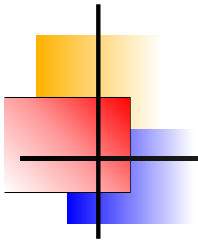


Algoritmos y Estructuras de Datos:

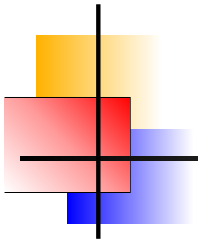
Introducción

Guillermo Román Díez

groman@fi.upm.es

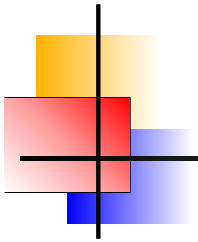
Universidad Politécnica de Madrid

Curso 2015-2016



Repaso Conceptos de Programación en Java

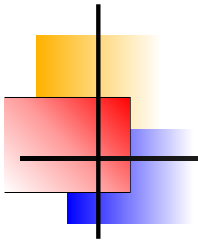
- Expresiones, constantes, métodos, comandos, ...
- Variables, declaración e inicialización, ...
- Bloques de código, flujo de control, condiciones, bucles, ...
- Objetos, `new`, campos, herencia, interfaces, ...
- Tipos básicos, operadores aritméticos, operadores lógicos, ...
- Arrays, declaración y acceso mediante índices, ...
- Excepciones, `try-catch-finally`, ...



Asignación y variables

- Las variables tienen un **tipo** y un **ámbito**
- Declaración de variables:
  - Atributos de clase → Visibles en toda la clase (o más...)
  - Parámetros de los métodos → Visibles en todo el método
  - Las variables locales → Visibles en su **ámbito**
- Inicialización por defecto
  - Si son de tipo primitivo → 0, 0.0, '\0'
  - Si son de tipo referencia → `null`
- Asignación de variables: **l-value** vs **r-value**

```
int x,y;
y = 7;
x = y = 3;
// Valor de x e y?
while ((x=y/2) != 0) {...}
```

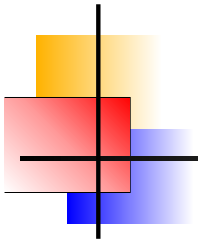


EJERCICIO: ASIGNACIONES

```
1  public static void main(String [] args) {
2     int v1 = 3;
3     int v2 = 5;
4     int v3;
5
6     v3 = v1;
7     v1 = v2;
8     v3 = v1 = 6;
9     v3 = 6 = v5;    // Error
10    v3 = 7;
11 }
```

Ejercicio

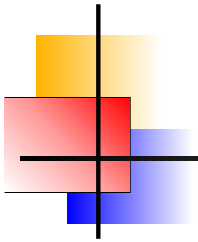
¿cuáles son los valores de las variables después de ejecutar cada punto de programa?



Ámbito de las variables

- Java tiene sus reglas de visibilidad establecidas
  - Otros lenguajes pueden tener sus propias reglas de visibilidad
- Puede ocurrir **shadowing**: Si hay dos variables que se llaman igual una puede **ocultar** a la otra

```
public class C {
  int x;
  public C(int x) { x = x; } // ??
  public void m1(int x) {
    for (int x = 1; x < 10; x++) // ??
      x++;
    if (x < 0) {
      int x = 1; // ??
      this.x = x; // ??
    }
  }
  public void m2(int c) { x = c; } // ??
}
```



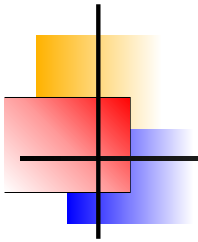
Paso de Parámetros por Valor

- Un ejemplo de código para intercambiar dos variables:

```
int x = 5;
int y = 4;
// codigo del intercambio ("swapping")
int tmp = x;
x = y;
y = tmp;
```

Pregunta

¿Podemos hacer esto llamando a un método que reciba `x` e `y` como parámetros?

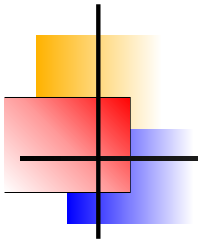


Paso de Parámetros por Valor

```
public static void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

public static void main(String [] args) {
  int x = 5;
  int y = 7;
  swap(x,y);
  System.out.println(x + " " + y);
}
```

Pregunta

¿Qué saldrá por pantalla?

