Tema 5: Java

Introducción:

- Sun Microsystems 1995
- Versión 1.5, Java 5
- Ventajas:
  - Sencillez
  - Bibliotecas definidas
  - No sólo compilador
  - Gratuito desde su origen
  - Buena adaptación para aplicaciones web

Java es una plataforma de desarrollo:

- Lenguaje
- Bibliotecas: (Java core) strings, procesos, E/S, propiedades del sistema, fecha/hora, applets, redes, internacionalización, seguridad, acceso a BDs…
- Herramientas:
  - compilador de bytecodes,
  - generador de documentación (javadoc),
  - depurador
- Entorno de ejecución.

Entornos de libre disposición:

- Netbeans (java.sun.com o www.net-beans.org)
- Eclipse (IBM, www.eclipse.org)
- Blue J (www.bluej.org).

Entornos comerciales:

- JBuilder (Borland) (www.borland.com/products/downloads) con una versión (foundation) de uso gratuito.
- JCreator Pro (www.jcreator.com) con una versión (LE) gratuita.

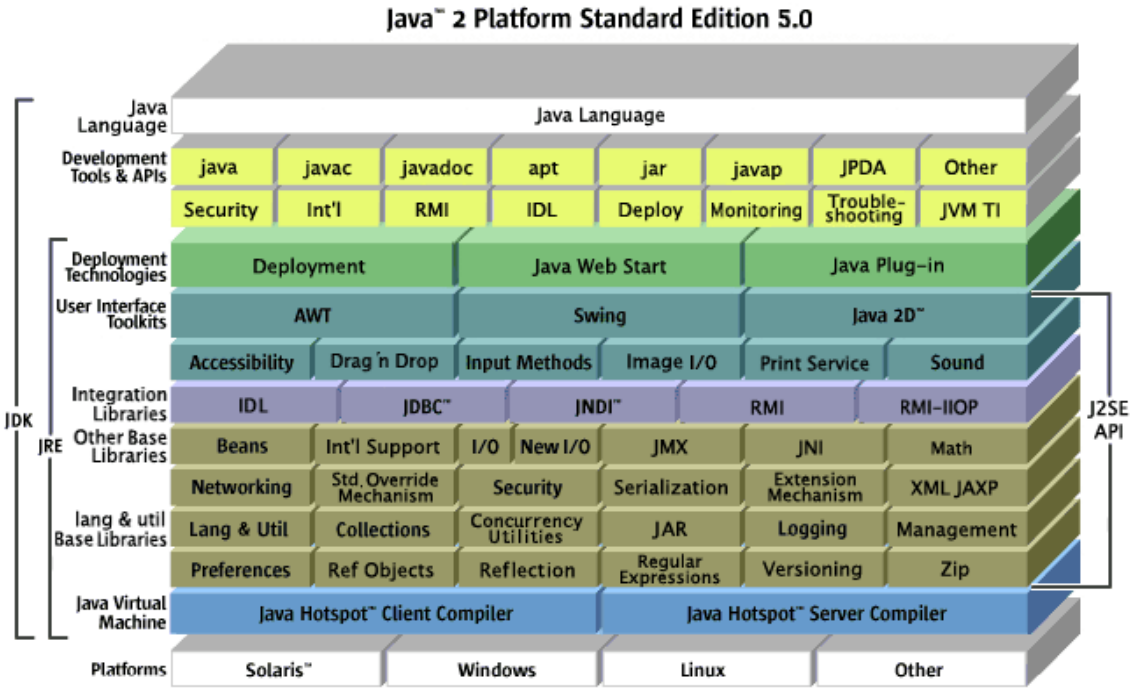
Especificación del lenguaje:

http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html

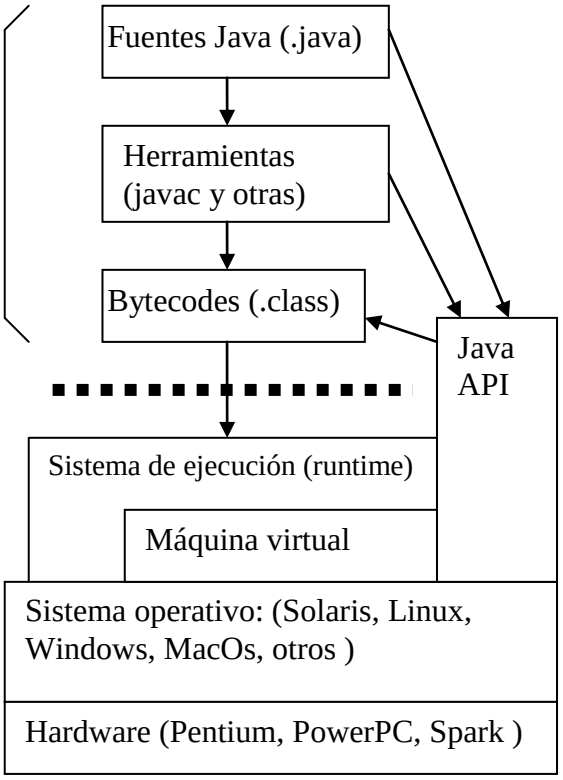
Documentación de la jerarquía de clases disponible

Tanto en línea como fuera de línea: javadoc de la API.

http://java.sun.com/j2se/1.6.0/docs/api/index.html



Esquema funcional de la plataforma Java:



Proceso de compilación

Java es más que un lenguaje La palabra Java se refiere a dos cosas inseparables:

- El lenguaje que nos sirve para crear programas
- La Máquina Virtual Java que sirve para ejecutarlos.

Como vemos en la figura, la API de Java y la Máquina Virtual Java (JVM) forman una capa intermedia (Java platform) que aísla el programa Java del hardware (hardware-based platform)

El lenguaje Java es a la vez compilado e interpretado:

- El compilador convierte el código fuente **.java,** a un conjunto de instrucciones (*bytecodes)* que se guardan en un archivo **.class** (estas instrucciones son independientes del tipo de ordenador).
- El intérprete ejecuta cada una de estas instrucciones en un PC específico (Windows, Mac, Linux...).

Cada intérprete Java es una implementación de la Máquina Virtual Java (JVM). Los *bytecodes* posibilitan el objetivo de "*write once, run anywhere*" que disponga de una implementación de la JVM.

Sola es necesario compilar una vez el programa, pero se interpreta cada vez que se ejecuta en un PC.

Un **ejemplo**: Hola mundo…

C++

```
#include <iostream>

int main( int argc, char *argv ){
        std::cout << "Hola mundo";
}
```

Java

```
import java.lang.*; //se importa por defecto
                    // no hace falta ponerlo
public class Programa{
        public static void main( String [] args ){
                System.out.println("Hola mundo");
        }
}
```

Notas:

- Un programa Java es una colección de clases. Cada clase puede tener su propio main( ) (para probar su funcionamiento), por lo que al ejecutar hay que indicar cuál es el la clase principal.
- Una clase es una colección de atributos (datos miembros) y métodos que definen el comportamiento de un objeto (cada clase define un nuevo tipo de datos). Una clase puede ser definida por el usuario o estar ya definida en algunos de las librerías (paquetes) incorporados a Java.
- Cada objeto instanciado de una clase contiene su propia copia de los atributos, excepto de los atributos estáticos que son comunes a todos los objetos (son atributos de la clase)
- En Java, siempre se codifica dentro de clases: no hay funciones no miembros como tales.
- Aunque no esté escrito explícitamente, la clase Programa hereda de **Object**. En Java, todas las clases descienden de la clase **Object** con mayor o menor profundidad.
- Las clases **Object** y **System** están en el paquete (similar a biblioteca) **java.lang**, que se importa por defecto.
- El prototipo del método **main( )** debe ser el que se ve en el ejemplo
- El código anterior debe estar por fuerza en un archivo llamado **Programa.java** para que el compilador lo acepte ya que la clase pública se llama Programa. Un fichero fuente (.java) puede contener más de una clase, pero sólo una puede ser **public**. El nombre del fichero fuente debe coincidir con el de la clase **public** (con la extensión .java). Si la clase no es **public**, no es necesario que su nombre coincida con el del fichero.
- Una clase puede ser **public** o **package** (default, si no calificamos), pero no **private** o **protected**.
- Las clases de Java se agrupan en **packages (**paquetes**)**, que son librerías de clases. Si las clases no se definen como pertenecientes a un package, se utiliza un package por defecto (default) que es el directorio activo.
- Un **package** es una agrupación de clases. Existen una serie de packages incluidos en el lenguaje (véase tabla inferior). Además el usuario puede crear sus propios packages. Lo habitual es juntar en packages las clases que estén relacionadas. Todas las clases que formen parte de un package deben estar en el mismo directorio

Paquete	**Descripción**
java.applet	Contiene las clases necesarias para crear applets que se ejecutan en la ventana del navegador
java.awt	Contiene clases para crear una aplicación GUI independiente de la plataforma
java.io	Entrada/Salida. Clases que definen distintos flujos de datos
java.lang	Contiene clases esenciales, se importa implícitamente sin necesidad de una sentencia **import**.
java.net	Se usa en combinación con las clases del paquete java.io para leer y escribir datos en la red.
java.util	Contiene otras clases útiles que ayudan al programador

PRINCIPALES CARACTERÍSTICAS QUE JAVA NO HEREDA DE C++:

- **Sobrecarga de operadores:** En Java no se pueden sobrecargar los operadores. Si creamos una clase y queremos por ejemplo sumar 2 objetos, en vez de sobrecargar operator+(es posible en C++, pero no en Java) simplemente creamos un método sumar( ) y lo invocamos.

- **Punteros:** En Java no existen los punteros como tales (ni existen los operadores * y ->).
  El inadecuado uso de los punteros provoca la mayoría de los errores de colisión de memoria, errores muy difíciles de detectar.

- **Creación y destrucción de objetos**: Para crear un objeto, Java usa la palabra reservada ***new***
  Ej: **Punto p1=new Punto(1, 2), p2;** en C++ → **Punto *p1=new Punto(1,2), *p2;**
  p2=p1; **p2=p1;** //apuntan al mismo objeto
  new reserva espacio en memoria para el objeto y devuelve una referencia (puntero en C++) que se guarda en la variable **p1** de tipo **Punto** que denominamos ahora objeto.

  En el lenguaje C++, los objetos que se crean con **new** se han de eliminar con **delete**. En Java no es necesario liberar la memoria reservada (no hay destructores como en C++), el recolector de basura (***garbage collector***) se encarga de hacerlo por nosotros, liberando al programador de dicha tarea. Para ello, el recolector de basura marca como "borrable" aquellos objetos que han perdido su referencia (por llegar al final del bloque en el que habían sido definidos, porque a la referencia se le asigna un valor **null** o porque a la referencia se le asigna la dirección de otro objeto).

  Ej: **p1=null;** //p1 ya no apunta a Punto(1,2), pero p2 aun si… → no borrable
  p2=new Punto(0,0); //ahora Punto(1,2) creado antes ya no es apuntado por nadie → borrable

  El recolector de basura se ejecuta siempre que el **sistema esté libre**, o cuando una asignación **solicitada no encuentre asignación suficiente** (si no falta memoria es posible que no se ejecute hasta finalizar el programa). Se puede "forzar" su ejecución invocando el método ***System.gc()***

  Como podemos observar, en Java las referencias son como los punteros en C++

- **En Java no es posible crear objetos constantes, solo es posible crear referencias constantes**
  Solución Java: crear una clase inmutable → objetos constantes
  En C++ es posible crear objetos constantes y punteros a objetos constantes → Los objetos constantes sólo podían ejecutar métodos constantes (***const***).
  Como en Java eso no es posible, nos olvidamos de los métodos *const* (no existen como tales)

  Ej: **final Punto p1=new Punto(1, 2);** en C++ → **Punto *const p1=new Punto(1,2);**
  p1=new Punto(0,0) //ERROR p1 es un "puntero" constante (no puede apuntar a otro sitio)
  p1.set(2,3); //PERMITIDO cambio las coordenadas del punto
  final int n; //PERMITIDO Java permite declarar una constante
  n=4; //e inicializarla después
  n=5; //ERROR //una vez inicializada una constante, ya no es posible cambiar su valor

  En Java **final** es como **const** en C++
  Una variable de tipo primitivo declarada como final no puede cambiar su valor (es una constante) pero un objeto final no significa que el objeto sea constante, sino que su referencia es contante

  No es lo mismo en C++ **Punto *const p1** que **const Punto * p1**
  Punto *const p1 → p1 es un puntero constante a un objeto Punto Java → **final Punto p1;**
  const Punto * p1→p1 es un puntero a un objeto Punto constante Java → no es posible

- **En Java no existen los inicializadores en los constructores (todo se codifica dentro del cuerpo del constructor) → menos confusión y los atributos estáticos se pueden inicializar en el momento de declararlos (mas intuitivo y "lógico")**

EN RESUMEN: JAVA ELIMINA TODA LA SINTAXIS CONFUSA DEL C++ (ES MÁS LEGIBLE)

NOMENCLATURA HABITUAL EN LA PROGRAMACIÓN EN JAVA

Java es sensible a mayúsculas y minúsculas: ***mas***, ***Mas*** y ***MAS*** son consideradas variables diferentes. Convenciones:

Tipo de identificador	**Convención**	**Ejemplo**
nombre de clase o interface	comienza por mayúscula	*String, Rectangulo, CuboGrafico*
objetos y variables locales	comienza con minúscula	a, total, alumno, alumnoRepetidor
métodos, atributos, paquetes	comienza por minúscula	area, calcularArea(), color, getColor()
constante (***final***)	En letras mayúsculas	PI, MAX_ANCHO

VARIABLES: en ***Java*** hay dos tipos principales de variables:

1. **Variables de *tipos primitivos***: Almacenan un valor único que puede ser entero, de punto flotante, carácter o booleano. ***Admiten los mismos operadores aritméticos, relacionales y lógicos que C++.***

Tipo	**Descripcion**
boolean	1 byte. Tiene dos valores **true** o **false**.
char	2 bytes Unicode de 16 bits (comprende el codigo ASCII).
byte	1 byte. Valor entero entre -2^7 hasta $2^7 -1$ (-128 a 127)
short	2 bytes. Valor entero entre -2^{15} hasta $2^{15}-1$ (-32768 a 32767)
int	4 bytes. Valor entero entre -2^{31} hasta $2^{31}-1$ (-2147483648 a 2147483647)
long	8 bytes. Valor entero $[-2^{63}, 2^{63}-1]$ (-9223372036854775808 a 9223372036854775807)
float	4 bytes. Valor decimal de 6 cifras decimales de precisión. (de -1.40239846e–45f a 3.40282347e+38f) Ej: float a=2.34f, b=.35f //se pone una f para indicar float
double	8 bytes. Valor decimal de 15 cifras decimales de precisión. $[-10^{-324}, 10^{-308}]$ (de -4.94065645841246544e–324 a 1.7976931348623157e+308.)

Cuando un tipo primitivo se asigna a otro se produce una **conversión automática** de tipos siempre que los dos tipos sean compatibles y tipo destino sea mayor que tipo origen (si no, hay que hacer **casting**)

2. Variables *referencia*: son punteros a ***arrays*** u ***objetos*** (guardan la dir de memoria en la que se almacena el objeto). No confundir referencia con objeto: una referencia es un puntero a un objeto. Java no permite (como en C++) referencias a variables de tipo primitivo.

Las únicas operaciones que admiten las variables referencias son:
- Los operadores = , = = y != para manipular la referencia
- Los operadores de casting, el operador . y el operador ***instanceof***

El operador =
En los tipos primitivos copian el valor de la derecha en la variable de la izq.
En las referencias hace que la ref de la izq apunte al mismo objeto al que apunta la ref de la dcha (como los punteros a objetos en C++), es decir, no copia objetos sino el valor de la dir de memoria del objeto.

El operador = = / !=
En los tipos primitivos true si las variables izq y dcha tienen el mismo / distinto valor.
En las referencias true si las referencias izq y dcha apuntan al mismo / distinto objeto (como los punteros a objetos en C++), es decir, no compara los objetos, si no las direcciones de memoria de los objetos

Los operadores de casting
Convierten una ref a un objeto de un tipo en una ref a un objeto de otro tipo, que debe ser un tipo ascendiente o descendiente en la jerarquía de herencia (en C++ podemos convertir a tipos no heredados)

El operador .
Permite acceder a los métodos y atributos del objeto al que apunta la referencia

El operador instanceof (ref instanceof nombreClase)
Permite saber si ref apunta a un objeto de tipo (o derivado de) *nombreClase* (parecido al **typeid** de C++)

Tipos primitivos vs Clases envolventes o wrappers (wrapper class)

Los tipos de datos primitivos no son objetos pero en ocasiones es necesario tratarlos como tales. Por ejemplo, hay determinadas clases que manipulan objetos (ArrayList, HashMap, …).

Para poder utilizar tipos primitivos con estas clases, Java proporciona las llamadas clases envolventes también llamadas clases contenedoras o wrappers.

Cada tipo primitivo tiene su correspondiente clase envolvente, que contiene métodos que permiten manipular el tipo de dato primitivo como si fuese un objeto:

Tipo primitivo	Clase Envolvente
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

Las conversiones entre los tipos primitivos y sus clases envolventes son automáticas (no hace falta hacer casting).

Para realizarlas se utiliza el Boxing/Unboxing.

Boxing: Convertir un tipo primitivo en su clase Wrapper.
Unboxing: Convertir un objeto Wrapper en su tipo primitivo.

Ejemplo de Boxing:	Ejemplo de Unboxing:

double x = 29.95;
Double y;
y = x; // boxing
 	
y = new Double(x);
	double x;
	Double y = 29.95; //y=new Double(29.95);
	x = y; // unboxing
	
	x = y.doubleValue();

Las clases envolventes contienen métodos que permiten, entre otras cosas, la conversión de datos de unos tipos primitivos a otros, o desde una cadena de caracteres a un numérico y viceversa. Igualmente se utilizan en las colecciones de datos de tipo Vector, Stack y Hashtable que sólo pueden contener objetos.

En la siguiente tabla se recogen los métodos más utilizados de las clases envolventes. Los métodos parselnt(), parseLong(), parseByte(), parseShort(),parseFloat() y parseDouble() permiten obtener los correspondientes valores numéricos a partir de una cadena de caracteres. Las diferentes versiones (sobrecargadas) del método toString() nos permiten obtener una cadena de caracteres del valor numérico dado como argumento. El método valueOf() en sus diferentes versiones nos permite obtener objetos de la clase a la que pertenezca, tomando como valor la cadena de caracteres recogida como argumento.

Nombre del Método	**Tarea que realiza**
static int parseInt(String s)	Permite obtener el valor entero de una cadena de caracteres.
String toString(int i)	Permite convertir un valor entero en una cadena de caracteres.
static Integer valueOf (String s)	Obtiene un objeto de tipo Integer con el valor de la cadena de caracteres.
static Integer valueOf (int i)	Obtiene un objeto de tipo Integer con el valor int
int intValue( )	Devuelve el valor int asociado al objeto Integer
static double parseDouble (String s)	Permite obtener el valor de tipo double de una cadena de caracteres
String toString(double d)	Permite convertir un valor double en una cadena de caracteres.
static Double valueOf(String s)	Obtiene un objeto de tipo Double con el valor de la cadena de caracteres
static Double valueOf (double i)	Obtiene un objeto de tipo Double con el valor doublé
double doubleValue( )	Devuelve el valor double asociado al objeto Double

Las clases envolventes son inmutables (no contienen métodos que permita modificar el objeto creado). Las clases envolventes permiten realizar operaciones de conversión, por ejemplo, convertir una cadena de caracteres en una variable numérica, o viceversa. Ej:

double numero = Double.parseDouble("12.34"); //convierte la cadena "12.34" en double

Las variables pueden ser:

- Variables ***de instancia (atributos*** o miembros de objetos): cada objeto tiene su propia copia. Pueden ser *tipos primitivos* o *referencias* (composición).

- Variables ***de clase (static)***: todos los objetos comparten dichas variables.

- Variables ***locales***: Se definen y crean *dentro de un bloque* { }. Se destruyen al finalizar dicho bloque.

