Curso de Java orientado a Android – Parte 2

Introducción

Java es un lenguaje de programación orientado a objetos (OOP). En este post cubriremos las características y principios básicos de la programación orientada a objetos proporcionando algunos ejemplos de código. Este post puede completarse con los siguientes: Pincipios de programación orientada a objetos y Principios del diseño de clases

- Objects
- Classes
- Getters y Setters
- Inheritance
- Keywords this y super
- Interface
- Access Modifiers
- Contructors
- Method overriding and overloading
- Polymorphism

Objects

Un objeto es un conjunto de software de estado y el comportamiento relacionado en memoria. Normalmente se utilizan para representar objetos del mundo real. Los objetos son esenciales para la comprensión OOP. Los objetos del mundo real comparten dos características: estado y el comportamiento. Por ejemplo, un coche tiene un estado (modelo actual , fabricante, color) y el comportamiento (conducción, cambio de marchas …)

El desarrollo de aplicaciones con código orientado a objetos aporta muchos beneficios, incluyendo código de fácil reutilización, ocultación de información, facilidad de depuración…

Classes

Una clase es un prototipo a partir de la cual se crean los objetos. En ella se definen los modelos – estado y el comportamiento de un objeto del mundo real. Las clases proporcionan una forma limpia para modelar el estado y comportamiento de los objetos del mundo real. Podemos distinguir dos propiedades, o secciones, principales a definir en una clase:

- Un conjunto de variables de clase (también llamadas campos)
- Un conjunto de métodos de clase (o funciones).

Para representar el estado de un objeto en clases tenemos las variables de clase. Los comportamientos de los objetos son representados usando métodos. La siguiente es una sencilla clase Java llamada de vehículo.

```
class Vehicle {
        int speed = 0;
        int gear = 1;
```

```
        void changeGear(int newGear) {
                gear = newGear;
        }
        void speedUp(int increment) {
                speed = speed + increment;
        }
        void printStates() {
                  System.out.println(" speed:" + speed + "
gear:" + gear);
        }
}
```

El estado del objeto Vehículo se representa con la velocidad (speed) y las marchas (gear). El comportamiento del objeto se puede cambiar utilizando los dos métodos ChangeGear() y speedUp(). Por último podemos saber cual es el estado actual con el método printStates()

Getters y Setters

Un conjunto de métodos se crean por lo general en una clase para leer/escribir los valores de las variables miembro . Estos son llamados **getters**(se utiliza para obtener los valores) y **setters**(se utiliza para cambiar la valores de variables miembro).

Los Getters y Setters son cruciales en las clases de Java, ya que se utilizan para gestionar el estado de un objeto. En la clase vehículo que hemos visto anteriormente, podemos añadir dos métodos (un getter y un setter) para cada variable miembro. El siguiente es el código completa la clase anteriror con los getters y setters correspondientes a la variables miembro speed y gear:

```
class Vehicle {
        int speed = 0;
        int gear = 1;
        // Start of getters and setters
        public int getSpeed() {
```

```
                return speed;
        }
        public void setSpeed(int s) {
                speed = s;
        }
        public int getGear() {
                return gear;
        }
        public void setGear(int g) {
                gear = g;
        }
        // End of getters and setters
        void changeGear(int newGear) {
                gear = newGear;
        }
        void speedUp(int increment) {
                speed = speed + increment;
        }
        void printStates() {
                  System.out.println(" speed:" + speed + "
gear:" + gear);
        }
}
```

Inheritance

La herencia proporciona un mecanismo poderoso y natural para la organización y estructuración del software. Se establece una relación padre-hijo entre dos objetos diferentes.

La programación orientada a objetos permite que las clases hereden estado y comportamiento de uso común de otras clases. En siguiente ejemplo, vehículo (Vehicle) se convierte en la clase padre (superclass) de camiones (Truck) y coches (Car). En el lenguaje de programación Java, permite que cada clase tenga una superclase directa, y cada superclase puede ser heredada por un número ilomitado de subclases.

```
public class Car extends Vehicle {
```

```
        int numOfSeats;
        //Set of statements defining
        //a car’s state and behavior
}

public class Truck extends Vehicle {
        public int loadWeight;
        //Set of statements defining
        //a truck’s state and behavior
}
```

Ahora la clase Truck y Car comparten el estado y comportamiento de la clase Vehicle.

