1. Herencia

Date	Empty
Status	Not started
Type	Empty
Units	6. Herencia y polimorfismo

Concepto de herencia

En todo lenguaje de programación, la utilización de código que ya está escrito es una de las tareas básica, ya que así no se duplican elementos y es más fácil de codificar. Para esto existe lo que se conoce como herencia. En el tema anterior ya se introdujo el uso de la misma pero cabe recordad los siguientes aspectos básicos a la hora de utilizar herencia:

- En java una clase solo puede heredar de una clase. En concreto de la clase Object
- Cuando una clase hereda de otra, todos los métodos y atributos de la clase superior (llamada superclase) pasan a formar parte de la clase
- Solo existe una excepción en del punto anterior, que es cuando el modificador de acceso de las propiedades y/o métodos es restrictivo (private - para nadie)
- Si un método de la superase necesita ser redefinido en la clase que hereda el método, este queda sobrescrito para todas las clases que vengan debajo
- Si una superclase ha escrito un constructor (y por lo tanto ha eliminado el constructor por defecto) la subclase está obligada a utilizar como mínimo ese constructor (añadiendo más parámetros si los necesita)
- PAra poder utilizar la funcionalidad de la superclase se utiliza la para super
    - En un contructor debe ir en la primera linea. Ej. super(variables)
    - En un método puede ir en cualquier parte, seguido del nombre

del nombre del método. Ej. super.nombreMétodo(variables)

- En java una clase solo puede tener una super clase
- Para poder utilizar la herencia se usa la palabra extendí seguido del nombre de la clase de la que se quiere heredar

Usos

Sea la clase Coche

```
public class Coche { protected String marca, modelo; protecte
d int bastidor; public Coche(String marca, String modelo, int
bastidor) { this.marca = marca; this.modelo = modelo; this.ba
stidor = bastidor; } public String getMarca() { return marca;
} public void setMarca(String marca) { this.marca = marca; }
public String getModelo() { return modelo; } public void setM
odelo(String modelo) { this.modelo = modelo; } public int get
Bastidor() { return bastidor; } public void setBastidor(int b
astidor) { this.bastidor = bastidor; } }
```

En este caso la clase coche representa el molde que se quiere que tenga todas las clases que vendrán por debajo, por lo que se escriben todas las cosas comunes que tendrán todos los coches. Esto servirá como punto de unión de todas las clases que hereden de coche. Una vez se tiene la base, si se quiere crear una clase que especialice a Coche se utiliza la herencia

```
public class CocheDeportivo extends Coche { public CocheDepor
tivo(String marca, String modelo, int bastidor) { super(marc
a, modelo, bastidor); } }
```

En este caso CocheDeportivo tendrá todas las características que tienen Coche y todas las que se marquen en la clase. Hay que tener en cuenta que en este ejemplo las variables y métodos de la clase Coche no son protected, por lo que se pasarán a las clases que van por debajo

En el caso de interesarnos que la clase más superior (superclase) no pueda ser instancia (no se puedan crear objetos de dicha clase), se podrá marcar como abstracta con la palabra abstract.

```
public abstract class Coche { String marca, modelo; int basti
dor; public Coche(String marca, String modelo, int bastidor)
{ this.marca = marca; this.modelo = modelo; this.bastidor = b
astidor; } public String getMarca() { return marca; } public
void setMarca(String marca) { this.marca = marca; } public St
ring getModelo() { return modelo; } public void setModelo(Str
ing modelo) { this.modelo = modelo; } public int getBastidor
() { return bastidor; } public void setBastidor(int bastidor)
{ this.bastidor = bastidor; } }
```

De esta forma la clase Coche no podrá ser utilizada para crear objetos

```
// no podrá ser utilizado Coche coche = new Coche(); // solo
podrá ser creado cuando se crea al menos uno de los métodos C
oche cocheCreado = new Coche("marca","modelo",123) { @Overrid
e public String getMarca() { return super.getMarca(); } };
```

En el lado contrario están las clases que representan los objetos que realmente se quieren instanciar. Esas clases se puede marcar como final, de forma que nadie podrá extender de ella. Para ello se utiliza la palabra final

```
public final class CocheDeportivo extends Coche { public Coch
eDeportivo(String marca, String modelo, int bastidor) { super
(marca, modelo, bastidor); } }
```

