Chapter 8
Class Design

OCP EXAM OBJECTIVES COVERED IN THIS CHAPTER:

- **Creating and Using Methods**
  - Create methods and constructors with arguments and return values
- **Reusing Implementations Through Inheritance**
  - Create and use subclasses and superclasses
  - Enable polymorphism by overriding methods
  - Utilize polymorphism to cast and call methods, differentiating object type versus reference type
  - Distinguish overloading, overriding, and hiding

In Chapter 2, "Java Building Blocks," we introduced the basic definition for a class in Java. In Chapter 7, "Methods and Encapsulation," we delved into methods and modifiers and showed how you can use them to build more structured classes. In this chapter, we'll take things one step further and show how class structure and inheritance is one of the most powerful features in the Java language.

At its core, proper Java class design is about code reusability, increased functionality, and standardization. For example, by creating a new class that extends an existing class, you may gain access to a slew of inherited primitives, objects, and methods, which increases code reuse. Through polymorphism, you may also gain access to a dynamic hierarchy that supports replacing method implementations in subclasses at runtime.

This chapter is the culmination of some of the most important topics in Java including class design, constructor overloading and inheritance, order of initialization, overriding/hiding methods, and polymorphism. Read this chapter carefully and make sure you understand all of the topics well. This chapter forms the basis of Chapter 9, "Advanced Class Design," in which we will expand our discussion of types to include abstract classes and interfaces.

Understanding Inheritance

When creating a new class in Java, you can define the class as inheriting from an existing class. *Inheritance* is the process by which a subclass automatically includes any `public` or `protected` members of the class, including primitives, objects, or methods, defined in the parent class.

For illustrative purposes, we refer to any class that inherits from another class as a *subclass* or *child class*, as it is considered a descendant of that class. Alternatively, we refer to the class that the child inherits from as the *superclass* or *parent class*, as it is considered an ancestor of the class. And inheritance is transitive. If child class X inherits from parent class Y, which in turn inherits from a parent class Z, then class X would be considered a subclass, or descendant, of class Z. By comparison, X is a direct descendant only of class Y, and Y is a direct descendant only of class Z.

In the last chapter, you learned that there are four access levels: `public`, `protected`, package-private, and `private`. When one class inherits from a parent class, all `public` and `protected` members are automatically available as part of the child class. Package-private members are available if the child class is in the same package as the parent class. Last but not least, `private` members are restricted to the class they are defined in and are never available via inheritance. This doesn't mean the parent class doesn't have `private` members that can hold data or modify an object; it just means the child class has no direct reference to them.

Let's take a look at a simple example with the `BigCat` and `Jaguar` classes. In this example, `Jaguar` is a subclass or child of `BigCat`, making `BigCat` a superclass or parent of `Jaguar`.

```
public class BigCat {
   public double size;
}

public class Jaguar extends BigCat {
   public Jaguar() {
      size = 10.2;
   }
   public void printDetails() {
      System.out.println(size);
   }
}
```

In the `Jaguar` class, `size` is accessible because it is marked `public`. Via inheritance, the `Jaguar` subclass can read or write `size` as if it were its own member.

Single vs. Multiple Inheritance

Java supports *single inheritance*, by which a class may inherit from only one direct parent class. Java also supports multiple levels of inheritance, by which one class may extend another class, which in turn extends another class. You can have any number of levels of inheritance, allowing each descendant to gain access to its ancestor's members.

To truly understand single inheritance, it may helpful to contrast it with *multiple inheritance*, by which a class may have multiple direct parents. By design, Java doesn't support multiple inheritance in the language because multiple inheritance can lead to complex, often difficult-to-maintain data models. Java does allow one exception to the single inheritance rule that you'll see in Chapter 9—a class may implement multiple interfaces.

Figure 8.1 illustrates the various types of inheritance models. The items on the left are considered single inheritance because each child has exactly one parent. You may notice that single inheritance doesn't preclude parents from having multiple children. The right side shows items that have multiple inheritance. As you can see, a `Dog` object has multiple parent designations. Part of what makes multiple inheritance complicated is determining which parent to inherit values from in case of a conflict. For example, if you have an object or method defined in all of the parents, which one does the child inherit? There is no natural ordering for parents in this example, which is why Java avoids these issues by disallowing multiple inheritance altogether.

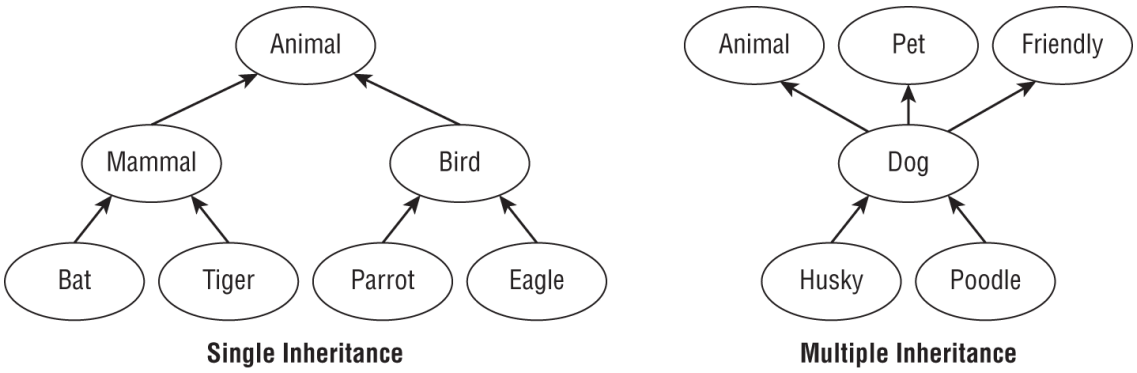

FIGURE 8.1 Types of inheritance

It is possible in Java to prevent a class from being extended by marking the class with the `final` modifier. If you try to define a class that inherits from a `final` class, then the class will fail to compile. Unless otherwise specified, throughout this chapter you can assume the classes we work with are not marked `final`.

Inheriting *Object*

Throughout our discussion of Java in this book, we have thrown around the word *object* numerous times—and with good reason. In Java, all classes inherit from a single class: `java.lang.Object`, or `Object` for short. Furthermore, `Object` is the only class that doesn't have a parent class.

You might be wondering, “None of the classes I’ve written so far extend `Object`, so how do all classes inherit from it?” The answer is that the compiler has been automatically inserting code into any class you write that doesn’t extend a specific class. For example, consider the following two equivalent class definitions:

```
public class Zoo { }

public class Zoo extends java.lang.Object { }
```

The key is that when Java sees you define a class that doesn’t extend another class, it automatically adds the syntax `extends java.lang.Object` to the class definition. The result is that every class gains access to any accessible methods in the `Object` class. For example, the `toString()` and `equals()` methods are available in `Object`; therefore, they are accessible in all classes. Without being overridden in a subclass, though, they may not be particularly useful. We will cover overriding methods later in this chapter.

On the other hand, when you define a new class that extends an existing class, Java does not automatically extend the `Object` class. Since all classes inherit from `Object`, extending an existing class means the child already inherits from `Object` by definition. If you look at the inheritance structure of any class, it will always end with `Object` on the top of the tree, as shown in Figure 8.2.

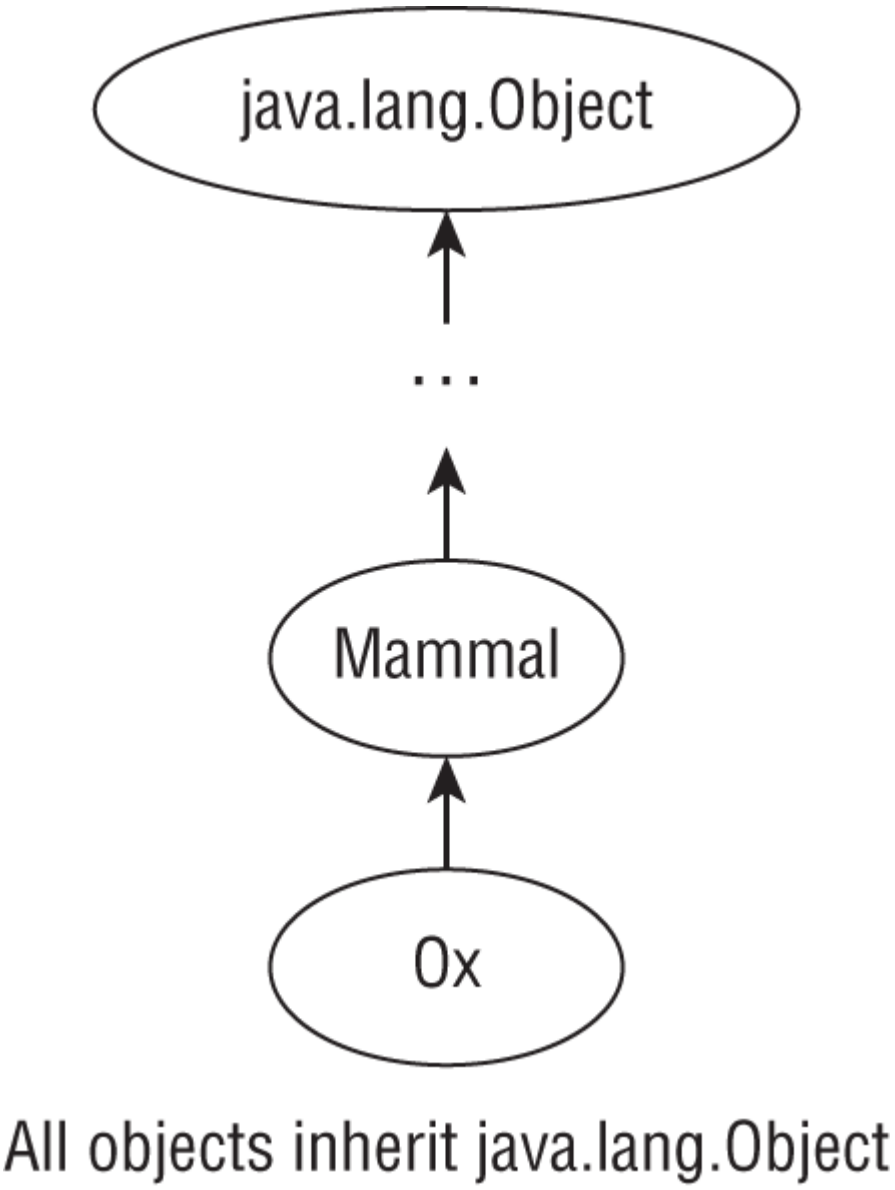

FIGURE 8.2 Java object inheritance

Primitive types such as `int` and `boolean` do not inherit from `Object`, since they are not classes. As you learned in Chapter 5, "Core Java APIs," through autoboxing they can be assigned or passed as an instance of an associated wrapper class, which does inherit `Object`.

Creating Classes

Now that we've established how inheritance works in Java, we can use it to define and create complex class relationships. In this section, we will review the basics for creating and working with classes.

Extending a Class

The full syntax of defining and extending a class using the `extends` keyword is shown in Figure 8.3.

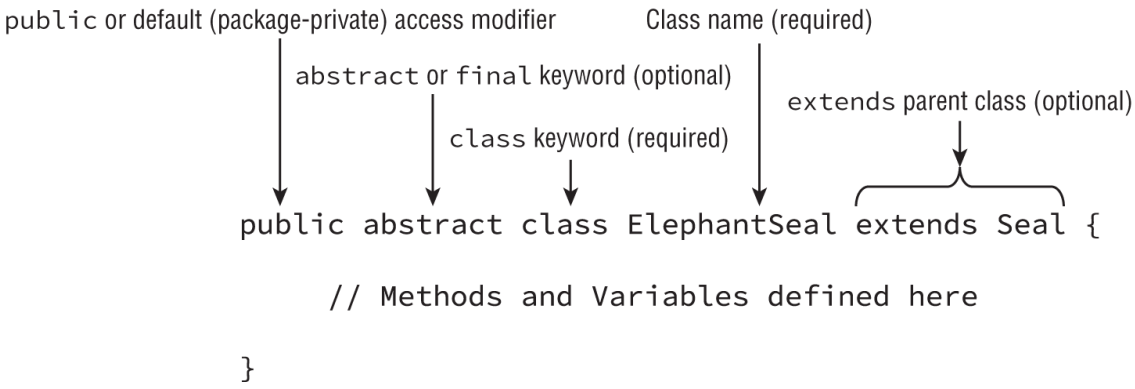

FIGURE 8.3 Defining and extending a class

Remember that `final` means a class cannot be extended. We'll discuss what it means for a class to be `abstract` in Chapter 9.

Let's create two files, `Animal.java` and `Lion.java`, in which the `Lion` class extends the `Animal` class. Assuming they are in the same package, an import statement is not required in `Lion.java` to access the `Animal` class.

Here are the contents of `Animal.java`:

```
public class Animal {
   private int age;
   protected String name;
   public int getAge() {
      return age;
   }
   public void setAge(int newAge) {
      age = newAge;
   }
}
```

And here are the contents of `Lion.java`:

```
public class Lion extends Animal {
   public void setProperties(int age, String n) {
      setAge(age);
      name = n;
   }
   public void roar() {
      System.out.print(name + ", age " + getAge() + ", says: Roar!");
   }
   public static void main(String[] args) {
      var lion = new Lion();
      lion.setProperties(3, "kion");
      lion.roar();
   }
}
```

The `extends` keyword is used to express that the `Lion` class inherits the `Animal` class. When executed, the `Lion` program prints the following:

```
kion, age 3, says: Roar!
```

Let's take a look at the members of the `Lion` class. The instance variable `age` is marked as `private` and is not directly accessible from the sub-class `Lion`. Therefore, the following would not compile:

```
public class Lion extends Animal {
   ...
   public void roar() {
      System.out.print("Lions age: "+age);  // DOES NOT COMPILE
   }
   ...
}
```

The `age` variable can be accessed indirectly through the `getAge()` and `setAge()` methods, which are marked as `public` in the `Animal` class.