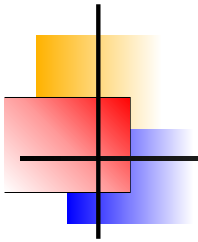


Ejercicio Variables

Ejercicio

Dibujar las variables durante la ejecución

```
public static void main(String [] args) {
  int a = 3;
  int b = 3;
  a = 2;

  int c = 3;
  int d = c;
  c = 2;

  int e = 3;
  e = 2;
  int d = e;
}
```



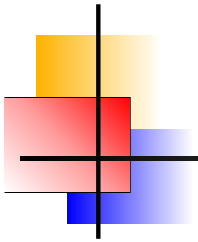
Evaluación de expresiones

- Java evalúa las expresiones **en cortocircuito**
- Si tenemos un `&&` corta en el momento en el que no se cumpla una condición $\rightarrow$ evalúa a **false**

```
v != null && v[i] == 0
```

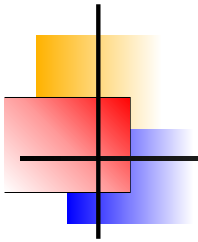
- Si tenemos un `||` corta en el momento en el que se cumpla una condición $\rightarrow$ evalúa a **true**

```
v == null || v[i] == 0
```



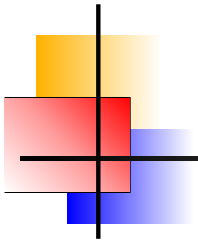
Arrays (vectores)

- Podemos declarar arrays de cualquier tipo
  - Tipos primitivos (`int`, `double`, `char`)
  - Como referencias (objetos, interfaces, clases abstractas, arrays)
- La declaración de la variable no establece el tamaño
- Es necesario inicializar el tamaño antes de utilizarlo
  - En caso contrario `NullPointerException`
- El rango de elementos de un array es `[0..longitud-1]`
- Si accedemos a una posición negativa o mayor que el tamaño del array $\rightarrow$ `ArrayIndexOutOfBoundsException`



Ejemplos Arrays

```
int [] a;
a[0] = 4;

int [] a2 = null;
a[2] = 10;

int [] b = new int[20];
b[0] = 7;
int [] c = new int[getIntegerFromUser()];

String [] d = new String[10];
d[0] = new String("Hola");

int [] e = new int[0];
int [] f = { };
int [] g = { 7, 9, 8 };

Clase [] h = new SubClase[5];
```

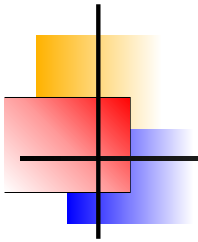


Paso de Parámetros por Referencia

```
public static void swap(int array[]) {
  int tmp = array[0];
  array[0] = array[1];
  array[1] = tmp;
}

public static void main(String [] args) {
  int array[] = {3,4};
  swap(array);
  System.out.println(array[0] + " " + array[1]);
}
```

Pregunta

¿Qué saldrá por pantalla?



Bucles

- Bucle `while`

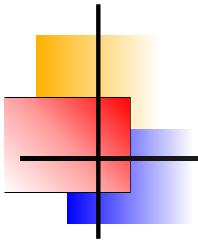
```
INIT
while (COND)
  BODY
```

- Bucle `do-while`

```
do
  BODY
while(COND)
```

- Bucle `for`

```
for (INIT; COND; POST)
  BODY
```



Equivalencia Bucles

- Equivalencia `while` - `do-while`

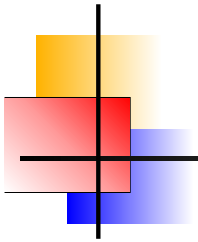
```
BODY
while (COND) {
  BODY
}
```

```
do {
  BODY
}
while (COND)
```

- Equivalencia `while` - `for`

```
for (INIT; COND; POST) {
  BODY
}
```

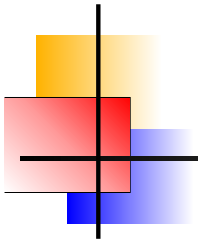
```
INIT
while(COND) {
  BODY
  POST
}
```



