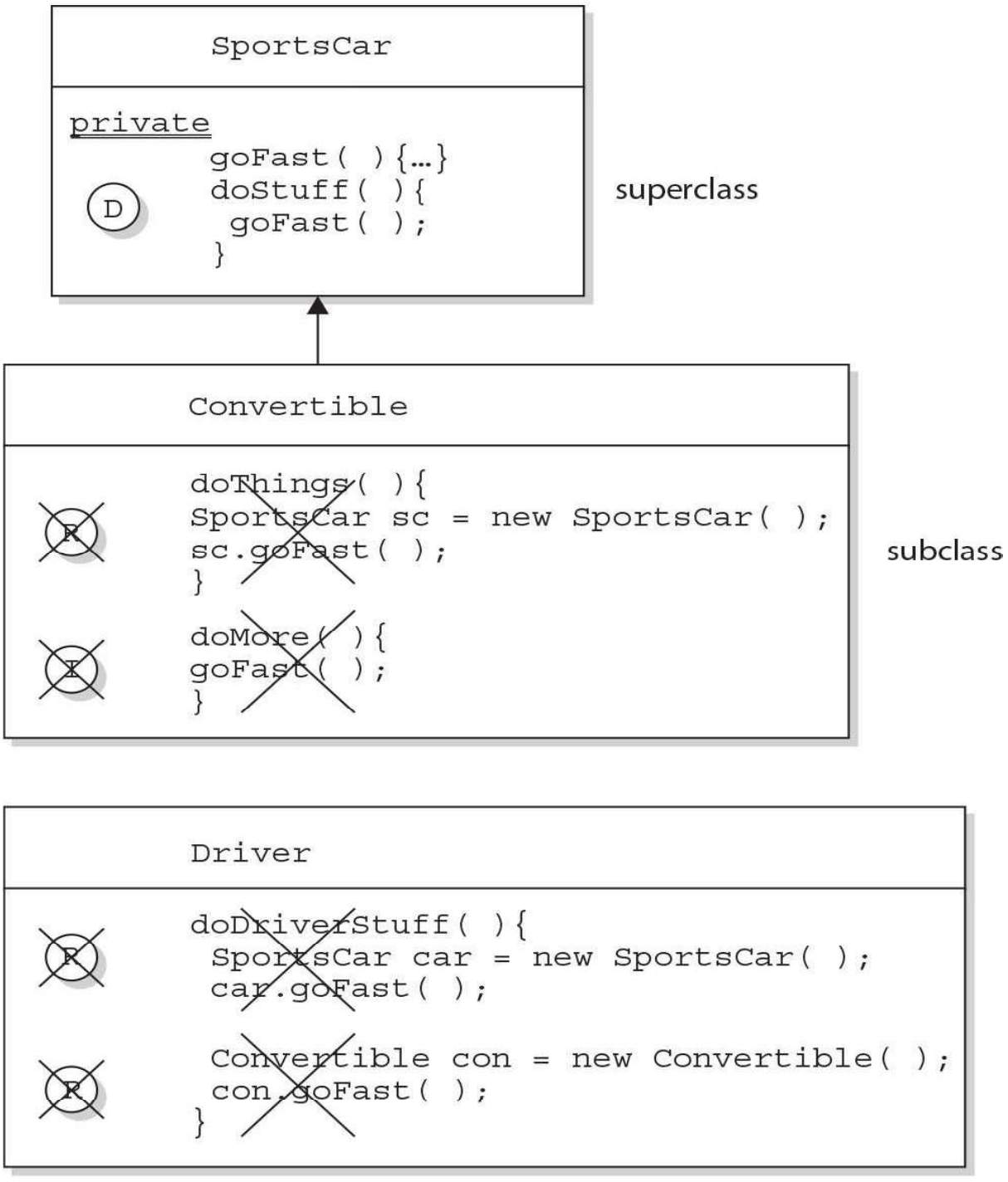
```
%javac Cloo.java
Cloo.java:4: Undefined method: doRooThings()
      System.out.println(doRooThings());
1 error
```

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 chapter, but for now, just remember that a method marked `private` cannot be overridden. Figure 1-3 illustrates the effects of the `public` and `private` modifiers on classes from the same or different packages.

