Tema 1- Conceptos de Java para Estructuras de Datos

Germán Moltó
Escuela Técnica Superior de Ingeniería Informática
Universidad Politécnica de Valencia

Índice general:

Objetivos y Bibliografía

- Objetivos:
  - Conocer los conceptos básicos de la POO y su soporte en el lenguaje Java
  - Aprender el concepto de Herencia en Java para afrontar el desarrollo de estructuras de datos y algoritmos que se realiza en temas posteriores.
  - Estudiar el concepto de paquete como mecanismo para la agrupación de clases.
  - Comprender el concepto de Polimorfismo, Interfaz y Clase Abstracta, así como su utilidad.
- Bibliografía:
  - Weiss, M.A. Estructuras de datos en Java. Adisson-Wesley, 2000. Capítulos 2 (Apartado 2.5) 3 y 4
  - Arnow, D., Weiss G. Introducción a la Programación con Java. Un enfoque orientado a objetos. Adisson-Wesley, 2001. Capítulo 14

Objetos y Clases en Java (I)

- En Java todo son objetos (i.e. instancias de una Clase dada), exceptuando los 8 tipos primitivos (i.e. byte, short, int, long, float, double, char y boolean).
- Una Clase consta de un conjunto de atributos (almacenan datos) y un conjunto de métodos (trabajan con esos datos).
- Un Objeto de una Clase se usa/manipula desde cualquier método de otra Clase, como por ejemplo TestClase.

```
public class TestClase {
        public static void main( String args[] ){ ... }
    }
```

- Ejemplo: sea TestCirculo una aplicación que maneja Círculos, i.e. Objetos de la clase Circulo

Objetos y Clases en Java (II)

```
public class TestClase {
        public static void main( String args[] ){
                ¿?
        }
}
```

- ¿Qué podemos hacer con un objeto?
  - **Especificación** de una Clase: descripción de QUÉ se puede hacer con un Objeto, es decir, qué métodos expone (especificación de Circulo).
  - **Implementación** de una Clase: detalles internos de CÓMO se satisface la Especificación o cómo se consigue hacer lo que se hace con un Objeto.

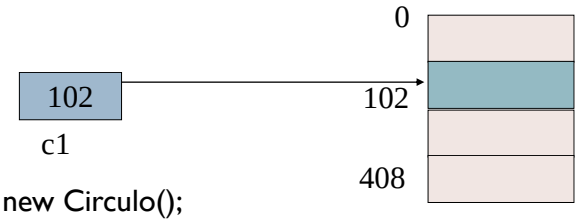
Objetos y Clases en Java (III)

```
public class TestClase {
        public static void main( String args[] ){
            Circulo c1 = new Circulo();
            Circulo c2;
            c2 = c1;
            }
    }
```

- Creación de Objetos de una Clase:
  - c1 es una variable Referencia al nuevo Objeto creado.
  - c2 es una variable Referencia a ningún Objeto, i.e. tras la declaración Circulo c2 ; c2 == null
  - ¿ Qué ocurre si se ejecuta c2 = c1; ?

Referencias en Java

- Una variable Referencia almacena la dirección de memoria en la que se encuentra el Objeto al que referencia.



Circulo c1 = new Circulo();

- Al hacer: Memoria
  - Circulo c2 = c1;
  - La variable referencia c2 pasa a tener el mismo valor que c1, es decir, ambas apuntan al mismo objeto.

Ejemplo de Utilización de Objetos Circulo

- Podemos crear un objeto de la clase Circulo de la manera especificada en los constructores que define la clase.

```
public class TestCirculo {
    public static void main( String args[] ){
            Circulo c1 = new Circulo();
            double radio = 4.0;
            Color color = Color.red;
            Circulo c2 = new Circulo(radio, color);
            System.out.println(c2);
            System.out.println(c2.toString());
    }
}
```

¿Qué diferencia hay en las dos últimas invocaciones?

Implementación de una Clase

- Implementación de una Clase:
  - Detalles internos de cómo se satisface la Especificación o cómo se consigue hacer lo que se hace sobre un Objeto.
- Componentes de una Clase:
  - **Atributos**, que almacenan los datos de la Clase
  - **Métodos**, que definen la funcionalidad de la Clase mediante, por lo general, la manipulación de los atributos.
- Elementos que aparecen en la implementación de una clase:
  - El operador **this**.
  - Modificadores estáticos, dinámicos y finales de una Clase y sus componentes.
  - Modificadores de visibilidad public y private de una Clase y sus componentes.

