Herencia

En las secciones anteriores, ha visto que se ha mencionado la herencia varias veces. En el lenguaje Java, las clases se pueden derivar de otras clases, heredando así los campos y métodos de esas clases.

Definiciones: Una clase que se deriva de otra clase se denomina subclase (también clase derivada, clase extendida o clase hija). La clase de la que se deriva la subclase se denomina superclase (también clase base o clase padre).

*A excepción de **Object**, que no tiene superclase, cada clase tiene una y solo una superclase directa (herencia simple). En ausencia de cualquier otra superclase explícita, cada clase es implícitamente una subclase de **Object**.*

*Las clases pueden derivarse de clases que se derivan de clases que se derivan de clases, y así sucesivamente, y en última instancia derivarse de la clase superior, **Object.** Se dice que una clase de este tipo desciende de todas las clases de la cadena de herencia que se remonta a **Object**.*

La idea de la herencia es simple pero poderosa: cuando quieres crear una nueva clase y ya existe una clase que incluye parte del código que deseas, puedes derivar tu nueva clase de la clase existente. Al hacer esto, puedes reutilizar los campos y métodos de la clase existente sin tener que escribirlos (¡y depurarlos!) tú mismo.

Una subclase hereda todos los miembros (campos, métodos y clases anidadas) de su superclase. Los constructores no son miembros, por lo que no son heredados por las subclases, pero el constructor de la superclase puede ser invocado desde la subclase.

La clase ***Object***, definida en el paquete **java.lang**, define e implementa un comportamiento común a todas las clases, incluidas las que usted escribe. En la plataforma Java, muchas clases derivan directamente de ***Object***, otras clases derivan de algunas de esas clases, y así sucesivamente, formando una jerarquía de clases.

En la parte superior de la jerarquía ***Object*** se encuentra la clase más general de todas. Las clases que se encuentran cerca de la parte inferior de la jerarquía proporcionan un comportamiento más especializado.

Un ejemplo de herencia

Aquí está el código de muestra para una posible implementación de una clase Bicycle que se presentó en la sección Clases y Objetos:

```
public class Bicycle {

    // the Bicycle class has three fields
    public int cadence;
    public int gear;
    public int speed;

    // the Bicycle class has one constructor
    public Bicycle(int startCadence, int startSpeed, int startGear) {
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;
    }

    // the Bicycle class has four methods
    public void setCadence(int newValue) {
        cadence = newValue;
    }

    public void setGear(int newValue) {
        gear = newValue;
    }

    public void applyBrake(int decrement) {
        speed -= decrement;
    }

    public void speedUp(int increment) {
        speed += increment;
    }
}
```

Una declaración de clase para una clase MountainBike que es una subclase de Bicycle podría verse así:

```
public class MountainBike extends Bicycle {

    // the MountainBike subclass adds one field
    public int seatHeight;

    // the MountainBike subclass has one constructor
    public MountainBike(int startHeight,
                        int startCadence,
                        int startSpeed,
                        int startGear) {
        super(startCadence, startSpeed, startGear);
        seatHeight = startHeight;
    }

    // the MountainBike subclass adds one method
    public void setHeight(int newValue) {
        seatHeight = newValue;
    }
}
```

MountainBike hereda todos los campos y métodos de Bicycle y agrega el campo seatHeight y un método para configurarlo. A excepción del constructor, es como si hubiera escrito una nueva clase MountainBike completamente desde cero, con cuatro campos y cinco métodos. Sin embargo, no tuvo que hacer todo el trabajo. Esto sería especialmente valioso si los métodos de la clase Bicycle fueran complejos y hubiera llevado un tiempo considerable depurarlos.

Qué puedes hacer en una subclase

Una **subclase** hereda todos los miembros **public** y **protected** de su padre, sin importar en qué paquete se encuentre la subclase. Si la subclase está en el mismo paquete que su padre, también hereda los miembros privados del paquete del padre. Puede utilizar los miembros heredados tal como están, reemplazarlos, ocultarlos o complementarlos con nuevos miembros:

- Los campos heredados se pueden utilizar directamente, como cualquier otro campo.
- Puedes declarar un campo en la subclase con el mismo nombre que el de la superclase, ocultándolo así (no recomendado).
- Puede declarar nuevos campos en la subclase que no estén en la superclase.
- Los métodos heredados se pueden utilizar directamente tal como están.
- Puede escribir un nuevo método de instancia en la subclase que tenga la misma firma que el de la superclase, anulándolo así.
- Puedes escribir un nuevo método estático en la subclase que tenga la misma firma que el de la superclase, ocultándolo así.
- Puede declarar nuevos métodos en la subclase que no estén en la superclase.
- Puede escribir un constructor de subclase que invoque al constructor de la superclase, ya sea implícitamente o utilizando la palabra clave super.

Miembros privados en una superclase

Una subclase no hereda los miembros privados de su clase padre. Sin embargo, si la superclase tiene métodos públicos o protegidos para acceder a sus campos privados, la subclase también puede utilizarlos.

