Chapter 8 - Class design

Calling the *super* Reference

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

```
1:  class Insect {
2:     protected int numberOfLegs = 4;
3:     String label = "buggy";
4:  }
5:
6:  public class Beetle extends Insect {
7:     protected int numberOfLegs = 6;
8:     short age = 3;
9:     public void printData() {
10:       System.out.print(this.label);   // Perfectly valid. Java looks firstly in current class, than in the p
11:       System.out.print(super.label);
12:       System.out.print(this.age);
13:       System.out.print(super.age);    // This one will not compile. Compiler will check only the parent clas
14:       System.out.print(numberOfLegs);
15:    }
16:    public static void main(String []n) {
17:       new Beetle().printData();
18:    }
19: }
```

That was a trick question—this program code would not compile!

Calling Parent Constructors with *super()*

In Java, the first statement of every constructor is either a call to another constructor within the class, using `this()`, or a call to a constructor in the direct parent class, using `super()`.

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.

Missing a Default No-Argument Constructor

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

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

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

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

Order of Initialization

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

Class Initialization

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

The most important rule with class initialization is that it happens at most once for each class. The class may also never be loaded if it is not used in the program. We summarize the order of initialization for a class as follows:

Initialize Class X

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

Instance Initialization

An instance is initialized anytime the `new` keyword is used. Instance initialization is a bit more complicated than class initialization, because a class or superclass may have many constructors declared but only a handful used as part of instance initialization.

First, start at the lowest-level constructor where the `new` keyword is used. Remember, the first line of every constructor is a call to `this()` or `super()`, and if omitted, the compiler will automatically insert a call to the parent no-argument constructor `super()`. Then, progress upward and note the order of constructors. Finally, initialize each class starting with the superclass, processing each instance initializer and constructor in the reverse order in which it was called. We summarize the order of initialization for an instance as follows:

Initialize Instance of X

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

Reviewing Constructor Rules

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

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

Inheriting Methods

Overriding a Method

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*.

Overriding a Generic Method

Overriding methods is complicated enough, but add generics to it and things only get more challenging.

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

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

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

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

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

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

On the other hand, you can override a method with generic parameters, but **you must match the signature including the generic type exactly.**

Generic Return Types

The generic parameter type must match its parent's type exactly.

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

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

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

Hiding Static Methods

A *hidden method* occurs when a child class defines a `static` method with the same name and signature as an inherited `static` method defined in a parent class.

The previous four rules for overriding a method must be followed when a method is hidden. In addition, a new rule is added for hiding a method:

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

Hiding Variables

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

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

What do you think the following application prints?

```
1  class Carnivore {
2     protected boolean hasFur = false;
3  }
4
5  public class Meerkat extends Carnivore {
6     protected boolean hasFur = true;
7
8     public static void main(String[] args) {
9        Meerkat m = new Meerkat();
10       Carnivore c = m;
11       System.out.println(m.hasFur);
12       System.out.println(c.hasFur);
13    }
14 }
```

It prints `true` followed by `false`

Understanding Polymorphism

Java supports *polymorphism*, the property of an object to take on many different forms. To put this more precisely, a Java object may be accessed using a reference with the same type as the object, a reference that is a superclass of the object, or a reference that defines an interface the object implements, either directly or through a superclass.

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

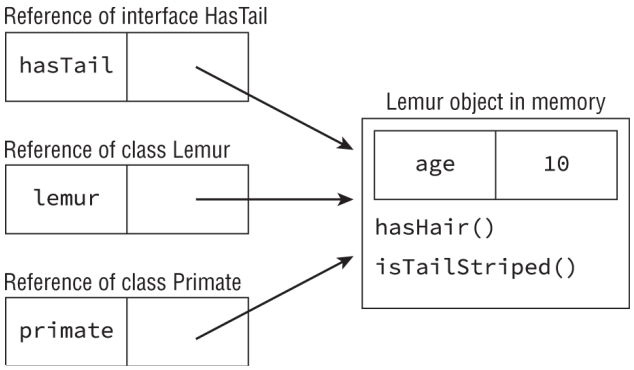
- An interface can define `abstract` methods.
- A class can implement any number of interfaces.
- A class implements an interface by overriding the inherited `abstract` methods.
- An object that implements an interface can be assigned to a reference for that interface.

Object vs. Reference

In Java, all objects are accessed by reference, so as a developer you never have direct access to the object itself. Conceptually, though, you should consider the object as the entity that exists in memory, allocated by the Java runtime environment. Regardless of the type of the reference you have for the object in memory, the object itself doesn't change.

We can summarize this principle with the following two rules:

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



Casting Objects

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

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

The *instanceof* Operator

The `instanceof` operator can be used to check whether an object belongs to a particular class or interface and to prevent `ClassCastException`s at runtime. Just as the compiler does not allow casting an object to unrelated types, it also does not allow `instanceof` to be used with unrelated types. We can demonstrate this with our unrelated `Bird` and `Fish` classes:

```
1    public static void main(String[] args) {
2       Fish fish = new Fish();
3       if (fish instanceof Bird) {  // DOES NOT COMPILE
4          Bird bird = (Bird) fish;  // DOES NOT COMPILE
5       }
6    }
```

Polymorphism and Method Overriding

In Java, polymorphism states that when you override a method, you replace all calls to it, even those defined in the parent class.

As an example, what do you think the following code snippet outputs?

```
1 class Penguin {
2    public int getHeight() { return 3; }
```

```
   public void printInfo() {
      System.out.print(this.getHeight());
   }
}

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

If you said `8`, then you are well on your way to understanding polymorphism.

Overriding vs. Hiding Members

While method overriding replaces the method everywhere it is called, `static` method and variable hiding does not. Strictly speaking, hiding members is not a form of polymorphism since the methods and variables maintain their individual properties. Unlike method overriding, hiding members is very sensitive to the reference type and location where the member is being used.

Let's take a look at an example:

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

The `CrestedPenguin` example is nearly identical to our previous `EmporerPenguin` example, although as you probably already guessed, it prints `3` instead of `8`