Implementación de una Clase: Definición de Atributos

- Los **Atributos** de una Clase almacenan los datos de la Clase
  - Modelo de la relación TIENE UN(A)
  - Modificadores de atributos
- Ejemplo: definición de los Atributos de la Clase Circulo

```
private  double radio;
private  String color;
private  static  final  double RADIO_POR_DEFECTO = 3.0;
private  static  final  Color COLOR_POR_DEFECTO = Color.black;
```

- Por lo general, los atributos se suelen definir privados y se definen métodos consultores y modificadores. De esta manera se garantiza la **encapsulación** u **ocultación de datos**.
- Modificador ***static***: Permite definir variables a nivel de clase (compartidas por todos los objetos).
- Modificador ***final***: Impide asignar un nuevo valor al atributo.

Implementación de una Clase: Definición de Constructores

- Los **Constructores** de una clase permiten dar un valor inicial a los atributos del objeto. Ejemplo: definición de los métodos Constructores de Circulo

```
public Circulo() {
    radio = RADIO_POR_DEFECTO;
    color = COLOR_POR_DEFECTO;
}
public Circulo(double radio, Color color) {
        this.radio = radio;
        this.color = color;
}
public Circulo(){
    this(RADIO_POR_DEFECTO, COLOR_POR_DEFECTO);
}
```

this hace referencia al objeto sobre el que se invoca el método y sirve para evitar el ***aliasing*** de nombres.

Implementación de una Clase: Definición de Métodos (I)

- Los **Métodos** de una clase definen su funcionalidad.
- Ejemplo: Definición de **Métodos Consultores** en la clase Circulo.
  - Al definir como privados todos los atributos de Circulo, se deben proporcionar métodos para acceder a su valor.

```
public double getRadio() {
        return radio;
}

public Color getColor() {
        return color;
}
```

Se podría utilizar *this* para referenciar a los atributos (i.e., this.radio) pero no es necesario puesto que no hay aliasing de nombres.

Implementación de una Clase: Definición de Métodos (II)

- Al definir como privados todos los atributos de Circulo, se deben proporcionar métodos para modificar su valor.
- Ejemplo: definición de los **Métodos Modificadores** de Circulo

```
public void setRadio(double nuevoRadio) {
   radio =  nuevoRadio;
}
public void setColor(Color nuevoColor) {
   color = nuevoColor;
}
```

Utilizar diferentes nombres para el argumento y el atributo evita el aliasing y ya no es necesario utilizar *this*.

Implementación de una Clase: Definición de Métodos (III)

- Ejemplo: definición de otros Métodos de Circulo

```
public double area() { return Math.PI * radio * radio; }
public String toString() {return "Circulo de radio " + radio}
public static Circulo leerCirculo(Scanner teclado) { ... }
```

- El uso de un método estático permite definirlo a nivel de clase, no a nivel de instancia (objeto individual).
  - Se deberá invocar como Circulo.leerCirculo(...).

¿Se podría utilizar *this* en la implementación del método area para referenciar al atributo radio?

Estado Actual de la Clase Circulo

```
public class Circulo {
   private double radio;
   private Color color;
   private static final double RADIO_POR_DEFECTO = 3.0;
   private static final Color COLOR_POR_DEFECTO = Color.black;

 /** crea un Circulo con radio r y color c */
   public Circulo(Color c, double r) {
     this.color = c;
      this.radio = r;
}
   /** crea un Circulo estándar.:  radio 3.0, color negro y centro en el origen*/
   public Circulo() {
      this(COLOR_POR_DEFECTO, RADIO_POR_DEFECTO);
   }
…
}
```


Ejercicios Propuestos

- **Ejercicio 1:** Escribe la clase Manzana que tiene un sabor, de tipo entero que representa una escala [0,10].
  - Por defecto, la manzana tiene un sabor de 5, aunque se puede especificar su sabor al crearla.
- **Ejercicio 2**: En lugar de trabajar con Círculos, se desea trabajar con otro tipo de Figuras (Cuadrados o Rectángulos o Triángulos). Si ya se dispone de la aplicación misCirculos, ¿cómo se afrontaría el diseño e implementación de la nueva aplicación?
- **Ejercicio 3**: Se desea realizar una aplicación que maneje un grupo de Círculos. En particular, se desea:
  - Construir el grupo con varios Circulos, mostrar el grupo por pantalla y obtener el área del grupo.

