Algoritmos y Estructuras de Datos:

Introducción

Lars-Åke Fredlund
(email: larsake.fredlund@upm.es)

Coordinador: Guillermo Román Díez
(email: guillermo.roman@upm.es)

Universidad Politécnica de Madrid

Curso 2021-2022

Normas de la Asignatura

- La asignatura se compone de dos partes: **teoría** y **prácticas**
  - Ambas partes deben superar su correspondiente nota mínima para poder superar la asignatura
  - Una vez superada la nota mínima, ésta se guarda para la convocatoria de Julio y para años posteriores
- **Teoría (55 % de la nota final)**
  - 2 Exámenes (50% de la nota teoría)
    - ★ Examen 1: 10 de Noviembre
    - ★ Examen 2: 21 de Enero
  - **Nota mínima teoría 4.5**
  - La nota de cada parte se guarda hasta la convocatoria de Julio
  - No hay nota mínima en cada parte de teoría
- Los alumnos interesados en la evaluación por prueba final deben solicitarlo por escrito al coordinador hasta el **25 de Septiembre**

Normas de la Asignatura

- **Prácticas (45 % de la nota final)**
  - **6 entregas individuales** (15% nota prácticas)
  - **7 Laboratorios por parejas** (85% nota prácticas)
  - **Nota mínima prácticas 4**
  - Las fechas **previstas** de los laboratorios están en la guía de aprendizaje y en Moodle
  - Las prácticas dispondrán de un tester automático, si la práctica no pasa **todas** las pruebas se considerará *no apta*
  - Pasar los tests no quiere decir que la práctica esté aprobada, hay una corrección manual posterior
  - La fecha límite de entrega de cada práctica se publicará en la guía del laboratorio correspondiente
  - Los ejercicios aceptados con posterioridad a la fecha límite tendrán una **reducción** en su nota del **20% por cada 24 horas**
  - La entrega de cada práctica NO es obligatoria

Normas de la asignatura

- Los alumnos con notas de prácticas o teoría guardadas de años anteriores, no necesitan cursar de nuevo la parte guardada
- Comprobad vuestra calificación en los listados cuando se publiquen en Moodle (carpeta calificaciones)
- Si entregan alguna práctica o se presentan a alguno de los exámenes perderán la nota guardada

- **La nota final debe ser superior a 5 para aprobar la asignatura**

Tutorias en tiempos de COVID

- Normalmente online usando Microsoft Teams
- Podeis pedir tutoria con qualquier profesor usando email/chat de Teams

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`, ...

Tipos en Java

- Hay dos familias de tipos en Java
- **Tipos primitivos**
  - Almacenan directamente el valor al que son inicializados o asignados
  - Los tipos básicos son `int`, `double`, `char`, ...
- **Referencias**
  - Almacenan la dirección de memoria del objeto o array al que están asignados
  - La dirección de memoria se conoce como **puntero** o **referencia**

Asignación y variables

- Las variables tienen un **tipo** y un **ámbito**
- Declaración de variables:
  - Atributos (campos) → 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:

```
int x,y;
y = 7;
x = y = 3;
// Valor de x e y?
```

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

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; } // ??
}
```

Evaluación de expresiones booleanas

- Java evalúa las expresiones booleanas **en cortocircuito**
- No evalúa la expresión completa si no es necesario

Evaluación de expresiones booleanas

- Java evalúa las expresiones booleanas **en cortocircuito**
- No evalúa la expresión completa si no es necesario
- Si tenemos un `&&` *corta* la evaluación en el momento en el que no se cumpla una condición $\rightarrow$ evalúa a todo a **false**

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

Evaluación de expresiones booleanas

- Java evalúa las expresiones booleanas **en cortocircuito**
- No evalúa la expresión completa si no es necesario
- Si tenemos un `&&` *corta* la evaluación en el momento en el que no se cumpla una condición $\rightarrow$ evalúa a todo a **false**

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

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

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

- Al hacerse de izquierda a derecha, nos permite tener información sobre la parte "derecha" de la expresión

Arrays (vectores)

- Podemos declarar arrays de cualquier tipo
  - Tipos primitivos (`int`, `double`, `char`)
  - Referencias (objetos, interfaces, clases abstractas, arrays)
- Notad: la declaración de la variable no crea ningún array
- Es necesario crear un array (con new), y guardar su referencia en el variable, antes de usar el variable
  - 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[2];
b[2] = 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 Clase[5];
h[2] = new SubClase();
```

