

Chapter 6

Class Design

OCJP EXAM OBJECTIVES COVERED IN THIS CHAPTER:

- **Using Object-Oriented Concepts in Java**
 - Create classes and records, and define and use instance and static fields and methods, constructors, and instance and static initializers.
 - Understand variable scopes, apply encapsulation, and create immutable objects. Use local variable type inference.
 - Implement inheritance, including abstract and sealed types as well as record classes. Override methods, including that of the `Object` class. Implement polymorphism and differentiate between object type and reference type. Perform reference type casting, identify object types using the `instanceof` operator, and pattern matching with the `instanceof` operator and the `switch` construct.
-

In [Chapter 1](#), “Building Blocks,” we introduced the basic definition of a class in Java. In [Chapter 5](#), “Methods,” we delved into methods and modifiers and showed how you can use them to build more structured classes. In this chapter, we take things a 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.

This chapter is the culmination of some of the most important topics in Java including inheritance, class design, constructors, order of initialization, overriding methods, abstract classes, and immutable objects. Read this chapter carefully and make sure you understand all of the topics well. This chapter forms the basis of [Chapter 7](#), “Beyond Classes,” in which we expand our discussion of types to include other top-level and nested types.

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 certain 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.

When working with other types, like interfaces, we tend to use the general terms *subtype* and *supertype*. You see this more in the next chapter.

Declaring a Subclass

Let's begin with the declaration of a class and its subclass. [Figure 6.1](#) shows an example of a superclass, `Mammal`, and subclass `Rhinoceros`.

 A sample structure of the declaration of a class and its subclass. The variables are public or package access, final keyword, class keyword, class name, extends parent class, super class, and sub class.

[FIGURE 6.1](#) Subclass and superclass declarations

We indicate a class is a subclass by declaring it with the `extends` keyword. We don't need to declare anything in the superclass other than making sure it is not marked `final`. More on that shortly.

One key aspect of inheritance is that it is transitive. Given three classes [X, Y, Z], if X extends Y, and Y extends Z, then X is considered a subclass or descendant of Z. Likewise, Z is a superclass or ancestor of X. We sometimes use the term *direct* subclass or descendant to indicate the class directly extends the parent class. For example, X is a direct descendant only of class Y, not Z.

In the previous chapter, you learned that there are four access levels: `public`, `protected`, `package`, and `private`. When one class inherits from a parent class, all `public` and `protected` members are automatically available as part of the child class. If the two classes are in the same package, then package members are available to the child 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 can't have `private` members that can hold data or modify an object; it just means the subclass doesn't have direct access to them.

Let's take a look at a simple example:

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

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

```

}

public class Spider {
    public void printDetails() {
        System.out.println(size); // DOES NOT COMPILE
    }
}

```

Jaguar is a subclass or child of BigCat, making BigCat a superclass or parent of Jaguar. In the Jaguar class, size is accessible because it is marked protected. Via inheritance, the Jaguar subclass can read or write size as if it were its own member. Contrast this with the Spider class, which has no access to size since it is not inherited.

Class Modifiers

Like methods and variables, a class declaration can have various modifiers. [Table 6.1](#) lists the modifiers you should know for the exam.

TABLE 6.1 Class modifiers

Modifier	Description	Chapter covered
final	The class may not be extended.	Chapter 6
abstract	The class is abstract, may contain abstract methods, and requires a concrete subclass to instantiate.	Chapter 6
sealed	The class may only be extended by a specific list of classes.	Chapter 7
non-sealed	A subclass of a sealed class permits potentially unnamed subclasses.	Chapter 7
static	Used for static nested classes defined within a class.	Chapter 7

We cover abstract classes later in this chapter. In the next chapter, we cover sealed and non-sealed classes, as well as static nested classes.

For now, let's talk about marking a class final. The final modifier prevents a class from being extended any further. For example, the following does not compile:

```

public class Mammal {}

public final class Rhinoceros extends Mammal {}

public class Clara extends Rhinoceros {} // DOES NOT COMPILE

```

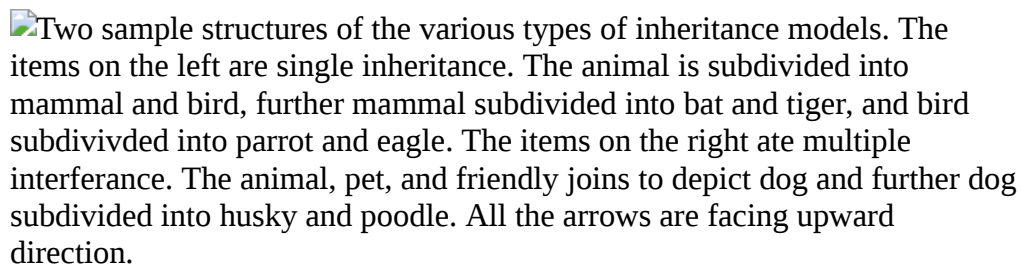
On the exam, pay attention to any class marked final. If you see another class extending it, you know immediately the code does not compile.

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 be 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, which you see in [Chapter 7](#)—a class may implement multiple interfaces.

[Figure 6.2](#) 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.

Two sample structures of the various types of inheritance models. The items on the left are single inheritance. The animal is subdivided into mammal and bird, further mammal subdivided into bat and tiger, and bird subdivided into parrot and eagle. The items on the right are multiple inheritance. The animal, pet, and friendly joins to depict dog and further dog subdivided into husky and poodle. All the arrows are facing upward direction.

[FIGURE 6.2](#) Types of inheritance

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*.