Java inicializa las variables de instancia (*atributos*) y de clase (*static*) a un valor por defecto: las numéricas (***byte, short, int, long, float, double***) a 0, ***boolean*** a false, ***char*** a '\0' y las referencias a **null**).
Las variables locales no se inicializan por defecto (hay que hacerlo explícitamente)

Ej: (marcamos en amarillo las líneas que se ejecutan y en rojo las líneas erróneas…)

```
public class Circulo{
        String s;  //atributo o variable de instancia (apunta a null: no apunta a ningún objeto aún)
        static final double PI=3.1416; //variable de clase
        double radio;                  //atributo o variable de instancia (se inicializa por defecto a 0)
        public Circulo(String s, double d)  {  this.s=s; radio=d;  }
        public double calcularArea() {
                double area=PI*radio*radio; //variable local (no se inicializada a nada por defecto)
                return area;
        }
        public boolean iguales(Circulo c) { //variable local (parámetro)
                return (s.equals(c.s) && radio== c.radio);
        }
        public void setRadio(double radio) {   this.radio=radio;  }
        public void ver( ) {  System.out.println(s + " radio: " + radio);  }
        public static void main( String [] args ) {
                Circulo f=null, g;  //f no apuntan a ningún objeto y g no se sabe a donde apunta
                g.calcularArea( );  //ERROR g no ha sido inicializado (no se sabe a donde apunta)
                g=null; //g apunta a null (no apunta a ningún objeto aún)
                f.calcularArea( );  //ERROR f apunta a null
                if (f==g)  System.out.println("f  y c apuntan a lo mismo"); //true: ambos apuntan a null
                Circulo c=new Circulo ("YO", 1);  //c es una ref que apunta al objeto Circulo creado
                f=new Circulo ("YO",1); //la ref f apunta ahora a un objeto Circulo clon del anterior
                f.calcularArea( );
                int a=5, b;   //b tiene un valor aleatorio (las variables locales no se inicializan por defecto)
                b=5;
                if (f.iguales(c))  System.out.println("f  y c son objetos iguales");
                f.setRadio(2);
                f.ver( );  //YO radio: 2.0
                c.ver( );  //YO radio: 2.0
                if (f.iguales(c) == false)  System.out.println("f  y c son objetos distintos");
                if (f==c)  System.out.println("f  y c apuntan al mismo objeto");
                if (a==b)  System.out.println("a y b almacenan el mismo valor");
                f=c; //la ref f apunta al mismo objeto al que apunta c
                      //el objeto al que apunta f antes queda desreferenciado → borrable
                if (f==c)  System.out.println("f  y c apuntan al mismo objeto");
                f.setRadio(5);  //como f y c apuntan al mismo objeto f. es lo mismo que c.
                f.ver( );  //YO radio: 5.0
                c.ver( );  //YO radio: 5.0
        }
}
```

COMPARACIÓN DE UN PROGRAMA JAVA Y SU EQUIVALENTE EN C++

JAVA

```
import java.io.*;




class Fecha {
  protected int dia, mes, anio;
  public Fecha(int d, int m, int a) {
    dia=d; mes=m; anio=a;
  }
  public Fecha(int a) { this(1,1,a); }
  public int getdia() { return dia; }
  public int getmes() { return mes; }
  public int getanio() { return anio; }

  public Fecha(Fecha f) { dia=f.dia; mes=f.mes; anio=f.anio; }
  Fecha clonar() { return new Fecha(this); }

  public String cadena() {
    return String.format("%02d/%02d/%02d", dia, mes, anio);
  }
  public String toString() {
            return cadena();
  }       //return  new String(dia+"/"+mes+ "/"+anio);
}         //return  ""+ dia+"/"+mes+ "/"+anio;




class FechaMutable extends Fecha {
  public FechaMutable(int d, int m, int a) { super(d, m, a); }
  public void set(int d, int m, int a) {
            dia=d; mes=m; anio=a;
  }

  public boolean equals(Object o ) {
            if (o instanceof FechaMutable) {
              FechaMutable fm=(FechaMutable) o;
              return (dia==fm.dia && mes==fm.mes
                                  && anio==fm.anio);
            }
            return false;
  }
}

class Persona {
  static private int n=0; //=0 por defecto
  private FechaMutable fm; //=null por defecto
  private int dni; //=0 por defecto
  private Fecha nace; //=null por defecto

  public Persona(FechaMutable fm, int dni, Fecha f) {
    n++;
    this.fm= new FechaMutable(fm.getdia(), fm.getmes(),
                                                fm.getanio());
    this.dni=dni;
    nace=f; //nace=new Fecha(f); //new  no necesario xq
  }                                  // Fecha es inmutable
```

C++

```
#include <iostream>
#include <sstream> //para usar stringstream
#include <string> //para usar string
using namespace std;

class Fecha {
protected:
  int dia, mes, anio;
public:
  Fecha(int d, int m, int a) { dia=d; mes=m; anio=a; }
  Fecha(int a) { Fecha(1,1,a); }
  int getdia() { return dia; }
  int getmes() { return mes; }
  int getanio() { return anio; }

  Fecha(Fecha *f) { dia=f->dia; mes=f->mes; anio=f->anio; }
  Fecha* clonar() { return new Fecha(this); }

  string toString() {
    stringstream s;
    s << dia << "/" << mes << "/" << anio;
    return s.str();
  }

/*friend ostream & operator<<(ostream &s, Fecha *f) {
    s << f->dia << "/" << f->mes << "/" << f->anio;
    return s;
  }*/
};

class FechaMutable: public Fecha {
public:
  FechaMutable(int d, int m, int a): Fecha(d, m, a) { }
  void set(int d, int m, int a) {
            dia=d; mes=m; anio=a;
  }

  bool iguales(FechaMutable *fm) {
    return (dia==fm->dia && mes==fm->mes
                                && anio==fm->anio);
  }

};




class Persona {
  static int n;
  FechaMutable *fm;
  int dni;
  Fecha *nace;
public:
  Persona(FechaMutable *fm, int dni, Fecha *f) {
    n++;
    this->fm= new FechaMutable(fm->getdia(), fm->getmes(),
                                                fm->getanio());
    this->dni=dni;
    nace=f; //nace=new Fecha(f); //new  no necesario xq
  }                                  // Fecha es inmutable
```

```
public Persona(Persona p) {
   n++;
   fm=new
FechaMutable(p.fm.getdia(),p.fm.getmes(),p.fm.getanio());
   dni=p.dni;
   nace=p.nace; //nace=new Fecha(p.nace);
  }

  public void finalize() { n--; }

  Persona clonar() { return new Persona(this); }

  boolean iguales(Persona p) {
    boolean iguales=true;
    if (!fm.equals(p.fm) || nace.getdia() != p.nace.getdia() ||
        nace.getmes() != p.nace.getmes() ||
        nace.getanio() != p.nace.getanio() )
      iguales=false;
    return iguales;
  }




  public String toString() {
            String s="" + fm + ", " + dni + " ("
                        + nace + ")";
            return s;
  }

  void ver() {
     System.out.print(fm + ", " + dni + " ("
                              + nace + ")\n");
  }

  void ver(int n) { System.out.print("FINAL\n"); }
}



public class Socio extends Persona {
  private final double cuota;
  final private Fecha alta;
  public Socio(Persona p, double c, Fecha  a) {
    super(p);
    cuota=c;
    alta=a;
  }
  public Socio(FechaMutable fm, int dni, Fecha f,
                    double c, Fecha a) {
    super(fm, dni, f);
    cuota=c;
    alta=a;
  }
  public Socio(Socio s) {
     super(s);
     cuota=s.cuota;
    alta=new Fecha(s.alta);
  }

  boolean iguales(Socio s) {  //no comparo alta
    return (super.iguales(s) && cuota==s.cuota);
  }
```

```
Persona(Persona *p) {
   n++;
   this->fm=new FechaMutable(p->fm->getdia(),
                        p->fm->getmes(),p->fm->getanio());
   dni=p->dni;
   nace=p->nace; //nace=new Fecha(p->nace);
  }

  virtual ~Persona() { n--; }

  Persona* clonar() { return new Persona(this); }

  bool iguales(Persona *p) {
    bool iguales=true;
    if (!fm->iguales(p->fm) || nace->getdia() != p->nace->getdia() ||
        nace->getmes() != p->nace->getmes() ||
        nace->getanio() != p->nace->getanio() )
      iguales=false;
    return iguales;
  }

/*friend ostream & operator<<(ostream &s, Persona *p) {
    s << p->fm << ", " << p->dni << " (" << p->nace << ")";
    return s;
  }*/

  virtual string toString() {
    stringstream s;
    s << fm->toString() << ", " << dni << " ("
      << nace->toString() << ")";
    return s.str();
  }

  virtual void ver() {
     cout << fm->toString() << ", " << dni << " ("
          << nace->toString() << ")" << endl;
  }
  void ver(int n) { cout << "FINAL\n"; }

};

int Persona::n=0;

class Socio: public Persona {
  const double cuota;
  Fecha * const alta; //no es lo mismo que const Fecha *alta;
public:
  Socio(Persona *p, double c, Fecha *a):Persona(p),
                                        cuota(c),
                                        { alta=a; }

  Socio(FechaMutable *fm, int dni, Fecha *f,
          double c, Fecha *a)
    :Persona(fm, dni, f), cuota(c), alta(a) {  /*alta=a;*/ }
  //el atributo alta no es constante por lo que no es obligatorio
  //ponerlo en la zona de inicializadores (puedo ponerlo o no)

  Socio(Socio *s):Persona(s),
                  cuota(s->cuota),
                  alta(new Fecha(s->alta)) { }

  bool iguales(Socio *s) {   //no comparo alta
    return (Persona::iguales(s) && cuota==s->cuota);
  }
```

```
 public String toString() {
        String s=super.toString() + ", " + cuota
                                  + ", " + alta;
        return s;
 }

 void ver() {  //sobreescribo el ver( ) del padre
        super.ver();
        System.out.print(cuota + ", " + alta + "\n");
        ver(1); //En Java al sobreescribir un método en
 }              //la hija solo oculto el del padre con el
                //mismo prototipo


public static void main(String[] args) {
   Fecha f1=new Fecha(1,1,2001);
   final FechaMutable fm1=new FechaMutable(2,2,2010);
   FechaMutable fm2=new FechaMutable(2,2,2000);
   Fecha f2=new Fecha(fm1);
   fm1.set(1,1,2001); //puedo cambiar el contenido
 //fm1=new FechaMutable(2,2,2002); //no puedo cambiar
la referencia
 //f2.set(2,2,2012); //set no definido en fecha
    f2=new Fecha(2,2,2012);
    Persona a=new Persona(fm1, 12, f1),
            b=new Persona(fm2, 13, f1);
    final Persona p=new Persona(fm1, 14,
                                new Fecha(5,5,1990));
    a=p.clonar();
    if (a.iguales(p)) System.out.println(a + " y "
                                  + p + " son iguales");
    if (p.iguales(b)) System.out.println(p + " y "
                                  + b + " son iguales");

    Socio s1=new Socio(a,12.5, f1), s3;
    Socio s2=new Socio(fm1, 14, new Fecha(5,5,1990),
                       13.0, f2);

    s3=s1; //copia punteros no objetos...
    if (s1.iguales(s2)) System.out.println(s1 + " y "
                                  + s2 + " son iguales");
        if (s3.iguales(s1)) System.out.println(s3 + " y "
                                  + s1 + " son iguales");

    System.out.println("-----");
    s3.ver(2);
    System.out.println("-----");
    Persona z;
    z=b; z.ver();
    z=s1; z.ver();
 }
}
```

```
/*friend ostream & operator<<(ostream &s, Socio *p) {
   s << (Persona *)p << ", " << p->cuota << ", " << p->alta;
   return s;
 }*/

 string toString() {
  stringstream s;
  s << Persona::toString() << ", " << cuota
                           << ", " << alta->toString();
        return s.str();
 }

 void ver() { //sobreescribo el ver( ) del padre
  Persona::ver();
  cout << cuota << ", " << alta->toString() << endl;
  Persona::ver(1); //En C++ al sobrecargar o sobreescribir
 }                 //un método en la hija oculto todos los del
};                 // padre con el mismo nombre

int main() {
 Fecha *f1=new Fecha(1,1,2001);
 FechaMutable* const fm1=new FechaMutable(2,2,2010);
 FechaMutable* fm2=new FechaMutable(2,2,2000);
 Fecha *f2=new Fecha(fm1);
 fm1->set(1,1,2001); //puedo cambiar el contenido
 //fm1=new FechaMutable(2,2,2002); //no puedo cambiar el
puntero
 //f2->set(2,2,2012); //set no definido en fecha
 f2=new Fecha(2,2,2012);
 Persona *a=new Persona(fm1, 12, f1),
         *b=new Persona(fm2, 13, f1);
 Persona * const p=new Persona(fm1, 14,
                               new Fecha(5,5,1990));
 a=p->clonar();
 if (a->iguales(p)) cout << a->toString() << " y "
                         << p->toString() << " son iguales\n";
 if (p->iguales(b)) cout << p->toString() << " y "
                         << b->toString() << " son iguales\n";

 Socio *s1=new Socio(a,12.5, f1), *s3;
 Socio *s2=new Socio(fm1, 14, new Fecha(5,5,1990),
                         13.0, f2);

 s3=s1; //copia punteros no objetos...
 if (s1->iguales(s2)) cout << s1->toString() << " y "
                           << s2->toString() << " son iguales\n";
 if (s3->iguales(s1)) cout << s3->toString() << " y "
                           << s1->toString() << " son iguales\n";
 cout << "-----\n";
 s3->Persona::ver(2);
 cout << "-----\n";
 Persona *z;
 z=b; z->ver();
 z=s1; z->ver();
 delete s1; delete s2;

 system("pause");  return EXIT_SUCCESS;
}
```

OBJETOS, MIEMBROS Y REFERENCIAS

EL OPERADOR NEW

El operador *new* crea una instancia de una clase (*objetos*) y devuelve una referencia a ese objeto:

MiPunto p2 = new MiPunto(); //en C++ MiPunto *p2 = new MiPunto();

ejemplo:

```
Punto p;   //se ha creado una referencia, pero aun no se ha creado ningún objeto
p = new Punto(); //new crea un objeto Punto y hacemos que la ref p apunte al objeto
```

CONSTRUCTORES

Todos los objetos se deben inicializar cuando se crean, para que tengan un estado bien definido. De dicha tarea se encarga el constructor (método con el mismo nombre que la clase)
Al igual que en C++, el constructor se llaman cada vez que se crea un objeto de una clase

Ejemplo constructor con parámetros:

```
class Punto {
   int x , y ;
   Punto ( int a , int b ) {
      x = a ; y = b ;
   }
}
```

Con este constructor se crearía un objeto de la clase Punto de la siguiente forma:

```
Punto p = new Punto ( 1 , 2 );
```

CONSTRUCTOR POR DEFECTO:

- **Si una clase no declara ningún constructor, Java incorpora un constructor por defecto que no hace nada** (en Java los atributos de una clase se inicializan por defecto a: 0 los numéricos (***byte, short, int, float, long, double***), false los ***boolean*** y null las referencias)
- Si se **declara algún constructor,** entonces Java no crea ninguno.

```
class Punto {
   int x , y ;
   Punto ( ) { } //constructor por defecto que crea Java si nosotros no creamos ninguno
}
```

En Java no existen los parámetros por defecto → Sobrecarga de constructores

```
class Punto {
   int x , y ; //por defecto a 0
   Punto ( int x, int y ) {
      this.x = x ; this.y = y ;
   }
   Punto ( ) {
      x = 0 ; y = 0;
//inicialización no necesaria
   }
}
```

```
class Punto {
   int x , y ;
   Punto ( int a , int b ) {
      x = a ; y = b ;
   }
   Punto ( ) {
      this (0,0); //llama al constructor
   }              //anterior
}
```

La palabra this solo puede aparecer en la primera sentencia de un constructor
Se usa para hacer una llamada a otro constructor de la misma clase

ÁMBITOS: DIFERENCIAS CON EL C++

En Java, en relación al **acceso de los métodos y de los atributos**, existen **4 ámbitos**:

- **Público** (public): acceso total: desde la propia clase y desde otras clases.
- **Privado** (private): acceso limitado a la propia clase.
- **Protegido**(protected): acceso limitado a la clase, a sus subclases (a cualquier nivel de descendencia) **y al resto de las clases del mismo paquete.**
- **De Paquete**: (por defecto). Acceso limitado a las **clases del mismo paquete.**

El acceso de paquete es lo más parecido que hay en Java a las clases friend de C++. Java no implementa clases friend como tales. Se permite la modificación de miembros **protected** pero en ningún caso se permite el acceso de una clase a los miembros privados de otra.

```
public class HelloWorld {
        char depaquete;                  //todas las clases del paquete pueden acceder
        protected int protegido=1;       //las clases hijas y las clases del paquete pueden acceder
        public String publica="Hola";    //todas las clases pueden acceder
        private double privada;          //solo la propia clase puede acceder
        public int metodoPublico(int parametro){
                return parametro;
        }
        String metodoPaquete(String Cadena){
                return  Cadena;
        }
        private  void metodoPrivado(double real){
                privada = real;
        }
        public static void main(String[] args) {
                System.out.println("Hello World");
        }
}

class DePaquete{

        public HelloWorld Hi= new HelloWorld();
        public void metodoAccede(){
                String a = Hi.metodoPaquete("hola");
                Hi.metodoPublico(1);
                //Hi.metodoPrivado(1.1);
                Hi.protegido = 5; //Aunque no sea herencia
        }
}
```

MODULARIDAD: PAQUETES, package e import.

- Los paquetes son una forma de organizar grupos de clases. Un paquete contiene un conjunto de clases relacionadas bien por finalidad, por ámbito o por herencia.
- Los paquetes resuelven el problema del conflicto entre los nombres de las clases. Al crecer el número de clases crece la probabilidad de designar con el mismo nombre a dos clases diferentes.
- Los nombres de los ***packages*** se escriben en minúsculas. El nombre de un ***package*** puede constar de varios nombres unidos por puntos (los ***packages*** de ***Java*** siguen esta norma, ej ***java.awt***).
- Todas las clases que forman parte de un ***package*** deben estar en el mismo directorio.
- Java proporciona varios paquetes predefinidos:
  - **java.lang**: contiene las clases Integer, Math, String, System…
  - **java.util**: Date, Random, StringTokenizer…
  - **java.io**: se usa para la E/S, contiene los canales (stream) System.in, System.out

Las clases pueden ser **públicas** o de **paquete** (por defecto, si no se indica nada)

No existen archivos de cabecera e implementación -- >
Fuente (java) → Compilación bytecodes → Ficheros (.clas y .jar)

Las clases son accedidas en los .class cuando son requeridas:
→ Al crear un objeto de dicha clase
→ Al referenciar un método éstatico de la clase

Colisiones → El interprete necesita distinguir entre dos clases con el mismo nombre→ java agrupa el concepto de **paquetes → directorios del sistema de ficheros**.

Proceso: los paquetes se buscan en los lugares indicados en la variable de entorno **CLASSPATH**
El intérprete encuentra el **CLASSPATH** en el entorno del sistema
- Los **puntos** en los paquetes son sustituidos por barras invertidas , busca las clases en el directorio resultante:

Ej: **package foo.bar.baz --> foo\bar\baz or foo/bar/baz** dependiendo del SO

INCLUSIÓN DE CLASES EN UN PAQUETE

Para indicar que una clase pertenece a un paquete:
- la primera linea del fichero **package <nombre de paquete>;**
- El fichero debe estar en un directorio **<nombre de paquete>**

Si no se indica paquete → paquete **por defecto**

REFERENCIA A OTROS PAQUETES: import (permite usar clases **public** de otros paquetes)

Las directivas **import deben ser el siguiente código después de la directiva package**:

import <nombre de paquete>.{<clase>|*};
→ *, indica todas las clases **publicas** incluidas en el paquete;
→ Subpaquetes. Se utiliza el punto para generar subdirectorios:

import java.util.Scanner; //importa la clase publica Scanner
import java.io.*; //importa todas las clases **public** del subpaquete **java.io**
import java.*; //importa todas las clases **public** del paquete **java,** pero no de los subpaquetes
//al importar un paquete no importa los subpaquetes (java.* no importa java.io.*)

En ***Java***, una clase, sus atributos y sus métodos pueden ser referidos con su nombre completo (el nombre del ***package*** más el de la clase, separados por un punto). La sentencia ***import*** permite abreviar los nombres, evitando el tener que escribir continuamente el nombre del ***package*** importado.

