Tema 4: 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:

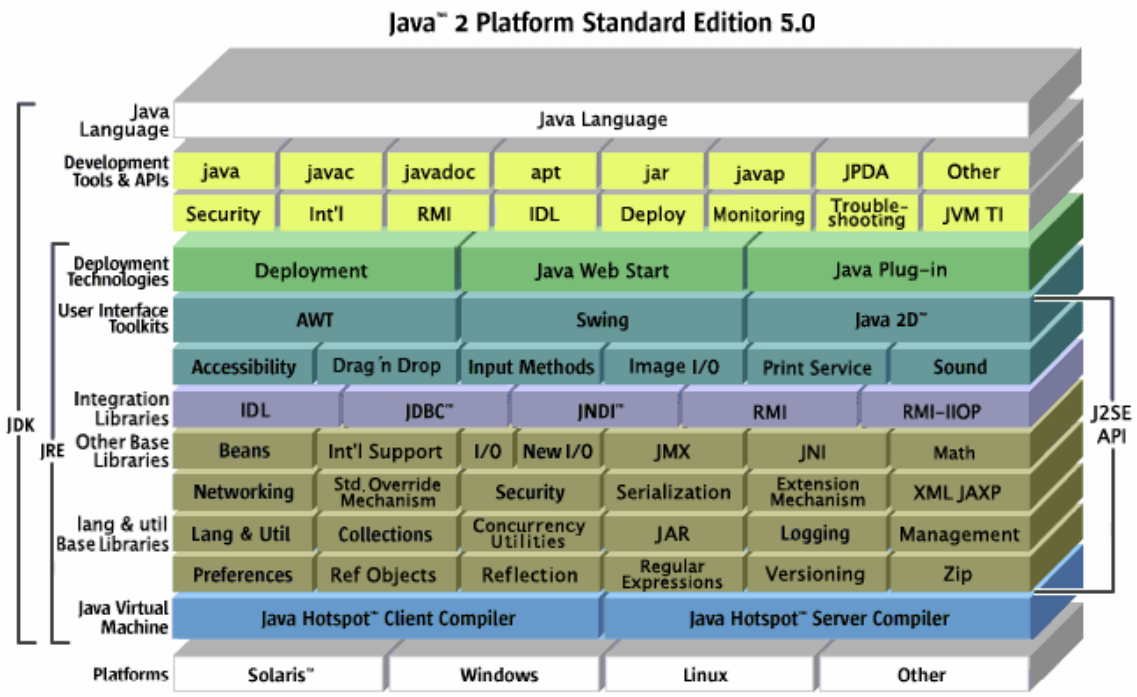
- Netbeans (java.sun.com o www.net-beans.org)
- Eclipse (IBM, www.eclipse.org)
- Blue J (www.bluej.org).

Entornos comerciales:

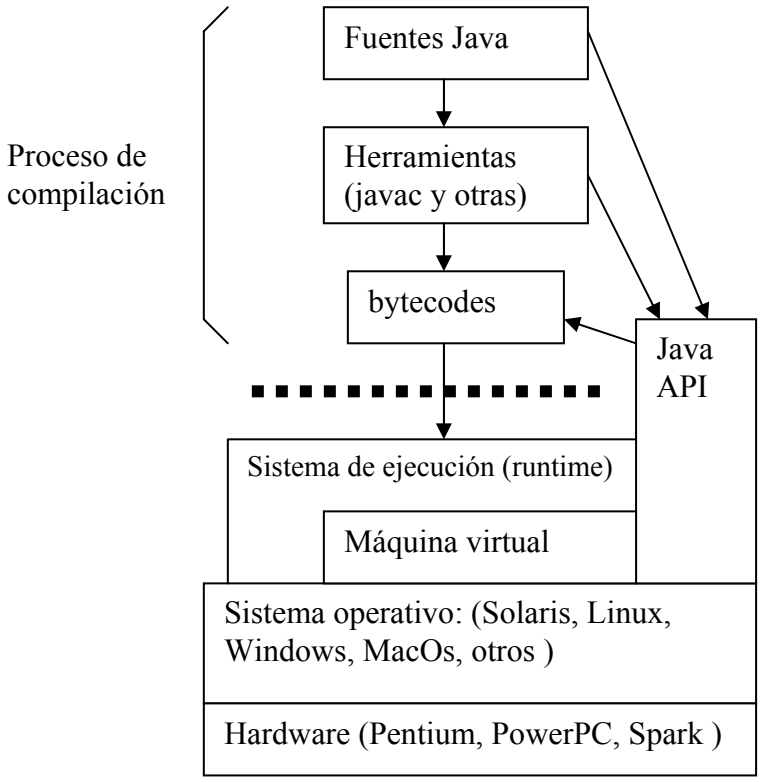
- JBuilder (Borland) (www.borland.com/products/downloads) con una versión (foundation) de uso gratuito.
- JCreator Pro (www.jcreator.com) con una versión (LE) gratuita.

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/1.5.0/docs/api/index.html



Esquema funcional de la plataforma Java 5:



Un **ejemplo**: Hola mundo…

C++

```
#include <iostream>

int main( int argc, char *argv ){

        std::cout << "Hola mundo";
}
```

Java

```
public class Programa{
        public static void main( String [] args ){

                System.out.println("Hola mundo");
        }
}
```

Notas:

- En Java, siempre se codifica dentro de clases: no hay funciones independientes como tales.
- Aunque no esté escrito explícitamente, la clase Programa hereda de Object. En Java, todos las clases descienden de 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 par que el compilador lo acepte.

Comentarios:

/* */

Por convención, para los multilínea:

```
/**
 *
 *
 */
```

Comentarios de una linea
//

Características

El lenguaje C++ fue un intento de tomar los principios de la programación estrucurada y emplearlos dentro de las restricciones de C. Todos los compiladores de C++ eran capaces de compilar programas de C sin clases,

Java utiliza convenciones casi idénticas para declaración de variables, paso de parámetros, y demás, pero sólo considera las partes de C++ que no estaban ya en C.

Las principales características que Java no hereda de C++ son:

- **Punteros:**
  - **El inadecuado uso de los punteros provoca la mayoría de los errores de colisión de memoria**, errores muy **difíciles de detectar**.
- **Variables globales**: **Efectos laterales**, fallos catastróficos En **Java lo único global es el nombre de las clases**.
- ***goto*:** Java tiene las sentencias *break* y *continue* que cubren los casos importantes de *goto*.
- **Asignación de memoria**: Java obtiene con *new* un descriptor al objeto del ***montículo***. La **memoria real** asignada a ese objeto se puede mover a la vez que el programa se ejecuta, pero sin tener que preocuparse de ello. Cuando no tenga **ninguna referencia de ningún objeto**, la memoria ocupada estará disponible. A esto se le llama ***recogida de basura***. El recolector de basura se ejecuta siempre que el **sistema esté libre**, o cuando una asignación **solicitada no encuentre asignación suficiente**.
- **Conversión de tipos insegura**: (*type casting*) no hay nada previsto para detectar si la conversión es correcta **en tiempo de ejecución**. En Java se puede hacer una **comprobación en tiempo de ejecución** de la compatibilidad de tipos y emitir una excepción cuando falla.