Recorrido de Arrays

Ejercicio

Recorrer un Array con un bucle `while`

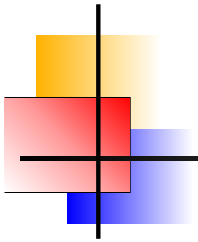


Recorrido de Arrays

Ejercicio

Recorrer un Array con un bucle `while`

```
int i = 0;
while (i < array.length) {
  System.out.println("Posicion " + array[i]);
}
```



Recorrido de Arrays

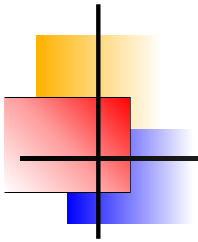
Ejercicio

Recorrer un Array con un bucle `while`

```
int i = 0;
while (i < array.length) {
  System.out.println("Posicion " + array[i]);
}
```

Ejercicio

Recorrer un Array con un bucle `for`



Recorrido de Arrays

Ejercicio

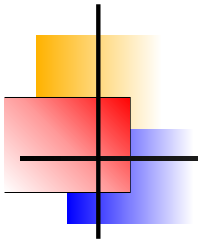
Recorrer un Array con un bucle `while`

```
int i = 0;
while (i < array.length) {
  System.out.println("Posicion " + array[i]);
}
```

Ejercicio

Recorrer un Array con un bucle `for`

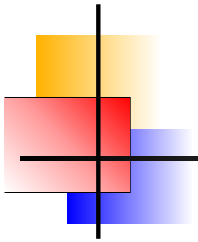
```
for (int i = 0; i < array.length; i ++) {
  System.out.println("Posicion " + array[i]);
}
```



Busqueda en un Array

Ejercicio

¿Cómo podemos buscar el elemento '*' en un array?

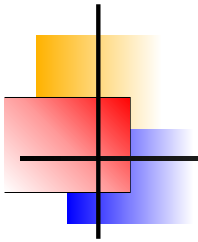


Busqueda en un Array

Ejercicio

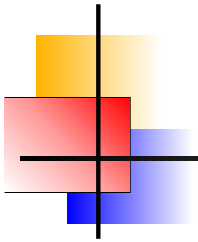
¿Cómo podemos buscar el elemento '*' en un array?

```
int i = 0;
while (i < v.length && v[i] != '*') {
  i++;
}
return i < v.length;
```



Salida de los bucles

- No es aconsejable salir de los bucles con `break` o `return`
- Tampoco es aconsejable el uso de `continue`
- Complica la comprensión y mantenimiento del código
- Hay que razonar sobre todo el bucle y no sólo sobre la condición
- No se respeta la invariante de bucle



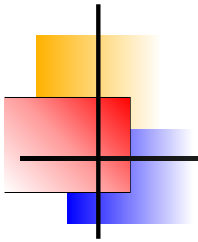
Clases, objetos y variables

- Las **clases** definen los **atributos** y **métodos** que tendrán los objetos de la clase

Ejemplo

Ver el código de la clase `Vehiculo`

- Los **objetos** se crean en tiempo de ejecución con `new`
- Cuando se declara una variable de tipo clase o interfaz se inicializa a **null**
  - Si se intenta usar ese objeto → `NullPointerException`
  - Es necesario hacer un `new` o utilizar una asignación a un objeto creado previamente
- Las variables de tipo clase o interfaz referencian a objetos
  - Se pueden cambiar el objeto al que referencia una variable
  - El *recolector de basura* (*garbage collector*) eliminará los objetos que no son referenciados



Ejercicio objetos

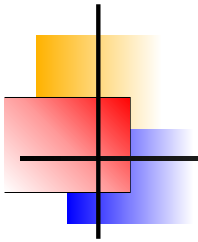
Ejercicio

Dibujar los objetos y las referencias de las variables

```
Automovil v1 = new Automovil(10,10,10);
Automovil v2 = new Automovil(20,20,20);
Automovil v3;

v3 = v1;
v1 = v2;
v3 = new Automovil(30,30,30);
v2 = null;
```

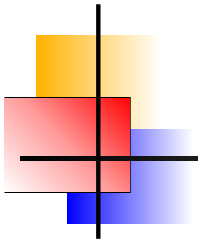
Pregunta

¿Liberará el Garbage Collector (recolector de basura) algún objeto?