Se importan por defecto el ***package java.lang*** y el ***package*** actual o por defecto (las clases del dir actual).
Ej:

```
java.awt.Font fuente=new java.awt.Font("Monospaced", Font.BOLD, 36);
```

Usando import:

```
import java.awt.Font;
Font fuente=new Font("Monospaced", Font.BOLD, 36);
```

Se pueden combinar ambas formas, por ejemplo, en la definición de la clase *BarTexto*

```
import java.awt.*;
public class BarTexto extends Panel implements java.io.Serializable{
//...
}
```

Panel es una clase que está en el paquete *java.awt*, y *Serializable* es un interface que está en *java.io*

COLISIONES (CONFLICTOS DE NOMBRES)

Ocurre cuando hay 2 clases en distintos paquetes que se llama igual
Paquete librería y paquete1 contienen la clase Punto → colisión

Solución 1: Se suele importar la que más se usa y la otra se referencia usando su nombre completo

```
import paquete1.*; //se incluye la más habitual
public class PruebaPunto {
        public static void main (String [] args){
         libreria.Punto a = new libreria.Punto(); //se referencia con path completo la que no se importa
           Punto p=new Punto(); //esta se refiere al paquete paquete1
           String c="hola";
           Prueba b=new Prueba();


        }
}
```

Solución 2: interesa cuando la colisión es mínima (solo unas pocas clases colisionan) y el resto de clases son diferentes. Las clases que colisionan se utilizan con su nombre completo
Si se quiere incluir ambos paquetes:

```
import libreria.*;
import paquete1.*;
public class PruebaPunto {
public static void main (String [] args){
      libreria.Punto a = new libreria.Punto();        // ya no es ambigua
      paquete1.Punto p = new paquete1.Punto();  // ya no es ambigua
}
}
```

COMPOSICIÓN:

Hay dos formas de reutilizar el código, mediante la composición y mediante la herencia.
La composición significa utilizar objetos dentro de otros objetos.

Punto
- int x - int y
+ Punto (int x, int y) + Punto( ) + void desplazar(int, int)

Rectangulo
- Punto origen - int ancho - int alto
+ Rectangulo (Punto p) + Rectangulo ( ) … + void desplazar(int, int) +int calcularArea()

```
package rectangulo;

public class Punto { // class Punto extends Object
  private int x;
  private int y;
  public Punto(int x, int y) {   this.x = x;   this.y = y;    }
  public Punto() {    x=0;    y=0;  }
  void desplazar(int dx, int dy) {   x+=dx;   y+=dy;  }
}
```

```
package rectangulo;

public class Rectangulo { // class Rectangulo extends Object
   private int ancho ;
   prívate int alto ;
   private Punto origen;

   public Rectangulo() { origen = new Punto(0, 0);   ancho=0;  alto=0; }
   public Rectangulo(Punto p) {    this(p, 0, 0);    }
   public Rectangulo(int w, int h) {    this(new Punto(0, 0), w, h);    }
   public Rectangulo(Punto p, int w, int h) {
            origen = p;   ancho = w;    alto = h;
   }
   void desplazar(int dx, int dy) {   origen.desplazar(dx, dy);    }
   int calcularArea() {  return ancho * alto;   }
}
```

```
package rectangulo;

public class RectanguloApp {

   public static void main(String[] args) {
      Rectangulo rect1=new Rectangulo(100, 200);
      Rectangulo rect2=new Rectangulo(new Punto(44, 70));
      Rectangulo rect3=new Rectangulo();
      rect1.desplazar(40, 20);
      System.out.println("el área es "+ rect1.calcularArea() );
      int areaRect=new Rectangulo(100, 50).calcularArea();
      System.out.println("el área es "+ areaRect);
   }
}
```

La composición es la creación de una clase agrupando objetos de otra clase: Los objetos de la nueva clase contienen uno o varios objetos de otras clases.

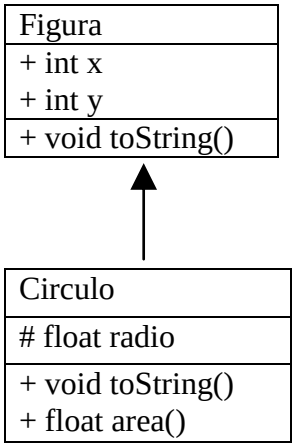
La composición es una relación TIENE UN entre clases (la clase compuesta TIENE UNA clase)

La herencia permite crear una clase a partir de la definición de una clase ya existente. La herencia permite crear una clase heredando todo el comportamiento y el código de la clase padre

La herencia es una relación ES UN entre clases (la clase derivada ES UNA clase base)

HERENCIA SIMPLE:

Se puede construir una clase a partir de otra mediante el mecanismo de la ***herencia***.
Para indicar que una clase deriva de otra se utiliza la palabra ***extends***
Una clase derivada hereda todos los atributos y métodos de la clase base. Puede quedarse con dichos métodos sin más o ***redefinirlos*** (***overridden***). También puede añadir sus propios atributos y métodos, y **sobrecargar** métodos heredados de la clase base.
Java no permite que una clase derive de varias (no es posible la herencia múltiple).
Todas las clases de ***Java*** creadas por el programador tienen una ***super-clase***. Cuando no se indica explícitamente una ***super-clase*** con la palabra ***extends***, la clase deriva de ***java.lang.Object***, que es la clase raíz de toda la jerarquía de clases de ***Java***. Como consecuencia, todas las clases tienen algunos métodos heredados de ***Object***.



Las clases Figura y Circulo son de paquete, al no poner **public.** No puede ser usadas por otra clase de otro paquete aunque se importe

Si Figura no tuviera constructor, Java crearía uno por defecto que no haría nada.

El método **toString( )** se hereda de Object y las clases lo suelen sobrecargarlo: dicho método se invoca automáticamente en los println y cuando Java necesita convertir un objeto en un String.

Si omitimos la línea **super(nx,ny)** en la clase Circulo, Java invocaría al constructor por defecto de la clase Base mediante **super( )**
Al no tener la clase base un constructor por defecto: ERROR

```
class Figura {  // class Figura extends Object { //clase de paquete
        public int x;
        public int y;
        public Figura ( int nx, int ny){
                x=nx;
                y=ny;
        }
        public String toString(){
                return new String( "("+x+","+y+")" );
               //return "("+x+","+y+")"; //también es válido
        }
}

class Circulo extends Figura { //clase de paquete
        public double radio;
        public Circulo (  double nx, double ny, double r ){

                super(nx, ny); //obligatorio: debe ser la 1ª linea
                radio = r;          //en el constructor de la clase derivada
        }
        public String toString(){
                return new String( "("+x+","+y+"):"+radio );
        }
}

public class Herencia{ //clase pública

        public static void main(String[] args){
                Figura p=new Figura (10.5,6.2);
                Circulo c=new Circulo(1,2,2.45); //Ojo, se copia la ref

                System.out.println( p +"\n"+ c );
        }
}
```

Nota: **super** se emplea para llamar al constructor de la clase padre. No es posible acceder a otra clase de la jerarquía (en C++ si es posible acceder a una clase "abuela").

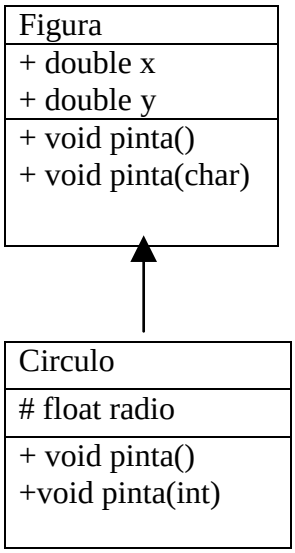
Si no se hubiera creado un constructor en la clase derivada Java generaría el siguiente:

```
public Circulo ( ) { super( ); } //constructor que invoca al constructor por defecto de la clase base
                                 //los atributos exclusivos de la clase derivada se inicializan a los valores por defecto
```

SOBRECARGA DE MÉTODOS HEREDADOS

Un método redefinido en la clase derivada oculta al de la clase Base
Un método sobrecargado en la clase derivada NO oculta al de la clase Base
(En C++ al redefinir o sobrecargar se ocultan todos los métodos con el mismo nombre de la clase Base)

Java siempre implementa las variables como referencias y los métodos como virtuales por lo que siempre se implementa por defecto el polimorfismo.

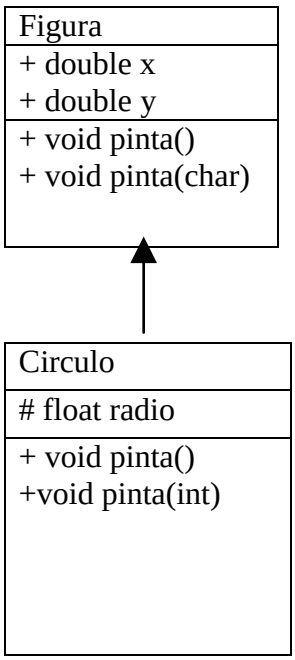



```
public class Figura {

        public double x;
        public double y;
        int dePaquete;
        public Figura( double nx, double ny){
                x=nx;
                y=ny;
        }
        public void pinta(){
                System.out.println( " Figura >> ("+x + ","+ y +") ");
        }
        public void pinta(char a){
             System.out.println( " Figura >> ("+ x +","+ y + ") char " + a );
           //pinta(); System.out.println(") char " + a );
        }
}
class Circulo extends Figura {
        public double radio;
        public Circulo(  double nx, double ny, double r ){
                super(nx, ny);
                radio = r;
        }
        public void pinta() {  //sobreescribo el pinta( ) del padre
              System.out.println( "Circulo >> ("+x + ","+ y +") :"+ radio);
        }
        public int pinta(int a) { //sobrecargo el pinta( ) del padre
        System.out.println( "Circulo >> ("+x + ","+ y +")" :+ radio +" int a " + a);
                return a;
        }
}
public class Herencia {
        public static void main(String[] args) {
                Figura p=new Figura (10.5,6.2);
                Circulo c=new Circulo(1,2,2.45);
                c.pinta();  //ejecuta el pinta de Circulo que oculta el pinta de Figura
                p.pinta();
                c.pinta(1); //ejecuta el pinta(int n) de Circulo
                c.pinta('a'); // método sobrecargado (ejecuta pinta(char a) de Figura
        }
}
```

Salida

```
Circulo >> (1.0,2.0):2.45
Figura >> (10.5,6.2)
Circulo >> (1.0,2.0):2.45 int a 1
Figura >> (1.0,2.0):2.45 char a
```

MÉTODOS HEREDADOS. ACCESO A LA SUPERCLASE




```
public class Figura{
        public double x;
        public double y;
        int dePaquete;
        public Figura ( double nx, double ny){
                x=nx;
                y=ny;
        }
        public void pinta(){
                System.out.println( " Figura >> " + this +"\n");
        }
        public void pinta(char a){
                System.out.println( " Figura >> " + this +" char " + a +"\n");
        }
}


class Circulo extends Figura {
        public double radio;
        public Circulo(  double nx, double ny, double r ){
                super(nx, ny);
                radio = r;
        }
        public void pinta() {
             System.out.println( "Circulo >> ("+x + ","+ y +") :"+ radio);
        }
        public int pinta(int a)     {
            System.out.print( "Circulo >> ("+x + ","+ y +")" :+ radio +" int a " + a);
            System.out.print( "y llama al  padre");

            super.pinta(); //no esta permitido desde fuera de derivada

            return a;
        }
}
public class Herencia{
        public static void main(String[] args){
                Figura p=new Figura (10.5,6.2);
                Circulo c=new Circulo(1,2,2.45);
                c.pinta();
                p.pinta();
                c.pinta(1);
                c.pinta('a');
        }
```

Nota : Es necesario llamar explícitamente al constructor de la superclase porque no existe Figura ();
En el main( ) no es posible invocar el pinta( ) de Figura con una ref Circulo (en C++ c-> Figura::pinta())

Salida

```
Circulo >> (1.0,2.0):2.45
Figura >> (10.5,6.2)
Circulo >> (1.0,2.0):2.45 int a 1 y llama al padre  Figura >> (1.0,2.0)
Figura >> (1.0,2.0):2.45 char a
```

MIEMBROS ESTÁTICOS (static)

Son compartidos por todos los objetos

Estatica
+ static int num - int otra
+Estatica() + static getNum () + void finalize()

```
public class Estatica {
   static int num;
   int otra ;
   static int getNum(){
         return num;
         //otra = 0; No se puede , no es static
   }

   public Estatica() {  num ++;  } //constructor

   public void finalize (){ //parecido a los destructores
         num--;
   }
   public static Estatica mayor(Estatico x, Estatico y) {
         if (x.otra > y.otra) return x;
         else return y;
   }
   public static void main(String[] args) {
         Estatica.num = 5; // Estatica::num = 5; en C++
         Estatica e= new Estatica();
         System.out.println(e.getNum());
         System.out.println(Estatica.getNum()); //mejor
   }
}
```

Las variables de clase (***static)*** se suelen utilizar para definir constantes comunes para todos los objetos de la clase (por ejemplo ***PI*** en la clase ***Math***) o variables que sólo tienen sentido para toda la clase (por ejemplo, un contador de objetos creados como ***numCirculos*** en la clase ***Circulo***).

Análogamente, puede también haber métodos que no actúen sobre objetos concretos sino sobre toda la clase (***métodos de clase*** o ***static)***. Los métodos de clase pueden recibir objetos de su clase como argumentos explícitos, pero no tienen argumento implícito ni pueden utilizar la referencia ***this***. Un ejemplo de métodos ***static*** son los métodos matemáticos de la clase ***java.lang.Math*** (*sin(), cos(), exp()..*)

Los métodos estáticos no pueden hacer referencia a atributos no estáticos de la clase, porque, al igual que en C++ se pueden invocar sin haber creado un objeto de dicha clase, pero si pueden hacer referencia a los atributos (estáticos o no) de los objetos pasados como parámetro

Para llamarlas se suele utilizar el nombre de la clase (se puede utilizar también el nombre de un objeto, si existe), ya que existen antes incluso de crearse un objeto de dicha clase.

INICIALIZADORES ESTÁTICOS (static)

Un ***inicializador static*** es un bloque {…} de código, precedido por la palabra ***static*** que se usa para inicializar atributos estáticos.
Se llama automáticamente una única vez al crear la clase (al utilizarla por primera vez).

Los atributos estáticos se inicializan cuando se carga la clase. A veces la inicialización es compleja.
Por ej, supongamos que necesitamos un vector estático que almacena las raíces cuadradas de los 100 primeros enteros.

Una posibilidad es proporcionar un método estático y obligar al programador a llamarlo antes de usar el vector.

Una alternativa mejor es usar un inicializador estático:

```
public class Raices {
   private static double raicesCuadradas [ ] = new double [100];

   static {
         for (int i=0; i<raicesCuadradas.length; i++)
            raicesCuadradas [ i]= Math.sqrt( (double) i);
   }

// resto de la clase

}
```

Para inicializar objetos o elementos más complicados es bueno utilizar un ***inicializador*** (un bloque de código {…}), ya que permite gestionar ***excepciones***[3] con ***try***…***catch***.

En una clase pueden definirse ***varios inicializadores static***, que se llamarán en el orden en que han sido definidos.

Resumen del proceso de creación de un objeto

El proceso de creación de objetos de una clase es el siguiente:
1. Al crear el 1er objeto de la clase o utilizar el 1er método o variable ***static*** se carga en memoria la clase
2. Se ejecutan los ***inicializadores static*** (sólo una vez).
3. Cada vez que se quiere crear un ***nuevo objeto***:

- se comienza reservando la memoria necesaria
- se da valor por defecto a las variables miembro de los tipos primitivos
- se ejecutan los constructores

CLASES Y MÉTODOS FINALES

Una ***clase*** declarada ***final*** no puede tener clases derivadas. Esto se puede hacer por motivos de ***seguridad*** y también por motivos de ***eficiencia***, porque cuando el compilador sabe que los métodos no van a ser redefinidos puede hacer optimizaciones adicionales.

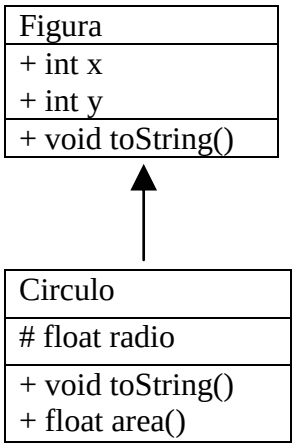
Análogamente, un ***método*** declarado como ***final*** no puede ser redefinido por una clase derivada, aunque si puede ser sobrecargado

```
public class Figura {
        public double x;
        public double y;
        int dePaquete;
        public Figura ( double nx, double ny){
                x=nx;
                y=ny;
        }
        final public void pinta(){ //metodo final, no puede sobreescribirse en la derivada
                System.out.println( " Figura >> " + this +"\n");
        }
}

final class Circulo extends Figura { //no se pueden crear clases que hereden de Circulo
        public double radio;
        public Circulo(  double nx, double ny, double r ){
                super(nx,ny);
                radio = r;
        }
        public void pinta() {  //ERROR pinta( ) es final en la clase base
             System.out.println( "Circulo >> ("+x + ","+ y +") :"+ radio);
        }
        public int pinta(int a)     { //si puedo sobrecargar un metodo final
             System.out.print( "Circulo >> ("+x + ","+ y +")" :+ radio +" int a " + a);
             System.out.print( "y llama al  padre");
             pinta();

             return a;
        }
}

class Circulo2 extends Circulo {  //ERROR no permitido porque Circulo es final
…
}

public class Herencia{
        public static void main(String[] args){
                Figura p=new Figura (10.5,6.2);
                Circulo c=new Circulo(1,2,2.45);
                c.pinta();
                p.pinta();
                c.pinta(1);
        }
}
```

CLASE ABSTRACTA

Clase que no se puede instanciar (En C++ la clase debe tener un método virtual puro, en Java **abstract**)
No es obligatorio que tenga métodos abstractos, pero si la clase tiene métodos abstractos → la clase debe ser **abstract**

Las clases abstractas se utilizan para definir la estructura general de una familia de clases derivadas. Se utilizan, como las interfaces, para imponer una funcionalidad común a un número de clases diferentes (que derivan de ella).



Una clase abstracta no se puede instanciar, pero una referencia de una clase abstracta puede apuntar a un objeto de una clase derivada

```
abstract class Figura{ //clase abstracta, no se puede instanciar

        public double x;
        public double y;

        public String toString() {
                return new String( "Figura("+x+","+y+")");
        }
}

class Circulo extends Figura {

        protected double radio;
        public Circulo(  double nx, double ny, double r ) {
                x= nx;
                y= ny;
                radio = r;
        }
        public String toString(){
                 return new String( "Circulo["+super.toString( )+"]"+"): "+radio );
        }
}

public class Abstract {

        public static void main(String[] args){
                Figura p;  // =new Figura ();//no se puede instanciar
                Circulo c=new Circulo(1,2,2.45);
                p=c;       //hago que p apunte a c
                System.out.println(p.toString());
                System.out.println(c );
                p=new Circulo(2,1,2.50);
                System.out.println(p );
        }
}
```

Aunque Figura es abstracta puede tener un constructor

```
public Figura( double nx, double ny) { x=nx; y=ny; }
```

si la Clase Figura hubiera tenido un constructor el Constructor de Circulo sería así:

```
public Circulo(  double nx, double ny, double r ) {
        super( nx,ny );
        radio = r;
}
```

Pantalla:

```
Circulo[Figura(1.0,2.0)]): 2.45
Circulo[Figura(1.0,2.0)]): 2.45
Circulo[Figura(2.0,1.0)]): 2.5
```

HERENCIA DE ABSTRACTA

Si no se implementa un método abstracto en una clase derivada Java muestra un error de compilación

sólo puede haber métodos abstractos en clases abstractas

```
abstract class Figura{

        public double x;
        public double y;

        public String toString(){
                return "";
        };
        public abstract void  pinta(); //método abstracto
                                //virtual void pinta( ) const=0 en C++
}
        The type Circulo must implement the inherited abstract method Figura.pinta()
class Circulo extends Figura {


        public double radio;
        public Circulo(  double nx, double ny, double r ){
                x= nx;
                y= ny;
                radio = r;
        }
        public String toString(){
                return new String( "Circulo("+x+","+y+"):"+radio );
        }
}
```

LA CLASE OBJECT (1) (java.lang.object)

La clase *Object* es la clase raíz de la cual derivan todas las clases. Esta derivación es implícita.
La clase *Object* define una serie de funciones miembro que heredan todas las clases. Destacan:

```
public class Object {
    public boolean equals(Object obj) {
        return (this == obj); //por defecto true si apuntan a lo mismo
    }
    protected native Object clone() throws CloneNotSupportedException;
    public String toString() {
        return getClass().getName() + "@" + Integer.toHexString(hashCode());
    }
   protected void finalize() throws Throwable { }
//otras funciones miembro...
}
```

Igualdad de dos objetos:
El método **equals** de la clase Object compara el objeto invocante con el objeto pasado como parámetro. El método se suele redefinir en las clases derivadas para comparar los objetos y no las referencias, es decir, para que devuelvan true si los valores de los atributos son iguales, aunque sean objetos distintos

Representación en forma de texto de un objeto:
El método **toString** imprime por defecto el nombre de la clase a la que pertenece el objeto y su código (hash). Este método se suele redefinir en las clases derivadas para mostrar la información que nos interese acerca del objeto. **toString** se llama automáticamente siempre que pongamos un objeto como argumento de la función System.out.println o concatenado con otro string.

