Java

Herencia

La herencia permite crear una clase conocida como *subclase o clase hija a* partir de otra clase conocida como *superclase o clase padre*. Por consiguiente, una subclase es una versión especializada (una extensión) de una superclase. Se heredan todas las variables de instancia y los métodos definidos en la superclase y se añaden elementos propios.

Sintaxis

```
class nombresubclase extends nombreSuperClase {

Declaraciones de variables y métodos
}
```

Puntos distintivos de la relación:

- ✓ *A la relación de Herencia también se la denomina "ES_UN". En el ejemplo de la página 2, un Rectángulo* ***es_una*** *Figura.*
- ✓ *La relación de herencia, si se mira del lado de la superclase, es una generalización de las subclases.*
- ✓ *Si se mira del lado de la superclase, es una especialización en cada subclase.*

Solo se puede especificar una superclase para cualquier subclase que se crea. Este tipo de herencia se denomina *herencia simple*. Java no soporta la herencia de múltiples superclases. Es decir no soporta *herencia múltiple. Solo puede existir una superclase después de la palabra clave* `extends`.

Clases abstractas

Las clases abstractas se utilizan para definir la estructura general de una familia de clases derivadas. Se utilizan, en general, para imponer una funcionalidad común a un número de clases deferentes que se conocen como clases concretas. Una *clase concreta* es una clase que no es abstracta. No se puede crear objetos de una clase abstracta, sólo se puede crear de una clase concreta.
Las clases abstractas no se pueden instanciar, pero se pueden hacer referencias de un tipo abstracto que se pueden utilizar para apuntar a una instancia de un objeto de una subclase concreta.
Las clases abstractas pueden definir métodos abstractos y concretos. Cualquier subclase concreta de una clase abstracta **debe** implementar los métodos abstractos de la clase abstracta. Las clases abstractas se designan utilizando las palabras clave `abstract` delante de la palabra clave `class` en la sentencia de declaración de la clase. Los métodos *abstractos* se declaran con el siguiente formato.
`abstract` *tipo nombre (lista de parámetros)*
Ejemplo

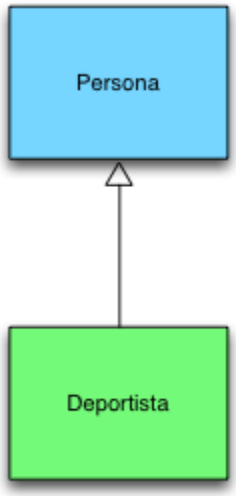
La clase abstracta `Figura` define el método `calculararea()`. Dos subclases de `Figura, Rectángulo` y `Triangulo` proporcionan dos implementaciones deferentes de `calculaearea()`

```
abstract class Figura{
double dimi dmj;
figura(double a,double b){
dimi=a;
dmj=b;
}
abstact double claculArArea(); // método abstracto
}
class Rectángulo extends Figura{
Rectángulo (double a; double b) {
  super (a,b);
}
double calcularArea(){
return dimi*dmj;
}
}
class triangulo extends Figura{
Triangulo (double a, double b){
  super(a,b);
{
double calcularArea(){
return 0.5*dimi*dmj;
{
{
```

En UML, una clase abstracta se identifica colocando el nombre en letra cursiva o agregando junto al nombre: {abs} o {abstract}.

¿Quién es super()? Constructores y super()

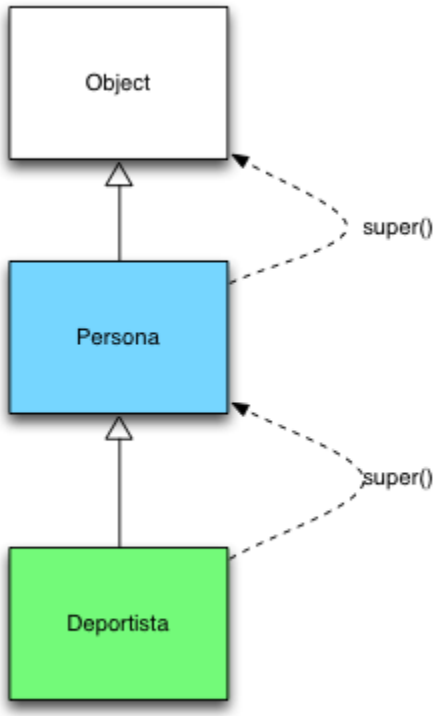
Supongamos que tenemos la siguiente jerarquía:




```
public class Persona {

private String nombre;

public String getNombre() {
return nombre;
}

public void setNombre(String nombre) {
this.nombre = nombre;
}
}

public class Deportista extends Persona{
```

```
}
```

Aunque en el código no aparezcan constructores existen **dos constructores por defecto uno** en cada clase.
Todos los constructores llaman por defecto al constructor de la clase superior a través de una llamada a super() (en este caso al constructor por defecto). **Esto es debido a que los constructores no se heredan entre jerarquías de clases.** Por lo tanto la palabra super() siempre es la primera línea de un constructor e invoca al constructor de la clase superior que comparta el mismo tipo de parametrización.




```
public Persona(String nombre) {
    super();   //invoca al constructor por defecto de Object
  this.nombre = nombre;
}
```

```
public Deportista(String nombre) {
   super(nombre);  //invoca al constructor sobrecargado de
Persona

}
```

Si no lo agregamos al constructor, el compilador añadirá super() por defecto.

Redefinición de métodos heredados

Una clase puede redefinir (volver a definir) cualquiera de los métodos heredados de su super-clase que no sean final. El nuevo método sustituye al heredado para todos los efectos en la clase que lo ha redefinido.

Los métodos de la super-clase que han sido redefinidos pueden ser todavía accedidos por medio de la palabra super desde los métodos de la clase derivada, aunque con este sistema sólo se puede subir un nivel en la jerarquía de clases.
Los métodos redefinidos pueden ampliar los derechos de acceso de la super-clase (por ejemplo ser public, en vez de protected), pero nunca restringirlos.
Los métodos de clase o static no pueden ser redefinidos en las clases derivadas.
Por ejemplo:
SupercClase

```
2
    public class FiguraGeometrica {
4       public String color;
        public void mostrarAtributos() {
6           System.out.println("El color de la figura es: "+color);
7       }
8   }
9
```

SubClase

```
4   public class Circulo extends FiguraGeometrica{
5
6       public float radio;
7
8       @Override
        public void mostrarAtributos() {
10          System.out.println("El radio del círculo es: "+radio);
11              }
12      public void atributosHeredadosYPropios(){
13          super.mostrarAtributos(); //activa el método definido en la superclase
14          this.mostrarAtributos(); //activa la redefinición del método
15      }
16  }
```

Calificador final

Este calificador puede aplicarse a clases, métodos y atributos.
Si se aplica a una clase, significa que será la última de su estructura, es decir, que no puede heredar otra clase de ella. Es la última.
Si se aplica a un método, significa que no puede ser redefinido en una subclase.
Si se aplica a un atributo, éste tiene valor constante, es decir, que no puede cambiar.