

Motivación

Conceptos

Tipos y sistemas  
de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y  
control de flujo

Paso de  
parámetros

Alcance de las  
variables

Gestión de  
memoria

Paradigmas  
de programación

Imperativo

Declarativo

OO

Concurrente

Otros  
paradigmas

Basado en  
interacción

Bibliografía

# Tema 1. Introducción (Parte 1)

## Lenguajes, Tecnologías y Paradigmas de Programación (LTP)

DSIC, ETSInf



#### Motivación

#### Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

#### Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

#### Otros paradigmas

Basado en interacción

#### Bibliografía

- 1 Motivación
- 2 Conceptos esenciales en lenguajes de programación
  - Tipos y sistemas de tipos
  - Polimorfismo
  - Reflexión
  - Procedimientos y control de flujo
  - Gestión de memoria
- 3 Principales paradigmas de programación: imperativo, funcional, lógico, orientado a objetos, concurrente
  - Paradigma imperativo
  - Paradigma declarativo
  - Paradigma orientado a objetos
  - Paradigma concurrente
- 4 Otros paradigmas: basado en interacción, emergentes
  - Paradigma basado en interacción
- 5 Bibliografía

# Objetivos del tema

## Motivación

### Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

### Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

### Otros paradigmas

Basado en interacción

### Bibliografía

- Conocer la evolución de los lenguajes de programación (LP) y cuáles han sido sus aportaciones más importantes en cuanto al impacto en el diseño de otros lenguajes.
- Entender los principales paradigmas de programación disponibles hoy en día y sus principales características.
- Comprender los distintos mecanismos de abstracción (genericidad, herencia y modularización) y paso de parámetros.
- Identificar aspectos fundamentales de los LP: alcance estático/dinámico, gestión de memoria.
- Entender los criterios que permiten elegir el paradigma/lenguaje de programación más adecuado en función de la aplicación, envergadura y metodología de programación.
- Entender las características de los LP en relación al modelo subyacente (paradigma) y a sus componentes fundamentales (sistemas de tipos y clases, modelo de ejecución, abstracciones).
- Entender las implicaciones de los recursos expresivos de un LP en cuanto a su implementación.

# Una historia que empezó en 1950

## Motivación

### Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

### Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

### Otros paradigmas

Basado en interacción

### Bibliografía

## AÑOS 50:

- Tiempo del programador barato, máquinas caras:  
*keep the machine busy*
- Cuando no se programaba directamente el hardware, el programa se compilaba a mano para obtener la máxima eficiencia para un hardware concreto:  
*conexión directa entre lenguaje y hardware*

## ACTUALIDAD:

- Tiempo del programador caro, máquinas baratas:  
*keep the programmer busy*
- El programa se construye para ser eficiente y se compila automáticamente para generar código portable que sea, a la vez, eficiente:  
*conexión directa entre diseño del programa y lenguajes: objetos, concurrencia, etc.*

## Motivación

### Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

### Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

### Otros paradigmas

Basado en interacción

### Bibliografía

# Enseñanza de los LP

Tres aproximaciones

- 1 Programación como un oficio
- 2 Programación como una rama de las matemáticas
- 3 Programación en términos de conceptos

# 1. Programación como un oficio

## Motivación

### Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

### Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

### Otros paradigmas

Basado en interacción

### Bibliografía

- Se estudia en un paradigma único y con un único lenguaje
- Puede ser contraproducente. Por ejemplo, aprender a manipular listas en ciertos lenguajes puede llevar a la conclusión errónea de que el manejo de listas es siempre tan complicado y costoso:

# 1. Programación como un oficio

## ZIP lists en Java

```
class Pair<A, B> {
    private A left;
    private B right;

    public Pair(A left, B right) {
        this.left = left;
        this.right = right;
    }

    public A left() { return left; }
    public B right() { return right; }
    public String toString() {
        return "(" + left + ", " +
            right + ")";
    }
}

public class MyZip {
    public static <A, B> List<Pair<A, B>> zip(List<A> as, List<B> bs) {
        Iterator<A> it1 = as.iterator();
        Iterator<B> it2 = bs.iterator();
        List<Pair<A, B>> result = new ArrayList<>();
        while (it1.hasNext() && it2.hasNext()) {
            result.add(new Pair<A, B>(it1.next(), it2.next()));
        }
        return result;
    }

    public static void main(String[] args) {
        List<Integer> x = Arrays.asList(1, 2, 3);
        List<String> y = Arrays.asList("a", "b", "c");
        List<Pair<Integer, String>> zipped = zip(x, y);
        System.out.println(zipped);
    }
}
```

