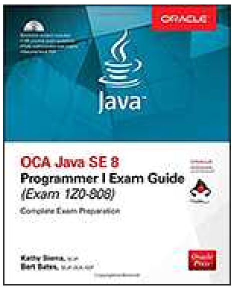

OCA Java SE 8 Programmer I Exam Guide (Exams 1Z0-808)

by Kathy Sierra and Bert Bates

Chapter 2: Object Orientation

CERTIFICATION OBJECTIVES

- Describe Encapsulation
- Implement Inheritance
- Use IS-A and HAS-A Relationships (OCP)
- Use Polymorphism
- Use Overriding and Overloading
- Understand Casting
- Use Interfaces
- Understand and Use Return Types
- Develop Constructors
- Use static Members
- Two-Minute Drill
- Self Test

Being an Oracle Certified Associate (OCA) 8 means you must be at one with the object-oriented aspects of Java. You must dream of inheritance hierarchies; the power of polymorphism must flow through you; and encapsulation must become second nature to you. (Coupling, cohesion, composition, and design patterns will become your bread and butter when you're an Oracle Certified Professional [OCP] 8.) This chapter will prepare you for all the object-oriented objectives and questions you'll encounter on the exam. We have heard of many experienced Java programmers who haven't really become fluent with the object-oriented tools that Java provides, so we'll start at the beginning.

CERTIFICATION OBJECTIVE: ENCAPSULATION (OCA OBJECTIVES 6.1 AND 6.5)

6.1 Create methods with arguments and return values; including overloaded methods.

6.5 Apply encapsulation principles to a class.

Imagine you wrote the code for a class and another dozen programmers from your company all wrote programs that used your class. Now imagine that later on, you didn't like the way the class behaved, because some of its instance variables were being set (by the other programmers from within their code) to values you hadn't anticipated. *Their* code brought out errors in *your* code. (Relax, this is just hypothetical.) Well, it is a Java program, so you should be able to ship out a newer version of the class, which they could replace in their programs without changing any of their own code.

This scenario highlights two of the promises/benefits of an object-oriented (OO) language: flexibility and maintainability. But those benefits don't come automatically. You have to do something. You have to write your classes and code in a way that supports flexibility and maintainability. So what if Java supports OO? It can't design your code for you. For example, imagine you made your class with `public` instance variables, and those other programmers were setting the instance variables directly, as the following code demonstrates:

```
public class BadOO {
  public int size;
  public int weight;
  ...
  }
  public class ExploitBadOO {
    public static void main (String [] args) {
      BadOO b = new BadOO();
      b.size = -5; // Legal but bad!!
    }
  }
```

And now you're in trouble. How are you going to change the class in a way that lets you handle the issues that come up when somebody changes the `size` variable to a value that causes problems? Your only choice is to go back in and write method code to adjust `size` (a `setSize(int a)` method, for example) and then insulate the `size` variable with, say, a private access modifier. But as soon as you make that change to your code, you break everyone else's!

The ability to make changes in your implementation code without breaking the code of others who use your code is a key benefit of encapsulation. You want to hide implementation details behind a public programming interface. By *interface*, we mean the set of accessible methods your code makes available for other code to call—in other words, your code's API. By hiding implementation details, you can rework your method code (perhaps also altering the way variables are used by your class) without forcing a change in the code that calls your changed method.

If you want maintainability, flexibility, and extensibility (and, of course, you do), your design must include encapsulation. How do you do that?

- Keep instance variables hidden (with an access modifier, often `private`).
- Make `public` accessor methods, and force calling code to use those methods rather than directly accessing the instance variable. These so-called accessor methods allow users of your class to **set** a variable's value or **get** a variable's value.
- For these accessor methods, use the most common naming convention of `set<SomeProperty>` and `get<SomeProperty>`.

Figure 2-1 illustrates the idea that encapsulation forces callers of our code to go through methods rather than accessing variables directly.

We call the access methods *getters* and *setters*, although some prefer the fancier terms *accessors* and *mutators*. (Personally, we don't like the word "mutate.") Regardless of what you call them, they're methods that other programmers must go through in order to access your instance variables. They look simple, and you've probably been using them forever:

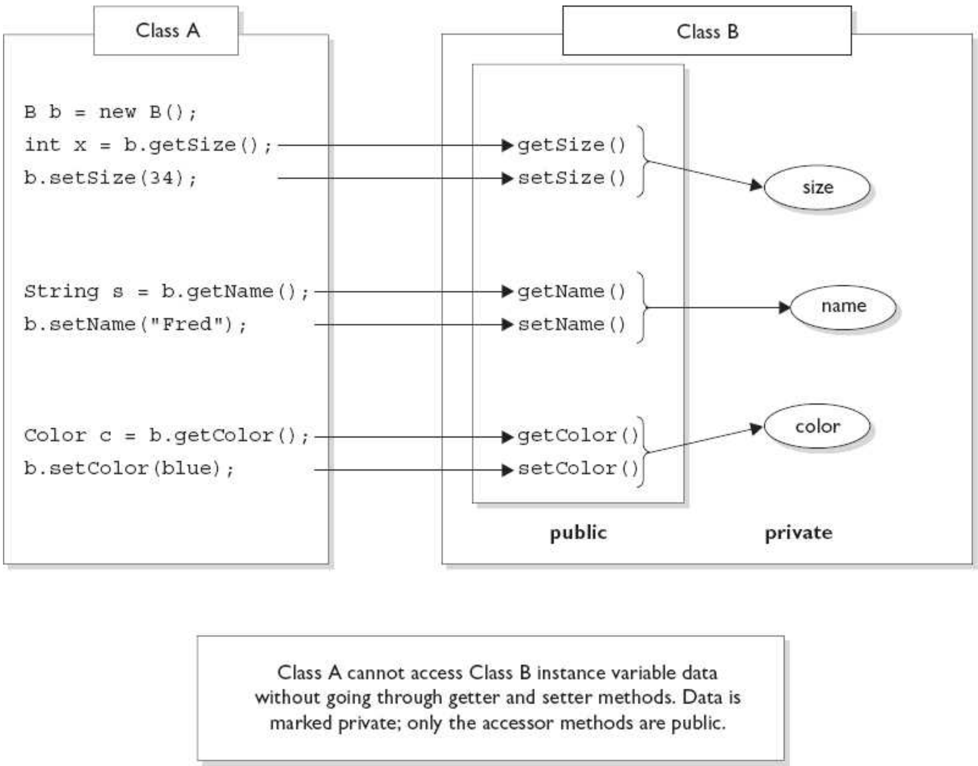

Figure 2-1: The nature of encapsulation

```
public class Box {
  // hide the instance variable; only an instance
  // of Box can access it
  private int size;
  // Provide public getters and setters
  public int getSize() {
    return size;
  }
  public void setSize(int newSize) {
    size = newSize;
  }
}
```

Wait a minute. How useful is the previous code? It doesn't even do any validation or processing. What benefit can there be from having getters and setters that add no functionality? The point is, you can change your mind later and add more code to your methods without breaking your API. Even if today you don't think you really need validation or processing of the data, good OO design dictates that you plan for the future. To be safe, force calling code to go through your methods rather than going directly to instance variables. *Always*. Then you're free to rework your method implementations later, without risking the wrath of those dozen programmers who know where you live.

Note: In Chapter 6 we'll revisit the topic of encapsulation as it applies to instance variables that are also reference variables. It's trickier than you might think, so stay tuned! (Also, we'll wait until Chapter 6 to challenge you with encapsulation-themed mock questions.)

Exam Watch

Look out for code that appears to be asking about the behavior of a method, when the problem is actually a lack of encapsulation. Look at the following example, and see if you can figure out what's going on:

```
class Foo {
  public int left = 9;
  public int right = 3;
  public void setLeft(int leftNum) {
    left = leftNum;
    right = leftNum/3;
  }
  // lots of complex test code here
}
```

Now consider this question: Is the value of right always going to be one-third the value of left? It looks like it will, until you realize that users of the `Foo` *class don't need to use the* `setLeft()` *method! They can simply go straight to the instance variables and change them to any arbitrary int value*.

CERTIFICATION OBJECTIVE: INHERITANCE AND POLYMORPHISM (OCA OBJECTIVES 7.1 AND 7.2)

7.1 Describe inheritance and its benefits.

7.2 Develop code that demonstrates the use of polymorphism; including overriding and object type versus reference type (sic).

Inheritance is everywhere in Java. It's safe to say that it's almost (almost?) impossible to write even the tiniest Java program without using inheritance. To explore this topic, we're going to use the `instanceof` operator, which we'll discuss in more detail in Chapter 4. For now, just remember that `instanceof` returns `true` if the reference variable being tested is of the type being compared to. This code

```
class Test {
  public static void main(String [] args) {
    Test t1 = new Test();
    Test t2 = new Test();
    if (!t1.equals(t2))
      System.out.println("they're not equal");
    if (t1 instanceof Object)
      System.out.println("t1's an Object");
  }
}
```

produces this output:

```
they're not equal
t1's an Object
```

Where did that `equals` method come from? The reference variable `t1` is of type `Test`, and there's no `equals` method in the `Test` class. Or is there? The second `if` test asks whether `t1` is an instance of class `Object`, and because it *is* (more on that soon), the `if` test succeeds.

Hold on…how can `t1` be an instance of type `Object`, when we just said it was of type `Test`? I'm sure you're way ahead of us here, but it turns out that every class in Java is a subclass of class `Object` (except, of course, class `Object` itself). In other words, every class you'll ever use or ever write will inherit from class `Object`. You'll always have an `equals` method, a `clone` method, `notify`, `wait`, and others available to use. Whenever you create a class, you automatically inherit all of class `Object`'s methods.

Why? Let's look at that `equals` method for instance. Java's creators correctly assumed that it would be very common for Java programmers to want to compare instances of their classes to check for equality. If class `Object` didn't have an `equals` method, you'd have to write one yourself—you and every other Java programmer. That one `equals` method has been inherited billions of times. (To be fair, `equals` has also been *overridden* billions of times, but we're getting ahead of ourselves.)

The Evolution of Inheritance

Up until Java 8, when the topic of inheritance was discussed, it usually revolved around subclasses inheriting methods from their superclasses. While this simplification was never perfectly correct, it became less correct with the new features available in Java 8. As the following table shows, it's now possible to inherit concrete methods from interfaces. This is a big change. For the rest of the chapter, when we talk about inheritance generally, we will tend to use the terms "subtypes" and "supertypes" to acknowledge that both classes and interfaces need to be accounted for. We will tend to use the terms "subclass" and "superclass" when we're discussing a specific example that's under discussion. Inheritance is a key aspect of most of the topics we'll be discussing in this chapter, so be prepared for LOTS of discussion about the interactions between supertypes and subtypes!

As you study the following table, you'll notice that as of Java 8 interfaces can contain two types of concrete methods, `static` and `default`. We'll discuss these important additions later in this chapter.

Table 2-1 summarizes the elements of classes and interfaces relative to inheritance.

Table 2-1: Inheritable Elements of Classes and Interfaces

Elements of Types	Classes	Interfaces
Instance variables	Yes	Not applicable
Static variables	Yes	Only constants
Abstract methods	Yes	Yes
Instance methods	Yes	Java 8, default methods
Static methods	Yes	Java 8, inherited no, accessible yes
Constructors	No	Not applicable
Initialization blocks	No	Not applicable

For the exam, you'll need to know that you can create inheritance relationships in Java by *extending* a class or by implementing an interface. It's also important to understand that the two most common reasons to use inheritance are

- To promote code reuse
- To use polymorphism

Let's start with reuse. A common design approach is to create a fairly generic version of a class with the intention of creating more specialized subclasses that inherit from it. For example:

```
class GameShape {
  public void displayShape() {
    System.out.println("displaying shape");
  }
  // more code
}

class PlayerPiece extends GameShape {
  public void movePiece() {
    System.out.println("moving game piece");
  }
  // more code
}


public class TestShapes {
  public static void main (String[] args) {
    PlayerPiece shape = new PlayerPiece();
    shape.displayShape();
    shape.movePiece();
  }
}
```

outputs:

```
displaying shape
moving game piece
```

Notice that the `PlayerPiece` class inherits the generic `displayShape()` method from the less-specialized class `GameShape` and also adds its own method, `movePiece()`. Code reuse through inheritance means that methods with generic functionality—such as `displayShape()`, which could apply to a wide range of different kinds of shapes in a game—don't have to be reimplemented. That means all specialized subclasses of `GameShape` are guaranteed to have the capabilities of the more general superclass. You don't want to have to rewrite the `displayShape()` code in each of your specialized components of an online game.

But you knew that. You've experienced the pain of duplicate code when you make a change in one place and have to track down all the other places where that same (or very similar) code exists.

The second (and related) use of inheritance is to allow your classes to be accessed polymorphically—a capability provided by interfaces as well, but we'll get to that in a minute. Let's say that you have a `GameLauncher` class that wants to loop through a list of different kinds of `GameShape` objects and invoke `displayShape()` on each of them. At the time you write this class, you don't know every possible kind of `GameShape` subclass that anyone else will ever write. And you sure don't want to have to redo *your* code just because somebody decided to build a dice shape six months later.

The beautiful thing about polymorphism ("many forms") is that you can treat any *subclass* of `GameShape` as a `GameShape`. In other words, you can write code in your `GameLauncher` class that says, "I don't care what kind of object you are as long as you inherit from (extend) `GameShape`. And as far as I'm concerned, if you extend `GameShape`, then you've definitely got a `displayShape()` method, so I know I can call it."

Imagine we now have two specialized subclasses that extend the more generic `GameShape` class, `PlayerPiece` and `TilePiece`:

```
class GameShape {
  public void displayShape() {
    System.out.println("displaying shape");
  }
  // more code
}

class PlayerPiece extends GameShape {
  public void movePiece() {
    System.out.println("moving game piece");
  }
  // more code
}

class TilePiece extends GameShape {
  public void getAdjacent() {
    System.out.println("getting adjacent tiles");
  }
  // more code
}
```

Now imagine a test class has a method with a declared argument type of `GameShape`, which means it can take any kind of `GameShape`. In other words, any subclass of `GameShape` can be passed to a method with an argument of type `GameShape`. This code

```
public class TestShapes {
  public static void main (String[] args) {
    PlayerPiece player = new PlayerPiece();
    TilePiece tile = new TilePiece();
    doShapes(player);
    doShapes(tile);
  }

  public static void doShapes(GameShape shape) {
    shape.displayShape();
 }
}
```

outputs:

```
displaying shape
displaying shape
```

The key point is that the `doShapes()` method is declared with a `GameShape` argument but can be passed any subtype (in this example, a subclass) of `GameShape`. The method can then invoke any method of `GameShape`, without any concern for the actual runtime class type of the object passed to the method. There are implications, though. The `doShapes()` method knows only that the objects are a type of `GameShape` since that's how the parameter is declared. And using a reference variable declared as type `GameShape`—regardless of whether the variable is a method parameter, local variable, or instance variable—means that *only* the methods of `GameShape` can be invoked on it. The methods you can call on a reference are totally dependent on the *declared* type of the variable, no matter what the actual object is, that the reference is referring to. That means you can't use a `GameShape` variable to call, say, the `getAdjacent()` method even if the object passed in *is* of type `TilePiece`. (We'll see this again when we look at interfaces.)