Objetos, miembros y referencias

El operador new

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

MiPunto p2 = new MiPunto();

ejemplo:

```
Punto p;
p = new Punto();
```

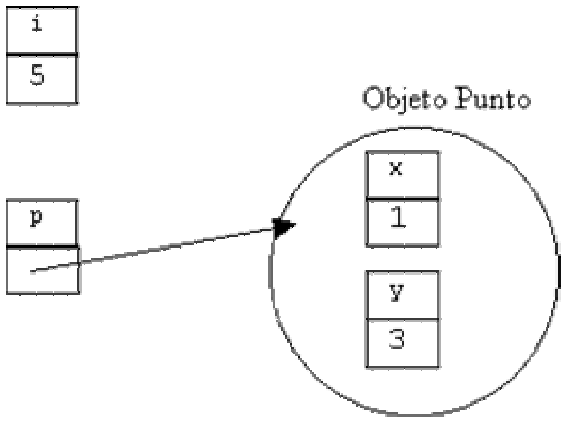
La primera línea del ejemplo declara una referencia (p) que es de Tipo Punto. La referencia no apunta a ningún sitio. En la segunda línea se crea un objeto de Tipo Punto y se hace que la referencia p apunte a él. Se puede hacer ambas operaciones en la misma expresión:

```
Punto p = new Punto();
```

A los miembros de un objeto se accede a través de su referencia.

- Referencias y objetos:

```
Punto a ,p = new Punto();
a=p;
p.x = 1;
p.y = 3;

int i = 5;
```




```
class Punto{int  x, y;};

class Circulo {
      Punto centro;   // dato miembro. Referencia a un objeto punto
      int radio;      // dato miembro. Valor primitivo
      float superficie() {              // método miembro.
            return 3.14 * radio * radio;
      }                                 // fin del método superficie


public static void main (String [] args){
          Circulo c = new Circulo();
          c.centro = new Punto();
          c.centro.x = 2;
          c.centro.y = 3;
          c.radio = 5;
          float s = c.superficie();
 } // fin de la clase Circulo
```

Referencia a un miembro de tipo Objeto e inicialización

Contructores

Ejemplo constructor con parametros:

```
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**
- Si se **declara algún constructor,** entonces ya no se puede usar el constructor no-args.

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

En Java no existen los parametros por defecto → Sobre carga de constructores

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

Uso de This

Para referenciar al propio constructor

```
MiPunto() {

  this( -1, -1 ); // Llama al constructor parametrizado

}
```

Para delimitar ambigüedad

```
class ThisUso {
        protected int x, y;

        public ThisUso(int x, int y) {
                        this.x = x;
                        this.y = y;
        }

};
```

Para enviar referencia de objeto propio

```
class ThisUso {
        protected int x, y;

        public ThisUso menor(ThisUso otro){
                        if (x < otro.x && y < otro.y )
                                        return this;
                        else
                                        return otro;

        }
```

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 peroa en ningún caso se permite el acceso de una clase a los miembros privados de otra.

```
public class HelloWorld {
    char depaquete;
    protected int protegido=1;
    public String publica="Hola";
    private double privada;
    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.

Las clases pueden ser **públicas** o de **paquete** (por defecto)

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 metodo 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
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

Los **nombres de paquetes** deben ser únicos (y por convenio en minúsculas y separados por _ cuando hay varias palabras , a tal fin es comun tener nombres de paquete con el dominio de internet invertido:

ej: es.uhu.mipaquete

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**

Ojo : Todas las clases en el mismo paquete tienen acceso a los miembros declarados de paquete (por defecto) a todas las clases no declaradas como publicas.

Referencia a otros Paquetes: Import

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

import <nombre de paquete>.{<clase>|*};

→ *, indica todas las clases incluidas en el paquete;
→ Subpaquetes. Se utiliza el punto para generar subdirectorios:

```
import java.util.Scanner;
import java.io.*;
```

Colisiones

Paquete librería y paquete1 contienen la clase Punto

```
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 patch completo la
otra.
           Punto p=new Punto();
           String c="hola";
           Prueba b=new Prueba();


        }
}
```

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
}
}
```

Depuración en Java

Java no tiene compilación condicional --> en C++ era utilizado para compatibilidad de plataformas , en java no es necesario.

Incluyendo clases con código de depuración y sustituyendo esas clases cuando el sistema funcione correctamente.

import es.uhu.debug;
// import es.uhu.real;

Ejemplos de utilización

- Heredar todas las clases de una aplicación de una clase común donde se incluyen comentarios en los constructores y llamadas desde el código de las clases derivadas. Cuando se comente el import adecuado desaparecerá el código de dicha clase.

```
package depuracion;

public class Base {
  static int num;

  public Base(){num_instancias++;}
    public void trace(Object obj, String traza){
        System.out.println("Ins:  " + num + " objeto : " + obj + " Traza :" + traza );
  }
  public static int instancias(){ return num;}
  public void finalize(){
              num --;
  }
 }
```

```
package real;
public class Base {
   public void trace(Object obj, String traza){ }
}
```

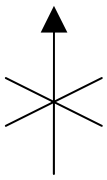
```
package pruebadepuracion;
import depuracion.*;
//import real.*;
public class MiClase extends Base{

public MiClase(){
   trace (this,"constructor de mi clase");
}
public String toString (){
              return ("Mi Clase");
  }
  public static void main(String[] args) {
              MiClase mia = new MiClase();
        mia.trace(mia, "ahora utilizo la clase");
        for (int i = 1;i < 10;i++)
              mia = new MiClase();
        System.out.println("\n Fin de Programa");
  }
}
```

Salida del Programa en modo depuración:

```
Instancias:  1 objeto : Mi Clase Traza :constructor de mi clase
Instancias:  1 objeto : Mi Clase Traza :ahora utilizo la clase
Instancias:  2 objeto : Mi Clase Traza :constructor de mi clase
Instancias:  3 objeto : Mi Clase Traza :constructor de mi clase
Instancias:  4 objeto : Mi Clase Traza :constructor de mi clase
Instancias:  5 objeto : Mi Clase Traza :constructor de mi clase
Instancias:  6 objeto : Mi Clase Traza :constructor de mi clase
Instancias:  7 objeto : Mi Clase Traza :constructor de mi clase
Instancias:  8 objeto : Mi Clase Traza :constructor de mi clase
Instancias:  9 objeto : Mi Clase Traza :constructor de mi clase
Instancias:  10 objeto : Mi Clase Traza :constructor de mi clase

