## Diseño de clases

## Comprender la herencia

La herencia es el proceso por el cual una subclase incluye automáticamente ciertos miembros de la clase, incluyendo primitivas, objetos o métodos, definidos en la clase padre.

Para fines ilustrativos, nos referimos a cualquier clase que hereda de otra clase como una subclase o clase hija, ya que se considera un descendiente de esa clase. Alternativamente, nos referimos a la clase de la que hereda el hijo como la superclase o clase padre, ya que se considera un ancestro de la clase.

Un aspecto clave de la herencia es que es transitiva. Dadas tres clases [X, Y, Z], si X extiende a Y, e Y extiende a Z, entonces X se considera una subclase o descendiente de Z. Del mismo modo, Z es una superclase o ancestro de X.

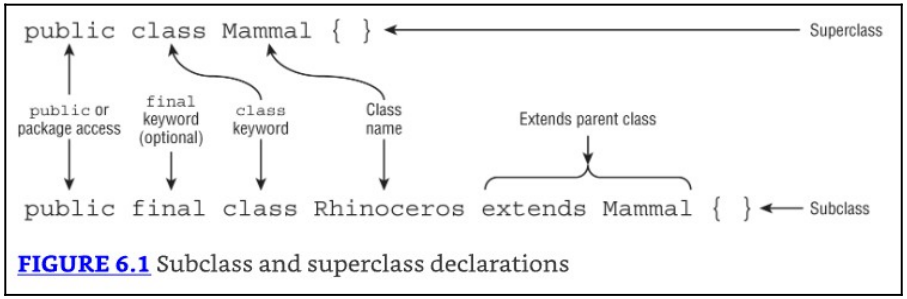


FIGURE 6.1 Subclass and superclass declarations

No necesitamos declarar nada en la superclase aparte de asegurarnos de que no esté marcada como final.

Hay cuatro niveles de acceso: *public*, *protected*, de paquete y *private*. Cuando una clase hereda, todos los miembros *public* y *protected* están automáticamente disponibles como parte de la clase hija. Si las dos clases están en el mismo paquete, entonces los miembros del paquete están disponibles para la clase hija. Por último, pero no menos importante, los miembros *private* solo son accesibles para la clase en la que están definidos y nunca están disponibles a través de la herencia. Esto no significa que la clase padre no pueda tener miembros *private* que puedan contener datos o modificar un objeto; sólo significa que la subclase no tiene acceso directo a ellos.

En el siguiente ejemplo: ***Jaguar*** es una subclase o hijo de ***BigCat***, lo que convierte a ***BigCat*** en una superclase o padre de ***Jaguar***. En la clase ***Jaguar***, se puede acceder a ***size*** porque está marcado como ***protected***. A través de la herencia, la subclase Jaguar puede leer o escribir ***size*** como si fuera su propio miembro. Contrasta esto con la clase ***Spider***, que no tiene acceso a ***size*** ya que no se hereda.

```java
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); // NO COMPILA
    }
}
```

## Modificadores de clase

| Modificador | Descripción |
|---|---|
| final | La clase no puede ser extendida. |
| abstract | La clase es abstracta, puede contener métodos abstractos y requiere una subclase concreta para ser instanciada. |
| sealed | La clase solo puede ser extendida por una lista específica de clases. |
| non-sealed | Una subclase de una clase sealed permite subclases potencialmente no nombradas. |
| static | Usado para clases anidadas estáticas definidas dentro de una clase. |

***final***: El modificador final evita que una clase se extienda más.

```java
public final class Rhinoceros extends Mammal {}
public class Clara extends Rhinoceros {} // ¡NO COMPILA!
```

## Single vs. Herencia Múltiple

Java soporta herencia simple, por la cual una clase puede heredar de una sola clase padre directa.

Java también soporta múltiples niveles de herencia, por la cual una clase puede extender otra clase, que a su vez extiende otra clase. Puedes haber cualquier número de niveles de herencia.

Para entender realmente la herencia simple, puede ser útil contrastarla con la herencia múltiple, por la cual una clase puede tener múltiples padres directos. Por diseño, Java no soporta la herencia múltiple porque puede llevar a modelos de datos complejos y a menudo difíciles de mantener. Java permite una excepción a la regla de la herencia simple (una clase puede implementar múltiples interfaces)

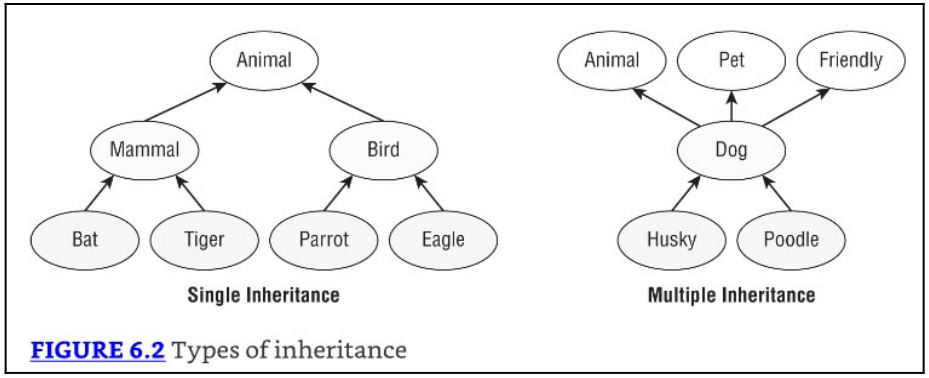


**FIGURE 6.2** Types of inheritance

## Heredando de Object

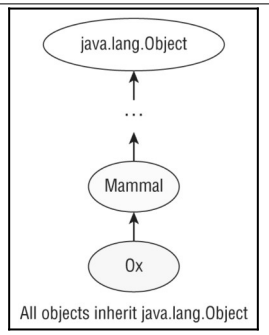
A lo largo de nuestra discusión de Java en este libro, hemos usado la palabra "objeto" numerosas veces, y con razón. En Java, todas las clases heredan de una única clase: ***java.lang.Object***, o ***Object*** para abreviar. Además, ***Object*** es la única clase que no tiene una clase padre.

Puede que te preguntes: "Ninguna de las clases que he escrito hasta ahora extiende ***Object***, así que ¿cómo heredan todas las clases de ella?". La respuesta es que el compilador ha estado insertando código automáticamente en cualquier clase que escribas que no extienda de una clase específica. Por ejemplo, los dos siguientes son equivalentes:

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

La clave es que cuando Java ve que defines una clase que no extiende otra clase, el compilador añade automáticamente la sintaxis ***extends java.lang.Object*** a la definición de la clase.

Por otro lado, cuando defines una nueva clase que extiende una clase existente, Java no extiende automáticamente la clase ***Object***. Dado que todas las clases heredan de ***Object***, extender una clase existente significa que el hijo ya hereda de ***Object*** por definición. Si observas la estructura de herencia de cualquier clase, siempre terminará con ***Object*** en la parte superior del árbol, como se muestra en la Figura 6.3.




## Creando clases

Ahora que hemos establecido cómo funciona la herencia en Java, podemos usarla para definir y crear relaciones de clases complejas.

```
1  // Animal.java
2  public class Animal {
3     private int age;
4     protected String name;
5     public int getAge() {
6        return age;
7     }
8     public void setAge(int newAge) {
9        age = newAge;
10    }
11 }
12
13 // Lion.java
14 public class Lion extends Animal {
15    protected void setProperties(int age, String n) {
16       setAge(age);
17       name = n;
18    }
19    public void roar() {
20       System.out.print(name + ", age " + getAge() + ", says: Roar!");
21    }
22    public static void main(String[] args) {
23       var lion = new Lion();
24       lion.setProperties(3, "Kion");
25       lion.roar();
26    }
27 }
```

La variable de instancia ***age*** está marcada como ***private*** y no es accesible directamente desde la subclase ***Lion***. Por lo tanto, el siguiente código no compilaría:

```
1 public class Lion extends Animal {
2    public void roar() {
3       System.out.print("Lions age: " + age); // DOES NOT COMPILE
4    }
5 }
```

Esto se debe a que los miembros ***private*** nunca se heredan, y los miembros del paquete solo se heredan si las dos clases están en el mismo paquete.

## Aplicando Modificadores de Acceso a Clases

Al igual que con las variables y los métodos, puedes aplicar modificadores de acceso a las clases. Como recordarás del Capítulo 1, una clase de nivel superior es aquella que no está definida dentro de otra clase. También recuerda que un archivo Java puede tener como máximo una clase de nivel superior.