IS-A and HAS-A Relationships

Note: As of Winter 2017, the OCA 8 exam won't ask you **directly** about IS-A and HAS-A relationships. But understanding IS-A and HAS-A relationships will help OCA 8 candidates with many of the questions on the exam.

IS-A

In OO, the concept of IS-A is based on inheritance (or interface implementation). IS-A is a way of saying, "This thing is a type of that thing." For example, a Mustang is a type of Horse, so in OO terms we can say, "Mustang IS-A Horse." Subaru IS-A Car. Broccoli IS-A Vegetable (not a very fun one, but it still counts). You express the IS-A relationship in Java through the keywords `extends` (for *class* inheritance) and `implements` (for *interface* implementation).

```
public class Car {
  // Cool Car code goes here
}

public class Subaru extends Car {
```

```
    // Important Subaru-specific stuff goes here
    // Don't forget Subaru inherits accessible Car members which
    // can include both methods and variables.
}
```

A Car is a type of Vehicle, so the inheritance tree might start from the `Vehicle` class as follows:

```
public class Vehicle { ... }
public class Car extends Vehicle { ... }
public class Subaru extends Car { ... }
```

In OO terms, you can say the following:

`Vehicle` is the superclass of `Car`.
`Car` is the subclass of `Vehicle`.
`Car` is the superclass of `Subaru`.
`Subaru` is the subclass of `Vehicle`.
`Car` inherits from `Vehicle`.
`Subaru` inherits from both `Vehicle` and `Car`.
`Subaru` is derived from `Car`.
`Car` is derived from `Vehicle`.
`Subaru` is derived from `Vehicle`.
`Subaru` is a subtype of both `Vehicle` and `Car`.

Returning to our IS-A relationship, the following statements are true:

"`Car` extends `Vehicle`" means "`Car IS-A Vehicle.`"
"`Subaru` extends `Car`" means "`Subaru IS-A Car.`"

And we can also say:

"Subaru IS-A Vehicle"

because a class is said to be "a type of" anything further up in its inheritance tree. If the expression (`Foo instanceof Bar)` is `true`, then class `Foo` IS-A `Bar`, even if `Foo` doesn't directly extend `Bar`, but instead extends some other class that is a subclass of `Bar`. Figure 2-2 illustrates the inheritance tree for `Vehicle`, `Car`, and `Subaru`. The arrows move from the subclass to the superclass. In other words, a class's arrow points toward the class from which it extends.

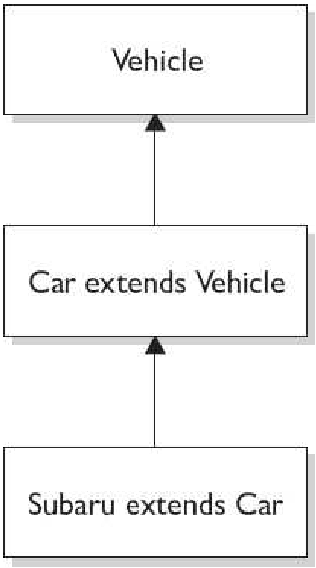

Figure 2-2: Inheritance tree for `Vehicle, Car, Subaru`

HAS-A

HAS-A relationships are based on use, rather than inheritance. In other words, class A HAS-A B if code in class A has a reference to an instance of class B. For example, you can say the following:

A `Horse` IS-A `Animal`. A `Horse` HAS-A `Halter`.

The code might look like this:

```
public class Animal { }
public class Horse extends Animal {
```

```
    private Halter myHalter;
  }
```

In this code, the `Horse` class has an instance variable of type `Halter` (a halter is a piece of gear you might have if you have a horse), so you can say that a "`Horse` HAS-A `Halter`." In other words, `Horse` has a reference to a `Halter`. `Horse` code can use that `Halter` reference to invoke methods on the `Halter` and get `Halter` behavior without having `Halter`-related code (methods) in the `Horse` class itself. Figure 2-3 illustrates the HAS-A relationship between `Horse` and `Halter`.

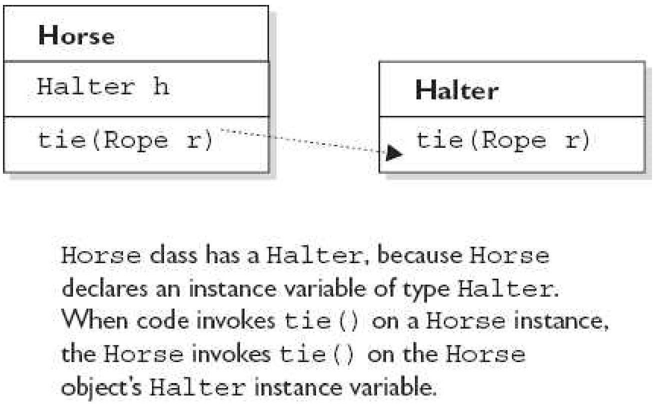

Figure 2-3: HAS-A relationship between `Horse` and `Halter`

HAS-A relationships allow you to design classes that follow good OO practices by not having monolithic classes that do a gazillion different things. Classes (and their resulting objects) should be specialists. As our friend Andrew says, "Specialized classes can actually help reduce bugs." The more specialized the class, the more likely it is that you can reuse the class in other applications. If you put all the `Halter`-related code directly into the `Horse` class, you'll end up duplicating code in the `Cow` class, `UnpaidIntern` class, and any other class that might need `Halter` behavior. By keeping the `Halter` code in a separate, specialized `Halter` class, you have the chance to reuse the `Halter` class in multiple applications.

Users of the `Horse` class (that is, code that calls methods on a `Horse` instance) think that the `Horse` class has `Halter` behavior. The `Horse` class might have a `tie(LeadRope rope)` method, for example. Users of the `Horse` class should never have to know that when they invoke the `tie()` method, the `Horse` object turns around and delegates the call to its `Halter` class by invoking `myHalter.tie(rope)`. The scenario just described might look like this:

```
public class Horse extends Animal {
  private Halter myHalter = new Halter();
  public void tie(LeadRope rope) {
    myHalter.tie(rope);  // Delegate tie behavior to the
                         // Halter object
   }
}
public class Halter {
  public void tie(LeadRope aRope) {
    // Do the actual tie work here
  }
}
```

In OO, we don't want callers to worry about which class or object is actually doing the real work. To make that happen, the `Horse` class hides implementation details from `Horse` users. `Horse` users ask the `Horse` object to do things (in this case, tie itself up), and the `Horse` will either do it or, as in this example, ask something else (like perhaps an inherited `Animal` class method) to do it. To the caller, though, it always appears that the `Horse` object takes care of itself. Users of a `Horse` should not even need to know that there is such a thing as a `Halter` class.

From the Classroom

Object-Oriented Design

IS-A and HAS-A relationships and encapsulation are just the tip of the iceberg when it comes to OO design. Many books and graduate theses have been dedicated to this topic. The reason for the emphasis on proper design is simple: money. The cost to deliver a software application has been estimated to be as much as ten times more expensive for poorly designed programs.

Even the best OO designers (often called "architects") make mistakes. It is difficult to visualize the relationships between hundreds, or even thousands, of classes. When mistakes are discovered during the implementation (code writing) phase of a project, the amount of code that must be rewritten can sometimes mean programming teams have to start over from scratch.

The software industry has evolved to aid the designer. Visual object modeling languages, such as the Unified Modeling Language (UML), allow designers to design and easily modify classes without having to write code first because OO components are represented graphically. This allows designers to create a map of the class relationships and helps them recognize errors before coding begins. Another innovation in OO design is design patterns. Designers noticed that many OO designs apply consistently from project to project and that it was useful to apply the same designs because it reduced the potential to introduce new design errors. OO designers then started to share these designs with each other. Now there are many catalogs of these design patterns both on the Internet and in book form.

Although passing the Java certification exam does not require you to understand OO design this thoroughly, hopefully this background information will help you better appreciate why the test writers chose to include encapsulation and IS-A and HAS-A relationships on the exam.

—Jonathan Meeks,
Sun Certified Java Programmer

CERTIFICATION OBJECTIVE: POLYMORPHISM (OCA OBJECTIVE 7.2)

7.2 Develop code that demonstrates the use of polymorphism; including overriding and object type versus reference type (sic).

Remember that any Java object that can pass more than one IS-A test can be considered polymorphic. Other than objects of type `Object`, *all* Java objects are polymorphic in that they pass the IS-A test for their own type and for class `Object`.

Remember, too, that the only way to access an object is through a reference variable. There are a few key things you should know about references:

- A reference variable can be of only one type, and once declared, that type can never be changed (although the object it references can change).
- A reference is a variable, so it can be reassigned to other objects (unless the reference is declared `final`).
- A reference variable's type determines the methods that can be invoked on the object the variable is referencing.
- A reference variable can refer to any object of the same type as the declared reference, or—this is the big one—**it can refer to any *subtype* of the declared type!**
- A reference variable can be declared as a class type or an interface type. If the variable is declared as an interface type, it can reference any object of any class that *implements* the interface.

Earlier we created a `GameShape` class that was extended by two other classes, `PlayerPiece` and `TilePiece`. Now imagine you want to animate some of the shapes on the gameboard. But not *all* shapes are able to be animated, so what do you do with class inheritance?

Could we create a class with an `animate()` method and have only *some* of the `GameShape` subclasses inherit from that class? If we can, then we could have `PlayerPiece`, for example, extend *both* the `GameShape` class and `Animatable` class, whereas the `TilePiece` would extend only `GameShape`. But no, this won't work! Java supports only single class inheritance! That means a class can have only one immediate superclass. In other words, if `PlayerPiece` is a class, there is no way to say something like this:

```
class PlayerPiece extends GameShape, Animatable { // NO!
  // more code
}
```

A *class* cannot *extend* more than one class: that means one parent per class. A class *can* have multiple ancestors, however, since class B could extend class A, and class C could extend class B, and so on. So any given class might have multiple classes up its inheritance tree, but that's not the same as saying a class directly extends two classes.

on the job Some languages (such as C++) allow a class to extend more than one other class. This capability is known as "multiple inheritance." The reason that Java's creators chose not to allow multiple class inheritance is that it can become quite messy. In a nutshell, the problem is that if a class extended two other classes, and both superclasses had, say, a `doStuff()` method, which version of `doStuff()` would the subclass inherit? This issue can lead to a scenario known as the "Deadly Diamond of Death," because of the shape of the class diagram that can be created in a multiple inheritance design. The diamond is formed when classes B and C both extend A, and both B and C inherit a method from A. If class D extends both B and C, and both B and C have overridden the method in A, class D has, in theory, inherited two different implementations of the same method. Drawn as a class diagram, the shape of the four classes looks like a diamond.

So if that doesn't work, what else could you do? You could simply put the `animate()` code in `GameShape`, and then disable the method in classes that can't be animated. But that's a bad design choice for many reasons—it's more error prone; it makes the `GameShape` class less cohesive; and it means the `GameShape` API "advertises" that all shapes can be animated when, in fact, that's not true since only some of the `GameShape` subclasses will be able to run the `animate()` method successfully.

Exam Watch

To reiterate, as of Java 8, interfaces can have concrete methods (called `default` methods). This allows for a form of multiple interhance, which we'll discuss later in the chapter.

So what *else* could you do? You already know the answer—create an `Animatable` *interface*, and have only the `GameShape` subclasses that can be animated implement that interface. Here's the interface:

```
public interface Animatable {
  public void animate();
}
```

And here's the modified `PlayerPiece` class that implements the interface:

```
class PlayerPiece extends GameShape implements Animatable {
  public void movePiece() {
    System.out.println("moving game piece");
  }
  public void animate() {
    System.out.println("animating...");
  }
  // more code
}
```

So now we have a `PlayerPiece` that passes the IS-A test for both the `GameShape` class and the `Animatable` interface. That means a `PlayerPiece` can be treated polymorphically as one of four things at any given time, depending on the declared type of the reference variable:

- An `Object` (since any object inherits from `Object`)
- A `GameShape` (since `PlayerPiece` extends `GameShape`)
- A `PlayerPiece` (since that's what it really is)
- An `Animatable` (since `PlayerPiece` implements `Animatable`)

The following are all legal declarations. Look closely:

```
PlayerPiece player = new PlayerPiece();
Object o = player;
GameShape shape = player;
Animatable mover = player;
```

There's only one object here—an instance of type `PlayerPiece`—but there are four different types of reference variables, all referring to that one object on the heap. Pop quiz: Which of the preceding reference variables can invoke the `displayShape()` method? Hint: Only two of the four declarations can be used to invoke the `displayShape()` method.

Remember that method invocations allowed by the compiler are based solely on the declared type of the reference, regardless of the object type. So looking at the four reference types again—`Object`, `GameShape`, `PlayerPiece`, and `Animatable`—which of these four types know about the `displayShape()` method?

You guessed it—both the `GameShape` class and the `PlayerPiece` class are known (by the compiler) to have a `displayShape()` method, so either of those reference types can be used to invoke `displayShape()`. Remember that to the compiler, a `PlayerPiece` IS-A `GameShape`, so the compiler says, "I see that the declared type is `PlayerPiece`, and since `PlayerPiece` extends `GameShape`, that means `PlayerPiece` inherited the `displayShape()` method. Therefore, `PlayerPiece` can be used to invoke the `displayShape()` method."

Which methods can be invoked when the `PlayerPiece` object is being referred to using a reference declared as type `Animatable`? Only the `animate()` method. Of course, the cool thing here is that any class from any inheritance tree can also implement `Animatable`, so that means if you have a method with an argument declared as type `Animatable`, you can pass in `PlayerPiece` objects, `SpinningLogo` objects, and anything else that's an instance of a class that implements `Animatable`. And you can use that parameter (of type `Animatable`) to invoke the `animate()` method, but not the `displayShape()` method (which it might not even have), or anything other than what is known to the compiler based on the reference type. The compiler always knows, though, that you can invoke the methods of class `Object` on any object, so those are safe to call regardless of the reference—class or interface—used to refer to the object.

We've left out one big part of all this, which is that even though the compiler only knows about the declared reference type, the Java Virtual Machine (JVM) at runtime knows what the object really is. And that means that even if the `PlayerPiece` object's `displayShape()` method is called using a `GameShape` reference variable, if the `PlayerPiece` overrides the `displayShape()` method, the JVM will invoke the `PlayerPiece` version! The JVM looks at the real object at the other end of the reference, "sees" that it has overridden the method of the declared reference variable type, and invokes the method of the object's actual class. But there is one other thing to keep in mind:

> **Polymorphic method invocations apply only to *instance methods*. You can always refer to an object with a more general reference variable type (a superclass or interface), but at runtime, the ONLY things that are dynamically selected based on the actual *object* (rather than the *reference* type) are instance methods. Not *static* methods. Not *variables*. Only overridden instance methods are dynamically invoked based on the real object's type**.

Because this definition depends on a clear understanding of overriding and the distinction between static methods and instance methods, we'll cover those later in the chapter.

CERTIFICATION OBJECTIVE: OVERRIDING/OVERLOADING (OCA OBJECTIVES 6.1 AND 7.2)

6.1 Create methods with arguments and return values; including overloaded methods.

7.2 Develop code that demonstrates the use of polymorphism; including overriding and object type versus reference type (sic).

The exam will use overridden and overloaded methods on many, many questions. These two concepts are often confused (perhaps because they have similar names?), but each has its own unique and complex set of rules. It's important to get really clear about which "over" uses which rules!