 Fin de Programa
```

Un Ejemplo de agregación

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



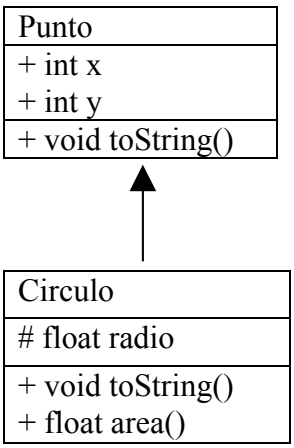
Circulo
+ **Punto centro** + double radio
+ Circulo(Punto np, double r) + String toString()

```
class Punto{

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

class Circulo{

        public Punto p;
        public double radio;
        public Circulo( Punto np, double r ){
                // Ojo, se copia la referencia
                p = np;
                radio = r;
        }
        public String toString(){
                return new String( "("+p.x+","+p.y+"):"+radio );
        }
}

public class Programa {

        public static void main(String[] args){

                Punto p=new Punto(10.5,6.2);
                Circulo c=new Circulo(p,2.45); //Ojo, se copia la ref

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

Notas:

Uso de toString → sobrecarga de String ofrecida por Java
Uso de las referencias en los parámetros
Casting automatico a String con el operador +
Clases públicas vs de paquete

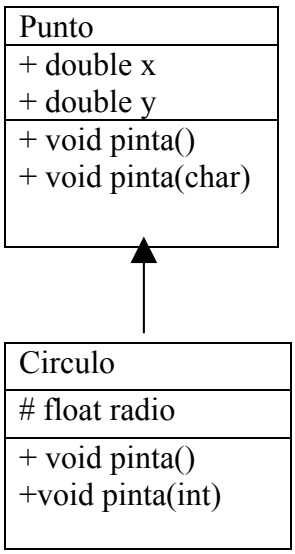
Herencia simple:




```
class Punto{

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

class Circulo extends Punto{


        public double radio;
        public Circulo(  double nx, double ny, double r ){

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

public class Herencia{

        public static void main(String[] args){

                Punto p=new Punto(10.5,6.2);
                Circulo c=new Circulo(1,2,2.45); //Ojo, se copia la ref

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

SobreCarga de metodos Heredados

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

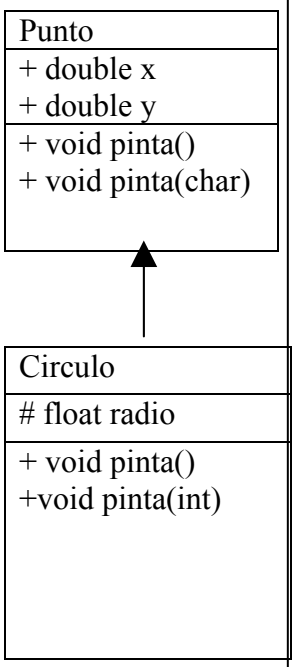



```
Public class Punto{

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

Salida

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

Metodos heredados. Acceso a la Superclase



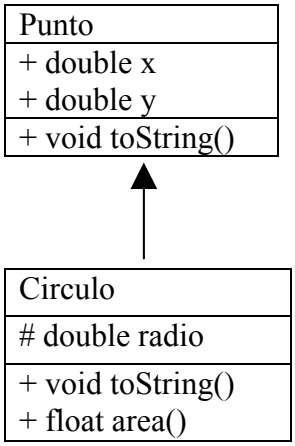

```
Public class Punto{
        public double x;
        public double y;
        int dePaquete;
        public Punto( double nx, double ny){
                x=nx;
                y=ny;
        }
        public void pinta(){
                System.out.println( " Punto >> " + this +"\n");
        }
 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


class Circulo extends Punto{
        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){
                Punto p=new Punto(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 Punto();

Salida

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

Polimorfismo: Up-casting y Down-casting




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

class Circulo extends Punto{
        public double radio;
        public double area(){return 3.14*radio;}
        public Circulo(  double nx, double ny, double r ){
                super(nx,ny);
                radio = r;
        }
        public String toString(){
                return new String( "Circulo("+x+","+y+"):"+radio );
        }
}

public class Polimorfismo{

        public static void main(String[] args){

                Punto p=new Punto(10.5,6.2);
                Circulo c1,c=new Circulo(1,2,2.45);
                System.out.println( p +"-->"+ c );
                p= c;
                System.out.println( "Polimorfico " + p +"-->"+ c );
                //p.area(); Error de compilación
                c1=(Circulo)p;
                System.out.println( "área " + c1.area() );
        }
}
```

Es necesario hacer un casting para poder utilizar un método de una clase derivada

Salida

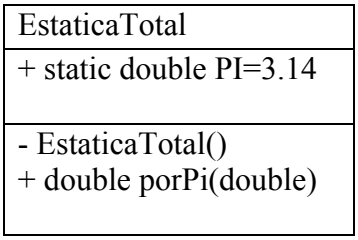
```
Punto(10.5,6.2)-->Circulo(1.0,2.0):2.45
Polimorfico Circulo(1.0,2.0):2.45-->Circulo(1.0,2.0):2.45
área 7.693
```

Miembros estáticos

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 ++;

         }
   public void finalize (){
         num--;
   }
   public static void main(String[] args) {
                  Estatica.num = 5;
                  Estatica e= new Estatica();
                  System.out.println(e.getNum());
                  }
```

No instanciable




```
public class EstaticaTotal {

        public static double PI =3.14;
        public static double porPi(double a){
                return PI*a;
        }
        private EstaticaTotal() {
        }
}

public class NoInstanciable{

        public static void main(String[] args) {
                EstaticaTotal instancia;
                //instancia= new EstaticaTotal();
                //No se puede instanciar
                System.out.println(EstaticaTotal.porPi(3));


        }

}
```

Static: bloque static:

En ocasiones es necesario realizar cierta computación antes de empezar a utilizar una clase. El **caso** más **común** es que sea necesaria computación para **inicializar los atributos de clase (estáticos)**. Esta computación se puede explicitar en un **bloque estático**.

La sintaxis es, dentro de la clase:

```
static{ … }
```

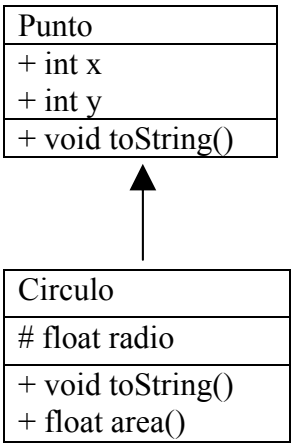
Punto
- double x - double y
+ Punto(double nx, double ny) + public String toString()