## Salida

```
[(1,a), (2,b), (3,c)]
```

## ZIP lists en Haskell

```
zip :: [a] -> [b] -> [(a,b)]
```

```
zip [] xs           = []
zip (x:xs) []       = []
zip (x:xs) (y:ys) = (x,y):zip xs ys
```

## Uso

```
: zip [1,2,3] ["a","b","c"]
[(1,"a"), (2,"b"), (3,"c")]
```

## Motivación

### Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

### Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

### Otros paradigmas

Basado en interacción

### Bibliografía

## 2. Programación como una rama de las matemáticas

- O bien se estudia en un lenguaje *ideal*, restringido (Dijkstra) o el resultado es demasiado teórico, alejado de la práctica.



## 2. Programación como una rama de las matemáticas

Ejemplo: verificación formal (de un programa de una línea)

### El programa

```
while (x<10) x:=x+1;
```

### La prueba

Partimos de la expresión (*Hoare triple*)

```
{ $x \leq 10$ } while (x<10) x:=x+1 { $x=10$ }
```

La condición del bucle es  $x < 10$ . Usamos el invariante de bucle  $x \leq 10$  y con estas asunciones podemos probar la expresión

```
{ $x < 10 \wedge x \leq 10$ } x:=x+1 { $x \leq 10$ }
```

Esta expresión se deriva formalmente de las reglas de la lógica de Floyd-Hoare, pero también puede justificarse de forma intuitiva: *La computación comienza en un estado donde se cumple  $x < 10 \wedge x \leq 10$ , lo que es equivalente a decir que  $x < 10$ . La computación añade 1 a x, por lo que tenemos que  $x \leq 10$  es cierto (en el dominio de los enteros)*

Bajo esta premisa, la regla para el bucle while nos permite sacar la conclusión

```
{ $x \leq 10$ } while (x<10) x:=x+1 { $\neg(x < 10) \wedge x \leq 10$ }
```

Y podemos ver que la postcondición de esta expresión es lógicamente equivalente a  $x=10$ .

### 3. Programación en términos de conceptos

- Se estudia un conjunto de **conceptos semánticos** y **estructuras de implementación** en términos de los cuales se describen de forma natural diferentes lenguajes y sus implementaciones

### 3. Programación en términos de conceptos

Un lenguaje de programación puede combinar características de distintos bloques

#### Lenguaje funcional

- (+) Polimorfismo
- (+) Estrategias
- (+) Orden superior

#### Lenguaje lógico

- (+) No determinismo
- (+) Variables lógicas
- (+) Unificación

#### Lenguaje *kernel*

- (+) Abstracción de datos
- (+) Recursión
- (+) ...

#### Lenguaje imperativo

- (+) Estados explícitos
- (+) Modularidad
- (+) Componentes

#### Lenguaje OO

- (+) Clases
- (+) Herencia

#### Lenguaje *dataflow*

- (+) Concurrencia

# Conceptos esenciales

## Motivación

## Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

## Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

## Otros paradigmas

Basado en interacción

## Bibliografía

Destacamos los siguientes conceptos:

- Tipos y sistemas de tipos
- Polimorfismo
- Reflexión
- Paso de parámetros
- Ámbito de las variables
- Gestión de memoria

# Tipos y sistemas de tipos

## Motivación

## Conceptos

### Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

## Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

## Otros paradigmas

Basado en interacción

## Bibliografía

Un **tipo** representa el conjunto de valores que puede adoptar una variable o expresión. Los tipos:

- Ayudan a detectar **errores** de programación  
*Solo los programas que utilizan las expresiones según el tipo que tienen aplicando las funciones permitidas son **legales***
- Ayudan a **estructurar** la información  
*Los tipos pueden verse como colecciones de valores que comparten ciertas propiedades*
- Ayudan a manejar estructuras de **datos**  
*Los tipos indican cómo utilizar las estructuras de datos que comparten el mismo tipo mediante ciertas operaciones*