Overridden Methods

Any time a type inherits a method from a supertype, you have the opportunity to override the method (unless, as you learned earlier, the method is marked `final`). The key benefit of overriding is the ability to define behavior that's specific to a particular subtype. The following example demonstrates a `Horse` subclass of `Animal` overriding the `Animal` version of the `eat()` method:

```
public class Animal {
  public void eat() {
    System.out.println("Generic Animal Eating Generically");
  }
}
class Horse extends Animal {
  public void eat() {
    System.out.println("Horse eating hay, oats, "
                        + "and horse treats");
  }
}
```

For abstract methods you inherit from a supertype, you have no choice: You *must* implement the method in the subtype ***unless the subtype is also abstract*.** Abstract methods must be *implemented* by the first concrete subclass, but this is a lot like saying the concrete subclass *overrides* the abstract methods of the supertype(s). So you could think of abstract methods as methods you're forced to override—eventually.

The `Animal` class creator might have decided that for the purposes of polymorphism, all `Animal` subtypes should have an `eat()` method defined in a unique way. Polymorphically, when an `Animal` reference refers not to an `Animal` instance, but to an `Animal` subclass instance, the caller should be able to invoke `eat()` on the `Animal` reference, but the actual runtime object (say, a `Horse` instance) will run its own specific `eat()` method. Marking the `eat()` method abstract is the `Animal` programmer's way of saying to all subclass developers, "It doesn't make any sense for your new subtype to use a generic `eat()` method, so you have to come up with your *own* `eat()` method implementation!" A (nonabstract) example of using polymorphism looks like this:

```
public class TestAnimals {
  public static void main (String [] args) {
    Animal a = new Animal();
    Animal b = new Horse(); // Animal ref, but a Horse object
    a.eat(); // Runs the Animal version of eat()
    b.eat(); // Runs the Horse version of eat()
  }
}
class Animal {
  public void eat() {
    System.out.println("Generic Animal Eating Generically");
  }
}
class Horse extends Animal {
  public void eat() {
    System.out.println("Horse eating hay, oats, "
                       + "and horse treats");
   }
   public void buck() { }
}
```

In the preceding code, the test class uses an `Animal` reference to invoke a method on a `Horse` object. Remember, the compiler will allow only methods in class `Animal` to be invoked when using a reference to an `Animal`. The following would not be legal given the preceding code:

```
Animal c = new Horse();
c.buck(); // Can't invoke buck();
          // Animal class doesn't have that method
```

To reiterate, the compiler looks only at the reference type, not the instance type. Polymorphism lets you use a more abstract supertype (including an interface) reference to one of its subtypes (including interface implementers).

The overriding method cannot have a more restrictive access modifier than the method being overridden (for example, you can't override a method marked `public` and make it `protected`). Think about it: If the `Animal` class advertises a `public eat()` method and someone has an `Animal` reference (in other words, a reference declared as type `Animal`), that someone will assume it's safe to call `eat()` on the `Animal` reference regardless of the actual instance that the `Animal` reference is referring to. If a subtype were allowed to sneak in and change the access modifier on the overriding method, then suddenly at runtime—when the JVM invokes the true object's (`Horse`) version of the method rather than the reference type's (`Animal`) version—the program would die a horrible death. (Not to mention the emotional distress for the one who was betrayed by the rogue subtype.)

Let's modify the polymorphic example we saw earlier in this section:

```
public class TestAnimals {
  public static void main (String [] args) {
    Animal a = new Animal();
    Animal b = new Horse();  // Animal ref, but a Horse object
    a.eat();                 // Runs the Animal version of eat()
    b.eat();                 // Runs the Horse version of eat()
  }
}
class Animal {
  public void eat() {
    System.out.println("Generic Animal Eating Generically");
  }
}
class Horse extends Animal {
  private void eat() {      // whoa! - it's private!
    System.out.println("Horse eating hay, oats, "
                       + "and horse treats");
  }
}
```

If this code compiled (which it doesn't), the following would fail at runtime:

```
Animal b = new Horse();  // Animal ref, but a Horse
                         // object, so far so good
b.eat();                 // Meltdown at runtime!
```

The variable b is of type `Animal`, which has a `public eat()` method. But remember that at runtime, Java uses virtual method invocation to dynamically select the actual version of the method that will run, based on the actual instance. An `Animal` reference can always refer to a `Horse` instance, because `Horse` IS-A(n) `Animal`. What makes that supertype reference to a subtype instance possible is that the subtype is guaranteed to be able to do everything the supertype can do. Whether the `Horse` instance overrides the inherited methods of `Animal` or simply inherits them, anyone with an `Animal` reference to a `Horse` instance is free to call all accessible `Animal` methods. For that reason, an overriding method must fulfill the contract of the superclass.

Note: In Chapter 5 we will explore exception handling in detail. Once you've studied Chapter 5, you'll appreciate this single handy list of overriding rules. The rules for overriding a method are as follows:

- The argument list must exactly match that of the overridden method. If they don't match, you can end up with an overloaded method you didn't intend.
- The return type must be the same as, or a subtype of, the return type declared in the original overridden method in the superclass. (More on this in a few pages when we discuss covariant returns.)
- The access level can't be more restrictive than that of the overridden method.
- The access level CAN be less restrictive than that of the overridden method.
- Instance methods can be overridden only if they are inherited by the subtype. A subtype within the same package as the instance's supertype can override any supertype method that is not marked `private` or `final`. A subtype in a different package can override only those non`final` methods marked `public` or `protected` (since `protected` methods are inherited by the subtype).
- The overriding method CAN throw any unchecked (runtime) exception, regardless of whether the overridden method declares the exception. (More in Chapter 5.)
- The overriding method must NOT throw checked exceptions that are new or broader than those declared by the overridden method. For example, a method that declares a `FileNotFoundException` cannot be overridden by a method that declares a `SQLException`, `Exception`, or any other nonruntime exception unless it's a subclass of `FileNotFoundException`.
- The overriding method can throw narrower or fewer exceptions. Just because an overridden method "takes risks" doesn't mean that the overriding subtype's exception takes the same risks. Bottom line: an overriding method doesn't have to declare any exceptions that it will

never throw, regardless of what the overridden method declares.

- You cannot override a method marked `final`.
- You cannot override a method marked `static`. We'll look at an example in a few pages when we discuss `static` methods in more detail.
- If a method can't be inherited, you cannot override it. Remember that overriding implies that you're reimplementing a method you inherited! For example, the following code is not legal, and even if you added an `eat()` method to `Horse`, it wouldn't be an override of `Animal`'s `eat()` method:

```
public class TestAnimals {
  public static void main (String [] args) {
    Horse h =  new Horse();
    h.eat(); // Not legal because Horse didn't inherit eat()
  }
}
class Animal {
  private void eat() {
    System.out.println("Generic Animal Eating Generically");
  }
}
class Horse extends Animal { }
```

Invoking a Supertype Version of an Overridden Method

Often, you'll want to take advantage of some of the code in the supertype version of a method, yet still override it to provide some additional specific behavior. It's like saying, "Run the supertype version of the method, and then come back down here and finish with my subtype additional method code." (Note that there's no requirement that the supertype version run before the subtype code.) It's easy to do in code using the keyword `super` as follows:

```
public class Animal {
  public void eat() { }
  public void printYourself() {
    // Useful printing code goes here
  }
}
class Horse extends Animal {
  public void printYourself() {
    // Take advantage of Animal code, then add some more
    super.printYourself();  // Invoke the superclass
                            // (Animal) code
                            // Then do Horse-specific
                            // print work here
  }
}
```

In a similar way, you can access an interface's overridden method with the syntax:

```
InterfaceX.super.doStuff();
```

Note: Using `super` to invoke an overridden method applies only to instance methods. (Remember that `static` methods can't be overridden.) And you can use `super` only to access a method in a type's supertype, not the supertype of the supertype—that is, you **cannot** say `super.super.doStuff()` and you **cannot** say: InterfaceX.`super.super.doStuff()`.

Exam Watch

If a method is overridden but you use a polymorphic (supertype) reference to refer to the subtype object with the overriding method, the compiler assumes you're calling the supertype version of the method. If the supertype version declares a checked exception, but the overriding subtype method does not, the compiler still thinks you are calling a method that declares an exception (more in Chapter 5). Let's look at an example:

```
class Animal {
  public void eat() throws Exception {
                       // throws an Exception
  }
}
class Dog2 extends Animal {
  public void eat() { /* no Exceptions */}
  public static void main(String [] args) {
    Animal a = new Dog2();
    Dog2 d = new Dog2();
```

```
      d.eat();            // ok
      a.eat();            // compiler error -
                          // unreported exception
    }
  }
```

This code will not compile because of the exception declared on the `Animal eat()` method. This happens even though, at runtime, the `eat()` method used would be the `Dog` version, which does not declare the exception.

Examples of Illegal Method Overrides

Let's take a look at overriding the `eat()` method of `Animal`:

```
public class Animal {
  public void eat() { }
}
```

Table 2-2 lists examples of illegal overrides of the `Animal eat()` method, given the preceding version of the `Animal` class.

Table 2-2: Examples of Illegal Overrides

Illegal Override Code	Problem with the Code
`private void eat() { }`	Access modifier more restrictive
`public void eat() throws IOException { }`	Declares a checked exception not defined by superclass version
`public void eat(String food) { }`	A legal overload, not an override, because the argument list changed
`public String eat() { }`	Not an override because of the return type, and not an overload either because there's no change in the argument list

Overloaded Methods

Over*loaded* methods let you reuse the same method name in a class, but with different arguments (and, optionally, a different return type). Overloading a method often means you're being a little nicer to those who call your methods because your code takes on the burden of coping with different argument types rather than forcing the caller to do conversions prior to invoking your method. The rules aren't too complex:

- Overloaded methods MUST change the argument list.
- Overloaded methods CAN change the return type.
- Overloaded methods CAN change the access modifier.
- Overloaded methods CAN declare new or broader checked exceptions.
- A method can be overloaded in the *same* type or in a *subtype*. In other words, if class A defines a `doStuff(int i)` method, the subclass B could define a `doStuff(String s)` method without overriding the superclass version that takes an `int`. So two methods with the same name but in different types can still be considered overloaded if the subtype inherits one version of the method and then declares another overloaded version in its type definition.

Exam Watch

Less experienced Java developers are often confused about the subtle differences between overloaded and overridden methods. Be careful to recognize when a method is overloaded rather than overridden. You might see a method that appears to be violating a rule for overriding, but that is actually a legal overload, as follows:

```
public class Foo {
  public void doStuff(int y, String s) { }
  public void moreThings(int x) { }
}
class Bar extends Foo {
  public void doStuff(int y, long s) throws IOException { }
}
```

It's tempting to see the `IOException` as the problem because the overridden `doStuff()` method doesn't declare an exception and `IOException` is checked by the compiler. But the `doStuff()` method is not overridden! Subclass Bar overloads the `doStuff()` method by varying the argument list, so the `IOException` is fine.

Legal Overloads

Let's look at a method we want to overload:

```
public void changeSize(int size, String name, float pattern) { }
```

The following methods are legal overloads of the `changeSize()` method:

```
public void changeSize(int size, String name) { }
private int changeSize(int size, float pattern) { }
public void changeSize(float pattern, String name)
                       throws IOException { }
```

Invoking Overloaded Methods

In Chapter 6 we will look at how boxing and var-args impact overloading. (You still have to pay attention to what's covered here, however.)

When a method is invoked, more than one method of the same name might exist for the object type you're invoking a method on. For example, the `Horse` class might have three methods with the same name but with different argument lists, which means the method is overloaded.

Deciding which of the matching methods to invoke is based on the arguments. If you invoke the method with a `String` argument, the overloaded version that takes a `String` is called. If you invoke a method of the same name but pass it a `float`, the overloaded version that takes a `float` will run. If you invoke the method of the same name but pass it a `Foo` object, and there isn't an overloaded version that takes a `Foo`, then the compiler will complain that it can't find a match. The following are examples of invoking overloaded methods:

```
class Adder {
  public int addThem(int x, int y) {
    return x + y;
  }

  // Overload the addThem method to add doubles instead of ints
  public double addThem(double x, double y) {
    return x + y;
  }
}
  // From another class, invoke the addThem() method
public class TestAdder {
  public static void main (String [] args) {
    Adder a = new Adder();
    int b = 27;
    int c = 3;
    int result = a.addThem(b,c);               // Which addThem is invoked?
    double doubleResult = a.addThem(22.5,9.3); // Which addThem?
  }
}
```

In this `TestAdder` code, the first call to `a.addThem(b,c)` passes two `int`s to the method, so the first version of `addThem()`—the overloaded version that takes two `int` arguments—is called. The second call to `a.addThem(22.5, 9.3)` passes two `double`s to the method, so the second version of `addThem()`—the overloaded version that takes two double arguments—is called.

Invoking overloaded methods that take object references rather than primitives is a little more interesting. Say you have an overloaded method such that one version takes an `Animal` and one takes a `Horse` (subclass of `Animal`). If you pass a `Horse` object in the method invocation, you'll invoke the overloaded version that takes a `Horse`. Or so it looks at first glance:

```
class Animal { }
class Horse extends Animal { }
class UseAnimals {
  public void doStuff(Animal a) {
    System.out.println("In the Animal version");
  }
  public void doStuff(Horse h) {
    System.out.println("In the Horse version");
  }
  public static void main (String [] args) {
    UseAnimals ua = new UseAnimals();
    Animal animalObj = new Animal();
    Horse horseObj = new Horse();
    ua.doStuff(animalObj);
    ua.doStuff(horseObj);
  }
}
```

The output is what you expect:

```
In the Animal version
```

```
In the Horse version
```

But what if you use an `Animal` reference to a `Horse` object?

```
Animal animalRefToHorse = new Horse();
  ua.doStuff(animalRefToHorse);
```

Which of the overloaded versions is invoked? You might want to answer, "The one that takes a `Horse` since it's a `Horse` object at runtime that's being passed to the method." But that's not how it works. The preceding code would actually print this:

```
in the Animal version
```

Even though the actual object at runtime is a `Horse` and not an `Animal`, the choice of which overloaded method to call (in other words, the signature of the method) is NOT dynamically decided at runtime.

Just remember that the *reference* type (not the object type) determines which overloaded method is invoked!

To summarize, which over*ridden* version of the method to call (in other words, from which class in the inheritance tree) is decided at *runtime* based on *object* type, but which over*loaded* version of the method to call is based on the *reference* type of the argument passed at *compile* time.

If you invoke a method passing it an `Animal` reference to a `Horse` object, the compiler knows only about the `Animal`, so it chooses the overloaded version of the method that takes an `Animal`. It does not matter that, at runtime, a `Horse` is actually being passed.

Exam Watch

Can `main()` be overloaded?

```
class DuoMain {
  public static void main(String[] args) {
    main(1);
  }
  static void main(int i) {
    System.out.println("overloaded main");
  }
}
```

Absolutely! But the only `main()` with JVM superpowers is the one with the signature you've seen about 100 times already in this book.