Si bien solo puedes tener una clase de nivel superior, puedes tener tantas clases (en cualquier orden) con acceso de paquete como desees. De hecho, ¡ni siquiera necesitas declarar una clase pública! El siguiente código declara tres clases, cada una con acceso de paquete:

```
1 // Bear.java
2 class Bird {}
3 class Bear {}
4 class Fish {}
```

Sin embargo, intentar declarar una clase de nivel superior con la palabra clave ***protected*** o ***private*** dará lugar a un error de compilación:

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

## Accediendo a la Referencia con *this*

¿Qué sucede cuando un parámetro de método tiene el mismo nombre que una variable de instancia existente? Veamos un ejemplo. ¿Qué crees que imprime el siguiente programa?

```
1  public class Flamingo {
2     private String color = null;
3     public void setColor(String color) {
4        color = color;
5     }
6     public static void main(String... unused) {
7        var f = new Flamingo();
8        f.setColor("PINK");
9        System.out.print(f.color);
10    }
11 }
```

Si dijiste ***null***, entonces estás en lo correcto. Java usa el ámbito más granular, por lo que cuando ve ***color = color***, piensa que estás asignando el valor del parámetro del método a sí mismo (no a la variable de instancia). La asignación se completa correctamente dentro del método, pero el valor de la variable de instancia color nunca se modifica y es ***null*** cuando se imprime en el método ***main()***.

La solución cuando tienes una variable local con el mismo nombre que una variable de instancia es usar la palabra clave ***this***. La referencia ***this*** se refiere a la instancia actual de la clase y se puede usar para acceder a cualquier miembro de la clase, incluyendo los miembros heredados. Se puede usar en cualquier método de instancia, constructor o bloque inicializador de instancia. No se puede usar cuando no hay una instancia implícita de la clase, como en un método estático o un bloque inicializador estático. Aplicamos esto a nuestra implementación de método anterior de la siguiente manera:

```
1 public void setColor(String color) {
2    this.color = color; // Establece la variable de instancia con el parámetro del método
3 }
```

El código corregido ahora imprimirá PINK como se esperaba. En muchos casos, la referencia ***this*** es opcional. Si Java encuentra una variable o método que no puede encontrar, verificará la jerarquía de clases para ver si está disponible.

En el siguiente código define una clase llamada ***Duck*** con tres variables de instancia: ***color***, ***height*** y ***length***. El método ***setData*** intenta asignar valores a estas variables, pero hay un error en la línea ***length = this.length;*** que causa que el valor del parámetro ***length*** se asigne a sí mismo en lugar de a la variable de instancia.

En el método ***main***, se crea un objeto ***Duck*** y se llama a ***setData***. Luego, se imprime el estado del objeto, mostrando que ***length*** permanece en 0 debido al error en la asignación.

La salida del código es: 0 2 white

```
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    }
17 }
```

## Llamando a la Referencia con *super*

En Java, una variable o método puede definirse tanto en una clase padre como en una clase hija. Esto significa que la instancia del objeto realmente contiene dos copias de la misma variable con el mismo nombre subyacente. Cuando esto sucede, ¿cómo hacemos referencia a la versión en la clase padre en lugar de la clase actual? Veamos un ejemplo.

```
1   // Reptile.java
2   public class Reptile {
3      protected int speed = 10;
4   }
5
6   // Crocodile.java
7   public class Crocodile extends Reptile {
8      protected int speed = 20;
9
10     public int getSpeed() {
11        return speed;
12     }
13
14     public static void main(String[] data) {
15        var croc = new Crocodile();
16        System.out.println(croc.getSpeed()); // 20
17     }
18  }
```

Una de las cosas más importantes para recordar sobre este código es que una instancia de ***Crocodile*** almacena dos valores separados para ***speed***: uno a nivel de ***Reptile*** y otro a nivel de ***Crocodile***. En la línea 11, Java primero verifica si hay una variable local o parámetro de método llamado ***speed***. Como no lo hay, entonces comprueba ***this.speed;*** y como existe, el programa imprime 20.

Pero, ¿qué pasa si queremos que el programa imprima el valor en la clase ***Reptile***? Dentro de la clase ***Crocodile***, podemos acceder al valor padre de ***speed*** utilizando la referencia o palabra clave ***super***. La referencia ***super*** es similar a la referencia ***this***, excepto que excluye cualquier miembro encontrado en la clase actual. En otras palabras, el miembro debe ser accesible a través de la herencia.

Veamos si has comprendido el uso de ***this*** y ***super***. ¿Qué imprime el siguiente programa?

```
1  class Insect {
2     protected int numberOfLegs = 4;
3     String label = "buggy";
4  }
5  public class Beetle extends Insect {
6     protected int numberOfLegs = 6;
7     short age = 3;
8
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
17    public static void main(String[] m) {
18       new Beetle().printData();
19    }
20 }
```

¡Esa era una pregunta con trampa! Este código no compilaría.

Como ***label*** está definida en la clase padre, es accesible a través de las referencias ***this*** y ***super***. Por esta razón, las líneas 10 y 11 compilarían y ambas imprimirían ***"buggy"*** si la clase compilara.
Por otro lado, la variable ***age*** solo está definida en la clase actual, lo que la hace accesible a través de ***this*** pero no de ***super***. Por esta razón, la línea 12 compila (e imprimiría 3), pero la línea 13 no.

Recuerda, mientras que ***this*** incluye miembros actuales y heredados, ***super*** solo incluye miembros heredados.

Por último, pero no menos importante, ¿qué imprimiría la línea 14 si la línea 13 estuviera comentada? Aunque ambas variables ***numberOfLegs*** son accesibles en ***Beetle***, Java verifica hacia afuera, comenzando con el ámbito más estrecho. Por esta razón, se utiliza el valor de ***numberOfLegs*** en la clase ***Beetle*** y se imprime 6. En este ejemplo, ***this.numberOfLegs*** y ***super.numberOfLegs*** se refieren a variables diferentes con valores distintos.

Como ***this*** incluye miembros heredados, a menudo solo usas ***super*** cuando tienes un conflicto de nombres a través de la herencia. Por ejemplo, tienes un método o variable definido en la clase actual que coincide con un método o variable en una clase padre.

## Declarando Constructores

Un constructor es un método especial que coincide con el nombre de la clase y no tiene tipo de retorno.

```
1  public class Bunny {
2      public Bunny() {
3          System.out.print("hop");
4      }
5  }
```

El nombre del constructor, ***Bunny***, coincide con el nombre de la clase, ***Bunny***, y no hay tipo de retorno, ni siquiera ***void***. Eso lo convierte en un constructor.

```
1  public class Bunny {
2      public bunny() {} // NO COMPILA
3      public void Bunny() {}
4  }
```

El primero no coincide con el nombre de la clase porque Java distingue entre mayúsculas y minúsculas. Como no coincide, Java sabe que no puede ser un constructor y se supone que es un método regular. Sin embargo, falta el tipo de retorno y no compila. El segundo método es un método perfectamente bueno, pero no es un constructor porque tiene un tipo de retorno.

Al igual que los parámetros de método, los parámetros de constructor pueden ser cualquier clase, array o tipo primitivo válido, incluyendo genéricos, pero no pueden incluir ***var***. Por ejemplo, lo siguiente no compila:

```
1  public class Bonobo {
2      public Bonobo (var food) { // NO COMPILA
3      }
4  }
```

Una clase puede tener múltiples constructores, siempre y cuando cada constructor tenga una firma de constructor única. En este caso, eso significa que los parámetros del constructor deben ser distintos. Al igual que los métodos con el mismo nombre pero diferentes firmas, declarar múltiples constructores con diferentes firmas se conoce como sobrecarga de constructores. La siguiente clase Turtle tiene cuatro constructores sobrecargados distintos:

```
1  public class Turtle {
2
3      private String name;
4
5      public Turtle() {
6          name = "John Doe";
7      }
8
9      public Turtle(int age) {}
10
11     public Turtle(long age) {}
12
13     public Turtle(String newName, String... favoriteFoods) {
14         name = newName;
15     }
16 }
```

Los constructores se utilizan al crear un nuevo objeto. Este proceso se llama instanciación porque crea una nueva instancia de la clase. Se llama a un constructor cuando escribimos ***new*** seguido del nombre de la clase que queremos instanciar. Aquí tienes un ejemplo:

```
1  new Turtle(15)
```

Cuando Java ve la palabra clave ***new***, asigna memoria para el nuevo objeto. Luego busca un constructor con una firma coincidente y lo llama.