Identidad vs. estado de un objeto

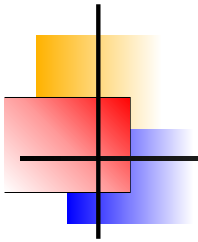
- La **identidad** de un objeto es la **dirección de memoria** en la que está el objeto
- El **estado** de un objeto son los valores que tienen los **atributos** de un objeto
- Las comparaciones son diferentes
  - Para comparar la identidad comparamos con `==`
  - Para comparar el estado utilizamos el método `equals`



Identidad vs. estado de un objeto

- La **identidad** de un objeto es la **dirección de memoria** en la que está el objeto
- El **estado** de un objeto son los valores que tienen los **atributos** de un objeto
- Las comparaciones son diferentes
  - Para comparar la identidad comparamos con `==`
  - Para comparar el estado utilizamos el método `equals`

```
Auotomovil v1 = new Auotomovil(10,10,10);
Vehiculo v2 = v1;
System.out.println(v1.equals(v2))   // cierto
System.out.println(v1 == v2)        // cierto

v2 = new Auotomovil(10,10,10);
System.out.println(v1.equals(v2))   // cierto
System.out.println(v1 == v2)        // falso
```



Tipos básicos y clases envoltorio

- Los tipos básicos se comportan de forma diferente a los objetos
- Las comparaciones de tipos básicos siempre son con ==
- Hay unas clases *envoltorio* que se comportan como clases y como tipos básicos
  - `Integer`, `Double`, `Float`, ...
  - Se pueden comparar con ==, pero mucho cuidado!! Mejor usar `equals`
  - Se pueden crear objetos asignando directamente valores (sin hacer `new`)

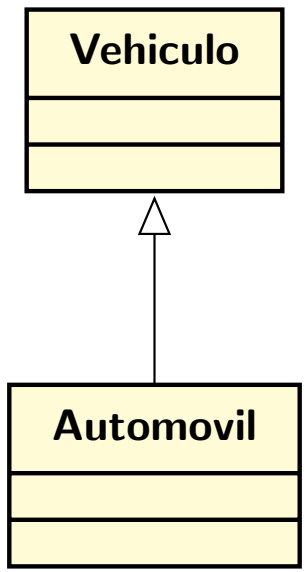
Ejemplo

Ver código `TestEnvoltorio`

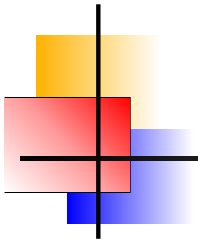
Herencia: relación "es-un"

- Un objeto de clase `Automovil` también **es-un** objeto de la clase `Vehiculo`

```
public class Vehiculo { ... }
public class Automovil extends Vehiculo { ... }
```

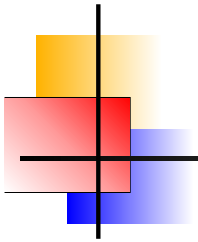


- La clase `Automovil` hace más específico el `Vehiculo` con nuevos atributos y métodos (o sobrescribiendo)



Herencia: Utilidad

- La herencia permite **reutilizar** código
  - Un `Automovil` tiene todos los atributos y métodos de un `Vehiculo`
- El **polimorfismo** permite a una misma variable referenciar objetos de una clase o de cualquiera de sus subclases
  - Una variable de tipo `Vehiculo` puede apuntar a objetos de tipo `Automovil`
- El **enlazado dinámico** permite invocar a un mismo método y dependiendo del objeto instanciado se ejecutará el método de una clase o de otra

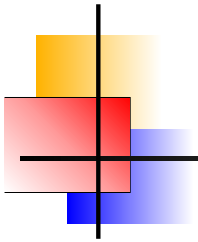


Upcasting

- Es un mecanismo de compilación para **convertir el tipo** yendo **hacia arriba** (up) en la jerarquía de clases
- Se hace de forma automática (no es necesario hacerlo explícito)

```
Vehiculo v = new Automovil ();
```

