# Relaciones de composición y Herencia



## POO en Java

Características:

- Composición “tiene un”
- Herencia “es un”
- Polimorfismo.
- Abstracción.
- Clases y objetos.



## Composición

```
public class Punto {
     int x;
     int y;
   public Punto(int x, int y) {
         this.x = x;
         this.y = y;
   }
   public Punto() {
     // llamada al otro constructor
         this(-1, -1); }
    public void mostrar(){
         System.out.println(“x =  “ + x +” y =  “ + y);  }
    public static void main(String  args[]) {
       Punto pun1 = new Punto();
       Punto pun2 = new Punto(2,3);
       System.out.println( "visualizar datos del punto" );
       pun1.mostrar();
       pun2.mostrar();}}
```

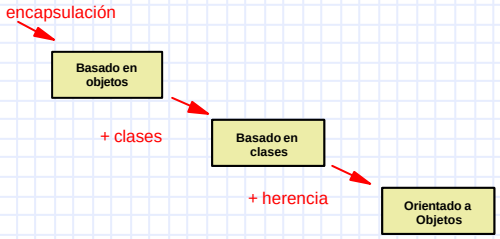


## Composición: clase Círculo

```
public class Circulo {
     Punto origen;
     int radio;

     public Circulo(int x, int y, int radio) {
       origen = new Punto(x,y);
       this.radio = radio;
     }

     public void mostrar() {
       origen.mostrar();
       System.out.println("radio = " + radio);
     }

    public static void main(String args[]) {
         Circulo cir = new Circulo(5, 5, 9);
         System.out.println( "visualizar datos del circulo" );
         cir.mostrar();   }}
```



## Tipos de programación






## Herencia: La palabra reservada extends

Cuando se crea un modelo de algo y luego se quiere una versión más actualizada:

```
public class Empleado
{
  String nombre;
  Date anionac;
  String puesto;
  int categoria;

  ...
}
```

```
public class jefe
{
  String nombre;
  Date anionac;
  String puesto;
  int categoria;
  String departamento;
  Empleado [] subordinados;

  ...
}
```

¡datos duplicados!

## Herencia: La palabra reservada extends

Se puede definir una clase a partir de otra definida previamente.

```
public class Empleado {
 String nombre;
 Date anionac;
 String puesto;
 int categoría; }
public class Jefe extends Empleado {
 String departamento;
 Empleado [ ] subordinados; }
```



## Herencia: La palabra reservada extends

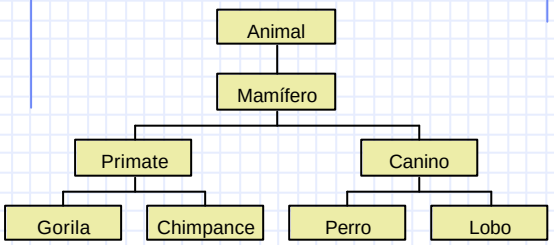
La clase Jefe para tener todas las variables y métodos de la clase Empleado no tiene que definirlos, los hereda de la clase padre.

Todo lo que se tiene que definir después son las características adicionales y especificar los cambios que se quieren aplicar.

Es una buena manera de generar código fácil de mantener y de actualizar. Las modificaciones sobre la clase Empleado repercuten sobre la clase Jefe únicamente compilando.



## Ejemplo de Herencia






## Herencia simple

Cuando una clase hereda sólo de otra clase, se llama herencia simple.

Herencia simple hace que el código sea reutilizable.

Java proporciona las interfaces que proporcionan las ventajas de la herencia múltiple y no presentan sus inconvenientes.



## Los constructores no se heredan

Una subclase hereda de una superclase (la clase padre), todos los métodos y las variables.

Una clase no hereda los constructores de la superclase.

Sólo hay dos formas de que una clase tenga constructor:

Utilizar el constructor por defecto.

Escribir uno o más constructores explícitos.