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, the following two are equivalent:

```
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, the compiler 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 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 6.3](#).


 A flow process of a java object inheritance. The structure starts from bottom, the variables ox to mammal, then to few loops to object. The program is All objects inherit java.lang.Object.

FIGURE 6.3 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](#), 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 review the basics for creating and working with classes.

Extending a Class

Let's create two files in the same package, `Animal.java` and `Lion.java`.

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

```
// Lion.java
public class Lion extends Animal {
```

```

    protected 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();
    }
}

```

There's a lot going on here, we know! The age variable exists in the parent `Animal` class and is not directly accessible in the `Lion` child class. It is indirectly accessible via the `setAge()` method. The name variable is protected, so it is inherited in the `Lion` class and directly accessible. We create the `Lion` instance in the `main()` method and use `setProperties()` to set instance variables. Finally, we call the `roar()` method, which 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 private and is not directly accessible from the subclass `Lion`. Therefore, the following would not compile:

```

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

```

Remember when working with subclasses that private members are never inherited, and package members are only inherited if the two classes are in the same package. If you need a refresher on access modifiers, it may help to read [Chapter 5](#) again.

Applying Class Access Modifiers

Like variables and methods, you can apply access modifiers to classes. As you might remember from [Chapter 1](#), a top-level class is one not defined inside another class. Also remember that a `.java` file can have at most one public top-level class.

While you can only have one public top-level class, you can have as many classes (in any order) with package access as you want. In fact, you don't even need to declare a public class! The following declares three classes, each with package access:

```

// Bear.java
class Bird {}
class Bear {}
class Fish {}

```

Trying to declare a top-level class with protected or private class will lead to a compiler error, though.

```
// ClownFish.java
protected class ClownFish{} // DOES NOT COMPILE
```

```
// BlueTang.java
private class BlueTang {} // DOES NOT COMPILE
```

Does that mean a class can never be declared protected or private? Not exactly. In [Chapter 7](#), we present nested types and show that when you define a class inside another, it can use any access modifier.

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 = null;
    public void setColor(String color) {
        color = color;
    }
    public static void main(String... unused) {
        var f = new Flamingo();
        f.setColor("PINK");
        System.out.print(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 (not the instance variable). 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, or 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; // Sets the instance variable with method parameter
}
```

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. reference 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 the `this` reference 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. This means the object instance actually holds two copies of the same variable with the same underlying name. When this happens, how do we reference the version in the parent class instead of the current class? Let's take a look at this example:

// **Reptile.java**

```
1: public class Reptile {
2:     protected int speed = 10;
3: }
```

// **Crocodile.java**

```
1: public class Crocodile extends Reptile {
2:     protected int speed = 20;
3:     public int getSpeed() {
4:         return speed;
5:     }
6:     public static void main(String[] data) {
7:         var croc = new Crocodile();
8:         System.out.println(croc.getSpeed()); // 20
9:     } }
```


One of the most important things to remember about this code is that an instance of `Crocodile` stores two separate values for `speed`: one at the `Reptile` level and one at the `Crocodile` level. On line 4, Java first checks to see if there is a local variable or method parameter named `speed`. Since there is not, it then checks `this.speed`; and since it exists, the program prints 20.

Declaring a variable with the same name as an inherited variable is referred to as *hiding* a variable and is discussed later in this chapter.

But what if we want the program to print the value in the `Reptile` class? Within the `Crocodile` class, we can access the parent value of `speed` instead, by using 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.

```
3:    public int getSpeed() {  
4:        return super.speed; // Causes the program to now print 10  
5:    }
```

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.println(this.label);  
11:         System.out.println(super.label);  
12:         System.out.println(this.age);  
13:         System.out.println(super.age);  
14:         System.out.println(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 (and

would print 3), but line 13 does not. Remember, while `this` includes current and inherited members, `super` only includes inherited members.

Last but not 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 is 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 are discussed later in this chapter.

Phew, that was a lot! Using `this` and `super` can take a little getting used to. Since we use them often in upcoming sections, make sure you understand the last example really well before moving forward.

Declaring Constructors

As you learned in [Chapter 1](#), 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 show how to create and call constructors.

Creating a Constructor

Let's start with a simple constructor:

```
public class Bunny {  
    public Bunny() {  
        System.out.print("hop");  
    }  
}
```

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`. For example, the following does not compile:

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

A class can have multiple constructors, as long as each constructor has a unique constructor 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(15)
```

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.

The 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 any time 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) {
        new Rabbit(); // Calls the default constructor
    }
}
```

In the Rabbit class, Java sees that no constructor was coded and creates one. The previous class is equivalent to the following, in which the default constructor is provided and therefore not inserted by the compiler:

```
public class Rabbit {  
    public Rabbit() {}  
    public static void main(String[] args) {  
        new Rabbit();    // Calls the user-defined constructor  
    }  
}
```

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 only makes an appearance in the compiled file with the .class extension.

For the exam, one of the most important rules you need to know is that the compiler *only inserts the default constructor when no constructors are defined*. 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:         var r1 = new Rabbit1();  
4:         var r2 = new Rabbit2();  
5:         var r3 = new Rabbit3(true);  
6:         var 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.

Calling Overloaded Constructors with *this()*

Have the basics about creating and referencing constructors? Good, because things are about to get a bit more complicated. Since a class can contain multiple overloaded constructors, these constructors can actually call one another. Let's start with a simple class containing two overloaded constructors:

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

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 bit of duplication, as `this.weight` is assigned 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 that we have five 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 rewrite the first constructor as the following:

```
public Hamster(int weight) { // Second constructor
    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) {    // Second constructor
    new Hamster(weight, "brown"); // Compiles, but creates an extra object
}
```

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. In this manner, we end up with two objects, one of which is 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, but with parentheses. When `this()` is used with parentheses, Java calls another constructor on the same instance of the class.