Siguiendo los pasos dados para la presentación de la aplicación misCirculos, desarrolla la aplicación que maneje un grupo de Círculos (GrupoDeCirculos).

Paquetes Java: Reutilización de Código

- El mecanismo Java más general para organizar un grupo de Clases que guardan alguna relación entre sí es el **Paquete**
- Paquetes Java estándar
  - java.lang : Contiene, entre otras, las clases Integer, Math, String y System
  - java.util : Contiene, entre otras, las clases Date, Random y StringTokenizer
  - java.io, java.awt (incluye la clase Color), etc.
- Manipulación:
  - la directiva **import**; la variable de entorno **CLASSPATH**; la instrucción **package.**
- Reglas de visibilidad dentro y fuera de un paquete.
- Es posible investigar los paquetes que incluye el API de Java mirando la documentación:
  - http://java.sun.com/javase/6/docs/api/

Paquetes Java: Reglas de Utilización

- La Clase C del Paquete p se especifica como p.C

```
public class MiClase(){
    ...
    java.util.Date fecha = new java.util.Date();
    ...
}
```

- Simplificación: uso de la directiva import (antes de la declaración de la clase).

```
import java.util.*;
public class MiClase(){
    ...
    Date fecha = new Date();
    ...
}
```


Paquetes Java: Reglas de Creación

- Para indicar que una Clase C pertenece al Paquete p:
  - la primera línea del fichero C.java es package p;
  - el fichero C.java debe estar en el subdirectorio p
  - El subdirectorio p debe de encontrarse en la lista de la variable de entorno CLASSPATH
- Por ejemplo: La clase MiClase del paquete org.grycap.paquete deberá estar en la ruta:
  - org/grycap/paquete/MiClase.java
- Para que el compilador/enlazador encuentre esa clase durante el proceso de compilación el directorio debe estar incluido dentro de la lista de directorios de la variable de entorno CLASSPATH.
  - Los IDEs (Entorno de Desarrollo Integrado) evitan tener que manipular de manera directa dicha variable.
  - Ejemplo: Herramientas→Preferencias→Librerías (BlueJ)

Paquetes Java: Reglas de Visibilidad

- Los Paquetes tienen varias reglas de visibilidad importantes:

1. Si una Clase o cualquiera de sus componentes no posee modificador de visibilidad alguno (public, private o protected) sólo es visible (puede ser referenciada) para las demás clases dentro del mismo Paquete. Esto se conoce como **ACCESO FRIENDLY**.
2. Sólo las clases públicas de un paquete se pueden usar fuera de él
3. Todas las clases que no forman parte de un paquete, pero se pueden alcanzar a través de la variable CLASSPATH se consideran parte del mismo paquete y el acceso amistoso se aplica entre ellas

Ejercicio de Modificadores de Visibilidad

```
package org.grycap.paquete;
class Clase1{
  private int aPrivado;
  int otroAtributo;
}
```

```
package org.grycap.otropaquete;
import org.grycap.paquete.*;
public class Clase2{
  public static void main(String args[]){
  Clase1 c1 = new Clase1();
}}
```

```
package org.grycap.paquete;
public class Clase3{
  public static void main(String args[]){
  Clase1 c1 = new Clase1();
}}
```

- ¿La creación en Clase 2 es correcta?

Clase 1 NO tiene modificador de visibilidad → Acceso friendly (solo accesible desde clases del mismo paquete). Clase 2 pertenece a otro paquete: org.grycap.paquete.Clase1 is not public in org.grycap.paquete; cannot be accessed from outside package.

- ¿Y la de Clase 3? Acceso Correcto

Modificadores de Visibilidad: Tabla Resumen

- Modificadores de visibilidad aplicados tanto a una clase como a cualquiera de sus miembros (atributos y métodos).

Visibilidad Modificador	Clase	Paquete	Subclase	Mundo
private	Sí	No	No	No
ninguno	Sí	Sí	No	No
protected	Sí	Sí	Sí	No
public	Sí	Sí	Sí	Sí

Reutilización de Código: Tipo de Relación y Mecanismo Java

Relación entre Clases	Mecanismo Java para representar una relación
Existe alguna relación entre las clases, aunque no realicen la misma funcionalidad.	**Paquete** Ejemplos: Clases de **java.util, javax.swing, etc.**
Existe una relación **TIENE UN(A)** o relación de composición entre las Clases	**Definición de Atributos** Ejemplos: Circulo y grupoCirculos
Existe una relación relación **ES UN(A)** o relación jerárquica entre las Clases	**Herencia Java** Ejemplos: clase base Figura y sus derivadas Circulo, Rectangulo, ...