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]);
  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]);
  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]);
  i ++;
}
```

Ejercicio

Recorrer un Array con un bucle `for`

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

Salida de los bucles

- No es recomendable salir de los bucles con `break` o `return`
- Tampoco es recomendable el uso de `continue`
- Complica la comprensión y mantenimiento del código
- Implica leer todo el bloque para determinar todas las condiciones de salida que no están en la condición de terminación
- No se garantiza que se respete la invariante de bucle

Busqueda en un Array

Ejercicio

¿Implementar un método que busque si el elemento 'e' está en el array 'a'?

```
private static boolean member (char[] a, char e) {
   ...
}
```

- OJO!! El static no siempre hay que ponerlo

Pregunta

¿para qué sirve static en un método? ¿y en un atributo?

Busqueda en un Array

Ejercicio

¿Implementar un método que busque si el elemento 'e' está en el array 'a'?

```
private static boolean member (char[] a, char e) {
  if (a == null) {
    throw new IllegalArgumentException ("a es null");
  }
  if (a.length == 0) {
    return false;
  }
  int i = 0;
  while (i < a.length && a[i] != e) {
    i++;
  }
  return i < a.length;
}
```

Paso de Parámetros en Java

- 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 en Java

- El paso de parámetros en Java siempre es **por valor**
- Se hace una copia del valor de la variable que se pasa como parámetro

- Para los **tipos básicos** implica que no se puede modificar el valor de una variable de tipo básico dentro de un método

- Para las **referencias** se pasa una copia de la referencia, es decir, de la dirección de memoria

Pregunta

¿se podría cambiar el contenido de un array o de un objeto dentro de un método que lo recibe como parámetro

Clases, objetos y variables

- Las **clases** definen los **atributos** y **métodos** que tendrán los objetos de la clase
- 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 $\rightarrow$ `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 puede reasignar 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

¿Cuántos objetos se crean?
¿Liberará el Garbage Collector (recolector de basura) algún objeto?

Identidad vs. estado de un objeto

- La **identidad** de un objeto la define la **dirección de memoria** en la que está almacenado el objeto
- El **estado** de un objeto lo conforman 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`

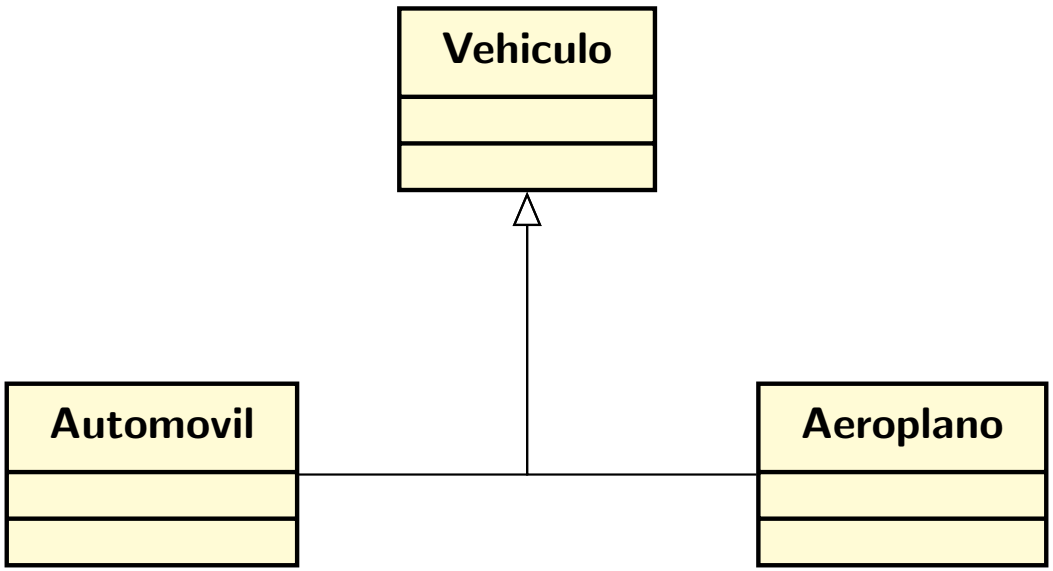
Tipos básicos y clases envoltorio

- 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 intentar comparar con ==, pero **no siempre funciona**. Usad siempre `equals` **para compararlos**
  - Se pueden crear objetos asignando directamente valores (sin hacer `new`)

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



- Las clases `Automovil` y `Aeroplano` ambos hacen 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` o de tipo `Aeroplano`
- 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 tanto a objetos de tipo `Automovil` como de tipo `Vehiculo`

Upcasting

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

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

Ejercicio

¿Qué llamadas serían válidas para ese método?

Upcasting

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

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

Ejercicio

¿Qué llamadas serían válidas para ese método?

- El método puede invocarse con un objeto de tipo `Vehiculo` o `Automovil` (o `Aeroplano`)
- El método puede devolver un objeto de tipo `Vehiculo` o `Automovil` (o `Aeroplano`)

Downcasting

- Mecanismo para convertir el tipo (casting) **hacia abajo** en la jerarquía de clases
- Debe hacerse **de forma explícita** en el programa usando el operador: `(TipoDestino)`

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

Downcasting

- Mecanismo para convertir el tipo (casting) **hacia abajo** en la jerarquía de clases
- Debe hacerse **de forma explícita** en el programa usando el operador: `(TipoDestino)`