The `name` variable can be accessed directly in the `Lion` class since it is marked `protected` in the `Animal` class.

Applying Class Access Modifiers

You already know that you can apply access modifiers to both methods and variables. It probably comes as little surprise that you can also apply access modifiers to class definitions, since we have been adding the `public` access modifier to most classes up to now.

In Java, a *top-level class* is a class that is not defined inside another class. Most of the classes in this book are top-level classes. They can only have `public` or package-private access. Applying `public` access to a class indicates that it can be referenced and used in any class. Applying default (package-private) access, which you'll remember is the lack of any access modifier, indicates the class can be accessed only by a class within the same package.

An *inner class* is a class defined inside of another class and is the opposite of a top-level class. In addition to `public` and package-private access, inner classes can also have `protected` and `private` access. We will discuss inner classes in Chapter 9.

As you might recall, a Java file can have many top-level classes but at most one `public` top-level class. In fact, it may have no `public` class at all. There's also no requirement that the single `public` class be the first class in the file. One benefit of using the package-private access is that you can define many classes within the same Java file. For example, the following definition could appear in a single Java file named `Groundhog.java`, since it contains only one `public` class:

```
class Rodent {}

public class Groundhog extends Rodent {}
```

If we were to update the `Rodent` class with the `public` access modifier, the `Groundhog.java` file would not compile unless the `Rodent` class was moved to its own `Rodent.java` file.

For simplicity, any time you see multiple `public` classes or interfaces defined in the same code sample in this book, assume each class is defined in its own Java file.

Accessing the *this* Reference

What happens when a method parameter has the same name as an existing instance variable? Let's take a look at an example. What do you think the following program prints?

```
public class Flamingo {
   private String color;
   public void setColor(String color) {
      color = color;
   }
   public static void main(String... unused) {
      Flamingo f = new Flamingo();
      f.setColor("PINK");
      System.out.println(f.color);
   }
}
```

If you said `null`, then you'd be correct. Java uses the most granular scope, so when it sees `color = color`, it thinks you are assigning the method parameter value to itself. The assignment completes successfully within the method, but the value of the instance variable `color` is never modified and is `null` when printed in the `main()` method.

The fix when you have a local variable with the same name as an instance variable is to use the `this` reference or keyword. The `this` reference refers to the current instance of the class and can be used to access any member of the class, including inherited members. It can be used in any instance method, constructor, and instance initializer block. It cannot be used when there is no implicit instance of the class, such as in a `static` method or `static` initializer block. We apply `this` to our previous method implementation as follows:

```
public void setColor(String color) {
   this.color = color;
}
```

The corrected code will now print `PINK` as expected. In many cases, the `this` reference is optional. If Java encounters a variable or method it cannot find, it will check the class hierarchy to see if it is available.

Now let's look at some examples that aren't common but that you might see on the exam.

```
1:  public class Duck {
2:     private String color;
3:     private int height;
4:     private int length;
5:
6:     public void setData(int length, int theHeight) {
7:        length = this.length;  // Backwards – no good!
8:        height = theHeight;    // Fine because a different name
9:        this.color = "white";  // Fine, but this. not necessary
```

```
10:    }
11:
12:    public static void main(String[] args) {
13:       Duck b = new Duck();
14:       b.setData(1,2);
15:       System.out.print(b.length + " " + b.height + " " + b.color);
16:    } }
```

This code compiles and prints the following:

```
0 2 white
```

This might not be what you expected, though. Line 7 is incorrect, and you should watch for it on the exam. The instance variable `length` starts out with a `0` value. That `0` is assigned to the method parameter `length`. The instance variable stays at `0`. Line 8 is more straightforward. The parameter `theHeight` and instance variable `height` have different names. Since there is no naming collision, `this` is not required. Finally, line 9 shows that a variable assignment is allowed to use `this` even when there is no duplication of variable names.

Calling the *super* Reference

In Java, a variable or method can be defined in both a parent class and a child class. When this happens, how do we reference the version in the parent class instead of the current class?

To achieve this, you can use the `super` reference or keyword. The `super` reference is similar to the `this` reference, except that it excludes any members found in the current class. In other words, the member must be accessible via inheritance. The following class shows how to apply `super` to use two variables with the same name in a method:

```
class Mammal {
   String type = "mammal";
}

public class Bat extends Mammal {
   String type = "bat";
   public String getType() {
      return super.type + ":" + this.type;
   }
   public static void main(String... zoo) {
      System.out.print(new Bat().getType());
   }
}
```

The program prints `mammal:bat`. What do you think would happen if the `super` reference was dropped? The program would then print `bat:bat`. Java uses the narrowest scope it can—in this case, the `type` variable defined in the `Bat` class. Note that the `this` reference in the previous example was optional, with the program printing the same output as it would if `this` was dropped.

Let's see if you've gotten the hang of `this` and `super`. What does the following program output?

```
1:  class Insect {
2:     protected int numberOfLegs = 4;
3:     String label = "buggy";
4:  }
5:
6:  public class Beetle extends Insect {
7:     protected int numberOfLegs = 6;
8:     short age = 3;
9:     public void printData() {
10:       System.out.print(this.label);
11:       System.out.print(super.label);
12:       System.out.print(this.age);
13:       System.out.print(super.age);
14:       System.out.print(numberOfLegs);
```

```
15:    }
16:    public static void main(String []n) {
17:       new Beetle().printData();
18:    }
19: }
```

That was a trick question—this program code would not compile! Let's review each line of the `printData()` method. Since `label` is defined in the parent class, it is accessible via both `this` and `super` references. For this reason, lines 10 and 11 compile and would both print `buggy` if the class compiled. On the other hand, the variable `age` is defined only in the current class, making it accessible via `this` but not `super`. For this reason, line 12 compiles, but line 13 does not. Remember, while `this` includes current and inherited members, `super` only includes inherited members. In this example, line 12 would print `3` if the code compiled.

Last but least, what would line 14 print if line 13 was commented out? Even though both `numberOfLegs` variables are accessible in `Beetle`, Java checks outward starting with the narrowest scope. For this reason, the value of `numberOfLegs` in the `Beetle` class is used and `6` would be printed. In this example, `this.numberOfLegs` and `super.numberOfLegs` refer to different variables with distinct values.

Since `this` includes inherited members, you often only use `super` when you have a naming conflict via inheritance. For example, you have a method or variable defined in the current class that matches a method or variable in a parent class. This commonly comes up in method overriding and variable hiding, which will be discussed later in this chapter.

Declaring Constructors

As you learned in Chapter 2, a constructor is a special method that matches the name of the class and has no return type. It is called when a new instance of the class is created. For the exam, you'll need to know a lot of rules about constructors. In this section, we'll show how to create a

constructor. Then, we'll look at default constructors, overloading constructors, calling parent constructors, `final` fields, and the order of initialization in a class.

Creating a Constructor

Let's start with a simple constructor:

```
public class Bunny {
   public Bunny() {
      System.out.println("constructor");
   }
}
```

The name of the constructor, `Bunny`, matches the name of the class, `Bunny`, and there is no return type, not even `void`. That makes this a constructor. Can you tell why these two are not valid constructors for the `Bunny` class?

```
public class Bunny {
   public bunny() { }     // DOES NOT COMPILE
   public void Bunny() { }
}
```

The first one doesn't match the class name because Java is case sensitive. Since it doesn't match, Java knows it can't be a constructor and is supposed to be a regular method. However, it is missing the return type and doesn't compile. The second method is a perfectly good method but is not a constructor because it has a return type.

Like method parameters, constructor parameters can be any valid class, array, or primitive type, including generics, but may not include `var`. The following does not compile:

```
class Bonobo {
   public Bonobo(var food) { // DOES NOT COMPILE
   }
}
```

A class can have multiple constructors, so long as each constructor has a unique signature. In this case, that means the constructor parameters must be distinct. Like methods with the same name but different signatures, declaring multiple constructors with different signatures is referred to as *constructor overloading*. The following `Turtle` class has four distinct overloaded constructors:

```
public class Turtle {
   private String name;
   public Turtle() {
      name = "John Doe";
   }
   public Turtle(int age) {}
   public Turtle(long age) {}
   public Turtle(String newName, String... favoriteFoods) {
      name = newName;
   }
}
```

Constructors are used when creating a new object. This process is called *instantiation* because it creates a new instance of the class. A constructor is called when we write `new` followed by the name of the class we want to instantiate. Here's an example:

```
new Turtle()
```

When Java sees the `new` keyword, it allocates memory for the new object. It then looks for a constructor with a matching signature and calls it.

Default Constructor

Every class in Java has a constructor whether you code one or not. If you don't include any constructors in the class, Java will create one for you without any parameters. This Java-created constructor is called the *default constructor* and is added anytime a class is declared without any constructors. We often refer to it as the default no-argument constructor for clarity. Here's an example:

```
public class Rabbit {
   public static void main(String[] args) {
      Rabbit rabbit = new Rabbit();      // Calls default constructor
   }
}
```

In the `Rabbit` class, Java sees no constructor was coded and creates one. This default constructor is equivalent to typing this:

```
   public Rabbit() {}
```

The default constructor has an empty parameter list and an empty body. It is fine for you to type this in yourself. However, since it doesn't do anything, Java is happy to generate it for you and save you some typing.

We keep saying *generated*. This happens during the compile step. If you look at the file with the `.java` extension, the constructor will still be missing. It is only in the compiled file with the `.class` extension that it makes an appearance.

Remember that a default constructor is only supplied if there are no constructors present. Which of these classes do you think has a default constructor?

```
public class Rabbit1 {}

public class Rabbit2 {
```

```
   public Rabbit2() {}
}

public class Rabbit3 {
   public Rabbit3(boolean b) {}
}

public class Rabbit4 {
   private Rabbit4() {}
}
```

Only `Rabbit1` gets a default no-argument constructor. It doesn't have a constructor coded, so Java generates a default no-argument constructor. `Rabbit2` and `Rabbit3` both have `public` constructors already. `Rabbit4` has a `private` constructor. Since these three classes have a constructor defined, the default no-argument constructor is not inserted for you.

Let's take a quick look at how to call these constructors:

```
1: public class RabbitsMultiply {
2:    public static void main(String[] args) {
3:       Rabbit1 r1 = new Rabbit1();
4:       Rabbit2 r2 = new Rabbit2();
5:       Rabbit3 r3 = new Rabbit3(true);
6:       Rabbit4 r4 = new Rabbit4();  // DOES NOT COMPILE
7:    } }
```

Line 3 calls the generated default no-argument constructor. Lines 4 and 5 call the user-provided constructors. Line 6 does not compile. `Rabbit4` made the constructor `private` so that other classes could not call it.

Having only `private` constructors in a class tells the compiler not to provide a default no-argument constructor. It also prevents other classes from instantiating the class. This is useful when a class has only `static` methods or the developer wants to have full control of all calls to create new instances of the class. Remember, `static` methods in the class, including a `main()` method, may access `private` members, including `private` constructors.

Calling Overloaded Constructors with *this()*

Remember, a single class can have multiple constructors. This is referred to as constructor overloading because all constructors have the same inherent name but a different signature. Let's take a look at this in more detail using a `Hamster` class.

```
public class Hamster {
   private String color;
   private int weight;
   public Hamster(int weight) {                 // First constructor
      this.weight = weight;
      color = "brown";
   }
   public Hamster(int weight, String color) {   // Second constructor
      this.weight = weight;
      this.color = color;
   }
}
```

One of the constructors takes a single `int` parameter. The other takes an `int` and a `String` . These parameter lists are different, so the constructors are successfully overloaded. There is a problem here, though.

There is a bit of duplication, as `this.weight` is assigned twice in the same way in both constructors. In programming, even a bit of duplication tends to turn into a lot of duplication as we keep adding "just one more thing." For example, imagine we had 20 variables being set like `this.weight`, rather than just one. What we really want is for the first constructor to call the second constructor with two parameters. So, how can you have a constructor call another constructor? You might be tempted to write this:

```
public Hamster(int weight) {
   Hamster(weight, "brown");      // DOES NOT COMPILE
}
```

This will not work. Constructors can be called only by writing `new` before the name of the constructor. They are not like normal methods that you can just call. What happens if we stick `new` before the constructor name?

```
public Hamster(int weight) {
   new Hamster(weight, "brown");  // Compiles, but incorrect
}
```

This attempt does compile. It doesn't do what we want, though. When this constructor is called, it creates a new object with the default `weight` and `color`. It then constructs a different object with the desired `weight` and `color` and ignores the new object. In this manner, we end up with two objects, with one being discarded after it is created. That's not what we want. We want `weight` and `color` set on the object we are trying to instantiate in the first place.

Java provides a solution: `this()`—yes, the same keyword we used to refer to instance members. When `this()` is used with parentheses, Java calls another constructor on the same instance of the class.

```
   public Hamster(int weight) {
      this(weight, "brown");
   }
```

Success! Now Java calls the constructor that takes two parameters, with `weight` and `color` set as expected.