```
package principal;

class Punto{

        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 ClaseConArray{
        public static Punto [] array = new Punto[5];
        static{
                System.out.println( "Ejecución del bloque estático" );
                for (int i=0; i<5; i++)
                        array[i] = new Punto(0,0);
        }
        /* ... Resto de implementación de la clase ... */
}

public class Programa {

        public static void main(String[] args){

                Punto p = new Punto(10,10);
                System.out.println( p );
                /* Asignamos el primer punto del array */
                ClaseConArray.array[0] = p;
                /* Mostramos los puntos que hay en el array */
                for ( int i=0; i<5; i++ )
                        System.out.println( ClaseConArray.array[i] );
        }
}
```

Salida:

```
(10.0,10.0)
Ejecución del
bloque estático
(10.0,10.0)
(0.0,0.0)
(0.0,0.0)
(0.0,0.0)
(0.0,0.0)
```

Clase Abstracta




```
abstract class Punto{

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

class Circulo extends Punto{


        protected 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 );
        }
}

public class Abstract{

        public static void main(String[] args){

                Punto p;// =new Punto();//no se puede
                Circulo c=new Circulo(1,2,2.45); /
                System.out.println(p.toString());
                System.out.println(c );
        }
}
```

Herencia de Abstracta

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

sólo puede haber métodos abstractos en clases abstractas

```
abstract class Punto{

        public double x;
        public double y;

        public String toString(){
                return "";
        };
        public abstract void  pinta();
}
The type Circulo must implement the inherited abstract method Punto.pinta()
class Circulo extends Punto{


        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 );
        }
}
```

Interfaces:

- Los interfaces son **una agrupación de métodos y constantes públicas**.
- Las **clases pueden "comprometerse" a implementar interfaces** (uno o varios).
- 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 el interfaz**.
- Los **interfaces también heredan** (extienden) **entre ellos**.

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

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

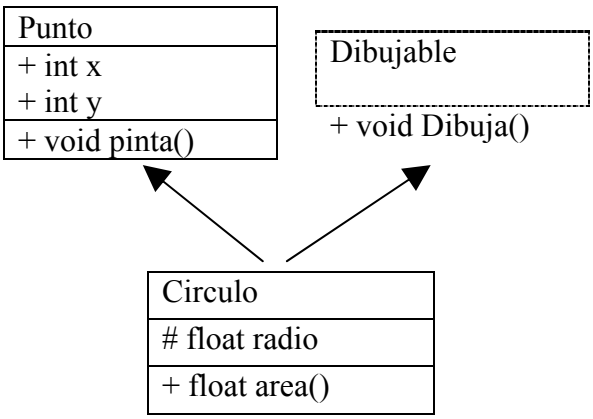
```
interface IntMul extends Int1, Int2{ … }
```

- Se pueden **declarar variables** dando como **clase** un **interfaz**.

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

- **Al poder declarar una variable como de un interfaz**, en realidad e**xpresamos 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.

Interfaces vs Herencia múltiple:




```
package Interfaces;
 class Punto{

        public double x;
        public double y;

        public void pinta(){
                System.out.print( "("+x+","+y+")");
        }
}
interface Dibujable{
        void dibuja();
        final double constante = 3.1416;
        }
class Circulo extends Punto 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.constante;
        }

        public static void main(String[] args){

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

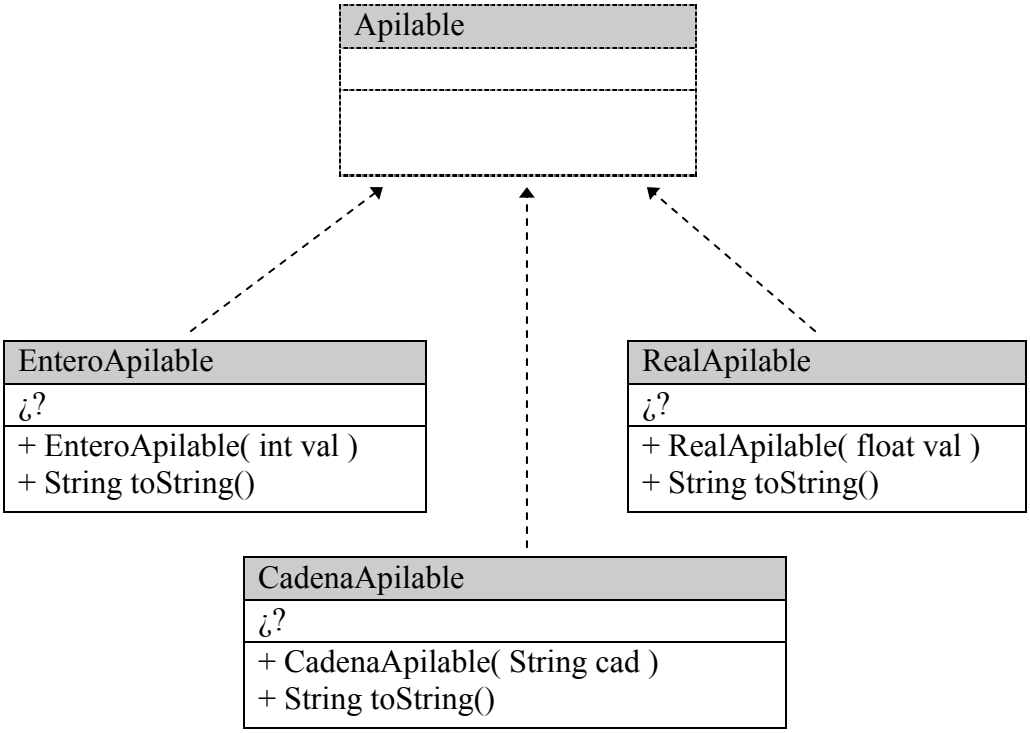
Agrupación de Constantes

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

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

Interfaces: marcado

Una utilización de los interfaces es **marcar** a **determinadas clases como clases sobre las que se puede llevar a cabo una determinada computación**, **sin necesidad de concretar ninguna constante ni método.**

Basta con declarar un **interfaz vacío**.



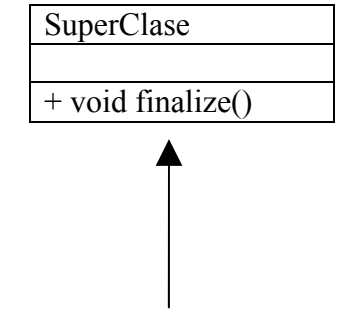
Destructores:

- **En Java no existen los destructores como tales**. Existe un **método definido en** la clase **Object** que es **llamado por el recolector de basura cuando detecta que ese objeto no está referenciado**. El método **public void finalize()**.

