Tema 4: Java

Conceptos y revisión de sintaxis

http://www.arrakis.es/~abelp/ApuntesJava/indice.htm
http://java.sun.com/docs/books/tutorial/java/TOC.html

Declaración de una clase básica:

```
[public][abstract][final] class <nombre>[extends
<superclase>][implements <lista_interfaces>]{

[ <atributo> | <método> | <constructor> | <clase> | <interfaz> |
<bloque estático> ]*

}
```

Declaraciones de atributos y métodos:

```
<ámbito> ::= [ public | private | protected ]

<atributo> ::= [<ámbito>] {<clase>|<tipo primitivo>}
<nombre_atributo>;
<método> ::= [<ámbito>] {<clase>|<tipo primitivo>}
<nombre_método>([<lista parámetros>]);

<lista_parámetros> ::= {<parámetro>| (<parámetro>,<lista_parámetros>)
}
<parámetro> ::= {<clase>|<tipo primitivo>} <nombre_parámetro>
```

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**: 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.

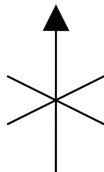
Las clases siempre serán públicas o de paquete (por defecto).

Primer programa:

```
public class Programa {

	public static void main(String[] args) {

		System.out.println( "Mensaje" );
	}

}
```

Algo un poco más elaborado:

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

Entrada de datos: System.in.

```
import java.io.*;

public class Programa {

      public static void main(String[] args) throws IOException {

            /* buffer de bytes donde guardar la entrada */
            byte[] buffer = new byte[200];
            /* lectura de la cadena */
            System.out.print( "Introduzca una cadena:" );
            System.in.read( buffer );
            /* Convertimos el buffer en una cadena
             * El método trim() elimina los caracteres
             * no válidos finales
             */
            String leido = (new String(buffer)).trim();
            System.out.println( "Se introdujo:" + leido );

            /* Siguiente entrada
             * Debemos borrar los caracteres que hay en
             * el buffer.
             */
            buffer = new byte[200];  // A lo bestia
            /* Lectura de un entero */
            System.out.print( "Introduzca un entero:" );
            System.in.read( buffer );
            leido = (new String(buffer)).trim();
            int entero = Integer.parseInt(leido);
            System.out.println( "Se introdujo:" + entero );

            /* Siguiente entrada
             * Debemos borrar los caracteres que hay en
             * el buffer.
             */
            buffer = new byte[200];  // A lo bestia
            /* Lectura de un double */
            System.out.print( "Introduzca un real:" );
            System.in.read( buffer );
            leido = (new String(buffer)).trim();
            double doble = Double.parseDouble(leido);
            System.out.println( "Se introdujo:" + doble );

      }
}
```

Entrada de datos: Clase Scanner. (Java 5)

```
import java.util.Scanner; // Entrada de un flujo
import java.util.Locale; // Para cambiar la localización

public class Programa {

      public static void main(String[] args){

            Scanner scan = new Scanner( System.in );
            /* Si no ponemos la línea siguiente la
             * localización espera que utilicemos la ,
             * como separador de decimales en lugar de .*/
            scan.useLocale( Locale.ENGLISH );

            // Entrada de cadena
            System.out.print( "Introduzca una cadena:" );
            String leida = scan.nextLine();
            System.out.println( "Se introdujo: " + leida );

            // Entrada de enteros
            System.out.print( "Introduzca un entero:" );
            int entero = scan.nextInt();
            System.out.println( "Se introdujo: " + entero );

            // Entrada de reales
            System.out.print( "Introduzca un real:" );
            double real = scan.nextDouble();
            System.out.println( "Se introdujo: " + real );
      }
}
```

Modularidad: paquetes, package e import.

- En Java **no tendremos** algo similar a los archivos de cabecera (**.hpp**). el programador escribirá los archivos de código (**.java**) y los podrá compilar a archivos de bytecodes (**.class**). Los archivos **.class incluyen la información necesaria para que** las clases que en ellos se incluyen **sean usadas por otros programas**.

- La **modularidad en Java** se consigue con los **paquetes** (***packages***). Por convención, los paquetes se nombran **en minúscula y con _ para separar palabras**.

- JDK nos **impone que los paquetes estén situados en directorios con el mismo nombre que el paquete** correspondiente.

- Especificamos a qué paquete pertenece un archivo poniendo como **primera línea de código** (no puede haber otro código antes) una **directiva**:

  package <nombre de paquete>;

- **Si no se especifica nada**, Java considera que se definen las clases en el **paquete por defecto**, que no tiene nombre. Los **archivos** estarán **en la raíz de compilación**.

- Para **incluir clases de otros paquetes** se utiliza la palabra reservada **import**. Las directivas import deben ser el siguiente código después de la directiva **package**. Su utilización es:

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

- **Los paquetes pueden tener subpaquetes**. Se utiliza el punto para referirse a subpaquetes. Por ejemplo:

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

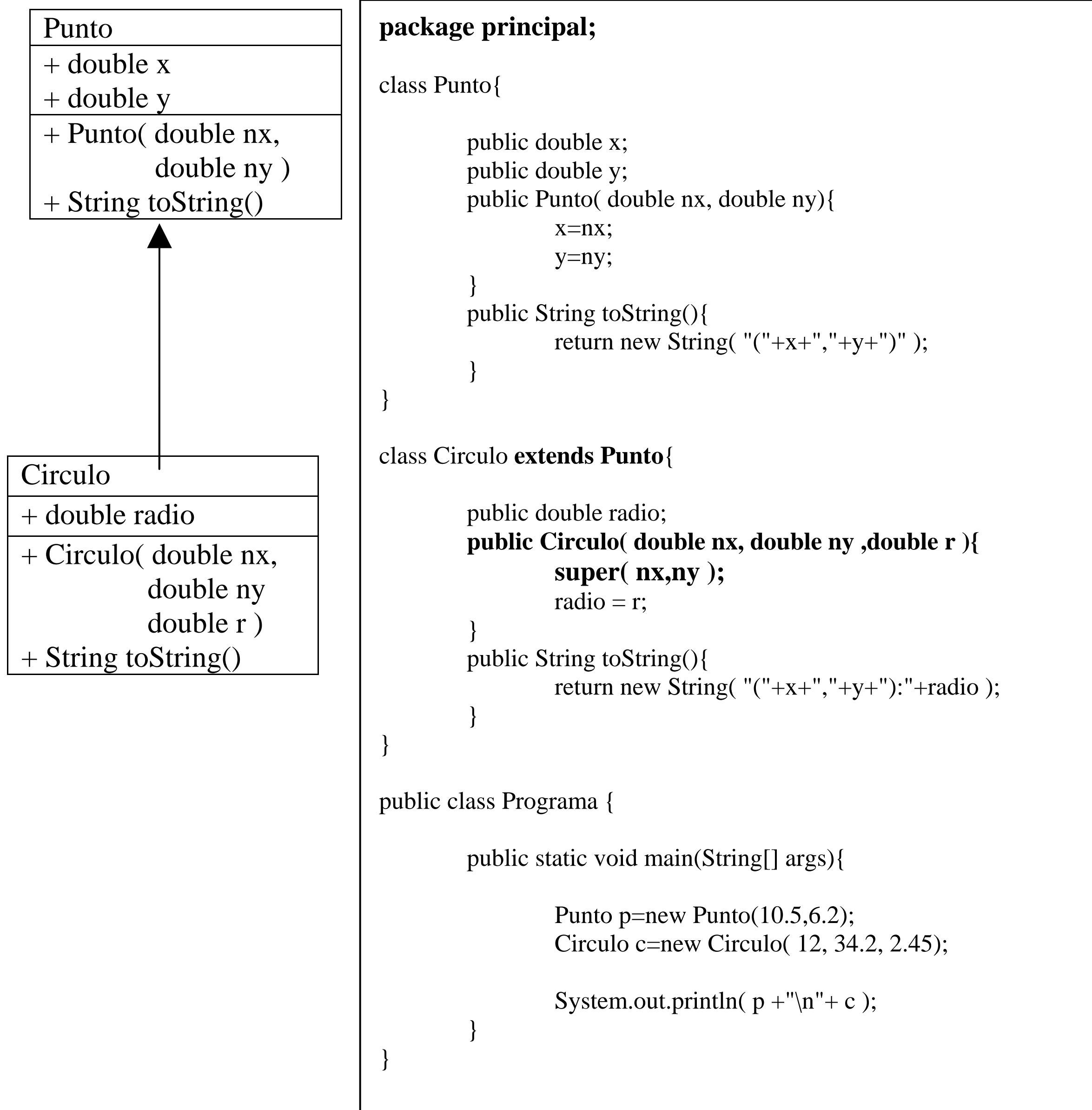
- **Ámbito de paquete: parecido a clases friend pero sólo con miembros protected.**

Herencia:

Sólo un tipo: no public ni private ni protected. Es equivalente a la herencia public de C++.