Keywords this y super

Dos palabras clave de Java que se puede encontrar al escribir que el código de clase con la herencia: **this** y **super**. La palabra clave **this** se utiliza como una referencia a la clase actual, mientras que **super** es una referencia a la clase padre que esta clase ha heredado. En otras palabras, **super** se utiliza para acceders a las variables y métodos miembro de la clase padre.
La palabra reservada **super** es especialmente útil cuando se desea reemplazar el método de la superclase en la clase hijo, pero se desea invocar el método de la superclase. Por ejemplo, en la clase de Car, se puede sobrescribir el método printStates() y a su vez llamar al método printStates de la clase Vehicle ():

```
public class Car extends Vehicle {
        int numOfSeats;
        void printStates() {
                super .printStates();
                  System.out.println(" Number of Seats:" +
numOfSeat);
        }
}
```

Llamando al método printStates() de la clase Car se invocará primero printStates() de la clase Vehicle y posteriormente se

mostrara el resultado de println.

Interface

Una interfaz es un contrato entre una clase y el mundo exterior. Cuando una clase implementa una interfaz, debe proporcionar el comportamiento especificado por esa interfaz. Tomando el ejemplo de Vehicle crearemos una interfaz.

```
public interface IVehicle {
        void changeGear(int newValue);
        void speedUp(int increment);
}
```

Ahora la clase Vehicle implementa la interfaz IVehicle utilizando la sintaxis siguiente:

```
class Vehicle implements IVehicle {
        int speed = 0;
        int gear = 1;
        public void changeGear(int newValue) {
                gear = newValue;
        }
        public void speedUp(int increment) {
                speed = speed + increment;
        }
        void printStates() {
                  System.out.println(" speed:" + speed + "
gear:" + gear);
        }
}
```

Se debe tener en cuenta que la clase de Vehicle debe proporcionar y la implementación de métodos los métodos changeGear() y speedUp().

Access Modifiers

Los modificadores de acceso determinan si otras clases pueden utilizar una variable o invocar un método en concreto. Hay cuatro tipos de control de acceso:

- A nivel de clase – public o default (sin modificador explícito).
- A nivel de miembro de clase (variable o método) – public, private, protected o default (sin modificador explícito).

Una clase puede ser declarada como pública con el modificador public, en cuyo caso esa clase es visible para todas las clases en todas partes. Si una clase no tiene modificador (por defecto), es visible sólo dentro de su propio package (un package es una agrupación de clases afines).
A nivel de miembro, además los modificadores public o default (package-private), existen dos modificadores de acceso adicionales: private y protected. El modificador private especifica que el miembro sólo puede ser accedido des de su propia clase. El modificador protected especifica se puede acceder al miembro dentro de su propio package y, además, por cualquier otra subclase de de esta.

Access Levels				
Modifier	**Class**	**Package**	**Subclass**	**AllOther**
public	**Y**	**Y**	**Y**	**Y**
protected	**Y**	**Y**	**Y**	**N**
Default	**Y**	**Y**	**N**	**N**
private	**Y**	**N**	**N**	**N**

Contructors

Los constructores se invocan para crear objetos. Son similares a las funciones, pero diferenciadas por la siguiendo:

- Los constructores tienen el mismo nombre que la clase
- No tienen ningún tipo de retorno.

Llamar a un constructor para crear un nuevo objeto sería inicializar miembros de un objeto. suponer El vehículo tiene el siguiente constructor:

```
public Vehicle(int speed, int gear){
        this.speed = speed;
        this.gear = gear;
}
```

Para crear un nuevo objeto de la clase Vehicle invocando el constructor deberemos utilizar la palabra reservada new:

```
Vehicle vehicle = new Vehicle(4, 2);
```

Esto creará un nuevo objeto de la clase Vehicle con sus dos variables de clase speed y gear inicializadas a 4 y 2.