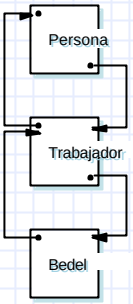
## Herencia: clase Persona

```
public class Persona {
       String nombre = "";
       int edad;

       public Persona(String nom, int ed) {
              nombre = nom;
              edad = ed;
       }

       public void mostrar() {
              System.out.println("Nombre: "+ nombre);
              System.out.println("Edad: "+ edad);
       }
       public static void main(String args[]) {
              Persona yo= new Persona("Luis", 32);
              yo.mostrar();
       }
}
```

## Herencia: clase Trabajador

```
public class Trabajador extends Persona {
 float sueldoHora;
 int numHoras;

 public Trabajador(String nom, int ed, float suel, int num) {
  super(nom, ed); // llamada al constructor de Persona
  sueldoHora = suel;
  numHoras = num;}

 public double sueldo() {
 return sueldoHora * numHoras; }

 public static void main(String args[]) {
  Trabajador yo= new Trabajador("Luis", 32, 200.5f, 45);
  yo.mostrar(); // se invoca al metodo heredado mostrar
  double pelas = yo.sueldo();
  System.out.println("Cobra: " + euros);   }
}
```

## Herencia: constructores






## Reescritura de un método

- La signatura de un método viene dada por su nombre y el tipo y número de los parámetros
  - El tipo devuelto no forma parte de la signatura
- Cuando se reescribe un método en una clase derivada
  - La signatura debe ser la misma que el método de la clase base
  - El tipo devuelto debe ser el mismo que el del método de la clase base
  - El atributo de acceso del método de la clase derivada tiene que ser igual o mas general que el de la clase base (NO puede ser mas restrictivo)



## Ejemplo

```
  //En Persona redefinimos el método mostrar()
public class Trabajador extends Persona {
      float sueldoHora;
      int numHoras;
      public Trabajador(String nom, int ed, float suel, int num) {
            super(nom, ed); // llamada al constructor de Persona
            sueldoHora = suel;
            numHoras = num;}
      // sobrecarga completa de mostrar
      // nombre y edad son atributos heredados de Persona
      public void mostrar() {
            System.out.println("Nombre: "+ nombre);
            System.out.println("Edad: "+ edad);
            System.out.println("Sueldo por hora: "+ sueldoHora);
            System.out.println("Horas trabajadas: "+ numHoras);}}
```



## Sobrecarga de método

- Sobrecarga parcial ampliando el método heredado
  - El método de la subclase puede llamar al método de la superclase



## Sobrecarga de método

```
public class Trabajador extends Persona {
      float sueldoHora;
      int numHoras;

      public Trabajador(String nom, int ed, float suel, int
num) {
      super(nom, ed); // llamada al constructor de Persona
      sueldoHora = suel;
      numHoras = num;}
      // sobrecarga parcial de mostrar
      // nombre y edad son atributos heredados de Persona
      public void mostrar() {
      super.mostrar(); //llamada al método de la clase padre
      System.out.println("Sueldo por hora: "+ sueldoHora);
      System.out.println("Horas trabajadas: "+ numHoras);}
}
```

## La clase genérica Object

- Todas las clases en Java heredan implícitamente de la clase *Object*. De esta forma, *Object* es la raiz de la jerarquía de herencia (de implementación) en Java.

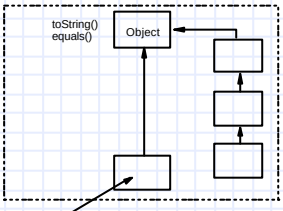


## La clase genérica Object

- *Object* define un conjunto de métodos útiles, que pueden ser redefinidos en cada clase. En particular:
  - *public boolean equals(Object o):* Permite definir el criterio de igualdad utilizado para los objetos de una determinada clase (el operador == únicamente chequea la igualdad de referencias). En *Object*, *equals* se define directamente como la identidad de referencias.
  - *public String toString():* Permite decidir la representación externa de un objeto como una cadena. Por defecto es el valor de su referencia, etiquetada con el nombre de la clase.