En java, los métodos sobrescritos siempre presentan un comportamiento polimórfico dinámico. Es como si en C++ siempre los declarásemos como virtuales.

Se programa con la palabra reservada **extends:**




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

public class Programa {

        public static void main(String[] args){

                Punto p=new Punto(10.5,6.2);
                Circulo c=new Circulo( 12, 34.2, 2.45);

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

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{ … }
```

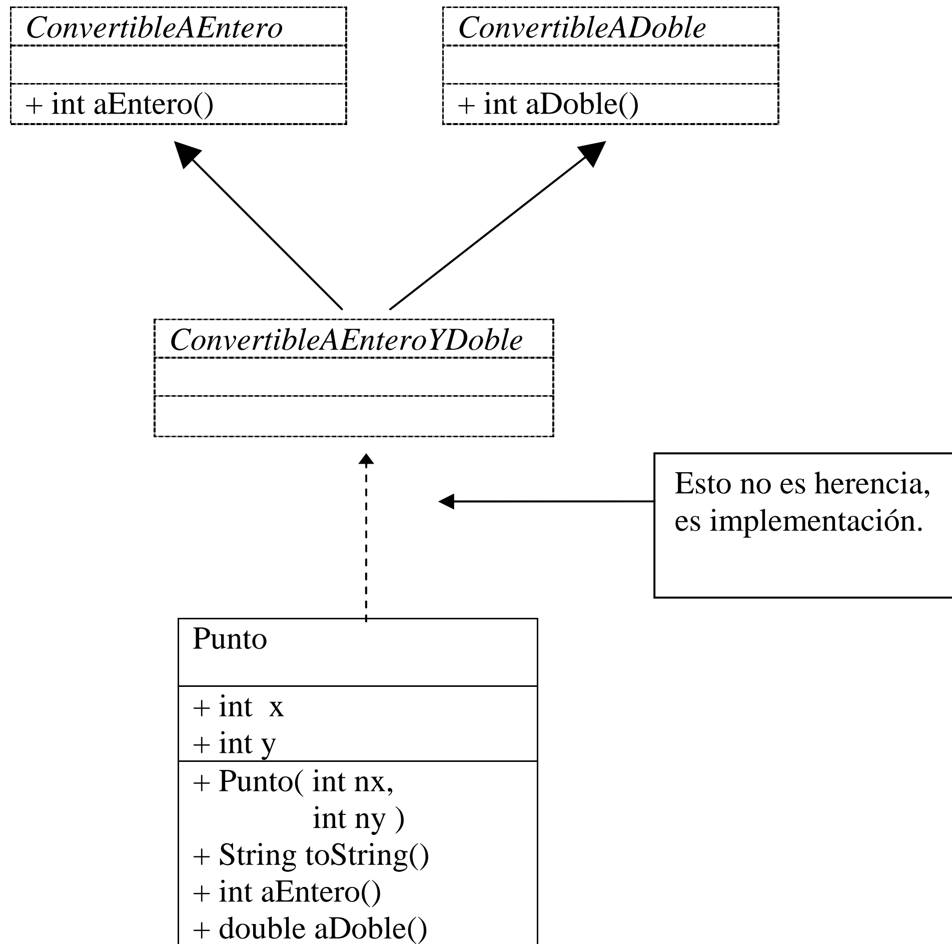
- Se pueden **declarar variables** dando como **clase** un **interfaz**.