## El Constructor Predeterminado

El constructor predeterminado tiene una lista de parámetros vacía y un cuerpo vacío. Está bien que lo escribas tú mismo. Sin embargo, como no hace nada, Java se complace en generarlo por ti y ahorrarte algo de escritura.

Todas las clases en Java tienen un constructor, ya sea que lo codifiques o no. Si no incluyes ningún constructor en la clase, Java creará uno por ti sin parámetros. Este constructor creado por Java se llama constructor predeterminado y se agrega cada vez que se declara una clase sin constructores. A menudo nos referimos a él como el constructor predeterminado sin argumentos. Aquí tienes un ejemplo:

```java
public class Conejo {
   public static void main(String[] args) {
      new Conejo(); // Llama al constructor predeterminado
   }
}
```

En la clase ***Conejo***, Java ve que no se codificó ningún constructor y crea uno. La clase anterior es equivalente a la siguiente, en la que se proporciona el constructor predeterminado y, por lo tanto, no lo inserta el compilador:

```java
public class Conejo {
   public Conejo() {} // Constructor predeterminado

   public static void main(String[] args) {
      new Conejo(); // Llama al constructor predeterminado
   }
}
```

Seguimos diciendo generado. Esto sucede durante el paso de compilación. Si miras el archivo con la extensión **.java**, el constructor seguirá faltando. Solo aparece en el archivo compilado con la extensión **.class**.

Para el examen, una de las reglas más importantes que necesitas saber es que el compilador solo inserta el constructor predeterminado cuando no se definen constructores. ¿Cuál de estas clases crees que tiene un constructor predeterminado?

```java
public class Conejo1 {}

public class Conejo2 {
   public Conejo2() {}
}

public class Conejo3 {
   public Conejo3(boolean b) {}
}

public class Conejo4 {
    private Conejo4() {}
}
```

Solo ***Conejo1*** obtiene un constructor predeterminado sin argumentos. ***Conejo2*** y ***Conejo3*** ya tienen constructores públicos. ***Conejo4*** tiene un constructor privado. Dado que estas tres clases tienen un constructor definido, el constructor predeterminado sin argumentos no se inserta para ti.

Veamos rápidamente cómo llamar a estos constructores:

```java
public class RabbitsMultiply {
 public static void main(String[] args) {
 var r1 = new Rabbit1();
 var r2 = new Rabbit2();
 var r3 = new Rabbit3(true);
 var r4 = new Rabbit4(); // NO COMPILA
 }}
```

La línea 3 llama al constructor predeterminado sin argumentos generado. Las líneas 4 y 5 llaman a los constructores proporcionados por el usuario. La línea 6 no compila. Rabbit4 hizo el constructor privado para que otras clases no puedan llamarlo.

*Tener solo constructores privados en una clase le dice al compilador que no proporcione un constructor predeterminado sin argumentos. También evita que otras clases instancien la clase. Esto es útil cuando una clase solo tiene métodos estáticos o el desarrollador quiere tener control total de todas las llamadas para crear nuevas instancias de la clase.*

## Llamando a Constructores Sobrecargados con *this()*

```java
public class Hamster {
   private String color;
   private int weight;

   public Hamster(int weight, String color) { // Primer constructor
      this.weight = weight;
      this.color = color;
   }

    public Hamster(int weight) { // Segundo constructor
       this.weight = weight;
       color = "brown";
    }
}
```

Uno de los constructores toma un solo parámetro ***int***. El otro toma un ***int*** y un ***String***. Estas listas de parámetros son diferentes, por lo que los constructores se sobrecargan correctamente.

Hay un poco de duplicación, ya que ***this.weight*** se asigna de la misma manera en ambos constructores. En programación, incluso un poco de duplicación tiende a convertirse en mucha duplicación a medida que seguimos agregando "solo una cosa más". Entonces, ¿cómo puedes hacer que un constructor llame a otro constructor? Podrías verte tentado de reescribir el primer constructor de la siguiente manera:

```
1 public Hamster(int weight) { // Segundo constructor
2    Hamster(weight, "brown"); // NO COMPILA
3 }
```

Esto no funcionará. Se puede llamar a los constructores solo escribiendo ***new*** antes del nombre del constructor. No son como los métodos normales a los que puedes llamar. ¿Qué pasa si ponemos ***new*** antes del nombre del constructor?

```
1 public Hamster(int weight) { // Segundo constructor
2    new Hamster(weight, "brown"); // Compila, pero crea un objeto adicional
3 }
```

Este intento compila. Sin embargo, no hace lo que queremos. Cuando se llama a este constructor, crea un nuevo objeto con el peso y el color predeterminados. Luego, construye un objeto diferente con el peso y el color deseados. De esta manera, terminamos con dos objetos, uno de los cuales se descarta después de su creación. Eso no es lo que queremos. Queremos que ***weight*** y ***color*** se establezcan en el objeto que estamos tratando de instanciar en primer lugar.

Java proporciona una solución: ***this()*** - sí, la misma palabra clave que usamos para referirnos a los miembros de la instancia, pero con paréntesis. Cuando se usa ***this()*** con paréntesis, Java llama a otro constructor en la misma instancia de la clase.

```
1 public Hamster(int weight) {
2   this(weight, "brown");
3 }
```

Ahora Java llama al constructor que toma dos parámetros, con peso y color establecidos como se espera.

***this vs. this()***: A pesar de usar la misma palabra clave, ***this*** y ***this()*** son muy diferentes. El primero, ***this***, se refiere a una instancia de la clase, mientras que el segundo, ***this()***, se refiere a una llamada de constructor dentro de la clase.

Llamar a ***this()*** tiene una regla, la llamada a ***this()*** debe ser la primera declaración en el constructor. El efecto secundario de esto es que solo puede haber una llamada a ***this()*** en cualquier constructor.

```
1 public Hamster(int weight) {
2  System.out.println("chew");
3  // Establecer peso y color predeterminado
4  this(weight, "brown"); // NO COMPILA
5 }
```

A pesar de que una declaración de impresión en la línea 4 no cambia ninguna variable, sigue siendo una declaración de Java y no se permite insertarla antes de la llamada a ***this()***. El comentario en la línea 5 está bien. Los comentarios no se consideran declaraciones y se permiten en cualquier lugar.

Hay una última regla para los constructores sobrecargados que debes conocer. Considera la siguiente definición de la clase Gopher:

```
1 public class Gopher {
2   public Gopher(int dugHoles) {
3     this(5); // NO COMPILA
4   }
5 }
```

El compilador es capaz de detectar que este constructor se está llamando a sí mismo infinitamente.

```
1 public class Gopher {
2   public Gopher() {
3     this(5); // NO COMPILA
4   }
5   public Gopher(int dugHoles) {
6     this(); // NO COMPILA
7   }
8 }
```

En este ejemplo, los constructores se llaman entre sí, y el proceso continúa infinitamente. Dado que el compilador puede detectar esto, informa un error.

## Llamando a Constructores Padre con *super()*

¿cómo se inicializan los miembros de instancia de la clase padre?

La primera declaración de cada constructor es una llamada a un constructor padre usando ***super()*** u otro constructor en la clase usando ***this()***. Lee la oración anterior dos veces para asegurarte de recordarla. ¡Es realmente importante!

Veamos la clase Animal y su subclase Zebra y veamos cómo sus constructores pueden escribirse correctamente para llamarse entre sí:

```java
public class Animal {
  private int age;
  public Animal(int age) {
    super(); // Se refiere al constructor en java.lang.Object
    this.age = age;
  }
}

public class Zebra extends Animal {
  public Zebra(int age) {
    super(age); // Se refiere al constructor en Animal
  }
  public Zebra() {
    this(4); // Se refiere al constructor en Zebra con argumento int
  }
}
```

En la clase Animal, la primera declaración del constructor es una llamada al constructor padre definido en ***java.lang.Object***, que no toma argumentos. En la segunda clase, Zebra, la primera declaración del primer constructor es una llamada al constructor de Animal, que toma un solo argumento. La clase Zebra también incluye un segundo constructor sin argumentos que no llama a ***super()*** sino que llama al otro constructor dentro de la clase Zebra usando ***this(4)***.

## super vs. super()

Al igual que ***this*** y ***this()***, ***super*** y ***super()*** no están relacionados en Java. El primero, ***super***, se utiliza para hacer referencia a los miembros de la clase padre, mientras que el segundo, ***super()***, llama a un constructor padre. Cada vez que veas la palabra clave ***super*** en el examen, asegúrate de que se esté utilizando correctamente.