# Tipos y sistemas de tipos

Lenguajes tipificados

## Motivación

## Conceptos

## Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

## Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

## Otros paradigmas

Basado en interacción

## Bibliografía

- En los lenguajes **tipificados**, las variables tienen un tipo asociado (e.g., C, C++, C#, Haskell, Java, Maude).
- Los lenguajes que no restringen el rango de valores que pueden adoptar las variables son **no tipificados** (e.g., Lisp, Prolog).
  - También puede entenderse que todos los valores tienen un tipo único o universal

# Tipos y sistemas de tipos

Lenguajes tipificados

## Motivación

## Conceptos

## Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

## Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

## Otros paradigmas

Basado en interacción

## Bibliografía

El sistema de tipos establece qué tipo de asociación de variables es posible:

- El **valor** asociado a la variable debe tener el tipo de ésta (e.g., C, Haskell)
- El **valor** asociado a la variable puede ser de otros tipos *compatibles* relacionados con el tipo de la variable (e.g., C++, C#, Java).
- De forma ortogonal, además, el tipo del valor asociado a una variable puede cambiar:
  - **Tipado Estático**: el tipo del valor no cambia durante la ejecución
  - **Tipado Dinámico**: el tipo del valor puede cambiar durante la ejecución

# Tipos y sistemas de tipos

Lenguajes tipificados

## Motivación

## Conceptos

## Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

## Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

## Otros paradigmas

Basado en interacción

## Bibliografía

$$\text{Lenguajes tipificados} = \text{Expresiones de programa} + \text{Sistemas de tipos}$$

- En los lenguajes con tipificación **explícita**, los tipos forman parte de la sintaxis.
- En los lenguajes con tipificación **implícita**, los tipos **no** forman parte de la sintaxis.



# Tipos y sistemas de tipos

## Ejemplos de tipificación

### Motivación

### Conceptos

#### Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

### Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

### Otros paradigmas

Basado en interacción

### Bibliografía

- Lenguaje **no tipificado**: Prolog

```
objeto(llave) .
objeto(pelota) .
cosa(X) <- objeto(X) .
```

La variable  $x$  no tiene tipo asociado.

- Lenguaje con **tipificación explícita**: Java

```
int x;
x = 42;
```

Todas las variables deben ser declaradas, y en la declaración debe especificarse su tipo explícitamente

- Lenguaje con **tipificación implícita**: Haskell

```
fac 0 = 1
fac x = x * fac (x-1)
```

El sistema de tipos infiere automáticamente el tipo de la variable  $x$

# Tipos y sistemas de tipos

## Motivación

## Conceptos

### Tipos y sistemas de tipos

Polimorfismo  
Sobrecarga  
Coerción  
Genericidad  
Inclusión  
Reflexión  
Procedimientos y control de flujo  
Paso de parámetros  
Alcance de las variables  
Gestión de memoria

## Paradigmas de programación

Imperativo  
Declarativo  
OO  
Concurrente

## Otros paradigmas

Basado en interacción

## Bibliografía

Para definir el tipo de las variables o expresiones usamos un *lenguaje de expresiones de tipo*.

## Ejemplo de lenguaje de expresiones de tipo

- Tipos básicos o primitivos: `Bool`, `Char`, `Int`, ...
- Variables de tipo: `a`, `b`, `c`, ...
- Constructores de tipo:
  - $\rightarrow$  para definir funciones,
  - $\times$  para definir pares,
  - `[]` para definir listas
  - ...
- Reglas de construcción de las expresiones:

$$\tau ::= \text{Bool} \mid \text{Char} \mid \text{Int} \mid \dots \mid \tau \rightarrow \tau \mid \tau \times \tau \mid [\tau]$$

# Tipos y sistemas de tipos

## Tipos monomórficos y tipos polimórficos

- Los tipos en cuya expresión de tipo **no** aparece ninguna variable de tipo se denominan **monotipos** o **tipos monomórficos**.
- Los tipos en cuya expresión de tipo aparece alguna variable de tipo se denominan **politipos** o **tipos polimórficos**.
- Un tipo polimórfico representa un conjunto infinito de monotipos

# Tipos y sistemas de tipos

## Ejemplo de expresiones de tipos

### Motivación

### Conceptos

#### Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

### Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

### Otros paradigmas

Basado en interacción

### Bibliografía

- Expresión de tipo predefinido. Tipos básicos `Bool`, `Int`, ...