Polymorphism in Overloaded and Overridden Methods How does polymorphism work with overloaded methods? From what we just looked at, it doesn't appear that polymorphism matters when a method is overloaded. If you pass an `Animal` reference, the overloaded method that takes an `Animal` will be invoked, even if the actual object passed is a `Horse`. Once the `Horse` masquerading as `Animal` gets in to the method, however, the `Horse` object is still a `Horse` despite being passed into a method expecting an `Animal`. So it's true that polymorphism doesn't determine which overloaded version is called; however, polymorphism does come into play when the decision is about which overridden version of a method is called. But sometimes a method is both overloaded and overridden. Imagine that the `Animal` and `Horse` classes look like this:

```
public class Animal {
  public void eat() {
    System.out.println("Generic Animal Eating Generically");
   }
}
public class Horse extends Animal {
  public void eat() {
    System.out.println("Horse eating hay ");
  }
  public void eat(String s) {
    System.out.println("Horse eating " + s);
  }
}
```

Notice that the `Horse` class has both overloaded and overridden the `eat()` method. Table 2-3 shows which version of the three `eat()` methods will run depending on how they are invoked.

Table 2-3: Examples of Legal and Illegal Overrides

Method Invocation Code	Result
`Animal a = new Animal(); a.eat();`	`Generic Animal Eating Generically`
`Horse h = new Horse`	`Horse eating hay`

`();` `h.eat();`	
`Animal ah = new` **`Horse();`** `ah.eat();`	`Horse eating hay` Polymorphism works—the actual object type (`Horse`), not the reference type (`Animal`), is used to determine which `eat()` is called.
`Horse he = new Horse();` `he.eat("Apples");`	`Horse eating Apples` The overloaded `eat(String s)` method is invoked.
`Animal a2 = new Animal();` `a2.eat("treats");`	Compiler error! Compiler sees that the `Animal` class doesn't have an `eat()` method that takes a `String`.
`Animal ah2 = new` **`Horse();`** `ah2.eat("Carrots");`	Compiler error! Compiler still looks only at the reference and sees that `Animal` doesn't have an `eat()` method that takes a `String`. Compiler doesn't care that the actual object might be a `Horse` at runtime.

Exam Watch

Don't be fooled by a method that's overloaded but not overridden by a subclass. It's perfectly legal to do the following:

```
public class Foo {
  void doStuff() { }
}
class Bar extends Foo {
  void doStuff(String s) { }
}
```

The `Bar` class has two `doStuff()` methods: the no-arg version it inherits from `Foo` (and does not override) and the overloaded `doStuff(String s)` defined in the `Bar` class. Code with a reference to a `Foo` can invoke only the no-arg version, but code with a reference to a `Bar` can invoke either of the overloaded versions.

Table 2-4 summarizes the difference between overloaded and overridden methods.

Table 2-4: Differences Between Overloaded and Overridden Methods

	Overloaded Method	Overridden Method
Argument (s)	Must change.	Must not change.
Return type	Can change.	Can't change except for covariant returns. (Covered later this chapter.)
Exceptions	Can change.	Can reduce or eliminate. Must not throw new or broader checked exceptions.
Access	Can change.	Must not make more restrictive (can be less restrictive).
Invocation	*Reference* type determines which overloaded version (based on declared argument types) is selected. Happens at *compile* time. The actual *method* that's invoked is still a virtual method invocation that happens at runtime, but the compiler will already know the *signature* of the method to be invoked. So at runtime, the argument match will already have been nailed down, just not the *class* in which the method lives.	*Object* type (in other words, *the type of the actual instance on the heap*) determines which method is selected. Happens at *runtime*.

We'll cover constructor overloading later in the chapter, where we'll also cover the other constructor-related topics that are on the exam. Figure 2-4 illustrates the way overloaded and overridden methods appear in class relationships.

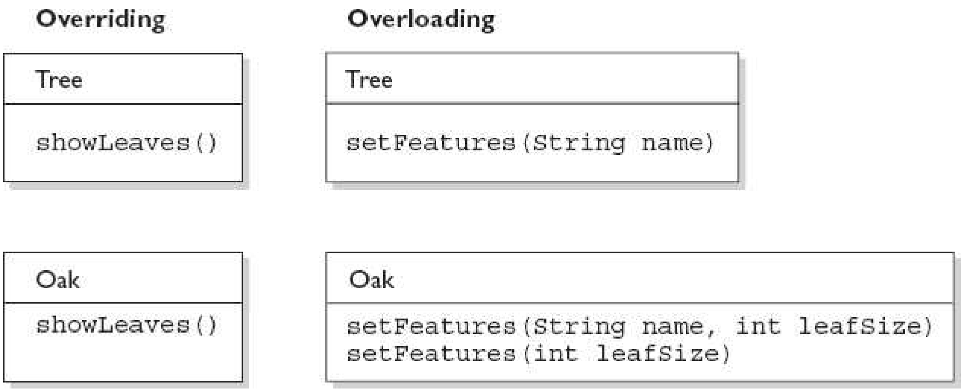

Figure 2-4: Overloaded and overridden methods in class relationships

CERTIFICATION OBJECTIVE: CASTING (OCA OBJECTIVES 2.2 AND 7.3)

2.2 Differentiate between object reference variables and primitive variables.

7.3 Determine when casting is necessary.

You've seen how it's both possible and common to use general reference variable types to refer to more specific object types. It's at the heart of polymorphism. For example, this line of code should be second nature by now:

```
Animal animal = new Dog();
```

But what happens when you want to use that `animal` reference variable to invoke a method that only class `Dog` has? You know it's referring to a `Dog`, and you want to do a `Dog`-specific thing? In the following code, we've got an array of `Animal`s, and whenever we find a `Dog` in the array, we want to do a special `Dog` thing. Let's agree for now that all this code is okay, except that we're not sure about the line of code that invokes the `playDead` method.

```
class Animal {
  void makeNoise() {System.out.println("generic noise"); }
}
class Dog extends Animal {
  void makeNoise() {System.out.println("bark"); }
  void playDead() { System.out.println("roll over"); }
}

class CastTest2 {
  public static void main(String [] args) {
    Animal [] a = {new Animal(), new Dog(), new Animal() };
    for(Animal animal : a) {
      animal.makeNoise();
      if(animal instanceof Dog) {
        animal.playDead();       // try to do a Dog behavior?
      }
    }
  }
}
```

When we try to compile this code, the compiler says something like this:

```
cannot find symbol
```

The compiler is saying, "Hey, class `Animal` doesn't have a `playDead()` method." Let's modify the `if` code block:

```
if(animal instanceof Dog) {
  Dog d = (Dog) animal;      // casting the ref. var.
  d.playDead();
}
```

The new and improved code block contains a cast, which in this case is sometimes called a *downcast*, because we're casting down the inheritance tree to a more specific class. Now the compiler is happy. Before we try to invoke `playDead`, we cast the `animal` variable to type `Dog`. What we're saying to the compiler is, "We know it's really referring to a `Dog` object, so it's okay to make a new `Dog` reference variable to refer to that object." In this case we're safe, because before we ever try the cast, we do an `instanceof` test to make sure.

It's important to know that the compiler is forced to trust us when we do a downcast, even when we screw up:

```
class Animal { }
class Dog extends Animal { }
class DogTest {
```

```
    public static void main(String [] args) {
      Animal animal = new Animal();
      Dog d = (Dog) animal;          // compiles but fails later
    }
  }
```

It can be maddening! This code compiles! But when we try to run it, we'll get an exception, something like this:

```
java.lang.ClassCastException
```

Why can't we trust the compiler to help us out here? Can't it see that `animal` is of type `Animal`? All the compiler can do is verify that the two types are in the same inheritance tree, so that depending on whatever code might have come before the downcast, it's possible that `animal` is of type `Dog`. The compiler must allow things that might possibly work at runtime. However, if the compiler knows with certainty that the cast could not possibly work, compilation will fail. The following replacement code block will NOT compile:

```
Animal animal = new Animal();
Dog d = (Dog) animal;
String s = (String) animal;    // animal can't EVER be a String
```

In this case, you'll get an error like this:

```
inconvertible types
```

Unlike downcasting, *upcasting* (casting *up* the inheritance tree to a more general type) works implicitly (that is, you don't have to type in the cast) because when you upcast you're implicitly restricting the number of methods you can invoke, as opposed to *down*casting, which implies that later on, you might want to invoke a more *specific* method. Here's an example:

```
class Animal { }
class Dog extends Animal { }

class DogTest {
  public static void main(String [] args) {
    Dog d = new Dog();
    Animal a1 = d;           // upcast ok with no explicit cast
    Animal a2 = (Animal) d;  // upcast ok with an explicit cast
  }
}
```

Both of the previous upcasts will compile and run without exception because a `Dog` IS-A(n) `Animal`, which means that anything an `Animal` can do, a `Dog` can do. A `Dog` can do more, of course, but the point is that anyone with an `Animal` reference can safely call `Animal` methods on a `Dog` instance. The `Animal` methods may have been overridden in the `Dog` class, but all we care about now is that a `Dog` can always do at least everything an `Animal` can do. The compiler and JVM know it, too, so the implicit upcast is always legal for assigning an object of a subtype to a reference of one of its supertype classes (or interfaces). If `Dog` implements `Pet` and `Pet` defines `beFriendly()`, then a `Dog` can be implicitly cast to a `Pet`, but the only `Dog` method you can invoke then is `beFriendly()`, which `Dog` was forced to implement because `Dog` implements the `Pet` interface.

One more thing…if `Dog` implements `Pet`, then, if `Beagle` extends `Dog` but `Beagle` does not *declare* that it implements `Pet`, `Beagle` is still a `Pet`! `Beagle` is a `Pet` simply because it extends `Dog`, and `Dog`'s already taken care of the `Pet` parts for itself and for all its children. The `Beagle` class can always override any method it inherits from `Dog`, including methods that `Dog` implemented to fulfill its interface contract.

And just one more thing…if `Beagle` does declare that it implements `Pet`, just so that others looking at the `Beagle` class API can easily see that `Beagle` IS-A `Pet` without having to look at `Beagle`'s superclasses, `Beagle` still doesn't need to implement the `beFriendly()` method if the `Dog` class (`Beagle`'s superclass) has already taken care of that. In other words, if `Beagle` IS-A `Dog`, and `Dog` IS-A `Pet`, then `Beagle` IS-A `Pet` and has already met its `Pet` obligations for implementing the `beFriendly()` method since it inherits the `beFriendly()` method. The compiler is smart enough to say, "I know `Beagle` already IS a `Dog`, but it's okay to make it more obvious by adding a cast."

So don't be fooled by code that shows a concrete class that declares it implements an interface but doesn't implement the *methods* of the interface. Before you can tell whether the code is legal, you must know what the supertypes of this implementing class have declared. If any supertype in its inheritance tree has already provided concrete (that is, nonabstract) method implementations, then regardless of whether the supertype declares that it implements the interface, the subclass is under no obligation to reimplement (override) those methods.

Exam Watch

The exam creators will tell you that they're forced to jam tons of code into little spaces "because of the exam engine." Although that's partially true, they ALSO like to obfuscate. The following code

```
Animal a = new Dog();
Dog d = (Dog) a;
d.doDogStuff();
```

can be replaced with this easy-to-read bit of fun:

```
Animal a = new Dog();
```

```
((Dog)a).doDogStuff();
```

In this case the compiler needs all those parentheses; otherwise, it thinks it's been handed an incomplete statement.

CERTIFICATION OBJECTIVE: IMPLEMENTING AN INTERFACE (OCA OBJECTIVE 7.5)

7.5 Use abstract classes and interfaces.

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

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

Assuming an interface `Bounceable` with two methods, `bounce()` and `setBounceFactor()`, the following class will compile:

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

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

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

- Provide concrete (nonabstract) implementations for all abstract methods from the declared interface.
- Follow all the rules for legal overrides, such as the following:
  - Declare no checked exceptions on implementation methods other than those declared by the interface method or subclasses of those declared by the interface method.
  - Maintain the signature of the interface method, and maintain the same return type (or a subtype). (But it does not have to declare the exceptions declared in the interface method declaration.)

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

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

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

Exam Watch

Implementation classes are NOT required to implement an interface's static or default methods. We'll discuss this in more depth later in the chapter.

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

  public void bounce() {
  // interesting BeachBall-specific bounce code
```

```
    }
    public void setBounceFactor(int bf) {
    // clever BeachBall-specific code for setting
    // a bounce factor

    }
    // if class Ball defined any abstract methods,
    // they'll have to be
    // implemented here as well.
  }
```

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

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

1. A class can implement more than one interface. It's perfectly legal to say, for example, the following:

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

   You can extend only one class, but you can implement many interfaces (which, as of Java 8, means a form of multiple inheritance, which we'll discuss shortly). In other words, subclassing defines who and what you are, whereas implementing defines a role you can play or a hat you can wear, despite how different you might be from some other class implementing the same interface (but from a different inheritance tree). For example, a Person extends HumanBeing (although for some, that's debatable). But a Person may also implement Programmer, Snowboarder, Employee, Parent, or PersonCrazyEnoughToTakeThisExam.

2. An interface can itself extend another interface. The following code is perfectly legal:

   ```
   public interface Bounceable extends Moveable { }   // ok!
   ```

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

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

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

As we mentioned earlier, a class is not allowed to extend multiple classes in Java. An interface, however, is free to extend multiple interfaces:

```
interface Bounceable extends Moveable, Spherical {  // ok!
  void bounce();
  void setBounceFactor(int bf);
}
interface Moveable {
  void moveIt();
}
interface Spherical {
  void doSphericalThing();
}
```

In the next example, `Ball` is required to implement `Bounceable`, plus all abstract methods from the interfaces that `Bounceable` extends (including any interfaces those interfaces extend, and so on, until you reach the top of the stack—or is it the bottom of the stack?). So `Ball` would need to look like the following:

```
class Ball implements Bounceable {

  public void bounce() { }             // Implement Bounceable's methods
  public void setBounceFactor(int bf) { }

  public void moveIt() { }            // Implement Moveable's method

  public void doSphericalThing() { }  // Implement Spherical
}
```

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

```
abstract class Ball implements Bounceable {
  public void bounce() { ... }   // Define bounce behavior
  public void setBounceFactor(int bf) { ... }
```