- **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 si 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 desde la subclase hacia las superclases. Esto es diferente en Java: sólo se llama al finalize de la clase que se elimina.



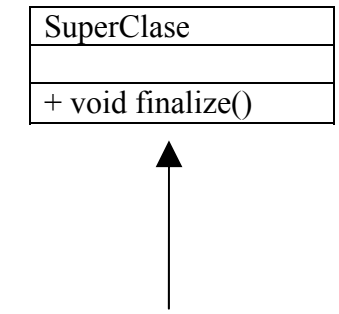

```
package principal;

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

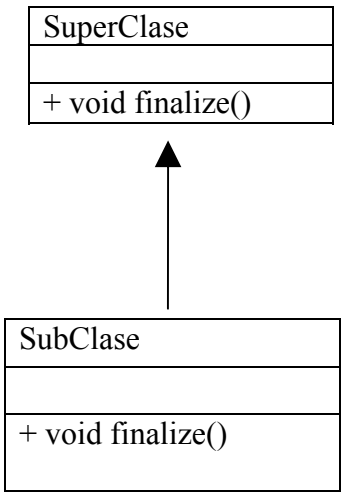
```
package principal;

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

Destructores:

La superclase es una parte integrante del objeto. Si se mantiene una referencia la superclase, el objeto concreto sigue referenciado:

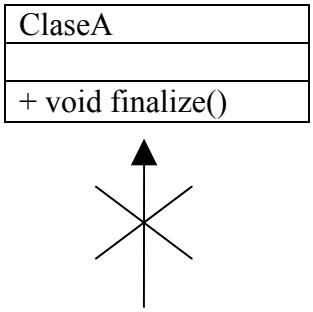



```
package principal;

class SuperClase{

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

class SubClase extends SuperClase{

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

public class Programa {

        public static void main(String[] args){

                SubClase sub = new SubClase();
                SuperClase sup = sub.ref();

                sub = null;
                System.out.println("Primera llamada");
                System.gc();
                sup = null;
                System.out.println("Segunda llamada");
                System.gc();
        }
}
```

```
Salida:

Primera llamada
Segunda llamada
Destruyendo!Sub
```

Destructores:

Otro caso se presenta **cuando un objeto tiene un atributo de otra clase**:

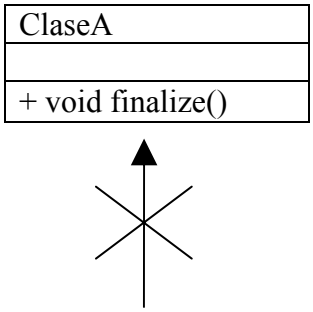



```
package principal;

class ClaseA{

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

class ClaseB{

        public ClaseA atributo;

        public ClaseB(){ atributo = new ClaseA(); }

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

public class Programa {

        public static void main(String[] args){

                ClaseB b = new ClaseB();
                b = null;
                System.gc();
        }
}
```

```
Salida:

Destruyendo! ClaseA
Destruyendo! ClaseB
```

Destructores:

Otro caso se presenta cuando un objeto tiene un atributo de otra clase:

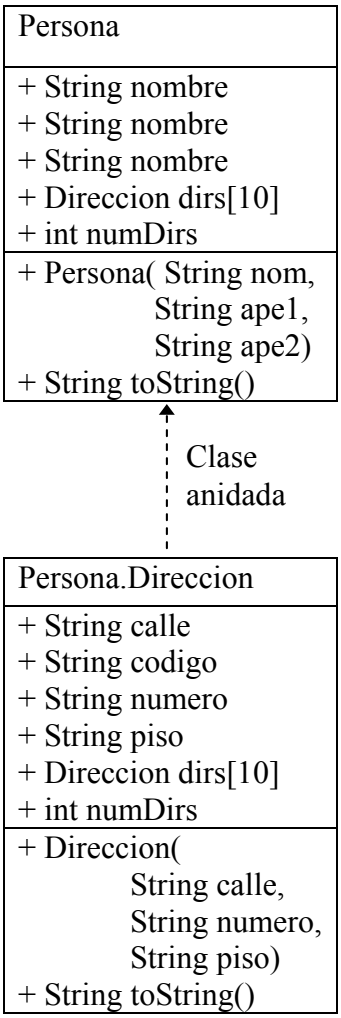



```
package principal;

class ClaseA{

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

class ClaseB{

        public ClaseA atributo;

        public ClaseB(){ atributo = new ClaseA(); }

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

public class Programa {

        public static void main(String[] args){

                ClaseB b = new ClaseB();
                ClaseA a = b.atributo;
                b = null;
                System.out.println( "Primera llamada" );
                System.gc();
                a = null;
                System.out.println( "Segunda llamada" );
                System.gc();
        }
}
```

```
Salida:

Primera llamada
Destruyendo! ClaseB
Segunda llamada
Destruyendo! ClaseA
```

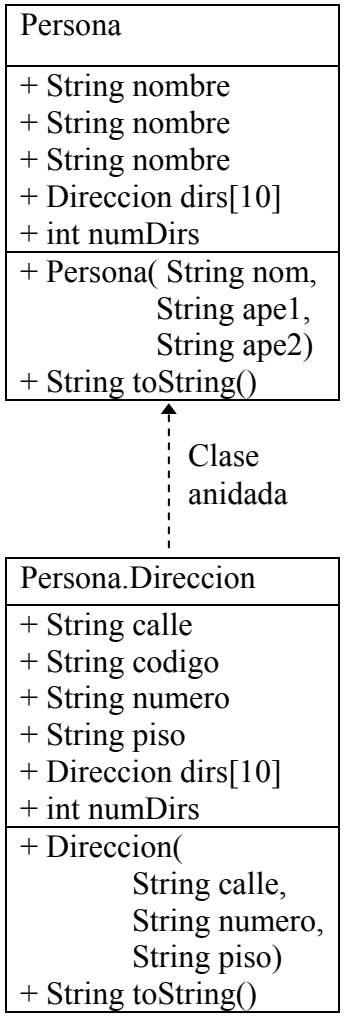
Ejemplo de uso : programación de un buffer de salida. Si existe en memoria un buffer con información almacenada que debe enviarse a una salida (archivo, flujo, …), si se eliminan todas las referencias y no hemos programado un finalizador, esa información será liberada sin ser enviada al archivo o flujo correspondiente. Sería conveniente añadir en el método de finalización una llamada a un vaciado del buffer .