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

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

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 `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("chew");
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 that 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. This is often referred to as a *cycle* and is similar to the infinite loops that we discussed in [Chapter 3](#), “Making Decisions.” Since the 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 an error.

Here we summarize the rules you should know about constructors that we covered in this section. Study them well!

- A class can contain many overloaded constructors, provided the signature for each is distinct.
- The compiler inserts a default no-argument constructor if no constructors are declared.
- If a constructor calls `this()`, then it must be the first line of the constructor.
- Java does not allow cyclic constructor calls.

Calling Parent Constructors with *super()*

Congratulations, you’re well on your way to becoming an expert in using constructors! There’s one more set of rules we need to cover, though, for calling constructors in the parent class. After all, how do instance members of the parent class get initialized?

The first statement of *every* constructor is a call to a parent constructor using `super()` or another constructor in the class using `this()`. Read the previous sentence twice to make sure you remember it. It’s really important!

For simplicity in this section, we often refer to `super()` and `this()` to refer to any parent or overloaded constructor call, 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 Animal class, 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 Zebra class 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).

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 keyword super on the exam, make sure it is being used properly.

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 and accessible 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"); // Calls the first Animal constructor
    }
    public Gorilla() {
        super(5); // Calls the second Animal constructor
    }
}
```

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 it is accessible and the appropriate input parameters to the parent constructor are provided.

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. While reading the next section, keep in mind the process the Java compiler performs.

Default Constructor Tips and Tricks

We've presented a lot of rules so far, and you might have noticed something. Let's say we have a class that doesn't include a no-argument constructor. What happens if we define a subclass with no constructors, or a subclass with a constructor that doesn't include a `super()` reference?

```

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

public class Seal extends Mammal {} // DOES NOT COMPILE

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

```

The answer is that neither subclass compiles. Since `Mammal` defines a constructor, the compiler does not insert a no-argument constructor. The compiler will insert a default no-argument constructor into `Seal`, though, but it will be a simple implementation that just calls a nonexistent parent default constructor.

```

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

```

Likewise, `Elephant` will not compile for similar reasons. The compiler doesn't see a call to `super()` or `this()` as the first line of the constructor so it inserts a call to a nonexistent no-argument `super()` automatically.

```

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

```

In these cases, the 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.

```

public class Seal extends Mammal {
    public Seal() {
        super(6); // Explicit call to parent constructor
    }
}

```

```

    }
}

public class Elephant extends Mammal {
    public Elephant() {
        super(4); // Explicit call to parent constructor
    }
}

```

Subclasses may include no-argument constructors even if their parent classes do not. For example, the following compiles because `Elephant` includes a no-argument constructor:

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

It's a lot to take in, we know. For the exam, you should be able to spot right away why classes such as our first `Seal` and `Elephant` implementations 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 to the `Mammal` class.

We conclude this section by adding two constructor rules to your skill set.

- If a constructor calls `super()` or `this()`, then it must be the first line of the constructor.
- If the constructor does not contain a `this()` or `super()` reference, then the compiler automatically inserts `super()` with no arguments as the first line of the constructor.

Congratulations, you've learned everything we can teach you about declaring constructors. Next, we move on to initialization and discuss how to use constructors.

Initializing Objects

In [Chapter 1](#), we covered order of initialization, albeit in a very simplistic manner. *Order of initialization* refers to how members of a class are assigned values. They can be given default values, like `0` for an `int`, or require explicit values, such as for `final` variables. In this section, we go into much more detail about how order of initialization works and how to spot errors on the exam.

Initializing Classes

We begin our discussion of order of initialization with class initialization. First, we initialize the class, which involves invoking all static members in the class hierarchy, starting with the highest superclass and working downward. This is sometimes referred to as *loading* the class. The Java Virtual Machine (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.

One of the most important rules 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. Initialize the superclass of X.
2. Process all static variable declarations in the order in which they appear in the class.
3. Process all static initializers in the order in which 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 {
    public static void main(String[] grass) {
        System.out.print("C");
        new Hippo();
        new Hippo();
        new Hippo();
    }
    static { System.out.print("B"); }
}
```

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.

Initializing *final* Fields

Before we delve into order of initialization for instance members, we need to talk about `final` fields (instance variables) for a minute. When we presented instance and class variables in [Chapter 1](#), we told you they are assigned a default value based on their type if no value is specified. For example, a `double` is initialized with `0.0`, while an object reference is initialized to `null`. A default value is only applied to a non-`final` field, though.

As you saw in [Chapter 5](#), `final` static variables must be explicitly assigned a value exactly once. Fields 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"; // Declaration assignment  
    {  
        volume = 10; // Instance initializer assignment  
    }  
}
```

Unlike static class members, though, `final` instance fields can also be set in a constructor. The constructor is part of the initialization process, so it is allowed to assign `final` instance variables. 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 exactly once.*

Let's try this in an example:

```
public class MouseHouse {  
    private final int volume;  
    private final String name;  
    public MouseHouse() {  
        this.name = "Empty House"; // Constructor assignment  
    }  
    {  
        volume = 10; // Instance initializer assignment  
    }  
}
```

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. If they are not assigned a value

when 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.

```
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. In terms of assigning values, each constructor is reviewed individually, which is why the second constructor does not compile. 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. Second, the constructor sets a value for the `volume` variable, even though it was already assigned a value by the instance initializer.



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.

What about `final` instance variables when a constructor calls another constructor in the same class? In that case, you have to follow the flow 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 as long as they are explicitly set.

Initializing Instances

We've covered class initialization and `final` fields, so now it's time to move on to order of initialization for objects. We'll warn you that this can be a bit cumbersome at first, but the exam isn't likely to ask questions more complicated than the examples in this section. We promise to take it slowly, though.