```
    // Don't implement the rest; leave it for a subclass
  }
  class SoccerBall extends Ball {  // class SoccerBall must
                                   // implement the interface
                                   // methods that Ball didn't
    public void moveIt() { ... }
    public void doSphericalThing() { ... }
    // SoccerBall can choose to override the Bounceable methods
    // implemented by Ball
    public void bounce() { ... }
  }
```

Figure 2-5 compares concrete and `abstract` examples of extends and implements for both classes and interfaces.

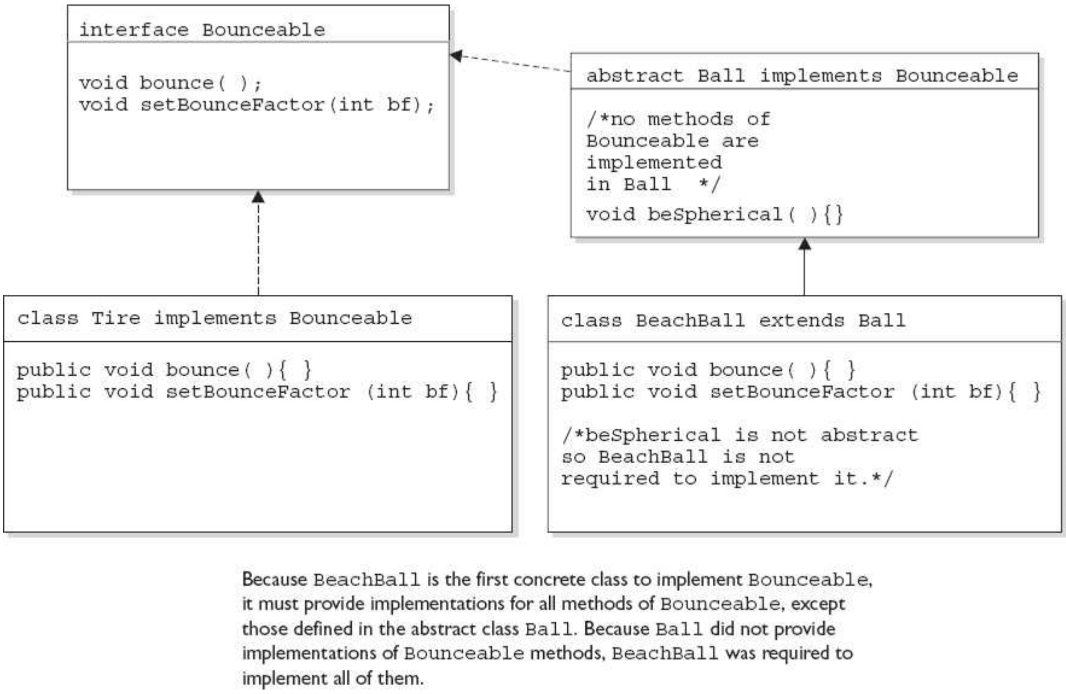

Figure 2-5: Comparing concrete and `abstract` examples of extends and implements

Java 8—Now with Multiple Inheritance!

It might have already occurred to you that since interfaces can now have concrete methods and classes can implement multiple interfaces, the spectre of multiple inheritance and the Deadly Diamond of Death can rear its ugly head! Well, you're partly correct. A class CAN implement interfaces with duplicate, concrete method signatures! But the good news is that the compiler's got your back, and if you DO want to implement both interfaces, you'll have to provide an overriding method in your class. Let's look at the following code:

Exam Watch

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

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

```
interface Vroom extends Fi, Baz { }          // OK. An interface can extend
                                             // multiple interfaces
class Yow extends Foo implements Fi { }      // OK. A class can do both
                                             // (extends must be 1st)
class Yow extends Foo implements Fi, Baz { } // OK. A class can do all three
                                             // (extends must be 1st)
```

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

```
interface I1 {
  default int doStuff() { return 1; }
}
interface I2 {
  default int doStuff() { return 2; }
}
public class MultiInt implements I1, I2 {  // needs to override
  public static void main(String[] args) {
    new MultiInt().go();
  }
  void go() {
    System.out.println(doStuff());
}
//  public int doStuff() {
//    return 3;
// }
}
```

As the code stands, it WILL NOT COMPILE because it's not clear which version of `doStuff()` should be used. In order to make the code compile, you need to override `doStuff()` in the class. Uncommenting the class's `doStuff()` method would allow the code to compile and when run produce the following output:

```
3
```

CERTIFICATION OBJECTIVE: LEGAL RETURN TYPES (OCA OBJECTIVES 2.2 AND 6.1)

2.2 Differentiate between object reference variables and primitive variables.

6.1 Create methods with arguments and return values; including overloaded methods.

This section covers two aspects of return types: what you can declare as a return type, and what you can actually return as a value. What you can and cannot declare is pretty straightforward, but it all depends on whether you're overriding an inherited method or simply declaring a new method (which includes overloaded methods). We'll take just a quick look at the difference between return type rules for overloaded and overriding methods, because we've already covered that in this chapter. We'll cover a small bit of new ground, though, when we look at polymorphic return types and the rules for what is and is not legal to actually return.

Return Type Declarations

This section looks at what you're allowed to declare as a return type, which depends primarily on whether you are overriding, overloading, or declaring a new method.

Return Types on Overloaded Methods

Remember that method overloading is not much more than name reuse. The overloaded method is a completely different method from any other method of the same name. So if you inherit a method but overload it in a subtype, you're not subject to the restrictions of overriding, which means you can declare any return type you like. What you can't do is change *only* the return type. To overload a method, remember, you must change the argument list. The following code shows an overloaded method:

```
public class Foo{
  void go() { }
}
public class Bar extends Foo {
  String go(int x) {
    return null;
  }
}
```

Notice that the `Bar` version of the method uses a different return type. That's perfectly fine. As long as you've changed the argument list, you're overloading the method, so the return type doesn't have to match that of the supertype version. What you're NOT allowed to do is this:

```
public class Foo{
  void go() { }
}
public class Bar extends Foo {
  String go() { // Not legal! Can't change only the return type
    return null;
  }
}
```

Overriding and Return Types and Covariant Returns

When a subtype wants to change the method implementation of an inherited method (an override), the subtype must define a method that matches the inherited version exactly. Or, since Java 5, you're allowed to change the return type in the overriding method as long as the new return type is a *subtype* of the declared return type of the overridden (superclass) method.

Let's look at a covariant return in action:

```
class Alpha {
  Alpha doStuff(char c) {
    return new Alpha();
  }
}
class Beta extends Alpha {
  Beta doStuff(char c) {    // legal override since Java 1.5
    return new Beta();
  }
}
```

Since Java 5, this code compiles. If you were to attempt to compile this code with a 1.4 compiler or with the source flag as follows,

```
javac -source 1.4 Beta.java
```

you would get a compiler error like this:

```
attempting to use incompatible return type
```

Other rules apply to overriding, including those for access modifiers and declared exceptions, but those rules aren't relevant to the return type discussion.

Exam Watch

For the exam, be sure you know that overloaded methods can change the return type, but overriding methods can do so only within the bounds of covariant returns. Just that knowledge alone will help you through a wide range of exam questions.

Returning a Value

You have to remember only six rules for returning a value:

1. You can return `null` in a method with an object reference return type.

   ```
   public Button doStuff() {
     return null;
   }
   ```

2. An array is a perfectly legal return type.

   ```
   public String[] go() {
     return new String[] {"Fred", "Barney", "Wilma"};
   }
   ```

3. In a method with a primitive return type, you can return any value or variable that can be implicitly converted to the declared return type.

   ```
   public int foo() {
     char c = 'c';
     return c; // char is compatible with int
   }
   ```

4. In a method with a primitive return type, you can return any value or variable that can be explicitly cast to the declared return type.

   ```
   public int foo() {
     float f = 32.5f;
     return (int) f;
   }
   ```

5. You must *not* return anything from a method with a void return type.

```
public void bar() {
  return "this is it";  // Not legal!!
}
```

   (Although you can say `return;`)

6. In a method with an object reference return type, you can return any object type that can be implicitly cast to the declared return type.

```
public Animal getAnimal() {
  return new Horse(); // Assume Horse extends Animal
}

public Object getObject() {
  int[] nums = {1,2,3};
  return nums;        // Return an int array, which is still an object
}

public interface Chewable { }
public class Gum implements Chewable { }

public class TestChewable {
 // Method with an interface return type
 public Chewable getChewable() {
   return new Gum(); // Return interface implementer
 }
}
```

Exam Watch

Watch for methods that declare an abstract class or interface return type, and know that any object that passes the IS-A test (in other words, would test true using the `instanceof` operator) can be returned from that method. For example:

```
public abstract class Animal { }
public class Bear extends Animal { }
public class Test {
  public Animal go() {
    return new Bear();  // OK, Bear "is-a" Animal
  }
}
```

This code will compile, and the return value is a subtype.

CERTIFICATION OBJECTIVE: CONSTRUCTORS AND INSTANTIATION (OCA OBJECTIVES 6.3 AND 7.4)

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

7.4 Use super and this to access objects and constructors.

Objects are constructed. You CANNOT make a new object without invoking a constructor. In fact, you can't make a new object without invoking not just the constructor of the object's actual class type, but also the constructor of each of its superclasses! Constructors are the code that runs whenever you use the keyword `new`. (Okay, to be a bit more accurate, there can also be initialization blocks that run when you say `new`, and we're going to cover `init` blocks and their static initialization counterparts after we discuss constructors.) We've got plenty to talk about here—we'll look at how constructors are coded, who codes them, and how they work at runtime. So grab your hardhat and a hammer, and let's do some object building.

Constructor Basics

Every class, *including abstract classes*, MUST have a constructor. Burn that into your brain. But just because a class must have a constructor doesn't mean the programmer has to type it. A constructor looks like this:

```
class Foo {
  Foo() { } // The constructor for the Foo class
}
```

Notice what's missing? There's no return type! Two key points to remember about constructors are that they have no return type, and their names must exactly match the class name. Typically, constructors are used to initialize the instance variable state, as follows:

```
class Foo {
  int size;
```

```
    String name;
    Foo(String name, int size) {
      this.name = name;
      this.size = size;
    }
  }
```

In the preceding code example, the `Foo` class does not have a no-arg constructor. That means the following will fail to compile:

```
Foo f = new Foo();            // Won't compile, no matching constructor
```

but the following will compile:

```
Foo f = new Foo("Fred", 43);  // No problem. Arguments match
                              // the Foo constructor.
```

So it's very common (and desirable) for a class to have a no-arg constructor, regardless of how many other overloaded constructors are in the class (yes, constructors can be overloaded). You can't always make that work for your classes; occasionally you have a class where it makes no sense to create an instance without supplying information to the constructor. A `java.awt.Color` object, for example, can't be created by calling a no-arg constructor, because that would be like saying to the JVM, "Make me a new Color object, and I really don't care what color it is…you pick." Do you seriously want the JVM making your style decisions?

Constructor Chaining

We know that constructors are invoked at runtime when you say `new` on some class type as follows:

```
Horse h = new Horse();
```

But what *really* happens when you say `new Horse()`? (Assume `Horse` extends `Animal` and `Animal` extends `Object`.)

1. The `Horse` constructor is invoked. Every constructor invokes the constructor of its superclass with an (implicit) call to `super()`, unless the constructor invokes an overloaded constructor of the same class (more on that in a minute).
2. The `Animal` constructor is invoked (`Animal` is the superclass of `Horse`).
3. The `Object` constructor is invoked (`Object` is the ultimate superclass of all classes, so class `Animal` extends `Object` even though you don't actually type "extends Object" into the `Animal` class declaration; it's implicit.) At this point we're on the top of the stack.
4. If class `Object` had any instance variables, then they would be given their explicit values. By *explicit* values, we mean values that are assigned at the time the variables are declared, such as `int x = 27`, where `27` is the explicit value (as opposed to the default value) of the instance variable.
5. The `Object` constructor completes.
6. The `Animal` instance variables are given their explicit values (if any).
7. The `Animal` constructor completes.
8. The `Horse` instance variables are given their explicit values (if any).
9. The `Horse` constructor completes.

Figure 2-6 shows how constructors work on the call stack.

Figure 2-6: Constructors on the call stack

Rules for Constructors

The following list summarizes the rules you'll need to know for the exam (and to understand the rest of this section). You MUST remember these, so be sure to study them more than once.

- Constructors can use any access modifier, including `private`. (A `private` constructor means only code within the class itself can instantiate an object of that type, so if the `private` constructor class wants to allow an instance of the class to be used, the class must provide a static method or variable that allows access to an instance created from within the class.)
- The constructor name must match the name of the class.
- Constructors must not have a return type.
- It's legal (but stupid) to have a method with the same name as the class, but that doesn't make it a constructor. If you see a return type, it's a method rather than a constructor. In fact, you could have both a method and a constructor with the same name—the name of the class—in the same class, and that's not a problem for Java. Be careful not to mistake a method for a constructor—be sure to look for a return type.
- If you don't type a constructor into your class code, a default constructor will be automatically generated by the compiler.
- The default constructor is ALWAYS a no-arg constructor.
- If you want a no-arg constructor and you've typed any other constructor(s) into your class code, the compiler won't provide the no-arg constructor (or any other constructor) for you. In other words, if you've typed in a constructor with arguments, you won't have a no-arg constructor unless you typed it in yourself!
- Every constructor has, as its first statement, either a call to an overloaded constructor (`this()`) or a call to the superclass constructor (`super()`), although remember that this call can be inserted by the compiler.
- If you do type in a constructor (as opposed to relying on the compiler-generated default constructor), and you do not type in the call to `super()` or a call to `this()`, the compiler will insert a no-arg call to `super()` for you as the very first statement in the constructor.
- A call to `super()` can either be a no-arg call or can include arguments passed to the super constructor.
- A no-arg constructor is not necessarily the default (that is, compiler-supplied) constructor, although the default constructor is always a no-arg constructor. The default constructor is the one the compiler provides! Although the default constructor is always a no-arg constructor, you're free to put in your own no-arg constructor.
- You cannot make a call to an instance method or access an instance variable until after the super constructor runs.
- Only static variables and methods can be accessed as part of the call to `super()` or `this()`. (Example: `super(Animal.NAME)` is OK, because `NAME` is declared as a static variable.)
- Abstract classes have constructors, and those constructors are always called when a concrete subclass is instantiated.
- Interfaces do not have constructors. Interfaces are not part of an object's inheritance tree.
- The only way a constructor can be invoked is from within another constructor. In other words, you can't write code that actually calls a constructor as follows:

```
class Horse {
  Horse() { } // constructor
  void doStuff() {
    Horse();  // calling the constructor - illegal!
  }
}
```

Determine Whether a Default Constructor Will Be Created

The following example shows a `Horse` class with two constructors:

```
class Horse {
  Horse() { }
  Horse(String name) { }
}
```

Will the compiler put in a default constructor for this class? No!

How about for the following variation of the class?

```
class Horse {
  Horse(String name) { }
}
```

Now will the compiler insert a default constructor? No!

What about this class?

```
class Horse { }
```

Now we're talking. The compiler will generate a default constructor for this class because the class doesn't have any constructors defined.

Okay, what about this class?