Al igual que llamar a ***this()***, llamar a ***super()*** solo se puede usar como la primera declaración del constructor. Por ejemplo, las siguientes dos definiciones de clase no compilarán:

```java
public class Zoo {
  public Zoo() {
    System.out.println("Zoo creado");
    super(); // NO COMPILA
  }
}
```

```java
public class Zoo {
  public Zoo() {
   super();
   System.out.println(“Zoo created”);
    super(); // No compila
  }
}
```

La primera clase no compilará porque la llamada al constructor principal debe ser la primera declaración del constructor. En el segundo fragmento de código, ***super()*** es la primera declaración del constructor, pero también se utiliza como la tercera declaración. Dado que solo se puede llamar a ***super()*** una vez como la primera declaración del constructor, el código no compilará.

Si la clase principal tiene más de un constructor, la clase secundaria puede usar cualquier constructor principal válido en su definición, como se muestra en el siguiente ejemplo:

```java
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, "Gorila"); // Llama al primer constructor de Animal
  }

  public Gorilla() {
    super(5); // Llama al segundo constructor de Animal
  }
}
```

En este ejemplo, el primer constructor secundario toma un argumento, ***age***, y llama al constructor principal, que toma dos argumentos, ***age*** y ***name***. El segundo constructor secundario no toma argumentos y llama al constructor principal, que toma un argumento, ***age***. En este ejemplo, observe que los constructores secundarios no están obligados a llamar a constructores principales coincidentes. Cualquier constructor principal válido es aceptable siempre y cuando se proporcionen los parámetros de entrada apropiados al constructor principal.

## Entendiendo las Mejoras del Compilador

El compilador Java inserta automáticamente una llamada al constructor sin argumentos ***super()*** si no llamas explícitamente a ***this()*** o ***super()*** como la primera línea de un constructor. Por ejemplo, las siguientes tres definiciones de clase y constructor son equivalentes, porque el compilador las convertirá automáticamente al último ejemplo:

```
1  public class Donkey {}
2
3  public class Donkey {
4    public Donkey() {}
5  }
6
7  public class Donkey {
8    public Donkey() {
9      super();
10   }
11 }
```

Hemos presentado muchas reglas hasta ahora, y es posible que hayas notado algo. Digamos que tenemos una clase que no incluye un constructor sin argumentos. ¿Qué pasa si definimos una subclase sin constructores, o una subclase con un constructor que no incluye una referencia ***super()***?

```
1  public class Mammal {
2    public Mammal(int age) {}
3  }
4
5  public class Seal extends Mammal {} // NO COMPILA
6
7  public class Elephant extends Mammal {
8    public Elephant() {} // NO COMPILA
9  }
```

La respuesta es que ninguna de las subclases compila. Dado que **Mammal** define un constructor, el compilador no inserta un constructor sin argumentos. El proceso que realiza el compilador Java se tratará en la siguiente sección.

El compilador insertará un constructor predeterminado sin argumentos en ***Seal***, aunque será una implementación simple que solo llama a un constructor principal predeterminado inexistente.

```
1  public class Seal extends Mammal {
2    public Seal() {
3      super(); // NO COMPILA
4    }
5  }
```

Del mismo modo, ***Elephant*** no compilará por razones similares. El compilador no ve una llamada a ***super()*** o ***this()*** como la primera línea del constructor, por lo que inserta automáticamente una llamada a un ***super()*** sin argumentos inexistente.

```
1  public class Elephant extends Mammal {
2    public Elephant() {
3      super(); // NO COMPILA
4    }
5  }
```

En estos casos, el compilador no ayudará y debes crear al menos un constructor en tu clase hija que llame explícitamente a un constructor principal a través del comando ***super()***.

```
1  public class Seal extends Mammal {
2    public Seal() {
3      super(6); // Llamada explícita al constructor principal
4    }
5  }
6
7  public class Elephant extends Mammal {
8    public Elephant() {
9      super(4); // Llamada explícita al constructor principal
10   }
11 }
```

Las subclases pueden incluir constructores sin argumentos incluso si sus clases principales no lo hacen. Por ejemplo, lo siguiente compila porque Elephant incluye un constructor sin argumentos:

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

***super()*** siempre se Refiere al Padre Más Directo: Una clase puede tener múltiples antepasados a través de la herencia. En nuestro ejemplo anterior, ***AfricanElephant*** es una subclase de ***Elephant***, que a su vez es una subclase de ***Mammal***. Sin embargo, para los constructores, ***super()*** siempre se refiere al padre más directo. En este ejemplo, llamar a ***super()*** dentro de la clase ***AfricanElephant*** siempre se refiere a la clase ***Elephant*** y nunca a la clase ***Mammal***.

Concluimos esta sección agregando tres reglas de constructor a tu conjunto de habilidades:

La primera línea de cada constructor es una llamada a un constructor principal usando ***super()*** o un constructor sobrecargado usando ***this()***.

Si el constructor no contiene una referencia ***this()*** o ***super()***, entonces el compilador inserta automáticamente ***super()*** sin argumentos como la primera línea del constructor.

Si un constructor llama a ***super()***, entonces debe ser la primera línea del constructor.

## Inicializando Objetos

## Inicializando Clases

Primero, inicializamos la clase, lo que implica invocar a todos los miembros estáticos en la jerarquía de clases, comenzando con la superclase más alta y trabajando hacia abajo. Esto a veces se conoce como cargar la clase. La (JVM) controla cuándo se inicializa la clase, aunque puedes asumir que la clase se carga antes de que se use. La clase se puede inicializar cuando el programa se inicia por primera vez, cuando se hace referencia a un miembro estático de la clase, o poco antes de que se cree una instancia de la clase.
Una de las reglas más importantes con la inicialización de clases es que ocurre como máximo una vez para cada clase. La clase también puede no cargarse nunca si no se utiliza en el programa. Resumimos el orden de inicialización para una clase de la siguiente manera:

**Inicializar la Clase X**

1. Si existe una superclase Y de X, entonces inicializa la clase Y primero.
2. Procesa todas las declaraciones de variables estáticas en el orden en que aparecen en la clase.
3. Procesa todos los inicializadores estáticos en el orden en que aparecen en la clase.

Echando un vistazo a un ejemplo, ¿qué imprime el siguiente programa?

```
1  public class Animal {
2    static { System.out.print("A"); }
3  }
4
5  public class Hippo extends Animal {
6    public static void main(String[] grass) {
7      System.out.print("C");
8      new Hippo();
9      new Hippo();
10     new Hippo();
11   }
12   static { System.out.print("B"); }
13 }
```

Imprime **ABC** exactamente una vez. Dado que el método ***main()*** está dentro de la clase ***Hippo***, la clase se inicializará primero, comenzando con la superclase e imprimiendo AB. Luego, se ejecuta el método ***main()***, imprimiendo C. Aunque el método ***main()*** crea tres instancias, la clase se carga solo una vez.

**¿Por qué el programa Hippo imprimió C después de AB?**

En el ejemplo anterior, la clase ***Hippo*** se inicializó antes de que se ejecutara el método ***main()***. Esto sucedió porque nuestro método ***main()*** estaba dentro de la clase que se ejecutaba, por lo que tenía que cargarse al inicio. ¿Qué pasa si en su lugar llamaste a ***Hippo*** dentro de otro programa?

```
1  public class HippoFriend {
2    public static void main(String[] grass) {
3      System.out.print("C");
4      new Hippo();
5    }
6  }
```

Asumiendo que la clase no se referencia en ningún otro lugar, este programa probablemente imprimirá ***CAB***, y la clase ***Hippo*** no se cargará hasta que se necesite dentro del método ***main()***. Decimos "probablemente" porque las reglas sobre cuándo se cargan las clases las determina la JVM en tiempo de ejecución. Para el examen, solo necesitas saber que una clase debe inicializarse antes de que se haga referencia a ella o se utilice. Además, la clase que contiene el punto de entrada del programa, también conocido como el método ***main()***, se carga antes de que se ejecute el método ***main()***.

## Inicialización de campos final

Antes de adentrarnos en el orden de inicialización de los miembros de instancia, necesitamos hablar por un minuto sobre los campos final (variables de instancia). Cuando presentamos las variables de instancia y de clase en el Capítulo 1, te dijimos que se les asigna un valor predeterminado basado en su tipo si no se especifica ningún valor. Por ejemplo, un double se inicializa con 0.0, mientras que una referencia de objeto se inicializa como nula. Sin embargo, un valor predeterminado solo se aplica a un campo no final.