  - La variable `v` es de tipo vehículo
  - El objeto se ha creado de tipo `Automovil`
- La variable `v` referencia a un objeto de tipo `Automovil`
- Esto se produce realmente en tiempo de ejecución
- La variable `v` es polimórfica porque puede referenciar objetos de tipo `Automovil` como de tipo `Vehiculo`



Upcasting

- También puede haber upcasting en el paso de parámetros (o en el return) a métodos

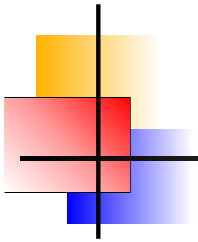
  `public Vehículo m (Vehículo v1)  ...`

  - El método puede invocarse con un objeto de tipo `Vehiculo` o `Automovil`
  - El método puede devolve con un objeto de tipo `Vehiculo` o `Automovil`

Ejercicio

¿Qué llamadas serían válidas para estos métodos?

```
public Automovil m1() { return new Automovil(); }
public Vehiculo  m2() { return new Automovil(); }
public void      m3(Vehiculo v) { ... }
```



Downcasting

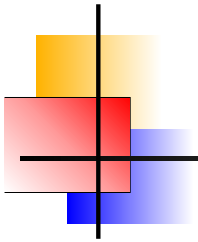
- Mecanismo para convertir el tipo (casting) yendo hacia abajo en la jerarquía de clases
- Debe hacerse de forma explícita en el programa

```
public class Vehiculo { ... }
public class Automovil extends Vehiculo { ... }
public class Aeroplano extends Vehiculo { ... }
```

Pregunta

¿son correctos los siguiente castings? ¿es upcasting o downcasting?

```
Vehiculo v = new Automovil();
Automovil a1 = v;
Automovil a2 = (Automovil) v;
Aeroplano p2 = (Aeroplano) v;
```

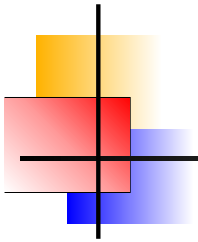


Errores de casting

- En **tiempo de compilación**
  - Se detectan upcastings incorrectos si el objeto creado no está en la jerarquía de clases de la variable
  - Se pueden detectar downcastings incorrectos si no están en la jerarquía
  - No se pueden detectar los tipos que tendrán las variables en tiempo de ejecución
- En **tiempo de ejecución**
  - El downcasting implica decir **tranquilo, sé lo que hago!**, pero...
  - Si el downcasting no es correcto $\rightarrow$ **ClassCastException**

Ejercicio

Ver código de `TestCasting`. ¿son correctos los castings? ¿fallarán en tiempo de complicación o de ejecución?



Instanceof

- Como hemos visto el downcasting es peligroso cuando no sabemos exactamente el objeto referenciado por una variable

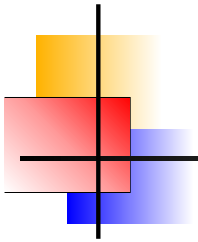
```
public void metodo (Vehiculo v) {
  Automovil a = (Automovil) v; // correcto??
}
```

- Para comprobarlo podemos usar **instanceof**

```
if (v instanceof Automovil)
  Automovil a = (Automovil) v;
else if (v instanceof Aeroplano)
  Aeroplano p = (Aeroplano) v;
```

- **instanceof** no implica hacer las cosas bien

```
if (v instanceof Automovil)
  Aeroplano a = (Aeroplano) v;
```



Enlazado dinámico y sobreescritura

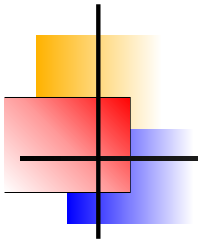
¿qué es la herencia múltiple?

```
public class Vehiculo { ... }
public class Automovil extends Vehiculo { ... }
public class Aeroplano extends Vehiculo { ... }
public class Batmovil  extends Automovil,
    Aeroplano { ... }
```