```
class Horse {
  void Horse() { }
}
```

It might look like the compiler won't create a constructor, since one is already in the `Horse` class. Or is it? Take another look at the preceding `Horse` class.

What's wrong with the `Horse()` constructor? It isn't a constructor at all! It's simply a method that happens to have the same name as the class. Remember, the return type is a dead giveaway that we're looking at a method, not a constructor.

How do you know for sure whether a default constructor will be created? Because you didn't write any constructors in your class.

How do you know what the default constructor will look like? Because…

- The default constructor has the same access modifier as the class.
- The default constructor has no arguments.
- The default constructor includes a no-arg call to the super constructor (`super()`).

Table 2-5 shows what the compiler will (or won't) generate for your class.

What happens if the super constructor has arguments? Constructors can have arguments just as methods can, and if you try to invoke a method that takes, say, an `int`, but you don't pass anything to the method, the compiler will complain as follows:

```
class Bar {
  void takeInt(int x) { }
}

class UseBar {
  public static void main (String [] args) {
    Bar b = new Bar();
    b.takeInt();  // Try to invoke a no-arg takeInt() method
  }
}
```

Table 2-5: Compiler-Generated Constructor Code

Class Code (What You Type)	Compiler-Generated Constructor Code (in Bold)
class Foo { }	class Foo { **Foo() {** **super();** **}** }
class Foo { Foo() { } }	class Foo { Foo() { **super();** } }
public class Foo { }	public class Foo { **public Foo() {** **super();** **}** }
class Foo { Foo(String s) { } }	class Foo { Foo(String s) { **super();** } }
class Foo { Foo(String s) { super(); } }	*Nothing; compiler doesn't need to insert anything.*
class Foo { void Foo() { }	class Foo { void Foo() { }

}	**Foo()** { **super();** } } (`void Foo()` is a method, not a constructor.)

The compiler will complain that you can't invoke `takeInt()` without passing an `int`. Of course, the compiler enjoys the occasional riddle, so the message it spits out on some versions of the JVM (your mileage may vary) is less than obvious:

```
UseBar.java:7: takeInt(int) in Bar cannot be applied to ()
  b.takeInt();
  ^
```

But you get the idea. The bottom line is that there must be a match for the method. And by match, we mean the argument types must be able to accept the values or variables you're passing and in the order you're passing them. Which brings us back to constructors (and here you were thinking we'd never get there), which work exactly the same way.

So if your super constructor (that is, the constructor of your immediate superclass/parent) has arguments, you must type in the call to `super()`, supplying the appropriate arguments. Crucial point: if your superclass does not have a no-arg constructor, you must type a constructor in your class (the subclass) because you need a place to put in the call to `super()` with the appropriate arguments.

The following is an example of the problem:

```
class Animal {
  Animal(String name) { }
}

class Horse extends Animal {
  Horse() {
    super();  // Problem!
  }
}
```

And once again the compiler treats us with stunning lucidity:

```
Horse.java:7: cannot resolve symbol
symbol  : constructor Animal  ()
location: class Animal
  super();  // Problem!
  ^
```

If you're lucky (and it's a full moon), *your* compiler might be a little more explicit. But again, the problem is that there just isn't a match for what we're trying to invoke with `super()`—an `Animal` constructor with no arguments.

Another way to put this is that if your superclass does *not* have a no-arg constructor, then in your subclass you will not be able to use the default constructor supplied by the compiler. It's that simple. Because the compiler can *only* put in a call to a no-arg `super()`, you won't even be able to compile something like this:

```
class Clothing {
  Clothing(String s) { }
}
class TShirt extends Clothing { }
```

Trying to compile this code gives us exactly the same error we got when we put a constructor in the subclass with a call to the no-arg version of `super()`:

```
Clothing.java:4: cannot resolve symbol
symbol  : constructor Clothing  ()
location: class Clothing
class TShirt extends Clothing { }
^
```

In fact, the preceding `Clothing` and `TShirt` code is implicitly the same as the following code, where we've supplied a constructor for `TShirt` that's identical to the default constructor supplied by the compiler:

```
class Clothing {
  Clothing(String s) { }
}
class TShirt extends Clothing {
                // Constructor identical to compiler-supplied
                // default constructor
  TShirt() {
    super();    // Won't work!
  }             // tries to invoke a no-arg Clothing constructor
```

```
                    // but there isn't one
}
```

One last point on the whole default constructor thing (and it's probably very obvious, but we have to say it or we'll feel guilty for years), ***constructors are never inherited***. They aren't methods. They can't be overridden (because they aren't methods, and only instance methods can be overridden). So the type of constructor(s) your superclass has in no way determines the type of default constructor you'll get. Some folks mistakenly believe that the default constructor somehow matches the super constructor, either by the arguments the default constructor will have (remember, the default constructor is always a no-arg) or by the arguments used in the compiler-supplied call to `super()`.

So although constructors can't be overridden, you've already seen that they can be overloaded, and typically are.

Overloaded Constructors

Overloading a constructor means typing in multiple versions of the constructor, each having a different argument list, like the following examples:

```
class Foo {
  Foo() { }
  Foo(String s) { }
}
```

The preceding `Foo` class has two overloaded constructors: one that takes a string, and one with no arguments. Because there's no code in the no-arg version, it's actually identical to the default constructor the compiler supplies—but remember, since there's already a constructor in this class (the one that takes a string), the compiler won't supply a default constructor. If you want a no-arg constructor to overload the with-args version you already have, you're going to have to type it yourself, just as in the `Foo` example.

Overloading a constructor is typically used to provide alternate ways for clients to instantiate objects of your class. For example, if a client knows the animal name, they can pass that to an `Animal` constructor that takes a string. But if they don't know the name, the client can call the no-arg constructor, and that constructor can supply a default name. Here's what it looks like:

```
public class Animal {
  String name;
  Animal(String name) {
    this.name = name;
  }

  Animal() {
    this(makeRandomName());
  }

  static String makeRandomName() {
    int x = (int) (Math.random() * 5);
    String name = new String[] {"Fluffy", "Fido",
                                "Rover", "Spike",
                                "Gigi"}[x];
    return name;
  }

  public static void main (String [] args) {
    Animal a = new Animal();
    System.out.println(a.name);
    Animal b = new Animal("Zeus");
    System.out.println(b.name);
  }
}
```

Running the code four times produces this output:

```
% java Animal
Gigi
Zeus

% java Animal
Fluffy
Zeus

% java Animal
Rover
Zeus

% java Animal
Fluffy
```

```
Zeus
```

There's a lot going on in the preceding code. Figure 2-7 shows the call stack for constructor invocations when a constructor is overloaded.

4. `Object()`
3. `Animal(String s)` calls `super()`
2. `Animal()` calls `this(randomlyChosenNameString)`
1. `main()` calls `new Animal()`

Figure 2-7: Overloaded constructors on the call stack

Take a look at the call stack, and then let's walk through the code straight from the top.

- **Line 2** Declare a `String` instance variable name.
- **Lines 3–5** Constructor that takes a `String` and assigns it to instance variable name.
- **Line 7** Here's where it gets fun. Assume every animal needs a name, but the client (calling code) might not always know what the name should be, so the `Animal` class will assign a random name. The no-arg constructor generates a name by invoking the `makeRandomName()` method.
- **Line 8** The no-arg constructor invokes its own overloaded constructor that takes a `String`, in effect calling it the same way it would be called if client code were doing a `new` to instantiate an object, passing it a `String` for the name. The overloaded invocation uses the keyword `this`, but uses it as though it were a method named `this()`. So line 8 is simply calling the constructor on line 3, passing it a randomly selected `String` rather than a client-code chosen name.
- **Line 11** Notice that the `makeRandomName()` method is marked `static`! That's because you cannot invoke an instance (in other words, nonstatic) method (or access an instance variable) until after the `super` constructor has run. And since the `super` constructor will be invoked from the constructor on line 3, rather than from the one on line 7, line 8 can use only a static method to generate the name. If we wanted all `animals` not specifically named by the caller to have the same default name, say, "Fred," then line 8 could have read `this("Fred");` rather than calling a method that returns a string with the randomly chosen name.
- **Line 12** This doesn't have anything to do with constructors, but since we're all here to learn, it generates a random integer between 0 and 4.
- **Line 13** Weird syntax, we know. We're creating a new `String` object (just a single `String` instance), but we want the string to be selected randomly from a list. Except we don't have the list, so we need to make it. So in that one line of code we
  1. Declare a `String` variable name.
  2. Create a `String` array (anonymously—we don't assign the array itself to a variable).
  3. Retrieve the string at index `[x]` (x being the random number generated on line 12) of the newly created `String` array.
  4. Assign the string retrieved from the array to the declared instance variable name. We could have made it much easier to read if we'd just written

     ```
     String[] nameList = {"Fluffy", "Fido", "Rover", "Spike",
                          "Gigi"};

     String name = nameList[x];
     ```

     But where's the fun in that? Throwing in unusual syntax (especially for code wholly unrelated to the real question) is in the spirit of the exam. Don't be startled! (Okay, be startled, but then just say to yourself, "Whoa!" and get on with it.)
- **Line 18** We're invoking the no-arg version of the constructor (causing a random name from the list to be passed to the other constructor).
- **Line 20** We're invoking the overloaded constructor that takes a string representing the name.

The key point to get from this code example is in line 8. Rather than calling `super()`, we're calling `this()`, and `this()` always means a call to another constructor in the same class. Okay, fine, but what happens after the call to `this()`? Sooner or later the `super()` constructor gets called, right? Yes, indeed. A call to `this()` just means you're delaying the inevitable. Some constructor, somewhere, must make the call to `super()`.

Key Rule: The first line in a constructor must be a call to `super()` or a call to `this()`.

No exceptions. If you have neither of those calls in your constructor, the compiler will insert the no-arg call to `super()`. In other words, if constructor `A()` has a call to `this()`, the compiler knows that constructor `A()` will not be the one to invoke `super()`.

The preceding rule means a constructor can never have both a call to `super()` and a call to `this()`. Because each of those calls must be the first statement in a constructor, you can't legally use both in the same constructor. That also means the compiler will not put a call to `super()` in any constructor that has a call to `this()`.

Thought question: What do you think will happen if you try to compile the following code?

```
class A {
  A() {
    this("foo");
  }
  A(String s) {
    this();
  }
}
```

Your compiler may not actually catch the problem (it varies depending on your compiler, but most won't catch the problem). It assumes you know what you're doing. Can you spot the flaw? Given that a `super` constructor must always be called, where would the call to `super()` go? Remember, the compiler won't put in a default constructor if you've already got one or more constructors in your class. And when the compiler doesn't put in a default constructor, it still inserts a call to `super()` in any constructor that doesn't explicitly have a call to the `super` constructor—unless, that is, the constructor already has a call to `this()`. So in the preceding code, where can `super()` go? The only two constructors in the class both have calls to `this()`, and, in fact, you'll get exactly what you'd get if you typed the following method code:

```
public void go() {
  doStuff();
}
public void doStuff() {
  go();
}
```

Now can you see the problem? Of course you can. The stack explodes! It gets higher and higher and higher until it just bursts open and method code goes spilling out, oozing out of the JVM right onto the floor. Two overloaded constructors both calling `this()` are two constructors calling each other—over and over and over, resulting in this:

```
% java A
Exception in thread "main" java.lang.StackOverflowError
```

The benefit of having overloaded constructors is that you offer flexible ways to instantiate objects from your class. The benefit of having one constructor invoke another overloaded constructor is to avoid code duplication. In the `Animal` example, there wasn't any code other than setting the name, but imagine if after line 4 there was still more work to be done in the constructor. By putting all the other constructor work in just one constructor, and then having the other constructors invoke it, you don't have to write and maintain multiple versions of that other important constructor code. Basically, each of the other not-the-real-one overloaded constructors will call another overloaded constructor, passing it whatever data it needs (data the client code didn't supply).

Constructors and instantiation become even more exciting (just when you thought it was safe) when you get to inner classes, but we know you can stand to have only so much fun in one chapter, and besides, you don't have to deal with inner classes until you tackle the OCP exam.

CERTIFICATION OBJECTIVE: INITIALIZATION BLOCKS (OCA OBJECTIVES 1.2 AND 6.3-ISH)

1.2 Define the structure of a Java class

6.3 Create and overload constructors; including impact on default constructors

We've talked about two places in a class where you can put code that performs operations: methods and constructors. Initialization blocks are the third place in a Java program where operations can be performed. Static initialization blocks run when the class is first loaded, and instance initialization blocks run whenever an instance is created (a bit similar to a constructor). Let's look at an example:

```
class SmallInit {
static int x;
int y;

static { x = 7 ; }       // static init block
{ y = 8; }               // instance init block
}
```

As you can see, the syntax for initialization blocks is pretty terse. They don't have names, they can't take arguments, and they don't return anything. A *static* initialization block runs *once* when the class is first loaded. An *instance* initialization block runs once *every time a new instance is created*. Remember when we talked about the order in which constructor code executed? Instance `init` block code runs right after the call to `super()` in a constructor—in other words, after all `super` constructors have run.

You can have many initialization blocks in a class. It is important to note that unlike methods or constructors, *the order in which initialization* *blocks appear in a class matters*. When it's time for initialization blocks to run, if a class has more than one, they will run in the order in which they appear in the class file—in other words, from the top down. Based on the rules we just discussed, can you determine the output of the following program?

```
class Init {
  Init(int x) { System.out.println("1-arg const"); }
  Init() { System.out.println("no-arg const"); }
  static { System.out.println("1st static init"); }
  { System.out.println("1st instance init"); }
  { System.out.println("2nd instance init"); }
  static { System.out.println("2nd static init"); }

  public static void main(String [] args) {
    new Init();
    new Init(7);
  }
}
```

To figure this out, remember these rules:

- `init` blocks execute in the order in which they appear.
- Static `init` blocks run once, when the class is first loaded.
- Instance `init` blocks run every time a class instance is created.
- Instance `init` blocks run after the constructor's call to `super()`.

With those rules in mind, the following output should make sense:

```
1st static init
2nd static init
1st instance init
2nd instance init
no-arg const
1st instance init
2nd instance init
1-arg const
```

As you can see, the instance `init` blocks each ran twice. Instance `init` blocks are often used as a place to put code that all the constructors in a class should share. That way, the code doesn't have to be duplicated across constructors.

Finally, if you make a mistake in your static `init` block, the JVM can throw an `ExceptionInInitializerError`. Let's look at an example:

```
class InitError {
  static int [] x = new int[4];
  static { x[4] = 5; }           // bad array index!
  public static void main(String [] args) { }
}
```

It produces something like this:

```
Exception in thread "main" java.lang.ExceptionInInitializerError
Caused by: java.lang.ArrayIndexOutOfBoundsException: 4
        at InitError.<clinit>(InitError.java:3)
```

Exam Watch

By convention, `init` blocks usually appear near the top of the class file, somewhere around the constructors. However, this is the OCA exam we're talking about. Don't be surprised if you find an `init` block tucked in between a couple of methods, looking for all the world like a compiler error waiting to happen!

CERTIFICATION OBJECTIVE: STATICS (OCA OBJECTIVE 6.2)

6.2 Apply the static keyword to methods and fields.

Static Variables and Methods

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

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

The following code declares and uses a `static` counter variable:

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

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

```
Frog count is now 3
```

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

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

When this code executes, it should still create three `Frog` instances in `main()`, but the result is…a compiler error! We can't get this code to compile, let alone run.

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

The JVM doesn't know which Frog object's `frogCount` you're trying to access. The problem is that `main()` is itself a `static` method and thus isn't running against any particular instance of the class; instead it's running on the class itself. A `static` method can't access a nonstatic (instance) variable because there is no instance! That's not to say there aren't instances of the class alive on the heap, but rather that even if there are, the `static` method doesn't know anything about them. The same applies to instance methods; a `static` method can't directly invoke a nonstatic method. Think static = class, nonstatic = instance. Making the method called by the JVM (`main()`) a `static` method means the JVM doesn't have to create an instance of your class just to start running code.

Exam Watch

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

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

Understand that this code will never compile, because you can't access a nonstatic (instance) variable from a `static` method. Just think of the compiler saying, "Hey, I have no idea which `Foo` object's `x` variable you're trying to print!" Remember, it's the class running the `main()` method, not an instance of the class.

Of course, the tricky part for the exam is that the question won't look as obvious as the preceding code. The problem you're being tested for—accessing a nonstatic variable from a `static` method—will be buried in code that might appear to be testing something else. For example, the preceding code would be more likely to appear as

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

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

Accessing Static Methods and Variables

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

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

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

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

```
class Frog {
  private static int frogCount = 0;  // static variable
  static int getCount() {            // static getter method
    return frogCount;
  }
  public Frog() {
     frogCount += 1;         // Modify the value in the constructor
  }
}

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

```
      new Frog();
      new Frog();
      new Frog();
      System.out.println("from static "+Frog.getCount()); // static context
      new Frog();
      new TestFrog().go();
      Frog f = new Frog();
      System.out.println("use ref var "+f.getCount());    // use reference var
    }
    void go() {
      System.out.println("from inst "+Frog.getCount());   // instance context
    }
  }
```

which produces the output:

```
from static 3
from instance 4
use ref var 5
```

But just to make it really confusing, the Java language also allows you to use an object reference variable to access a `static` member. Did you catch the last line of `main()`? It included this invocation:

```
f.getCount(); // Access a static using an instance variable
```

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

Invoking static methods from interfaces is almost the same as invoking static methods from classes, except the "instance variable syntax trick" just discussed works only for static methods in classes. The following code demonstrates how interface static methods can and cannot be invoked:

```
interface FrogBoilable {
  static int getCtoF(int cTemp) {              // interface static method
    return (cTemp * 9 / 5) + 32;
  }
  default String hop() { return "hopping"; }   // interface default method
}

public class DontBoilFrogs implements FrogBoilable {
  public static void main(String[] args) {
    new DontBoilFrogs().go();
  }
  void go() {
    System.out.println(hop());                        // 1 – ok for default m
    // System.out.println(getCtoF(100));              // 2 - cannot find symbol

    System.out.println(FrogBoilable.getCtoF(100));  // 3 – ok for static m
    DontBoilFrogs dbf = new DontBoilFrogs();
    // System.out.println(dbf.getCtoF(100));        // 4 - cannot find symbol
  }
}
```

Let's review the code:

- **Line 1** is a legal invocation of an interface's `default` method.
- **Line 2** is an **illegal** attempt to invoke an interface's `static` method.
- **Line 3** is THE legal way to invoke an interface's `static` method.
- **Line 4** is another **illegal** attempt to invoke an interface's `static` method.