Calling `this()` has one special rule you need to know. If you choose to call it, the `this()` call must be the first statement in the constructor. The side effect of this is that there can be only one call to `this()` in any constructor.

```
3:    public Hamster(int weight) {
4:       System.out.println("in constructor");
5:       // Set weight and default color
6:       this(weight, "brown");     // DOES NOT COMPILE
7:    }
```

Even though a print statement on line 4 doesn't change any variables, it is still a Java statement and is not allowed to be inserted before the call to `this()`. The comment on line 5 is just fine. Comments aren't considered statements and are allowed anywhere.

There's one last rule for overloaded constructors you should be aware of. Consider the following definition of the `Gopher` class:

```
public class Gopher {
   public Gopher(int dugHoles) {
      this(5);  // DOES NOT COMPILE
   }
}
```

The compiler is capable of detecting that this constructor is calling itself infinitely. Since this code can never terminate, the compiler stops and reports this as an error. Likewise, this also does not compile:

```
public class Gopher {
   public Gopher() {
      this(5);  // DOES NOT COMPILE
   }
   public Gopher(int dugHoles) {
      this();   // DOES NOT COMPILE
   }
}
```

In this example, the constructors call each other, and the process continues infinitely. Since the compiler can detect this, it reports this as an error.

THIS* VS. *THIS()

Despite using the same keyword, `this` and `this()` are very different. The first, `this`, refers to an instance of the class, while the second, `this()`, refers to a constructor call within the class. The exam may try to trick you by using both together, so make sure you know which one to use and why.

Calling Parent Constructors with *super()*

In Java, the first statement of every constructor is either a call to another constructor within the class, using `this()`, or a call to a constructor in the direct parent class, using `super()`. If a parent constructor takes arguments, then the `super()` call also takes arguments. For simplicity in this section, we refer to the `super()` command as any parent constructor, even those that take arguments. Let's take a look at the `Animal` class and its subclass `Zebra` and see how their constructors can be properly written to call one another:

```
public class Animal {
   private int age;
```

```
   public Animal(int age) {
      super();     // Refers to constructor in java.lang.Object
      this.age = age;
   }
}

public class Zebra extends Animal {
   public Zebra(int age) {
      super(age);  // Refers to constructor in Animal
   }
   public Zebra() {
      this(4);     // Refers to constructor in Zebra with int argument
   }
}
```

In the first class, `Animal`, the first statement of the constructor is a call to the parent constructor defined in `java.lang.Object`, which takes no arguments. In the second class, `Zebra`, the first statement of the first constructor is a call to `Animal`'s constructor, which takes a single argument. The class `Zebra` also includes a second no-argument constructor that doesn't call `super()` but instead calls the other constructor within the `Zebra` class using `this(4)`.

Like calling `this()`, calling `super()` can only be used as the first statement of the constructor. For example, the following two class definitions will not compile:

```
public class Zoo {
   public Zoo() {
      System.out.println("Zoo created");
      super();     // DOES NOT COMPILE
   }
}

public class Zoo {
   public Zoo() {
      super();
      System.out.println("Zoo created");
```

```
      super();     // DOES NOT COMPILE
   }
}
```

The first class will not compile because the call to the parent constructor must be the first statement of the constructor. In the second code snippet, `super()` is the first statement of the constructor, but it is also used as the third statement. Since `super()` can only be called once as the first statement of the constructor, the code will not compile.

If the parent class has more than one constructor, the child class may use any valid parent constructor in its definition, as shown in the following example:

```
public class Animal {
   private int age;
   private String name;
   public Animal(int age, String name) {
      super();
      this.age = age;
      this.name = name;
   }
   public Animal(int age) {
      super();
      this.age = age;
      this.name = null;
   }
}

public class Gorilla extends Animal {
   public Gorilla(int age) {
      super(age,"Gorilla");
   }
   public Gorilla() {
      super(5);
   }
}
```

In this example, the first child constructor takes one argument, `age`, and calls the parent constructor, which takes two arguments, `age` and `name`. The second child constructor takes no arguments, and it calls the parent constructor, which takes one argument, `age`. In this example, notice that the child constructors are not required to call matching parent constructors. Any valid parent constructor is acceptable as long as the appropriate input parameters to the parent constructor are provided.

SUPER* VS. *SUPER()

Like `this` and `this()`, `super` and `super()` are unrelated in Java. The first, `super`, is used to reference members of the parent class, while the second, `super()`, calls a parent constructor. Anytime you see the `super` keyword on the exam, make sure it is being used properly.

Understanding Compiler Enhancements

Wait a second, we said the first line of every constructor is a call to either `this()` or `super()`, but we've been creating classes and constructors throughout this book, and we've rarely done either. How did these classes compile? The answer is that the Java compiler automatically inserts a call to the no-argument constructor `super()` if you do not explicitly call `this()` or `super()` as the first line of a constructor. For example, the following three class and constructor definitions are equivalent, because the compiler will automatically convert them all to the last example:

```
public class Donkey {}

public class Donkey {
   public Donkey() {}
}

public class Donkey {
```

```
   public Donkey() {
      super();
   }
}
```

Make sure you understand the differences between these three `Donkey` class definitions and why Java will automatically convert them all to the last definition. Keep the process that the Java compiler performs in mind while reading the next section.

ARE CLASSES WITH ONLY *PRIVATE* CONSTRUCTORS CONSIDERED *FINAL*?

Remember, a `final` class cannot be extended. What happens if you have a class that is not marked `final` but only contains `private` constructors—can you extend the class? The answer is "yes," but only an inner class defined in the class itself can extend it. An inner class is the only one that would have access to a `private` constructor and be able to call `super()`. Other top-level classes cannot extend such a class. Don't worry—knowing this fact is not required for the exam. We include it here for those who were curious about declaring only `private` constructors.

Missing a Default No-Argument Constructor

What happens if the parent class doesn't have a no-argument constructor? Recall that the default no-argument constructor is not required and is inserted by the compiler only if there is no constructor defined in the class. For example, do you see why the following `Elephant` class declaration does not compile?

```
public class Mammal {
   public Mammal(int age) {}
}
```

```
public class Elephant extends Mammal {  // DOES NOT COMPILE
}
```

Since `Elephant` does not define any constructors, the Java compiler will attempt to insert a default no-argument constructor. As a second compile-time enhancement, it will also auto-insert a call to `super()` as the first line of the default no-argument constructor. Our previous `Elephant` declaration is then converted by the compiler to the following declaration:

```
public class Elephant extends Mammal {
   public Elephant() {
      super();  // DOES NOT COMPILE
   }
}
```

Since the `Mammal` class has at least one constructor declared, the compiler does not insert a default no-argument constructor. Therefore, the `super()` call in the `Elephant` class declaration does not compile. In this case, the Java compiler will not help, and you must create at least one constructor in your child class that explicitly calls a parent constructor via the `super()` command. We can fix this by adding a call to a parent constructor that takes a fixed argument.

```
public class Elephant extends Mammal {
   public Elephant() {
      super(10);
   }
}
```

This code will compile because we have added a constructor with an explicit call to a parent constructor. Notice that the class `Elephant` now has a no-argument constructor even though its parent class `Mammal` doesn't. Subclasses may define explicit no-argument constructors even if their parent classes do not, provided the constructor of the child maps to a parent constructor via an explicit call of the `super()` command. This means

that subclasses of the `Elephant` can rely on compiler enhancements. For example, the following class compiles because `Elephant` now has a no-argument constructor, albeit one defined explicitly:

```
public class AfricanElephant extends Elephant {}
```

You should be wary of any exam question in which a class defines a constructor that takes arguments and doesn't define a no-argument constructor. Be sure to check that the code compiles before answering a question about it, especially if any classes inherit it. For the exam, you should be able to spot right away why classes such as our first `Elephant` implementation did not compile.

***SUPER()* ALWAYS REFERS TO THE MOST DIRECT PARENT**

A class may have multiple ancestors via inheritance. In our previous example, `AfricanElephant` is a subclass of `Elephant`, which in turn is a subclass of `Mammal`. For constructors, though, `super()` always refers to the most direct parent. In this example, calling `super()` inside the `AfricanElephant` class always refers to the `Elephant` class, and never the `Mammal` class.

Constructors and *final* Fields

As you might recall from Chapter 7, `final static` variables must be assigned a value exactly once. You saw this happen in the line of the declaration and in a `static` initializer. Instance variables marked `final` follow similar rules. They can be assigned values in the line in which they are declared or in an instance initializer.

```
public class MouseHouse {
   private final int volume;
```

```
   private final String name = "The Mouse House";
   {
      volume = 10;
   }
}
```

Like other `final` variables, once the value is assigned, it cannot be changed. There is one more place they can be assigned a value—the constructor. The constructor is part of the initialization process, so it is allowed to assign `final` instance variables in it. For the exam, you need to know one important rule. By the time the constructor completes, all `final` instance variables must be assigned a value. Let's try this out in an example:

```
public class MouseHouse {
   private final int volume;
   private final String type;
   public MouseHouse() {
      this.volume = 10;
      type = "happy";
   }
}
```

In our `MouseHouse` implementation, the values for `volume` and `type` are assigned in the constructor. Remember that the `this` keyword is optional since the instance variables are part of the class declaration, and there are no constructor parameters with the same name.

Unlike local `final` variables, which are not required to have a value unless they are actually used, `final` instance variables *must* be assigned a value. Default values are not used for these variables. If they are not assigned a value in the line where they are declared or in an instance initializer, then they must be assigned a value in the constructor declaration. Failure to do so will result in a compiler error on the line that declares the constructor.

```
public class MouseHouse {
   private final int volume;
   private final String type;
   {
      this.volume = 10;
   }
   public MouseHouse(String type) {
      this.type = type;
   }
   public MouseHouse() {  // DOES NOT COMPILE
      this.volume = 2;    // DOES NOT COMPILE
   }
}
```

In this example, the first constructor that takes a `String` argument compiles. Although a `final` instance variable can be assigned a value only once, each constructor is considered independently in terms of assignment. The second constructor does not compile for two reasons. First, the constructor fails to set a value for the `type` variable. The compiler detects that a value is never set for `type` and reports an error on the line where the constructor is declared. Second, the constructor sets a value for the `volume` variable, even though it was already assigned a value by the instance initializer. The compiler reports this error on the line where `volume` is set.

On the exam, be wary of any instance variables marked `final` . Make sure they are assigned a value in the line where they are declared, in an instance initializer, or in a constructor. They should be assigned a value only once, and failure to assign a value is considered a compiler error in the constructor.

What about `final` instance variables when a constructor calls another constructor in the same class? In that case, you have to follow the constructor logic pathway carefully, making sure every `final` instance variable is assigned a value exactly once. We can replace our previous bad constructor with the following one that does compile:

```
public MouseHouse() {
   this(null);
}
```

This constructor does not perform any assignments to any `final` instance variables, but it calls the `MouseHouse(String)` constructor, which we observed compiles without issue. We use `null` here to demonstrate that the variable does not need to be an object value. We can assign a `null` value to `final` instance variables, so long as they are explicitly set.

Order of Initialization

In Chapter 2, we presented the order of initialization. With inheritance, though, the order of initialization for an instance gets a bit more complicated. We'll start with how to initialize the class and then expand to initializing the instance.

Class Initialization

First, you need to initialize the class, which involves invoking all `static` members in the class hierarchy, starting with the highest superclass and working downward. This is often referred to as loading the class. The JVM controls when the class is initialized, although you can assume the class is loaded before it is used. The class may be initialized when the program first starts, when a `static` member of the class is referenced, or shortly before an instance of the class is created.

The most important rule with class initialization is that it happens at most once for each class. The class may also never be loaded if it is not used in

the program. We summarize the order of initialization for a class as follows:

Initialize Class X

1. If there is a superclass Y of X, then initialize class Y first.
2. Process all `static` variable declarations in the order they appear in the class.
3. Process all `static` initializers in the order they appear in the class.

Taking a look at an example, what does the following program print?

```
public class Animal {
   static { System.out.print("A"); }
}

public class Hippo extends Animal {
   static { System.out.print("B"); }
   public static void main(String[] grass) {
      System.out.print("C");
      new Hippo();
      new Hippo();
      new Hippo();
   }
}
```

It prints `ABC` exactly once. Since the `main()` method is inside the `Hippo` class, the class will be initialized first, starting with the superclass and printing `AB`. Afterward, the `main()` method is executed, printing `C`. Even though the `main()` method creates three instances, the class is loaded only once.

WHY THE HIPPO PROGRAM PRINTED *C* AFTER *AB*

In the previous example, the `Hippo` class was initialized before the `main()` method was executed. This happened because our `main()` method was inside the class being executed, so it had to be loaded on startup. What if you instead called `Hippo` inside another program?

```
public class HippoFriend {
   public static void main(String[] grass) {
      System.out.print("C");
      new Hippo();
   }
}
```

Assuming the class isn't referenced anywhere else, this program will likely print `CAB`, with the `Hippo` class not being loaded until it is needed inside the `main()` method. We say *likely*, because the rules for when classes are loaded are determined by the JVM at runtime. For the exam, you just need to know that a class must be initialized before it is referenced or used. Also, the class containing the program entry point, aka the `main()` method, is loaded before the `main()` method is executed.

Instance Initialization

An instance is initialized anytime the `new` keyword is used. In our previous example, there were three `new Hippo()` calls, resulting in three `Hippo` instances being initialized. Instance initialization is a bit more complicated than class initialization, because a class or superclass may have many constructors declared but only a handful used as part of instance initialization.