Herencia

En este documento se explica el concepto de herencia en Java y cómo se utiliza en la programación. Se presenta un ejemplo de una clase Coche y cómo se utiliza la herencia para crear una clase CocheDeportivo que hereda las características de Coche y agrega algunas específicas para los coches deportivos. Además, se explica cómo utilizar la abstracción y la sobrescritura de métodos para personalizar comportamientos específicos de las clases hijas. Finalmente, se presenta un ejemplo de cómo se pueden utilizar objetos de diferentes subclases en una colección de su clase padre común.En Java, la herencia es un mecanismo que permite definir una nueva clase basada en una clase ya existente, tomando sus atributos y métodos y

redefiniéndolos o agregando nuevos. Esto permite reutilizar código y simplificar la programación, ya que no es necesario volver a escribir código que ya existe en otra clase.

Un ejemplo de uso de la herencia en Java puede ser la creación de una clase Coche, que tenga atributos como marca, modelo y bastidor, y métodos como getMarca, setModelo, etc. A partir de esta clase se puede crear una subclase llamada CocheDeportivo, que herede todas las características de Coche y además tenga atributos y métodos específicos para los coches deportivos, como potencia y velocidad máxima. Para definir que una clase hereda de otra en Java, se utiliza la palabra clave "extends", seguida del nombre de la clase que se quiere heredar.

A continuación se presenta un ejemplo de la clase Coche y la subclase CocheDeportivo:

```
public class Coche { protected String marca, modelo; protecte
d int bastidor; public Coche(String marca, String modelo, int
bastidor) { this.marca = marca; this.modelo = modelo; this.ba
stidor = bastidor; } public String getMarca() { return marca;
} public void setMarca(String marca) { this.marca = marca; }
public String getModelo() { return modelo; } public void setM
odelo(String modelo) { this.modelo = modelo; } public int get
Bastidor() { return bastidor; } public void setBastidor(int b
astidor) { this.bastidor = bastidor; } } public class CocheDe
portivo extends Coche { protected int potencia; protected dou
ble velocidadMaxima; public CocheDeportivo(String marca, Stri
ng modelo, int bastidor, int potencia, double velocidadMaxim
a) { super(marca, modelo, bastidor); this.potencia = potenci
a; this.velocidadMaxima = velocidadMaxima; } public int getPo
tencia() { return potencia; } public void setPotencia(int pot
encia) { this.potencia = potencia; } public double getVelocid
adMaxima() { return velocidadMaxima; } public void setVelocid
adMaxima(double velocidadMaxima) { this.velocidadMaxima = vel
ocidadMaxima; } }
```

En este ejemplo, la clase Coche tiene atributos marca, modelo y bastidor, y métodos getMarca, setModelo, etc. La subclase CocheDeportivo hereda todos estos atributos y métodos, y además tiene atributos potencia y velocidadMaxima, y métodos getPotencia y setVelocidadMaxima.

Otro concepto importante en la herencia es la sobrescritura de métodos. Esto permite que una subclase modifique o redefina un

métodos. Esto permite que una subclase modifique o redefina un método que ha sido heredado de la superclase. Para sobrescribir un método en Java, se utiliza la anotación "@Override" antes de la definición del método en la subclase.

A continuación se presenta un ejemplo de sobrescritura de métodos en la clase CocheDeportivo:

```
public class CocheDeportivo extends Coche { protected int pot
encia; protected double velocidadMaxima; public CocheDeportiv
o(String marca, String modelo, int bastidor, int potencia, do
uble velocidadMaxima) { super(marca, modelo, bastidor); this.
potencia = potencia; this.velocidadMaxima = velocidadMaxima;
} @Override public String toString() { return "CocheDeportivo
[marca=" + marca + ", modelo=" + modelo + ", potencia=" + pot
encia + ", velocidadMaxima=" + velocidadMaxima + "]"; } }
```

En este ejemplo, se sobrescribe el método toString de la superclase Coche para que muestre también los atributos potencia y velocidadMaxima de la subclase.