```
{ …
   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: Ejemplo

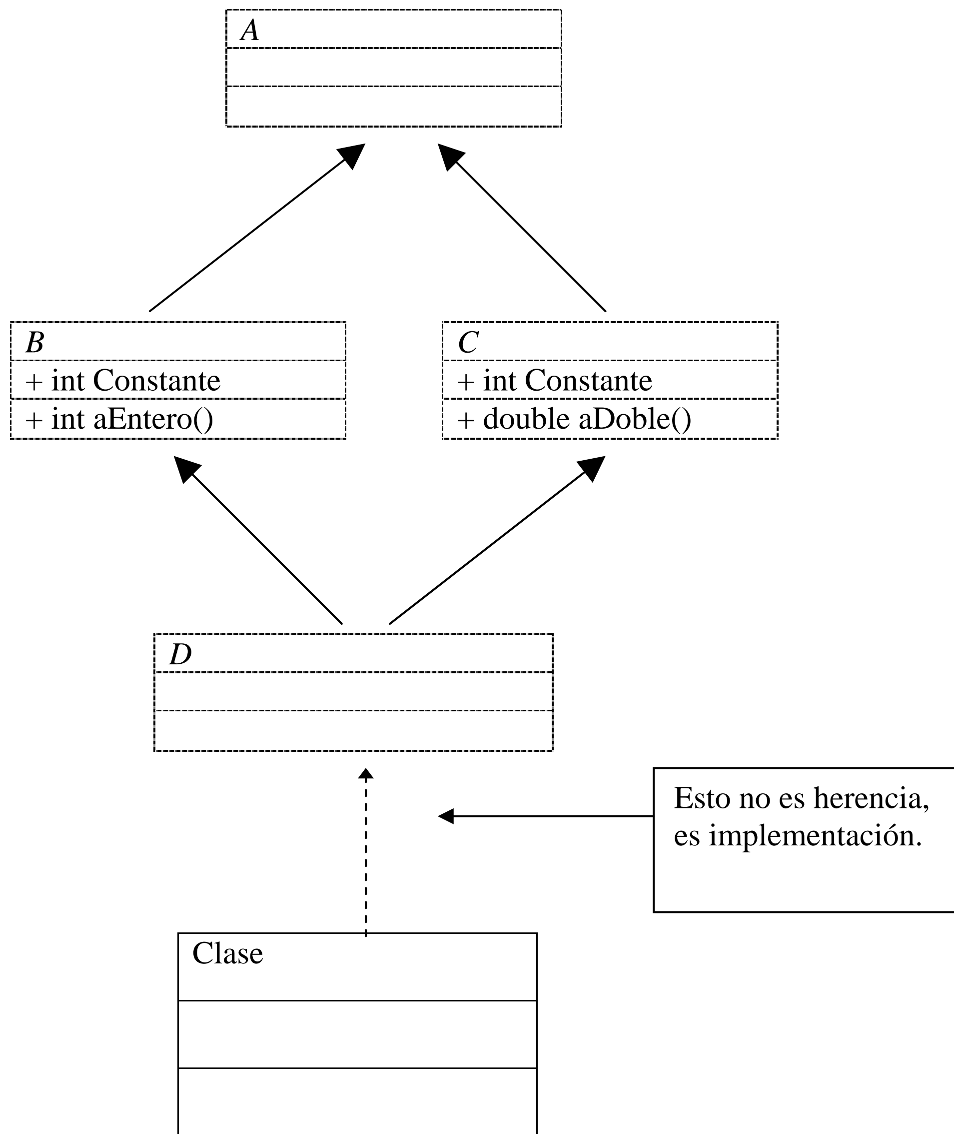


Interfaces: Ejemplo

```
package principal;

import java.util.Random; // Generador de números aleatorios

interface ConvertibleAEntero{
	int aEntero();
}

interface ConvertibleADoble{
	int aDoble();
}

interface ConvertibleAEnteroYDoble
	extends  ConvertibleAEntero,
	         ConvertibleADoble{
}

class Punto implements ConvertibleAEnteroYDoble{

	public int x;
	public int y;
	public Punto( int nx, int ny){
		x=nx;
		y=ny;
	}
	public String toString(){
		return new String( "("+x+","+y+")"+aEntero() );
	}
	/* Métodos del interfaz */
	public int aEntero(){
		return 2*x+3*y;
	}
	public int aDoble(){
		return aEntero();
	}
}

public class Programa {

	public static void main(String[] args){
		/* Generamos un array de puntos */
		Random r = new Random();
		ConvertibleAEntero [] array = new Punto[10];
		for ( int i=0; i<10 ;i++ )
			array[i] = new Punto(
			                        r.nextInt()%100,
			                        r.nextInt()%100 );
		/* Mostramos la representación entera*/
		for ( int i=0; i<10 ;i++ )
			System.out.println( array[i].aEntero() );
	}
}
```

Interfaces: caso del diamante

- **Pueden heredar de varios superinterfaces**. **Existen casos** en los que hay **ambigüedad**.
- **Los casos de ambigüedad se dan al definir constantes**. **Siempre que se define una constante, se obliga a su inicialización**.
- La **ambigüedad viene de la herencia múltiple**, no del hecho de que varios interfaces tengan un antepasado común.
- **En un subinterfaz podemos redefinir una constante de interfaces ascendientes.**



Interfaces: caso del diamante

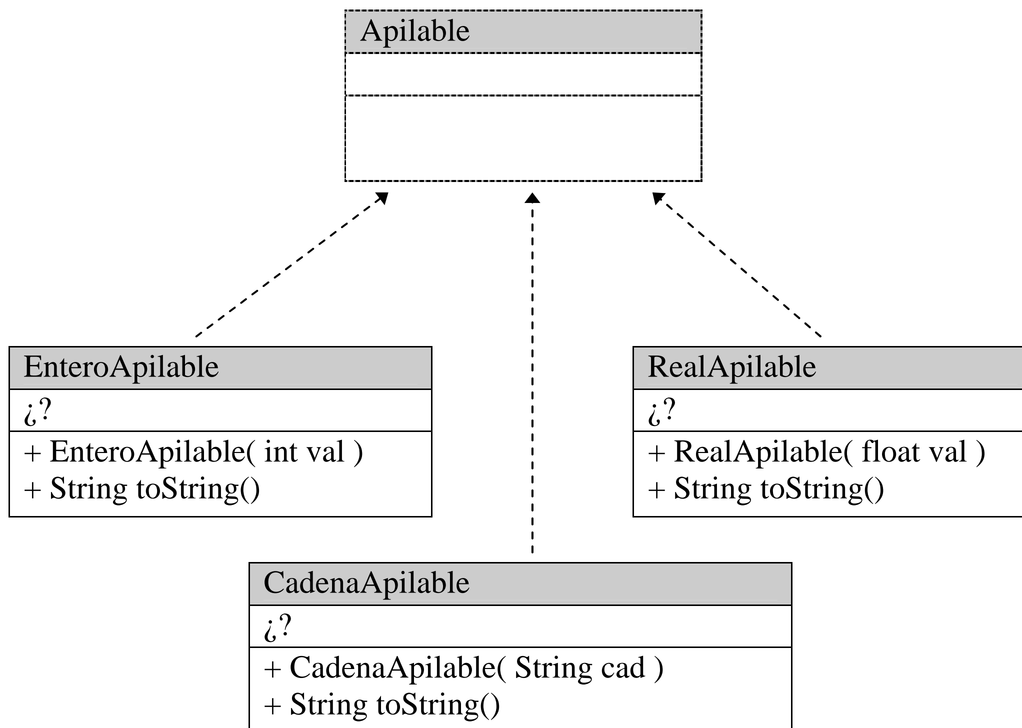
```
package principal;

interface A{
}

interface B extends A{
	int CONSTANTE = 1;
}

interface C extends A{
	int CONSTANTE = 2;
}

interface D extends B,C{
	// Solución 1
	//int CONSTANTE = 3;
}

class Clase implements D{
	// Solución 2
	//static int CONSTANTE = 3;
}

public class Programa {

	public static void main(String[] args){

		/* Si no se hace referencia a CONSTANTE,
		 * el programa compilará y funcionará
		 */
		// La siguiente línea es ambigua
		System.out.println( Clase.CONSTANTE );
	}
}
```

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**.

En el caso de las prácticas: Apilable se podría ajustar a este tipo de interfaces.



La referencia this

Igual que C++
Utilización de otros constructores.

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

```
package principal;

class Punto{

        private int x;
        private int y;

        public Punto( int x, int y ){
                this.x = x; // Referencia a sí mismo
                this.y = y; // Referencia a sí mismo
        }
        public Punto(){
                this(0,0); // llamada a otro constructor
        }
        public String toString(){
                return new String( "("+x+","+y+")" );
        }
}

public class Programa {

        public static void main(String[] args){

                Punto p = new Punto();
                System.out.println( p );
        }
}
```

Debe ir al comienzo de la definición del método

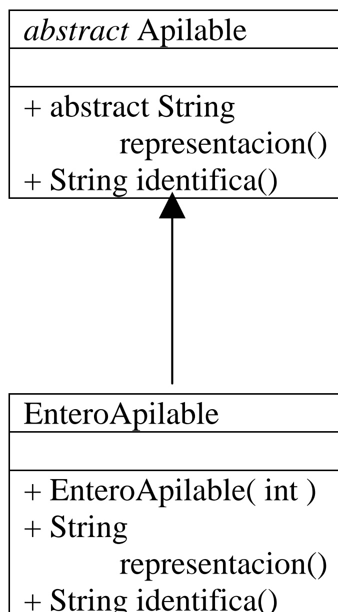
Clase abstracta:

Basta con anteponer la palabra reservada **abstract** al nombre de la clase:

Sólo las clases abstractas pueden contener métodos **abstract**, que se declaran como

```
[<ámbito>] abstract {<clase>|<tipo primitivo>} <nombre>( [<lista_parámetros>] )
```

Pueden existir clases abstractas que definan todos sus métodos (ninguno sea abstracto).