## La clase genérica Object




no se necesita extends para heredar de Object



## API de Object

**Method Summary**

| | |
|---|---|
| protected Object | clone( )<br>Creates and returns a copy of this object. |
| boolean | equals(Object obj)<br>Indicates whether some other object is "equal to" this one. |
| protected void | finalize( )<br>Called by the garbage collector on an object when garbage collection determines that there are no more references to the object. |
| Class | getClass( )<br>Returns the runtime class of an object. |
| int | hashCode( )<br>Returns a hash code value for the object. |
| String | toString( )<br>Returns a string representation of the object. |



## Ejemplo

```
public class Alumno{
      String nombre;
      String apellidos;
      public Alumno(String nombre, String apellidos){
            this.nombre= nombre;
            this.apellidos= apellidos;
      }
      public boolean equals(Alumno al){
            if (apellidos.equals(al.apellidos))
                  return true;
            else  return false;
      }
```



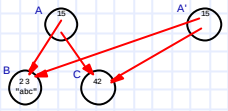
## Ejemplo

```
public static void main(String args[]) {
    Alumno alum = new Alumno("Julian", "Fdez");

    Class tipoObjeto = alum.getClass();
    System.out.println("nombre de la clase: " +
                tipoObjeto.getName());
    Alumno alum2 = alum;
    alum2.apellidos = "Rodriguez";
    System.out.println("son el mismo objeto?:"+
                (alum==alum2));
    alum2 = new Alumno("Antonio", "Rodriguez");
    System.out.println("son iguales los
          objetos?:"+ alum.equals(alum2));
    }}
```

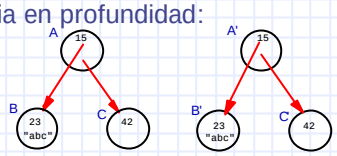
## Copias de Objetos

La asignación en Java simplemente copia la referencia

Copia superficial:




Copia en profundidad:






## clone( )

- Crea un objeto que es copia "superficial" del objeto actual independientemente de su tipo.
- El tipo objeto a copiar debe haber declarado que es susceptible de copia implementando el interfaz vacío *Clonable*
- Es un método protegido de *Object*, por tanto para que pueda ser invocado desde otro paquete hay que redeclararlo como público y simplemente invocar a la clase base.



## clone( )

```
public class ClaseClonable
implements Cloneable {
     // Otro código de la clase
        public Object clone ()
throws CloneNotSupportedException
        {
            return super.clone();
        }
    }
```



## clone( )

```
public abstract class Transaction implements
  Cloneable{
   private long amount;
   private Date date;
public Object clone(){
     Object o = null;
     try{
      o = super.clone();  //copia superficial
     }
     catch(CloneNotSupportedException e){
       throw new InternalError(); }
     return o;
}
```



## clone( )

- Si los miembros de una clase son otros objetos que necesiten "copia profunda" habrá que añadir el código necesario
  - Llamada a los métodos clone() de cada uno de los objetos implicados para crear una copia



## clone( )

```
public abstract class Transaction
  implements Cloneable{
   private long amount;
   private Date date;
   public Object clone(){
     Transaction t = null;
     try{
   t = (Transaction)super.clone();//superf
   t.date = (Date)t.date.clone();//profund
   }
     catch(CloneNotSupportedException e){
       throw new InternalError(); }
     return t;
}
```

## clone( )

- Reglas generales
  - Implementar el interfaz *Cloneable* si se desea que la clase se pueda copiar
  - No utilizar constructores en la implementación de *clone()* si no llamadas a los métodos *clone()* de los otros objetos a copiar
    - Puede existir problemas en el tipo de objeto creado
  - Todas las subclases de una clase *Clonable* también son susceptibles de ser clonadas.
    - Deben reescribir el método *clone()*



## Ejemplo

```
public class Alumno implements Cloneable {
 String nombre;
 String apellidos;
 public Alumno(String nombre, String apellidos){
      this.nombre= nombre;
      this.apellidos= apellidos;
      }
 public boolean equals(Alumno al){
      if (apellidos.equals(al.apellidos))return true;
            else  return false;
      }
      public Object clone () throws
CloneNotSupportedException{
            return super.clone();
      }
      public String toString(){
            return nombre + " " + apellidos + " ";    }
```