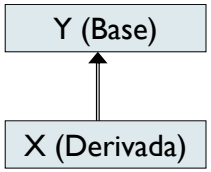
Ejercicio de Persona

Herencia en Lenguajes de POO y su Soporte en Java

- Introducción al concepto de Herencia y su soporte en Java. En particular, se incidirá en los siguientes aspectos:
  - Características y sintaxis de la Herencia en Java
  - La clase **Object** y el modelo de jerarquía de Clases Java
  - Características de una clase derivada
    - Modificadores de visibilidad.
    - Constructores de la derivada.
    - Sobrescritura de métodos.
- Ejemplos: Diseño de la clase **Figura** y reformulación de **Circulo** para que sea una de sus derivadas. Diseño de la clase **GrupoDeFiguras.**

La relación ES UN(A)

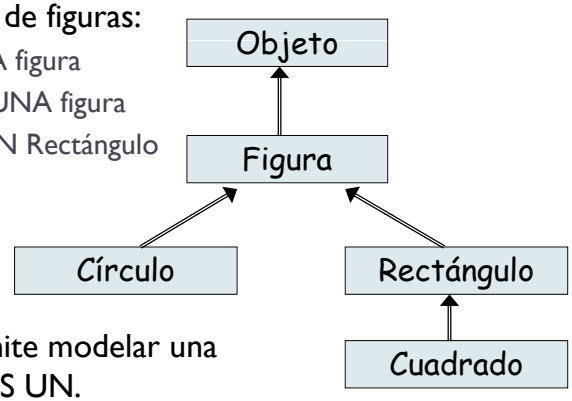
- Si X ES UN(A) Y,
  - se dice que la Clase derivada X es una variación de la Clase base Y
  - se dice que X e Y forman una Jerarquía: X → Y, donde la Clase X es una subClase de Y e Y es una superClase de X
  - la relación es transitiva: si X ES UN(A) Y e Y ES UN(A) Z, entonces X ES UN(A) Z



Ejemplo: ¿Qué relación guardan entre sí un cuadrado, un triángulo, un círculo y un rectángulo?

Ejemplo de relación ES UN(A)

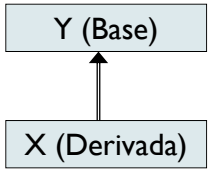
- Jerarquía de clases de figuras:
  - Un círculo ES UNA figura
  - Un rectángulo ES UNA figura
  - Un cuadrado ES UN Rectángulo



- La Herencia permite modelar una relación de tipo ES UN.
- Permite definir una jerarquía de clases.

El Mecanismo de la Herencia

- La Herencia permite la formación de Jerarquías de Clases para reutilizar código.
- X hereda (puede referenciar) todos los atributos y métodos que no sean privados en Y.



- X es una clase completamente nueva e independiente.
  - Los cambios que sufra X NO afectan a Y, lo que simplifica el mantenimiento del código.
- X es de tipo compatible con Y.
  - Una referencia de tipo Y es polimórfica. Sin embargo, no son compatibles ni Y con X, ni X con sus clases hermanas.

Herencia: Sintáxis básica de Java

```
public class Derivada extends Base{
   …
}
```

- Los atributos y métodos no privados de Base se heredan en Derivada (salvo los constructores).
  - Esto significa que dichos atributos pueden ser referenciados y dichos métodos pueden ser invocados.
- Usos del operador ***super:***
  - Referenciar a cualquier componente de Base (atributos, métodos o constructores), aunque no es muy común.
  - Se suele utilizar para invocar constructores de la clase padre y en casos de sobreescritura parcial de métodos.

Ejemplo Básico de Herencia

```
public class Animal{
   protected String queHablo;
    public Animal(){
            queHablo = "Groar!";
    }
   public String rugir(){
    return queHablo;
    }}
```

```
public class Leon extends Animal {
    public Leon(){
            queHablo = "Grawl!";
    }
}
```

```
public class TestLeon {
    public static void main (String args[]){
            Leon l = new Leon();
            System.out.println(l.rugir());
    }
}
```

¿Qué saca por pantalla este código?

