**METODOLOGÍA DE LA PROGRAMACIÓN**

**Tema 1. Programación Orientada a Objetos. HERENCIA**

1. **Introducción a la POO**

La Programación Orientada a Objetos consiste en identificar objetos en el mundo real y representarlos en los programas mediante jerarquía de clases.

**Conceptos básicos**

Objeto: Ente descrito por un estado (datos) y un comportamiento (operaciones).

Clase: Plantilla o prototipo a partir de la cual se crean los objetos.

Principio de encapsulación:

Es un principio de la POO que establece que a los datos solo se debe acceder a través de los métodos.

Lo ideal es utilizar métodos observadores y modificadores **get()** y **set()** para cada dato para facilitar la modificación del código.

Herencia:

Mecanismo para organizar y estructurar clases en jerarquías, las subclases heredan daos y comportamientos de la superclase.

Tipos:

Simple **(Java)**: Una clase solamente hereda directamente de una superclase.

Múltiple: Una clase puede heredar directamente de varias superclases.

**Ejemplo de clases y subclases**

|  |
| --- |
| **Bicicleta** |
| **Estado:**  - Dos ruedas  - Velocidad  - Dirección  - … |
| **Comportamientos:**  - Acelerar  - Frenar  - Girar derecha  - Girar izquierda  - … |

|  |
| --- |
| **Bicicleta de montaña** |
| **Estado:**  - …  - Tipo de amortiguador  - Ruedas gordas  - … |
| **Comportamientos:**  - … |

|  |
| --- |
| **Bicicleta de carretera** |
| **Estado:**  - …  - Coeficiente aerodinámico  - Ruedas finas  - … |
| **Comportamientos:**  - … |

**Relaciones entre objetos**

Composición (tiene un): Se da cuando un objeto tiene a otro.

Ejemplo: Una bicicleta **tiene** dos ruedas.

Herencia (es un): Se da cuando un objeto es más específico que otro.

Ejemplo: Una bicicleta **es una** bicicleta con dos ruedas gordas.

**Notación UML (Unified Modeling Languaje)**

Se trata de una representación gráfica que permite expresar características y relaciones entre clases.

**Ventajas del uso de la POO**

* Modularidad.
* Ocultación (protección) de la información.
* Reusabilidad del código.
* Facilidad de depuración.
* Escalabilidad.

**Inconvenientes de la POO**

* Aprender requiere un esfuerzo importante.

1. **Herencia en JAVA**

**Clases, variables y objetos en JAVA**

Declaración de clases

class Bicycle {

private int gear;

public void setGear(int newGear) {

gear = newGear;

}

}

Declaración de variables

Bicycle unaBicicleta, otraBicicleta;

Creación de objetos

unaBicicleta = new Bicycle();

Uso de objetos

otraBicicleta = unaBicicleta;

unaBicicleta.setGear(23);

System.out.println(unaBicicleta.toString());

System.out.println(otraBicicleta.toString());

**Subclases y superclases**

Para definir subclases en Java se utiliza la palabra reservada **extends**

Class <Subclase> extends <Superclase>

* La Subclase hereda datos y métodos (salvo constructores) de la Superclase.
* La Subclase puede añadir datos y métodos, o bien redefinir los que hereda de la Superclase.
* En la Subclase se pueden utilizar métodos propios y heredados de la Superclase.

**La clase Object**

Si no se indica ninguna superclase en la declaración es como si se indicase **Object**.

**Object** tiene varios métodos que se pueden redefinir en las subclases

* clone clona un objeto.
* equals comprueba si dos objetos son iguales.
* getClass devuelve la clase del objeto.
* toString proporciona una representación en formato **String** de un objeto.

**Uso de métodos heredados**

En la subclase se pueden utilizar los métodos propios y los públicos (**public**) de la superclase.

**Reescritura de métodos heredados**

En la subclase se pueden reescribir los métodos heredados.

Es buena idea utilizar la anotación @Override para indicar que se está sobrescribiendo un método heredado.

**Acceso a métodos reescritos**

Si un método está reescrito (override) en una subclase, se puede llamar al método de la superclase poniendo el prefijo super.

**Constructores en las subclases**

Los constructores **no** se heredan. Pero se pueden invocar los de la superclase desde la subclase con super(…) para construir la parte del objeto heredada de la superclase. De hecho por defecto se invoca al constructor por defecto super().

**Niveles de acceso a datos y métodos**

* **Private:** Solo son accesibles en la clase en la que están definidos.
* **Protected:** Son accesibles en las subclases y también desde las clases del mismo  **package**.
* **Public:** Son accesibles desde cualquier clase.

En cualquier caso es mejor utilizar los métodos **observadores** y  **modificadores** para acceder a ellos (principio de encapsulación).

**Modificadores**

* **Abstract**

Aplicado a clases (clases abstractas)

No se pueden instanciar, solo sirven para ser heredadas.

class **abstract** nombreClase …

Aplicado a métodos de clases abstractas (métodos abstractos)

Son métodos sin implementación.