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 the instance initializers and constructors within each class, in the reverse order in which each class was instantiated. We summarize the order of initialization for an instance as follows:

Initialize Instance of X

1. Initialize *Class X* if it has not been previously initialized.
2. Initialize the superclass instance of X.
3. Process all instance variable declarations in the order in which they appear in the class.
4. Process all instance initializers in the order in which they appear in the class.
5. Initialize the constructor, including any overloaded constructors referenced with `this()`.

Let's try an 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:    } }
```

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 created on line 13. Again, since no superclass is 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 yield 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 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:    } }
```

The output looks like this:


```
0
Ready
swimmy
1
Constructor
```

No superclass is 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. Next we create an instance declared on line 14. 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 prints Constructor.

Ready for a more difficult example, the kind you might see on the exam? 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 second rule, the superclass instance of `GiraffeFamily` is initialized first. Per our fourth rule, the instance initializer in the superclass `GiraffeFamily` is called, and `B` is printed on line 3. Per the fifth 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 fourth and fifth 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 uncommon on the exam. If you see one, just write down what is going on as you read the code.

We conclude this section by listing important rules you should know for the exam:

- A class is initialized at most once by the JVM before it is referenced or used.
- All `static final` variables must be assigned a value exactly once, either when they are declared or in a static initializer.
- All `final` fields must be assigned a value exactly once, either when they are declared, in an instance initializer, or in a constructor.
- Non-`final static` and instance variables defined without a value are assigned a default value based on their type.
- The order of initialization is as follows: variable declarations, then initializers, and finally constructors.

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 classes, each of which extends from the `Animal` class. Furthermore, each class defines an `eat()` method with an identical implementation. In this scenario, it's a lot better to define `eat()` once in the `Animal` class than to have to maintain the same method in five separate classes.

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 review the rules for method inheritance and how Java handles such scenarios.

We refer to the ability of an object to take on many different forms as *polymorphism*. We cover this more in the next chapter, but for now you just need to know that an object can be used in a variety of ways, in part based on the reference variable used to call the object.

Overriding a Method

What if a method with the same signature is defined in both the parent and child classes? 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 is composed of the name of the method and method parameters. It does not include the return type, access modifiers, optional specifiers, or any declared exceptions.

When you override a method, you may still 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 Marsupial {
    public double getAverageWeight() {
        return 50;
    }
}
public class Kangaroo extends Marsupial {
    public double getAverageWeight() {
        return super.getAverageWeight()+20;
    }
    public static void main(String[] args) {
        System.out.println(new Marsupial().getAverageWeight()); // 50.0
        System.out.println(new Kangaroo().getAverageWeight());  // 70.0
    }
}
```

In this example, the `Kangaroo` class overrides the `getAverageWeight()` method but in the process calls the parent version using the `super` reference.

Method Overriding Infinite 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 `Marsupial` method; it would call the current `Kangaroo` method. 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*.

While these rules may seem confusing or arbitrary at first, they are needed for consistency. Without these rules in place, it is possible to create contradictions within the Java language.

Rule #1: Method Signatures

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.



We covered overloading a method in [Chapter 5](#), and it is similar to overriding a method, as both involve defining a method using the same name. Overloading differs from overriding in that overloaded methods use a different parameter list. For the exam, it is important that you understand this distinction and that overridden methods have the same signature and a lot more rules than overloaded methods.

Rule #2: Access Modifiers

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;
    }
}
```

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. What would this program print?

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

The answer is, we don't know. The reference type for the object is Camel, where the method is declared public, but the object is actually an instance of type BactrianCamel, where the method 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.

Rule #3: Checked Exceptions

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 polymorphic reasons similar to 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. One implication of this rule is that overridden methods are free to declare any number of new unchecked exceptions.



If you don't know what a checked or unchecked exception is, don't worry. We cover this in [Chapter 11](#), "Exceptions and Localization." For now, you just need to know that the rule applies only to checked exceptions. It's also helpful to know that both IOException and

`FileNotFoundException` are checked exceptions, and that `FileNotFoundException` is a subclass of `IOException`.

Let's try an example:

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

    protected void hide() {}

    protected void exitShell() throws FileNotFoundException {}
}

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

    public void hide() throws FileNotFoundException {} // DOES NOT COMPILE

    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 for overridden methods. The first overridden method `sleep()` in `GalapagosTortoise` compiles without issue because the declared exception is narrower than the exception declared in the parent class.

The overridden `hide()` method does not compile because it declares a new checked exception not present in the parent declaration. The overridden `exitShell()` also does not compile, since `IOException` is a broader checked exception than `FileNotFoundException`. We revisit these exception classes, including memorizing which ones are subclasses of each other, in [Chapter 11](#).

Rule #4: Covariant Return Types

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";
    }
}

public 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. Remember, a broader access modifier is acceptable in an overridden method.

From [Chapter 4](#), “Core APIs,” we learned 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 instance, 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 as 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.

That’s everything you need to know about overriding methods for this chapter. In [Chapter 9](#), “Collections and Generics,” we revisit overriding methods involving generics. There’s always more to learn!



Real World Scenario

Marking Methods with the `@Override` Annotation

An annotation is a metadata tag that provides additional information about your code. You can use the `@Override` annotation to tell the compiler that you are attempting to override a

method.

```
public class Fish {
    public void swim() {};
}
public class Shark extends Fish {
    @Override
    public void swim() {};
}
```

When the method is correctly overridden, adding the annotation doesn't impact the code. On the other hand, when the method is incorrectly overridden, this annotation can prevent you from making a mistake. The following does not compile because of the presence of the `@Override` annotation:

```
public class Fish {
    public void swim() {};
}
public class Shark extends Fish {
    @Override
    public void swim(int speed) {}; // DOES NOT COMPILE
}
```

The compiler sees that you are attempting a method override and looks for an inherited version of `swim()` that takes an `int` value. Since the compiler doesn't find one, it reports an error. While knowing advanced topics (such as how to create annotations) is not required for the exam, knowing how to use them properly is.