Pregunta

¿de quien hereda `Batmovil` los métodos de `Vehiculo`

- Hay una ambigüedad que deber resolverse
- Java **NO** permite **herencia mútliple**
- Para esto Java proporciona los **interfaces**, como veremos más adelante



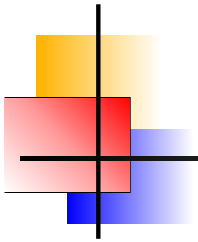
SOBREESCRITURA

Pregunta

¿qué es la sobreescritura (overriding)

- Las subclases pueden **redefinir** un método (o un atributo) de una superclase para especializarlo

```
public class Vehiculo {
  public void display () {
    System.out.println("Soy un Vehiculo");
  }
}
public class Automovil extends Vehiculo {
  public void display () {
    System.out.println("Soy un Automovil");
  }
}
```



Enlazado dinámico y sobreescritura

Pregunta

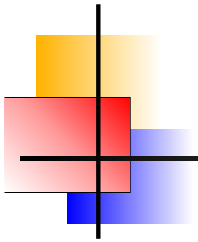
¿cuál es el objetivo fundamental de la sobreescritura?

- El **enlazado dinámico**: permitir invocar a un método sobre una variable polimórfica de forma que el método ejecutado dependa del tipo el objeto referenciado

```
Vehiculo v[] = new Vehiculo[3];
v[0] = new Automovil();
v[1] = new Aeroplano();
v[2] = new Barco();
for (int i = 0; i < v.length; i++)
  v[i].display();     // dibuja el objeto
```

Pregunta

¿qué método será invocado en cada iteración?



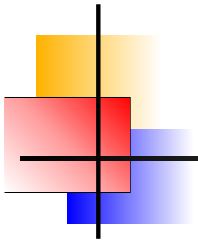
Enlazado dinámico

Pregunta

¿qué ocurre si un método no se sobrescribe en una clase?

- Si una clase no sobrescribe un método se subirá por la jerarquía hasta la primera clase que lo implemente

```
public class Vehiculo {
  public void display () {...}
}
public class Automovil extends Vehiculo {
  public void display () {...}
}
public class Aeroplano extends Vehiculo {
  public void display () {...}
}
public class Barco extends Vehiculo {
}
```

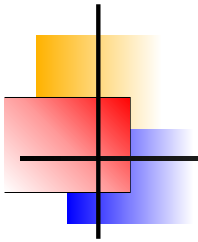


Enlazado dinámico sin sobreescritura

Pregunta

¿todas las líneas son correctas? ¿qué método ejecutan?

```
public class Vehiculo {
  public void display() { ... }
}
public class Automovil extends Vehiculo {
  public int matricula() { ... }
}

Vehiculo v = new Automovil();
v.display();
v.matricula();

Automovil a = new Automovil();
a.display();
a.matricula();
```



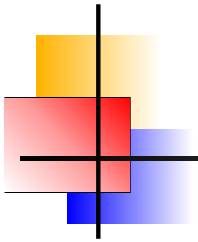
Sobrecarga vs. sobreescritura

Pregunta

¿cuál es la diferencia entre la sobrecarga y la sobreescritura?

- La sobrecarga permite nombrar con el mismo identificador diferentes métodos (o variables)
- La sobrecarga se resuelve en tiempo de compilación y lo marca el tipo de la variable utilizada para la llamada
  - El compilador la resuelve en tiempo de compilación mediante los parámetros utilizados
- El tipo devuelto por el método no *diferencia* los métodos, tiene que ser con los parámetros de llamada

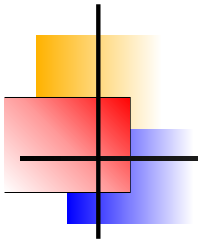
```
public class Vehiculo {
  public int display(Vehiculo v) {...}
  public Vehiculo display(Vehiculo v) {...} //ERROR
}
```



Interfaces

- Un **interfaz** es un conjunto de cabeceras de métodos
- **NO** tienen código ni tiene constructores
- **NO** se pueden crear objetos de tipo interfaz
- Una clase que **implementa** un interfaz tiene que implementar todos sus métodos

```
public interface II {
  public void m1();
  public void m3();
}
public class C implements I {
  public void m1() { /*CODIGO */ }
  public void m3() { /*CODIGO */ }
}
      I var = new C ();
      var.m1 ();
```