First, start at the lowest-level constructor where the `new` keyword is used. Remember, the first line of every constructor is a call to `this()` or

`super()`, and if omitted, the compiler will automatically insert a call to the parent no-argument constructor `super()`. Then, progress upward and note the order of constructors. Finally, initialize each class starting with the superclass, processing each instance initializer and constructor in the reverse order in which it was called. We summarize the order of initialization for an instance as follows:

Initialize Instance of X

1. If there is a superclass Y of X, then initialize the instance of Y first.
2. Process all instance variable declarations in the order they appear in the class.
3. Process all instance initializers in the order they appear in the class.
4. Initialize the constructor including any overloaded constructors referenced with `this()`.

Let's try a simple example with no inheritance. See if you can figure out what the following application outputs:

```
1:  public class ZooTickets {
2:     private String name = "BestZoo";
3:     { System.out.print(name+"-"); }
4:     private static int COUNT = 0;
5:     static { System.out.print(COUNT+"-"); }
6:     static { COUNT += 10; System.out.print(COUNT+"-"); }
7:
8:     public ZooTickets() {
9:        System.out.print("z-");
10:    }
11:
12:    public static void main(String... patrons) {
13:       new ZooTickets();
14:    }
15: }
```

The output is as follows:

```
0-10-BestZoo-z-
```

First, we have to initialize the class. Since there is no superclass declared, which means the superclass is `Object`, we can start with the `static` components of `ZooTickets`. In this case, lines 4, 5, and 6 are executed, printing `0-` and `10-`. Next, we initialize the instance. Again, since there is no superclass declared, we start with the instance components. Lines 2 and 3 are executed, which prints `BestZoo-`. Finally, we run the constructor on lines 8–10, which outputs `z-`.

Next, let's try a simple example with inheritance.

```
class Primate {
   public Primate() {
      System.out.print("Primate-");
   }
}

class Ape extends Primate {
   public Ape(int fur) {
      System.out.print("Ape1-");
   }
   public Ape() {
      System.out.print("Ape2-");
   }
}

public class Chimpanzee extends Ape {
   public Chimpanzee() {
      super(2);
      System.out.print("Chimpanzee-");
   }
   public static void main(String[] args) {
      new Chimpanzee();
   }
}
```

The compiler inserts the `super()` command as the first statement of both the `Primate` and `Ape` constructors. The code will execute with the parent constructors called first and yields the following output:

```
Primate-Ape1-Chimpanzee-
```

Notice that only one of the two `Ape()` constructors is called. You need to start with the call to `new Chimpanzee()` to determine which constructors will be executed. Remember, constructors are executed from the bottom up, but since the first line of every constructor is a call to another constructor, the flow actually ends up with the parent constructor executed before the child constructor.

The next example is a little harder. What do you think happens here?

```
1:  public class Cuttlefish {
2:     private String name = "swimmy";
3:     { System.out.println(name); }
4:     private static int COUNT = 0;
5:     static { System.out.println(COUNT); }
6:     { COUNT++; System.out.println(COUNT); }
7:
8:     public Cuttlefish() {
9:        System.out.println("Constructor");
10:    }
11:
12:    public static void main(String[] args) {
13:       System.out.println("Ready");
14:       new Cuttlefish();
15:    }
16: }
```

The output looks like this:

```
0
Ready
```

```
swimmy
1
Constructor
```

There is no superclass declared, so we can skip any steps that relate to inheritance. We first process the `static` variables and `static` initializers—lines 4 and 5, with line 5 printing `0`. Now that the `static` initializers are out of the way, the `main()` method can run, which prints `Ready`. Lines 2, 3, and 6 are processed, with line 3 printing `swimmy` and line 6 printing `1`. Finally, the constructor is run on lines 8–10, which print `Constructor`.

Ready for a more difficult example? What does the following output?

```
1:  class GiraffeFamily {
2:     static { System.out.print("A"); }
3:     { System.out.print("B"); }
4:
5:     public GiraffeFamily(String name) {
6:        this(1);
7:        System.out.print("C");
8:     }
9:
10:    public GiraffeFamily() {
11:       System.out.print("D");
12:    }
13:
14:    public GiraffeFamily(int stripes) {
15:       System.out.print("E");
16:    }
17: }
18: public class Okapi extends GiraffeFamily {
19:    static { System.out.print("F"); }
20:
21:    public Okapi(int stripes) {
22:       super("sugar");
23:       System.out.print("G");
```

```
24:    }
25:    { System.out.print("H"); }
26:
27:    public static void main(String[] grass) {
28:       new Okapi(1);
29:       System.out.println();
30:       new Okapi(2);
31:    }
32: }
```

The program prints the following:

```
AFBECHG
BECHG
```

Let's walk through it. Start with initializing the `Okapi` class. Since it has a superclass `GiraffeFamily`, initialize it first, printing `A` on line 2. Next, initialize the `Okapi` class, printing `F` on line 19.

After the classes are initialized, execute the `main()` method on line 27. The first line of the `main()` method creates a new `Okapi` object, triggering the instance initialization process. Per the first rule, the superclass instance of `GiraffeFamily` is initialized first. Per our third rule, the instance initializer in the superclass `GiraffeFamily` is called, and `B` is printed on line 3. Per the fourth rule, we initialize the constructors. In this case, this involves calling the constructor on line 5, which in turn calls the overloaded constructor on line 14. The result is that `EC` is printed, as the constructor bodies are unwound in the reverse order that they were called.

The process then continues with the initialization of the `Okapi` instance itself. Per the third and fourth rules, `H` is printed on line 25, and `G` is printed on line 23, respectively. The process is a lot simpler when you don't have to call any overloaded constructors. Line 29 then inserts a line break in the output. Finally, line 30 initializes a new `Okapi` object. The order and initialization are the same as line 28, sans the class initialization,

so `BECHG` is printed again. Notice that `D` is never printed, as only two of the three constructors in the superclass `GiraffeFamily` are called.

This example is tricky for a few reasons. There are multiple overloaded constructors, lots of initializers, and a complex constructor pathway to keep track of. Luckily, questions like this are rare on the exam. If you see one, just write down what is going on as you read the code.

Reviewing Constructor Rules

Let's review some of the most important constructor rules that we covered in this part of the chapter.

1. The first statement of every constructor is a call to an overloaded constructor via `this()`, or a direct parent constructor via `super()`.
2. If the first statement of a constructor is not a call to `this()` or `super()`, then the compiler will insert a no-argument `super()` as the first statement of the constructor.
3. Calling `this()` and `super()` after the first statement of a constructor results in a compiler error.
4. If the parent class doesn't have a no-argument constructor, then every constructor in the child class must start with an explicit `this()` or `super()` constructor call.
5. If the parent class doesn't have a no-argument constructor and the child doesn't define any constructors, then the child class will not compile.
6. If a class only defines `private` constructors, then it cannot be extended by a top-level class.
7. All `final` instance variables must be assigned a value exactly once by the end of the constructor. Any `final` instance variables not assigned a value will be reported as a compiler error on the line the constructor is declared.

Make sure you understand these rules. The exam will often provide code that breaks one or many of these rules and therefore doesn't compile.

When taking the exam, pay close attention to any question involving two or more classes related by inheritance. Before even attempting to answer the question, you should check that the constructors are properly defined using the previous set of rules. You should also verify the classes include valid access modifiers for members. Once those are verified, you can continue answering the question.

Inheriting Members

Now that we've created a class, what can we do with it? One of Java's biggest strengths is leveraging its inheritance model to simplify code. For example, let's say you have five animal classes that each extend from the `Animal` class. Furthermore, each class defines an `eat()` method with identical implementations. In this scenario, it's a lot better to define `eat()` once in the `Animal` class with the proper access modifiers than to have to maintain the same method in five separate classes. As you'll also see in this section, Java allows any of the five subclasses to replace, or override, the parent method implementation at runtime.

Calling Inherited Members

Java classes may use any `public` or `protected` member of the parent class, including methods, primitives, or object references. If the parent class and child class are part of the same package, then the child class may also use any package-private members defined in the parent class. Finally, a child class may never access a `private` member of the parent class, at least not through any direct reference. As you saw earlier in this chapter, a `private` member `age` was accessed indirectly via a `public` or `protected` method.

To reference a member in a parent class, you can just call it directly, as in the following example with the output function `displaySharkDetails()`:

```
class Fish {
   protected int size;
   private int age;

   public Fish(int age) {
      this.age = age;
   }

   public int getAge() {
      return age;
   }
}

public class Shark extends Fish {
   private int numberOfFins = 8;

   public Shark(int age) {
      super(age);
      this.size = 4;
   }

   public void displaySharkDetails() {
      System.out.print("Shark with age: "+getAge());
      System.out.print(" and "+size+" meters long");
      System.out.print(" with "+numberOfFins+" fins");
   }
}
```

In the child class, we use the `public` method `getAge()` and `protected` member `size` to access values in the parent class. Remember, you can use `this` to access visible members of the current or a parent class, and you can use `super` to access visible members of a parent class.

```
   public void displaySharkDetails() {
      System.out.print("Shark with age: "+super.getAge());
```

```
        System.out.print(" and "+super.size+" meters long");
        System.out.print(" with "+this.numberOfFins+" fins");
    }
```

In this example, `getAge()` and `size` can be accessed with `this` or `super` since they are defined in the parent class, while `numberOfFins` can only be accessed with `this` and not `super` since it is not an inherited property.

Inheriting Methods

Inheriting a class not only grants access to inherited methods in the parent class but also sets the stage for collisions between methods defined in both the parent class and the subclass. In this section, we'll review the rules for method inheritance and how Java handles such scenarios.

Overriding a Method

What if there is a method defined in both the parent and child classes with the same signature? For example, you may want to define a new version of the method and have it behave differently for that subclass. The solution is to override the method in the child class. In Java, *overriding* a method occurs when a subclass declares a new implementation for an inherited method with the same signature and compatible return type. Remember that a method signature includes the name of the method and method parameters.

When you override a method, you may reference the parent version of the method using the `super` keyword. In this manner, the keywords `this` and `super` allow you to select between the current and parent versions of a method, respectively. We illustrate this with the following example:

```
public class Canine {
```

```
    public double getAverageWeight() {
        return 50;
    }
}
public class Wolf extends Canine {
    public double getAverageWeight() {
        return super.getAverageWeight()+20;
    }
    public static void main(String[] args) {
        System.out.println(new Canine().getAverageWeight());
        System.out.println(new Wolf().getAverageWeight());
    }
}
```

In this example, in which the child class `Wolf` overrides the parent class `Canine`, the method `getAverageWeight()` runs, and the program displays the following:

```
50.0
70.0
```

METHOD OVERRIDING AND RECURSIVE CALLS

You might be wondering whether the use of `super` in the previous example was required. For example, what would the following code output if we removed the `super` keyword?

```
public double getAverageWeight() {
   return getAverageWeight()+20;  // StackOverflowError
}
```

In this example, the compiler would not call the parent `Canine` method; it would call the current `Wolf` method since it would think you were executing a recursive method call. A *recursive method* is one that calls itself as part of execution. It is common in programming but must have a termination condition that triggers the end to recursion at some point or depth. In this example, there is no termination condition; therefore, the application will attempt to call itself infinitely and produce a `StackOverflowError` at runtime.

To override a method, you must follow a number of rules. The compiler performs the following checks when you override a method:

1. The method in the child class must have the same signature as the method in the parent class.
2. The method in the child class must be at least as accessible as the method in the parent class.
3. The method in the child class may not declare a checked exception that is new or broader than the class of any exception declared in the parent class method.
4. If the method returns a value, it must be the same or a subtype of the method in the parent class, known as *covariant return types*.

DEFINING SUBTYPE AND SUPERTYPE

When discussing inheritance and polymorphism, we often use the word *subtype* rather than *subclass*, since Java includes interfaces. A *subtype* is the relationship between two types where one type inherits the other. If we define X to be a subtype of Y, then one of the following is true:

- X and Y are classes, and X is a subclass of Y.
- X and Y are interfaces, and X is a subinterface of Y.
- X is a class and Y is an interface, and X implements Y (either directly or through an inherited class).

Likewise, a *supertype* is the reciprocal relationship between two types where one type is the ancestor of the other. Remember, a subclass is a subtype, but not all subtypes are subclasses.

The first rule of overriding a method is somewhat self-explanatory. If two methods have the same name but different signatures, the methods are overloaded, not overridden. Overloaded methods are considered independent and do not share the same polymorphic properties as overridden methods.

OVERLOADING VS. OVERRIDING

Overloading and overriding a method are similar in that they both involve redefining a method using the same name. They differ in that an overloaded method will use a different list of method parameters. This distinction allows overloaded methods a great deal more freedom in syntax than an overridden method would have. For example, compare the overloaded `fly()` with the overridden `eat()` in the `Eagle` class.

```
public class Bird {
   public void fly() {
      System.out.println("Bird is flying");
   }
   public void eat(int food) {
      System.out.println("Bird is eating "+food+" units of food");
   }
}

public class Eagle extends Bird {
   public int fly(int height) {
      System.out.println("Bird is flying at "+height+" meters");
      return height;
   }
   public int eat(int food) {  // DOES NOT COMPILE
      System.out.println("Bird is eating "+food+" units of food");
      return food;
   }
}
```

The `fly()` method is overloaded in the subclass `Eagle`, since the signature changes from a no-argument method to a method with one `int` argument. Because the method is being overloaded and not overridden, the return type can be changed from `void` to `int`.

The `eat()` method is overridden in the subclass `Eagle`, since the signature is the same as it is in the parent class `Bird`—they both take a single argument `int`. Because the method is being overridden, the re-

turn type of the method in the `Eagle` class must be compatible with the return type for the method in the `Bird` class. In this example, the return type `int` is not a subtype of `void`; therefore, the compiler will throw an exception on this method definition.