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. For example, these two declarations compile:

```
public class Beetle {
    private String getSize() {
        return "Undefined";
    }
}

public class RhinocerosBeetle extends Beetle {
    private int getSize() {
        return 5;
    }
}
```


Notice that the return type differs in the child method from `String` to `int`. In this example, the method `getSize()` in the parent class is not inherited, so the method in the child class is a new method and not an override of the method in the parent class.

What if the `getSize()` method was declared `public` in `Beetle`? In this case, the method in `RhinocerosBeetle` would be an invalid override. The access modifier in `RhinocerosBeetle` is more restrictive, and the return types are not covariant.

Hiding Static Methods

A static method cannot be overridden because class objects do not inherit from each other in the same way as instance objects. On the other hand, they can be hidden. 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 to 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 fifth 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()` declaration in the `Panda` class, then the program prints "Bear is eating" instead.

See if you can figure out why each of the method declarations in the `SunBear` class does not compile:

```
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 SunBear extends Bear {
    public void sneeze() { // DOES NOT COMPILE
        System.out.println("Sun Bear sneezes quietly");
    }
    public static void hibernate() { // DOES NOT COMPILE
        System.out.println("Sun Bear is going to sleep");
    }
    protected static void laugh() { // DOES NOT COMPILE
        System.out.println("Sun Bear 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. The second method, `hibernate()`, does not compile for the opposite reason. The method is marked `static` in the child class but not in the parent class.

Finally, the `laugh()` method does not compile. Even though both versions of the method are marked `static`, the version in `SunBear` 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.

Hiding Variables

As you saw with method overriding, there are lots 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 much 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); // true
        System.out.println(c.hasFur); // false
    }
}

```

Confused about the output? Both of these classes define a `hasFur` variable, but with different values. Even though only one object is created by the `main()` method, both variables exist independently of each other. The output changes depending on the reference variable used.

If you didn't understand the last example, don't worry. We cover polymorphism in more detail in the next chapter. 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.

Writing *final* Methods

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

Creating Abstract Classes

When designing a model, we sometimes want to create an entity that cannot be instantiated directly. For example, imagine that we have a Canine class with subclasses Wolf, Fox, and Coyote. We want other developers to be able to create instances of the subclasses, but perhaps we don't want them to be able to create a Canine instance. In other words, we want to force all objects of Canine to have a particular type at runtime.

Introducing Abstract Classes

Enter abstract classes. An *abstract class* is a class declared with the abstract modifier that cannot be instantiated directly and may contain abstract methods. Let's take a look at an example based on the Canine data model:

```
public abstract class Canine {}

public class Wolf extends Canine {}
public class Fox extends Canine {}
public class Coyote extends Canine {}
```

In this example, other developers can create instances of Wolf, Fox, or Coyote, but not Canine. Sure, they can pass a variable reference as a Canine, but the underlying object must be a subclass of Canine at runtime.

But wait, there's more! An abstract class can contain abstract methods. An *abstract method* is a method declared with the abstract modifier that does not define a body. Put another way, an abstract method forces subclasses to override the method.

Why would we want this? Polymorphism, of course! By declaring a method abstract, we can guarantee that some version will be available on an instance without having to specify what that version is in the abstract parent class.

```
public abstract class Canine {
    public abstract String getSound();
    public void bark() { System.out.println(getSound()); }
}

public class Wolf extends Canine {
    public String getSound() {
        return "Woouooooof!";
    }
}

public class Fox extends Canine {
    public String getSound() {
        return "Squeak!";
    }
}
```

```
public class Coyote extends Canine {
    public String getSound() {
        return "Roar!";
    } }

```

We can then create an instance of Fox and assign it to the parent type Canine. The overridden method will be used at runtime.

```
public static void main(String[] p) {
    Canine w = new Fox();
    w.bark(); // Squeak!
}

```

Easy so far. But there are some rules you need to be aware of:

- Only instance methods can be marked abstract within a class, not variables, constructors, or static methods.
- An abstract class can include zero or more abstract methods, while a non-abstract class cannot contain any.
- A non-abstract class that extends an abstract class must implement all inherited abstract methods.
- Overriding an abstract method follows the existing rules for overriding methods that you learned about earlier in the chapter.

Let's see if you can spot why each of these class declarations does not compile:

```
public class FennecFox extends Canine {
    public int getSound() {
        return 10;
    } }

```

```
public class ArcticFox extends Canine {}

```

```
public class Direwolf extends Canine {
    public abstract rest();
    public String getSound() {
        return "Roof!";
    } }

```

```
public class Jackal extends Canine {
    public abstract String name;
    public String getSound() {
        return "Laugh";
    } }

```

First off, the FennecFox class does not compile because it is an invalid method override. In particular, the return types are not covariant. The ArcticFox class does not compile because it does not override the abstract getSound() method. The Direwolf class does not compile because it is not abstract but declares an abstract method rest(). Finally, the Jackal class does not compile because variables cannot be marked abstract.

An abstract class is most commonly used when you want another class to inherit properties of a particular class, but you want the subclass to fill in some of the implementation details.

Earlier, we said that an abstract class is one that cannot be instantiated. This means that if you attempt to instantiate it, the compiler will report an exception, as in this example:

```
abstract class Alligator {  
    public static void main(String... food) {  
        var a = new Alligator(); // DOES NOT COMPILE  
    }  
}
```

An abstract class can be initialized, but only as part of the instantiation of a non-abstract subclass.