Herencia: Métodos Constructores de una Clase Derivada

- La clase Derivada debe definir explícitamente sus propios constructores.
  - En caso contrario, Java proporciona el constructor por defecto:

```
public Derivada(){
    super();
}
```

- Constructor por defecto:
  - Invoca al constructor sin parámetros de base (inicializando los atributos heredados de Base) y luego inicializando al valor por defecto a los atributos heredados de Derivada.
- Implicaciones:
  - Si la clase Base NO define un constructor por defecto (sin argumentos), la clase Derivada obligatoriamente deberá definir un constructor que invoque al de la clase Base con los argumentos correspondientes.

Ejemplo: Definición de Constructores

- Constructores de la clase Figura:

```
public Figura(Color c, String t) {this.color = c;  this.tipo = t;}
public Figura(){ this(COLOR_POR_DEFECTO,TIPO_POR_DEFECTO);}
```

- Constructores de la clase Circulo, en base a los de la clase Figura:

```
 public Circulo(Color c, double r){
        super(c, "Circulo");
        this.radio = r;
}
public Circulo() {
        super(); // ← No es necesaria ya que se realiza automáticamente.
        this.radio = RADIO_POR_DEFECTO;
}
```


Ejercicio: Modificadores de Visibilidad

```
public class Base{
     public int bPublico; protected int bProtegido;  private int bPrivado;
}
public class Derivada extends Base{
    public int dPublico; private int dPrivado;
}
public class Test {
    public static void main(String args[]){
      Base b = new Base();
      Derivada d = new Derivada();
      System.out.println(b.bPublico + " " + b.bProtegido + " " + b.bPrivado);
      System.out.println(d.dPublico + " " + d.dPrivado);
   }
} // NOTA: Se asume que todas las clases pertenecen al mismo paquete.
```

1. En el main de Test, ¿Qué accesos son incorrectos?
2. Si main fuera un método de Base, ¿Qué accesos serían incorrectos?
3. Si main fuera un método de Derivada, ¿Qué accesos serían incorrectos?

Herencia en Java: Sobrescritura de un método de la Clase Base

- Cualquier método no privado de Base que se defina de nuevo en derivada se ***sobrescribe***.
- Para ello, definimos en Derivada un método:
  - Con la misma signatura que en Base (nombre del método y lista de parámetros).
  - Con el mismo tipo de resultado que en Base.
  - Sin añadir excepciones a lista de *throws* del método definido en Base.
- ***Sobreescritura parcial***:
  - Cuando no se desea cambiar completamente el comportamiento del método de la clase Base. Se utiliza *super* para invocar el método de la clase Base.

Ejemplo de sobrescritura del método *toString* de Figura en la clase *Circulo (I)*

- La clase Object es la raíz de la jerarquía de herencia en Java
  - Cualquier clase hereda implícitamente de la clase Object
- Método toString():
  - Especificación: public String toString();
  - Imlementación: [Object.java]
- Método toString de la clase Figura (sobrescritura del método definido en Object):
  - public String toString(){
    return "Figura de tipo " + tipo + " color" + color + "centro " + centro;
    }

Ejemplo de sobrescritura del método *toString* de Figura en la clase *Circulo (II)*

- Método toString en la clase Circulo (sobrescritura del método definido en Figura)
  - public String toString(){
    return "Circulo de radio" + radio + " color" + color + " centro" + centro;
    }
- Método toString en la clase Circulo (sobrescritura parcial del método definido en Figura)
  - public String toString(){
    return super.toString() + " y radio" + radio;
    }

Ejercicio de Yogures

Ejemplo de Sobreescritura: Equals

- El método equals está definido e implementado en la clase Object
  - Permite decidir si dos referencias (objetos) son iguales.
  - Criterio por defecto: Dos referencias son iguales si apuntan exactamente al mismo objeto.
- El criterio puede ser modificado para objetos de una clase sobreescribiendo el método equals (Ej. Figura):
  - Respetar el perfil (cabecera) del método equals
  - Modificar criterio atendiendo a los atributos de la clase.