Como viste en el Capítulo 5, las variables estáticas finales deben tener un valor asignado explícitamente exactamente una vez. Los campos marcados como final siguen reglas similares. Se les puede asignar valores en la línea en la que se declaran o en un inicializador de instancia.

```java
public class MouseHouse {
   private final int volumen;
   private final String name = "La casa del ratón"; // Asignación de declaración
   {
      volumen = 10; // Asignación del inicializador de instancia
   }
}
```

Sin embargo, a diferencia de los miembros de clase estáticos, los campos de instancia final también se pueden establecer en un constructor. El constructor es parte del proceso de inicialización, por lo que se nos permite asignar variables de instancia final. Para el examen, necesitas saber una regla importante: para cuando el constructor se completa, a todas las variables de instancia final se les debe asignar un valor exactamente una vez.

```java
public class MouseHouse {
   private final int volume;
   private final String name;
   public MouseHouse() {
      this.name = "Empty House"; // Asignación del constructor
   }

   {
      volume = 10; // Asignación del inicializador de instancia
   }
}
```

A diferencia de las variables locales finales, a las que no se les exige un valor a menos que se utilicen realmente, a las variables de instancia final se les debe asignar un valor. Si no se les asigna un valor cuando se declaran o en un inicializador de instancia, entonces se les debe asignar un valor en la declaración del constructor. Si no lo haces, se producirá un error del compilador en la línea que declara el constructor.

```java
public class MouseHouse {
   private final int volume;
   private final String type;

   { this.volume = 10; }
   public MouseHouse(String type) {
      this.type = type;
   }
   public MouseHouse() { // NO COMPILA
      this.volume = 2; // NO COMPILA
   }
}
```

En este ejemplo, el primer constructor que toma un argumento ***String*** compila. En términos de asignación de valores, cada constructor se revisa individualmente, por lo que el segundo constructor no compila. Primero, el constructor no establece un valor para la variable ***type***. El compilador detecta que nunca se establece un valor para ***type*** e informa un error en la línea donde se declara el constructor. Segundo, el constructor establece un valor para la variable ***volume***, aunque ya se le asignó un valor mediante el inicializador de instancia.

*En el examen, ten cuidado con las variables de instancia marcadas como final. Asegúrate de que se les asigne un valor en la línea donde se declaran, en un inicializador de instancia o en un constructor. Solo se les debe asignar un valor una vez, y si no se les asigna un valor, se considera un error del compilador en el constructor.*

¿Qué pasa con las variables de instancia final cuando un constructor llama a otro constructor en la misma clase? En ese caso, tienes que seguir el flujo cuidadosamente, asegurándote de que a cada variable de instancia final se le asigne un valor exactamente una vez. Podemos reemplazar nuestro constructor incorrecto anterior con el siguiente que sí compila:

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

Este constructor no realiza ninguna asignación a ninguna variable de instancia final, pero llama al constructor ***MouseHouse(String)***, que observamos que compila sin problemas. Usamos ***null*** aquí para demostrar que la variable no necesita ser un valor de objeto. Podemos asignar un valor nulo a las variables de instancia final siempre y cuando se establezcan explícitamente.

## Inicialización de instancias

Primero, comienza en el constructor de nivel más bajo donde se usa la palabra clave new. Recuerda, la primera línea de cada constructor es una llamada a this() o super(), y si se omite, el compilador insertará automáticamente una llamada al constructor sin argumentos principal super(). Luego, avanza hacia arriba y observa el orden de los constructores. Finalmente, inicializa cada clase comenzando con la superclase, procesando cada inicializador de instancia y constructor en el orden inverso en que se llamó. Resumimos el orden de inicialización de una instancia de la siguiente manera:

**Inicializar instancia de X**

1. Inicializa la clase X si no ha sido inicializada previamente.

2. Si hay una superclase Y de X, entonces inicializa primero la instancia de Y.

3. Procesa todas las declaraciones de variables de instancia en el orden en que aparecen en la clase.

4. Procesa todos los inicializadores de instancia en el orden en que aparecen en la clase.
5. Inicializa el constructor, incluyendo cualquier constructor sobrecargado al que se haga referencia con ***this()***.

```
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  7:
8  public ZooTickets() {
9  System.out.print("z-");
10  }
11  11:
12  public static void main(String... patrons) {
13  new ZooTickets();
14  }}
```

La salida es la siguiente: **0-10-BestZoo-z-**

```
1  class Primate {
2     public Primate() {
3        System.out.print("Primate-");
4     }
5  }
7  class Ape extends Primate {
8     public Ape(int fur) {
9        System.out.print("Ape1-");
10    }
11    public Ape() {
12       System.out.print("Ape2-");
13    }
14 }
16 public class Chimpanzee extends Ape {
17    public Chimpanzee() {
18       super(2);
19       System.out.print("Chimpanzee-");
20    }
21    public static void main(String[] args) {
22       new Chimpanzee();
23    }
24 }
```

Primero, tenemos que inicializar la clase. Dado que no hay ninguna superclase declarada, lo que significa que la superclase es **Object**, podemos comenzar con los componentes estáticos de **ZooTickets**. En este caso, se ejecutan las líneas 4, 5 y 6, imprimiendo 0- y 10-. A continuación, inicializamos la instancia creada en la línea 13. De nuevo, como no se declara ninguna superclase, comenzamos con los componentes de la instancia. Se ejecutan las líneas 2 y 3, lo que imprime BestZoo-. Finalmente, ejecutamos el constructor en las líneas 8-10, lo que genera z-.

A continuación, probemos un ejemplo sencillo con herencia:

El compilador inserta el comando super() como la primera declaración de los constructores Primate y Ape. El código se ejecutará con los constructores principales llamados primero y producirá la siguiente salida:

Primate-Ape1-Chimpanzee-

Observa que solo se llama a uno de los dos constructores ***Ape()***. Necesitas comenzar con la llamada a ***new Chimpanzee()*** para determinar qué constructores se ejecutarán. Recuerda, los constructores se ejecutan de abajo hacia arriba, pero como la primera línea de cada constructor es una llamada a otro constructor, el flujo termina con el constructor padre ejecutado antes que el constructor hijo.

El siguiente ejemplo es un poco más difícil. ¿Qué crees que pasa aquí?

```
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  7:
8  public Cuttlefish() {
9  System.out.println("Constructor");
10  }
11  11:
12  public static void main(String[] args) {
13  System.out.println("Ready");
14  new Cuttlefish();
15  }}
```

Se imprime:
0
Ready
swimmy
1

Constructor

Sin superclase declarada, podemos omitir cualquier paso relacionado con la herencia. Primero procesamos las variables estáticas y los inicializadores estáticos—líneas 4 y 5, con la línea 5 imprimiendo 0. Ahora que los inicializadores estáticos están fuera del camino, el método main() puede ejecutarse, lo que imprime Listo. Luego creamos una instancia declarada en la línea 14. Las líneas 2, 3 y 6 se procesan, con la línea 3 imprimiendo swimmy y la línea 6 imprimiendo 1. Finalmente, se ejecuta el constructor en las líneas 8-10, lo que imprime Constructor.

"Listo para un ejemplo más difícil, del tipo que podrías ver en el examen?
¿Cuál es la siguiente salida?"

```
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 : }
```

El programa imprime lo siguiente:
AFBECHG
BECHG

Vamos a revisarlo. Comienza inicializando la clase Okapi. Dado que tiene una superclase GiraffeFamily, inicialízala primero, imprimiendo A en la línea 2. Luego, inicializa la clase Okapi, imprimiendo F en la línea 19.

Después de que las clases son inicializadas, ejecuta el método main() en la línea 27. La primera línea del método main() crea un nuevo objeto Okapi, activando el proceso de inicialización de instancia. Según la primera regla, la instancia de la superclase GiraffeFamily se inicializa primero. Según nuestra tercera regla, el inicializador de instancia en la superclase GiraffeFamily es llamado, y B se imprime en la línea 3. Según la cuarta regla, inicializamos los constructores. En este caso, esto implica llamar al constructor en la línea 5, que a su vez llama al constructor sobrecargado en la línea 14. El resultado es que EC se imprime, ya que los cuerpos de los constructores se desarrollan en el orden inverso en que fueron llamados.

El proceso continúa con la inicialización de la instancia de Okapi. Según la tercera y cuarta regla, H se imprime en la línea 25, y G se imprime en la línea 23, respectivamente. El proceso es mucho más simple cuando no tienes que llamar a ningún constructor sobrecargado. La línea 29 inserta un salto de línea en la salida. Finalmente, la línea 30 inicializa un nuevo objeto Okapi. El orden y la inicialización son los mismos que en la línea 28, sin la inicialización de la clase, por lo que se imprime BECHG de nuevo. Observa que D nunca se imprime, ya que solo dos de los tres constructores en la superclase GiraffeFamily son llamados.