Declaring Abstract Methods

An abstract method is always declared without a body. It also includes a semicolon (;) after the method declaration. As you saw in the previous example, an abstract class may include non-abstract methods, in this case with the bark() method. In fact, an abstract class can include all of the same members as a non-abstract class, including variables, static and instance methods, constructors, etc.

It might surprise you to know that an abstract class is not required to include any abstract methods. For example, the following code compiles even though it doesn't define any abstract methods:

```
public abstract class Llama {  
    public void chew() {}  
}
```

Even without abstract methods, the class cannot be directly instantiated. For the exam, keep an eye out for abstract methods declared outside abstract classes, such as the following:

```
public class Egret { // DOES NOT COMPILE  
    public abstract void peck();  
}
```

The exam creators like to include invalid class declarations, mixing non-abstract classes with abstract methods.

Like the final modifier, the abstract modifier can be placed before or after the access modifier in class and method declarations, as shown in this Tiger class:

```
abstract public class Tiger {  
    abstract public int claw();  
}
```

The abstract modifier cannot be placed after the class keyword in a class declaration or after the return type in a method declaration. The following Bear and howl() declarations do

not compile for these reasons:

```
public class abstract Bear {    // DOES NOT COMPILE
    public int abstract howl(); // DOES NOT COMPILE
}
```



It is not possible to define an abstract method that has a body or default implementation. You can still define a default method with a body—you just can't mark it as `abstract`. As long as you do not mark the method as `final`, the subclass has the option to override the inherited method.

Creating a Concrete Class

An abstract class becomes usable when it is extended by a concrete subclass. A *concrete class* is a non-abstract class. The first concrete subclass that extends an abstract class is required to implement all inherited abstract methods. This includes implementing any inherited abstract methods from inherited interfaces, as you see in the next chapter.

When you see a concrete class extending an abstract class on the exam, check to make sure that it implements all of the required abstract methods. Can you see why the following walrus class does not compile?

```
public abstract class Animal {
    public abstract String getName();
}

public class Walrus extends Animal {} // DOES NOT COMPILE
```

In this example, we see that `Animal` is marked as `abstract` and `walrus` is not, making `walrus` a concrete subclass of `Animal`. Since `walrus` is the first concrete subclass, it must implement all inherited abstract methods—`getName()` in this example. Because it doesn't, the compiler reports an error with the declaration of `walrus`.

We highlight the *first* concrete subclass for a reason. An abstract class can extend a non-abstract class and vice versa. Anytime a concrete class is extending an abstract class, it must implement all of the methods that are inherited as abstract. Let's illustrate this with a set of inherited classes:

```
public abstract class Mammal {
    abstract void showHorn();
    abstract void eatLeaf();
}

public abstract class Rhino extends Mammal {
```

```

    void showHorn() {} // Inherited from Mammal
}

public class BlackRhino extends Rhino {
    void eatLeaf() {} // Inherited from Mammal
}

```

In this example, the BlackRhino class is the first concrete subclass, while the Mammal and Rhino classes are abstract. The BlackRhino class inherits the eatLeaf() method as abstract and is therefore required to provide an implementation, which it does.

What about the showHorn() method? Since the parent class, Rhino, provides an implementation of showHorn(), the method is inherited in the BlackRhino as a non-abstract method. For this reason, the BlackRhino class is permitted but not required to override the showHorn() method. The three classes in this example are correctly defined and compile.

What if we changed the Rhino declaration to remove the abstract modifier?

```

public class Rhino extends Mammal { // DOES NOT COMPILE
    void showHorn() {}
}

```

By changing Rhino to a concrete class, it becomes the first non-abstract class to extend the abstract Mammal class. Therefore, it must provide an implementation of both the showHorn() and eatLeaf() methods. Since it only provides one of these methods, the modified Rhino declaration does not compile.

Let's try one more example. The following concrete class Lion inherits two abstract methods, getName() and roar():

```

public abstract class Animal {
    abstract String getName();
}

public abstract class BigCat extends Animal {
    protected abstract void roar();
}

public class Lion extends BigCat {
    public String getName() {
        return "Lion";
    }
    public void roar() {
        System.out.println("The Lion lets out a loud ROAR!");
    }
}

```

In this sample code, BigCat extends Animal but is marked as abstract; therefore, it is not required to provide an implementation for the getName() method. The class Lion is not marked as abstract, and as the first concrete subclass, it must implement all of the inherited abstract methods not defined in a parent class. All three of these classes compile successfully.

Creating Constructors in Abstract Classes

Even though abstract classes cannot be instantiated, they are still initialized through constructors by their subclasses. For example, consider the following program:

```
abstract class Mammal {
    abstract CharSequence chew();
    public Mammal() {
        System.out.println(chew()); // Does this line compile?
    }
}

public class Platypus extends Mammal {
    String chew() { return "yummy!"; }
    public static void main(String[] args) {
        new Platypus();
    }
}
```

Using the constructor rules you learned about earlier in this chapter, the compiler inserts a default no-argument constructor into the `Platypus` class, which first calls `super()` in the `Mammal` class. The `Mammal` constructor is only called when the abstract class is being initialized through a subclass; therefore, there is an implementation of `chew()` at the time the constructor is called. This code compiles and prints `yummy!` at runtime.

For the exam, remember that abstract classes are initialized with constructors in the same way as non-abstract classes. For example, if an abstract class does not provide a constructor, the compiler will automatically insert a default no-argument constructor.