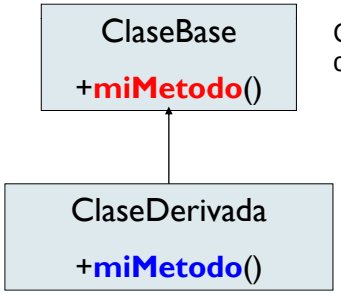
```
public boolean equals(Object x){
      Figura f = (Figura) x;
      return (color.equals(f.color) && tipo.equals(f.tipo));
  }
```

Polimorfismo en Java

- El polimorfismo es una consecuencia directa de la Herencia:
  - Una variable referencia de una clase Base puede referenciar a objetos tanto de la clase Base como de cualquier clase Derivada de esta.
- Toda variable referencia tiene siempre dos tipos:
  - Tipo estático:
    - El tipo con el que ha sido declarada la variable (nunca varía)
  - Tipo dinámico:
    - El tipo del objeto al que referencia en tiempo de ejecución (puede variar)
  - Si ambos tipos no coinciden, entonces la variable es polimórfica.
- Ejemplo:
  - Figura f = new Rectangulo();
  - La variable f es polimórfica:
    - Tipo estático: Figura
    - Tipo dinámico: Rectángulo

Enlace Dinámico

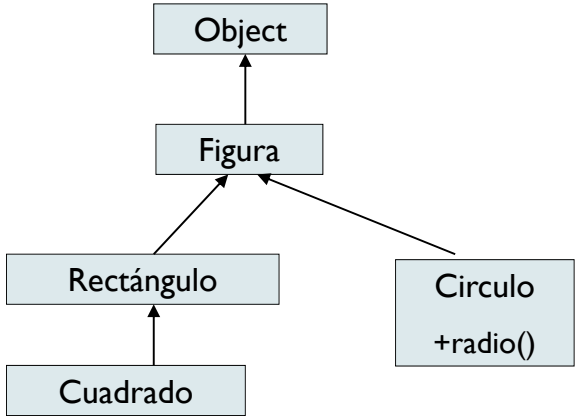
- Ante situaciones de sobreescritura de métodos, el método a ejecutar es el definido por el tipo dinámico de la variable polimórfica.
- Se aplica en tiempo de ejecución, previa comprobación del acceso en la clase del tipo estático.



ClaseBase cb = new ClaseDerivada();
cb.miMetodo();

Se ejecutará el código de *miMetodo* definido en ClaseDerivada.

Jerarquía de Clases: Figuras



Ejemplo de Polimorfismo: Enlace Dinámico

```
public class TestPolimorfismo {
 public static void main (String args[]) {
   Figura f = new Circulo();
   System.out.println("Radio del Circulo:" + f.radio());
 }
```

cannot resolve symbol - method radio

- El compilador únicamente considera el tipo estático de la variable para comprobar el acceso:
  - El método radio no está definido en la clase Figura. Error en tiempo de compilación.
  - Aunque el tipo dinámico de f sea Circulo.

Ejemplo de Polimorfismo: Conversión de Restricción (I)

```
public class TestPolimorfismo {
   public static void main (String args[]) {
     Figura f = new Circulo();
     System.out.println("Radio del Circulo:" + ( (Circulo) f ).radio());
  }
}
```

- Ya que f es de tipo dinámico Circulo, podemos realizar un **casting** a tipo Circulo y, posteriormente, invocar el método radio.
  - El operador casting **()** permite transformar una referencia de un tipo a otro equivalente.
  - Si el tipo destino NO es equivalente (i.e. ((Cuadrado) f)), entonces se produce la excepción **ClassCastException**.

Ejemplo de Polimorfismo: Conversión de Ampliación

```
public class TestPolimorfismo {
 public static void main (String args[]) {
   Figura f = new Circulo();
   System.out.print(f);
}
```

- System.out.println invocará al método *toString* de f, que originalmente está definido en la clase Object aunque ha sido sobrescrito en la clase Figura (y luego en Circulo). Se produce una conversión de ampliación.
- El enlace dinámico garantiza que se ejecutará el correspondiente método de la clase Circulo.