Any time you see a method on the exam with the same name as a method in the parent class, determine whether the method is being overloaded or overridden first; doing so will help you with questions about whether the code will compile.

What's the purpose of the second rule about access modifiers? Let's try an illustrative example:

```
public class Camel {
   public int getNumberOfHumps() {
      return 1;
   }
}

public class BactrianCamel extends Camel {
   private int getNumberOfHumps() {  // DOES NOT COMPILE
      return 2;
   }
}

public class Rider {
   public static void main(String[] args) {
      Camel c = new BactrianCamel();
      System.out.print(c.getNumberOfHumps());
   }
}
```

In this example, `BactrianCamel` attempts to override the `getNumberOfHumps()` method defined in the parent class but fails because the access modifier `private` is more restrictive than the one defined in the

parent version of the method. Let's say `BactrianCamel` was allowed to compile, though. Would the call to `getNumberOfHumps()` in `Rider.main()` succeed or fail? As you will see when we get into polymorphism later in this chapter, the answer is quite ambiguous. The reference type for the object is `Camel`, where the method is declared `public`, but the object is actually an instance of type `BactrianCamel`, which is declared `private`. Java avoids these types of ambiguity problems by limiting overriding a method to access modifiers that are as accessible or more accessible than the version in the inherited method.

The third rule says that overriding a method cannot declare new checked exceptions or checked exceptions broader than the inherited method. This is done for similar polymorphic reasons as limiting access modifiers. In other words, you could end up with an object that is more restrictive than the reference type it is assigned to, resulting in a checked exception that is not handled or declared. We will discuss what it means for an exception to be checked in Chapter 10, "Exceptions." For now, you should just recognize that if a broader checked exception is declared in the overriding method, the code will not compile. Let's try an example:

```
public class Reptile {
   protected void sleepInShell() throws IOException {}

   protected void hideInShell() throws NumberFormatException {}

   protected void exitShell() throws FileNotFoundException {}
}

public class GalapagosTortoise extends Reptile {
   public void sleepInShell() throws FileNotFoundException {}

   public void hideInShell() throws IllegalArgumentException {}

   public void exitShell() throws IOException {} // DOES NOT COMPILE
}
```

In this example, we have three overridden methods. These overridden methods use the more accessible `public` modifier, which is allowed per our second rule over overridden methods. The overridden `sleepInShell()` method declares `FileNotFoundException`, which is a subclass of the exception declared in the inherited method, `IOException`. Per our third rule of overridden methods, this is a successful override since the exception is narrower in the overridden method.

The overridden `hideInShell()` method declares an `IllegalArgumentException`, which is a superclass of the exception declared in the inherited method, `NumberFormatException`. While this seems like an invalid override since the overridden method uses a broader exception, both of these exceptions are unchecked, so the third rule does not apply.

The third overridden `exitShell()` method declares `IOException`, which is a superclass of the exception declared in the inherited method, `FileNotFoundException`. Since these are checked exceptions and `IOException` is broader, the overridden `exitShell()` method does not compile in the `GalapagosTortoise` class. We'll revisit these exception classes, including memorizing which ones are subclasses of each other, in Chapter 10.

The fourth and final rule around overriding a method is probably the most complicated, as it requires knowing the relationships between the return types. The overriding method must use a return type that is covariant with the return type of the inherited method.

Let's try an example for illustrative purposes:

```
public class Rhino {
   protected CharSequence getName() {
      return "rhino";
   }
   protected String getColor() {
      return "grey, black, or white";
```

```
   }
}

class JavanRhino extends Rhino {
   public String getName() {
      return "javan rhino";
   }
   public CharSequence getColor() {  // DOES NOT COMPILE
      return "grey";
   }
}
```

The subclass `JavanRhino` attempts to override two methods from `Rhino`: `getName()` and `getColor()`. Both overridden methods have the same name and signature as the inherited methods. The overridden methods also have a broader access modifier, `public`, than the inherited methods. Per the second rule, a broader access modifier is acceptable.

From Chapter 5, you should already know that `String` implements the `CharSequence` interface, making `String` a subtype of `CharSequence`. Therefore, the return type of `getName()` in `JavanRhino` is covariant with the return type of `getName()` in `Rhino`.

On the other hand, the overridden `getColor()` method does not compile because `CharSequence` is not a subtype of `String`. To put it another way, all `String` values are `CharSequence` values, but not all `CharSequence` values are `String` values. For example, a `StringBuilder` is a `CharSequence` but not a `String`. For the exam, you need to know if the return type of the overriding method is the same or a subtype of the return type of the inherited method.

A simple test for covariance is the following: Given an inherited return type A and an overriding return type B, can you assign an instance of B to a reference variable for A without a cast? If so, then they are covariant. This rule applies to primitive types and object types alike. If one of the return types is `void`, then they both must be `void`, as nothing is covariant with `void` except itself.

The last three rules of overriding a method may seem arbitrary or confusing at first, but as you'll see later in this chapter when we discuss polymorphism, they are needed for consistency. Without these rules in place, it is possible to create contradictions within the Java language.

Overriding a Generic Method

Overriding methods is complicated enough, but add generics to it and things only get more challenging. In this section, we'll provide a discussion of the aspects of overriding generic methods that you'll need to know for the exam.

Review of Overloading a Generic Method

In Chapter 7, you learned that you cannot overload methods by changing the generic type due to type erasure. To review, only one of the two methods is allowed in a class because type erasure will reduce both sets of arguments to `(List input)`.

```
public class LongTailAnimal {
   protected void chew(List<Object> input) {}
   protected void chew(List<Double> input) {}  // DOES NOT COMPILE
}
```

For the same reason, you also can't overload a generic method in a parent class.

```
public class LongTailAnimal {
   protected void chew(List<Object> input) {}
}

public class Anteater extends LongTailAnimal {
   protected void chew(List<Double> input) {}  // DOES NOT COMPILE
}
```

Both of these examples fail to compile because of type erasure. In the compiled form, the generic type is dropped, and it appears as an invalid overloaded method.

Generic Method Parameters

On the other hand, you can override a method with generic parameters, but you must match the signature including the generic type exactly. For example, this version of the `Anteater` class does compile because it uses the same generic type in the overridden method as the one defined in the parent class:

```
public class LongTailAnimal {
   protected void chew(List<String> input) {}
}

public class Anteater extends LongTailAnimal {
   protected void chew(List<String> input) {}
}
```

The generic type parameters have to match, but what about the generic class or interface? Take a look at the following example. From what you know so far, do you think these classes will compile?

```
public class LongTailAnimal {
   protected void chew(List<Object> input) {}
}

public class Anteater extends LongTailAnimal {
   protected void chew(ArrayList<Double> input) {}
}
```

Yes, these classes do compile. However, they are considered overloaded methods, not overridden methods, because the signature is not the same. Type erasure does not change the fact that one of the method arguments is a `List` and the other is an `ArrayList`.

GENERICS AND WILDCARDS

Java includes support for generic wildcards using the question mark (`?`) character. It even supports bounded wildcards.

```
void sing1(List<?> v) {}                  // unbounded wildcard
void sing2(List<? super String> v) {}     // lower bounded wildcard
void sing3(List<? extends String> v) {}  // upper bounded wildcard
```

Using generics with wildcards, overloaded methods, and overridden methods can get quite complicated. Luckily, wildcards are out of scope for the 1Z0-815 exam. They are required knowledge, though, when you take the 1Z0-816 exam.

Generic Return Types

When you're working with overridden methods that return generics, the return values must be covariant. In terms of generics, this means that the return type of the class or interface declared in the overriding method must be a subtype of the class defined in the parent class. The generic parameter type must match its parent's type exactly.

Given the following declaration for the `Mammal` class, which of the two subclasses, `Monkey` and `Goat`, compile?

```
public class Mammal {
   public List<CharSequence> play() { ... }
   public CharSequence sleep() { ... }
}

public class Monkey extends Mammal {
   public ArrayList<CharSequence> play() { ... }
}

public class Goat extends Mammal {
   public List<String> play() { ... }  // DOES NOT COMPILE
   public String sleep() { ... }
}
```

The `Monkey` class compiles because `ArrayList` is a subtype of `List`. The `play()` method in the `Goat` class does not compile, though. For the return types to be covariant, the generic type parameter must match. Even though `String` is a subtype of `CharSequence`, it does not exactly match the generic type defined in the `Mammal` class. Therefore, this is considered an invalid override.

Notice that the `sleep()` method in the `Goat` class does compile since `String` is a subtype of `CharSequence`. This example shows that covariance applies to the return type, just not the generic parameter type.

For the exam, it might be helpful for you to apply type erasure to questions involving generics to ensure that they compile properly. Once you've determined which methods are overridden and which are being overloaded, work backward, making sure the generic types match for overridden methods. And remember, generic methods cannot be overloaded by changing the generic parameter type only.

Redeclaring *private* Methods

What happens if you try to override a `private` method? In Java, you can't override `private` methods since they are not inherited. Just because a child class doesn't have access to the parent method doesn't mean the child class can't define its own version of the method. It just means, strictly speaking, that the new method is not an overridden version of the parent class's method.

Java permits you to redeclare a new method in the child class with the same or modified signature as the method in the parent class. This method in the child class is a separate and independent method, unrelated to the parent version's method, so none of the rules for overriding methods is invoked. Let's return to the `Camel` example we used in the previous section and show two related classes that define the same method:

```
public class Camel {
   private String getNumberOfHumps() {
      return "Undefined";
   }
}

public class DromedaryCamel extends Camel {
   private int getNumberOfHumps() {
      return 1;
   }
}
```

This code compiles without issue. Notice that the return type differs in the child method from `String` to `int`. In this example, the method `getNumberOfHumps()` in the parent class is redeclared, so the method in the child class is a new method and not an override of the method in the parent class. As you saw in the previous section, if the method in the parent class were `public` or `protected`, the method in the child class would not compile because it would violate two rules of overriding methods. The parent method in this example is `private`, so there are no such issues.

Hiding Static Methods

A *hidden method* occurs when a child class defines a `static` method with the same name and signature as an inherited `static` method defined in a parent class. Method hiding is similar but not exactly the same as method overriding. The previous four rules for overriding a method must be followed when a method is hidden. In addition, a new rule is added for hiding a method:

5. The method defined in the child class must be marked as `static` if it is marked as `static` in a parent class.

Put simply, it is method hiding if the two methods are marked `static`, and method overriding if they are not marked `static`. If one is marked `static` and the other is not, the class will not compile.

Let's review some examples of the new rule:

```
public class Bear {
   public static void eat() {
      System.out.println("Bear is eating");
   }
}

public class Panda extends Bear {
   public static void eat() {
      System.out.println("Panda is chewing");
   }
   public static void main(String[] args) {
      eat();
   }
}
```

In this example, the code compiles and runs. The `eat()` method in the `Panda` class hides the `eat()` method in the `Bear` class, printing `"Panda is chewing"` at runtime. Because they are both marked as `static`, this is

not considered an overridden method. That said, there is still some inheritance going on. If you remove the `eat()` method in the `Panda` class, then the program prints `"Bear is eating"` at runtime.

Let's contrast this with an example that violates the fifth rule:

```
public class Bear {
   public static void sneeze() {
      System.out.println("Bear is sneezing");
   }
   public void hibernate() {
      System.out.println("Bear is hibernating");
   }
   public static void laugh() {
      System.out.println("Bear is laughing");
   }
}

public class Panda extends Bear {
   public void sneeze() {           // DOES NOT COMPILE
      System.out.println("Panda sneezes quietly");
   }
   public static void hibernate() { // DOES NOT COMPILE
      System.out.println("Panda is going to sleep");
   }
   protected static void laugh() {  // DOES NOT COMPILE
      System.out.println("Panda is laughing");
   }
}
```

In this example, `sneeze()` is marked `static` in the parent class but not in the child class. The compiler detects that you're trying to override using an instance method. However, `sneeze()` is a `static` method that should be hidden, causing the compiler to generate an error. In the second method, `hibernate()` is an instance member in the parent class but a `static` method in the child class. In this scenario, the compiler thinks that you're trying to hide a `static` method. Because `hibernate()` is an

instance method that should be overridden, the compiler generates an error. Finally, the `laugh()` method does not compile. Even though both versions of method are marked `static`, the version in `Panda` has a more restrictive access modifier than the one it inherits, and it breaks the second rule for overriding methods. Remember, the four rules for overriding methods must be followed when hiding `static` methods.

Creating *final* Methods

We conclude our discussion of method inheritance with a somewhat self-explanatory rule— `final` methods cannot be replaced.

By marking a method `final`, you forbid a child class from replacing this method. This rule is in place both when you override a method and when you hide a method. In other words, you cannot hide a `static` method in a child class if it is marked `final` in the parent class.

Let's take a look at an example:

```
public class Bird {
   public final boolean hasFeathers() {
      return true;
   }
   public final static void flyAway() {}
}

public class Penguin extends Bird {
   public final boolean hasFeathers() {  // DOES NOT COMPILE
      return false;
   }
   public final static void flyAway() {}  // DOES NOT COMPILE
}
```

In this example, the instance method `hasFeathers()` is marked as `final` in the parent class `Bird`, so the child class `Penguin` cannot override the parent method, resulting in a compiler error. The `static` method

`flyAway()` is also marked `final`, so it cannot be hidden in the subclass. In this example, whether or not the child method used the `final` keyword is irrelevant—the code will not compile either way.

This rule applies only to inherited methods. For example, if the two methods were marked `private` in the parent `Bird` class, then the `Penguin` class, as defined, would compile. In that case, the `private` methods would be redeclared, not overridden or hidden.