The primary difference between a constructor in an abstract class and a non-abstract class is that a constructor in an abstract class can be called only when it is being initialized by a non-abstract subclass. This makes sense, as abstract classes cannot be instantiated.

Spotting Invalid Declarations

We conclude our discussion of abstract classes with a review of potential issues you're more likely to encounter on the exam than in real life. The exam writers are fond of questions with methods marked as `abstract` for which an implementation is also defined. For example, can you see why each of the following methods does not compile?

```
public abstract class Turtle {
    public abstract long eat() // DOES NOT COMPILE
    public abstract void swim() {}; // DOES NOT COMPILE
    public abstract int getAge() { // DOES NOT COMPILE
        return 10;
    }
    public abstract void sleep; // DOES NOT COMPILE
    public void goInShell(); // DOES NOT COMPILE
}
```

The first method, `eat()`, does not compile because it is marked `abstract` but does not end with a semicolon (`;`). The next two methods, `swim()` and `getAge()`, do not compile because they are marked `abstract`, but they provide an implementation block enclosed in braces (`{}`). For the exam, remember that an abstract method declaration must end in a semicolon without any braces. The next method, `sleep`, does not compile because it is missing parentheses, (`()`), for method arguments. The last method, `goInShell()`, does not compile because it is not marked `abstract` and therefore must provide a body enclosed in braces.

Make sure you understand why each of the previous methods does not compile and that you can spot errors like these on the exam. If you come across a question on the exam in which a class or method is marked `abstract`, make sure the class is properly implemented before attempting to solve the problem.

abstract and final Modifiers

What would happen if you marked a class or method both `abstract` and `final`? If you mark something `abstract`, you intend for someone else to extend or implement it. But if you mark something `final`, you are preventing anyone from extending or implementing it. These concepts are in direct conflict with each other.

Due to this incompatibility, Java does not permit a class or method to be marked both `abstract` and `final`. For example, the following code snippet will not compile:

```
public abstract final class Tortoise { // DOES NOT COMPILE
    public abstract final void walk(); // DOES NOT COMPILE
}
```

In this example, neither the class nor the method declarations will compile because they are marked both `abstract` and `final`. The exam doesn't tend to use `final` modifiers on classes or methods often, so if you see them, make sure they aren't used with the `abstract` modifier.

abstract and private Modifiers

A method cannot be marked as both `abstract` and `private`. This rule makes sense if you think about it. How would you define a subclass that implements a required method if the method is not inherited by the subclass? The answer is that you can't, which is why the compiler will complain if you try to do the following:

```
public abstract class Whale {
    private abstract void sing(); // DOES NOT COMPILE
}

public class HumpbackWhale extends Whale {
    private void sing() {
        System.out.println("Humpback whale is singing");
    }
}
```

In this example, the abstract method `sing()` defined in the parent class `Whale` is not visible to the subclass `HumpbackWhale`. Even though `HumpbackWhale` does provide an implementation, it is not considered an override of the abstract method since the abstract method is not inherited. The compiler recognizes this in the parent class and reports an error as soon as `private` and `abstract` are applied to the same method.



While it is not possible to declare a method `abstract` and `private`, it is possible (albeit redundant) to declare a method `final` and `private`.

If we changed the access modifier from `private` to `protected` in the parent class `Whale`, would the code compile?

```
public abstract class Whale {  
    protected abstract void sing();  
}  
  
public class HumpbackWhale extends Whale {  
    private void sing() { // DOES NOT COMPILE  
        System.out.println("Humpback whale is singing");  
    }  
}
```

In this modified example, the code will still not compile, but for a completely different reason. If you remember the rules for overriding a method, the subclass cannot reduce the visibility of the parent method, `sing()`. Because the method is declared `protected` in the parent class, it must be marked as `protected` or `public` in the child class. Even with abstract methods, the rules for overriding methods must be followed.

abstract and static Modifiers

As we discussed earlier in the chapter, a static method can only be hidden, not overridden. It is defined as belonging to the class, not an instance of the class. If a static method cannot be overridden, then it follows that it also cannot be marked `abstract` since it can never be implemented. For example, the following class does not compile:

```
abstract class Hippopotamus {  
    abstract static void swim(); // DOES NOT COMPILE  
}
```

For the exam, make sure you know which modifiers can and cannot be used with one another, especially for abstract classes and interfaces.

Creating Immutable Objects

As you might remember from [Chapter 4](#), an immutable object is one that cannot change state after it is created. The *immutable objects pattern* is an object-oriented design pattern in which an object cannot be modified after it is created.

Immutable objects are helpful when writing secure code because you don't have to worry about the values changing. They also simplify code when dealing with concurrency since immutable objects can be easily shared between multiple threads.

Declaring an Immutable Class

Although there are a variety of techniques for writing an immutable class, you should be familiar with a common strategy for making a class immutable:

1. Mark the class as `final` or make all of the constructors `private`.
2. Mark all the instance variables `private` and `final`.
3. Don't define any setter methods.
4. Don't allow referenced mutable objects to be modified.
5. Use a constructor to set all properties of the object, making a copy if needed.

The first rule prevents anyone from creating a mutable subclass. The second and third rules ensure that callers don't make changes to instance variables and are the hallmarks of good encapsulation, a topic we discuss along with records in [Chapter 7](#).