Se implementan en las subclases.

public **abstract** nombreMetodo(<args.>);

* **Static**

Aplicado a datos

En este caso se llaman datos estáticos o variables de clase. Solo existe una copia para todas las instancias de la clase.

Son accesibles a través de la clase y de las instancias.

float X = **Math**.PI;

Aplicado a métodos

Es análogo a los datos, son métodos de clase a los que normalmente se accede a través de la clase.

**Math**.sin(34);

* **Final**

Aplicado a datos

No se pueden ocultar (hidding) en las subclases declarándolos otra vez.

Si se aplican a datos **static**, éstos no se pueden modificar.

Aplicado a métodos de instancia

No se pueden reescribir (overriding) para cambiar la implementación en una subclase.

Aplicado a métodos **static**

No se pueden ocultar (hidding) redefiniéndolos en una subclase.

Aplicado a clases

No se pueden declarar subclases, pero sí instancias.

**Tema 2. Programación Orientada a Objetos. POLIMORFISMO**

1. **Introducción al Polimorfismo**

El polimorfismo permite :

* Hacer programas que procesan de una manera uniforme objetos de diferentes clases, siempre que compartan una superclase a cualquier nivel.
* Extender los programas sin más que añadir clases a la jerarquía e instanciar objetos de esas clases.

**Mecanismos del Polimorfismo**

Enlace entre la llamada a un método y el código que se ejecuta

Estático: En tiempo de compilación.

Dinámico: En tiempo de ejecución

El polimorfismo requiere enlace **dinámico**.

Una variable de una clase puede referenciar a un objeto de una subclase, pero no al revés.

Bicycle b1, b2;

MountainBike mb1;

b1 = new Bicycle(20, 10, 1);

b2 = new MountainBike(20, 10, 5, “Dual”);

mb1 = new Bicycle(40, 20, 8); **//No funciona**

Cuando se invoca un método sobre una variable, se ejecuta el código correspondiente a la clase del objeto, que puede ser distinto al de la clase de la variable.

La máquina virtual de Java (JVM) llama al método de la clase del objeto y no al de la variable. Así, con la siguiente llamada se ejecuta el método definido en la clase **MountainBike**.

b2.printStates();

**Resumen de polimorfismo**

Una variable de una clase puede contener un objeto de una subclase a cualquier nivel, pero no al revés.

A una variable se le pueden aplicar los métodos de la clase (propios y heredados). Pero **NO** se le pueden aplicar métodos o acceder a atributos no definidos en la clase.

El código que se ejecuta al invocar un método es el de la clase del objeto, que puede ser distinto del de la clase de la variable.

**Clases y métodos abstractos**

Las clases y métodos abstractos tienen un interés especial cuando se utiliza polimorfismo.

Las clases abstractas no se pueden instanciar, esto significa que no se pueden crear objetos con el operador **new**.

Pero sí se pueden declarar variables de clases abstractas.

- Pueden contener objetos de subclases no abstractas y así se pueden tratar estos objetos polimórficamente.

- Sobre estas variables se puede invocar métodos abstractos definidos en la clase abstracta. Estos métodos deberán estar implementados en la subclase del objeto contenido en la variable.

1. **Ejemplo de clases y métodos abstractos**

**- Definición de la clase**

public **abstract** class Employee {

private String firstName;

private String lastName;

private String socialSecurityNumber;

public void setFirstName( String first ) {

firstName= first;

}

. . . . . . .

// abstract method overr. by concrete subclasses

public **abstract** double earnings();

}

El método **earnings()** se implementa en las subclases.

public class CommissionEmployee extends Employee {

private double grossSales;

private double commissionRate; // override abstract method earnings in Employee @Override

**public double earnings(){**

**return getCommissionRate() \* getGrossSales();**

**}**

}

En esta subclase se implementa **earnings()** y ya no es abstractos

Una clase no abstracta debe tener implementados **TODOS** los métodos abstractos de las superclases abstractas.

public class BasePlusCommisionEmployee extends

CommissionEmployee

{

private double baseSalary; //base salary per week

//override method earnings in CommissionEmployee

@Override

public double earnings()

{

return getBaseSalary() + **super.earnings();**

}

}

Como cualquier método reescrito, **earnings()** puede utilizar la implementación de la superclase, si en ésta el método no es abstracto.

**- Ejemplo de uso**

System.out.printf( "%s\n%s: $%,.2f\n\n",

salariedEmployee "earned",

**salariedEmployee.earnings()**);

System.out.printf( "%s\n%s: $%,.2f\n\n",

hourlyEmployee, "earned",

**hourlyEmployee.earnings()**);

System.out.printf( "%s\n%s: $%,.2f\n\n",

commissionEmployee, "earned",

**commissionEmployee.earnings()**);

System.out.printf( "%s\n%s: $%,.2f\n\n", basePlusCommissionEmployee,"earned", **basePlusCommissionEmployee.earnings()**);

Cada variable tiene un objeto de su misma clase.