FIGURE 1-3

Effects of public and private access

The effect of private access control



Three ways to access a method:

(D) Invoking a method declared in the same class

(R) Invoking a method using a reference of the class

(I) Invoking an inherited method

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");
  }
}
```

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 you'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 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, 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.*

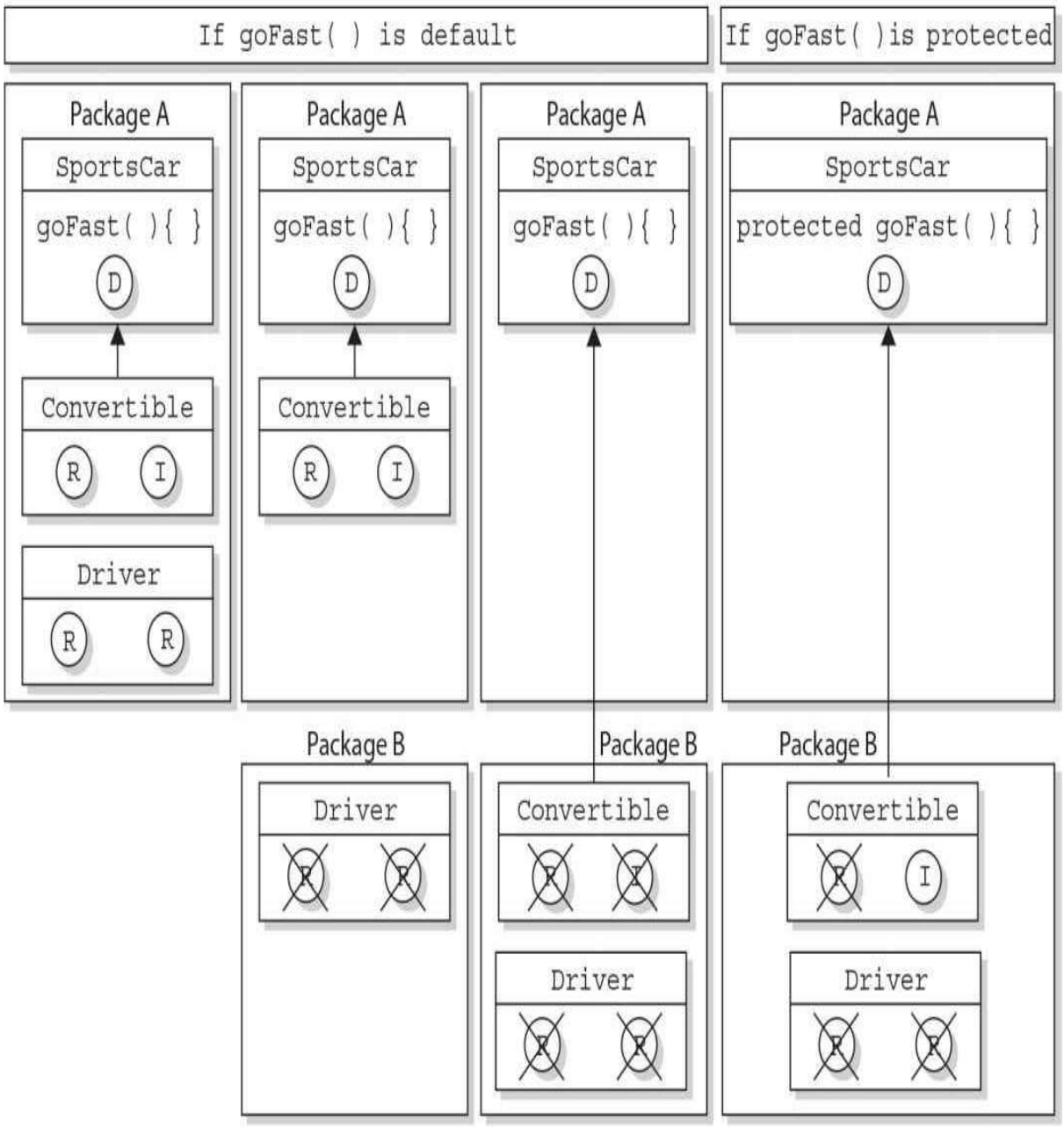
But there's still one more issue we haven't looked at: What does a `protected` member look like to other classes trying to use the subclass-outside-the-package to get to the subclass's inherited `protected` superclass member? For example, using our previous `Parent`/`Child` classes, what happens if some 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.

FIGURE 1-4

Effects of `protected` access



Key:

D