The fourth rule for creating immutable objects is subtle. Basically, it means you shouldn't expose an accessor (or getter) method for mutable instance fields. Can you see why the following creates a mutable object?

```
import java.util.*;
public final class Animal { // Not an immutable object declaration
    private final ArrayList<String> favoriteFoods;

    public Animal() {
        this.favoriteFoods = new ArrayList<String>();
        this.favoriteFoods.add("Apples");
    }

    public List<String> getFavoriteFoods() {
        return favoriteFoods;
    } }
```

We carefully followed the first three rules, but unfortunately, a malicious caller could still modify our data.

```
var zebra = new Animal();
System.out.println(zebra.getFavoriteFoods()); // [Apples]

zebra.getFavoriteFoods().clear();
zebra.getFavoriteFoods().add("Chocolate Chip Cookies");
System.out.println(zebra.getFavoriteFoods()); // [Chocolate Chip Cookies]
```

Oh no! Zebras should not eat Chocolate Chip Cookies! It's not an immutable object if we can change its contents! If we don't have a getter for the `favoriteFoods` object, how do callers access it? Simple: by using delegate or wrapper methods to read the data.

```
import java.util.*;
public final class Animal { // An immutable object declaration
    private final List<String> favoriteFoods;

    public Animal() {
        this.favoriteFoods = new ArrayList<String>();
        this.favoriteFoods.add("Apples");
    }

    public int getFavoriteFoodsCount() {
        return favoriteFoods.size();
    }

    public String getFavoriteFoodsItem(int index) {
        return favoriteFoods.get(index);
    } }
}
```

In this improved version, the data is still available. However, it is a true immutable object because the mutable variable cannot be modified by the caller.

Copy on Read Accessor Methods

Besides delegating access to any private mutable objects, another approach is to make a copy of the mutable object any time it is requested.

```
public ArrayList<String> getFavoriteFoods() {
    return new ArrayList<String>(this.favoriteFoods);
}
```

Of course, changes in the copy won't be reflected in the original, but at least the original is protected from external changes. This can be an expensive operation if called frequently by the caller.

Performing a Defensive Copy

So, what's this about the fifth and final rule for creating immutable objects? In designing our class, let's say we want a rule that the data for `favoriteFoods` is provided by the caller and that it always contains at least one element. This rule is often called an invariant; it is true any time we have an instance of the object.

```
import java.util.*;
public final class Animal { // Not an immutable object declaration
    private final ArrayList<String> favoriteFoods;

    public Animal(ArrayList<String> favoriteFoods) {
```

```

        if (favoriteFoods == null || favoriteFoods.size() == 0)
            throw new RuntimeException("favoriteFoods is required");
        this.favoriteFoods = favoriteFoods;
    }

    public int getFavoriteFoodsCount() {
        return favoriteFoods.size();
    }

    public String getFavoriteFoodsItem(int index) {
        return favoriteFoods.get(index);
    } }

```

To ensure that `favoriteFoods` is provided, we validate it in the constructor and throw an exception if it is not provided. So is this immutable? Not quite! A malicious caller might be tricky and keep their own secret reference to our `favoriteFoods` object, which they can modify directly.

```

var favorites = new ArrayList<String>();
favorites.add("Apples");

var zebra = new Animal(favorites); // Caller still has access to favorites
System.out.println(zebra.getFavoriteFoodsItem(0)); // [Apples]

favorites.clear();
favorites.add("Chocolate Chip Cookies");
System.out.println(zebra.getFavoriteFoodsItem(0)); // [Chocolate Chip Cookies]

```

Whoops! It seems like `Animal` is not immutable anymore, since its contents can change after it is created. The solution is to make a copy of the list object containing the same elements.

```

    public Animal(List<String> favoriteFoods) {
        if (favoriteFoods == null || favoriteFoods.size() == 0)
            throw new RuntimeException("favoriteFoods is required");
        this.favoriteFoods = new ArrayList<String>(favoriteFoods);
    }

```

The copy operation is called a *defensive copy* because the copy is being made in case other code does something unexpected. It's the same idea as defensive driving: prevent a problem before it exists. With this approach, our `Animal` class is once again immutable.

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 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 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.

We reviewed overloaded, overridden, hidden, and redeclared methods and showed how they differ. 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 then moved on to abstract classes, which are just like regular classes except that they cannot be instantiated and may contain abstract methods. An abstract class can extend a non-abstract class, and vice versa. Abstract classes can be used to define a framework that other developers write subclasses against. An abstract method is one that does not include a body when it is declared. An abstract method can be placed only inside an abstract class or interface. Next, an abstract method can be overridden with another abstract declaration or a concrete implementation, provided the rules for overriding methods are followed. The first concrete class must implement all of the inherited abstract methods, whether they are inherited from an abstract class or an interface.

Finally, this chapter showed you how to create immutable objects in Java. Although there are a number of different techniques to do so, we included the most common one you should know for the exam. Immutable objects are extremely useful in practice, especially in multithreaded applications, since they do not change.

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 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 use `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. Methods marked `final` may not be overridden or hidden.

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 is required. Method overriding is important to polymorphism because it replaces all calls to the method, even those made in a superclass.

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.

Be able to write code that creates and extends abstract classes. In Java, classes and methods can be declared as `abstract`. An abstract class cannot be instantiated. An instance of an abstract class can be obtained only through a concrete subclass. Abstract classes can include any number of abstract and non-abstract methods, including zero. Abstract methods

follow all the method override rules and may be defined only within abstract classes. The first concrete subclass of an abstract class must implement all the inherited methods. Abstract classes and methods may not be marked as `final`.

Create immutable objects. An immutable object is one that cannot be modified after it is declared. An immutable class is commonly implemented with a private constructor, no setter methods, and no ability to modify mutable objects contained within the class.

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) {
        var 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 modifier pairs can be used together in a method declaration? (Choose all that apply.)
 1. `static` and `final`
 2. `private` and `static`
 3. `static` and `abstract`
 4. `private` and `abstract`
 5. `abstract` and `final`
 6. `private` and `final`
 3. Which of the following statements about methods are true? (Choose all that apply.)
 1. Overloaded methods must have the same signature.