Duplicación de objetos:
El método **clone** crea un objeto duplicado (clónico) de otro objeto. El método original heredado de ***Object devuelve un objeto idéntico*** haciendo una copia binaria de los valores de los atributos, de forma que los atributos de tipos primitivos copian sus valores y los atributos referencias también, haciendo que las referencias original y copia apunten al mismo objeto. Si se desea poder clonar una clase, dicha clase debe implementar la interface ***Cloneable*** y redefinir el método ***clone()***. Si la clase no implementa la interface ***Cloneable*** y hace una llamada al método ***clone()*** de Object se lanza la excepción una *CloneNotSupportedException*.

Finalización:
El método **finalize** se llama cuando va a ser liberada la memoria que ocupa el objeto por el recolector de basura (garbage collector). Normalmente, no es necesario redefinir este método en las clases, solamente en contados casos especiales. La forma en la que se redefine este método es el siguiente.

```
class CualquierClase{
        //..
        protected void finalize() trows Throwable{
                //código que libera recursos externos
                super.finalize();
        }
}
```

La última sentencia que contenga la redefinición de finalize ha de ser una llamada a la función del mismo nombre de la clase base. Previamente le añadimos cierta funcionalidad, habitualmente, la liberación de recursos, cerrar un archivo, etc.

LA CLASE OBJECT (2) (java.lang.object)

Métodos generales **de Object:**

`protected Object clone():`
Crea y devuelve una copia de este objeto. **Si se quiere implementar para clonar objetos, será necesario implementar el interfaz *Cloneable*.**

`boolean equals(Object obj):`
Verdadero cuando *obj* "es igual" a este objeto.

`protected void finalize():`
Es llamado por el recolector de basura cuando éste determina que no quedan referencias al objeto.

`Class<? extends Object> getClass():`
Devuelve la clase de un objeto en tiempo de ejecución.

`int hashCode():`
Devuelve el valor *hash* de un objeto. Único en una ejecución del programa java.

`String toString():`
Devuelve una cadena representación del objeto.

Punto implements Cloneable
+ double x + double y
+ Punto(double nx, double ny) + String toString() + boolean equals(Object o) + Object clone() throws CloneNotSupportedException

En realidad, se suele implementar con una llamada a **super.clone()**

```
import java.util.Scanner;
import java.util.Locale;
import java.math.*;

class Punto implements Cloneable{
        public double x;
        public double y;

        public Punto( double nx, double ny ){
                x = nx;
                y = ny;
        }

        public boolean equals( Object o ) {
          if ( this.getClass( )= =o.getClass( ) ) {//son misma clase
        // if (o instanceof Punto) {
                Punto otro = (Punto) o; //casting
                return (Math.abs(otro.x - x) < 1e-10)
                        &&(Math.abs(otro.y - y) < 1e-10);
          }
          else
              return false;
        }

        public Object clone() throws
                                CloneNotSupportedException{
                return (Object)(new Punto( x,y ));
        }
        public String toString(){
                return new String( "("+x+","+y+")" );
               // return "("+x+","+y+")" ;      //tambien valido
        }
}
```

OBJECT (3) (java.lang.object)

```
public class Programa {

	public static void main(String[] args) throws Exception {
		Scanner s = new Scanner( System.in );  // Lectura de un punto por teclado
		s.useLocale( Locale.ENGLISH );

		System.out.println( "Coordenadas del primer punto" );
		Punto p1 = new Punto( s.nextDouble(), s.nextDouble()); //1, 2
		System.out.println( "Coordenadas del segundo punto" );
		Punto p2 = new Punto( s.nextDouble(), s.nextDouble()); //1, 2

		System.out.println( p1 + (p1.equals(p2)?"iguales":"distintos") + p2);
		System.out.println( p1 + (p1==p2?"= =":"!=") + p2);

		Punto p3 = (Punto)p2.clone();
		System.out.println( p2 + (p2.equals(p3)?"iguales":"distintos") + p3);
		System.out.println( p2 + (p2==p3?"==":"!=") + p3);

		p3.x=0;  p3.y=0;
		System.out.println( p2 + (p2.equals(p3)?"iguales":"distintos") + p3);
		System.out.println( p2 + (p2==p3?"==":"!=") + p3);

		p3=p2; //hacemos que p3 apunte al mismo objeto al que apunto p2
		System.out.println( p2 + (p2.equals(p3)?"iguales":"distintos") + p3);
		System.out.println( p2 + (p2==p3?"==":"!=") + p3);

		Object obj = new Integer(4);
		System.out.println( p3 + (p3.getClass().equals(obj.getClass())?" misma":"
distinta")+" clase " + obj);

		obj = new Punto(10,10);
		System.out.println( p3 + (p3.getClass().equals(obj.getClass())?" misma":"
distinta")+" clase " + obj);
	}
}
```

Salida:

```
Coordenadas del primer punto
1
2
Coordenadas del segundo punto
1
2
(1.0,2.0)iguales(1.0,2.0)
(1.0,2.0)!=(1.0,2.0)
(1.0,2.0)iguales(1.0,2.0)
(1.0,2.0)!=(1.0,2.0)
(1.0,2.0)distintos(0.0,0.0)
(1.0,2.0)!=(0.0,0.0)
(1.0,2.0)iguales(1.0,2.0)
(1.0,2.0)==(1.0,2.0)
(1.0,2.0) distinta clase 4
(1.0,2.0) misma clase (10.0,10.0)
```

DUPLICACIÓN DE OBJETOS (http://www.sc.ehu.es/sbweb/fisica/cursoJava/fundamentos/clonico/clonico1.htm)

En algunas situaciones deseamos que un objeto que se pasa a un método o un objeto que llama a un método no se modifique en el curso de la llamada. En todos estos casos, puede ser muy útil realizar una copia del objeto original y realizar las trasformaciones en la copia dejando intacto el original, es decir, crear un clon (un clon de un objeto es un objeto con distinta identidad pero mismo contenido).

EL INTERFACE Cloneable

Para clonar una clase hay que implementar la interface ***Cloneable*** y sobrecargar el método *protected* ***clone()*** de ***Object*** declarándolo ***public***. El método original ***clone()*** de ***Object devuelve un objeto idéntico*** haciendo una copia binaria de los atributos (los atributos de tipos primitivos copian sus valores y los atributos referencias también, haciendo que las referencias original y copia apunten al mismo objeto) y lanza una excepción ***CloneNotSupportedException*** si la clase no implementa la interfaz ***Cloneable***.
La interface ***Cloneable*** es muy simple (no define ninguna función):

```
public interface Cloneable {  }
```

DUPLICACIÓN DE UN OBJETO SIMPLE

La clase base ***Object*** de todas las clases en el lenguaje Java, tiene una función miembro denominada ***clone()***, que se sobrecarga en la clase derivada para realizar una duplicación de un objeto de dicha clase.

Sea la clase Punto estudiada antes. Para hacer una copia de un objeto de esta clase, se ha de:
1) implementar el interface ***Cloneable***
2) redefinir la función miembro ***clone()*** de la clase base ***Object***

```
public class Punto implements Cloneable{
   private int x;
   private int y;
//constructores ...

   public Object clone() { //en Object clone() es protected pero al sobrecargarla lo hacemos public
      Object obj=null;        //Punto obj=null; → try { obj=(Punto)super.clone(); } catch (…)
      try{
         obj=super.clone();              //se llama al clone( ) de la clase base (en este caso Object)
      }catch (CloneNotSupportedException ex) {      //que hace copia binaria de los atributos x, y
         System.out.println(" no se puede duplicar");
      }
      return obj;
   }
   public String toString() { return origen+" ancho: "+ancho+" alto: "+alto;   }
 //otras funciones miembro
}
```

En la redefinición de ***clone()***, la llamada a ***super.clone( )*** se ha de hacer forzosamente dentro de un bloque **try... catch**, para capturar la excepción ***CloneNotSuportedException*** que nunca se producirá si la clase implementa el interface ***Cloneable***.

```
      Punto punto=new Punto(20, 30);
      Punto pCopia=(Punto)punto.clone();
```

La promoción (casting) es necesaria ya que ***clone()*** devuelve un objeto de la clase base ***Object*** que ha de ser promocionado a la clase ***Punto***.

DUPLICACIÓN DE UN OBJETO COMPUESTO (http://coderevisited.com/cloneable-interface-in-java/)

Una vez que obtenemos el clon del objeto con ***super.clone(),*** es posible que haya que hacer algunas modificaciones en función de la naturaleza de la clase: Si los atributos del objeto son de tipo primitivo o son una referencia a un objeto inmutable, no es necesario un tratamiento posterior. Si hay atributos que referencian objetos mutables, será necesario llamar al clon de esas referencias mutables, para que el clon no afecte al objeto original (sean totalmente independientes).

Cuando un objeto contiene atributos que son objetos de otra clase, en la redefinición de la función miembro ***clone()*** de la clase ***Contenedora*** se ha de efectuar una duplicación de dichos atributos subobjetos llamando a la versión ***clone()*** definida en dichos subobjetos (si dichos subobjetos no tienen el método ***clone()*** se llamará a su "constructor de copia" o a cualquier método que tenga que duplique su información). De no hacerlo, los atributos subobjetos apuntarán al mismo objeto en el original y la copia.

Si la clase es inmutable (no tiene métodos set) no importa que original y copia apunten al mismo objeto, pero si es mutable sí.

Ej: Un objeto de clase ***Rectangulo*** contiene un subobjeto de la clase mutable ***Punto***. En la redefinición de la función miembro ***clone()*** de la clase ***Rectangulo*** se ha de efectuar una duplicación de dicho subobjeto llamando a la versión ***clone()*** definida en la clase **Punto**.

Si Punto no fuera mutable no sería necesario ya que no importaría que original y copia apunten al mismo objeto.

Recuérdese que la llamada a ***clone()*** siempre devuelve un objeto de la clase base ***Object*** que ha de ser promocionado (**casting**) a la clase derivada adecuada.

```
public class Rectangulo implements Cloneable {
   private int ancho ;
   private int alto ;
   private Punto origen;
//los constructores
   public Object clone() { //en Object clone() es protected pero al sobrecargarla lo hacemos public
      Rectangulo obj=null;  //no ponemos Object obj=null porque entonces no podemos poner obj.origen ya que Object no tiene ese campo
      try{
         obj=(Rectangulo)super.clone();  //devuelve un objeto que tiene una copia binaria de los
       //obj.origen=(Punto)obj.origen.clone();          //atributos ancho, alto y origen
      }catch (CloneNotSupportedException ex) {        //obj.origen = = origen (apuntan a lo mismo)
         System.out.println(" no se puede duplicar");
      }
      obj.origen=(Punto)obj.origen.clone();  //devuelve un objeto Punto que tiene una copia binaria de x, y
    //obj.origen= new Punto(origen);           //si se omite entonces obj.origen==this.origen
      return obj;
   }
   public String toString() {
      String texto=origen+" ancho: "+ancho+" alto: "+alto;
      return texto;
   }
 //otras funciones miembro
}
```

En la redefinición de ***clone()***, la llamada a ***super.clone( )*** se ha de hacer forzosamente dentro de un bloque **try... catch**, para capturar la excepción ***CloneNotSuportedException*** que nunca se producirá si la clase implementa el interface ***Cloneable***. (**obj.origen=(Punto)obj.origen.clone();** también lo podemos meter dentro del try)

```
      Rectangulo rect=new Rectangulo(new Punto(0, 0), 4, 5);
      Rectangulo rCopia=(Rectangulo)rect.clone();
```

La promoción (**casting**) es necesaria ya que ***clone()***devuelve un objeto de la clase base ***Object*** que ha de ser promocionado a la clase ***Punto***.

DUPLICACIÓN DE UN OBJETO COMPUESTO EN CLASES NO FINALES Y HERENDADAS

(http://coderevisited.com/cloneable-interface-in-java/)

Cuando una clase es **final** (no puede tener clases hijas), la implementación del método ***clone()*** se puede realizar invocando "el constructor de copia" o cualquier otro método parecido.

Si la clase (simple o compuesta) tiene **constructor de copia o similar** el método **clone( )** es muy simple:

```
public class Rectangulo implements Cloneable {
        public Rectangulo(Rectangulo r) { …. }  //constructor de copia…
        public Object clone() { return new Rectangulo(this); } //este es el codigo del metodo clone()
}
```

Pero cuando una clase no es **final**, la implementación de ***clone()*** debería retornar un objeto obtenido invocando ***super.clone( )*** ya que ***hace una copia binaria de todos los atributos del objeto al que apunta*** Si todas las superclases de una clase (todos sus ascendientes en la jerarquía de clases) obedecen esta regla, la invocación de ***super.clone( )*** invocará al final el método ***clone( )*** de la clase ***Object*** creando una instancia de la clase (objeto) correcta.
Si no se respeta esta regla el método ***clone()*** en las clases hijas fallará al llamar al ***clone()*** del padre

```
public class Animal implements Cloneable {
   private String nombre;

   public Animal(String nombre) { this.nombre = nombre; }
   public String getNombre( ) { return nombre; }
   public String ladrar( ) { return "guau guau guau… asi es como ladro"; }

   public Animal clone() {
     //violacion del contrato de llamar a super.clone()
       return new Animal(nombre);
   }
}

public class Perro extends Animal {
   private String ladrido;
   public Perro(String nom, String lad) { super(nom); ladrido=lad; }
   public String ladrar () { return ladrido; }

   public Perro clone() {
      return (Perro) super.clone();
   }
}

public class CloneableTest {
   public static void main(String[] args) {
      Perro dog = new Perro ("Milú", "Bow Bow!!");
      Perro dogClone = dog.clone();   //Do you think, you can do this
      System.out.println(dogClone.getName());
  }
}
```

Si clone() de Animal fuera asi:

```
public Object clone( ) {
      try{
         return (Animal)super.clone();
      }catch (CloneNotSupportedException ex) {
         System.out.println(" no se puede duplicar");
      }
}
```

El programa no daría error.

Si lo dejamos como está para que no dé error tendríamos que implementar clone() de Perro sin invocar super.clone(), por ejemplo asi:

```
public Perro clone( ) {
      return new Perro(this.getNombre(), this.ladrido);
}
```

pero obligaría a que la clase padre (Animal) tuviera un constructor de copia:

```
public Animal(Animal a) {
      this.nombre = a.nombre;
}

public Perro clone( ) {
      return new Perro(this.getNombre(), this.ladrido);
```

Al ejecutar el main() se produce un error en el casting a Perro ya que super.clone() devuelve un Animal:

```
Exception in thread "main" java.lang.ClassCastException: Animal cannot be cast to Perro
        at Perro.clone(Perro.java:6)
        at CloneableTest.main(CloneableTest.java:4)
```

EJEMPLO COMPLETO DE DUPLICACIÓN DE OBJETOS COMPUESTOS Y HEREDADOS:

```
package clonico;

public final class Punto implements Cloneable {
   private int x;      public int getX() { return x; }
   private int y;      public int getY() { return y; }
   public Punto(int x, int y)  {  this.x = x;   this.y = y;    }
   public Punto() {  x=0;   y=0;   }
   public Object clone() {
      Object obj=null;
      try{
         obj=super.clone();
      }catch(CloneNotSupportedException ex){
         System.out.println(" no se puede duplicar");
      }
      return obj;
   }
   public void trasladar(int dx, int dy) { x+=dx;  y+=dy;  }
   public String toString(){
      String texto="raiz: ("+x+", "+y+")";
      return texto;
   }
}
```

```
package clonico;

public class Rectangulo implements Cloneable {
   private int ancho ;
   private int alto ;
   private Punto ini;
   private String nombre;

   public Rectangulo() {
      this(new Punto(0, 0), 0, 0, "anonimo");
   }
   public Rectangulo(Punto p) { this(p, 0, 0, "anonimo"); }
   public Rectangulo(Punto p, int w, int h, String nom) {
      ini= new Punto(p.getX(), p.getY()); //ini=p.clone( );
      ancho = w;    alto = h;    nombre=nom;
   }
   public Object clone() {
      Rectangulo obj=null;
      try{
         obj=(Rectangulo)super.clone();
         obj.ini=(Punto)this.ini.clone();
      }catch(CloneNotSupportedException ex){
         System.out.println(" no se puede duplicar");
      }
      return obj;
   }
   public void mover(int dx, int dy, int w, int h, String s) {
       ini.trasladar(dx, dy); ancho=w; alto=h; nombre=s;
   }
   public int area() {    return ancho * alto;    }
   public String toString(){
      String texto=ini+" "+ancho+"x"+alto+" "+nombre;
      return texto;
   }
}
```

Si Punto no implementa Cloneable
➔ Al no tener clone()
obj.ini=
new Punto(ini.getX(), ini.getY());

Salida: Si se omite lo gris (copia y original apuntan a mismos objetos Punto (mutable) y String (inmutable)

```
punto raiz: (30, 35)
copia raiz: (20, 30)
rectang raiz: (33, 38) 2x4 cambio
copia raiz: (33, 38) 4x5 UNO
hija raiz:(31, 36) 3x6 TRES-raiz: (21, 31) MIL 8
copia raiz: (31, 36) 4x5 DOS-raiz: (21, 31) PI 5
```

```
package clonico;

public class Hija extends Rectangulo {
   private Punto otro;
   private String cadena;
   private int n;

   public Hija( ) {
      super( ); otro= new Punto(0, 0); cadena="nada"; n=0;
   }
   public Hija(Punto p1, int w, int h, String nom,
               Punto p2, String cad, int n) {
      super(p1, w, h, nom);
      otro= new Punto(p2.getX(), p2.getY()); //otro=p2.clone( );
      cadena=cad; this.n=n;
   }
   public Object clone() {
      Hija obj=null;
         obj=(Hija)super.clone();
         obj.otro=(Punto)this.otro.clone();
      return obj;
   }
   public void cambiar(int dx, int dy, int w, int h, String nom,
                       String s, int i) {
         mover(dx, dy, w, h, nom);
         otro.trasladar(dx, dy); cadena=s; n=i;
   }
   public String toString() {
      String s=super.toString()+" - "+otro+" "+cadena+" "+ n;
      return s;
   }
}
```

```
package clonico;

public class ClonicoApp {
   public static void main(String[] args) {
      final Punto punto=new Punto(20, 30);
      Punto pCopia=(Punto)punto.clone();
      punto.trasladar(10, 5);
      System.out.println("punto "+ punto+"\ncopia "+ pCopia);

      Rectangulo rect=new Rectangulo(punto, 4, 5, "UNO");
      Rectangulo rCopia=(Rectangulo)rect.clone();
      rect.mover(3, 3, 2, 4, "cambio");
      System.out.println("rectang "+ rect+"\ncopia "+rCopia);

      Hija hij=new Hija (punto, 4, 5, "DOS", pCopia, "PI",5);
      Hija hCopia=(Hija)hij.clone();
      hij.cambiar(1,1,3,6, "TRES", "MIL",8);
      System.out.println("hija "+ hij+"\ncopia "+hCopia);
      try  {          //espera a que pulse una tecla + INTRO
         System.in.read();
      }catch (Exception e) {  }
   }
}
```

Salida: con lo gris (copia y original apuntan a objetos distintos)

```
punto raiz: (30, 35)
copia raiz: (20, 30)
rectang raiz: (33, 38) 2x4 cambio
copia raiz: (30, 35) 4x5 UNO
hija raiz: (31, 36) 3x6 TRES-raiz: (21, 31) MIL 8
copia raiz: (30, 35) 4x5 DOS-raiz: (20, 30) PI 5
```

DUPLICACIÓN DE OBJETOS: CONSIDERACIONES A TENER EN CUENTA

A guide to object cloning in java
http://howtodoinjava.com/2012/11/08/a-guide-to-object-cloning-in-java/

1) Si hay un atributo final que referencia un objeto mutable (lo que es constante es la referencia y no el objeto, es decir el objeto al que apunta la referencia puede ser modificado con los métodos set que tenga la clase y lo que no se puede cambiar es el objeto al que apunta la referencia) para que la clase pueda implementar correctamente la interfaz Cloneable hay que eliminar dicho modificador final.