```
goFast( ){ }
doStuff( ){
 goFast( );
}
```

Where goFast is *Declared* in the same class.

R

```
doThings( ){
 SportsCar sc = new SportsCar( );
 sc.goFast( );
}
```

Invoking goFast( ) using a *Reference* to the class in which goFast( ) was declared.

I

```
doMore( ){
 goFast( );
}
```

Invoking the goFast( ) method *Inherited* from a superclass.

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

Default Details

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

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

Notice we didn't place an access modifier in front of the variable `x`. Remember, 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 something like this:

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

The compiler gives an 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 1-1 shows all the combinations of access and visibility; you really should spend some time with it. Next, we're going to dig into the other (nonaccess) modifiers that you can apply to member

declarations.

TABLE 1-1 Determining Access to Class Members

Visibility	Public	Protected	*Default*	Private
From the same class	Yes	Yes	Yes	Yes
From any class in the same package	Yes	Yes	Yes	No
From 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` and `synchronized`, 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. In the case of reference variables, what this means is that you might be able to change the values in the object the `final` reference variable refers to, but you CANNOT force the `final` reference variable to refer to a different object.

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, 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.

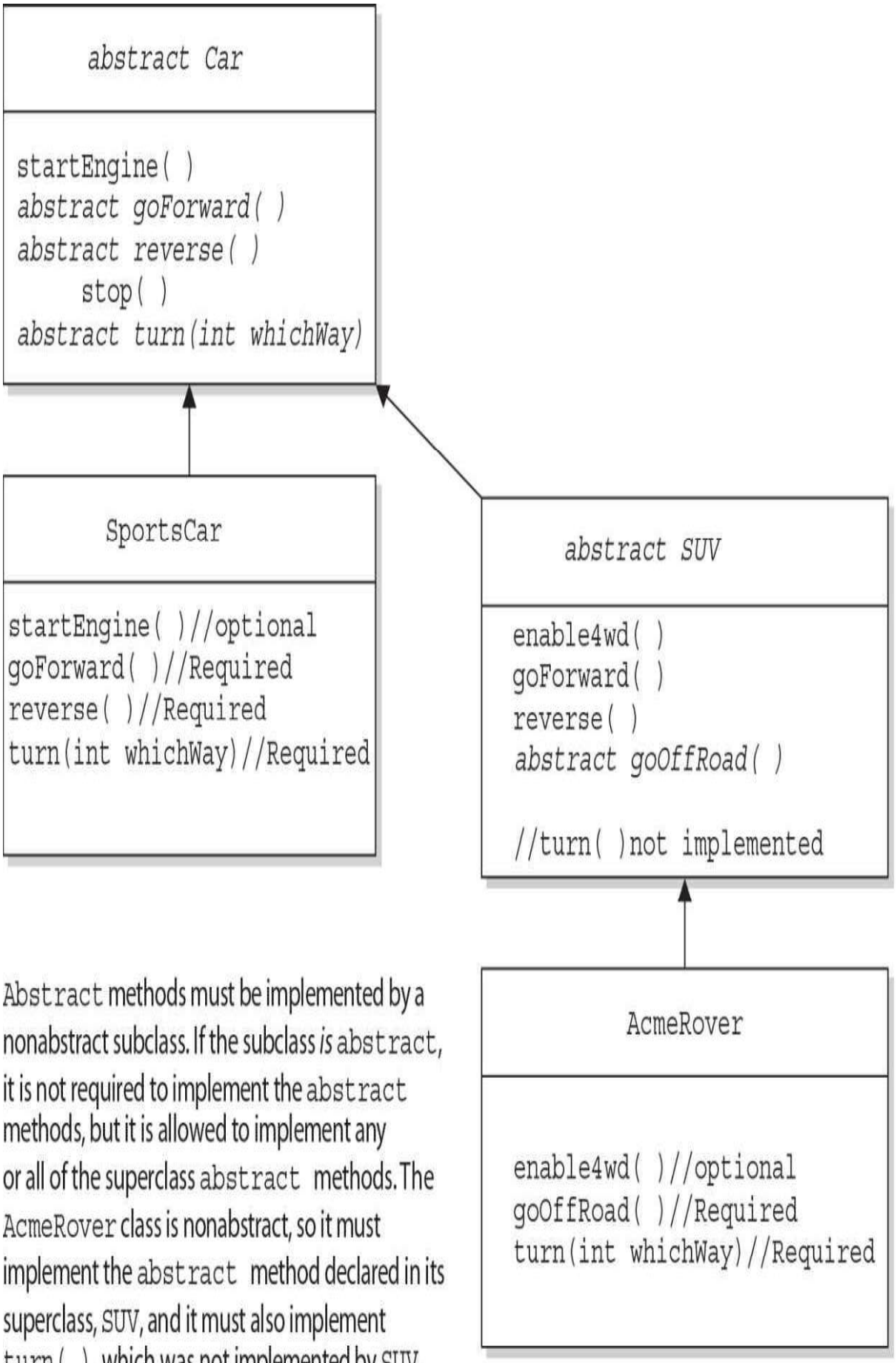
Concrete just means nonabstract, so if you have an `abstract` class extending another `abstract` class, the `abstract` subclass doesn't need to provide implementations for the inherited `abstract` methods. Sooner or later, though, somebody's going to make a nonabstract subclass (in other words, a class 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, because `Car`, like `Vehicle`, is `abstract`. Figure 1-5 illustrates the effects of the `abstract` modifier on concrete and `abstract` subclasses.

FIGURE 1-5

The effects of the `abstract` modifier on concrete and `abstract` subclasses



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

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

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

Class B won't compile because it doesn't implement the inherited `abstract` method `foo()`. Although the `foo(int i)` method in class B might appear to be an implementation of the superclass'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 36later 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. In Chapter 10, we'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).

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:

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

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

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

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 (preferred but optional), 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 Animal objects
                                        // (no space before the argument is legal)
```