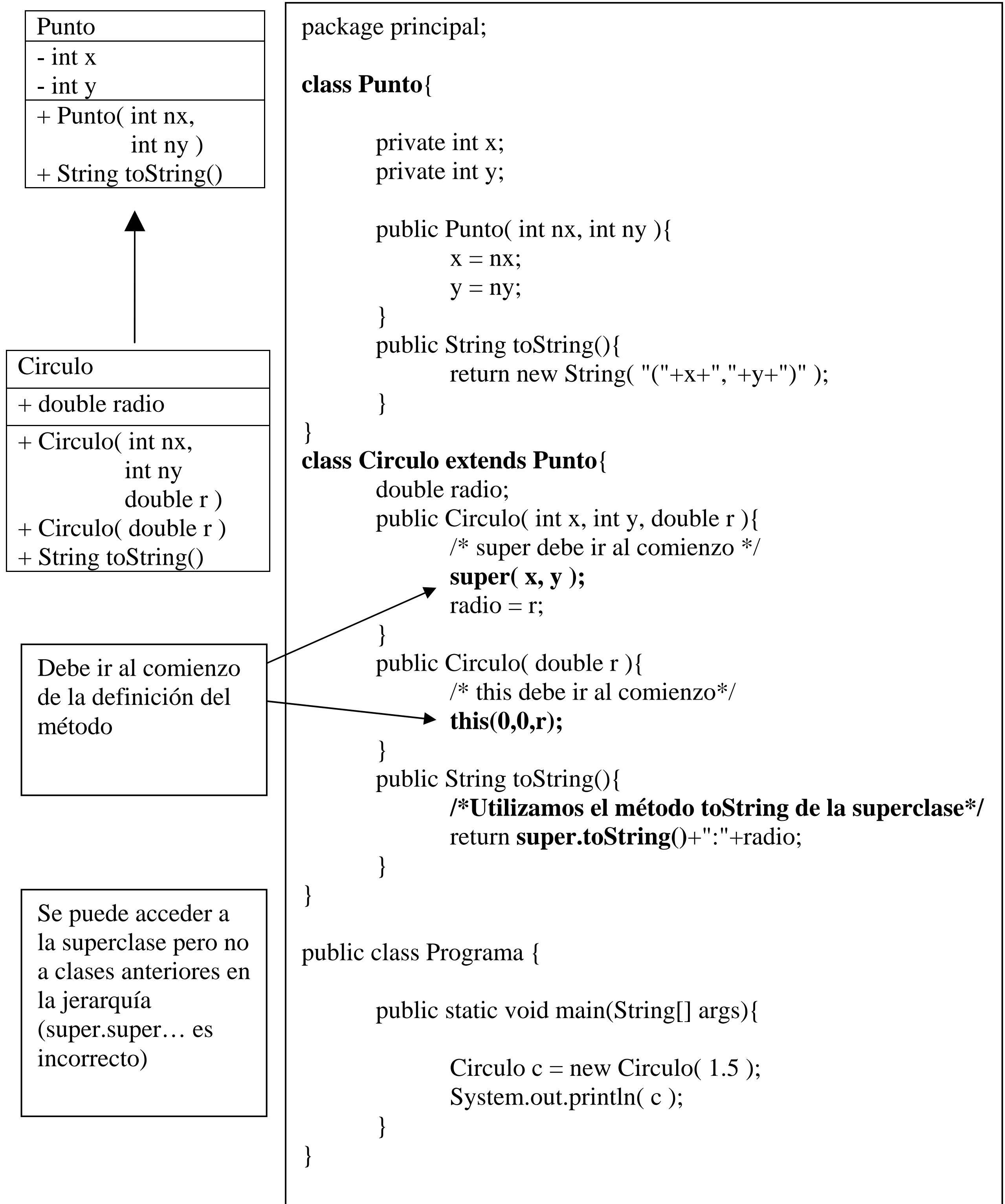



```
package principal;

abstract class Apilable{

        /* El siguiente método es abstracto y debe ser
         * definido en las subclases que queramos que sean
         * concretas (no abstractas)
         */
        public abstract String representacion();
        /* El siguiente método NO es abstracto y NO es necesario
         * que sea definido en las subclases para que dejen de
         * ser abstractas
         */
        public String identifica(){
                return "Apilable";
        }
}

class EnteroApilable extends Apilable{
        int val;
        public EnteroApilable( int nval ){
                val = nval;
        }
        /* El siguiente método hay que definirlo por fuerza
         * si queremos que se puedan instanciar EnteroApilables
         */
        public String representacion(){
                return new Integer( val ).toString();
        }
        /* El siguiente método hay que redefinirlo porque
         * queremos cambiar el de la superclase
         */
        public String identifica(){
                return super.identifica()+":EnteroApilable";
        }
}

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

                EnteroApilable ea = new EnteroApilable( 5 );
                /* Se pueden declarar variables de una clase
                 * abstracta. Lo que no se puede es instanciar
                 * la clase abstracta.
                 */
                Apilable a = new EnteroApilable( 10 );
                System.out.println( ea.identifica()
                                + " " + ea.representacion()+"\n"
                                + a.identifica()
                                + " " + a.representacion()
                                );
        }
}
```

Acceso a la superclase.

Palabra reservada **super.**

- Para acceder a los **métodos de la superclase**
- Para determinar los **constructores de la superclases** que hay que utilizar.




```
package principal;

class Punto{

        private int x;
        private int y;

        public Punto( int nx, int ny ){
                x = nx;
                y = ny;
        }
        public String toString(){
                return new String( "("+x+","+y+")" );
        }
}
class Circulo extends Punto{
        double radio;
        public Circulo( int x, int y, double r ){
                /* super debe ir al comienzo */
                super( x, y );
                radio = r;
        }
        public Circulo( double r ){
                /* this debe ir al comienzo*/
                this(0,0,r);
        }
        public String toString(){
                /*Utilizamos el método toString de la superclase*/
                return super.toString()+":"+radio;
        }
}

public class Programa {

        public static void main(String[] args){

                Circulo c = new Circulo( 1.5 );
                System.out.println( c );
        }
}
```

Instanciación: Constructores/Destructores

- En Java, como en C++, **existen** los **constructores**.

- **Una clase** puede tener un **número variable de constructores**.

- **Si no se define** ningún constructor para una clase, Java genera el **constructor por defecto sin argumentos**.

- **Diferencia**:
    - En **C++ se permitía** utilizar el constructor sin argumentos **omitiendo los paréntesis**. En **Java**, **siempre** tendremos que poner los **paréntesis.**
- Los constructores tienen el **mismo nombre que la clase y no tienen una clase/tipo primitivo de devolución**. Implícitamente **devuelven una referencia a un objeto de la clase**.

- **Java no implementa argumentos por defecto**

new y finalize

El **operador de instanciación en Java** es **new**. La **forma de instanciar** una clase es **variable = new constructor([<argumentos>]);**

Ejemplos:

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

o, más compacto:

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

- **New** se encarga de la **reserva de memoria y** de llamar al **constructor**.

- Los **constructores** se llaman **en cascada**, **como** en el caso de **C++**.

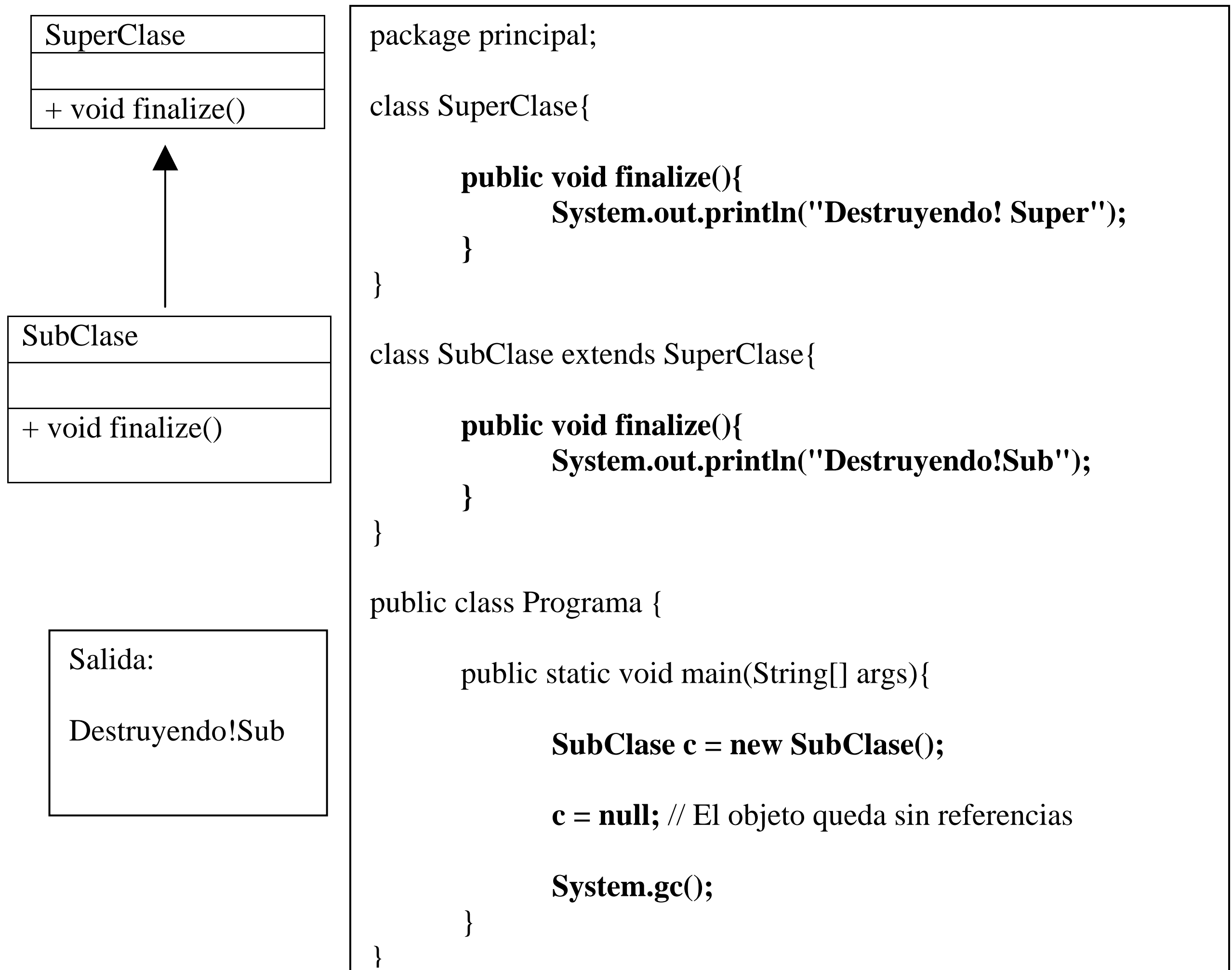
Destructores:

- **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.

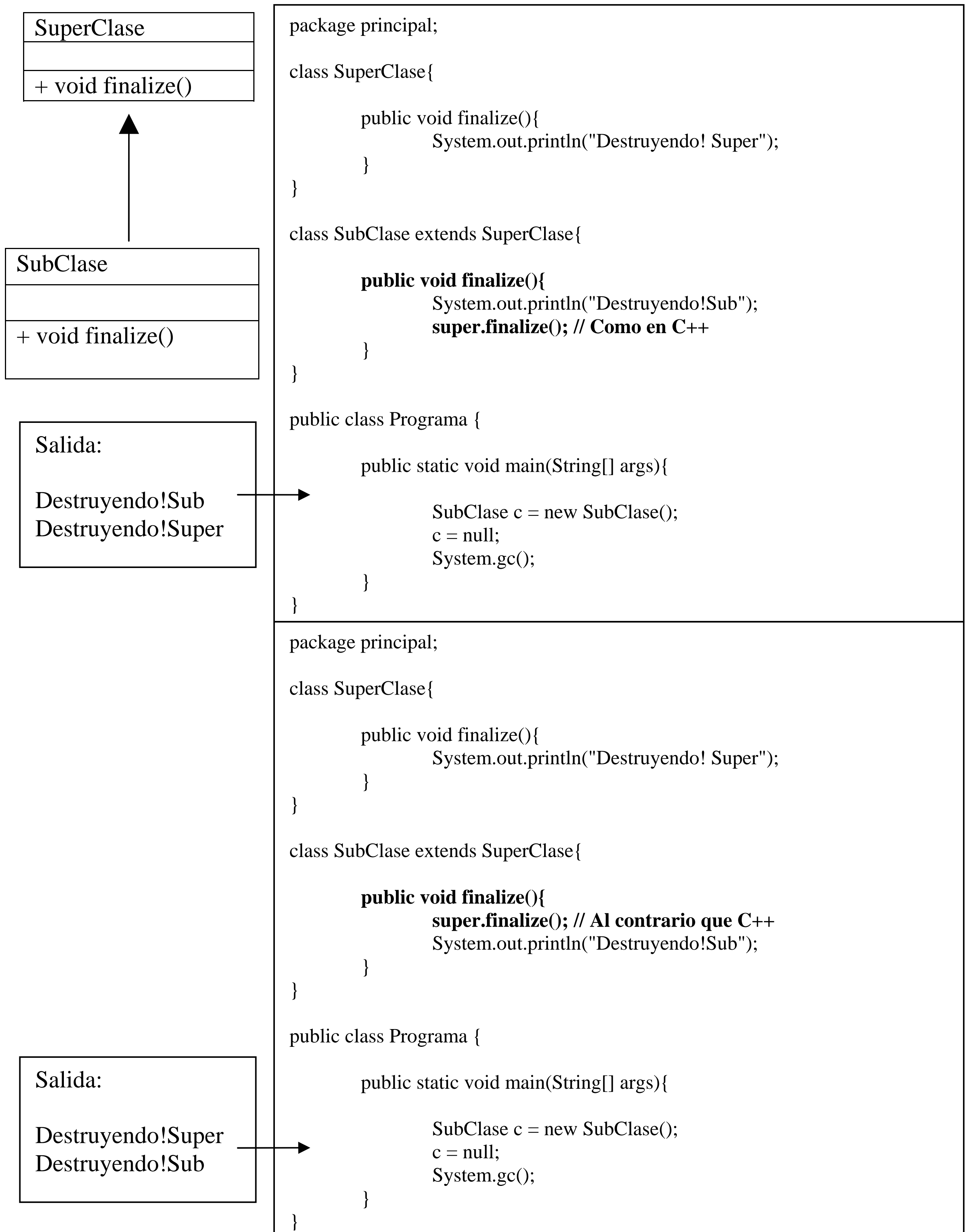
- El método **finalize no tiene por qué llamarse en cascada**. En C++, los destructores se van llamando desde la subclase hacia las superclases. Esto es diferente en Java: sólo se llama al finalize de la clase que se elimina.




```
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();
        }
}
```

Destructores:

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



Destructores:

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

SuperClase
+ void finalize()

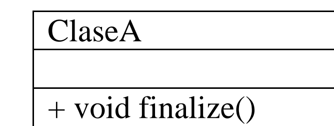
SubClase
+ void finalize()

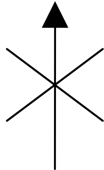
```
package principal;

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();
	}
}
```

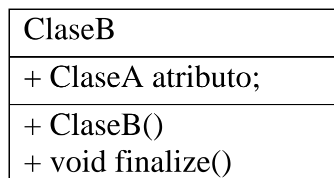
```
Salida:

Primera llamada
Segunda llamada
Destruyendo!Sub
```

Destructores:

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





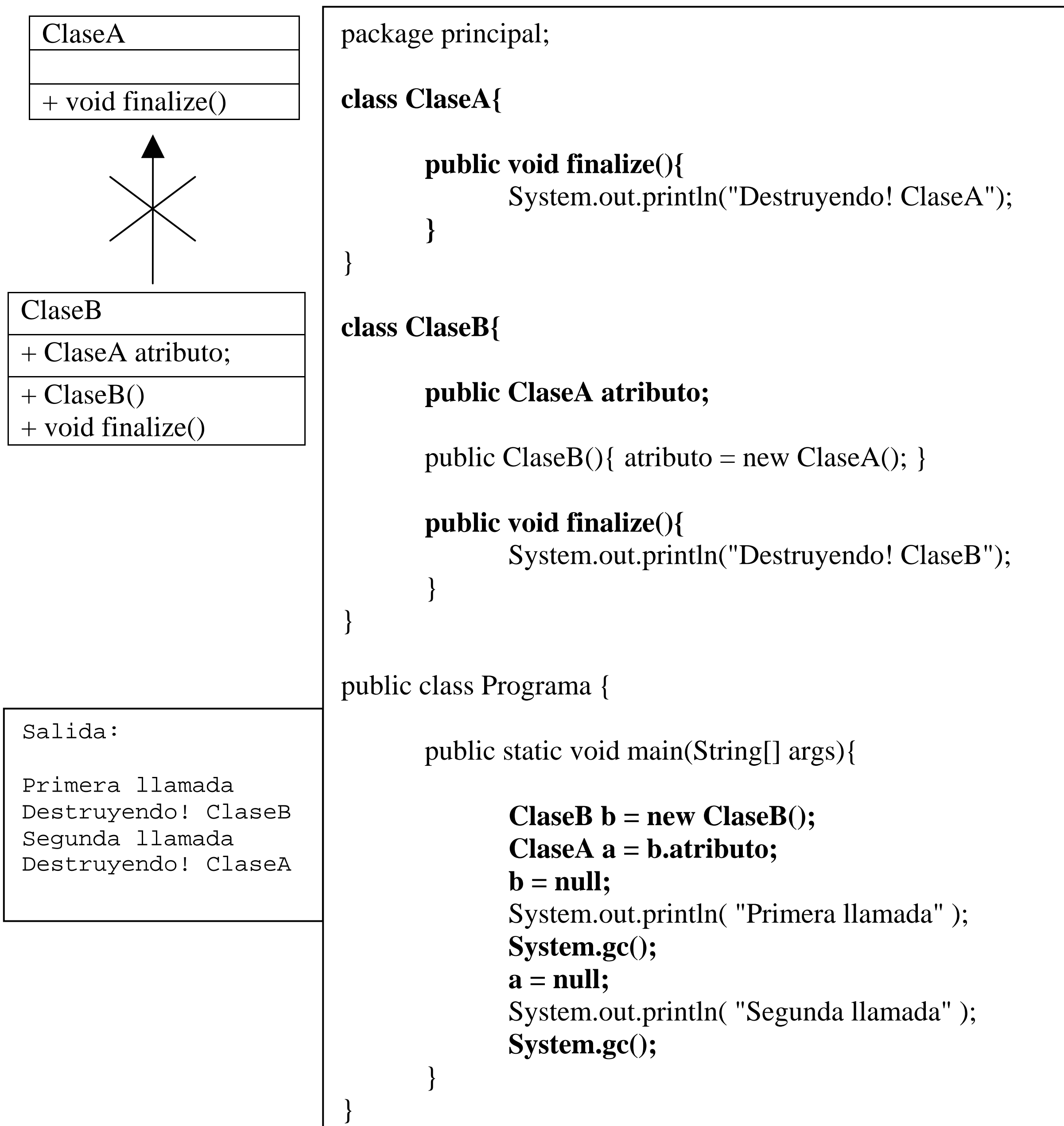



```
package principal;

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
```

NOTA: podemos pensar que el recolector de basura Java realiza todas las tareas de finalización y que el método finalize no tiene ningún sentido. Cuando programamos con las clases estándar, esto es, en general, cierto. El problema es cuando programamos nuevas clases.

Ejemplo: programación de un buffer de salida. Si tenemos en memoria un buffer con información almacenada que debe enviarse a una salida (archivo, flujo, …), si eliminamos 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 (flush o algo por el estilo)

Static: atributos y métodos de clase:

- Los **métodos y atributos static se denominan también métodos y atributos de clase**. La idea es que **codifican un atributo común a todos los objetos de la clase (asociado a la clase) o un método asociado a la clase, no a los objetos concretos (su comportamiento no depende de ningún objeto concreto)**

- **El que un atributo sea común a todos los objetos de la clase se plasma en que sólo habrá una copia del atributo global a todos los objetos de la clase.**

- **Los objetos concretos tendrán acceso a los atributos y métodos de clase.**

- **El que el comportamiento de un método de clase no dependa de ningún objeto concreto de la clase se consigue obligando a que estos métodos sólo tengan acceso a variables locales y argumentos (propias del método) y a aquellos atributos que no pertenecen a ningún objeto concreto de la clase: los atributos de clase. Es decir, un método estático no tendrá acceso a los atributos de instancia (porque no es ejecutado realmente por ninguna instancia sino que es ejecutado por la clase).**

- **No es necesario que exista ningún objeto de la clase para que se pueda invocar a un método de clase (estático).**

- Para hacer una **invocación** a un método estático, se hace:

```
<clase>.<método_estático>([<lista_argumentos>]);
```

Ejemplo:

```
int i = Integer.parseInt( "45" );
```

Static: atributos y métodos de clase:

Ejemplo de una clase que lleva una cuenta de los objetos que hay actualmente instanciados.

Punto
- double x - double y - static int cuentaInstancias
+ Punto(double nx, double ny) + public static String cadenaCuenta() + public void finalize()

```
package principal;

class Punto{

        private double x;
        private double y;
        private static int cuentaInstancias = 0;
        public static String cadenaCuenta(){
                return new Integer(cuentaInstancias).toString();
                /*alternativa
                 * return new String(""+cuentaInstancias);
                 */
        }
        public Punto( double nx, double ny ){
                cuentaInstancias++;
                x = nx;
                y = ny;
        }
        // ¡Ejemplo de programa donde necesitamos un finalizador!
        public void finalize(){ cuentaInstancias--; }
}

public class Programa {

        public static void main(String[] args){

                /* Mostramos la cuenta de puntos */
                System.out.println( "Tenemos: "
                                + Punto.cadenaCuenta()
                                + " Puntos");
                /* Generamos 5 puntos */
                Punto [] array = new Punto[5];
                for ( int i=0; i<5; i++ )
                        array[i] = new Punto(i*2, i*3);
                /* Mostramos la cuenta de puntos */
                System.out.println( "Tenemos: "
                                + Punto.cadenaCuenta()
                                + " Puntos");
                /* Eliminamos las referncias a los 5 puntos */
                for ( int i=0; i<5; i++ ){
                        array[i] = null;
                        System.gc();
                        System.out.println( "Tenemos: "
                                        + Punto.cadenaCuenta()
                                        + " Puntos");
                }
        }
}
```

```
Tenemos: 0 Puntos
Tenemos: 5 Puntos
Tenemos: 4 Puntos
Tenemos: 3 Puntos
Tenemos: 2 Puntos
Tenemos: 1 Puntos
Tenemos: 0 Puntos
```

Static: boque 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()