   Si la clase Rectangulo anterior tuviera el atributo ***ini*** declarado contante

```
private final Punto ini;    //la clase Punto es mutable y el atributo ini que lo referencia es final
```

   entonces la implementación del método clone() fallaría en la línea marcada en gris:

```
public Object clone() {
      Rectangulo obj=null;
      try{
         obj=(Rectangulo)super.clone();  //hace copia binaria de todo (las referencias apuntan al mismo del original)
         obj.ini=(Punto)this.ini.clone();   //al ser ini final no podemos cambiar el lugar al que ya apunta tras ejecutar
      }catch(CloneNotSupportedException ex){                                          //la línea anterior
         System.out.println(" no se puede duplicar");
      }
      return obj;
 }
```

2) Cuando se clona un objeto con ***clone()*** no se invoca ningun constructor, por lo que hay que asegurar que todos los atributos se han establecido correctamente:

   - si el programa lleva la cuenta del número de objetos que se crean incrementando un contador de objetos en el/los constructor/es, en el método ***clone()*** sobrecargado también tendremos que incrementar dicho contador.
   - si el programa asigna a cada objeto un código único de forma automática en el contructor, entonces en el método ***clone()*** sobrecargado también habría que hacerlo (a menos que queramos que el clon sea exactamente igual al original incluso en el código… dependiendo del enunciado y restricciones del problema habría que hacerlo o no)

```
public final class Punto implements Cloneable {
   private int x,y, codigo;
   static int n = 0;
   public Punto(int x, int y)  {  this.x = x;   this.y = y;  n++;  codigo=n; }
   public Punto() {  this(0,0);  }
   public Object clone() {
      Object obj=null;
      try{
         obj=super.clone();   //hace copia binaria: la n no se incrementa y obj.codigo = = this.codigo
         n++;  codigo=n;       //si quiero que se haga lo mismo que en constructor
      }catch(CloneNotSupportedException ex){
         System.out.println(" no se puede duplicar");
      }
      return obj;
   }
}
```

3) Si no sabemos si una clase implementa la interfaz ***Cloneable*** (por tanto no sabemos si tiene o no un método ***clone()***) podemos averiguarlo de la siguiente manera:

```
if(obj1 instanceof Cloneable){   //true si es una instancia de la interfaz Cloneable
     obj2 = obj1.clone();
}
```

DUPLICACIÓN DE OBJETOS: ALTERNATIVAS AL USO DEL CLONE()

A veces la implementación correcta de la interfaz ***Cloneable*** no es posible hacerla siguiendo el esquema anterior. En esos casos **una alternativa es implementar constructores de copia** en todas las clases y llamarlos directamente o bien llamar a los constructores de copia desde dentro del método ***clone()***:

```
package clonico;

public final class Mutable implements Cloneable {
   int a,b;
   Mutable(int aa, int bb) { a=aa; b=bb; }
   Mutable(Mutable m) { a=m.a; b=m.b; }
   public Object clone() {
      return new Mutable(this);
   /* try {   // ESTO TAMBIEN SERIA VALIDO
         return super.clone(); //hace copia binaria
      } catch (CloneNotSupportedException e) {
         System.out.println(" no se puede duplicar");
      }
      return null;     */
   }
   void set(int aa, int bb) { a=aa; b=bb; }
   public String toString() { return "("+a+","+b+")"; }
}
```

```
package clonico;

public class Base implements Cloneable {
   private int x;
   private final int codigo;
   private final Mutable m;
   static int n = 0;
   public Base(int xx, Mutable mt)  {
      x = xx;   n++;   codigo=n; m=new Mutable(mt);
   }
   public Base(Base b)  {
      x = b.x;   n++;   codigo=n; m=new Mutable(b.m);
   }
   void set(int xx, int a, int b) { x=xx; m.set(a,b); }
   public Object clone() {
      return new Base(this);
      /*    ESTO NO SERIA VALIDO: falla en ERROR
      Object obj=null;
      try{
         obj=super.clone(); n++;  //hace copia binaria
       //codigo=n;                  //ERROR
       //m=new Mutable(m);  //ERROR
      }catch(CloneNotSupportedException ex){
         System.out.println(" no se puede duplicar");
      }
      return obj;
      */
   }
   public String toString(){
      return ""+x+" "+codigo+" "+m;
   }
}
```

Salida:

```
(4,4) (0,0)
8 1 (3,3)
5 2 (1,1)
9 3 (5,5) (8,8)
6 4 (4,4) (0,0)
```

```
package clonico;

public class Hija extends Base {
   private final Mutable mh;
   public Hija(int xx, Mutable m1, Mutable m2)  {
      super(xx, m1);     mh=new Mutable(m2);
   }
   public Hija(Hija h)  { super(h); mh=new Mutable(h.mh); }
   public Object clone() {
      return new Hija(this);
      /*   ESTO NO SERIA VALIDO: falla en ERROR
      Object obj=super.clone();  //hace copia binari
     //mh=new Mutable(mh);  //ERROR
      return obj;
      */
   }
   void set(int xx, int a, int b, int c, int d) {
        super.set(xx, a, b); mh.set(c,d);
   }
   public String toString(){
      return super.toString()+" "+mh;
   }
}
```

Si hago **constructores de copia** no hace falta que implemente ***Cloneable***
(lo normal es hacer uno u otro pero no ambos)

```
package clonico;

public class ClonicoApp {
   public static void main(String[] args) {
      final Mutable m1=new Mutable(0,0), xm1;
      final Mutable m2=new Mutable(m1), xm2;
      m1.set(1,1); //OK final es la referencia, no el objeto
     //m1=new Mutable(1,1); //ERROR m1 es final
      Base b1=new Base(5, m1), b2=new Base(b1);
      b1.set(8,3,3); m1.set(4,4);
      Hija h1=new Hija(6, m1, m2), h2=new Hija(h1);
      h1.set(9,5,5,8,8);
      System.out.println(m1+"
"+m2+"\n"+b1+"\n"+b2+"\n"+h1+"\n"+h2);

      xm1=new Mutable(0,0);  xm2=(Mutable) xm1.clone();
      xm1.set(1,1); //OK final es la referencia, no el objeto
     //xm1=new Mutable(1,1); //ERROR m1 es final
      Base xb1=new Base(5, xm1), xb2=(Base) xb1.clone();
      xb1.set(8,3,3); xm1.set(4,4);
     Hija xh1=new Hija(6, xm1, xm2), xh2=(Hija) xh1.clone();
      xh1.set(9,5,5,8,8);
      System.out.println(xm1+"
"+xm2+"\n"+xb1+"\n"+xb2+"\n"+xh1+"\n"+xh2);
   }
}
```

Salida: (continuación)

```
(4,4) (0,0)
8 5 (3,3)
5 6 (1,1)
9 7 (5,5) (8,8)
6 8 (4,4) (0,0)
```

INTERFACES:

Las interfaces se utilizan para imponer una funcionalidad común en las clases que las implementan.

- Los interfaces son **una agrupación de métodos y constantes públicas**.
- Las **clases pueden "comprometerse" a implementar interfaces** (uno o varias).
- Con los interfaces se **puede suplir la carencia de herencia múltiple de Java**. Basta con definir comportamientos con interfaces y luego hacer que las clases implementen todos los interfaces que el programador desee. La **relación ya no es de especialización** ("es un") **sino** de ***implementa***.
- Los interfaces **sólo contienen los prototipos de los métodos** (no se pueden definir). Los **métodos** de un interfaz son **implícitamente *public abstract***. No es necesario explicitarlo.
- **Las constantes** que se definen en un interfaz son **implícitamente *public static final***. No es necesario explicitarlo.
- **Los interfaces**, como las clases, **pueden ser públicos o de paquete**. **Si** son **públicos**, deben estar **en un archivo con el mismo nombre que la interface**.
- Los **interfaces también heredan** (extienden) **entre ellos**. Cuando una interface deriva de otra, incluye todas sus constantes y declaraciones de métodos.

```
interface SubInt extends Int1 { … }
```

- Y éstos **sí pueden heredar de varios superinterfaces**. **Existen casos en los que hay ambigüedad**.

```
interface IntMul extends Int1, Int2{ … }
```

- Se pueden **declarar variables** dando como **clase** una **interface**, es decir, de cara al polimorfismo, el nombre de una interface se puede utilizar como un nuevo tipo de referencia.

```
{ …
   IntMul a = new <clase que implementa IntMul>;
…}
```

  En este sentido, el nombre de una interface puede ser utilizado en lugar del nombre de cualquier clase que la implemente, aunque sólo podrá usar los métodos de la interface.
  Una interface puede ser utilizada como valor de retorno o como argumento de un método

- **Al poder declarar una variable como de un interfaz**, en realidad **expresamos que esa variable referenciará a un objeto que "se compromete" a implementar determinadas funcionalidades** (los métodos y constantes del interfaz correspondiente).
- Existen muchos **interfaces predefinidos** en la jerarquía de clases estándar.

Diferencia entre ***interface*** y una ***clase abstract:***

1. Una clase ***abstract*** puede definir los métodos que tiene, una interfaz no.

2. Una clase no puede heredar de dos clases ***abstract***, pero sí puede heredar de una clase ***abstract*** e implementar una ***interface***, o bien implementar dos o más ***interfaces***.

3. Las ***interfaces*** permiten mucha más flexibilidad para conseguir que dos clases tengan el mismo comportamiento, aunque no desciendan una de otra, es decir, independientemente de su situación en la jerarquía de clases de ***Java***.

4. Las ***interfaces*** tienen una ***jerarquía*** propia, independiente y más flexible que la de las clases, ya que tienen permitida la ***herencia múltiple***.

EJEMPLO COMPLETO DE INTERFACE:

Crear una interface denominado Parlanchin que contenga la declaración de un método habla( ).
Crear una jerarquía de clases que deriva de *Animal* que implemente el interface *Parlanchin*
Crear otra jerarquía de clases completamente distinta, la que deriva de la clase base *Reloj* y hacer que una de las clases de dicha jerarquía *Cucu* implementa el interface *Parlanchin* (por tanto, debe de definir obligatoriamente la función *habla* declarada en dicho interface)

Definir la función *hazleHablar* de modo que conozca al objeto que se le pasa no por una clase base, sino por el interface *Parlanchin*. A dicha función le podemos pasar cualquier objeto que implemente el interface *Parlanchin*, este o no en la misma jerarquía de clases.

```
package polimorfismo1;

public interface Parlanchin {
   public abstract void habla();
}
```

```
package polimorfismo1;

public abstract class Animal implements Parlanchin{
   public abstract void habla();
   public void saludo( ) { System.out.println("HOLA"); }
}

class Perro extends Animal{
   public void habla(){
      System.out.println("¡Guau!");
   }
   public void muerde( ) { … }  //BOCADO
}

class Gato extends Animal{
   public void habla(){
      System.out.println("¡Miau!");
   }
   public void araña( ) { … }
}
```

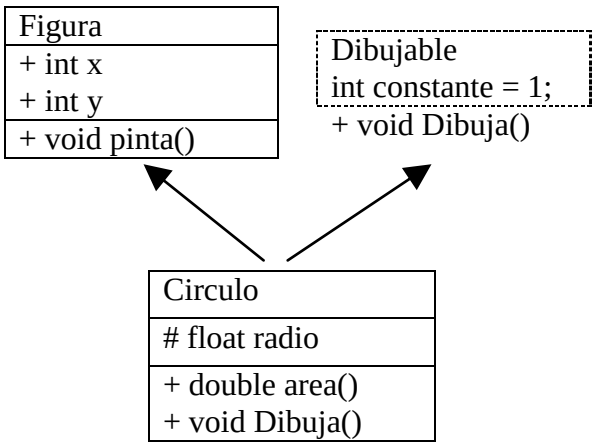
```
package polimorfismo1;

public abstract class Reloj {
 …
}

class Cucu extends Reloj implements Parlanchin{
   public void habla(){
      System.out.println("¡Cucu, cucu, ..!");
   }
   public void hora(){ … }  //las 12h

}
```

```
package polimorfismo1;
public class PoliApp {

 public static void main(String[] args) {
  Gato gato=new Gato();
  hazleHablar(gato); //Miau
  Cucu cucu=new Cucu();
  hazleHablar(cucu); //Cucu

  Parlanchin p;
  p=cucu;
  p.habla();  //Cucu
  if (p instanceof Cucu) {
     Cucu c=(Cucu) p;
     c.hora(); //las 12h
  }

  p=new Perro();
  p.habla(); //Guau
  //p.muerde( ); //ERROR p solo puede invocar habla( )
  if (p instanceof Perro) {
     ((Perro)p).muerde();//BOCADO
  }

  Animal a=(Animal)p;
  a.habla( ); //Guau
  a.saludo( ); //HOLA
  ((Perro)a).muerde( ); //Guau
  a=gato;
  a.habla( ); //Miau
  Habla(gato); //Miau y HOLA
  //Habla(cucu); //ERROR Habla solo admite Animal
 }

 static void hazleHablar(Parlanchin sujeto) { sujeto.habla();  }
 static void Habla(Animal a) {
    a.habla();
    a.saludo();
 }
}
```

```
¡Miau!
¡Cucu, cucu, ..!
¡Cucu, cucu, ..!
las 12h
¡Guau!
BOCADO
¡Guau!
HOLA
BOCADO
¡Miau!
¡Miau!
HOLA
```

Si solamente hubiese herencia simple, *Cucu* tendría que derivar de la clase *Animal* (lo que no es lógico) o bien no se podría pasar a la función *hazleHablar*. Con interfaces, cualquier clase en cualquier familia puede implementar el interface *Parlanchin*, y se podrá pasar un objeto de dicha clase a la función *hazleHablar*. Esta es la razón por la cual los interfaces proporcionan más polimorfismo que el que se puede obtener de una simple jerarquía de clases

INTERFACES VS HERENCIA MÚLTIPLE:




```
package Interfaces;
 class Figura{

        public double x;
        public double y;

        public void pinta(){
                System.out.print( "("+x+","+y+")");
        }
}
interface Dibujable{
        void dibuja();
        final double PI = 3.1416;
}
class Circulo extends Figura implements Dibujable{
        public double radio;
        public Circulo(  double nx, double ny, double r ){
                x= nx;
                y= ny;
                radio = r;
        }
        public void dibuja(){
                System.out.print("Dibuja>>");
           pinta();
        }
        Public double area (){
                return radio * Dibujable.PI;
        }

        public static void main(String[] args){

                Circulo c=new Circulo(1,2,2.45);
                Dibujable d;
                c.pinta();
                c.dibuja( );
                d= c;
                d.dibuja();
                //d.area(); //ERROR
                System.out.println("area : " + c.area());
                System.out.println(Dibujable.PI);
        }
}
```

Agrupación de Constantes

```
public interface Meses {
   int ENERO = 1 , FEBRERO = 2 . . . ;
   String [] NOMBRES_MESES = { " " , "Enero" , "Febrero" , . . . };
}

System.out.println(Meses.NOMBRES_MESES[Meses.ENERO]);
```

DESTRUCTORES:

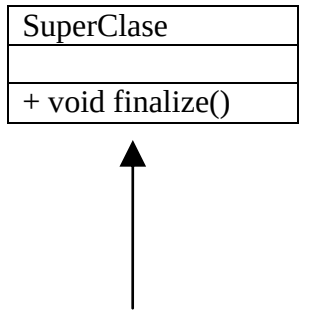
- **En Java no existen los destructores como tales**. El sistema se ocupa automáticamente de liberar la memoria de los objetos que no son apuntados por ninguna referencia. Internamente Java lleva un contador de cuántas referencias hay sobre cada objeto. El objeto podrá ser borrado cuando el número de referencias sea cero.

- Existe un **método definido en** la clase **Object** que es **llamado por el recolector de basura cuando** se va a destruir un objeto. El método **public void finalize()**.Se utilizan para ciertas operaciones de terminación distintas de liberar memoria (ej: cerrar ficheros).

- En Java no se sabe exactamente cuándo se va a activar el **garbage collector**. Si no falta memoria es posible que no se llegue a activar en ningún momento**. Es posible llamar explícitamente al recolector de basura: System.gc();**

- **Nota:** aunque podemos **llamar explícitamente al recolector de basura**, esta **no es una práctica recomendada en Java**. El recolector de basura está pensado para funcionar por su cuenta y liberar al programador de la tarea de devolver la memoria al sistema.

- El método **finalize no tiene por qué llamarse en cascada**. En C++, los destructores se van llamando automáticamente desde la subclase hacia las superclases. Esto es diferente en Java: sólo se llama al finalize( ) de la clase que se elimina. Por tanto, para realizar su tarea correctamente, un finalizador debería terminar siempre llamando al finalizador de su super-clase.



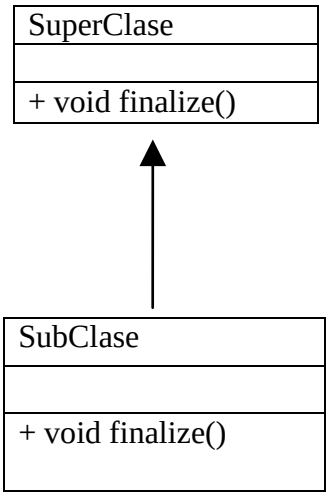

```
package principal;

class SuperClase{

        public void finalize(){
                System.out.println("Destruyendo! Super");
        }
}

class SubClase extends SuperClase{

        public void finalize(){
                System.out.println("Destruyendo!Sub");
        }
}

public class Programa {

        public static void main(String[] args){

                SubClase c = new SubClase();

                c = null; // El objeto queda sin referencias

                System.gc();
        }
}
```

Salida:

Destruyendo!Sub

DESTRUCTORES:

Es **posible** hacer la **llamada en cascada explícitamente**, pero podemos hacerlo **en el orden que queramos**:




```
package principal;

class SuperClase{

        public void finalize(){
                System.out.println("Destruyendo! Super");
        }
}

class SubClase extends SuperClase{

        public void finalize(){
                System.out.println("Destruyendo!Sub");
                super.finalize(); // Como en C++
        }
}

public class Programa {

        public static void main(String[] args){

                SubClase c = new SubClase();
                c = null;
                System.gc();
        }
}
```

Salida:

Destruyendo!Sub
Destruyendo!Super

```
package principal;

class SuperClase{

        public void finalize(){
                System.out.println("Destruyendo! Super");
        }
}

class SubClase extends SuperClase{

        public void finalize(){
                super.finalize(); // Al contrario que C++
                System.out.println("Destruyendo!Sub");
        }
}

public class Programa {

        public static void main(String[] args){

                SubClase c = new SubClase();
                c = null;
                System.gc();
        }
}
```

Salida:

Destruyendo!Super
Destruyendo!Sub

INFORMACIÓN DE CLASE EN TIEMPO DE EJECUCIÓN: instanceof

Es posible hacer casting "hacia abajo" (hacia abajo en la jerarquía de clases).

instanceof

Java comprueba si el casting es válido en tiempo de ejecución.

```
Salida:

(0.0,0.0)
```

```
public class Programa {

        public static void main(String[] args){

                /*
                 * El casting "hacia abajo" es útil y
                 * válido pero es potencialmente peligroso
                 */
                Figura p;
                Circulo c;
                /* ¡¡Caso correcto!! */
                p = new Circulo(0,0,10);
                c = (Circulo)p;
                System.out.println( c );
        }

}
```

Solución: asegurarnos de que hacemos un casting válido

Java comprueba si el casting es válido en tiempo de ejecución.