Method overriding and overloading

Dentro de la misma clase, se pueden crear dos métodos con el mismo nombre pero se diferencia en el número de argumentos y sus tipos. Esto se llama sobrecarga de métodos.
La sobrecarga de métodos ocurre cuando una clase hereda un método de una super clase, pero ofrece su propia aplicación de ese método. En el siguiente código, la clase Car sobrecarga el método SpeedUp() definido en la clase de Vehicle.

```
public class Car extends Vehicle {
        int numOfSeats;
        public void speedUp(int increment) {
                speed = speed + increment + 2;
        }
}
```

Supongamos que creamos un objeto de tipo Car y llamamos al método llamó a la speedUp(). Entonces, el método del Vehicle es ignorado y se ejecuta el de la clase Car:

```
Car car = new Car();
car.speedUp(2);
```

Polymorphism

En el contexto de la programación orientada a objetos, el polimorfismo significa que las diferentes subclases de la misma clase padre puede tener comportamientos diferentes, pero comparten algunas de las funcionalidades de la clase padre.
Para demostrar el polimorfismo, añadiremos el método showInfo () a la clase de Vehicle. Este método imprime toda la información en un objeto del tipo de Vehicle:

```
public void showInfo() {
        System.out.println("The vehicle has a speed of: " +
this.speed
                + " and at gear " + this.gear);
}
```

Sin embargo, si la subclase Truck utiliza este método, la variable miembro loadWeight no será imprimida, ya que no es un miembro de la clase padre Vehicle. Para resolver esto, podemos sobreescrivir el método showInfo() de la siguiente manera:

```
public void showInfo() {
        super.showInfo();
        System.out.println("The truck has is carrying a load
of: "
                + this.loadWeight);
}
```

Podemos observar que el método showInfo() de Truck, llamará showInfo() de la clase padre y agregar a ella su propio comportamiento – el cual imprime el valor de loadWeight.
Podemos hacer lo mismo con la clase Car.

```
public void showInfo() {
        super.showInfo();
        System.out.println("The car has "
                + this.numOfSeats + " seats.");
```

```
}
```

Ahora, podemos hacer un test de polimorfismo. Crearemos 3 objetos, cada uno de un tipo de Vehicle diferente

```
class TestPolymorphism {
        public static void main(String[] args) {
                Vehicle vehicle1, vehicle2, vehicle3;
                vehicle1 = new Vehicle(50,2);
                vehicle2 = new Car(50,2,4);
                vehicle3 = new Truck(40,2,500);
                System.out.println("Vehicle 1 info:");
                vehicle1.showInfo();
                System.out.println("\nVehicle 2 info:");
                vehicle2.showInfo();
                System.out.println("\nVehicle 3 info:");
                vehicle3.showInfo();
        }
}
```

El resultado de ejecutar esta clase será la creación de tres tipos de objeto diferentes:

```
Vehicle 1 info:
The vehicle has a speed of: 50 and at gear 2
Vehicle 2 info:
The vehicle has a speed of: 50 and at gear 2
The car has 4 seats.
Vehicle 3 info:
The vehicle has a speed of: 40 and at gear 2
The truck has is carrying a load of: 500
```

En el ejemplo anterior, la JVM ha llamado el método de cada objeto en lugar de llamar el objeto de Vehicle.

Observaciones

En este segundo post hemos introducido varios conceptos de programación orientada a objetos enfocada al entorno Java. Esta parte es muy importante, se debe tener en cuenta que un correcto diseño nos va asegurar en el futuro un fácil mantenimiento del código. Bien porqué tengamos que añadir funcionalidades o porqué debemos solucionar bugs. Esta parte a

nivel teórico se puede ampliar con los siguientes posts:

- Pincipios de programación orientada a objetos
- Principios del diseño de clases