Ejemplo de Polimorfismo: Conversión de Restricción (II)

```
public class TestPolimorfismo {
  public static void main (String args[]) {
    Figura f = new Circulo();
    Circulo c = f;
  }
}
```

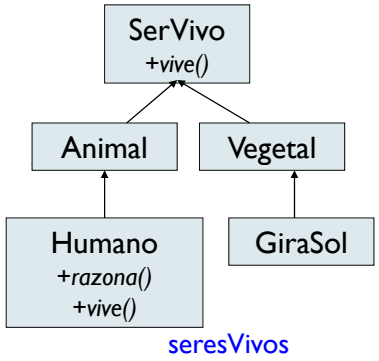
- Aunque el tipo dinámico de f sea *Circulo*, al realizar la última asignación, el compilador indica que *"incompatible types - found Figura but expected Circulo"*.
  - El compilador únicamente tiene en cuenta los tipos estáticos para comprobar que las asignaciones son correctas.
- La asignación se debería realizar de la siguiente manera:
  - Circulo c = (Circulo) f;

Comprobación de Tipo Dinámico: Operador instanceof

- Mostrar por pantalla sólo aquellos elementos que son de tipo Circulo, además de cuántos hay (se asume que hay un vector de Figuras ya inicializado correctamente):

```
public class TestPolimorfismo {
   Figura coleccion[];
 public static void main (String args[]) {
   System.out.println("Cuántos Círculos hay en la colección?");
   int numCirculos = 0;
   for(int i=0; i<coleccion.length; i++) {
            Figura f = coleccion[i];
            if (f instanceof Circulo) {
                     System.out.print( f.toString());
                     numCirculos++;
            }
 }
  System.out.println("Hay " + numCirculos + " circulos.");
}
```

Ejercicio de Herencia y Polimorfismo



- Dada la jerarquía de herencia, ¿Qué instrucciones son correctas y cuales no?

1. SerVivo sv = new Humano();
2. Humano hu = new Humano();
3. Vegetal vg = new GiraSol();
4. sv.razona();
5. sv.vive();
6. hu.razona();
7. hu = vg;
8. sv = vg;

Restringiendo la Especialización: Final

- Empleando la palabra clave **final**
- Los atributos, métodos y las clases pueden ser declaradas como finales.
  - Atributo final: Representa una constante cuyo valor no puede ser modificado.
  - Método final: Impide que el método sea sobreescrito en alguna clase descendiente.
  - Clase final: No pueden tener subclases, representa una hoja dentro del árbol de jerarquía de herencia.
- El uso de **final** genera código más eficiente ya que no se utiliza enlace dinámico y la llamada se resuelve en tiempo de compilación.

Clases Abstractas (I)

- ¿Cómo obligar a que todas las clases derivadas deban forzosamente implementar un determinado método?
  - Empleando la palabra clave **abstract**

```
public abstract class  Figura {
        protected String tipo;
        protected Color color;
        public Figura(String nombre){....}
        public abstract double area();
        ...
}
```

```
public class Circulo extends Figura {
        ...
        public double area(){ return Math.PI * radio * radio;}
        ...
}
```


Clases Abstractas (II)

```
public class Rectangulo extends Figura {
        ...
        public double area(){ return base * altura;}
        ...
}
```

```
public class Cuadrado extends Rectangulo {
        ...
        //No hace falta redefinir el método area
        ...
}
```

- La clase Cuadrado hereda la implementación del método area de la clase Rectángulo. No es necesario cambiar la implementación y, por lo tanto, no hace falta redefinir el método.

Clases Abstractas (III)

- Una clase abstracta …
  - Tiene al menos un método abstracto, etiquetado como tal.
  - Debe ser etiquetada por el programador como abstracta.
  - No puede ser instanciada (vía el operador new).
  - Su(s) constructor(es) pueden ser utilizados (vía super)
  - Puede ser utilizada para definir referencias polimórficas de la clase.
- Una subclase de una clase abstracta puede:
  - Implementar todos los métodos abstractas → Puede ser instanciada
  - No implementar los métodos abstractos → Sigue siendo abstracta
- Utilización de estas clases:
  - Cuando se quiere obligar a que las subclases tengan un cierto comportamiento
- Ejemplo estándar de Clase abstracta: Number

Clase Abstracta: La clase Figura (I)

```
package lasFiguras;
import java.awt.*;  //Para poder referenciar la clase Color
public abstract class Figura { // extends Object
   protected String tipo;
   protected  String color;
   protected  static final String TIPO_POR_DEFECTO = "Circulo";
   protected  static final Color COLOR_POR_DEFECTO = Color.black;

   public Figura(Color c, String t){
      this.tipo = t;  this.color = c;
   }
   public Figura(){
      this(TIPO_POR_DEFECTO, COLOR_POR_DEFECTO);
   }
…
```


Clase Abstracta: La clase Figura (II)

```
…
 public boolean equals(Object x){
       Figura f = (Figura) x;
       return (color.equals(f.color) && tipo.equals(f.tipo));
   }

    public abstract double area();

} //Fin de la clase Figura
```

¿Para qué sirven los constructores de una clase abstracta si no se puede instanciar?