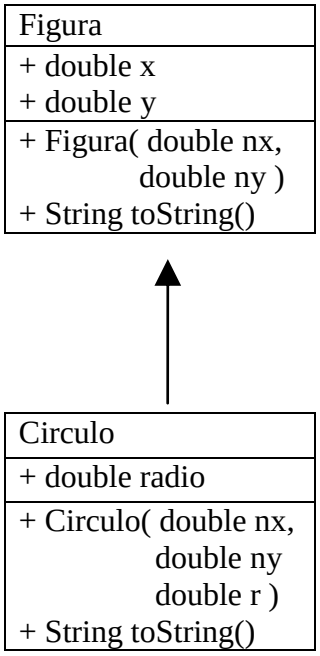
```
public class Programa {

        public static void main(String[] args){
                /*
                 * El casting "hacia abajo" es útil y
                 * válido pero es potencialmente peligroso
                 */
                Figura p;
                Circulo c;
                /* ¡¡Caso correcto!! */
                p = new Circulo(0,0,10);

               //if (Circulo.class.isInstance(p)) {
                if ( p instanceof Circulo ){
                        c = (Circulo)p;
                        System.out.println( c );
                }
        }

}
```

p **instanceof** Circulo **significa lo mismo que** Circulo**.class.isInstance(**p**)**

¿p pertenece a la clase Circulo **o a una clase derivada de Circulo**?

INFORMACIÓN DE CLASE EN TIEMPO DE EJECUCIÓN: Class

Es posible conocer la clase a la que pertenece un objeto en tiempo de ejecución e incluso tener información de si algo es clase o subclase. Estas funcionalidades las provee la clase **Class** de la jerarquía estándar.



Pantalla:

```
p1: class Figura
p2: class Circulo
c: [(0.0,0.0)]): 1.0

p2 es de la misma
clase que c o
subclase de la clase
de c

c y p1 son de la
misma clase
```

```
package principal;

class Figura {
        public double x;
        public double y;
        public Figura( double nx, double ny){
                x=nx;
                y=ny;
        }
        public String toString() {  return new String( "("+x+","+y+")" ); }
}

class Circulo extends Figura {
        public double radio;
        public Circulo( double nx, double ny, double nr){
                super( nx,ny );
                radio = nr;
        }
        public String toString(){
                return new String( "["+super.toString( )+"]"+"): "+radio );
        }
}

public class Programa {
    public static void main(String[] args){
        //El casting "hacia abajo" es peligroso por lo que hay que asegurarse

        Figura p1= new Figura(1,2), p2=new Circulo(0,0,1);
        Circulo c=null;  //las locales es mejor inicializarlas a null

        System.out.println("p1: "  + p1.getClass( )+"\np2: "
                                   + p2. getClass( ) );
        if (p1 instanceof Circulo) //true si p1 es un (o subclase de) Circulo
                c=(Circulo) p1;
        else if (p2 instanceof Circulo) //mejor (p2.getClass() == Circulo.class)
                c=(Circulo) p2;

        System.out.println("c: " + c);

        p1=c;     //no hace falta casting ya que p1 es superclase de c

        if ( c.getClass().isInstance( p2 ) ){
                System.out.println( "p2 es de la misma clase que c o "+
                                    "subclase de la clase de c");
        }
        if (  c.getClass( ) == p1.getClass( ) ) //MEJOR
                                System.out.println("c y p1 son de la misma clase");
    }
}
```

p **instanceof** Circulo significa lo mismo que Circulo**.class.isInstance(**p**)**	¿p es de la clase Circulo o subclase de Circulo?
p.getClass() == Circulo.class significa lo mismo que p.getClass() .equals(Circulo.class)	¿p es de la clase Circulo?
c.getClass().isInstance(p)	¿p es de la misma clase o subclase de la clase de c?
c.getClass( ) == p.getClass( )	¿p y c son de la misma clase?

CLASES DE UTILIDAD

Programando en ***Java*** nunca se parte de cero: siempre se parte de la infraestructura definida por el API de ***Java***, cuyos ***packages*** proporcionan una buena base para que el programador haga sus aplicaciones.

ARRAYS

Los ***arrays* de *Java* se tratan como objetos** de una clase predefinida.

1. Los array tienen un atributo ***length*** que indica el número de elementos de un array (ej ***vect.length***).

2. Se pueden crear ***arrays*** de objetos de cualquier tipo. En principio un ***array*** de objetos es un ***array de referencias*** que hay que completar llamando al operador ***new***.

3. Los elementos de un ***array*** se inicializan al valor por defecto del tipo correspondiente (0, false o null).

4. A diferencia de C++, Java realiza chequeo de los límites de un array. Ej: vect[vect.length+4] ERROR

INICIALIZACIÓN DE ARRAYS:

1. Los ***arrays*** se pueden inicializar con valores entre llaves {...} separados por comas.

2. Los ***arrays de objetos*** se pueden inicializar con varias llamadas a ***new*** dentro de llaves {...}.

3. Si se igualan dos referencias a un array no se copia el array, sino que se tiene un array con dos nombres, apuntando al mismo y único objeto.

4. Creación de una ***referencia*** a un array. Son posibles dos formas:

```
double[] x; // preferible //aun no se ha creado el array sino solo se ha declarado
double x[];               //una referencia a un array
```

5. Creación del ***array*** con el operador ***new***:

```
x = new double[100];      //ahora se ha creado el array con 100 double incializados a 0
```

6. Las dos etapas 4 y 5 se pueden unir en una sola:

```
double[] x = new double[100];
```

Ejemplos de creación de arrays**:**

```
// crear un array de 10 enteros, que por defecto se inicializan a cero
int x[] = new int[10];
int y[];
y=x;  //x e y apuntan al mismo array

// crear arrays inicializando con determinados valores
int v[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
String dias[] = {"lunes", "martes", "miercoles", "jueves", "viernes", "sabado", "domingo"};

// array de 5 objetos
MiClase listaObj[] = new MiClase[5]; // de momento hay 5 referencias a null
for( i = 0 ; i < 5;i++)
  listaObj[i] = new MiClase(...);

// array anónimo
obj.metodo(new String[]={"uno", "dos", "tres"});
```

EXPANSIÓN DINÁMICA DE VECTORES:

```
int [ ] a = new int [5]; //si una vez creado un vector, posteriormente deseamos ampliarlo
int [ ] aux =a;
a = new int [7];   //es buena idea ampliarlo duplicando su tamaño en vez de ampliarlo poco
for(int i = 0 ; i < aux.length; i++)  //la operacion es costosa porque hay que copiar todo
  a[i]=aux[i];
```

Es aconsejable expandir un vector multiplicando su tamaño por una constante (al doble es una buena decisión de diseño). Si un vector tiene tamaño N debemos ampliarlo a 2N y así sucesivamente.

ARRAYS BIDIMENSIONALES

Los arrays bidimensionales de ***Java*** se crean de un modo muy similar al de C++.
En ***Java*** una ***matriz*** es un ***vector*** de ***vectores fila***, o más en concreto un vector de referencias a los vectores fila. Con este esquema, cada fila podría tener un número de elementos diferente.

Una matriz se puede crear directamente en la forma,

```
int [][] mat = new int[3][4];
```

o bien se puede crear de modo dinámico dando los siguientes pasos:

1. Crear la ***referencia*** indicando con un doble corchete que es una ***referencia a matriz***,

```
int[][] mat;
```

2. Crear el vector de referencias a las filas,

```
mat = new int[nfilas][];
```

3. Reservar memoria para los vectores correspondientes a las filas,

```
for (int i=0; i<nfilas; i++);
  mat[i] = new int[ncols];
```

A continuación se presentan algunos ejemplos de creación de arrays bidimensionales:

```
double mat[][] = new double[3][3]; //crea una matriz 3x3 que se inicializan a cero

int [][] b = {{1, 2, 3},
              {4, 5, 6}, // esta coma es permitida
             };

//crea una matriz de 3 filas, cuya fila 0 tiene 5 col, fila 1 4 col y fila 2 8 col
//rellenarlo con los 10 primeros números naturales y mostrarlo en pantalla
int [][] t = new int[3][]; // se crea el array de referencias a arrays
t[0] = new int[5];
t[1] = new int[4];
t[2] = new int[8];
int n=0;                                                    Salida:
for(int i=0; i<t.length; i++) {                             0 1 2 3 4
  for(int j=0; j<t[i].length; j++) {                        5 6 7 8
    t[i][j]=(n++)%10;                                       9 0 1 2 3 4 5 6
    System.out.print(t[i][j]+ " ");
  }
  System.out.println();
}
```

En el caso de una matriz ***b***, ***b.length*** es el número de filas y ***b[0].length*** es el número de columnas (de la fila 0).

CLASES String y StringBuffer

Las clases ***String*** y ***StringBuffer*** están orientadas a manejar cadenas de caracteres.
Ambas clases pertenecen al package ***java.lang***, y por lo tanto no hay que importarlas

LA CLASE String:

Las cadenas de caracteres en Java se manipulan mediante la clase String.
El tipo String es inmutable, lo que significa, que una vez que un objeto String se ha creado, su contenido no puede modificarse. La clase ***String*** está orientada a manejar cadenas de caracteres constantes, es decir, que no pueden cambiar.

Los strings u objetos de la clase *String* se pueden crear explícitamente o implícitamente.

Implícitamente: `String str="El primer programa";`

Explícitamente: `String str=new String("El primer programa");`

Para crear un string nulo: `String str="";` o `String str=new String();`
No confundir con: `String str; //crea una ref a un String (que aun no se ha creado)`

Con los Strings se pueden usar los operadores + y +=

```
String str="Hola";      //str apunta a la cadena Hola
str = str + " Adios";   //str apunta a una nueva cadena Hola Adios
str += " Adios";        //equivale a la línea anterior
```

Hace que la referencia `str` apunte a un nuevo strings cuyo contenido es el original + la cadena adicional

```
String str=new String("Hola"); // Lo anterior es equivalente a esto
str = new String("Hola Adios");
```

MÉTODOS DE LA CLASE String

La siguiente tabla muestra los métodos más importantes de la clase ***String***:

Métodos de String	Función que realizan
String(...)	Constructores para crear Strings a partir de arrays de bytes o de caracteres (ver documentación on-line)
String(String str) y String(StringBuffer sb)	Costructores a partir de un objeto String o StringBuffer
charAt(int)	Devuelve el carácter en la posición especificada
getChars(int, int, char[], int)	Copia los caracteres indicados en la posición indicada de un array de caracteres
indexOf(String, [int])	Devuelve la posición en la que aparece por primera vez un String en otro String, a partir de una posición dada (opcional)
lastIndexOf(String, [int])	Devuelve la última vez que un String aparece en otro empezando en una posición y hacia el principio
length()	Devuelve el número de caracteres de la cadena
replace(char, char)	Sustituye un carácter por otro en un String
startsWith(String)	Indica si un String comienza con otro String o no
substring(int, int)	Devuelve un String extraído de otro
toLowerCase()	Convierte en minúsculas (puede tener en cuenta el locale)
toUpperCase()	Convierte en mayúsculas (puede tener en cuenta el locale)
trim()	Elimina los espacios en blanco al comienzo y final de la cadena
valueOf()	Devuelve la representación como String de sus argumento. Admite Object, arrays de caracteres y los tipos primitivos

Para obtener la longitud, número de caracteres de un string se llama a la función miembro length.

```
String str="El primer programa";
int longitud=str.length(); //18
```

Podemos conocer si un string comienza con un determinado prefijo, llamando al método startsWith, que devuelve true o false, según que el string comience o no por dicho prefijo

```
boolean resultado=str.startsWith("El");   //true
```

Podemos saber si un string finaliza con un conjunto de caracteres, mediante endsWith.

```
boolean resultado=str.endsWith("programa");
```

Para obtener la posición de la 1ª ocurrencia y sucesivas de la letra p, se usa la función indexOf.

```
int pos=str.indexOf('p');    //busca la primera ocurrencia de p
pos=str.indexOf('p', pos+1); //busca la siguiente (busca a partir de pos+1)
int pos=str.indexOf("pro");  //busca la primera ocurrencia de pro
```

Comparacion de Strings (= = compara las referencias, equals compara el contenido)

```
String str1="El lenguaje Java";
String str2=new String("El lenguaje Java");
if(str1==str2)   System.out.println("Los mismos objetos"); //false
else             System.out.println("Distintos objetos");  //true

if(str1.equals(str2)) System.out.println("El mismo contenido");//true
else                  System.out.println("Distinto contenido");//false

str2=str1;  //ambos apuntan al mismo objeto  (str1==str2) es true
```

La función *comapareTo* devuelve un entero < 0 si el objeto string es menor (en orden alfabético) que el string dado, 0 si son iguales, y > 0 si el objeto string es mayor que el string dado.

```
String str="Tomás";
int resultado=str.compareTo("Alberto");  //> 0 ya que Tomas > Alberto
```

Extraer un substring de un string: *substring*. (las posiciones se cuenta desde cero)

```
String str="El lenguaje Java";
String subStr=str.substring(12); //extrae "Java" que esta a partir de pos 12
String subStr=str.substring(3, 11); //extrae "lenguaje" que esta entre [3, 11)
```

Convertir un número en string: *valueOf*

```
int valor=10;
String str=String.valueOf(valor);
```

Convertir un string en número:

Se convierte el string en un objeto de la clase envolvente con el metodo estático valueOf, y se convierte el objeto de la clase envolvente en un tipo primitivo mediante el método tipo_primitivoValue

```
String str="  12 ";
int numero=Integer.valueOf(str).intValue();

String str="12.35 ";
double numero=Double.valueOf(str).doubleValue();
```

Otra forma de convertir un string en número entero:

```
String str="  12 ";
String str1=str.trim(); //quita los espacios en blanco
int numero=Integer.parseInt(str.trim()); //para convertirlo en int
```

STRINGBUFFER (java.lang.StringBuffer)

La clase String almacena una cadena constante, de modo que cuando se realizan variaciones directas sobre ellas (sustitución, etc) lo que se hace es crear una cadena nueva diferente a la anterior, con el consecuente gasto de memoria.

Imaginemos una función miembro a la cual se le pasa un array de cadenas de caracteres.

```
public class CrearMensaje{
        public String getMensaje(String[] palabras){
                String mensaje="";
                for(int i=0; i<palabras.length; i++){
                        mensaje+=" "+palabras[i];
                }
                return mensaje;
        }
//...
}
```

Cada vez que se añade una nueva palabra, se reserva una nueva porción de memoria y se desecha la vieja porción de memoria que es más pequeña (una palabra menos) para que sea liberada por el recolector de basura (grabage collector). Si el bucle se realiza 1000 veces, habrá 1000 porciones de memoria que el recolector de basura ha de identificar y liberar

Para evitar este efecto, ***Java*** **incluye una clase para realizar trabajos con cadenas que se van a modificar con frecuencia:** ***StringBuffer***.

Sus **métodos principales** son ***append*** e ***insert***. Ambos métodos están sobrecargados para trabajar con los diferentes tipos primitivos. Además, gracias al método ***toString***, las clases definidas por el programador se pueden adaptar para trabajar correctamente con esta clase.

> ***append*** **añade al final la cadena** que pasemos como argumento.
> ***insert*** **inserta** la cadena segundo argumento **en la posición especificada** en el primer argumento.
> Son también útiles ***length***, ***setLength*** y ***subString***.

```
public class CrearMensaje{
        public String getMensaje(String[] palabras){
                StringBuffer mensaje=new StringBuffer();
                for(int i=0; i<palabras.length; i++){
                        mensaje.append("  ");
                        mensaje.append(palabras[i]);
                }
                return mensaje.toString();
        }
//...
}
```

MÉTODOS DE LA CLASE StringBuffer

La siguiente tabla muestra los métodos más importantes de la clase ***StringBuffer***:

Métodos de StringBuffer	Función que realizan
StringBuffer(), StringBuffer(int), StringBuffer(String)	Constructores
append(...)	Tiene muchas definiciones diferentes para añadir un String o una variable (int, long, double, etc.) a su objeto
capacity()	Devuelve el espacio libre del StringBuffer
charAt(int)	Devuelve el carácter en la posición especificada
getChars(int, int, char[], int)	Copia los caracteres indicados en la posición indicada de un array de caracteres
insert(int,)	Inserta un String o un valor (int, long, double, ...) en la posición especificada de un StringBuffer
length()	Devuelve el número de caracteres de la cadena
reverse()	Cambia el orden de los caracteres
setCharAt(int, char)	Cambia el carácter en la posición indicada
setLength(int)	Cambia el tamaño de un StringBuffer
toString()	Convierte en objeto de tipo String

```
public class Programa {

        public static void main(String[] args){
                StringBuffer buffer = new StringBuffer( "Inicial" );
                buffer.append("final");
                buffer.insert(7,"-");
                buffer.delete( 0,2 );
                buffer.insert( 0,"INI" );
                buffer.append(" "+buffer.length());
                String cadena = buffer.toString();
                System.out.println( cadena );
        }

}
```

Salida:

```
INIicial-final 14
```

EXCEPCIONES EN JAVA

Una excepción es una condición anormal que se produce en el código en tiempo de ejecución (es un error en tiempo de ejecución). Una excepción Java es un objeto que describe un error.

Las excepciones en Java se utilizan de forma muy similar a C++.

- **Son clases que pueden ser creadas por el programador.**
- **Deben heredar de Throwable directa o indirectamente. Exception es subclase de Throwable,** con lo que basta con que nuestras excepciones hereden de Exception.
- **Para lanzarlas se utiliza la palabra reservada throw:**

  throw new <constructor de clase de excepcion>(<parámetros>)

  (**NOTA**: observe que hay que explicitar el new)

- **Cuando un método lanza alguna excepción, es necesario explicitar en su cabecera las excepciones que puede lanzar. (Diferente de C++)**
- Las excepciones **se capturan en un bloque try – catch**

```
try{
     …
     código
     …
}catch( <clase de excepción 1>e ){

}[catch(<clase de excepción 2> e ){
}]*
```

- Los **bloques catch se denominan manejadores de excepciones**. El funcionamiento es:
  - Si se lanza una excepción dentro del bloque try, se busca en los bloques catch, dónde es tratada la excepción.
    - En caso de que haya coincidencia, se ejecuta el manejador y se considera que la excepción ha sido tratada y solucionada. Se pasa el flujo de control al final de la construcción try-catch o al bloque finally en caso de que exista.
    - En caso de que no haya coincidencia (se ejecuta el bloque finally en caso de que exista y) la excepción se lanza hacia el contexto superior.
- **NOTA:** al igual que en C++, **cuando se da una excepción** en un bloque try, **se pasa a los bloques catch en el mismo momento en que se lance dicha excepción**, es decir, no se tiene por qué ejecutar todo el código dentro del try.
- **Bloque finally**: **dentro de una construcción try-catch, se puede incluir un bloque finally. Este bloque se ejecuta siempre.**

EXCEPCIONES EN JAVA:

MiExcepcion
- String cadena
+ MiExcepcion(String cadena) + String toString()

PuntoPositivo
- int x - int y
+ PuntoPositivo(int nx, int ny) throws MiExcepcion + String toString()

```
package principal;

import java.util.Scanner;

class MiExcepcion extends Exception{
        private String cadena;
        public MiExcepcion( String ncad ){
                cadena = ncad;
        }
        public String toString(){ return cadena; }
}

class PuntoPositivo{

        private int x;
        private int y;
        public PuntoPositivo( int nx, int ny) throws MiExcepcion {
                if ( nx < 0 || ny <0 )
                        throw new MiExcepcion( "Error: coordenadas
negativas" );
                x=nx;
                y=ny;
        }
        public String toString(){
                return new String( "("+x+":"+y+")" );
        }
}

public class Programa {

        public static void main(String[] args){

                Scanner teclado = new Scanner( System.in );
                boolean salir = false;

                while (!salir){
                        System.out.println("Introduzca las coordenadas:");
                        try{
                                PuntoPositivo p = new
                                        PuntoPositivo( teclado.nextInt(),
                                                teclado.nextInt() );
                                System.out.println(p);
                        }catch( Exception e ){
                                System.out.println( e );
                                salir = true;
                        }finally{
                                System.out.println( "Esto se ejecuta
siempre" );
                        }
                }

                System.out.println( "Finalizando el programa" );
        }
}
```

GENÉRICOS (JAVA 5)

A partir de Java 5 se pueden definir clases genéricas utilizando plantillas. Anteriormente, era posible hacer esto utilizando objetos de la clase **Object**.