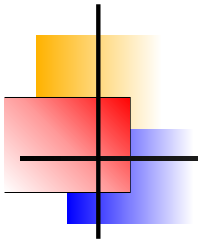
Interfaces

- Un interfaz puede extender a otros interfaces

```
public interface I1 {
  public void m1(String s);
  public void m2();
}
public interface I2 extends I1 {
  public void m3();
}
```

Pregunta

¿qué métodos tiene el interfaz `I2`? ¿tiene sentido re-declarar alguno de los métodos en `I2`?

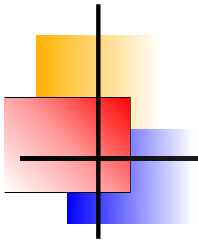


Interfaces

- Una misma clase puede implementar muchos interfaces
- Pero sólo puede extender de una clase
- El tipo con el que se declara una variable indica qué métodos podrán ser invocados

Ejemplo

Ver clase `TestInterfaces` y probar las diferentes ejecuciones



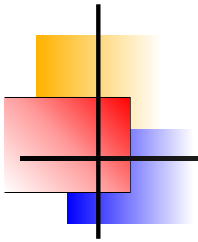
Tipos Genéricos

- Las variables de tipo genéricas se usan para generalizar el tipo de los elementos de los TADs contenedores
- Por ejemplo: Tenemos un interfaz para implementar un par de `Integer` y `String`

```
public interface PairIntegetString {
  public Integer getX();
  public String  getY();
  public void    putX(Integer x);
  public void    putY(String y);
  public void    replaceBy(Pair p);
}
```

Pregunta

¿qué podríamos hacer para tener un interfaz que tenga un par de Strings?

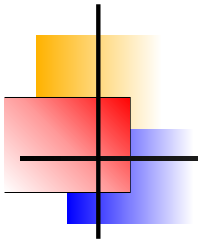


Tipos Genéricos

- Podemos **generalizar** el concepto de **par** utilizando el concepto de **tipos genéricos**

```
public interface Pair<X,Y> {
  public X    getX();
  public Y    getY();
  public void putX(X x);
  public void putY(Y y);
  public void replaceBy(Pair<X,Y> p);
}
```

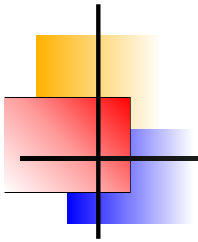
- Las variables 'X' e 'Y' son variables de tipo
- Son diferentes a las variables ordinarias (variables de valores) porque sólo pueden aparecer en una expresión de tipo
  - Al declarar una clase o un interfaz
  - En el tipo de un atributo o de una variable local, en el tipo de un método, en el tipo de un parámetro de un método, ...



Métodos Genéricos

- Java permite métodos genéricos
- La clase no es genérica, pero el método sí lo es

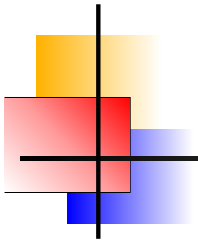
```
public class UnaClase {
  public <E> E identidad(E elem) { return elem; }
}

public static void main(String [] args) {
  UnaClase u = new UnaClase();

  System.out.println(u.identidad(3).toString());
  System.out.println(u.identidad("Hola").toString());
}
```



- En Java no pueden crearse Arrays de tipo genérico (sí declararse)

```
public class ArraySolito<X> {
  private X [] xs;

  public ArraySolito(int size) {
    this.xs = new X[size];              // no compila
  }
}
```

- Hay algunas soluciones *artesanas* para crear arrays de genéricos pero que tienen algunos problemas



Arrays genéricos

- Crear un array de 'Object' y hacer "downcasting"
  - Da problemas al devolver el array como String porque en el fondo son Objects

```
public ArraySolito(int size) {
  this.xs = (X []) new Object[size];
}
```

- Usar la clase `Array` del paquete `java.lang.reflect`

```
public ArraySolito(int size) {
  this.xs = (X []) Array.newInstance("hola".getClass()
      ,size);
}
```