Clases internas o anidadas




```
import java.util.Scanner;

class Persona{
        public String nombre;
        public String apellido1;
        public String apellido2;
        public Direccion [] dirs = new Direccion[10];
        public int numDirs = 0;
        public Persona(   String nom,
                          String ape1,
                          String ape2 ){

                nombre = nom;
                apellido1 = ape1;
                apellido2 = ape2;
        }

        public Persona(   String nom, String ape1, String ape2,
                          Direccion dir){
                this( nom,ape1,ape2 );
                dirs[numDirs++]=dir;
        }

        /* Clase anidada */
        public class Direccion{
                public String calle;
                public String codigo;
                public int numero;
                public String piso;
                public Direccion( String nc, int nn, String np ){
                        calle = nc;
                        numero = nn;
                        piso = np;
                        codigo = new String( ""+nombre.charAt(0)
                                        +apellido1.charAt(0)
                                        +apellido2.charAt(0)
                                        +calle.charAt(0)
                                        +numero );
                }
                public String toString(){
                        return new String( codigo+":"+calle+",nº: "
                                        +numero+" piso: "+piso );
                }
        }
        /* Fin de clase anidada */

        public void aniadirDir( Direccion dir ){
                if (numDirs<10)
                        dirs[numDirs++]=dir;
        }

        public String toString(){
                return new String( ""+nombre+" "+apellido1
                                                +" "+apellido2 );
        }
}
```

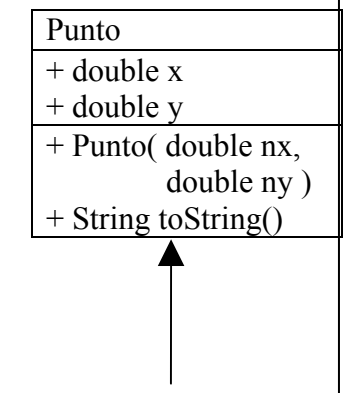
Clases internas o anidadas




```
public class Programa {

        public static void main(String[] args){

                Scanner teclado = new Scanner( System.in );
                System.out.println( "Introduzca nombre y apellidos" );
                Persona p = new Persona( teclado.nextLine(),
                                         teclado.nextLine(),
                                         teclado.nextLine());
                System.out.println( "¿Cuántas direcciones?" );
                for (  int i = Integer.parseInt(teclado.nextLine());
                       i>0; i-- ){
                    Persona.Direccion dir;
                    System.out.println( "Introduzca una dirección
(calle, número, piso):" );
                    dir = p.new Direccion( teclado.nextLine(),
                            Integer.parseInt(teclado.nextLine()),
                            teclado.nextLine());
                    p.aniadirDir( dir );
                }

                System.out.println( "Persona introducida:\n"+p );
                if ( p.numDirs > 0 ){
                    Persona.Direccion tmp; // Sólo por ejemplo
                    for ( int i=0; i<p.numDirs ; i++ ){
                            tmp = p.dirs[i]; // Sólo por ejemplo
                            System.out.println( "\t" + tmp );
                    }
                }else
                    System.out.println( "No se conocen
direcciones" );
        }
}
```

Información de clase en tiempo de ejecución: Class e instanceof

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



Caso incorrecto

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

```
package principal;

class Punto{

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

class Circulo extends Punto{

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

public class Programa {

        public static void main(String[] args){

                /*
                 * El casting "hacia abajo" es útil y
                 * válido pero es potencielmente peligroso
                 */
                Punto p;
                Circulo c;
                /* ¡¡Caso incorrecto!!: ¡lanza una excepción que no
                 * se trata y terminar!
                 */
                p = new Punto(0,0);
                c = (Circulo)p;
                System.out.println( c );
        }

}
```

```
Salida:

Exception in thread "main" java.lang.ClassCastException:
principal.Punto
      at principal.Programa.main(Programa.java:42)
```

Información de clase en tiempo de ejecución: Class e instanceof

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

Caso correcto

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
                 */
                Punto p;
                Circulo c;
                /* ¡¡Caso correcto!! */
                p = new Circulo(0,0,10);
                c = (Circulo)p;
                System.out.println( c );
        }

}
```

```
Salida:

(0.0,0.0)
```

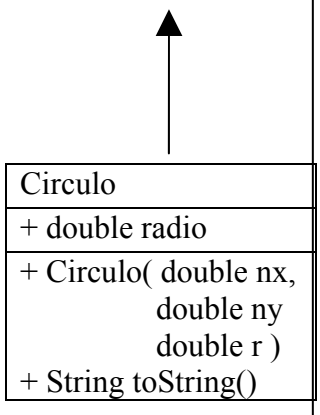
Solución: asegurarnos de que hacemos un casting válido

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
                 */
                Punto p;
                Circulo c;
                /* ¡¡Caso correcto!! */
                p = new Circulo(0,0,10);

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

}
```

Información de clase en tiempo de ejecución: Class e instanceof

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.




```
package principal;

class Punto{

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

class Circulo extends Punto{

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

public class Programa {
        public static void main(String[] args){

                /*
                 * El casting "hacia abajo" es útil y
                 * válido pero es potencialmente peligroso
                 */
                Punto p;
                Circulo c;
                /* ¡¡Caso incorrecto!!: ¡lanza una excepción que no
                 * se trata y terminar!
                 */
                p = new Circulo(0,0,1);
                c = new Circulo(1,2,2.3);

                System.out.println("p:" + p.getClass()
                                +"\nc:" + c.getClass());

                if ( c.getClass().isInstance( p ) ){
                        System.out.println( "p contiene algo que es de "+
                                        "la misma clase que c o "+
                                        "subclase de la clase de c");
                }else{
                        System.out.println("p no contiene algo de clase "+
                                        "c ni subclase de esta");

                }
        }

}
```

Static import: (Java 5)

Construcción que sirve para hacer más cómodo el acceso a métodos de clase (static). Utilizando static import evitamos tener que escribir el nombre de la clase delante de los métodos

Ejemplo: math.