`Bool` es el tipo de los valores booleanos `True` y `False`

- Expresión de tipo funcional.

`Int`  $\rightarrow$  `Int` es el tipo de la función `fact` vista antes, que devuelve el factorial de un número.

- Expresión de tipo parametrizado.

`[a]`  $\rightarrow$  `Int` es el tipo de la función `length`, que calcula la longitud de una lista.

# Tipos y sistemas de tipos

## Ejemplo de expresiones de tipos

### Motivación

### Conceptos

#### Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

### Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

### Otros paradigmas

Basado en interacción

### Bibliografía

- Expresión de tipo predefinido. Tipos básicos `Bool`, `Int`, ...

`Bool` es el tipo de los valores booleanos `True` y `False`

tipos monomórficos

- Expresión de tipo funcional.

`Int → Int` es el tipo de la función `fact` vista antes, que devuelve el factorial de un número.

tipo polimórfico

- Expresión de tipo parametrizado.

`[a] → Int` es el tipo de la función `length`, que calcula la longitud de una lista.

variable de tipo

constructor de tipo

## Motivación

## Conceptos

Tipos y sistemas de tipos

**Polimorfismo**

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

## Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

## Otros paradigmas

Basado en interacción

## Bibliografía

- Es una característica de los lenguajes que permite manejar valores de distintos tipos usando una interfaz uniforme
- Se aplica tanto a funciones como a tipos:
  - Una función puede ser polimórfica con respecto a uno o varios de sus argumentos.

*La suma (+) puede aplicarse a valores de diferentes tipos como enteros, reales, ...*

- Un tipo de datos puede ser polimórfico con respecto a los tipos de los elementos que contiene.

*Una lista con elementos de un tipo arbitrario es un tipo polimórfico*

## Motivación

## Conceptos

Tipos y sistemas de tipos

## Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

## Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

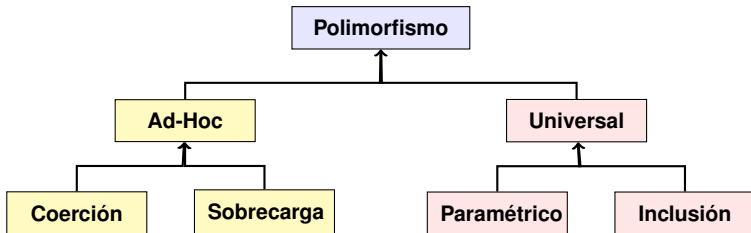
## Otros paradigmas

Basado en interacción

## Bibliografía

## Polimorfismo

## Tipos de polimorfismo



- Ad-hoc o aparente: trabaja sobre un número finito de tipos no relacionados
  - Sobrecarga
  - Coerción
- Universal o verdadero: trabaja sobre un número potencialmente infinito de tipos *con cierta estructura común*
  - paramétrico (genericidad)
  - de inclusión (herencia)

- **Sobrecarga:** existencia de distintas funciones con el mismo nombre.

- Los operadores aritméticos  $+$ ,  $-$ ,  $*$ ,  $/$ ,  $\dots$  suelen estar sobrecargados:

$$(+)\quad :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$$

```
(+) :: Float -> Float -> Float
```

```
(+) :: Complex -> Complex -> Complex
```

```
(+) :: Int -> Float -> Float
```

corresponden a distintos usos de +

- El operador `+` no puede recibir el polítipo

$$(+) :: a \rightarrow a \rightarrow a$$

porque significaría dotar de significado (e implementación) a la *suma* de caracteres, funciones, listas, etc., lo cual puede interesarnos o no



# Polimorfismo. Sobrecarga

## Ejemplo de sobrecarga en Java (1)

En Java, la sobrecarga de métodos se realiza cambiando el tipo de los parámetros:

```
/* overloaded methods */  
  
int myAdd(int x, int y, int z) {  
    ...  
}  
  
double myAdd(double x, double y, double z) {  
    ...  
}
```