```
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 tipo de la variable no está en la jerarquía de clases de la variable
  - Se detectan downcastings incorrectos si no se puede garantizar que el tipo al que se está conviertiendo está en la jerarquía
  - No se pueden detectar los tipos que tendrán las variables en tiempo de ejecución por lo que se pueden escapar errores en compilación
- En **tiempo de ejecución**
  - El downcasting implica decir **tranquilo, sé lo que hago!**...
  - ...pero si el downcasting no es correcto $\rightarrow$ **ClassCastException**

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??
}
```

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

- 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;
```

Herencia múltiple

Pregunta

¿Qué es la herencia múltiple?

Herencia múltiple

Pregunta

¿Qué es la herencia múltiple?

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

Herencia múltiple

Pregunta

¿Qué es la herencia múltiple?

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

Pregunta

¿de quién hereda `Batmovil` los métodos?

Herencia múltiple

Pregunta

¿Qué es la herencia múltiple?

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

Pregunta

¿de quién hereda `Batmovil` los métodos?

- 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)?

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?

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

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 sin sobreescritura

Pregunta

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

```
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

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

```
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 {
}
```

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

Enlazado dinámico

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?

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
- Ojo! El tipo devuelto por el método *NO diferencia* los métodos, tiene que ser con los parámetros de llamada

Interfaces

Interfaces

- Un **interfaz** es un conjunto de cabeceras de métodos
- **NO** tienen código ni tiene constructores
- **NO** se pueden instanciar objetos (con `new`) de tipo interfaz
- Una clase que **implementa** un interfaz tiene que implementar todos sus métodos

```
public interface I {
  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 los métodos `m1` o `m2` en el intrefaz `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

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 PairStringInteger {
  public String   getX();
  public Integer  getY();
  public void     putX(String x);
  public void     putY(Integer y);
}
```

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 PairStringInteger {
  public String   getX();
  public Integer  getY();
  public void     putX(String x);
  public void     putY(Integer y);
}
```

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);
}
```

- El interfaz `Pair<X,Y>` utiliza genéricos
- `X` e `Y` son de tipo **genérico** y pueden aparecer en una expresión de tipo, es decir:
  - Al declarar una clase o un interfaz con `< ... >`
  - 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 la definición de métodos genéricos
- La clase no tiene por qué ser genérica, pero el método sí lo es

```
public class UnaClase {
  public <E> List<E> copiar(List<E> l) { return ...; }
}

  List<String> ls= new ArrayList({...});
  List<Integer> lc = new ArrayList({...});

  List<String> ls2 =  copiar(ls);
  List<Integer> lc2 = copiar(lc);

  List<Integer> le = copiar(ls); // Error!!
  // cannot convert from List<String> to List<Integer>
```