## Ejemplo (cont.)

```
public static void main(String args[]) {
 Alumno alum = new Alumno("Julian", "Fdez");
 Class tipoObjeto = alum.getClass();
 System.out.println("nombre de la clase:" +tipoObjeto.getName());
 Alumno alum2 = alum;
 alum2.apellidos = "Rodriguez";
 System.out.println("son el mismo objeto?:"+ (alum==alum2));
 alum2 = new Alumno("Antonio", "Rodriguez");
 System.out.println("son iguales os objetos?:"+
alum.equals(alum2));
 try {  alum2 = (Alumno)alum.clone();
 }catch (CloneNotSupportedException e){
       System.out.println("no tiene implementada la clonacion");
 }
 alum.nombre="Fran";
 alum2.nombre="Moises";
 System.out.println("objetos: "+ alum + alum2);
 }
}
```

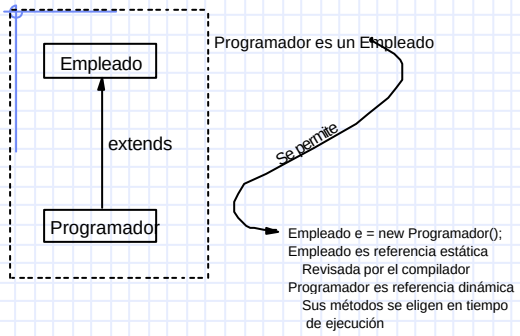
## Referencia estática

La referencia estática se fija en tiempo de compilación

```
Rectangle r = new Rectangle();
Circle c = new Circle();
...
c = r; //Error
```



## Referencia dinámica

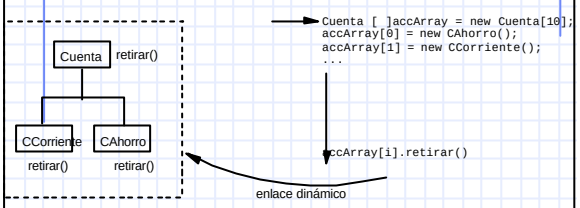





## Enlace dinámico

- El método a ejecutarse se determina en ejecución en función de la referencia dinámica (no de la estática)

## Enlace dinámico






## Enlace dinámico

```
class Shape{ ... double getArea(){...};}
class Rectangle extends Shape{ ... double
getArea(){...};}
class Triangle extends Shape{ ... double
getArea(){...};}

Shape s;  Rectangle r;  Triangle t;
// ...
if (/*...*/) s = r; else s = t;
double answer = s.getArea();  // ¿Cual ejecuta?

r = s; // ¿legal?
```



## Enlace dinámico


```
public static void main(String args[]) {
 Persona per;
 Trabajador yo= new Trabajador("Balta", 99, 200.5f, 45);

 per = yo;     // Asignacion legal
 // se ejecuta el método mostrar de Trabajador
 per.mostrar(); // per se trata como cualquier Trabajador
 pelas = per.sueldo();
       // Ilegal (sueldo se define en Trabajador)

 boolean b1, b2;
 b1 = (per instanceof Persona);  // True
 b2 = (per instanceof Trabajador); // True
 System.out.println("b1: "+ b1 + " b2: "+ b2);
  }
```



## Enlace dinámico

**Nota:**

En C++ sólo hay enláce dinámico en los métodos virtual

En Java, todos los métodos son virtuales



## La palabra reservada super

- Una clase utiliza super para apuntar a su superclase.
- Super se utiliza para apuntar a los miembros de la superclase.
- Los métodos de la superclase se invocan como si el objeto fuera parte de la subclase.



## La palabra reservada super

```
public class Empleado{
 private String nombre;
 private int salario;
 public String getDetails(){
 return "Nombre: "+ nombre + "\nsalario:  "
  + salario; }}
public class Jefe extends Empleado {
 private String departamento;
 public String getDetails(){
  return super.getDetails()+
  "\nDepartamento: " + departamento;
//llama al método del padre
 }}
```