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. Here are 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.

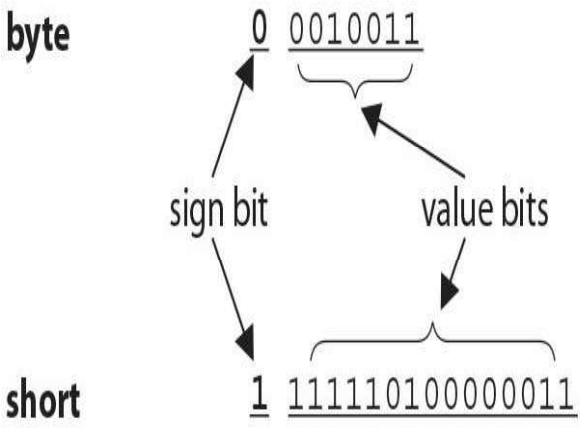
You will also need to know that the number types (both integer and floatingpoint types) are all signed and how that affects their ranges.

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



sign bit: 0 = positive
1 = negative

value bits:

byte: 7 bits can represent 2^7 or 128 different values:
0 thru 127 -or- –128 thru –1

short: 15 bits can represent 2^{15} or 32768 values:
0 thru 32767 -or- –32768 thru –1

Table 1-2 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.

TABLE 1-2 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

Determining the range for floating-point numbers is complicated, 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 ($2^{16} - 1$). Remember from the OCA 8 exam 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:

```
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
}
```