La abstracción es otro concepto importante en la herencia. Una clase abstracta es una clase que no se puede instanciar directamente, sino que se utiliza como base para crear otras clases. Una clase abstracta puede tener métodos sin definir, que deben ser implementados en las subclases. En Java, se utiliza la palabra clave "abstract" para definir una clase abstracta.

A continuación se presenta un ejemplo de una clase Coche abstracta:

```
public abstract class Coche { protected String marca, modelo;
protected int bastidor; public Coche(String marca, String mod
elo, int bastidor) { this.marca = marca; this.modelo = model
o; this.bastidor = bastidor; } public String getMarca() { ret
urn marca; } public void setMarca(String marca) { this.marca
= marca; } public String getModelo() { return modelo; } publi
c void setModelo(String modelo) { this.modelo = modelo; } pub
lic int getBastidor() { return bastidor; } public void setBas
tidor(int bastidor) { this.bastidor = bastidor; } public abst
ract void acelerar(int v); }
```

En este ejemplo, la clase Coche se ha definido como abstracta y se ha añadido un método acelerar sin definir. Todas las subclases de Coche

deberán implementar este método para poder ser instanciadas.

Sobrecarga y sobreescritura

Sobrecarga y sobrescritura

La sobrecarga y la sobrescritura son dos conceptos importantes en la programación orientada a objetos en Java. La sobrecarga se refiere a la definición de varios métodos con el mismo nombre en una misma clase, pero con diferentes parámetros. Esto permite que se puedan utilizar diferentes variantes del mismo método según las necesidades del programa. La sobrecarga se utiliza para dar más opciones al programador en cuanto a los parámetros que puede utilizar para llamar a un método.

La sobrescritura, por otro lado, se refiere a la definición de un método en una subclase que tiene el mismo nombre y los mismos parámetros que un método definido en la superclase. La sobrescritura se utiliza para personalizar el comportamiento de un método en una subclase específica. Cuando se llama a un método en una subclase que ha sido sobrescrito, se ejecutará el método de la subclase en lugar del método de la superclase.

En resumen, la sobrecarga se utiliza para tener varios métodos con el mismo nombre pero diferentes parámetros en una misma clase, mientras que la sobrescritura se utiliza para personalizar el comportamiento de un método en una subclase específica.

Ejemplo de sobrecarga y sobrescritura

```
public class EjemploSobrecargaSobrescritura { public static v
oid main(String[] args) { Coche coche = new Coche("Seat", "Ib
iza", 123); coche.acelerar(); coche.acelerar(50); CocheDeport
ivo cocheDeportivo = new CocheDeportivo("Ferrari", "Testaross
a", 456, 500, 300.0); cocheDeportivo.acelerar(); cocheDeporti
vo.acelerar(200); } } public class Coche { protected String m
arca, modelo; protected int bastidor, velocidad; public Coche
(String marca, String modelo, int bastidor) { this.marca = ma
rca; this.modelo = modelo; this.bastidor = bastidor; } public
void acelerar() { System.out.println("Acelerando..."); veloci
dad += 10; } public void acelerar(int v) { System.out.println
("Acelerando a " + v + " km/h..."); velocidad = v; } } public
class CocheDeportivo extends Coche { protected int potencia;
protected double velocidadMaxima; public CocheDeportivo(Strin
```

```
g marca, String modelo, int bastidor, int potencia, double ve
locidadMaxima) { super(marca, modelo, bastidor); this.potenci
a = potencia; this.velocidadMaxima = velocidadMaxima; } @Over
ride public void acelerar() { System.out.println("Acelerando
a fondo..."); velocidad += 50; } }
```

En este ejemplo, se muestra cómo utilizar la sobrecarga y la sobrescritura en Java. Se define una clase Coche con un método acelerar que aumenta la velocidad del coche en 10 km/h, y otro método acelerar con un parámetro que establece la velocidad a la que se quiere acelerar.

Se define también una subclase CocheDeportivo que hereda de Coche y tiene su propio método acelerar que aumenta la velocidad del coche en 50 km/h. Se utiliza la sobrescritura para personalizar el comportamiento del método acelerar en la subclase CocheDeportivo.