La **forma de declarar** una clase genérica es poner una **lista de variables de clase** separadas por coma, **entre menor y mayor (< >) detrás del nombre de la clase**. Dentro del ámbito de la clase podemos utilizar estas variables de clase para declarar las clases de los parámetros y de variables locales.

Vamos a ver un ejemplo hecho en Java 1.4 y otro en Java 1.5

Punto genérico hasta Java 1.4 (compatible con posteriores…)

Punto
+ **Object** x + **Object** y
+Punto (**Object** nx, **Object** ny) + String toString()

Pantalla:

```
(1,2)
(3,2)
1---1
```

```
package principal;

class Punto{
  public Object x;
  public Object y;
  public Punto( Object nx, Object ny ){
        x = nx;
        y = ny;
  }
  public String toString(){
        return new String( "("+x+","+y+")" );
  }
}

public class Programa {
  public static void main(String[] args){
        Punto p1 = new Punto( new Integer(1), new Integer(2) );
        Punto p2 = new Punto( 3, new Integer(2) );
        System.out.println( p1.toString()+"\n"+p2);
        Integer x = (Integer)p1.x;
        int z = (Integer)p1.x;      //int z =( (Integer)p1.x ).intValue();
      //Integer y = p2.y; //ERROR falta (Integer)p2.y;
        System.out.println(x+"---"+z);
  }
}
```

Nota:
Todos tipos primitivos de Java tiene asociado una clase envolvente (ver página 6) que se invoca automáticamente cuando es necesario, como ocurre con el int 3 que es convertido a un objeto de tipo Integer(3) y como ocurre cuando el objeto Integer es convertido a int al asignarlo a z

Inconvenientes Genéricos hasta Java 1.4:

1) **Los atributos x, y pueden ser de distinto tipo (pueden ser cualquier tipo no primitivo)**: ej, x puede ser un **Integer** (clase envolvente del tipo primitivo **int**) e y puede ser un **Double** (clase envolvente del tipo primitivo **double**) o y puede ser una Fecha o un **String**.

   Punto p = new Punto("pepe", **new Fecha(2,2,2015)**);
   Punto p = new Punto(3.56, **new Persona(25, "luis")**);

2) **Al hacer referencia al atributo x en el main( ) hemos tenido que hacer un casting explícito, ya que x es de tipo Object (lo mismo que y)**

   Integer x = **(Integer)**p1.x;
   int z = (**Integer**)p1.x;

GENÉRICOS (JAVA 5)

Punto genérico hasta Java 5

Punto<**T**>
+ **T** x + **T** y
+Punto (**T** nx, **T** ny) + String toString()

Pantalla:

```
(1,2)
(3,2)
1---1
```

```
package principal;

class Punto<T>{
   public T x;
   public T y;
   public Punto( T nx, T ny ){
         x = nx;
         y = ny;
   }
   public String toString(){
         return new String( "("+x+","+y+")" );
   }
}

public class Programa {
   public static void main(String[] args){
         Punto<Integer> p1 = new Punto<Integer>( new
Integer(1), new Integer(2) );
         Punto<Integer> p2 = new Punto< Integer >( 3, 2 );
         System.out.println( p1.toString()+"\n"+p2);
         Integer x = p1.x; //no hay que hacer casting (Integer)p1.x
         int z = p1.x; //no hay que hacer casting (Integer)p1.x
         Integer y = p2.y; //NO DA ERROR si falta (Integer)p2.y;
         System.out.println(x+"---"+z);
   }
}
```

Nota:
Todos tipos primitivos de Java tiene asociado una clase envolvente (ver página 6) que se invoca automáticamente cuando es necesario, como ocurre con el int 3, 2 que es convertido a un objeto de tipo Integer(3) e Integer(2)) y como ocurre cuando el objeto Integer es convertido a int al asignarlo a z

Ventajas Genéricos Java 5:

1) **Los atributos x, y deben ser del mismo tipo (el tipo puede ser cualquier tipo no primitivo)**: ej, Si x es un **Integer** (clase envolvente del tipo primitivo **int**) y también lo es, si x es de tipo Fecha y también lo es.

   Punto p = new Punto("pepe", **new Fecha(2,2,2015)**); //ERROR
   Punto<String> p = new Punto<String>("pepe", "luis"); //CORRECTO

2) **Al hacer referencia al atributo x en el main( ) NO hemos tenido que hacer un casting explicito, ya que JAVA deduce que x (y) son de tipo Integer**

   Integer x = **(Integer)**p1.x; //no es necesario el casting
   int z = (**Integer**)p1.x; //no es necesario el casting

IMPORTANTE:
Los parámetros genéricos de tipo (<T>) sólo pueden representar referencias (no tipos primitivos)

Punto<**int**> p1; //ERROR
Punto<**Integer**> p1; //CORRECTO: Usamos la clase envolvente de **int** si queremos guardar enteros

GENÉRICOS (JAVA 5)

Ejemplo: cola genérica utilizando genéricos. (Se puede hacer utilizando un array de **Object**)
(pero es mejor usar un array de T)

Punto
- double x - double y
+Punto(double nx, double ny) + String toString()

Cola<T>
- Object array[ ] - int elementos - int tamaño
+ Cola<T>(int tamaño) + void encolar(T e) + T desencolar() + T foco() + int encolados()

Mejor:

Cola<T>
- T array[] - int elementos - int tamaño
+ Cola<T>(int tamaño) + void encolar(T e) + T desencolar() + T foco() + int encolados()

```
package principal;

class Punto{                            //clase no generica

        private double x;
        private double y;
        public Punto( double nx, double ny ){
                x = nx;
                y = ny;
        }
        public String toString(){
                return new String( "("+x+","+y+")" );
        }
}

class Cola<T>{                          //clase generica
        //private Object array[ ];
          private T array[ ];
          private int elementos=0;
          private int tamaño=0;
          public Cola( ) { this(5); } //tamaño predeterminado de la cola
          public Cola( int tam ){
                tamaño = tam;
                //array = new Object[tamaño];
                 array = ( T [ ] )new Object[tamaño];
          }
          public void encolar( T e ){     //metodo generico
                if ( elementos < tamaño )
                        //array[elementos++] = (Object)e;
                         array[elementos++] = e;
          }
          public T desencolar( ){         //metodo generico
                if ( elementos > 0 ){
                        T tmp = foco();
                        // Movemos todos una posición adelante
                        for ( int i=1; i<elementos; i++ )
                                array[i-1]=array[i];
                        // Eliminamos la referencia!
                        array[elementos-1] = null;
                        elementos--;
                        return tmp;
                }
                return null;
          }
          public T foco( ){               //metodo generico
                if ( elementos > 0 )
                        //return (T)array[0];
                         return array[0];
                return null;
          }
          public int encolados( ){
                return elementos;
          }
}
```

GENÉRICOS

Continuación del ejemplo anterior.

Punto
- double x - double y
+PuntoPositivo(double nx, double ny) + String toString()

Cola<T>
- Object array[ ] - int elementos - int tamaño
+ Cola<T>(int tamaño) + void encolar(T e) + T desencolar() + T foco() + int encolados()

Mejor:

Cola<T>
- T array[] - int elementos - int tamaño
+ Cola<T>(int tamaño) + void encolar(T e) + T desencolar() + T foco() + int encolados()

Pantalla:

```
2 6 8 3 8 6
(2.0,4.0)
(0.0,6.0)
(1.0,8.0)
(2.0,3.0)
5 6 7 8
Ultimo: 8
1.0 2.0 3.0 4.0
Ultimo: 4.0
ocho siete seis cinco
Ultimo: cinco
```

Los parámetros genéricos de tipo (<T>) sólo pueden representar referencias (no tipos primitivos).

Por ello, los arrays ti y td son de tipo Integer y Double (clases envolventes de los tipos primitivos int y double)

```
public class Programa {       //clase no generica
  public static <T> void cargar(Cola<T> c, T[] t) { //metodo generico
            for (int i=0; i<t.length; i++)
                        c.encolar( t[i] );
  }
  public static <T> Cola<T> cargar(T[] t) {          //metodo generico sobrecargado
            Cola<T> aux=new Cola<T>(t.length);
            for (int i=0; i<t.length; i++)
                        aux.encolar( t[i] );
            return aux;
  }
  public static Cola<String> cargar(String[] t) {      //especializacion generico
            Cola<String> aux=new Cola<String>(t.length);
            for (int i= t.length-1; i>=0; i--)
                        aux.encolar( t[i] );
            return aux;
  }
  public static <T> T vaciar(Cola<T> c) {              //metodo genérico vaciar
            T ultimo=null;
            while( c.encolados() > 0 ) {
                        ultimo=c.desencolar();
                        System.out.print( ultimo+" ");
            }
            return ultimo;
  }
  public static void main(String[] args){
            Cola<Integer> cint = new Cola<Integer>(10);
            Cola<Punto> cpunto = new Cola<Punto>(10);
            int [] array = { 6, 8, 3 };
            cint.encolar( new Integer(2) );
            cpunto.encolar( new Punto(2,4) );
            for ( int i=0; i<array.length; i++ ) {
                        cint.encolar( new Integer(array[i]) );
                        cpunto.encolar( new Punto(i,array[i]) );
            }
            cint.encolar( 8 ); // boxing: 8 →new Integer(8)
            cint.encolar( array[0] ); // boxing: array[0] → new Integer(array[0])
            while( cint.encolados() > 0 )
                        System.out.print( cint.desencolar()+" ");
            System.out.println();
            while( cpunto.encolados() > 0 )
                        System.out.println( cpunto.desencolar() );

            Integer [] ti = {5, 6, 7, 8 };
            String [] ts = {"cinco", "seis", "siete", "ocho" };
            Double [] td=new Double[4];
            for (int i=0; i< td.length; i++)
                        td[i]=(double)i+1;
            Cola<Integer> icola=new Cola<Integer>(ti.length);
            cargar(icola,ti);          //ejecuta metodo generico
            int i=vaciar(icola);       //ejecuta metodo genérico vaciar
            System.out.println("\nUltimo: " + i);

            Cola<Double> dcola=cargar(td);  //ejecuta metodo genérico sobrecargado
            double d=vaciar(dcola);         //ejecuta metodo genérico vaciar
            System.out.println("\nUltimo: " + d);

            Cola<String> scola=cargar(ts);    //ejecuta especializacion generico
            String s=vaciar(scola);           //ejecuta metodo genérico vaciar
            System.out.println("\nUltimo: " + s);
  }
}
```

GENÉRICOS (JAVA 5)

Ejemplo: clase genérica con múltiples parámetros genéricos e interfaz genérica

Par<**C,V**>
+**C** getClave() +**V** getValor()

```
public interface Par<C, V> {   //interfaz genérica
   public C getClave();
   public V getValor();
}
```

Pareja<**C, V**> **implements** Par<**C, V**>
- **C** clave - **V** valor
+ Pareja(**C** clave, **V** valor) + **C** getClave() + **V** getValor() + String toString()

Observaciones:
Al crear un objeto de una clase genérica, los parámetros de tipo pueden ser de tipo parametrizado.

Si una clase implementa una interfaz podemos usar referencias del tipo de la interfaz para apuntar a objetos de dicha clase.

Con una referencia del tipo de la interfaz sólo se pueden invocar los métodos definidos en la interfaz (no se puede invocar ningún método que no esté definido en la interfaz).

Aparte de los métodos definidos en una interfaz, una referencia de una interfaz también puede invocar los métodos de la clase **Object**, como por ejemplo **toString( )**, **equals( )**, etc. ya que la interfaz también los hereda de **Object**.

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class Pareja<C, V> implements Par<C, V> {  //clase genérica

   private C clave;
   private V valor;

   public Pareja(C clave, V valor) {
         this.clave = clave;
         this.valor = valor;
   }

   public void saludo()  { System.out.println("hola"); }
   public C getClave( )  { return clave; }
   public V getValor( )  { return valor; }
   public String toString( ){
                  return new String( clave+": "+valor );
   }

   public static void main(String[] args){
         Par<String, Integer> p1;
         p1= new Pareja<String, Integer>("edad", 8);
         //autoboxing: 8 --> new Integer(8)
         Pareja<String, String>  p2;
         p2= new Pareja<String, String>("hola", "mundo");
         Pareja<Integer, Double>  p3;
         p3= new Pareja<Integer, Double>(5, 7.89);
         ///autoboxing 5 --> new Integer(5),   7.89 --> new Double(7.89)
         List<Integer> lista=new ArrayList<Integer>(Arrays.asList(1,2,3));
         Par<String, List<Integer>> p4;
         p4= new Pareja<String, List<Integer>>("edad", lista);
         System.out.println(p1+"\n"+p2+"\n"+p3+"\n"+p4);
         p3.saludo();
         System.out.println(p3.toString());
         //p4.saludo(); //ERROR saludo() no está definido en la interfaz
         ((Pareja<String, List<Integer>>)p4).saludo();
         System.out.println(p4.toString());
   }
}
```

Genéricos y Herencia: Observaciones

Una clase (genérica o no) puede derivar de una clase genérica o de una clase no genérica

```
class Derivada extends Base { ... }          class Derivada<P> extends Base { ... }
class Derivada extends Base<P> { ... }       class Derivada<P> extends Base<P> { ... }
```

Un método genérico de una subclase puede sobreescribir el método genérico de su superclase

GENÉRICOS (JAVA 5)

Métodos genéricos y parámetros de tipos delimitados

Los parámetros genéricos de tipo (<**T**>) pueden representar referencias (no tipos primitivos), las cuales pueden ser de cualquier tipo de clase.

Puede haber momentos en los que desea restringir los tipos que se pueden utilizar como argumentos de tipo en un tipo parametrizado <**T**>. Por ejemplo, un método que funciona con números solamente podría querer aceptar objetos de la clase **Number** o de sus subclases (**Integer**, **Double**, **Float**, **Long**, etc.).

Es posible delimitar los tipos a los que pueden hacer referencia esos parámetros usando la palabra **extends** seguido de su límite superior. En este contexto, **extends** se utiliza para significar que "extiende" (como en las clases) o "implementa" (como en las interfaces). Asi:

<T **extends** Number>	indica que **T** puede ser de la clase **Number** y de cualquier otra que herede de **Number** (que la extienda)
<T **extends** Cloneable>	indica que **T** puede ser de cualquier clase que implemente la interfaz **Cloneable**
<T>	equivale a poner <T **extends** Object>

<T **extends** B1 & B2 & B3>	indica que T puede ser de la clase B1 o derivada de B1 y debe implementar las interfaces B2 y B3 (B1 puede ser una interfaz)
	`class A { /* ... */ }`
	`interface B { /* ... */ }`
	`interface C { /* ... */ }`
	`class D <T extends A & B & C> { /* ... */ }`
	Si `A` no aparece al principio se produce un error de compilación:
	`class D <T extends B & A & C> { /* ... */ }  //error`

Utilidad:

Los parámetros de tipo acotadas son clave para la implementación de algoritmos genéricos. Considere este método que cuenta cuantos elementos en una matriz T [] son mayores que un elemento elem.

```
public static <T> int countGreaterThan(T[] anArray, T elem) {
    int count = 0;
    for (T e : anArray)
        if (e > elem)  // compiler error
            ++count;
    return count;
}
```

El método no compila porque el operador (>) solo se aplica a tipos primitivos (**int**, **double**) y no a objetos. **Solución**: usar un parámetro de tipo limitado por la interfaz **Comparable <T>** o por la clase **Number**:

```
public interface Comparable<T> {
    public int compareTo(T o);
}
//public static < T extends Number > int countGreaterThan2(T[] anArray, T elem){
public static < T extends Comparable<T> > int countGreaterThan(T[] anArray, T elem){
    int count = 0;
    for (T e : anArray)
        if (e.compareTo(elem) > 0)
            ++count;
    return count;
}
```

Solo las clases que implementen esa interfaz van a poder ser pasadas como parámetro al método

Las clases envolventes de los tipos primitivos (**Integer**, **Float**, etc.) y la clase **String** implementan esa interfaz (tienen el método `compareTo`), por lo que pueden pasarse objetos de esos tipos (y de cualquier otra clase que la implemente).

FLUJOS EN JAVA

Un ***stream*** es una conexión entre el programa y la fuente o destino de los datos.

El package ***java.io*** contiene las clases necesarias para la comunicación del programa con el exterior.

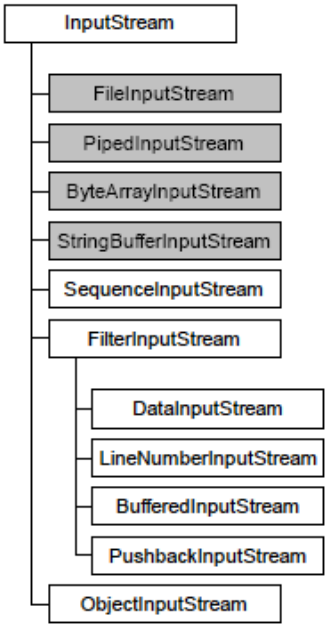


Figura 9.1. Jerarquía de clases InputStream.

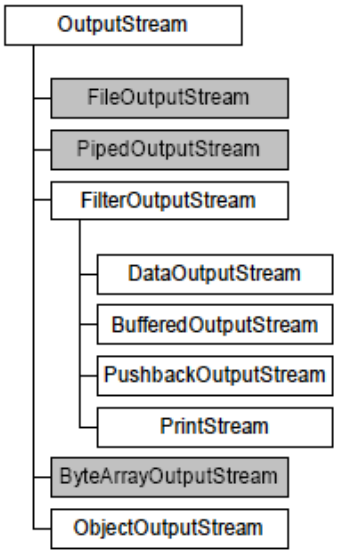


Figura 9.2. Jerarquía de clases OutputStream.

Palabra	Significado
InputStream, OutputStream	Lectura/Escritura de bytes
Reader, Writer	Lectura/Escritura de caracteres
File	Archivos
String, CharArray, ByteArray, StringBuffer	Memoria (a través del tipo primitivo indicado)
Piped	Tubo de datos
Buffered	Buffer
Filter	Filtro
Data	Intercambio de datos en formato propio de Java
Object	Persistencia de objetos
Print	Imprimir

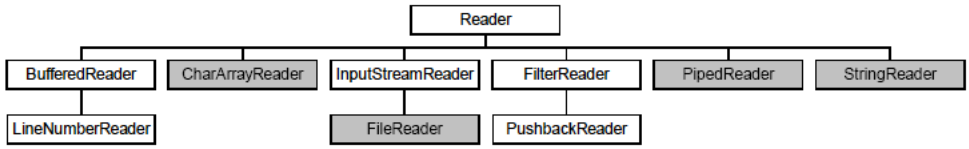
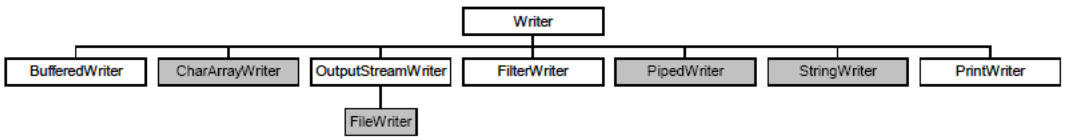


Figura 9.3. Jerarquía de clases Reader.



LECTURA Y ESCRITURA DE ARCHIVOS

```
File f1 = new File("c:\\windows\\notepad.exe");  // La barra '\' se escribe '\\'
File f2 = new File("c:\\windows");               // Un directorio
File f3 = new File(f2, "notepad.exe");           // Es igual a f1
```

Si ***File*** representa un archivo que existe los métodos de la Tabla 9.6 dan información de él.

Métodos	Función que realizan
boolean isFile()	true si el archivo existe
long length()	tamaño del archivo en bytes
long lastModified()	fecha de la última modificación
boolean canRead()	true si se puede leer
boolean canWrite()	true si se puede escribir
delete()	borrar el archivo
RenameTo(File)	cambiar el nombre

Tabla 9.6. Métodos de File para archivos.

Si representa un directorio se pueden utilizar los de la Tabla 9.7:

Métodos	Función que realizan
boolean isDirectory()	true si existe el directorio
mkdir()	crear el directorio
delete()	borrar el directorio
String[] list()	devuelve los archivos que se encuentran en el directorio

Tabla 9.7. Métodos de File para directorios.