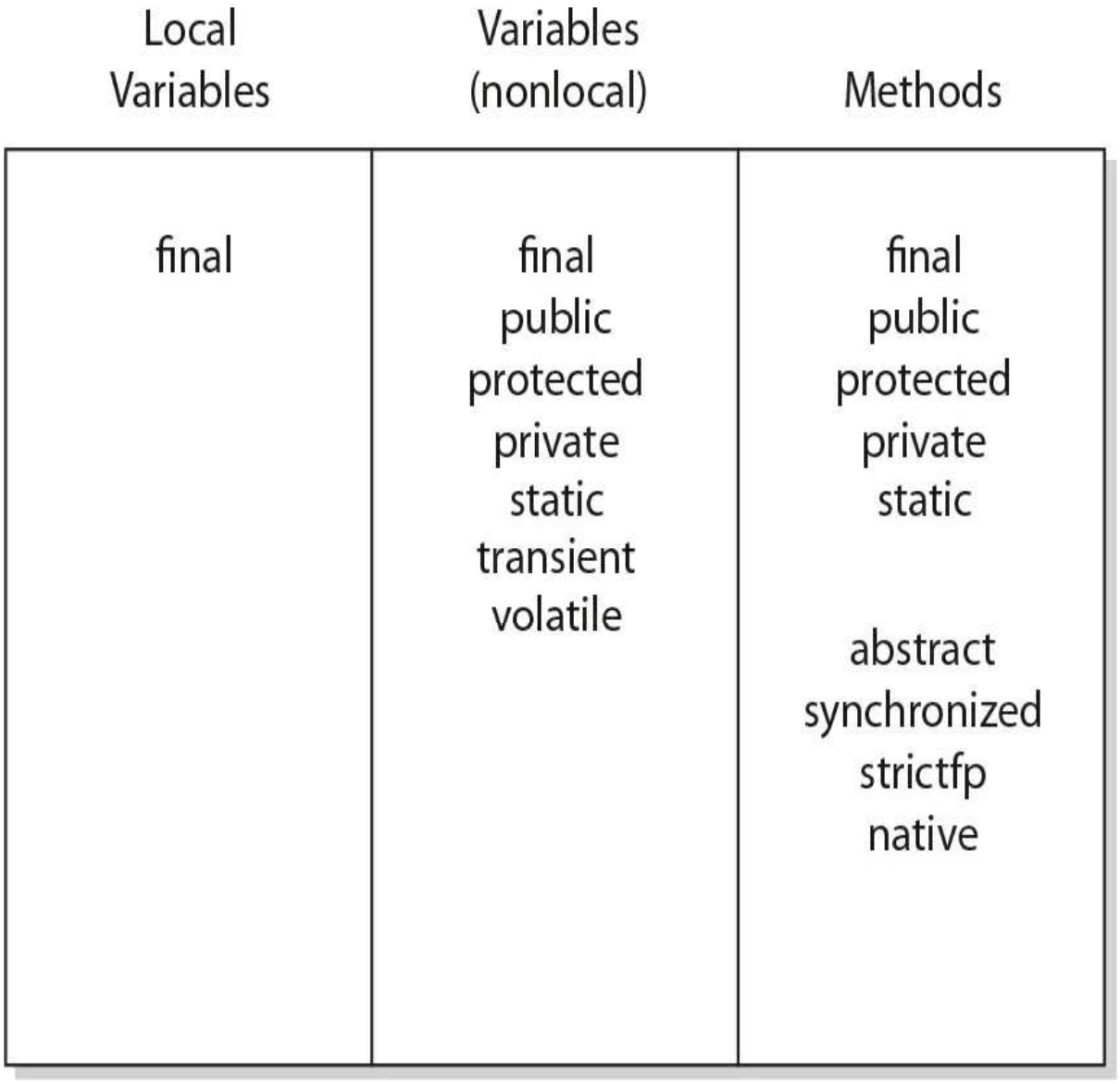
The preceding `Employee` class says that each employee instance will know its own name, title, and manager. In other words, each instance can have its own unique values for those three fields. For the exam, you need to know that instance variables

- 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

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

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. 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

programmers use that expression, you'll know that they're just too lazy to phrase it in a technically precise way. You can tell them we said that—unless they know 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 you saw earlier, local variables can be marked `final`. Remember, 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)*