Abstracción

En el ejemplo anterior, la clase coche (marcada como abstracta) puede tener tantos métodos como accesibles como se quiera. Estos métodos podrán ser accedidos y utilizador por la clases que extiendan de ella (CocheDeportivo)

```
public abstract class Coche { Protected String marca, modelo;
Protected int bastidor, velocidad; public Coche(String marca,
String modelo, int bastidor) { this.marca = marca; this.model
o = modelo; this.bastidor = bastidor; } public String getMarc
a() { return marca; } public void setMarca(String marca) { th
is.marca = marca; } public String getModelo() { return model
o; } public void setModelo(String modelo) { this.modelo = mod
elo; } public int getBastidor() { return bastidor; } public v
oid setBastidor(int bastidor) { this.bastidor = bastidor; } p
ublic int getVelocidad() { return velocidad; } public void se
tVelocidad(int velocidad) { this.velocidad = velocidad; } }
```

Adicionalmente, las clases abstractas podrán tener tantos métodos abstractos como sean necesarios. Estos métodos se caracterizan por tener tan solo firma (método de acceso, retorno, nombre y parámetros) y no tener el cuerpo definido. Esto hace que cualquier clase que extienda de ella se vea obligada a escribir el cuerpo de todos los métodos abstractos.

```
public abstract class Coche { protected String marca, modelo;
protected int bastidor, velocidad; public Coche(String marca,
String modelo, int bastidor) { this.marca = marca; this.model
o = modelo; this.bastidor = bastidor; } public abstract void
acelerar(int vAcelerar); public String getMarca() { return ma
rca; } public void setMarca(String marca) { this.marca = marc
a; } public String getModelo() { return modelo; } public void
setModelo(String modelo) { this.modelo = modelo; } public int
getBastidor() { return bastidor; } public void setBastidor(in
t bastidor) { this.bastidor = bastidor; } public int getVeloc
idad() { return velocidad; } public void setVelocidad(int vel
ocidad) { this.velocidad = velocidad; } }
```

El método abstracto acelerar no se puede definir, ya que dependerá del uso que le den los objetos que extiendan de Coche. En ese caso, cada clase que utilice la herencia se verá obligada a sobreescribir el método definiendo su cuerpo

```
public final class CocheDeportivo extends Coche { public Coch
eDeportivo(String marca, String modelo, int bastidor) { super
(marca, modelo, bastidor); this.velocidad = 0; } @Override pu
blic void acelerar(int vAcelerar) { this.velocidad += vAceler
ar*0.5; } }
```

```
public final class CocheUtilitario extends Coche { public Coc
heUtilitario(String marca, String modelo, int bastidor) { sup
er(marca, modelo, bastidor); this.velocidad = 0; } @Override
public void acelerar(int vAcelerar) { this.velocidad = vAcele
rar; } }
```

De esta forma, ambas clases finales han extendido de la misma super clase y tienen los mismos métodos (aquellos que son abstractos) pero alguno de ellos con comportamiento diferente

```
public class Entrada { public static void main(String[] args)
{ CocheDeportivo deportivo = new CocheDeportivo("Ford","Focu
s",1234); CocheDeportivo utilitario = new CocheDeportivo("For
d","Mondel",2345); deportivo.acelerar(100); utilitario.aceler
ar(200); } }
```

```
d.(200); } }
```

Adicionalmente, como ambos objetos pertenecen al mismo tipo genérico (Coche), se podrían juntar en una colección de tipo Coche

```
public class Entrada { public static void main(String[] args)
{ ArrayList<Coche> coches = new ArrayList(); CocheDeportivo d
eportivo = new CocheDeportivo("Ford","Focus",1234); CocheUtil
itario utilitario = new CocheUtilitario("Ford","Mondeo",234
5); coches.add(deportivo); coches.add(utilitario); for (Coche
c: coches) { c.acelerar(100); } /* deportivo.acelerar(100); u
tilitario.acelerar(200);*/ } }
```

En este ejemplo, como ambos objetos tienen la misma superclase pueden pertenecer a la colección, y esta al ser recorrida itera objetos de tipo Coche, pudiendo utilizar el método acelerar (abstracto en su definición inicial) ya que ha sido definido en las clases de CocheDeportico y CocheUtilitario