La Clase Rectangulo

```
public class Rectangulo extends Figura {
   protected double base;
   protected double altura;

   public Rectangulo(Color c, double b, double a){
     super(c, "Rectángulo");  this.base = b; this.altura = a;
   }
   public Rectangulo (){
    this(Color.black, 10.0, 10);
   }
   public double area(){ return base * altura;}
   ...
}
```

Interfaces en Java

- En ocasiones es necesario que una clase herede de más de una superclase:
  - Herencia Múltiple
- Java sólo permite que una clase herede explícitamente, vía extends, de una única superclase (distinta de Object).
- En ocasiones es necesario imponer a una clase una funcionalidad genérica que no posee Object.
- Para ello, se debe definir una superclase especial, sólo con métodos abstractos, por lo que forzosamente todas sus subclases los deben implementar si quieren dejar de ser abstractas.

Interfaces en Java: Características

- Una **Interfaz** …
  - Consta sólo de perfiles de métodos que indican qué hace la clase (especificación) pero sin implementación.
  - Permite especificar un comportamiento genérico.
- Para poder imponer el comportamiento que describe:
  - Sus métodos son públicos y abstractos, por lo que han de ser forzosamente implementados por sus subclases. Se dice que una clase **implementa un** interfaz.
  - Sus atributos son públicos y finales, por lo que NO tiene constructores ni puede ser instanciada, PERO sí se pueden definir variables Referencia (polimórficas) del interfaz.
- Una subclase puede implementar todas las interfaces que quiera.

Interfaces en Java: Sintaxis y Utilización

- Sintaxis de Interfaces:
  - Uso de la palabra clave *interface* en lugar de *class*.
  - Se asume que los métodos son public y abstract.

```
public interface MiInterfaz {
   int metodoQueImplementar(Object o);
}
```

- En las subClases del interfaz, se añade a su cabecera implements

```
public class MiClase implements MiInterfaz {
    ...
   public int metodoQueImplementar(Object o){
        /* Implementacion del método */
    }
}
```

¿Qué ocurriría si MiClase implementa el interfaz pero no proporciona código al método?

Ejemplo de Interfaces (I)

- Ejemplo de uso de interfaces:
  - Forzar a que las clases proporcionen una descripción de ellas.

```
public interface Describible {
    String getDescripcion();
}
```

```
public class Libro implements Describible {
        ...
    public String getDescripcion(){
        return "Este objeto es un libro";
    }
        ...
}
```

- Si se añaden nuevos métodos al interfaz, deberán ser implementados en las correspondientes clases.

Ejemplo de Interfaces (II)

- Una clase Java puede implementar varias interfaces.

```
public interface Estudioso {
    Conocimiento estudia(Conceptos c);
}
```

```
public interface Trabajador {
    Dinero trabaja(Tareas t);
}
```

```
public class EstudianteTrabajador implements Estudioso,Trabajador{
        public Conocimiento estudia(Conceptos c){
            //Implementación
        }
        public Dinero trabaja(Tareas t){
            //Implementación
        }
}
```

- Es posible que tener referencias de tipo interfaz que sean polimórficas, i.e.: Estudioso e = new EstudianteTrabajador();

Extensión de Interfaces

- Una Interfaz A puede extender de otra interfaz B.
- Implementar la interfaz A requiere dar código a todos los métodos de A (los de A y los heredados de B).

```
public interface OrdenadorPortable extends Ordenador{
    float capacidadBateria();
}
```

```
public interface Ordenador{
    void encender();
}
```

```
public class PDA implements OrdenadorPortable{
    float capacidadBateria() { … }
    void encender(){ … }
}
```


Uso Alternativo de Interfaces

- Una interfaz también suele ser utilizada para agrupar constantes.

```
public interface DiasSemana
{
    int LUNES = 1 , MARTES = 2;
    String [] NOMBRE_DIAS = { "Lunes" , "Martes"};
}
```

```
public class MisDias implements DiasSemana
{
 public int metodo(){
    System.out.println(NOMBRE_DIAS[LUNES]);
  }
}
```