Este ejemplo es complicado por varias razones. Hay múltiples constructores sobrecargados, muchos inicializadores y una ruta de constructor compleja para seguir. Afortunadamente, preguntas como esta son poco comunes en el examen. Si ves una, simplemente escribe lo que está sucediendo mientras lees el código.

Concluimos esta sección listando reglas importantes que debes conocer para el examen:

- Una clase es inicializada como máximo una vez por la JVM antes de ser referenciada o usada.
- Todas las variables static final deben ser asignadas un valor exactamente una vez, ya sea cuando son declaradas o en un inicializador estático.
- Todos los campos final deben ser asignados un valor exactamente una vez, ya sea cuando son declarados, en un inicializador de instancia, o en un constructor.
- Las variables estáticas y de instancia no final definidas sin un valor son asignadas un valor por defecto basado en su tipo.
- El orden de inicialización es el siguiente: declaraciones de variables, luego inicializadores, y finalmente constructores.

## Heredando Miembros

Ahora que hemos creado una clase, ¿qué podemos hacer con ella? Una de las mayores fortalezas de Java es aprovechar su modelo de herencia para simplificar el código. Por ejemplo, digamos que tienes cinco clases, cada una de las cuales extiende de la clase Animal. Además, cada clase define un método eat() con una implementación idéntica. En este escenario, es mucho mejor definir eat() una vez en la clase Animal que tener que mantener el mismo método en cinco clases separadas.

Heredar una clase no solo otorga acceso a los métodos heredados en la clase padre, sino que también prepara el escenario para colisiones entre métodos definidos tanto en la clase padre como en la subclase. En esta sección, revisamos las reglas para la herencia de métodos y cómo Java maneja tales escenarios.

Nos referimos a la capacidad de un objeto para tomar muchas formas diferentes como polimorfismo. Cubriremos esto más en el siguiente capítulo, pero por ahora solo necesitas saber que un objeto puede ser usado en una variedad de formas, en parte basado en la variable de referencia usada para llamar al objeto.

## Sobrescribiendo un Método

¿Qué pasa si un método con la misma firma se define tanto en la clase padre como en la hija? Por ejemplo, puedes querer definir una nueva versión del método y hacer que se comporte de manera diferente para esa subclase. La solución es sobrescribir el método en la clase hija. En Java, sobrescribir un método ocurre cuando una subclase declara una nueva implementación para un método heredado con la misma firma y tipo de retorno compatible.

NOTA

Recuerda que una firma de método está compuesta por el nombre del método y los parámetros del método. No incluye el tipo de retorno, modificadores de acceso, especificadores opcionales, o cualquier excepción declarada.

Cuando sobrescribes un método, aún puedes hacer referencia a la versión del padre del método usando la palabra clave super. De esta manera, las palabras clave this y super te permiten seleccionar entre la versión actual y la versión del padre de un método, respectivamente. Ilustramos esto con el siguiente ejemplo:

```
1  public class Marsupial {
2     public double getAverageWeight() {
3        return 50;
4     }
5  }
6
7  public class Kangaroo extends Marsupial {
8     public double getAverageWeight() {
9        return super.getAverageWeight() + 20;
10    }
11
12    public static void main(String[] args) {
13       System.out.println(new Marsupial().getAverageWeight()); // 50.0
14       System.out.println(new Kangaroo().getAverageWeight()); // 70.0
15    }
16 }
```

En este ejemplo, la clase Kangaroo sobrescribe el método getAverageWeight() pero en el proceso llama a la versión del padre usando la referencia super.

**Sobrescritura de Métodos y Llamadas Infinitas**

Podrías estar preguntándote si el uso de ***super*** en el ejemplo anterior era necesario. Por ejemplo, ¿qué salida produciría el siguiente código si eliminásemos la palabra clave ***super***?

```
1  public double getAverageWeight() {
2     return getAverageWeight() + 20; // StackOverflowError
3  }
```

En este ejemplo, el compilador no llamaría al método del padre Marsupial; llamaría al método actual Kangaroo. La aplicación intentará llamarse a sí misma infinitamente y producirá un StackOverflowError en tiempo de ejecución.

Para sobrescribir un método, debes seguir una serie de reglas. El compilador realiza las siguientes comprobaciones cuando sobrescribes un método:

1. El método en la clase hija debe tener la misma firma que el método en la clase padre.
2. El método en la clase hija debe ser al menos tan accesible como el método en la clase padre.
3. El método en la clase hija no puede declarar una excepción verificada (checked exception) que sea nueva o más amplia que la clase de cualquier excepción declarada en el método de la clase padre.
4. Si el método devuelve un valor, debe ser el mismo o un subtipo del método en la clase padre, conocido como tipos de retorno covariantes.

Si bien estas reglas pueden parecer confusas o arbitrarias al principio, son necesarias para la consistencia. Sin estas reglas en su lugar, es posible crear contradicciones dentro del lenguaje Java.

**Regla #1: Firmas de Métodos**

La primera regla para sobrescribir un método es autoexplicativa. Si dos métodos tienen el mismo nombre pero diferentes firmas, los métodos están sobrecargados, no sobrescritos. Los métodos sobrecargados se consideran independientes y no comparten las mismas propiedades polimórficas que los métodos sobrescritos.

**NOTA**

Cubrimos la sobrecarga de un método en el Capítulo 5, y es similar a la sobrescritura de un método, ya que ambos implican definir un método usando el mismo nombre. La **sobrecarga** difiere de la **sobrescritura** en que los métodos sobrecargados usan una lista de parámetros diferente. Para el examen, es importante que entiendas esta distinción y que los métodos sobrescritos tienen la misma firma y muchas más reglas que los métodos sobrecargados.

**Regla #2: Modificadores de Acceso**

¿Cuál es el propósito de la segunda regla sobre los modificadores de acceso? Probemos un ejemplo ilustrativo:

```
1  public class Camel {
2     public int getNumberOfHumps() {
3        return 1;
4     }
5  }
6
7  public class BactrianCamel extends Camel {
8     private int getNumberOfHumps() { // NO COMPILA
9        return 2;
10    }
11 }
```

En este ejemplo, BactrianCamel intenta sobrescribir el método getNumberOfHumps() definido en la clase padre pero falla porque el modificador de acceso private es más restrictivo que el definido en la versión padre del método. Digamos que a BactrianCamel se le permitiera compilar, sin embargo. ¿Compilaría esta clase?

```
1  public class Rider {
2
3     public static void main(String[] args) {
4        Camel c = new BactrianCamel();
5        System.out.print(c.getNumberOfHumps()); //???
6     }
7  }
```

La respuesta es, no lo sabemos. El tipo de referencia para el objeto es ***Camel***, donde el método está declarado ***public***, pero el objeto es realmente una instancia de tipo ***BactrianCamel***, donde el método está declarado ***private***. Java evita este tipo de problemas de ambigüedad limitando la sobrescritura de un método a modificadores de acceso que sean tan accesibles o más accesibles que la versión en el método heredado.

**Regla #3: Excepciones Comprobadas (Checked Exceptions)**

La tercera regla dice que sobrescribir un método no puede declarar nuevas excepciones comprobadas o excepciones comprobadas más amplias que el método heredado. Esto se hace por razones polimórficas similares a la limitación de los modificadores de acceso. En otras palabras, podrías terminar con un objeto que es más restrictivo que el tipo de referencia al que está asignado, resultando en una excepción comprobada que no es manejada o declarada. Una implicación de esta regla es que los métodos sobrescritos son libres de declarar cualquier número de nuevas excepciones no comprobadas.

NOTA
*Si no sabes lo qué es una excepción comprobada o no comprobada, no te preocupes. Cubrimos esto en el Capítulo 11, "Excepciones y Localización". Por ahora, sólo necesitas saber que la regla se aplica sólo a excepciones comprobadas. También es útil saber que tanto IOException como FileNotFoundException son excepciones comprobadas y que FileNotFoundException es una subclase de IOException.*

```
1  public class Reptile {
2     protected void sleep() throws IOException {}
3     protected void hide() {}
4     protected void exitShell() throws FileNotFoundException {}
5  }
6
7  public class GalapagosTortoise extends Reptile {
8     public void sleep() throws FileNotFoundException {}
9     public void hide() throws FileNotFoundException {} // NO COMPILA
10    public void exitShell() throws IOException {} // NO COMPILA
11 }
```

En este ejemplo, tenemos tres métodos sobrescritos. Estos métodos sobrescritos usan el modificador public, más accesible, lo cual está permitido según nuestra segunda regla para métodos sobrescritos. El primer método sobrescrito, sleep(), en GalapagosTortoise compila sin problemas porque la excepción declarada es más específica que la excepción declarada en la clase padre.