Real World Scenario

WHY MARK A METHOD AS *FINAL?*

Although marking methods as `final` prevents them from being overridden, it does have advantages in practice. For example, you'd mark a method as `final` when you're defining a parent class and want to guarantee certain behavior of a method in the parent class, regardless of which child is invoking the method.

In the previous example with `Bird`, the author of the parent class may want to ensure the method `hasFeathers()` always returns `true`, regardless of the child class instance on which it is invoked. The author is confident that there is no example of a `Bird` in which feathers are not present.

The reason methods are not commonly marked as `final` in practice, though, is that it may be difficult for the author of a parent class method to consider all of the possible ways her child class may be used. For example, although all adult birds have feathers, a baby chick doesn't; therefore, if you have an instance of a `Bird` that is a chick, it would not have feathers. For this reason, the `final` modifier is often used when the author of the parent class wants to guarantee certain behavior at the cost of limiting polymorphism.

Hiding Variables

As you saw with method overriding, there are a lot of rules when two methods have the same signature and are defined in both the parent and child classes. Luckily, the rules for variables with the same name in the parent and child classes are a lot simpler. In fact, Java doesn't allow variables to be overridden. Variables can be hidden, though.

A *hidden variable* occurs when a child class defines a variable with the same name as an inherited variable defined in the parent class. This creates two distinct copies of the variable within an instance of the child class: one instance defined in the parent class and one defined in the child class.

As when hiding a `static` method, you can't override a variable; you can only hide it. Let's take a look at a hidden variable. What do you think the following application prints?

```
class Carnivore {
   protected boolean hasFur = false;
}

public class Meerkat extends Carnivore {
   protected boolean hasFur = true;

   public static void main(String[] args) {
      Meerkat m = new Meerkat();
      Carnivore c = m;
      System.out.println(m.hasFur);
      System.out.println(c.hasFur);
   }
}
```

It prints `true` followed by `false`. Confused? Both of these classes define a `hasFur` variable, but with different values. Even though there is only one object created by the `main()` method, both variables exist indepen-

dently of each other. The output changes depending on the reference variable used.

If you didn't understand the last example, don't worry. The next section on polymorphism will expand on how overriding and hiding differ. For now, you just need to know that overriding a method replaces the parent method on all reference variables (other than `super`), whereas hiding a method or variable replaces the member only if a child reference type is used.

Understanding Polymorphism

Java supports *polymorphism*, the property of an object to take on many different forms. To put this more precisely, a Java object may be accessed using a reference with the same type as the object, a reference that is a superclass of the object, or a reference that defines an interface the object implements, either directly or through a superclass. Furthermore, a cast is not required if the object is being reassigned to a super type or interface of the object.

INTERFACE PRIMER

We'll be discussing interfaces in detail in the next chapter. For this chapter, you need to know the following:

- An interface can define `abstract` methods.
- A class can implement any number of interfaces.
- A class implements an interface by overriding the inherited `abstract` methods.
- An object that implements an interface can be assigned to a reference for that interface.

As you'll see in the next chapter, the same rules for overriding methods and polymorphism apply.

Let's illustrate this polymorphism property with the following example:

```
public class Primate {
   public boolean hasHair() {
      return true;
   }
}

public interface HasTail {
   public abstract boolean isTailStriped();
}

public class Lemur extends Primate implements HasTail {
   public boolean isTailStriped() {
      return false;
   }
   public int age = 10;
   public static void main(String[] args) {
      Lemur lemur = new Lemur();
      System.out.println(lemur.age);

      HasTail hasTail = lemur;
      System.out.println(hasTail.isTailStriped());

      Primate primate = lemur;
      System.out.println(primate.hasHair());
   }
}
```

This code compiles and prints the following output:

```
10
false
true
```

The most important thing to note about this example is that only one object, `Lemur`, is created and referenced. Polymorphism enables an in-

stance of `Lemur` to be reassigned or passed to a method using one of its supertypes, such as `Primate` or `HasTail`.

Once the object has been assigned to a new reference type, only the methods and variables available to that reference type are callable on the object without an explicit cast. For example, the following snippets of code will not compile:

```
HasTail hasTail = lemur;
System.out.println(hasTail.age);              // DOES NOT COMPILE

Primate primate = lemur;
System.out.println(primate.isTailStriped()); // DOES NOT COMPILE
```

In this example, the reference `hasTail` has direct access only to methods defined with the `HasTail` interface; therefore, it doesn't know the variable `age` is part of the object. Likewise, the reference `primate` has access only to methods defined in the `Primate` class, and it doesn't have direct access to the `isTailStriped()` method.

Object vs. Reference

In Java, all objects are accessed by reference, so as a developer you never have direct access to the object itself. Conceptually, though, you should consider the object as the entity that exists in memory, allocated by the Java runtime environment. Regardless of the type of the reference you have for the object in memory, the object itself doesn't change. For example, since all objects inherit `java.lang.Object`, they can all be reassigned to `java.lang.Object`, as shown in the following example:

```
Lemur lemur = new Lemur();

Object lemurAsObject = lemur;
```

Even though the `Lemur` object has been assigned to a reference with a different type, the object itself has not changed and still exists as a `Lemur` object in memory. What has changed, then, is our ability to access methods within the `Lemur` class with the `lemurAsObject` reference. Without an explicit cast back to `Lemur`, as you'll see in the next section, we no longer have access to the `Lemur` properties of the object.

We can summarize this principle with the following two rules:

1. The type of the object determines which properties exist within the object in memory.
2. The type of the reference to the object determines which methods and variables are accessible to the Java program.

It therefore follows that successfully changing a reference of an object to a new reference type may give you access to new properties of the object, but remember, those properties existed before the reference change occurred.

Let's illustrate this property using the previous example in Figure 8.4.

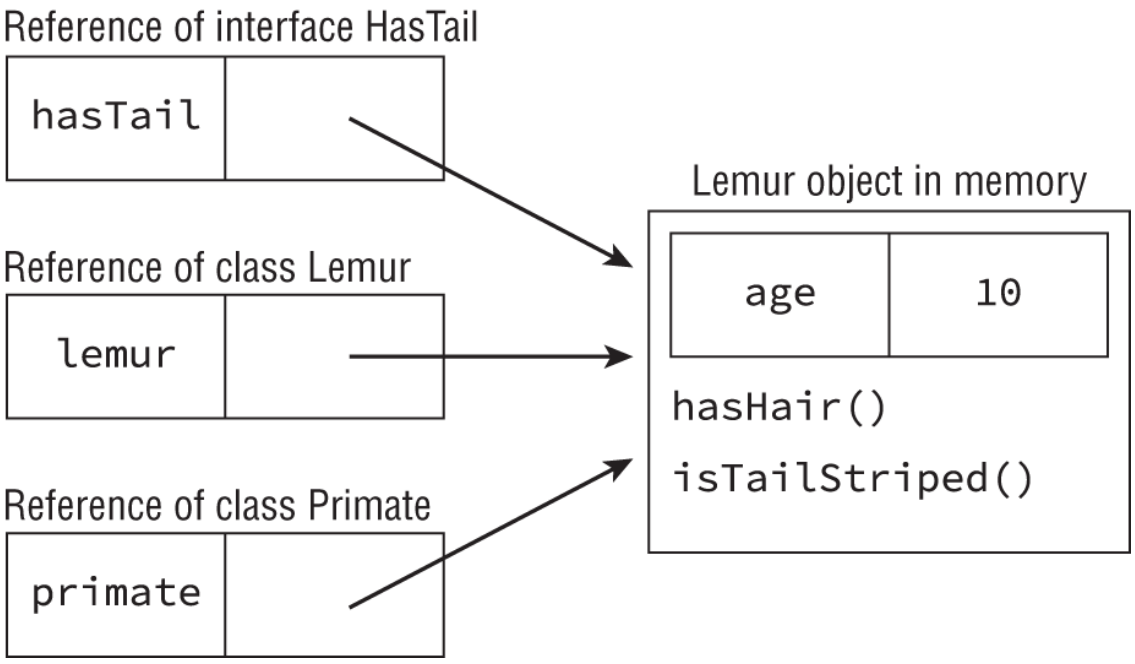

FIGURE 8.4 Object vs. reference

As you can see in the figure, the same object exists in memory regardless of which reference is pointing to it. Depending on the type of the reference, we may only have access to certain methods. For example, the `hasTail` reference has access to the method `isTailStriped()` but doesn't have access to the variable `age` defined in the `Lemur` class. As you'll learn in the next section, it is possible to reclaim access to the variable `age` by explicitly casting the `hasTail` reference to a reference of type `Lemur`.

Casting Objects

In the previous example, we created a single instance of a `Lemur` object and accessed it via superclass and interface references. Once we changed the reference type, though, we lost access to more specific members defined in the subclass that still exist within the object. We can reclaim those references by casting the object back to the specific subclass it came from:

```
Primate primate = new Lemur();  // Implicit Cast

Lemur lemur2 = primate;         // DOES NOT COMPILE
System.out.println(lemur2.age);

Lemur lemur3 = (Lemur)primate;  // Explicit Cast
System.out.println(lemur3.age);
```

In this example, we first create a `Lemur` object and implicitly cast it to a `Primate` reference. Since `Lemur` is a subclass of `Primate`, this can be done without a cast operator. Next, we try to convert the `primate` reference back to a `lemur` reference, `lemur2`, without an explicit cast. The result is that the code will not compile. In the second example, though, we explicitly cast the object to a subclass of the object `Primate`, and we gain access to all the methods and fields available to the `Lemur` class.

Casting objects is similar to casting primitives, as you saw in Chapter 3, "Operators." When casting objects, you do not need a cast operator if the current reference is a subtype of the target type. This is referred to as an implicit cast or type conversion. Alternatively, if the current reference is not a subtype of the target type, then you need to perform an explicit cast with a compatible type. If the underlying object is not compatible with the type, then a `ClassCastException` will be thrown at runtime.

We summarize these concepts into a set of rules for you to memorize for the exam:

1. Casting a reference from a subtype to a supertype doesn't require an explicit cast.
2. Casting a reference from a supertype to a subtype requires an explicit cast.
3. The compiler disallows casts to an unrelated class.
4. At runtime, an invalid cast of a reference to an unrelated type results in a `ClassCastException` being thrown.

The third rule is important; the exam may try to trick you with a cast that the compiler doesn't allow. In our previous example, we were able to cast a `Primate` reference to a `Lemur` reference, because `Lemur` is a subclass of `Primate` and therefore related. Consider this example instead:

```
public class Bird {}

public class Fish {
   public static void main(String[] args) {
      Fish fish = new Fish();
      Bird bird = (Bird)fish;  // DOES NOT COMPILE
   }
}
```

In this example, the classes `Fish` and `Bird` are not related through any class hierarchy that the compiler is aware of; therefore, the code will not

compile. While they both extend `Object` implicitly, they are considered unrelated types since one cannot be a subtype of the other.

While the compiler can enforce rules about casting to unrelated types for classes, it cannot do the same for interfaces, since a subclass may implement the interface. We'll revisit this topic in the next chapter. For now, you just need to know the third rule on casting applies to class types only, not interfaces.

Casting is not without its limitations. Even though two classes share a related hierarchy, that doesn't mean an instance of one can automatically be cast to another. Here's an example:

```
public class Rodent {}

public class Capybara extends Rodent {
   public static void main(String[] args) {
      Rodent rodent = new Rodent();
      Capybara capybara = (Capybara)rodent;  // ClassCastException
   }
}
```

This code creates an instance of `Rodent` and then tries to cast it to a subclass of `Rodent`, `Capybara`. Although this code will compile, it will throw a `ClassCastException` at runtime since the object being referenced is not an instance of the `Capybara` class. The thing to keep in mind in this example is the `Rodent` object created does not inherit the `Capybara` class in any way.

When reviewing a question on the exam that involves casting and polymorphism, be sure to remember what the instance of the object actually

is. Then, focus on whether the compiler will allow the object to be referenced with or without explicit casts.

The *instanceof* Operator

In Chapter 3, we presented the `instanceof` operator, which can be used to check whether an object belongs to a particular class or interface and to prevent `ClassCastException` s at runtime. Unlike the previous example, the following code snippet doesn't throw an exception at runtime and performs the cast only if the `instanceof` operator returns `true` :

```
if(rodent instanceof Capybara) {
   Capybara capybara = (Capybara)rodent;
}
```

Just as the compiler does not allow casting an object to unrelated types, it also does not allow `instanceof` to be used with unrelated types. We can demonstrate this with our unrelated `Bird` and `Fish` classes:

```
public static void main(String[] args) {
   Fish fish = new Fish();
   if (fish instanceof Bird) {  // DOES NOT COMPILE
      Bird bird = (Bird) fish;  // DOES NOT COMPILE
   }
}
```

In this snippet, neither the `instanceof` operator nor the explicit cast operation compile.

Polymorphism and Method Overriding

In Java, polymorphism states that when you override a method, you replace all calls to it, even those defined in the parent class. As an example, what do you think the following code snippet outputs?

```
class Penguin {
   public int getHeight() { return 3; }
   public void printInfo() {
      System.out.print(this.getHeight());
   }
}

public class EmperorPenguin extends Penguin {
   public int getHeight() { return 8; }
   public static void main(String []fish) {
      new EmperorPenguin().printInfo();
   }
}
```

If you said `8`, then you are well on your way to understanding polymorphism. In this example, the object being operated on in memory is an `EmperorPenguin`. The `getHeight()` method is overridden in the subclass, meaning all calls to it are replaced at runtime. Despite `printInfo()` being defined in the `Penguin` class, calling `getHeight()` on the object calls the method associated with the precise object in memory, not the current reference type where it is called. Even using the `this` reference, which is optional in this example, does not call the parent version because the method has been replaced.