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, and Chapter 6 discusses Collections 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
```

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.

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.

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

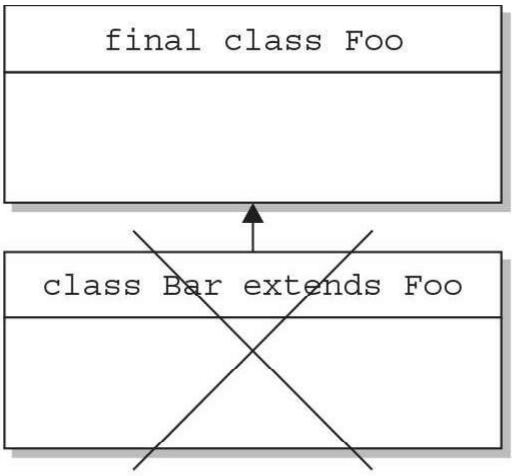
variable cannot be changed. In other words, a `final` reference still allows you to modify the state of the object it refers to, but you can't modify the reference variable to make it refer to a different object. Burn this in: there are no `final` objects, only `final` references.

We'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

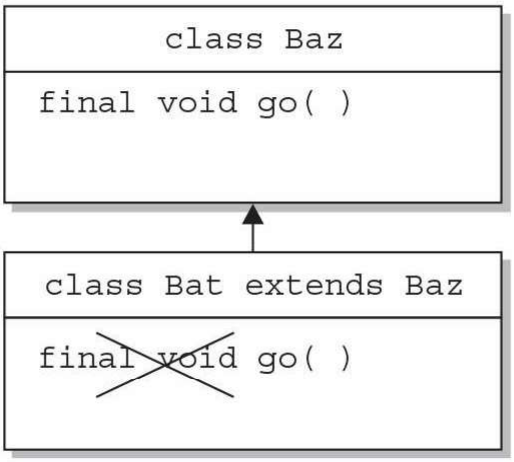
final class



final *class* cannot be subclassed

---

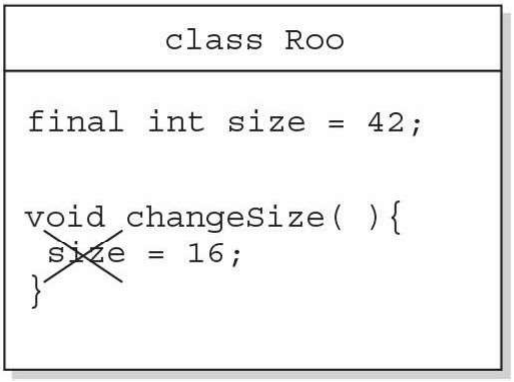
final method



final *method* cannot be overridden by a subclass

---

final variable



final *variable* cannot be assigned a new value once the initial method is made (the initial assignment of a value must happen before the constructor completes).

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 Java's coolest features; 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. In Chapter 5, we'll do a deep dive into serialization.

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
- 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 (OCP Objective 2.4)

2.4 Use enumerated types including methods, and constructors in an enum type.

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 enums` as existing in an array of type `CoffeeSize`, and as you'll see in a later chapter, you can iterate through the values of an `enum` by invoking the `values()` method on any `enum` type.

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 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.

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 `enum`s 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:

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, unless you go out of your way to

develop your class to create "immutable objects," which is a design approach we'll discuss in Chapter 2. A reference variable marked `final` can never 56be 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.

Remember that although 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.

Class Access Modifiers (OCP Objective 1.2)

- 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) (OCP Objectives 1.2, 2.1, and 2.2)

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

Interface Implementation (OCP Objectives 1.2, 2.1, 2.2, and 2.5)

- 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 the entire 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 (OCP Objective 1.2)

- 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 (OCP Objective 2.2)

- 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 (OCP Objectives 2.1 and 2.2)

- `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`.

Methods with var-args (OCP 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 (OCP Objectives 1.2 and 2.4)

- 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 (OCP Objectives 2.1 and 2.2)

- Instance variables can
  - Have any access control
  - Be marked `final` or `transient`
- Instance variables can't be `abstract` or `synchronized`.
- 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 (OCP Objective 1.2)

- 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 (OCP Objective 1.6)

- 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 (OCP Objective 2.4)

- 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.

Q SELF TEST

The following questions will help 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: "I am smart enough to understand `enums`," and "OK, so that other guy knows `enums` better than I do, but I bet he can't <insert something you *are* good at> like me."

1. 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:

```
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

4. 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

5. 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

6. 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

7. 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

8. 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

9. 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() {;}`

A SELF TEST 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. (OCP Objectives 1.2 and 2.5)

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. (OCP Objective 1.2 and 1.3)

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

   ☒ B, C, D, E, and F are incorrect; these lines all use correct syntax. (OCP Objective 2.4)

4. ☑ 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. (OCP Objective 1.2)

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

   ☒ B, C, D, E, and F are incorrect based on the above information. (OCP Objectives 1.2, 2.1, and 2.5)

6. ☑ 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 Objectives 1.6 and 2.4)

7. ☑ 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. (OCP Objective 1.6)

8. ☑ 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. (OCP Objectives 1.2 and 2.5)

9. ☑ 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. (OCP Objective 2.5)