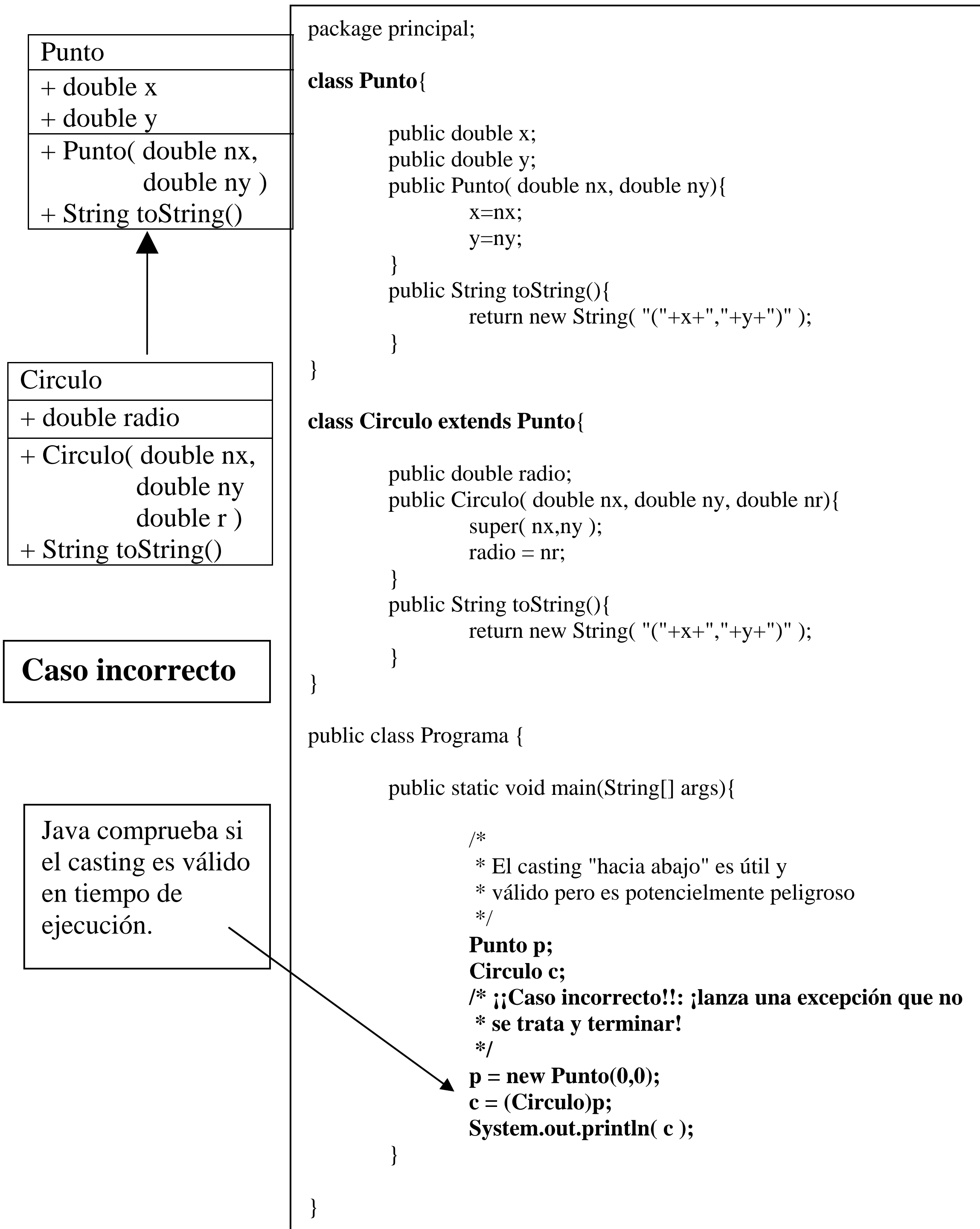
```
package principal;

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)
```

Información de clase en tiempo de ejecución: Class e instanceof

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




```
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 potencielmente 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)
```

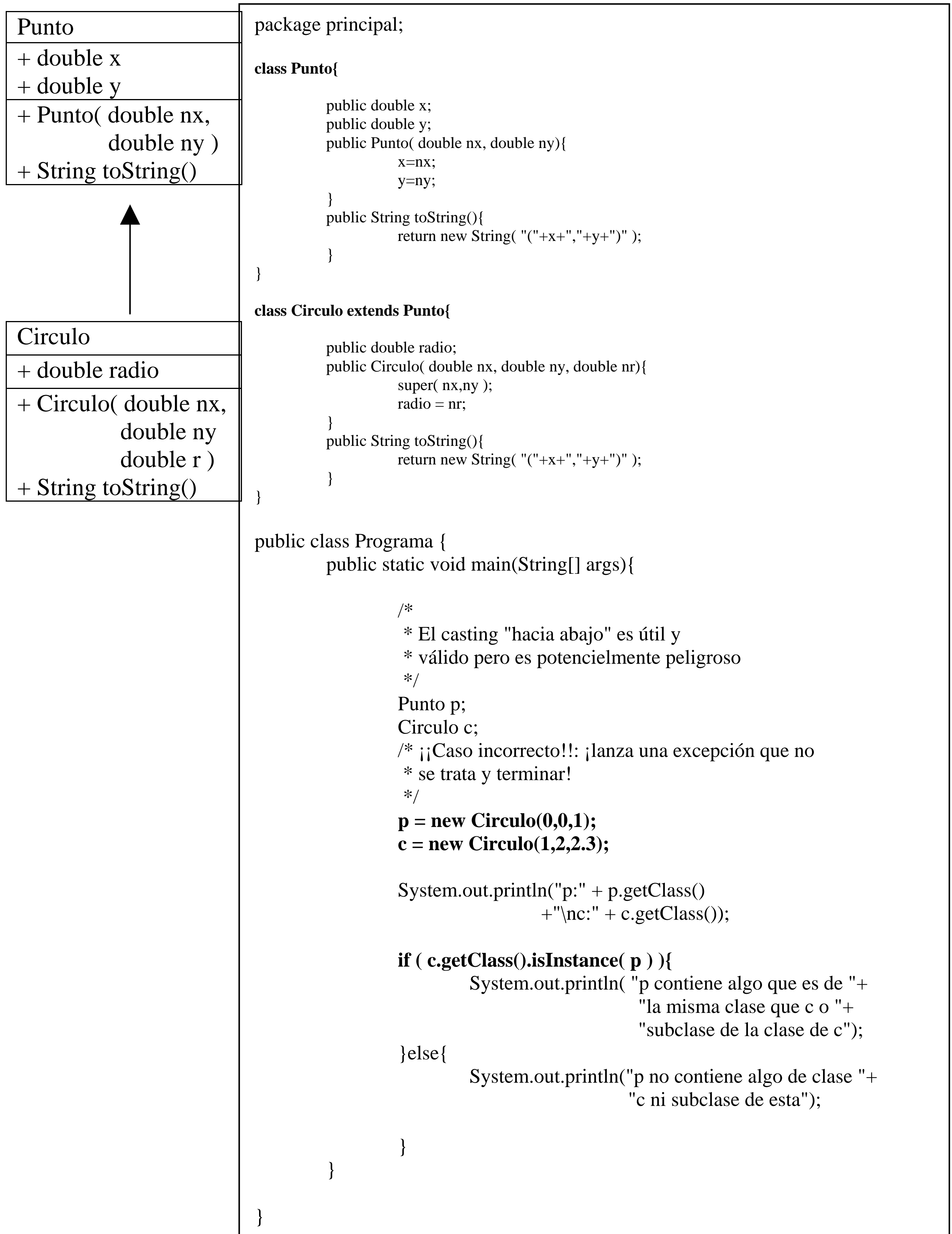
Solución: asegurarnos de que hacemos un casting válido

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 potencielmente 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 potencielmente 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");