The facet of polymorphism that replaces methods via overriding is one of the most important properties in all of Java. It allows you to create complex inheritance models, with subclasses that have their own custom implementation of overridden methods. It also means the parent class does not need to be updated to use the custom or overridden method. If the method is properly overridden, then the overridden version will be used in all places that it is called.

Remember, you can choose to limit polymorphic behavior by marking methods `final`, which prevents them from being overridden by a subclass.

CALLING THE PARENT VERSION OF AN OVERRIDDEN METHOD

As you saw earlier in the chapter, there is one exception to overriding a method where the parent method can still be called, and that is when the `super` reference is used. How can you modify our `EmperorPenguin` example to print `3`, as defined in the `Penguin` `getHeight()` method? You could try calling `super.getHeight()` in the `printInfo()` method of the `Penguin` class.

```
class Penguin {
   ...
   public void printInfo() {
      System.out.print(super.getHeight());  // DOES NOT COMPILE
   }
}
```

Unfortunately, this does not compile, as `super` refers to the super-class of `Penguin`, in this case `Object`. The solution is to override `printInfo()` in the `EmperorPenguin` class and use `super` there.

```
public class EmperorPenguin extends Penguin {
   ...
   public void printInfo() {
      System.out.print(super.getHeight());
   }
   ...
}
```

This new version of `EmperorPenguin` uses the `getHeight()` method declared in the parent class and prints `3`.

Overriding vs. Hiding Members

While method overriding replaces the method everywhere it is called, `static` method and variable hiding does not. Strictly speaking, hiding

members is not a form of polymorphism since the methods and variables maintain their individual properties. Unlike method overriding, hiding members is very sensitive to the reference type and location where the member is being used.

Let's take a look at an example:

```
class Penguin {
   public static int getHeight() { return 3; }
   public void printInfo() {
      System.out.println(this.getHeight());
   }
}
public class CrestedPenguin extends Penguin {
   public static int getHeight() { return 8; }
   public static void main(String... fish) {
      new CrestedPenguin().printInfo();
   }
}
```

The `CrestedPenguin` example is nearly identical to our previous `EmporerPenguin` example, although as you probably already guessed, it prints `3` instead of `8`. The `getHeight()` method is `static` and is therefore hidden, not overridden. The result is that calling `getHeight()` in `CrestedPenguin` returns a different value than calling it in the `Penguin`, even if the underlying object is the same. Contrast this with overriding a method, where it returns the same value for an object regardless of which class it is called in.

What about the fact that we used `this` to access a `static` method in `this.getHeight()`? As discussed in Chapter 7, while you are permitted to use an instance reference to access a `static` variable or method, it is often discouraged. In fact, the compiler will warn you when you access `static` members in a non-`static` way. In this case, the `this` reference had no impact on the program output.

Besides the location, the reference type can also determine the value you get when you are working with hidden members. Ready? Let's try a more complex example:

```
class Marsupial {
   protected int age = 2;
   public static boolean isBiped() {
      return false;
   }
}

public class Kangaroo extends Marsupial {
   protected int age = 6;
   public static boolean isBiped() {
      return true;
   }

   public static void main(String[] args) {
      Kangaroo joey = new Kangaroo();
      Marsupial moey = joey;
      System.out.println(joey.isBiped());
      System.out.println(moey.isBiped());
      System.out.println(joey.age);
      System.out.println(moey.age);
   }
}
```

The program prints the following:

```
true
false
6
2
```

Remember, in this example, only *one object*, of type `Kangaroo` , is created and stored in memory. Since `static` methods can only be hidden, not

overridden, Java uses the reference type to determine which version of `isBiped()` should be called, resulting in `joey.isBiped()` printing `true` and `moey.isBiped()` printing `false`.

Likewise, the `age` variable is hidden, not overridden, so the reference type is used to determine which value to output. This results in `joey.age` returning `6` and `moey.age` returning `2`.

Real World Scenario

DON'T HIDE MEMBERS IN PRACTICE

Although Java allows you to hide variables and `static` methods, it is considered an extremely poor coding practice. As you saw in the previous example, the value of the variable or method can change depending on what reference is used, making your code very confusing, difficult to follow, and challenging for others to maintain. This is further compounded when you start modifying the value of the variable in both the parent and child methods, since it may not be clear which variable you're updating.

When you're defining a new variable or `static` method in a child class, it is considered good coding practice to select a name that is not already used by an inherited member. Redeclaring `private` methods and variables is considered less problematic, though, because the child class does not have access to the variable in the parent class to begin with.

For the exam, make sure you understand these examples as they show how hidden and overridden methods are fundamentally different. In practice, overriding methods is the cornerstone of polymorphism and is an extremely powerful feature.

Summary

This chapter took the basic class structures we've presented throughout the book and expanded them by introducing the notion of inheritance. Java classes follow a multilevel single-inheritance pattern in which every class has exactly one direct parent class, with all classes eventually inheriting from `java.lang.Object`.

Inheriting a class gives you access to all of the `public` and `protected` members of the class. It also gives you access to package-private members of the class if the classes are in the same package. All instance methods, constructors, and instance initializers have access to two special reference variables: `this` and `super`. Both `this` and `super` provide access to all inherited members, with only `this` providing access to all members in the current class declaration.

Constructors are special methods that use the class name and do not have a return type. They are used to instantiate new objects. Declaring constructors requires following a number of important rules. If no constructor is provided, the compiler will automatically insert a default no-argument constructor in the class. The first line of every constructor is a call to an overloaded constructor, `this()`, or a parent constructor, `super()`; otherwise, the compiler will insert a call to `super()` as the first line of the constructor. In some cases, such as if the parent class does not define a no-argument constructor, this can lead to compilation errors. Pay close attention on the exam to any class that defines a constructor with arguments and doesn't define a no-argument constructor.

Classes are initialized in a predetermined order: superclass initialization; `static` variables and `static` initializers in the order that they appear; instance variables and instance initializers in the order they appear; and finally, the constructor. All `final` instance variables must be assigned a value exactly once. If by the time a constructor finishes, a `final` instance variable is not assigned a value, then the constructor will not compile.

We reviewed overloaded, overridden, hidden, and redeclared methods and showed how they differ, especially in terms of polymorphism. A

method is overloaded if it has the same name but a different signature as another accessible method. A method is overridden if it has the same signature as an inherited method, with access modifiers, exceptions, and a return type that are compatible. A `static` method is hidden if it has the same signature as an inherited static method. Finally, a method is redeclared if it has the same name and possibly the same signature as an uninherited method.

We also introduced the notion of hiding variables, although we strongly discourage this in practice as it often leads to confusing, difficult-to-maintain code.

Finally, this chapter introduced the concept of polymorphism, central to the Java language, and showed how objects can be accessed in a variety of forms. Make sure you understand when casts are needed for accessing objects, and be able to spot the difference between compile-time and runtime cast problems.

Exam Essentials

Be able to write code that extends other classes. A Java class that extends another class inherits all of its `public` and `protected` methods and variables. If the class is in the same package, it also inherits all package-private members of the class. Classes that are marked `final` cannot be extended. Finally, all classes in Java extend `java.lang.Object` either directly or from a superclass.

Be able to distinguish and make use of *this*, *this()*, *super*, and *super()*. To access a current or inherited member of a class, the `this` reference can be used. To access an inherited member, the `super` reference can be used. The `super` reference is often used to reduce ambiguity, such as when a class reuses the name of an inherited method or variable. The calls to `this()` and `super()` are used to access constructors in the same class and parent class, respectively.

Evaluate code involving constructors. The first line of every constructor is a call to another constructor within the class using `this()` or a call to a constructor of the parent class using the `super()` call. The compiler will insert a call to `super()` if no constructor call is declared. If the parent class doesn't contain a no-argument constructor, an explicit call to the parent constructor must be provided. Be able to recognize when the default constructor is provided. Remember that the order of initialization is to initialize all classes in the class hierarchy, starting with the superclass. Then, the instances are initialized, again starting with the superclass. All `final` variables must be assigned a value exactly once by the time the constructor is finished.

Understand the rules for method overriding. Java allows methods to be overridden, or replaced, by a subclass if certain rules are followed: a method must have the same signature, be at least as accessible as the parent method, must not declare any new or broader exceptions, and must use covariant return types. The generic parameter types must exactly match in any of the generic method arguments or a generic return type. Methods marked `final` may not be overridden or hidden.

Understand the rules for hiding methods and variables. When a `static` method is overridden in a subclass, it is referred to as method hiding. Likewise, variable hiding is when an inherited variable name is reused in a subclass. In both situations, the original method or variable still exists and is accessible depending on where it is accessed and the reference type used. For method hiding, the use of `static` in the method declaration must be the same between the parent and child class. Finally, variable and method hiding should generally be avoided since it leads to confusing and difficult-to-follow code.

Recognize the difference between method overriding and method overloading. Both method overloading and overriding involve creating a new method with the same name as an existing method. When the method signature is the same, it is referred to as method overriding and must follow a specific set of override rules to compile. When the method

signature is different, with the method taking different inputs, it is referred to as method overloading, and none of the override rules are required. Method overriding is important to polymorphism because it replaces all calls to the method, even those made in a superclass.

Understand polymorphism. An object may take on a variety of forms, referred to as polymorphism. The object is viewed as existing in memory in one concrete form but is accessible in many forms through reference variables. Changing the reference type of an object may grant access to new members, but the members always exist in memory.

Recognize valid reference casting. An instance can be automatically cast to a superclass or interface reference without an explicit cast. Alternatively, an explicit cast is required if the reference is being narrowed to a subclass of the object. The Java compiler doesn't permit casting to unrelated class types. Be able to discern between compiler-time casting errors and those that will not occur until runtime and that throw a `ClassCastException`.

Review Questions

The answers to the chapter review questions can be found in the Appendix.

1. Which code can be inserted to have the code print `2`?

```
public class BirdSeed {
   private int numberBags;
   boolean call;

   public BirdSeed() {
      // LINE 1
      call = false;
      // LINE 2
   }
```

```
   public BirdSeed(int numberBags) {
      this.numberBags = numberBags;
   }

   public static void main(String[] args) {
      BirdSeed seed = new BirdSeed();
      System.out.print(seed.numberBags);
   } }
```

1. Replace line 1 with `BirdSeed(2);`
2. Replace line 2 with `BirdSeed(2);`
3. Replace line 1 with `new BirdSeed(2);`
4. Replace line 2 with `new BirdSeed(2);`
5. Replace line 1 with `this(2);`
6. Replace line 2 with `this(2);`
7. The code prints `2` without any changes.

2. Which of the following statements about methods are true? (Choose all that apply.)
   1. Overloaded methods must have the same signature.
   2. Overridden methods must have the same signature.
   3. Hidden methods must have the same signature.
   4. Overloaded methods must have the same return type.
   5. Overridden methods must have the same return type.
   6. Hidden methods must have the same return type.
3. What is the output of the following program?

```
1:  class Mammal {
2:     private void sneeze() {}
3:     public Mammal(int age) {
4:        System.out.print("Mammal");
5:     } }
6:  public class Platypus extends Mammal {
7:     int sneeze() { return 1; }
8:     public Platypus() {
9:        System.out.print("Platypus");
10:    }
11:    public static void main(String[] args) {
```

```
12:       new Mammal(5);
13:    } }
```

1. `Platypus`
2. `Mammal`
3. `PlatypusMammal`
4. `MammalPlatypus`
5. The code will compile if line 7 is changed.
6. The code will compile if line 9 is changed.

4. Which of the following complete the constructor so that this code prints out `50`? (Choose all that apply.)

```
class Speedster {
   int numSpots;
}
public class Cheetah extends Speedster {
   int numSpots;

   public Cheetah(int numSpots) {
      // INSERT CODE HERE
   }

   public static void main(String[] args) {
      Speedster s = new Cheetah(50);
      System.out.print(s.numSpots);
   }
}
```

1. `numSpots = numSpots;`
2. `numSpots = this.numSpots;`
3. `this.numSpots = numSpots;`
4. `numSpots = super.numSpots;`
5. `super.numSpots = numSpots;`
6. The code does not compile, regardless of the code inserted into the constructor.
7. None of the above

5. What is the output of the following code?

```
1:  class Arthropod {
2:     protected void printName(long input) {
3:        System.out.print("Arthropod");
4:     }
5:     void printName(int input) {
6:        System.out.print("Spooky");
7:     } }
8:  public class Spider extends Arthropod {
9:     protected void printName(int input) {
10:       System.out.print("Spider");
11:    }
12:    public static void main(String[] args) {
13:       Arthropod a = new Spider();
14:       a.printName((short)4);
15:       a.printName(4);
16:       a.printName(5L);
17:    } }
```

    1. `SpiderSpiderArthropod`
    2. `SpiderSpiderSpider`
    3. `SpiderSpookyArthropod`
    4. `SpookySpiderArthropod`
    5. The code will not compile because of line 5.
    6. The code will not compile because of line 9.
    7. None of the above

6. Which of the following statements about overridden methods are true? (Choose all that apply.)
    1. An overridden method must contain method parameters that are the same or covariant with the method parameters in the inherited method.
    2. An overridden method may declare a new exception, provided it is not checked.
    3. An overridden method must be more accessible than the method in the parent class.