```
public class Fichero {
public static void main(String args[]) {
String nombreF;
try {
        nombreF = "fichero.bin";
        File canal = new File (nombreF);
        System.out.println("Nombre: "+canal.getName());
        System.out.println("camino: "+canal.getPath());
        if (canal.exists()) {
                System.out.println("Fichero existente ");
                if (canal.canRead()) {
                        System.out.println("Se puede leer");
                }
                if (canal.canWrite()) {
                        System.out.println("Se puede escribir");
                }
                System.out.println("La longitud del fichero  es "+ canal.length()+ " bytes");
        }
        else {
                System.out.println("El fichero no existe.");
        }
        } catch (Exception e) {
                System.out.println(e.getMessage());
        }
```

```
Nombre: fichero.bin
camino: fichero.bin
Fichero existente
Se puede leer
Se puede escribir
La longitud del fichero es 179 bytes
```

CONSTRUCCIÓN DE UN ARCHIVO:

FileInputStream y ***FileOutputStream*** permiten leer y escribir ***bytes*** en archivos.
FileReader y ***FileWriter***, permiten leer y escribir ***caracteres*** en archivos

Ambos se forman a partir de un objeto de tipo file o directamente lo crea a partir de un *String* con su nombre.

```
FileReader fr1 = new FileReader("archivo.txt");
```

Equivale a

```
File f = new File("archivo.txt");
FileReader fr2 = new FileReader(f);
```

Lectura de archivos de texto

```
String texto = new String();
try {
       FileReader fr = new FileReader("archivo.txt");
       entrada = new BufferedReader(fr);
       String s;
       while((s = entrada.readLine()) != null)
              texto += s;
       entrada.close();
}
catch(java.io.FileNotFoundException fnfex) {
System.out.println("Archivo no encontrado: " + fnfex);}
```

Escritura de archivos de texto

```
try {
       FileWriter fw = new FileWriter("escribeme.txt");
       BufferedWriter bw = new BufferedWriter(fw);
       PrintWriter salida = new PrintWriter(bw);
       salida.println("Hola, soy la primera línea");
       salida.close();
       // Modo append
       bw = new BufferedWriter(new FileWriter("escribeme.txt", true));
       salida = new PrintWriter(bw);
       salida.print("Y yo soy la segunda. ");
       double b = 123.45;
       salida.println(b);
       salida.close();
}
cacth(java.io.IOException ioex) { }
```

Archivos que no son de texto

Para evitar conversiones de tipo, se graba en un formato propietario de java independiente de la plataforma:

```
// Escritura de una variable double
DataOutputStream dos = new DataOutputStream(
new BufferedOutputStream(new FileOutputStream("prueba.dat")));
double d1 = 17/7;
dos.writeDouble(d);
dos.close();
// Lectura de la variable double
DataInputStream dis = new DataInputStream(
  new BufferedInputStream(new FileInputStream("prueba.dat")));
double d2 = dis.readDouble();
```

SERIALIZACIÓN

```
public class MiClase implements Serializable { }
```

Clases → ***ObjectInputStream*** y ***ObjectOutputStream***,
Métodos →
writeObject()
readObject() : Ojo para que sea util el objeto devuelto hay que realizar un casting

```
ObjectOutputStream objout = new ObjectOutputStream(
new FileOutputStream("archivo.x"));
String s = new String("Me van a serializar");
objout.writeObject(s);
ObjectInputStream objin = new ObjectInputStream(new FileInputStream("archivo.x"));
String s2 = (String)objin.readObject();
```

Notas:

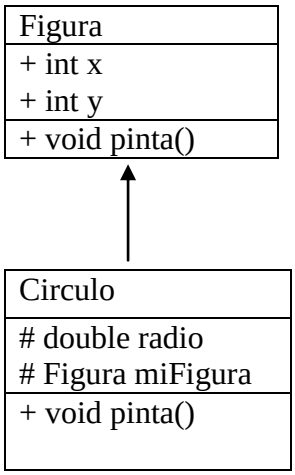
- Las variables y objetos ***static*** no son serializados.
- ***transient*** permite indicar que un objeto o variable miembro no sea serializado.
    - Al recuperarlo, lo que esté marcado como ***transient*** será ***0***, ***null*** o ***false***

Redefinición de ***Serializable*** → . No están obligadas a hacerlo →comportamiento por defecto. Si los define:

```
private void writeObject(ObjectOutputStream stream) throws IOException
private void readObject(ObjectInputStream stream) throws IOException
```

```
static double g = 9.8;
private void writeObject(ObjectOutputStream stream) throws IOException {
stream.defaultWriteObject();
stream.writeDouble(g);
}
private void readObject(ObjectInputStream stream) throws IOException {
stream.defaultReadObject();
g = stream.readDouble(g);
}
```

SERIALIZABLE




```
class Figura implements Serializable{
        public double x;
        public double y;

        public Figura ( double nx, double ny){
                x=nx;
                y=ny;
        }
        public void pinta(){
                System.out.println( " Figura >> ("+x+","+y+")\n");
        }
}

class Circulo extends Figura {
        public double radio;
        Figura mi Figura;
        public Circulo(  double nx, double ny, double r ){
                super(nx,ny);
                radio = r;
                mi Figura = new Figura (1,1);
        }
        public void pinta(){
                System.out.println( "Circulo >> " + this +"\n");
                mi Figura.pinta();
        }
        public static void main(String[] args){
                 Figura p=new Figura (10.5,6.2);
                Circulo c=new Circulo(1,2,2.45);
                p.pinta();
                c.pinta();
                try {//escritura
                        FileOutputStream fos = new FileOutputStream("fichero.bin");
                        ObjectOutputStream out = new ObjectOutputStream(fos);
                        out.writeObject(c);
                        out.writeObject(c. mi Figura);
                        out.writeObject(p);
                //Lectura
                        FileInputStream fis = new FileInputStream("fichero.bin");
                        ObjectInputStream in = new ObjectInputStream(fis);
                        c.mi Figura = p;
                        c = (Circulo)in.readObject();
                        c.mi Figura = (Figura)in.readObject();
                        p = (Figura)in.readObject();
                        }
                }
                catch (IOException e)
                {
                        System.out.println("Error de fichero: "+e);
                }
                catch (ClassNotFoundException e){
                        System.out.println("Error: clase not found "+e);
                }
                p.pinta();
                c.pinta();
        }
}
```

APENDICE 1: INFORMACIÓN DE TIPOS. EL OBJETO CLASS

Java crea un objeto **Class** para cada clase que forme parte de un programa. Es decir, cada vez que creamos (compilamos y escribimos) una nueva clase, Java crea un determinado objeto **Class** (y ese objeto se almacena en un archivo .class de nombre idéntico).

Todos los objetos **Class** son instancias de la clase **Class<T>** (es una clase genérica parametrizada**)**, la cual tiene una serie de métodos que permiten obtener información de dicha clase.

Class.forName(nombreClase)	Devuelve el objeto Class asociado con la clase o interface llamada nombreClase
nombreClase.class (no es un método, es un literal)	Devuelve el objeto Class asociado con la clase o interface llamada nombreClase
objetoClass.getName( )	Devuelve el nombre de la clase completamente cualificado (ej: libClase.Fecha)
objetoClass.getSimpleName( )	Devuelve el nombre sin el paquete (ej: Fecha)
objetoClass.isInterface( )	True si el objeto objetoClass es una interface
objetoClass.getInterfaces( )	Devuelve una matriz de objetos Class con las interfaces que implementa el objeto objetoClass
objetoClass.getSuperclass( )	Devuelve el objeto Class que representa la clase de la que hereda la clase objetoClass
objetoClass.newInstance( )	Crea un objeto de la clase objetoClass. La clase debe tener un constructor sin parámetros

Usando la clase Class es posible conocer la clase a la que pertenece un objeto en tiempo de ejecución (e incluso tener información de si algo es clase o subclase).

Dado un objeto, podemos obtener su **Class** invocando el método **getClass( )** heredado de **Object**

```
class Figura { . . . }
class Circulo extends Figura { . . . }
public class Programa {
        public static void main(String[] args){
                Figura p = new Circulo(0,0,1);
                Circulo c = new Circulo(1,2,2.3);
                if (p.getClass() == Circulo.class)
                        System.out.println("p apunta a un objeto de clase Circulo");
                System.out.println("p: " + p.getClass() +" --- c: " + c.getClass());
                if ( c.getClass().isInstance( p ) )
                        System.out.println( "p es de la misma clase que c o subclase de la clase de c");
                if (  c.getClass( ) == p.getClass( ) ) //MEJOR
                        System.out.println("c y p son de la misma clase");
        }
}
```

Pantalla:

```
p  apunta a un objeto de clase Circulo
p: class Circulo --- c: class Circulo
p es de la misma clase que c o subclase de la clase de c
c y p son de la misma clase
```

p **instanceof** Circulo significa lo mismo que Circulo**.class.isInstance(**p**)**	¿p es de la clase Circulo o subclase de Circulo?
p.getClass() == Circulo.class significa lo mismo que p.getClass() .equals(Circulo.class)	¿p es de la clase Circulo?
c.getClass().isInstance(p)	¿p es de la misma clase o subclase de la clase de c?
c.getClass( ) == p.getClass( )	¿p y c son de la misma clase?

Ej: El siguiente programa ilustra el funcionamiento de algunos de los métodos de la clase **Class<T>**

`T` – el tipo de la clase que modela el objeto `Class`. Ej, el tipo de `String.class` es `Class<String>`.
Use `Class<?>` si no conoce la clase que se va a modelar (el símbolo ? indica "cualquier clase")

```
package libClase;

interface Piedra { }
interface Papel { }
interface Tijera { }

class Base {
      public Base() { System.out.println("Objeto Base creado!!!"); }
      public Base(int i) {}
}

class Derivada extends Base
implements Piedra, Papel, Tijera {
      public Derivada() { super(1); }
}

public class Test {
      static String info(Class<?> cc) {
            String s="Nombre de clase: " + cc.getName();
            s=s+ " Es una interfaz? [" + cc.isInterface() + "]";
            s=s+" Nombre simple: "+ cc.getSimpleName();
            return s;
      }
      public static void main(String[] args){
            Class<?> d=null, c=null;
            try {
                  d=Class.forName("libClase.Derivada");
            } catch (Exception e) {
                  System.out.println("La clase 'Derivada' no existe");
                  System.exit(1);
            }
            System.out.println(Test.info(d));
            System.out.println("-----------");
            c=Derivada.class;
            System.out.println(Test.info(c));

            Class<?> [] t=c.getInterfaces();
            for(int i=0; i<t.length; i++)
                  System.out.println(Test.info(t[i]));
            Class<?> up=c.getSuperclass();
            Object obj=null;
            try {
                  obj=up.newInstance();
            } catch (InstantiationException e) {
                  System.out.println("No lo puede instanciar");
                  System.exit(1);
            } catch (IllegalAccessException e) {
                  System.out.println("No puedo acceder");
                  System.exit(1);
            }
            System.out.println(Test.info(obj.getClass()));
      }
}
```

Pantalla:

```
Nombre de clase: libClase.Derivada Es una interfaz? [false] Nombre simple: Derivada
-----------
Nombre de clase: libClase.Derivada Es una interfaz? [false] Nombre simple: Derivada
Nombre de clase: libClase.Piedra Es una interfaz? [true] Nombre simple: Piedra
Nombre de clase: libClase.Papel Es una interfaz? [true] Nombre simple: Papel
Nombre de clase: libClase.Tijera Es una interfaz? [true] Nombre simple: Tijera
Objeto Base creado!!!
Nombre de clase: libClase.Base Es una interfaz? [false] Nombre simple: Base
```

APENDICE 2: GENÉRICOS, HERENCIA Y SUBTIPOS

Es posible asignar un objeto de un tipo a un objeto de otro tipo, siempre que los tipos sean compatibles. Ej: Un **Integer** puede asignarse a un **Object** o a un **Number**, ya que ambos son supertipos de **Integer** .

```
Object someObject = new Object();
Integer someInteger = new Integer(10);
someObject = someInteger;   // OK

public void someMethod(Number n) { /* ... */ }
someMethod(new Integer(10));   // OK
someMethod(new Double(10.1));   // OK
```

Con los **Genéricos** ocurre lo mismo:

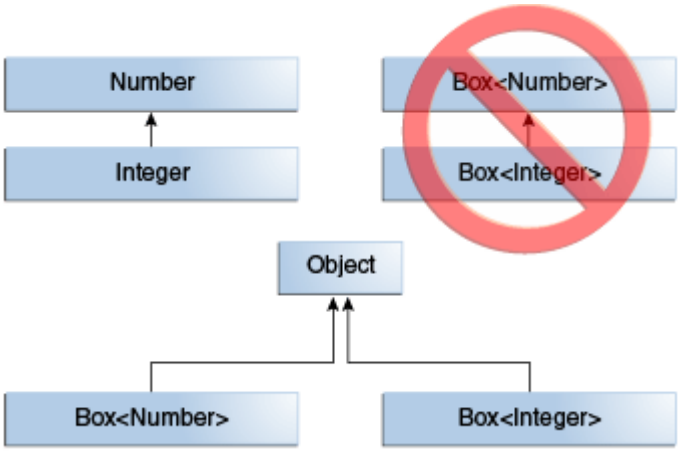
Ej: podemos invocar un Genérico pasándole **Number** como argumento de tipo y añadir a dicho genérico cualquier dato (**Integer**, **Double**, etc) compatible con **Number**

```
Box<Number> box = new Box<Number>();
box.add(new Integer(10));   // OK
box.add(new Double(10.1));  // OK
```

En cambio, dado el siguiente método:

```
public void boxTest(Box<Number> n) { /* ... */ }
```

Podría pensar erróneamente que al `boxTest()` podríamos pasarle como parámetro un `Box<Integer>` o un `Box<Double>` y la respuesta es no ya que, aunque `Integer` y `Double` derivan de `Number`, ello no implica que `Box<Integer>` y `Box<Double>` deriven de `Box<Number>`



Si queremos indicar que un objeto de una clase Genérica acepte diferentes tipos (cualquier clase) debemos usar el carácter comodín (?) Ej:

```
Dada la clase genérica Box<T>
Box<?> caja; //caja es una referencia de tipo Box<T> que acepta cualquier clase
Box<?> caja = new Box<Integer>() //OK    Box<?> caja = new Box<Fecha>() //OK
```

Al usar un objeto genérico, el carácter comodín (?), representa un tipo desconocido (significa “cualquier tipo no primtico”, “cualquier clase”). El ? se puede utilizar en una variedad de situaciones: como el tipo de un parámetro, un campo o variable local o como tipo de retorno.

El comodín nunca se usa como un argumento de tipo al invocar un método genérico, ni en la creación de un objeto de una clase genérica, o un supertipo.

ACOTAR SUPERIOR E INFERIORMENTE CON EL COMODÍN (?)

Es posible acotar superiormente o inferiormente (ambos a la vez no) un comodín (?) para restringir los tipos que puede aceptar.

Para acotarlo superiormente usamos la palabra **extends** seguido de su límite superior. Aquí **extends** se utiliza para significar que "extiende" (como en las clases) o "implementa" (como en las interfaces).

Para acotarlo inferiormente usamos la palabra **super** seguido de su límite inferior. Aquí **super** se utiliza para significar "cualquier superclase".

List<? **extends** Number> (**Number** es el límite superior)	indica que la clase List<T> acepta listas de la clase **Number** y listas de subclases de **Number** (cualquier otra clase que herede y extienda **Number**) Ej: List<**Number**>, List<**Integer**>, List<**Double**>
List<**Number**>	Sólo acepta List<**Number**>, no acepta List<**Integer**> o List<**Double**>
List<? **super** Number>	indica que List<T> acepta listas de la clase **Number** y listas de cualquier superclase de **Number** (de la que herede) Ej: List<**Number**> y List<**Object**> SI las acepta (son superclases de **Number**) List<**Integer**> o List<**Double**> NO las acepta (son subclases de **Number**)

Ej: Método que sume una lista de enteros o una lista de float, o una lista de double

```
public static double sumOfList(List<? extends Number> list) {
    double s = 0.0;
    for (Number n : list)
        s += n.doubleValue();
    return s;
}
```

Como la clase **Number** es la superclase de todas las clases envolventes de los tipos numéricos primitivos (**Integer**, **Float**, **Double**, etc.) escogemos **Number** como límite superior.

Para recorrer la lista de forma genérica usamos una variable **Number** (límite superior) para garantizar que el código funciona para cualquier tipo de lista que acepte (**Number** acepta cualquier cosa que sea **Number** o derive de ella, que son los tipos de los que puede ser la lista aceptada por parámetro)

```
List<Integer> li = Arrays.asList(1, 2, 3);
List<Double> ld = Arrays.asList(1.2, 2.3, 3.0);
System.out.println("sum = " + sumOfList(li));      //muestra en pantalla: sum = 6.0
System.out.println("sum = " + sumOfList(ld)); //muestra en pantalla: sum = 6.5
```

Ej: Método que imprime por pantalla los elementos de una lista de cualquier tipo

```
public static void printList(List<?> list) { //limite superior Object
    for (Object elem: list)
        System.out.print(elem + " ");
    System.out.println();
}
...
List<Integer> li = Arrays.asList(1, 2, 3);
List<String>  ls = Arrays.asList("uno", "dos", "tres");
printList(li); //muestra en pantalla: 1 2 3
printList(ls); //muestra en pantalla: uno dos tres
```

Si en vez de poner `List<?> list`

ponemos `List<Object> list`

No aceptaría como parámetros objetos de tipo

`List<Integer>` o `List<String>`

APENDICE 3: CÓMO JAVA IMPLEMENTA LOS GENÉRICOS

BORRADO DE TIPO EN LOS GENÉRICOS

Los genéricos se introdujeron en el lenguaje Java para proporcionar controles más estrictos de tipo en tiempo de compilación y para apoyar la programación genérica. **Para implementar los genéricos, el compilador Java realiza un borrado de tipos**: elimina los parámetros de tipo y reemplaza cada uno con su límite (si el parámetro de tipo está limitado con **extends**), o con **Object** si el parámetro de tipo no está limitado:

Ej: <T> → sustituye las T por **Object** <T **extends** Number> → sustituye las T por **Number**

Además:
-Inserte casting de tipo en caso necesario para preservar la seguridad de tipos.
-Genera métodos puente para preservar el polimorfismo en tipos genéricos extendidos.

Código original	Código tras el borrado de tipos

public class Node<**T**> {
  private **T** data;
  private Node<**T**> next;
  public Node(**T** data, Node<**T**> next) }
  this.data = data;
  this.next = next;
  }
  public **T** getData() { return data; }
  // ...
}
	public class Node {
	  private **Object** data;
	  private Node next;
	  public Node(**Object** data, Node next) {
	  this.data = data;
	  this.next = next;
	  }
	  public **Object** getData() { return data; }
	  // ...
	}

public class Node<**T extends Comparable<T>**> {
  private **T** data;
  private Node<**T**> next;
  public Node(**T** data, Node<**T**> next) {
  this.data = data;
  this.next = next;
  }
  public **T** getData() { return data; }
  // ...
}
	public class Node {
	  private **Comparable** data;
	  private Node next;
	  public Node(**Comparable** data, Node next) {
	  this.data = data;
	  this.next = next;
	  }
	  public **Comparable** getData() { return data; }
	  // ...
	}

// Contar el nº de ocurrencias de elem en anArray.
public static <**T**> int count(**T**[] anArray, **T** elem) {
  int cnt = 0;
  for (**T** e : anArray)
  if (e.equals(elem))
  ++cnt;
  return cnt;
}
	// Contar el nº de ocurrencias de elem en anArray.
	public static int count(**Object**[] anArray, **Object** elem) {
	  int cnt = 0;
	  for (**Object** e : anArray)
	  if (e.equals(elem))
	  ++cnt;
	  return cnt;
	}

class Shape { ... } class Circle extends Shape { ... } class Rectangle extends Shape { ... } public static <**T extends Shape**> void draw(**T** shape) { … }	public static void draw(**Shape** shape) { /* ... */ }

Los Genéricos (Plantillas) en Java no se implementan y funcionan igual que en C++
En C++ se crea una clase (o método genérico) por cada tipo que se instancie
En Java sólo se crea una clase (o método genérico) común para todos los tipos que se instancien

public static <**T**> void proceso(**T** t);	**Filosofía C++** (crea 3 métodos)	**Filosofía Java** (crea un solo método)

proceso(5);
proceso(5.67);
proceso(new Fecha(1,1,3));
	public static void proceso(**Integer** t);
	public static void proceso(**Double** t);
	public static void proceso(**Fecha** t);
		public static void proceso(**Object** t);