A cada una se le aplica el método **earnings()** de su clase.

**NO POLIMORFISMO.**

Employee[] employees = new Employee[ 4 ];

employees[ 0 ] = salariedEmployee;

employees[ 1 ] = hourlyEmployee;

employees[ 2 ] = commissionEmployee;

employees[ 3 ] = basePlusCommissionEmployee;

// POLIMORFISM: generically process each element in

// array employees

for ( **Employee currentEmployee**: employees ){

System.out.printf( "earned $%,.2f\n\n",

**currentEmployee.earnings()**);

}

Cada componente tiene un objeto de una clase distinta

A cada empleado se le aplica el método **earnings()** de su clase, no el de la variable.

**POLIMORFISMO.**

**Compatibilidad entre clases, objetos y métodos (casting)**

A una **variable** sólo se le pueden aplicar los métodos definidos en su clase (propios o heredados, abstractos o no).

Pero se le pueden asignar objetos de una sublcase

CommissionEmployee currentEmployee;

currentEmployee = new BasePlusCommissionEmployee();

Cuando se le aplica un método, se invoca la **implementación** de la clase del objeto (no la de la clase de la variable).

System.out.println(currentEmployee.earnings());

Pero a una variable **no** se le puede aplicar un método que no esté definido en la clase de la variable (aunque esté definido en la clase del objeto).

Sys...println(currentEmployee.getBaseSalary());

**Identificación dinámica de la clase**

**-** Operador **instanceof:**

Es un operador binario que sirve para determinar si un objeto es de una determinada clase (o de una de sus subclases)

Es útil si queremos hacer algo especial con sus elementos de una clase particular de una jerarquía, cuando se usa polimorfismo.

Ejemplo:

if (currentEmployee **instanceof** BasePlusCommissionEmployee)

{ … }

**-** Los métodos **getClass()** y **getName()**

**- getClass()** está definido en la clase **Object** como **public** y **final**, por lo que se puede usar, pero no se puede reescribir, en las subclases.

Devuelve un objeto de la clase **Class**, este objeto guarda información de otro objeto, por ejemplo el nombre\_de\_su\_clase.

**- getName()** es un método definido en la clase **Class** y devuelve el valor del campo nombre\_de\_su\_clase (precedido por su package).

Ejemplo:

for (int j=0; j<employees.length; j++)

System.out.printf(“Employee %d is a %s\n”, j,

**employees[j].getClass().getName()**);

…

if (**obj1.getClass().equals(obj2.getCLass())**)

System.out.println(“Son de la misma clase”);

**- getSImpleName()**

**Class.getSimpleName()** es un método definido en la clase **Class** y devuelve el valor del campo nombre\_de\_su\_clase.

Ejemplo:

…

if (

obj1.getClass().getSimpleName().equals(

“**HourlyEmployee**”))

{

System.out.println(“el tipo de obj1 es hourlyemployee”);

}

**Downcasting**

Si se quiere aplicar a una variable un método declarado en la clase del objeto y no en la clase de la variable hay que utilizar downcasting.

El downcasting consiste en rebajar la categoría de una variable de una clase a una subclase. Para ello se aplica un operador que consiste en poner el nombre de la subclase entre paréntesis.

(BasePlusCommissionEmployee)currentEmployee

1. **Interfaces en Java**

Una interface es una declaración de prototipos de métodos y constantes.

- No incluye la implementación de los métodos

- Los métodos se implementan en las clases que implementan la interface

Ejemplo:

La interface **Driveable** podrá ser implementada por cualquier clase de objetos que se deban conducir, por ejemplo bicicletas o coches.

publicinterface Driveable{

voidturnLeft(intamount);

voidturnRight(intamount);

voidspeedUp(intincrement);

voidslowDown(intdecrement);

}

Las clases pueden implementar interfaces, esta relación se expresa con la palabra **implements**.

class Bicycle **implements** Driveable

La relación **implements** tiene alguna similitud con **extends**, pero no son lo mismo

- Los objetos de la clase también son objetos de la interface. Luego se pueden definir variables de la interface y asignarles objetos de una clase que la implementa. A estas variables se les puede aplicar casting.

- Varias clases pueden implementar la misma interface, de este modo todas tienen una interface común.

- Las clases no abstractas deben implementar todos los métodos de la interface.

- Una clase puede implementar varias interfaces. Así la relación **implements**  es de alguna manera el sustituto de la herencia múltiple en Java.

**¿Para qué sirven las interfaces?**

Las interfaces se suelen utilizar para expresar comportamientos comunes a clases que no están relacionadas de otra forma.

Ejemplo:

- Las clases **Coches**, **Motos**, etc. o incluso la clase **Caballo** pueden

implementar la interface **Driveable** y de ese modo pueden ser utilizadas de la misma forma en los programas.