4. An overridden method may declare a broader checked exception than the method in the parent class.
5. If an inherited method returns `void`, then the overridden version of the method must return `void`.
6. None of the above

7. Which of the following pairs, when inserted into the blanks, allow the code to compile? (Choose all that apply.)

```
1:  public class Howler {
2:     public Howler(long shadow) {
3:        _________________;
4:     }
5:     private Howler(int moon) {
6:        super();
7:     }
8:  }
9:  class Wolf extends Howler {
10:    protected Wolf(String stars) {
11:       super(2L);
12:    }
13:    public Wolf() {
14:       _________________;
15:    }
16: }
```

1. `this(3)` at line 3, `this("")` at line 14
2. `this()` at line 3, `super(1)` at line 14
3. `this((short)1)` at line 3, `this(null)` at line 14
4. `super()` at line 3, `super()` at line 14
5. `this(2L)` at line 3, `super((short)2)` at line 14
6. `this(5)` at line 3, `super(null)` at line 14
7. Remove lines 3 and 14.

8. What is the result of the following?

```
1:  public class PolarBear {
2:     StringBuilder value = new StringBuilder("t");
```

```
3:     { value.append("a"); }
4:     { value.append("c"); }
5:     private PolarBear() {
6:        value.append("b");
7:     }
8:     public PolarBear(String s) {
9:        this();
10:       value.append(s);
11:    }
12:    public PolarBear(CharSequence p) {
13:       value.append(p);
14:    }
15:    public static void main(String[] args) {
16:       Object bear = new PolarBear();
17:       bear = new PolarBear("f");
18:       System.out.println(((PolarBear)bear).value);
19:    } }
```

1. `tacb`
2. `tacf`
3. `tacbf`
4. `tcafb`
5. `taftacb`
6. The code does not compile.
7. An exception is thrown.

9. Which of the following method signatures are valid overrides of the `hairy()` method in the `Alpaca` class? (Choose all that apply.)

```
import java.util.*;

public class Alpaca {
   protected List<String> hairy(int p) { return null; }
}
```

1. `List<String> hairy(int p) { return null; }`
2. `public List<String> hairy(int p) { return null; }`
3. `public List<CharSequence> hairy(int p) { return null; }`

4. `private List<String> hairy(int p) { return null; }`
5. `public Object hairy(int p) { return null; }`
6. `public ArrayList<String> hairy(int p) { return null; }`
7. None of the above

10. How many lines of the following program contain a compilation error?

```
1:  public class Rodent {
2:     public Rodent(var x) {}
3:     protected static Integer chew() throws Exception {
4:        System.out.println("Rodent is chewing");
5:        return 1;
6:     }
7:  }
8:  class Beaver extends Rodent {
9:     public Number chew() throws RuntimeException {
10:       System.out.println("Beaver is chewing on wood");
11:       return 2;
12:    } }
```

1. None
2. 1
3. 2
4. 3
5. 4
6. 5

11. Which of the following statements about polymorphism are true? (Choose all that apply.)
    1. An object may be cast to a subtype without an explicit cast.
    2. If the type of a method argument is an interface, then a reference variable that implements the interface may be passed to the method.
    3. A method that takes a parameter with type `java.lang.Object` can be passed any variable.
    4. All cast exceptions can be detected at compile-time.

5. By defining a `final` instance method in the superclass, you guarantee that the specific method will be called in the parent class at runtime.
6. Polymorphism applies only to classes, not interfaces.

12. Which of the following statements can be inserted in the blank so that the code will compile successfully? (Choose all that apply.)

```
public class Snake {}
public class Cobra extends Snake {}
public class GardenSnake extends Cobra {}
public class SnakeHandler {
   private Snake snake;
   public void setSnake(Snake snake) { this.snake = snake; }
   public static void main(String[] args) {
      new SnakeHandler().setSnake(_____________);
   }
}
```

1. `new Cobra()`
2. `new Snake()`
3. `new Object()`
4. `new String("Snake")`
5. `new GardenSnake()`
6. `null`
7. None of the above. The class does not compile, regardless of the value inserted in the blank.

13. Which of these classes compile and will include a default constructor created by the compiler? (Choose all that apply.)

1.

```
public class Bird {
```

2.

```
public class Bird {
   public bird() {}
}
```

3.

```
public class Bird {
   public bird(String name) {}
}
```

4.

```
public class Bird {
   public Bird() {}
}
```

5.

```
public class Bird {
   Bird(String name) {}
}
```

6.

```
public class Bird {
   private Bird(int age) {}
}
```

7.

```
public class Bird {
   public Bird bird() {return null;}
}
```

14. Which of the following statements about inheritance are correct? (Choose all that apply.)
    1. A class can directly extend any number of classes.
    2. A class can implement any number of interfaces.
    3. All variables inherit `java.lang.Object`.
    4. If class `A` is extended by `B`, then `B` is a superclass of `A`.
    5. If class `C` implements interface `D`, then `C` is subtype of `D`.
    6. Multiple inheritance is the property of a class to have multiple direct superclasses.
15. What is the result of the following?

```
1:  class Arachnid {
2:     static StringBuilder sb = new StringBuilder();
3:     { sb.append("c"); }
4:     static
5:     { sb.append("u"); }
6:     { sb.append("r"); }
7:  }
8:  public class Scorpion extends Arachnid {
9:     static
10:    { sb.append("q"); }
11:    { sb.append("m"); }
12:    public static void main(String[] args) {
13:       System.out.print(Scorpion.sb + " ");
14:       System.out.print(Scorpion.sb + " ");
15:       new Arachnid();
16:       new Scorpion();
17:       System.out.print(Scorpion.sb);
18:    } }
```

    1. `qu qu qumrcrc`
    2. `u u ucrcrm`
    3. `uq uq uqmcrcr`
    4. `uq uq uqcrcrm`
    5. `qu qu qumcrcr`
    6. `qu qu qucrcrm`
    7. The code does not compile.

16. Which of the following methods are valid overrides of the `friendly()` method in the `Llama` class? (Choose all that apply.)

```
import java.util.*;

public class Llama {
   void friendly(List<String> laugh, Iterable<Short> s) {}
}
```

    1. `void friendly(List<CharSequence> laugh, Iterable<Short> s) {}`
    2. `void friendly(List<String> laugh, Iterable<Short> s) {}`
    3. `void friendly(ArrayList<String> laugh, Iterable<Short> s) {}`
    4. `void friendly(List<String> laugh, Iterable<Integer> s) {}`
    5. `void friendly(ArrayList<CharSequence> laugh, Object s) {}`
    6. `void friendly(ArrayList<String> laugh, Iterable... s) {}`
    7. None of the above

17. Which of the following statements about inheritance and variables are true? (Choose all that apply.)
    1. Instance variables can be overridden in a subclass.
    2. If an instance variable is declared with the same name as an inherited variable, then the type of the variable must be covariant.
    3. If an instance variable is declared with the same name as an inherited variable, then the access modifier must be at least as accessible as the variable in the parent class.
    4. If a variable is declared with the same name as an inherited `static` variable, then it must also be marked `static`.
    5. The variable in the child class may not throw a checked exception that is new or broader than the class of any exception thrown in the parent class variable.
    6. None of the above

18. Which of the following are true? (Choose all that apply.)
    1. `this()` can be called from anywhere in a constructor.
    2. `this()` can be called from anywhere in an instance method.

3. `this.variableName` can be called from any instance method in the class.
4. `this.variableName` can be called from any `static` method in the class.
5. You can call the default constructor written by the compiler using `this()`.
6. You can access a `private` constructor with the `main()` method in the same class.

19. Which statements about the following classes are correct? (Choose all that apply.)

```
1:  public class Mammal {
2:     private void eat() {}
3:     protected static void drink() {}
4:     public Integer dance(String p) { return null; }
5:  }
6:  class Primate extends Mammal {
7:     public void eat(String p) {}
8:  }
9:  class Monkey extends Primate {
10:    public static void drink() throws RuntimeException {}
11:    public Number dance(CharSequence p) { return null; }
12:    public int eat(String p) {}
13: }
```

1. The `eat()` method in `Mammal` is correctly overridden on line 7.
2. The `eat()` method in `Mammal` is correctly overloaded on line 7.
3. The `drink()` method in `Mammal` is correctly hidden on line 10.
4. The `drink()` method in `Mammal` is correctly overridden on line 10.
5. The `dance()` method in `Mammal` is correctly overridden on line 11.
6. The `dance()` method in `Mammal` is correctly overloaded on line 11.
7. The `eat()` method in `Primate` is correctly hidden on line 12.
8. The `eat()` method in `Primate` is correctly overloaded on line 12.

20. What is the output of the following code?

```
1:  class Reptile {
2:     {System.out.print("A");}
3:     public Reptile(int hatch) {}
4:     void layEggs() {
5:        System.out.print("Reptile");
6:     } }
7:  public class Lizard extends Reptile {
8:     static {System.out.print("B");}
9:     public Lizard(int hatch) {}
10:    public final void layEggs() {
11:       System.out.print("Lizard");
12:    }
13:    public static void main(String[] args) {
14:       Reptile reptile = new Lizard(1);
15:       reptile.layEggs();
16:    } }
```

1. `AALizard`
2. `BALizard`
3. `BLizardA`
4. `ALizard`
5. The code will not compile because of line 10.
6. None of the above

21. Which statement about the following program is correct?

```
1:  class Bird {
2:     int feathers = 0;
3:     Bird(int x) { this.feathers = x; }
4:     Bird fly() {
5:        return new Bird(1);
6:     } }
7:  class Parrot extends Bird {
8:     protected Parrot(int y) { super(y); }
9:     protected Parrot fly() {
10:       return new Parrot(2);
11:    } }
12: public class Macaw extends Parrot {
13:    public Macaw(int z) { super(z); }
```

```
14:    public Macaw fly() {
15:       return new Macaw(3);
16:    }
17:    public static void main(String... sing) {
18:       Bird p = new Macaw(4);
19:       System.out.print(((Parrot)p.fly()).feathers);
20:    } }
```

1. One line contains a compiler error.
2. Two lines contain compiler errors.
3. Three lines contain compiler errors.
4. The code compiles but throws a `ClassCastException` at runtime.
5. The program compiles and prints `3`.
6. The program compiles and prints `0`.

22. What does the following program print?

```
1:  class Person {
2:     static String name;
3:     void setName(String q) { name = q; } }
4:  public class Child extends Person {
5:     static String name;
6:     void setName(String w) { name = w; }
7:     public static void main(String[] p) {
8:        final Child m = new Child();
9:        final Person t = m;
10:       m.name = "Elysia";
11:       t.name = "Sophia";
12:       m.setName("Webby");
13:       t.setName("Olivia");
14:       System.out.println(m.name + " " + t.name);
15:    } }
```

1. `Elysia Sophia`
2. `Webby Olivia`
3. `Olivia Olivia`
4. `Olivia Sophia`
5. The code does not compile.

6. None of the above

23. What is the output of the following program?

```
1:  class Canine {
2:     public Canine(boolean t) { logger.append("a"); }
3:     public Canine() { logger.append("q"); }
4:
5:     private StringBuilder logger = new StringBuilder();
6:     protected void print(String v) { logger.append(v); }
7:     protected String view() { return logger.toString(); }
8:  }
9:
10: class Fox extends Canine {
11:    public Fox(long x) { print("p"); }
12:    public Fox(String name) {
13:       this(2);
14:       print("z");
15:    }
16: }
17:
18: public class Fennec extends Fox {
19:    public Fennec(int e) {
20:       super("tails");
21:       print("j");
22:    }
23:    public Fennec(short f) {
24:       super("eevee");
25:       print("m");
26:    }
27:
28:    public static void main(String... unused) {
29:       System.out.println(new Fennec(1).view());
30:    } }
```

1. `qpz`
2. `qpzj`
3. `jzpa`
4. `apj`

5. `apjm`
6. The code does not compile.
7. None of the above

24. Which statements about polymorphism and method inheritance are correct? (Choose all that apply.)
    1. It cannot be determined until runtime which overridden method will be executed in a parent class.
    2. It cannot be determined until runtime which hidden method will be executed in a parent class.
    3. Marking a method `static` prevents it from being overridden or hidden.
    4. Marking a method `final` prevents it from being overridden or hidden.
    5. The reference type of the variable determines which overridden method will be called at runtime.
    6. The reference type of the variable determines which hidden method will be called at runtime.
25. What is printed by the following program?

```
1:  class Antelope {
2:     public Antelope(int p) {
3:        System.out.print("4");
4:     }
5:     { System.out.print("2"); }
6:     static { System.out.print("1"); }
7:  }
8:  public class Gazelle extends Antelope {
9:     public Gazelle(int p) {
10:       super(6);
11:       System.out.print("3");
12:    }
13:    public static void main(String hopping[]) {
14:       new Gazelle(0);
15:    }
16:    static { System.out.print("8"); }
17:    { System.out.print("9"); }
18: }
```

1. `182640`
2. `182943`
3. `182493`
4. `421389`
5. The code does not compile.
6. The output cannot be determined until runtime.

26. How many lines of the following program contain a compilation error?

```
1:  class Primate {
2:     protected int age = 2;
3:     { age = 1; }
4:     public Primate() {
5:        this().age = 3;
6:     }
7:  }
8:  public class Orangutan {
9:     protected int age = 4;
10:    { age = 5; }
11:    public Orangutan() {
12:       this().age = 6;
13:    }
14:    public static void main(String[] bananas) {
15:       final Primate x = (Primate)new Orangutan();
16:       System.out.println(x.age);
17:    }
18: }
```

1. None, and the program prints `1` at runtime.
2. None, and the program prints `3` at runtime.
3. None, but it causes a `ClassCastException` at runtime.
4. 1
5. 2
6. 3
7. 4