El método sobrescrito hide() no compila porque declara una nueva excepción comprobada que no está presente en la declaración del padre. El método sobrescrito exitShell() tampoco compila, ya que IOException es una excepción comprobada más amplia que FileNotFoundException. Revisitamos estas clases de excepción, incluyendo la memorización de cuáles son subclases de otras, en el Capítulo 11.

**Regla #4: Tipos de Retorno Covariantes**

La cuarta y última regla sobre la sobrescritura de un método es probablemente la más complicada, ya que requiere conocer las relaciones entre los tipos de retorno. El método de sobrescritura debe usar un tipo de retorno que sea covariante con el tipo de retorno del método heredado.

Veamos un ejemplo con fines ilustrativos:

```
1  public class Rhino {
2     protected CharSequence getName() {
3        return "rhino";
4     }
5
6     protected String getColor() {
7        return "grey, black, or white";
8     }
9  }
10
11 public class JavanRhino extends Rhino {
12    public String getName() {
13       return "javan rhino";
14    }
15
16    public CharSequence getColor() { // NO COMPILA
17       return "grey";
18    }
19 }
```

La subclase JavanRhino intenta sobrescribir dos métodos de Rhino: getName() y getColor(). Ambos métodos sobrescritos tienen el mismo nombre y firma que los métodos heredados. Los métodos sobrescritos también tienen un modificador de acceso más amplio, public, que los métodos heredados. Recuerda, un modificador de acceso más amplio es aceptable en un método sobrescrito.

Del Capítulo 4, "APIs Centrales", aprendimos que String implementa la interfaz CharSequence, haciendo que String sea un subtipo de CharSequence. Por lo tanto, el tipo de retorno de getName() en JavanRhino es covariante con el tipo de retorno de getName() en Rhino.

Por otro lado, el método sobrescrito getColor() no compila porque CharSequence no es un subtipo de String. Dicho de otra manera, todos los valores String son valores CharSequence, pero no todos los valores CharSequence son valores String. Por ejemplo, un StringBuilder es un CharSequence pero no un String. Para el examen, necesitas saber si el tipo de retorno del método de sobrescritura es el mismo o un subtipo del tipo de retorno del método heredado.

**CONSEJO**
*Una prueba simple para la covariancia es la siguiente: dado un tipo de retorno heredado A y un tipo de retorno de sobrescritura B, ¿puedes asignar una instancia de B a una variable de referencia para A sin un cast? Si es así, entonces son covariantes. Esta regla se aplica a tipos primitivos y tipos de objeto por igual. Si uno de los tipos de retorno es void, entonces ambos deben ser void, ya que nada es covariante con void excepto él mismo.*

Eso es todo lo que necesitas saber sobre la sobrescritura de métodos para este capítulo. En el Capítulo 9, "Colecciones y Genéricos", revisaremos la sobrescritura de métodos que involucran genéricos. ¡Siempre hay más que aprender!

**Marcando Métodos con la Anotación @Override**

Una anotación es una etiqueta de metadatos que proporciona información adicional sobre tu código. Puedes usar la anotación ***@Override*** para decirle al compilador que estás intentando sobrescribir un método.

```
1  public class Fish {
2     public void swim() {};
3  }
4
5  public class Shark extends Fish {
6     @Override
7     public void swim() {};
8  }
```

Cuando se usa correctamente, la anotación no impacta el código. Por otro lado, cuando se usa incorrectamente, esta anotación puede evitar que cometas un error. Lo siguiente no compila debido a la presencia de la anotación ***@Override***:

```java
public class Fish {
   public void swim() {};
}

public class Shark extends Fish {
   @Override
   public void swim(int speed) {}; // NO COMPILA
}
```

El compilador ve que estás intentando una sobrescritura de método y busca una versión heredada de ***swim()*** que tome un valor ***int***. Como el compilador no encuentra una, reporta un error. Aunque no se requiere conocer temas avanzados (como cómo crear anotaciones) para el examen, sí es necesario saber cómo usarlas correctamente.

**Redeclarando Métodos Privados**

¿Qué pasa si intentas sobrescribir un método privado? En Java, no puedes sobrescribir métodos privados ya que no se heredan. Solo porque una clase hija no tenga acceso al método del padre, no significa que la clase hija no pueda definir su propia versión del método. Simplemente significa, estrictamente hablando, que el nuevo método no es una versión sobrescrita del método de la clase padre.

Java te permite redeclarar un nuevo método en la clase hija con la misma firma o una firma modificada que el método en la clase padre. Este método en la clase hija es un método separado e independiente, sin relación con el método de la versión del padre, por lo que ninguna de las reglas para la sobrescritura de métodos es invocada. Por ejemplo, estas dos declaraciones compilan:

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

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

Observa que el tipo de retorno difiere en el método hijo de ***String*** a ***int***. En este ejemplo, el método ***getSize()*** en la clase padre es redeclarado, así que el método en la clase hija es un nuevo método y no una sobrescritura del método en la clase padre.

¿Qué pasaría si el método ***getSize()*** fuera declarado ***public*** en Beetle? En este caso, el método en ***RhinocerosBeetle*** sería una sobrescritura inválida. El modificador de acceso en ***RhinocerosBeetle*** es más restrictivo, y los tipos de retorno no son covariantes.

**Ocultando Métodos Estáticos**

Un método estático no puede ser sobrescrito porque los objetos de clase no heredan entre sí de la misma manera que los objetos de instancia. Por otro lado, pueden ser ocultados. Un método oculto ocurre cuando una clase hija define un método estático con el mismo nombre y firma que un método estático heredado definido en una clase padre. Ocultar métodos es similar, pero no exactamente igual, a sobrescribir métodos. Las cuatro reglas anteriores para sobrescribir un método deben seguirse cuando un método es ocultado. Además, se añade una quinta regla para ocultar un método:

El método definido en la clase hija debe ser marcado como static si está marcado como static en una clase padre.
En pocas palabras, es ocultamiento de método si los dos métodos están marcados como static y sobrescritura de método si no están marcados como static. Si uno está marcado como static y el otro no, la clase no compilará.

Veamos algunos ejemplos de la nueva regla:

```java
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();
   }
}
```

En este ejemplo, el código compila y se ejecuta. El método eat() en la clase Panda oculta el método eat() en la clase Bear, imprimiendo "Panda is chewing" en tiempo de ejecución. Debido a que ambos están marcados como static, esto no se considera un método sobrescrito. Dicho esto, todavía hay algo de herencia ocurriendo. Si eliminas la declaración de eat() en la clase Panda, entonces el programa imprime "Bear is eating".

Observa si puedes averiguar por qué cada una de las declaraciones de método en la clase ***SunBear*** no compila:

```
1  public class Bear {
2     public static void sneeze() {
3        System.out.println("Bear is sneezing");
4     }
5
6     public void hibernate() {
7        System.out.println("Bear is hibernating");
8     }
9
10    public static void laugh() {
11      System.out.println("Bear is laughing");
12    }
13 }
14
15 public class SunBear extends Bear {
16    public void sneeze() { // NO COMPILA
17       System.out.println("Sun Bear sneezes quietly");
18    }
19
20    public static void hibernate() { // NO COMPILA
21       System.out.println("Sun Bear is going to sleep");
22    }
23
24    protected static void laugh() { // NO COMPILA
25       System.out.println("Sun Bear is laughing");
26    }
27 }
```

En este ejemplo, ***sneeze()*** está marcado como ***static*** en la clase padre pero no en la clase hija. El compilador detecta que estás intentando sobrescribir usando un método de instancia. Sin embargo, ***sneeze()*** es un método estático que debería ser ocultado, causando que el compilador genere un error. El segundo método, ***hibernate()***, no compila por la razón opuesta. El método está marcado como ***static*** en la clase hija pero no en la clase padre.

Finalmente, el método ***laugh()*** no compila. Aunque ambas versiones del método están marcadas como ***static***, la versión en ***SunBear*** tiene un modificador de acceso más restrictivo que el que hereda, y rompe la segunda regla para sobrescribir métodos. Recuerda, las cuatro reglas para sobrescribir métodos deben ser seguidas al ocultar métodos estáticos.

## Ocultando Variables

Como viste con la sobrescritura de métodos, hay muchas reglas cuando dos métodos tienen la misma firma y están definidos tanto en la clase padre como en la clase hija. Afortunadamente, las reglas para variables con el mismo nombre en las clases padre e hija son mucho más simples. De hecho, Java no permite que las variables sean sobrescritas. Sin embargo, las variables sí pueden ser ocultadas.