Figure 2-8 illustrates the effects of the `static` modifier on methods and variables.

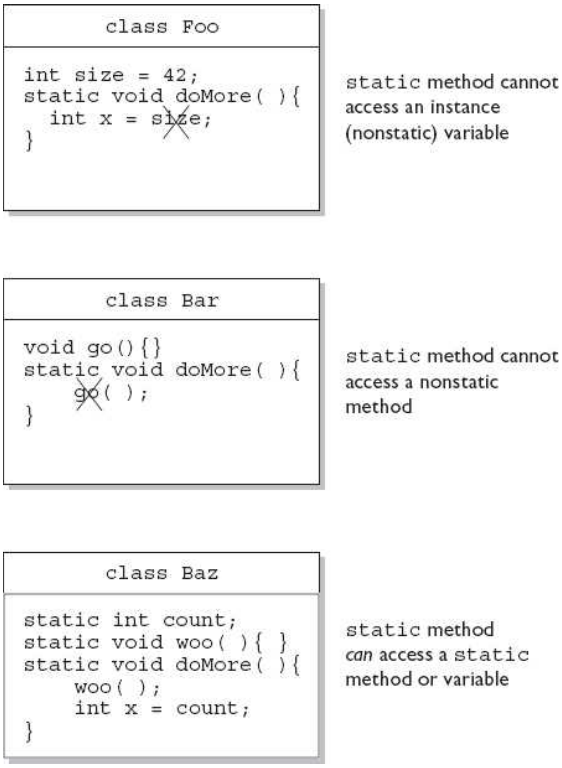

Figure 2-8: The effects of `static` on methods and variables

Finally, remember that *`static` methods can't be overridden*! This doesn't mean they can't be redefined in a subclass, but redefining and overriding aren't the same thing. Let's look at an example of a redefined (remember, not overridden) `static` method:

```
class Animal {
  static void doStuff() {
    System.out.print("a ");
  }
}
class Dog extends Animal {
  static void doStuff() {           // it's a redefinition,
                                    // not an override
    System.out.print("d ");
  }
  public static void main(String [] args) {
    Animal [] a = {new Animal(), new Dog(), new Animal()};
    for(int x = 0; x < a.length; x++) {
      a[x].doStuff();               // invoke the static method
    }
    Dog.doStuff();                  // invoke using the class name
  }
}
```

Running this code produces this output:

```
a a a d
```

Remember, the syntax a `[x].doStuff()` is just a shortcut (the syntax trick)—the compiler is going to substitute something like `Animal.doStuff()` instead. Notice also that you can invoke a `static` method by using the class name.

Notice that we didn't use the *enhanced* for *loop* here (covered in Chapter 5), even though we could have. Expect to see a mix of both Java 1.4 and Java 5–8 coding styles and practices on the exam.

CERTIFICATION SUMMARY

We started the chapter by discussing the importance of encapsulation in good OO design, and then we talked about how good encapsulation is implemented: with private instance variables and public getters and setters.

Next, we covered the importance of inheritance, so that you can grasp overriding, overloading, polymorphism, reference casting, return types, and constructors.

We covered IS-A and HAS-A. IS-A is implemented using inheritance, and HAS-A is implemented by using instance variables that refer to other objects.

Polymorphism was next. Although a reference variable's type can't be changed, it can be used to refer to an object whose type is a subtype of its own. We learned how to determine what methods are invocable for a given reference variable.

We looked at the difference between overridden and overloaded methods, learning that an overridden method occurs when a subtype inherits a method from a supertype and then reimplements the method to add more specialized behavior. We learned that, at runtime, the JVM will invoke the subtype version on an instance of a subtype and the supertype version on an instance of the supertype. `Abstract` methods must be "overridden" (technically, `abstract` methods must be implemented, as opposed to overridden, since there really isn't anything to override).

We saw that overriding methods must declare the same argument list and return type or they can return a subtype of the declared return type of the supertype's overridden method), and that the access modifier can't be more restrictive. The overriding method also can't throw any new or broader checked exceptions that weren't declared in the overridden method. You also learned that the overridden method can be invoked using the syntax `super.doSomething();`.

Overloaded methods let you reuse the same method name in a class, but with different arguments (and, optionally, a different return type). Whereas overriding methods must not change the argument list, overloaded methods must. But unlike overriding methods, overloaded methods are free to vary the return type, access modifier, and declared exceptions any way they like.

We learned the mechanics of casting (mostly downcasting) reference variables and when it's necessary to do so.

Implementing interfaces came next. An interface describes a *contract* that the implementing class must follow. The rules for implementing an interface are similar to those for extending an `abstract` class. As of Java 8, interfaces can have concrete methods, which are labeled `default`. Also, remember that a class can implement more than one interface and that interfaces can extend another interface.

We also looked at method return types and saw that you can declare any return type you like (assuming you have access to a class for an object reference return type), unless you're overriding a method. Barring a covariant return, an overriding method must have the same return type as the overridden method of the superclass. We saw that, although overriding methods must not change the return type, overloaded methods can (as long as they also change the argument list).

Finally, you learned that it is legal to return any value or variable that can be implicitly converted to the declared return type. So, for example, a `short` can be returned when the return type is declared as an `int`. And (assuming `Horse` extends `Animal`), a `Horse` reference can be returned when the return type is declared an `Animal`.

We covered constructors in detail, learning that if you don't provide a constructor for your class, the compiler will insert one. The compiler-generated constructor is called the default constructor, and it is always a no-arg constructor with a no-arg call to `super()`. The default constructor will never be generated if even a single constructor exists in your class (regardless of the arguments of that constructor); so if you need more than one constructor in your class and you want a no-arg constructor, you'll have to write it yourself. We also saw that constructors are not inherited and that you can be confused by a method that has the same name as the class (which is legal). The return type is the giveaway that a method is not a constructor because constructors do not have return types.

We saw how all the constructors in an object's inheritance tree will always be invoked when the object is instantiated using `new`. We also saw that constructors can be overloaded, which means defining constructors with different argument lists. A constructor can invoke another constructor of the same class using the keyword `this()`, as though the constructor were a method named `this()`. We saw that every constructor must have either `this()` or `super()` as the first statement (although the compiler can insert it for you).

After constructors, we discussed the two kinds of initialization blocks and how and when their code runs.

We looked at `static` methods and variables. `static` members are tied to the class or interface, not an instance, so there is only one copy of any `static` member. A common mistake is to attempt to reference an instance variable from a `static` method. Use the respective class or interface name with the dot operator to access `static` members.

And, once again, you learned that the exam includes tricky questions designed largely to test your ability to recognize just how tricky the questions can be.

TWO-MINUTE DRILL

Here are some of the key points from each certification objective in this chapter.

Encapsulation, IS-A, HAS-A* (OCA Objective 6.5)

- Encapsulation helps hide implementation behind an interface (or API).
- Encapsulated code has two features:
  - Instance variables are kept protected (usually with the `private` modifier).
  - Getter and setter methods provide access to instance variables.
- IS-A refers to inheritance or implementation.

- IS-A is expressed with the keyword `extends` or `implements`.
- IS-A, "inherits from," and "is a subtype of" are all equivalent expressions.
- HAS-A means an instance of one class "has a" reference to an instance of another class or another instance of the same class. *HAS-A is NOT on the exam, but it's good to know.

Inheritance (OCA Objective 7.1)

- Inheritance allows a type to be a subtype of a supertype and thereby inherit `public` and `protected` variables and methods of the supertype.
- Inheritance is a key concept that underlies IS-A, polymorphism, overriding, overloading, and casting.
- All classes (except class `Object`) are subclasses of type `Object`, and therefore they inherit `Object`'s methods.

Polymorphism (OCA Objective 7.2)

- Polymorphism means "many forms."
- A reference variable is always of a single, unchangeable type, but it can refer to a subtype object.
- A single object can be referred to by reference variables of many different types—as long as they are the same type or a supertype of the object.
- The reference variable's type (not the object's type) determines which methods can be called!
- Polymorphic method invocations apply only to overridden *instance* methods.

Overriding and Overloading (OCA Objectives 6.1 and 7.2)

- Methods can be overridden or overloaded; constructors can be overloaded but not overridden.
- With respect to the method it overrides, the overriding method
  - Must have the same argument list
  - Must have the same return type or a subclass (known as a covariant return)
  - Must not have a more restrictive access modifier
  - May have a less restrictive access modifier
  - Must not throw new or broader checked exceptions
  - May throw fewer or narrower checked exceptions, or any unchecked exception
- `final` methods cannot be overridden.
- Only inherited methods may be overridden, and remember that private methods are not inherited.
- A subclass uses `super.overriddenMethodName()` to call the superclass version of an overridden method.
- A subclass uses `MyInterface.super.overriddenMethodName()` to call the super interface version on an overridden method.
- Overloading means reusing a method name but with different arguments.
- Overloaded methods
  - Must have different argument lists
  - May have different return types, if argument lists are also different
  - May have different access modifiers
  - May throw different exceptions
- Methods from a supertype can be overloaded in a subtype.
- Polymorphism applies to overriding, not to overloading.
- Object type (not the reference variable's type) determines which overridden method is used at runtime.
- Reference type determines which overloaded method will be used at compile time.

Reference Variable Casting (OCA Objective 7.3)

- There are two types of reference variable casting: downcasting and upcasting.
  - **Downcasting** If you have a reference variable that refers to a subtype object, you can assign it to a reference variable of the subtype. You must make an explicit cast to do this, and the result is that you can access the subtype's members with this new reference variable.
  - **Upcasting** You can assign a reference variable to a supertype reference variable explicitly or implicitly. This is an inherently safe operation because the assignment restricts the access capabilities of the new variable.

Implementing an Interface (OCA Objective 7.5)

- When you implement an interface, you are fulfilling its contract.
- You implement an interface by properly and concretely implementing all the abstract methods defined by the interface.
- A single class can implement many interfaces.

Return Types (OCA Objectives 7.2 and 7.5)

- Overloaded methods can change return types; overridden methods cannot, except in the case of covariant returns.
- Object reference return types can accept `null` as a return value.
- An array is a legal return type, both to declare and return as a value.
- For methods with primitive return types, any value that can be implicitly converted to the return type can be returned.
- Nothing can be returned from a `void`, but you can return nothing. You're allowed to simply say `return` in any method with a `void` return type to bust out of a method early. But you can't return nothing from a method with a non-`void` return type.
- Methods with an object reference return type can return a subtype.
- Methods with an interface return type can return any implementer.

Constructors and Instantiation (OCA Objectives 6.3 and 7.4)

- A constructor is always invoked when a new object is created.
- Each superclass in an object's inheritance tree will have a constructor called.
- Every class, even an abstract class, has at least one constructor.
- Constructors must have the same name as the class.
- Constructors don't have a return type. If you see code with a return type, it's a method with the same name as the class; it's not a constructor.
- Typical constructor execution occurs as follows:
  - The constructor calls its superclass constructor, which calls its superclass constructor, and so on all the way up to the `Object` constructor.
  - The `Object` constructor executes and then returns to the calling constructor, which runs to completion and then returns to its calling constructor, and so on back down to the completion of the constructor of the actual instance being created.
- Constructors can use any access modifier (even `private`!).
- The compiler will create a default constructor if you don't create any constructors in your class.
- The default constructor is a no-arg constructor with a no-arg call to `super()`.
- The first statement of every constructor must be a call either to `this()` (an overloaded constructor) or to `super()`.
- The compiler will add a call to `super()` unless you have already put in a call to `this()` or `super()`.
- Instance members are accessible only after the `super` constructor runs.
- `Abstract` classes have constructors that are called when a concrete subclass is instantiated.
- Interfaces do not have constructors.
- If your superclass does not have a no-arg constructor, you must create a constructor and insert a call to `super()` with arguments matching those of the superclass constructor.
- Constructors are never inherited; thus they cannot be overridden.

- A constructor can be directly invoked only by another constructor (using a call to `super()` or `this()`).
- Regarding issues with calls to `this()`:
  - They may appear only as the first statement in a constructor.
  - The argument list determines which overloaded constructor is called.
  - Constructors can call constructors, and so on, but sooner or later one of them better call `super()` or the stack will explode.
  - Calls to `this()` and `super()` cannot be in the same constructor. You can have one or the other, but never both.

Initialization Blocks (OCA Objective 1.2 and 6.3-ish)

- Use `static init` blocks—`static { /* code here */ }`—for code you want to have run once, when the class is first loaded. Multiple blocks run from the top down.
- Use normal `init` blocks—`{ /* code here }`—for code you want to have run for every new instance, right after all the super constructors have run. Again, multiple blocks run from the top of the class down.

Statics (OCA Objective 6.2)

- Use `static` methods to implement behaviors that are not affected by the state of any instances.
- Use `static` variables to hold data that is class specific as opposed to instance specific—there will be only one copy of a `static` variable.
- All `static` members belong to the class, not to any instance.
- A `static` method can't access an instance variable directly.
- Use the dot operator to access `static` members, but remember that using a reference variable with the dot operator is really a syntax trick, and the compiler will substitute the class name for the reference variable; for instance:

  ```
  d.doStuff();
  ```

  becomes

  ```
  Dog.doStuff();
  ```
- To invoke an interface's static method use `MyInterface.doStuff()` syntax.
- `static` methods can't be overridden, but they can be redefined.

SELF TEST

1. Given: ?

```
public abstract interface Frobnicate { public void twiddle(String s); }
```

Which is a correct class? (Choose all that apply.)