Java 1.4

```
package principal;

import java.lang.Math; // En realidad no es necesario por ser lang

public class Programa {

        public static void main(String[] args) {

                double d = Math.sqrt( 4.0 ) * Math.PI;

                System.out.println( d );
        }
}
```

Java 5

```
package principal;

import static java.lang.Math.*;

public class Programa {

        public static void main(String[] args) {

                double d = sqrt( 4.0 ) * PI;

                System.out.println( d );
        }
}
```

Excepciones en Java

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" );
        }
}
```

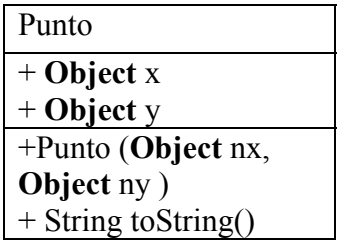
Genéricos (Java 5)

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…)




```
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 p = new Punto( new Integer(1), new
Integer(2) );
                System.out.println( p );
                Integer x = (Integer)p.x;
        }
}
```

Genéricos (Java 5)

Punto genérico hasta Java 5

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

```
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> p = new Punto<Integer>( new
Integer(1), new Integer(2) );
                System.out.println( p );
                Integer x = p.x;
        }
}
```

Genéricos (Java 5)

Ejemplo: cola genérica utilizando genéricos. (Se puede hacer utilizando un array de Object)

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()

```
package principal;

class Punto{

        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>{

        private Object array[];
        private int elementos=0;
        private int tamaño=0;
        public Cola( int tam ){
                tamaño = tam;
                array = new Object[tamaño];
        }
        public void encolar( T e ){
                if ( elementos < tamaño )
                        array[elementos++] = (Object)e;
        }
        public T desencolar(){
                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] = null;
                        elementos--;
                        return tmp;
                }
                return null;
        }
        public T foco(){
                if ( elementos > 0 )
                        return (T)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()

```
public class Programa {

        public static void main(String[] args){

                Cola<Integer> cint = new Cola<Integer>(10);
                Cola<Punto> cpunto = new Cola<Punto>(10);

                for ( int i=0; i<5; i++ ){
                        cint.encolar( new Integer(i) );
                        cpunto.encolar( new Punto(i,i) );
                }
                while( cint.encolados() > 0 )
                        System.out.println( cint.desencolar() );

                while( cpunto.encolados() > 0 )
                        System.out.println( cpunto.desencolar() );
        }
}
```

Interfaces y genéricos:Comparable

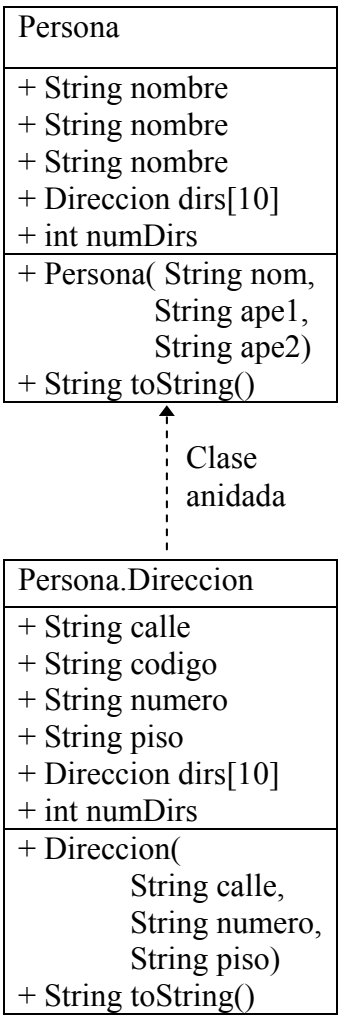
Se pueden definir interfaces en base a genéricos:

Comparable<**Q**>
+ boolean mayor (Q otro) + boolean menor (Q otro) + boolean igual (Q otro)

Punto1D
- int x
+Punto1D(int nx) + String toString() + boolean mayor (Punto1D otro) + boolean menor (Punto1D otro) + boolean igual (Punto1D otro)

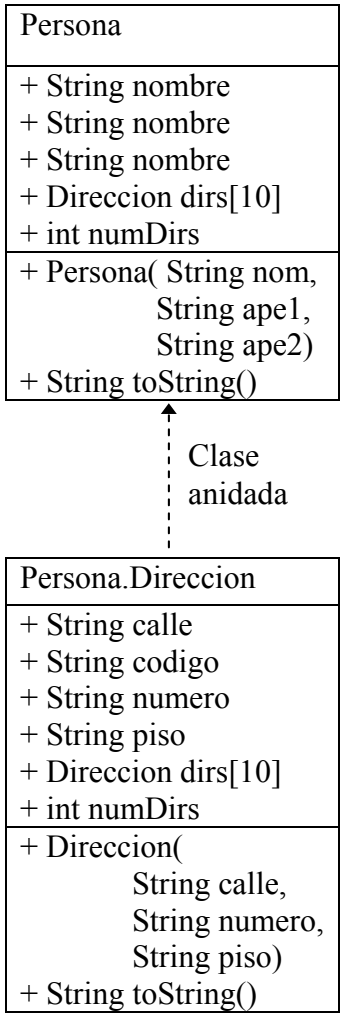
```
package principal;

import java.util.Random;

interface Comparable<Q>{
        boolean mayor( Q otro );
        boolean menor( Q otro );
        boolean igual( Q otro );
}

class Punto1D implements Comparable<Punto1D> {

        public int x;
        public Punto1D( int nx){
                x = nx;
        }
        public String toString(){
                return new String( "("+x+")" );
        }
        /* Métodos del interfaz */
        public boolean mayor( Punto1D otro ){
                return x > otro.x;
        }
        public boolean menor( Punto1D otro ){
                return x < otro.x;
        }
        public boolean igual( Punto1D otro ){
                return (otro.x==x);
        }
}

public class Programa {

        public static void main(String[] args){

                Comparable array[] = new Comparable[10];
                Random r = new Random();
                for (int i=0; i<10; i++)
                        array[i] = new Punto1D( r.nextInt()%100 );
                for (int i=0; i<10; i+=2)
                        System.out.println( array[i]
                          +((array[i].mayor(array[i+1]))?">":"<=")
                          + array[i+1]);
        }
}
```

Clases internas o anidadas




```
import java.util.Scanner;

class Persona{
        public String nombre;
        public String apellido1;
        public String apellido2;
        public Direccion [] dirs = new Direccion[10];
        public int numDirs = 0;
        public Persona(   String nom,
                          String ape1,
                          String ape2 ){

                nombre = nom;
                apellido1 = ape1;
                apellido2 = ape2;
        }

        public Persona(   String nom, String ape1, String ape2,
                          Direccion dir){
                this( nom,ape1,ape2 );
                dirs[numDirs++]=dir;
        }

        /* Clase anidada */
        public class Direccion{
                public String calle;
                public String codigo;
                public int numero;
                public String piso;
                public Direccion( String nc, int nn, String np ){
                        calle = nc;
                        numero = nn;
                        piso = np;
                        codigo = new String( ""+nombre.charAt(0)
                                        +apellido1.charAt(0)
                                        +apellido2.charAt(0)
                                        +calle.charAt(0)
                                        +numero );
                }
                public String toString(){
                        return new String( codigo+":"+calle+",nº: "
                                        +numero+" piso: "+piso );
                }
        }
        /* Fin de clase anidada */

        public void aniadirDir( Direccion dir ){
                if (numDirs<10)
                        dirs[numDirs++]=dir;
        }

        public String toString(){
                return new String( ""+nombre+" "+apellido1
                                                +" "+apellido2 );
        }
}
```