Una variable oculta ocurre cuando una clase hija define una variable con el mismo nombre que una variable heredada definida en la clase padre. Esto crea dos copias distintas de la variable dentro de una instancia de la clase hija: una instancia definida en la clase padre y una definida en la clase hija. Como al ocultar un método estático, no puedes sobrescribir una variable; solo puedes ocultarla. Echemos un vistazo a una variable oculta. ¿Qué crees que imprime la siguiente aplicación?

```
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); // true
12       System.out.println(c.hasFur); // false
13    }
14 }
```

¿Confundido por la salida? Ambas clases definen una variable hasFur, pero con valores diferentes. Aunque solo se crea un objeto por el método main(), ambas variables existen independientemente una de la otra. La salida cambia dependiendo de la variable de referencia utilizada.

Si no entendiste el último ejemplo, no te preocupes. Cubrimos el polimorfismo con más detalle en el siguiente capítulo. Por ahora, solo necesitas saber que sobrescribir un método reemplaza el método padre en todas las variables de referencia (excepto super), mientras que ocultar un método o variable reemplaza el miembro solo si se usa un tipo de referencia hijo.

## Escribiendo Métodos Finales

Concluimos nuestra discusión sobre la herencia de métodos con una regla algo autoexplicativa: los métodos final no pueden ser sobrescritos. Al marcar un método como final, prohíbes que una clase hija reemplace este método. Esta regla está vigente tanto cuando sobrescribes un método como cuando ocultas un método. En otras palabras, no puedes ocultar un método estático en una clase hija si está marcado como final en la clase padre.

```java
public class Bird {
    public final boolean hasFeathers() {
        return true;
    }

    public final static void flyAway() {}
}

public class Penguin extends Bird {
    public final boolean hasFeathers() { // NO COMPILA
        return false;
    }

    public final static void flyAway() {} // NO COMPILA
}
```

En este ejemplo, el método de instancia ***hasFeathers()*** está marcado como final en la clase padre ***Bird***, por lo que la clase hija ***Penguin*** no puede sobrescribir el método padre, resultando en un error de compilación. El método estático ***flyAway()*** también está marcado como final, por lo que no puede ser ocultado en la subclase. En este ejemplo, si el método hijo usa o no la palabra clave final es irrelevante; el código no compilará de ninguna manera.

Esta regla se aplica solo a los métodos heredados. Por ejemplo, si los dos métodos estuvieran marcados como ***private*** en la clase padre ***Bird***, entonces la clase ***Penguin***, tal como está definida, compilaría. En ese caso, los métodos privados serían redeclarados, no sobrescritos ni ocultados.

## Creando Clases Abstractas

Al diseñar un modelo, a veces queremos crear una entidad que no pueda ser instanciada directamente. Por ejemplo, imagina que tenemos una clase Canine con subclases Wolf, Fox, y Coyote. Queremos que otros desarrolladores puedan crear instancias de las subclases, pero tal vez no queremos que puedan crear una instancia de Canine. En otras palabras, queremos forzar que todos los objetos de Canine tengan un tipo particular en tiempo de ejecución.

### Introduciendo Clases Abstractas

Aquí entran las clases abstractas. Una clase abstracta es una clase declarada con el modificador abstract que no puede ser instanciada directamente y puede contener métodos abstractos. Echemos un vistazo a un ejemplo basado en el modelo de datos de Canine:

```java
public abstract class Canine {}

public class Wolf extends Canine {}

public class Fox extends Canine {}

public class Coyote extends Canine {}
```

En este ejemplo, otros desarrolladores pueden crear instancias de Wolf, Fox, o Coyote, pero no de Canine. Claro, pueden pasar una referencia de variable como un Canine, pero el objeto subyacente debe ser una subclase de Canine en tiempo de ejecución.

Pero espera, ¡hay más! Una clase abstracta puede contener métodos abstractos. Un método abstracto es un método declarado con el modificador abstract que no define un cuerpo. Dicho de otra manera, un método abstracto fuerza a las subclases a sobrescribir el método.

¿Por qué querríamos esto? ¡Polimorfismo, por supuesto! Al declarar un método abstracto, podemos garantizar que alguna versión estará disponible en una instancia sin tener que especificar cuál es esa versión en la clase padre abstracta.

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

public class Wolf extends Canine {
    public String getSound() {
        return "Wooo000of!";
    }
}

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

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

Luego podemos crear una instancia de Fox y asignarla al tipo padre Canine. El método sobrescrito será usado en tiempo de ejecución.

```
1 public static void main(String[] p) {
2     Canine w = new Fox();
3     w.bark(); // Squeak!
4 }
```

Fácil hasta ahora. Pero hay algunas reglas que debes tener en cuenta:

Solo los métodos de instancia pueden ser marcados como abstractos dentro de una clase, no las variables, constructores o métodos estáticos.
Un método abstracto solo puede ser declarado en una clase abstracta.
Una clase no abstracta que extiende una clase abstracta debe implementar todos los métodos abstractos heredados.
Sobrescribir un método abstracto sigue las reglas existentes para sobrescribir métodos que aprendiste anteriormente en el capítulo.
Veamos si puedes detectar por qué cada una de estas declaraciones de clase no compila:

```
1  public class FennecFox extends Canine {
2     public int getSound() {
3        return 10;
4     }
5  }
6
7  public class ArcticFox extends Canine {}
8
9  public class Direwolf extends Canine {
10    public abstract rest();
11    public String getSound() {
12       return "Roof!";
13    }
14 }
15
16 public class Jackal extends Canine {
17    public abstract String name;
18    public String getSound() {
19       return "Laugh";
20    }
21 }
```

Primero, la clase FennecFox no compila porque es una sobrescritura de método inválida. En particular, los tipos de retorno no son covariantes. La clase ArcticFox no compila porque no sobrescribe el método abstracto getSound(). La clase Direwolf no compila porque no es abstracta pero declara un método abstracto rest(). Finalmente, la clase Jackal no compila porque las variables no pueden ser marcadas como abstractas.

Una clase abstracta se usa más comúnmente cuando quieres que otra clase herede propiedades de una clase particular, pero quieres que la subclase complete algunos de los detalles de la implementación.

Anteriormente, dijimos que una clase abstracta es aquella que no puede ser instanciada. Esto significa que si intentas instanciarla, el compilador reportará una excepción, como en este ejemplo:

```
1 abstract class Alligator {
2    public static void main(String... food) {
3       var a = new Alligator(); // NO COMPILA
4    }
5 }
```

Una clase abstracta puede ser inicializada, pero solo como parte de la instanciación de una subclase no abstracta.

**Declarando Métodos Abstractos**

Un método abstracto siempre se declara sin un cuerpo. También incluye un punto y coma (;) después de la declaración del método. Como viste en el ejemplo anterior, una clase abstracta puede incluir métodos no abstractos, en este caso con el método bark(). De hecho, una clase abstracta puede incluir todos los mismos miembros que una clase no abstracta, incluyendo variables, métodos estáticos y de instancia, constructores, etc.

Podría sorprenderte saber que no se requiere que una clase abstracta incluya ningún método abstracto. Por ejemplo, el siguiente código compila aunque no define ningún método abstracto:

```
1 public abstract class Llama {
2    public void chew() {}
3 }
```

Incluso sin métodos abstractos, la clase no puede ser instanciada directamente. Para el examen, mantente atento a los métodos abstractos declarados fuera de las clases abstractas, como el siguiente:

```
1 public class Egret { // NO COMPILA
2    public abstract void peck();
3 }
```

A los creadores del examen les gusta incluir declaraciones de clase inválidas, mezclando clases no abstractas con métodos abstractos.

Al igual que el modificador final, el modificador abstract puede colocarse antes o después del modificador de acceso en las declaraciones de clase y método, como se muestra en esta clase Tiger:

```
1 abstract public class Tiger {
2    abstract public int claw();
3 }
```

El modificador abstract no puede colocarse después de la palabra clave class en una declaración de clase o después del tipo de retorno en una declaración de método. Las siguientes declaraciones de Bear y howl() no compilan por estas razones:

```
1 public class abstract Bear { // NO COMPILA
2 }
3
4 public int abstract howl(); // NO COMPILA
```

**NOTA**
*No es posible definir un método abstracto que tenga un cuerpo o una implementación predeterminada. Todavía puedes definir un método predeterminado con un cuerpo, simplemente no puedes marcarlo como abstracto. Siempre y cuando no marques el método como final, la subclase tiene la opción de sobrescribir el método heredado.*

### 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 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.