A.
```
public abstract class Frob implements Frobnicate {
  public abstract void twiddle(String s) { }
}
```

B.
```
public abstract class Frob implements Frobnicate { }
```

C.
```
public class Frob extends Frobnicate {
  public void twiddle(Integer i) { }
}
```

D.
```
public class Frob implements Frobnicate {
  public void twiddle(Integer i) { }
}
```

E.
```
public class Frob implements Frobnicate {
  public void twiddle(String i) { }
  public void twiddle(Integer s) { }
}
```

2. Given: ?

```
class Top {
  public Top(String s) { System.out.print("B"); }
}
public class Bottom2 extends Top {
```

```
    public Bottom2(String s) { System.out.print("D"); }
    public static void main(String [] args) {
      new Bottom2("C");
      System.out.println(" ");
    }
  }
```

What is the result?

A. `BD`

B. `DB`

C. `BDC`

D. `DBC`

E. Compilation fails

3. Given: ?

```
class Clidder {
  private final void flipper() { System.out.println("Clidder"); }
}
public class Clidlet extends Clidder {
  public final void flipper() { System.out.println("Clidlet"); }
  public static void main(String [] args) {
    new Clidlet().flipper();
  }
}
```

What is the result?

A. `Clidlet`

B. `Clidder`

C. `Clidder`
   `Clidlet`

D. `Clidlet`
   `Clidder`

E. Compilation fails

Special Note: The next question crudely simulates a style of question known as "drag-and-drop." Up through the SCJP 6 exam, drag-and-drop questions were included on the exam. As of spring 2014, Oracle DOES NOT include any drag-and-drop questions on its Java exams, but just in case Oracle's policy changes, we left a few in the book.

4. Using the **fragments** below, complete the following **code** so it compiles. Note that you may not have to fill in all of the slots. ?

Code:

```
class AgedP {
  __________ __________ __________ __________ __________
  public AgedP(int x) {
    __________ __________ __________ __________ __________
  }
}
public class Kinder extends AgedP {
  __________ __________ __________ _________ ________ __________
  public Kinder(int x) {
    __________ __________ __________ __________ __________ ();
  }
}
```

Fragments: Use the following fragments zero or more times:

AgedP	super	this	
(	)	{	}
;			

5. Given: ?

```
class Bird {
  { System.out.print("b1 "); }
```

```
  public Bird() { System.out.print("b2 "); }
}
class Raptor extends Bird {
  static { System.out.print("r1 "); }
  public Raptor() { System.out.print("r2 "); }
  { System.out.print("r3 "); }
  static { System.out.print("r4 "); }
}
class Hawk extends Raptor {
  public static void main(String[] args) {
    System.out.print("pre ");
    new Hawk();
    System.out.println("hawk ");
  }
}
```

What is the result?

A. `pre b1 b2 r3 r2 hawk`

B. `pre b2 b1 r2 r3 hawk`

C. `pre b2 b1 r2 r3 hawk r1 r4`

D. `r1 r4 pre b1 b2 r3 r2 hawk`

E. `r1 r4 pre b2 b1 r2 r3 hawk`

F. `pre r1 r4 b1 b2 r3 r2 hawk`

G. `pre r1 r4 b2 b1 r2 r3 hawk`

H. The order of output cannot be predicted

I. Compilation fails

Note: You'll probably never see this many choices on the real exam!

6. Given the following: ?

```
 1. class X { void do1() { } }
 2. class Y extends X { void do2() { } }
 3.
 4. class Chrome {
 5.   public static void main(String [] args) {
 6.     X x1 = new X();
 7.     X x2 = new Y();
 8.     Y y1 = new Y();
 9.     // insert code here
10. } }
```

Which of the following, inserted at line 9, will compile? (Choose all that apply.)

A. `x2.do2();`

B. `(Y)x2.do2();`

C. `((Y)x2).do2();`

D. None of the above statements will compile

7. Given: ?

```
public class Locomotive {
  Locomotive() { main("hi"); }

  public static void main(String[] args) {
    System.out.print("2 ");
  }
  public static void main(String args) {
    System.out.print("3 " + args);
  }
}
```

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

A. 2 will be included in the output

B. 3 will be included in the output

C. `hi` will be included in the output

D. Compilation fails

E. An exception is thrown at runtime

8. Given: ?

```
 3. class Dog {
 4.   public void bark() { System.out.print("woof "); }
 5. }
 6. class Hound extends Dog {
 7.   public void sniff() { System.out.print("sniff "); }
 8.   public void bark() { System.out.print("howl "); }
 9. }
10. public class DogShow {
11.   public static void main(String[] args) { new DogShow().go(); }
12.   void go() {
13.     new Hound().bark();
14.     ((Dog) new Hound()).bark();
15.     ((Dog) new Hound()).sniff();
16.   }
17. }
```

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

A. `howl howl sniff`

B. `howl woof sniff`

C. `howl howl` followed by an exception

D. `howl woof` followed by an exception

E. Compilation fails with an error at line 14

F. Compilation fails with an error at line 15

9. Given: ?

```
 3. public class Redwood extends Tree {
 4.   public static void main(String[] args) {
 5.     new Redwood().go();
 6.   }
 7.   void go() {
 8.     go2(new Tree(), new Redwood());
 9.     go2((Redwood) new Tree(), new Redwood());
10.   }
11.   void go2(Tree t1, Redwood r1) {
12.      Redwood r2 = (Redwood)t1;
13.      Tree t2 = (Tree)r1;
14.   }
15. }
16. class Tree { }
```

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

A. An exception is thrown at runtime

B. The code compiles and runs with no output

C. Compilation fails with an error at line 8

D. Compilation fails with an error at line 9

E. Compilation fails with an error at line 12

F. Compilation fails with an error at line 13

10. Given: ?

```
 3. public class Tenor extends Singer {
 4.   public static String sing() { return "fa"; }
 5.   public static void main(String[] args) {
 6.     Tenor t = new Tenor();
```

```
    Singer s = new Tenor();
    System.out.println(t.sing() + " " + s.sing());
  }
}
class Singer { public static String sing() { return "la"; } }
```

What is the result?

A. `fa fa`

B. `fa la`

C. `la la`

D. Compilation fails

E. An exception is thrown at runtime

11. Given: ?

```
class Alpha {
  static String s = " ";
  protected Alpha() { s += "alpha "; }
}
class SubAlpha extends Alpha {
  private SubAlpha() { s += "sub "; }
}
public class SubSubAlpha extends Alpha {
  private SubSubAlpha() { s += "subsub "; }
  public static void main(String[] args) {
    new SubSubAlpha();
    System.out.println(s);
  }
}
```

What is the result?

A. `subsub`

B. `sub subsub`

C. `alpha subsub`

D. `alpha sub subsub`

E. Compilation fails

F. An exception is thrown at runtime

12. Given: ?

```
class Building {
  Building() {  System.out.print("b ");  }
  Building(String name) {
    this();   System.out.print("bn " + name);
  }
}
public class House extends Building {
  House() {  System.out.print("h ");   }
  House(String name) {
    this();   System.out.print("hn "  + name);
  }
  public static void main(String[] args)  { new House("x "); }
}
```

What is the result?

A. `h hn x`

B. `hn x h`

C. `b h hn x`

D. `b hn x h`

E. `bn x h hn x`

F. `b bn x h hn x`

G. `bn x b h hn x`

H. Compilation fails

13. Given: ?

```
class Mammal {
  String name = "furry ";
  String makeNoise() { return "generic noise"; }
}
class Zebra extends Mammal {
  String name = "stripes ";
  String makeNoise() { return "bray"; }
}
public class ZooKeeper {
  public static void main(String[] args) { new ZooKeeper().go(); }
  void go() {
    Mammal m = new Zebra();
    System.out.println(m.name + m.makeNoise());
  }
}
```

What is the result?

A. `furry bray`

B. `stripes bray`

C. `furry generic noise`

D. `stripes generic noise`

E. Compilation fails

F. An exception is thrown at runtime

14. Given: ?

```
 1. interface FrogBoilable {
 2.   static int getCtoF(int cTemp) {
 3.     return (cTemp * 9 / 5) + 32;
 4.   }
 5.   default String hop() { return "hopping "; }
 6. }
 7. public class DontBoilFrogs implements FrogBoilable {
 8.   public static void main(String[] args) {
 9.     new DontBoilFrogs().go();
10.   }
11.   void go() {
12.     System.out.print(hop());
13.     System.out.println(getCtoF(100));
14.     System.out.println(FrogBoilable.getCtoF(100));
15.     DontBoilFrogs dbf = new DontBoilFrogs();
16.     System.out.println(dbf.getCtoF(100));
17.   }
18. }
```

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

A. `hopping 212`

B. Compilation fails due to an error on line 2

C. Compilation fails due to an error on line 5

D. Compilation fails due to an error on line 12

E. Compilation fails due to an error on line 13

F. Compilation fails due to an error on line 14

G. Compilation fails due to an error on line 16

15. Given: ?

```
interface I1 {
  default int doStuff() { return 1; }
}
interface I2 {
  default int doStuff() { return 2; }
}
public class MultiInt implements I1, I2 {
  public static void main(String[] args) {
    new MultiInt().go();
  }
  void go() {
    System.out.println(doStuff());
  }
  int doStuff() {
    return 3;
  }
}
```

What is the result?

A. `1`

B. `2`

C. `3`

D. The output is unpredictable

E. Compilation fails

F. An exception is thrown at runtime

16. Given: ?

```
interface MyInterface {
  default int doStuff() {
    return 42;
  }
}
public class IfaceTest implements MyInterface {
  public static void main(String[] args) {
    new IfaceTest().go();
  }
  void go() {
    // INSERT CODE HERE
  }
  public int doStuff() {
    return 43;
  }
}
```

Which line(s) of code, inserted independently at // INSERT CODE HERE, will allow the code to compile? (Choose all that apply.)

A. `System.out.println("class: " + doStuff());`

B. `System.out.println("iface: " + super.doStuff());`

C. `System.out.println("iface: " + MyInterface.super.doStuff());`

D. `System.out.println("iface: " + MyInterface.doStuff());`

E. `System.out.println("iface: " + super.MyInterface.doStuff());`

F. None of the lines, A–E will allow the code to compile

Answers

1. ☑ **B** and **E** are correct. B is correct because an `abstract` class need not implement any or all of an interface's methods. E is correct because the class implements the interface method and additionally overloads the `twiddle()` method.

☒ **A, C**, and **D** are incorrect. A is incorrect because `abstract` methods have no body. C is incorrect because classes implement interfaces; they don't extend them. D is incorrect because overloading a method is not implementing it. (OCA Objectives 7.1 and 7.5)

2. ☑ **E** is correct. The implied `super()` call in `Bottom2`'s constructor cannot be satisfied because there is no no-arg constructor in `Top`. A default, no-arg constructor is generated by the compiler only if the class has no constructor defined explicitly.

   ☒ **A, B, C**, and **D** are incorrect based on the above. (OCA Objective 6.3)

3. ☑ **A** is correct. Although a `final` method cannot be overridden, in this case, the method is private and, therefore, hidden. The effect is that a new, accessible, method flipper is created. Therefore, no polymorphism occurs in this example, the method invoked is simply that of the child class, and no error occurs.

   ☒ **B, C, D**, and **E** are incorrect based on the preceding. (OCA Objective 7.2)

   Special Note: This next question crudely simulates a style of question known as "drag-and-drop." Up through the SCJP 6 exam, drag-and-drop questions were included on the exam. As of spring 2014, Oracle DOES NOT include any drag-and-drop questions on its Java exams, but just in case Oracle's policy changes, we left a few in the book.

4. Here is the answer:

```
    class AgedP {
      AgedP() {}
      public AgedP(int x) {
      }
    }
    public class Kinder extends AgedP {
      public Kinder(int x) {
        super();
    }
}
```

   As there is no droppable tile for the variable `x` and the parentheses (in the `Kinder` constructor) are already in place and empty, there is no way to construct a call to the superclass constructor that takes an argument. Therefore, the only remaining possibility is to create a call to the no-arg superclass constructor. This is done as `super();`. The line cannot be left blank, as the parentheses are already in place. Further, since the superclass constructor called is the no-arg version, this constructor must be created. It will not be created by the compiler because another constructor is already present. (OCA Objectives 6.3 and 7.4) Note: As you can see, many questions test for OCA Objective 7.1, we're going to stop mentioning objective 7.1.

5. ☑ **D** is correct. Static `init` blocks are executed at class loading time; instance `init` blocks run right after the call to `super()` in a constructor. When multiple `init` blocks of a single type occur in a class, they run in order, from the top down.

   ☒ **A, B, C, E, F, G, H**, and **I** are incorrect based on the above. Note: You'll probably never see this many choices on the real exam! (OCA Objective 6.3)

6. ☑ **C** is correct. Before you can invoke `Y`'s `do2` method, you have to cast `x2` to be of type `Y`.

   ☒ **A, B**, and **D** are incorrect based on the preceding. B looks like a proper cast, but without the second set of parentheses, the compiler thinks it's an incomplete statement. (OCA Objective 7.3)

7. ☑ **A** is correct. It's legal to overload `main()`. Since no instances of `Locomotive` are created, the constructor does not run and the overloaded version of `main()` does not run.

   ☒ **B, C, D**, and **E** are incorrect based on the preceding. (OCA Objectives 1.3 and 6.3)

8. ☑ **F** is correct. Class `Dog` doesn't have a `sniff` method.

   ☒ **A, B, C, D**, and **E** are incorrect based on the above information. (OCA Objectives 7.2 and 7.3)

9. ☑ **A** is correct. A `ClassCastException` will be thrown when the code attempts to downcast a `Tree` to a `Redwood`.

   ☒ **B, C, D, E**, and **F** are incorrect based on the above information. (OCA Objective 7.3)

10. ☑ **B** is correct. The code is correct, but polymorphism doesn't apply to `static` methods.

☒ **A, C, D**, and **E** are incorrect based on the above information. (OCA Objectives 6.2 and 7.2)

11. ☑ **C** is correct. Watch out, because `SubSubAlpha` extends `Alpha`! Because the code doesn't attempt to make a `SubAlpha`, the private constructor in `SubAlpha` is okay.

☒ **A, B, D, E**, and **F** are incorrect based on the above information. (OCA Objectives 6.3 and 7.2)

12. ☑ **C** is correct. Remember that constructors call their superclass constructors, which execute first, and that constructors can be overloaded.

☒ **A, B, D, E, F, G**, and **H** are incorrect based on the above information. (OCA Objectives 6.3 and 7.4)

13. ☑ **A** is correct. Polymorphism is only for instance methods, not instance variables.

☒ **B, C, D, E**, and **F** are incorrect based on the above information. (OCA Objective 6.3)

14. ☑ **E** and **G** are correct. Neither of these lines of code uses the correct syntax to invoke an interface's static method.

☒ **A, B, C, D**, and **F** are incorrect based on the above information. (OCP Objectives 6.2 and 7.5)

15. ☑ **E** is correct. This is kind of a trick question; the implementing method must be marked `public`. If it was, all the other code is legal, and the output would be 3. If you understood all the multiple inheritance rules and just missed the access modifier, give yourself half credit.

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

16. ☑ **A** and **C** are correct. A uses correct syntax to invoke the class's method, and C uses the correct syntax to invoke the interface's overloaded `default` method.

☒ **B, D, E**, and **F** are incorrect. (OCP Objective 7.5)