Clases internas o anidadas




```
public class Programa {

        public static void main(String[] args){

                Scanner teclado = new Scanner( System.in );
                System.out.println( "Introduzca nombre y apellidos" );
                Persona p = new Persona( teclado.nextLine(),
                                         teclado.nextLine(),
                                         teclado.nextLine());
                System.out.println( "¿Cuántas direcciones?" );
                for (  int i = Integer.parseInt(teclado.nextLine());
                       i>0; i-- ){
                    Persona.Direccion dir;
                    System.out.println( "Introduzca una dirección
(calle, número, piso):" );
                    dir = p.new Direccion( teclado.nextLine(),
                            Integer.parseInt(teclado.nextLine()),
                            teclado.nextLine());
                    p.aniadirDir( dir );
                }

                System.out.println( "Persona introducida:\n"+p );
                if ( p.numDirs > 0 ){
                    Persona.Direccion tmp; // Sólo por ejemplo
                    for ( int i=0; i<p.numDirs ; i++ ){
                            tmp = p.dirs[i]; // Sólo por ejemplo
                            System.out.println( "\t" + tmp );
                    }
                }else
                    System.out.println( "No se conocen
direcciones" );
        }
}
```

Object (1) (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 llama 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 ){
                Punto otro = (Punto)o;
                return (Math.abs(otro.x - x) < 1e-10)
                        &&(Math.abs(otro.y - y) < 1e-10);
        }

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

Object (2) (java.lang.object)

```
public class Programa {

	public static void main(String[] args) throws Exception {
		// Lectura de un punto:
		Scanner s = new Scanner( System.in );
		s.useLocale( Locale.ENGLISH );
		System.out.println( "Coordenadas del primer punto" );
		Punto p1 = new Punto( s.nextDouble(),
				s.nextDouble());
		System.out.println( "Coordenadas del segundo punto" );
		Punto p2 = new Punto( s.nextDouble(),
				 s.nextDouble());
		System.out.println( p1 + (p1.equals(p2)?"==":"!=") + p2);
		Punto p3 = (Punto)p2.clone();
		System.out.println( p2 + (p2.equals(p3)?"==":"!=") + p3);

		p3.x=0;
		p3.y=0;
		System.out.println( p2 + (p2.equals(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);
	}
}
```

StringBuffer (java.lang.StringBuffer)

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

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 o la cadena de representación del argumento.

***insert* inserta** la cadena o representación pasada como segundo argumento **en la posición especificada** en el primer argumento.

Son también útiles ***length***, ***setLength*** y ***subString***.

```
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
```

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.

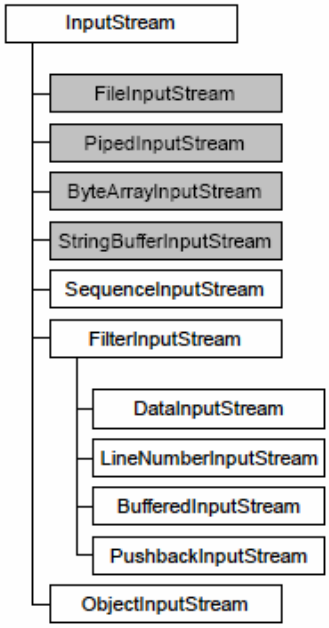


Figura 9.1. Jerarquía de clases InputStream.

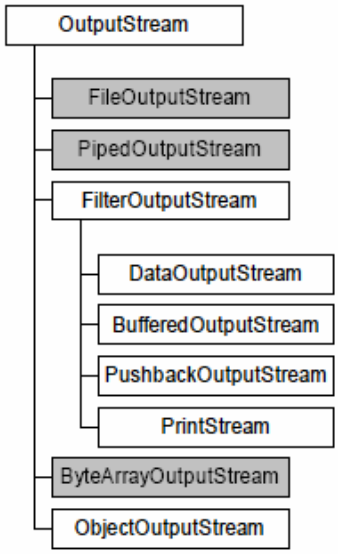


Figura 9.2. Jerarquía de clases OutputStream.

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

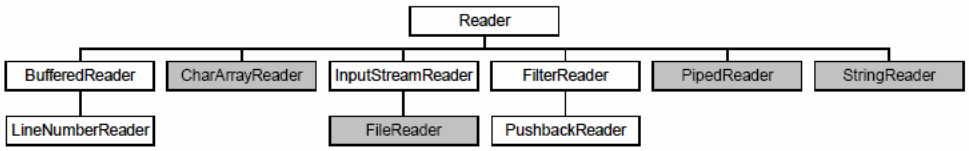
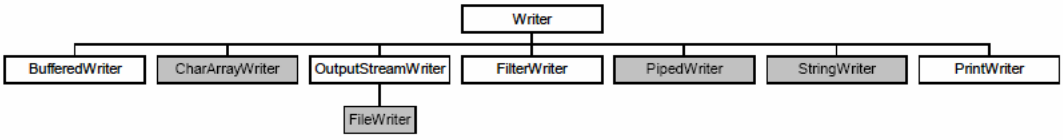


Figura 9.3. Jerarquía de clases Reader.



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:

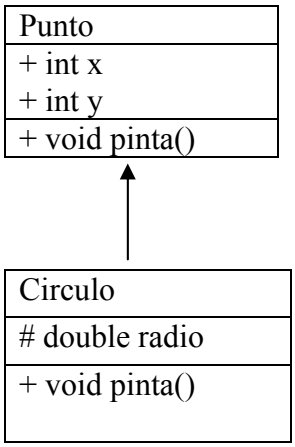
- Las variables y objetos ***static*** no son serializados.
- ***transient*** permite indicar que un objeto o variable miembro no sea serializado.
  - Al recuperarlo, lo que esté marcado como ***transient*** será ***0***, ***null*** o ***false***

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 Punto implements Serializable{
        public double x;
        public double y;

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

class Circulo extends Punto{
        public double radio;
        Punto miPunto;
        public Circulo(  double nx, double ny, double r ){
                super(nx,ny);
                radio = r;
                miPunto = new Punto(1,1);
        }
        public void pinta(){
                System.out.println( "Circulo >> " + this +"\n");
                miPunto.pinta();
        }
        public static void main(String[] args){
                Punto p=new Punto(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.miPunto);
                        out.writeObject(p);
                //Lectura
                        FileInputStream fis = new FileInputStream("fichero.bin");
                        ObjectInputStream in = new ObjectInputStream(fis);
                        c.miPunto = p;
                        c = (Circulo)in.readObject();
                        c.miPunto = (Punto)in.readObject();
                        p = (Punto)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();
        }
}
```