                }
        }

}
```

Sobrecarga de operadores: no existe como tal.

Ejemplo: toString()

Final

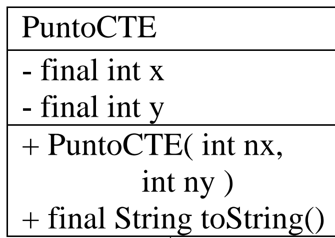
Final tiene varias utilizaciones en C++. Su significado viene a ser siempre invariabilidad. Dependiendo de a qué elemento se asocie, este significado de invariabilidad cambia:

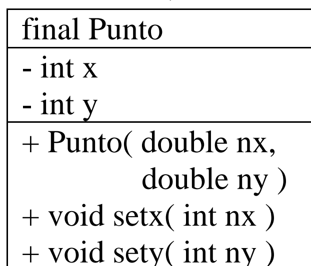
- **Atributos y variables locales**: cuando un atributo se cualifica como final, se establece que la referencia que contiene no se puede cambiar. Es decir, no se puede asignar otra referencia.
    - Es **obligatorio asignar** un atributo o variable final **cuando se declara o** bien **en el constructor** del objeto.
    - **Diferencias con C++**: **Java no hace restricciones con respecto a los métodos que se pueden invocar** en el objeto al que hace referencia la variable (no se puede restringir la utilización de métodos de modificación por parte de los objetos constantes).
    - **Lo que es constante es la referencia que contiene la constante, no el objeto en sí.**

- **Clases**: una clase final es una clase se la que no se puede heredar (no se puede extender, no puede ser superclase de otra). Esto se hace por seguridad.

- **Métodos**: un método final es un método que no puede ser sobrescrito en las subclases.

Final



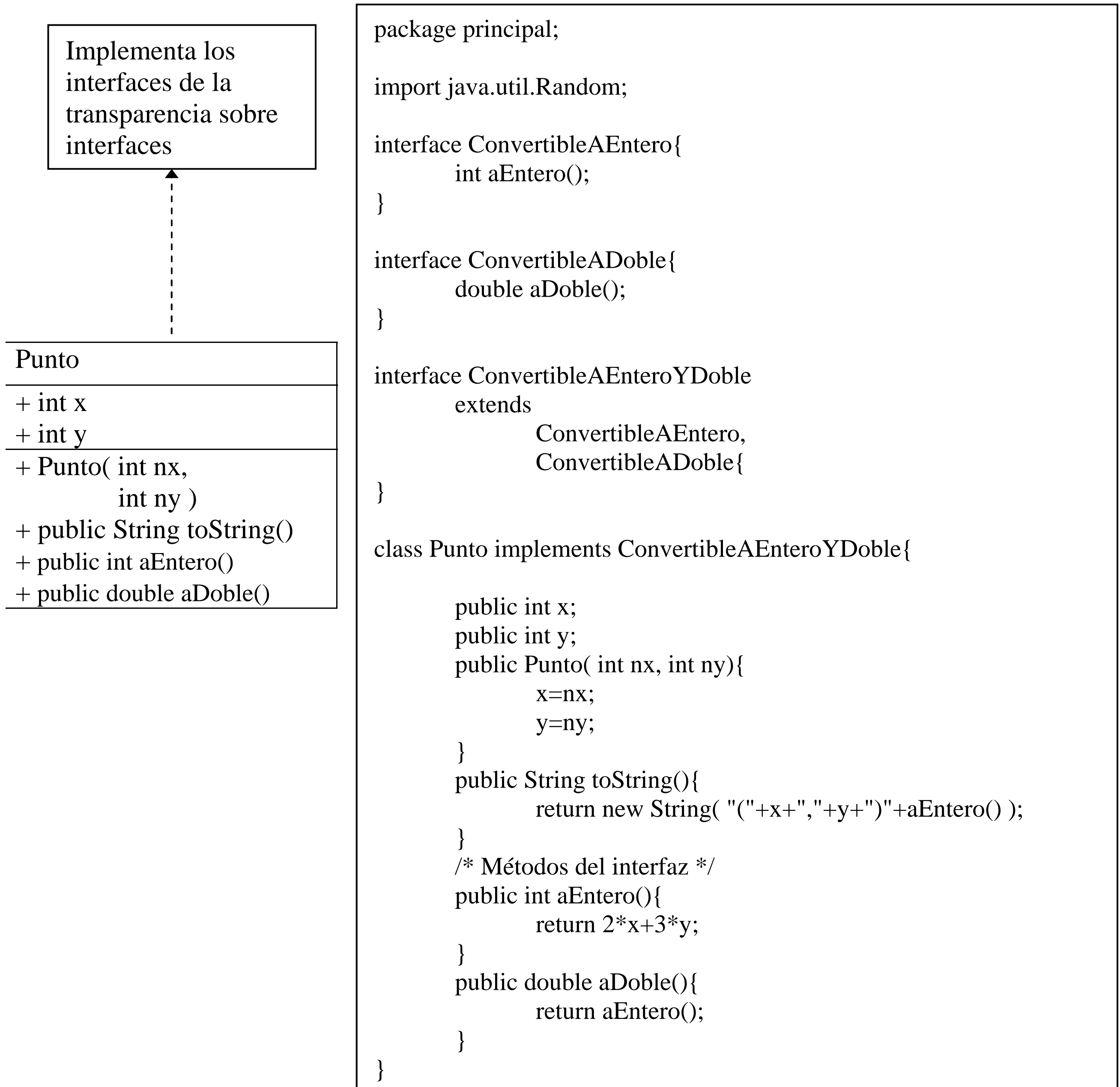



```
package principal;

class PuntoCTE{

	private final int x;
	private final int y;
	public PuntoCTE( int nx, int ny){
		// Inicialización de las constants en el constructor
		x=nx;
		y=ny;
	}
	public final String toString(){
		return new String( "("+x+","+y+")" );
	}
}

final class Punto extends PuntoCTE{
	// Se cambian los privilegios de acceso de las coordenadas
	private int x;
	private int y;
	public Punto( int nx, int ny){
		super(nx,ny);
	}
	public void setx( int nx ){
		x = nx;
	}
	public void sety( int ny ){
		y = ny;
	}
}

public class Programa {

	public static void main(String[] args) {

		Punto p = new Punto(10,20);
		p = new Punto(30,30); //Sí válido
		p.setx(20); // Sí válido

		final Punto fp = new Punto(10,20);
		//fp = new Punto(30,30); No válido!!
		fp.setx(20); // Sí válido

		PuntoCTE pc = new PuntoCTE(1,1);
		pc = new PuntoCTE(2,2); // Sí válido
		// pc no tiene métodos para cambiar sus coordenadas

		final PuntoCTE fpc = new PuntoCTE(1,1);
		//fpc = new PuntoCTE(2,2); No válido!!
		// pc no tiene métodos para cambiar sus coordenadas

		System.out.println( ""+ p +" "+ fp +" "
					+ pc +" "+ fpc );
	}
}
```

Static: clase no instanciable (*singleton*): ejemplo de ordenación (1).

Ejemplo con utilización de interfaces, métodos estáticos y clases no instancianbles.

Una clase no puede ser a la vez abstracta y final.




```
package principal;

import java.util.Random;

interface ConvertibleAEntero{
        int aEntero();
}

interface ConvertibleADoble{
        double aDoble();
}

interface ConvertibleAEnteroYDoble
        extends
                ConvertibleAEntero,
                ConvertibleADoble{
}

class Punto implements ConvertibleAEnteroYDoble{

        public int x;
        public int y;
        public Punto( int nx, int ny){
                x=nx;
                y=ny;
        }
        public String toString(){
                return new String( "("+x+","+y+")"+aEntero() );
        }
        /* Métodos del interfaz */
        public int aEntero(){
                return 2*x+3*y;
        }
        public double aDoble(){
                return aEntero();
        }
}
```

Static: clase no instanciable (estática o *singleton*): ejemplo de ordenación (2).

- No se puede heredar de la clase
- El constructor es privado (no se puede instanciar)
- Todos los métodos son estáticos

```
final class Ordenacion{

	private Ordenacion(){}
	public static void ordenar( ConvertibleAEntero [] array ){

		/* Método de la burbuja */
		for ( int j=0; j<array.length-1 ; j++ ){
			for ( int i=0; i<array.length-1 ;i++ ){
				if ( array[i].aEntero() < array[i+1].aEntero()
){
					ConvertibleAEntero tmp = array[i];
					array[i] = array[i+1];
					array[i+1] = tmp;
				}
			}
		}
	}
}

public class ProgOrdenacion {

	public static void main(String[] args){
		/* Generamos un array de puntos */
		Random r = new Random();
		/* Podríamos haber puesto array como un
		 * ConvertibleAEntero []
		*/
		Punto [] array = new Punto[10];
		for ( int i=0; i<10 ;i++ )
			array[i] = new Punto(
						r.nextInt()%100,
						r.nextInt()%100 );
		/* Los mostramos */
		System.out.println( "Sin ordenar:" );
		for ( int i=0; i<10 ;i++ )
			System.out.println( array[i] );
		/* Los ordenamos */
		Ordenacion.ordenar( array );
		/* Los mostramos */
		System.out.println( "Ordenados:" );
		for ( int i=0; i<10 ;i++ )
			System.out.println( array[i] );
	}
}
```

```
Salida:

Sin ordenar:
(-39,-98)-372
(-71,-27)-223
(86,38)286
(-60,22)-54
(26,70)262
(-36,7)-51
(14,5)43
(-16,14)10
(-13,42)100
(43,-74)-136
Ordenados:
(86,38)286
(26,70)262
(-13,42)100
(14,5)43
(-16,14)10
(-36,7)-51
(-60,22)-54
(43,-74)-136
(-71,-27)-223
(-39,-98)-372
```

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…)

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

```
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]);
        }
}
```