# Polimorfismo. Sobrecarga

## Ejemplo de sobrecarga en Java (2)

```
public class Overload {  
    public void numbers(int x, int y) {  
        System.out.println("Method that gets integer numbers");  
    }  
    public void numbers(double x, double y, double z) {  
        System.out.println("Method that gets real numbers");  
    }  
    public int numbers(String st) {  
        System.out.println("The length of " + st + " is " +  
            st.length());  
        return st.length();  
    }  
    public static void main(...) {  
        Overload s = new Overload();  
        int a = 1;  
        int b = 2;  
        s.numbers(a,b);  
        s.numbers(3.2,5.7,0.0);  
        a = s.numbers("Madagascar");  
    }  
}
```

No tiene por qué haber coincidencia en cuanto al número ni en cuanto al tipo de los parámetros/resultado

## Motivación

## Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

**Coerción**

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

## Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

## Otros paradigmas

Basado en interacción

## Bibliografía

# Polimorfismo. Coerción

## Polimorfismo Ad-Hoc: Coerción

- **Coerción:** conversión (implícita o explícita) de valores de un tipo a otro.
- Cuando es implícita suele hacerse usando una jerarquía de tipos o de su representación.

*Por ejemplo, en la mayoría de lenguajes, para los argumentos de los operadores aritméticos existe coerción entre valores enteros y reales*

- Algunos lenguajes permiten forzar una coerción explícita.
  - Lenguajes de la familia de C (sentencia *Cast*)
  - En Java es posible transformar:
    - una variable primitiva de un tipo básico a otro
    - un objeto de una clase a una superclase

# Polimorfismo. Coerción

## Ejemplo de coerción en Java

### Motivación

### Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

**Coerción**

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

### Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

### Otros paradigmas

Basado en interacción

### Bibliografía

### Conversión implícita en Java:

```
int num1 = 100        // 4 bytes
long num2 = num1       // 8 bytes
```

### Conversión explícita en Java:

```
int num1 = 100                // 4 bytes
short num2 = (short) num1     // 2 bytes
char c = (char) num1          // 2 bytes

String s = Integer.toString(num1)
```

# Polimorfismo. Genericidad

## Polimorfismo Universal: Genericidad

### Motivación

### Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

**Genericidad**

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

### Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

### Otros paradigmas

Basado en interacción

### Bibliografía

- **Genericidad/Paramétrico:** la definición de una función o la declaración de una clase presenta una estructura que es común a un número potencialmente infinito de tipos

- En Haskell podemos definir y usar tipos genéricos y funciones genéricas
- En Java podemos definir y usar clases genéricas y métodos genéricos

# Polimorfismo. Genericidad

## Ejemplo de genericidad en Haskell

### Motivación

### Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

**Genericidad**

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

### Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

### Otros paradigmas

Basado en interacción

### Bibliografía

Usando **un tipo genérico** (con variables de tipo), podemos definir una estructura de datos para representar y manipular las entradas (de cualquier tipo) de un *diccionario*:

```
type Entry k v = (k,v)
```

```
getKey :: Entry k v -> k
```

```
getKey (x,y) = x
```

```
getValue :: Entry k v -> v
```

```
getValue (x,y) = y
```

Con una **función genérica** podemos calcular la longitud de una lista cuyos elementos son de cualquier tipo:

```
length :: [a] -> Integer
```

```
length [] = 0
```

```
length (x:xs) = 1 + (length xs)
```

# Polimorfismo. Genericidad

## Ejemplo de genericidad en Java (1/2)

### Motivación

### Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

**Genericidad**

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

### Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

### Otros paradigmas

Basado en interacción

### Bibliografía

Podemos usar una clase genérica (con parámetros) para definir una entrada de un *diccionario*:

Deben empezar con mayúscula

```
public class Entry<K,V>{
    private final K mKey;
    private final V mValue;

    public Entry(K k, V v){
        mKey = k;
        mValue = v;
    }

    public K getKey() {
        return mKey;
    }

    public V getValue() {
        return mValue;
    }
}
```

Podemos definir un **método genérico** para calcular la longitud de un array de *cualquier* tipo:

```
public static <T> int lengthA(T[] inputArray){
    ...
}
```