Una clase anidada tiene acceso a todos los miembros privados de la clase que la contiene (campos y métodos). Por lo tanto, una clase anidada pública o protegida heredada por una subclase tiene acceso indirecto a todos los miembros privados de la superclase.

Conversión de Objetos

Hemos visto que un objeto es del tipo de datos de la clase de la que fue instanciado. Por ejemplo, si escribimos

```
1 | public MountainBike myBike = new MountainBike();
```

entonces myBike es de tipo MountainBike.

MountainBike desciende de Bicycle y **Object.** Por lo tanto, a MountainBike es un Bicycle y también es un **Object**, y se puede utilizar donde quiera que se requieran objetos Bicycle u **Object.**

Lo inverso no es necesariamente cierto: una Bicycle puede ser una MountainBike, pero no necesariamente lo es. De manera similar, un **Object** puede ser una Bicycle o una MountainBike, pero no necesariamente lo es.

La conversión muestra el uso de un objeto de un tipo en lugar de otro tipo, entre los objetos permitidos por la herencia y las implementaciones. Por ejemplo, si escribimos

```
Object obj = new MountainBike();
```

Entonces obj es tanto un Object como una MountainBike(hasta que a obj se le asigne otro objeto que no sea una MountainBike). Esto se denomina *conversión implícita*.

Si, por el contrario, escribimos

```
MountainBike myBike = obj;
```

Obtendríamos un error en tiempo de compilación porque el compilador no sabe si obj es una MountainBike. Sin embargo, podemos decirle al compilador que prometemos asignar a MountainBike un obj mediante una conversión explícita:

```
MountainBike myBike = (MountainBike)obj;
```

Esta conversión inserta una comprobación en tiempo de ejecución que indica que a obj se le asigna un valor para MountainBike que el compilador pueda asumir con seguridad que obj es un valor MountainBike. Si obj no es un valor MountainBike en tiempo de ejecución, se generará una excepción.

Nota: *Puede realizar una prueba lógica del tipo de un objeto en particular utilizando el operador instanceof. Esto puede evitarle un error de ejecución debido a una conversión incorrecta. Por ejemplo:*

```
if (obj instanceof MountainBike) {
    MountainBike myBike = (MountainBike)obj;
}
```

Aquí el operador instanceof verifica que obj se refiere a una MountainBike para que podamos realizar la conversión sabiendo que no se lanzará ninguna excepción en tiempo de ejecución.

Herencia múltiple de estado, implementación y tipo

Una diferencia significativa entre las clases y las interfaces es que las clases pueden tener campos, mientras que las interfaces no. Además, se puede crear una instancia de una clase para crear un objeto, lo que no se puede hacer con las interfaces. Un objeto almacena su estado en campos, que se definen en clases. Una de las razones por las que el lenguaje de programación Java no permite extender más de una clase es para evitar los problemas de herencia múltiple de estado, que es la capacidad de heredar campos de varias clases. Por ejemplo, supongamos que se puede definir una

nueva clase que extiende varias clases. Cuando se crea un objeto mediante la instanciación de esa clase, ese objeto heredará los campos de todas las superclases de la clase. ¿Qué sucede si los métodos o constructores de diferentes superclases instancian el mismo campo? ¿Qué método o constructor tendrá prioridad? Como las interfaces no contienen campos, no hay que preocuparse por los problemas que resultan de la herencia múltiple de estado.

La herencia múltiple de implementación es la capacidad de heredar definiciones de métodos de varias clases. Con este tipo de herencia múltiple surgen problemas, como conflictos de nombres y ambigüedad. Cuando los compiladores de lenguajes de programación que admiten este tipo de herencia múltiple encuentran superclases que contienen métodos con el mismo nombre, a veces no pueden determinar a qué miembro o método acceder o invocar. Además, un programador puede introducir involuntariamente un conflicto de nombres al agregar un nuevo método a una superclase. Los métodos predeterminados introducen una forma de herencia múltiple de implementación. Una clase puede implementar más de una interfaz, que puede contener métodos predeterminados que tienen el mismo nombre. El compilador de Java proporciona algunas reglas para determinar qué método predeterminado utiliza una clase en particular.

El lenguaje de programación Java admite la herencia múltiple de tipos, que es la capacidad de una clase de implementar más de una interfaz. Un objeto puede tener varios tipos: el tipo de su propia clase y los tipos de todas las interfaces que implementa la clase. Esto significa que, si se declara que una variable es del tipo de una interfaz, su valor puede hacer referencia a cualquier objeto que se instancia desde cualquier clase que implemente la interfaz.

Al igual que ocurre con la herencia múltiple de implementación, una clase puede heredar distintas implementaciones de un método definido (como `default` o `static`) en las interfaces que extiende. En este caso, el compilador o el usuario deben decidir cuál utilizar.

Bibliografía: https://dev.java/learn/inheritance/what-is-inheritance/