# Polimorfismo. Genericidad

## Ejemplo de genericidad en Java (2/2)

### Motivación

### Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

**Genericidad**

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

### Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

### Otros paradigmas

Basado en interacción

### Bibliografía

Ejemplo de uso de entradas de un diccionario:

parametrización

```
Entry<Integer,String> elem1 = new Entry<>(3,"Programming");  
System.out.println(elem1.getValue());
```

Ejemplo de uso del método genérico para la longitud de un array:

```
Integer[] intArray = {1, 2, 3, 5};  
Double[] doubleArray = {1.1, 2.2, 3.3};  
  
System.out.println("Array length =" + lengthA(intArray));  
System.out.println("Array length =" + lengthA(doubleArray));
```



# Polimorfismo. Genericidad

## Algunas consideraciones de la genericidad en Java

### Motivación

### Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

**Genericidad**

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

### Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

### Otros paradigmas

Basado en interacción

### Bibliografía

- Una clase genérica es una clase convencional, salvo que dentro de su declaración utiliza una **variable de tipo** (parámetro), que será definido cuando sea utilizado.
- Dentro de una clase genérica se pueden utilizar otras clases genéricas
- Una clase genérica puede tener varios parámetros

## Polimorfismo Universal: Inclusión/Herencia

## Conceptos

## Gestión de memoria

### Concurrente

Basado en  
interacción

## Bibliografía

- **Inclusión o Herencia:** la definición de una función trabaja sobre tipos que están relacionados siguiendo una jerarquía de inclusión.
- En la orientación a objetos la herencia es el mecanismo más utilizado para permitir la **reutilización y extensibilidad**.

La herencia organiza las clases en una estructura jerárquica formando **jerarquías de clases**

# Polimorfismo. Inclusión

## Polimorfismo Universal: Inclusión/Herencia

### IDEA:

Una clase B heredará de una clase A cuando queremos que B tenga la estructura y comportamientos de la clase A. Además podremos

- añadir nuevos atributos a B
- añadir nuevos métodos a B

Y dependiendo del lenguaje podremos

- redefinir métodos heredados
- heredar de varias clases (en Java solo podemos heredar de una clase)

## Motivación

## Conceptos

Tipos y sistemas  
de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

**Inclusión**

Reflexión

Procedimientos y  
control de flujoPaso de  
parámetrosAlcance de las  
variablesGestión de  
memoria**Paradigmas  
de programación**

Imperativo

Declarativo

OO

Concurrente

**Otros  
paradigmas**Basado en  
interacción**Bibliografía**

# Polimorfismo. Inclusión

## Ejemplo de herencia en Java (1/2)

```
public class Bicycle {
    protected int cadence;
    protected int gear;
    protected int speed;
    public Bicycle (int startCad, int startSpeed,
                    int startGear) {
        cadence = startCad;
        speed = startSpeed;
        gear = startGear;
    }

    public void setCadence(int newValue) {
        cadence = newValue; }
    public void setGear(int newValue) {
        gear = newValue; }
    public void applyBrake(int decrement) {
        speed -= decrement; }
    public void speedUp(int increment) {
        speed += increment; }
}
```

## Motivación

## Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

**Inclusión**

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

## Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

## Otros paradigmas

Basado en interacción

## Bibliografía

# Polimorfismo. Inclusión

## Ejemplo de herencia en Java (2/2)

```
public class MountainBike extends Bicycle {  
    public int seatHeight;  
    public MountainBike(int startHeight, int startCad,  
                        int startSpeed, int startGear){  
        super(startCad, startSpeed, startGear);  
        seatHeight = startHeight;  
    }  
}
```

- Las subclases se definen usando la palabra clave **extends**
- Se pueden añadir atributos, métodos y redefinir métodos

## Motivación

## Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

**Inclusión**

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

## Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

## Otros paradigmas

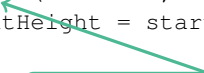
Basado en interacción

## Bibliografía

# Polimorfismo. Inclusión

## Ejemplo de herencia en Java (2/2)

```
public class MountainBike extends Bicycle {  
    public int seatHeight;  
    public MountainBike(int startHeight, int startCad,  
                        int startSpeed, int starteGear){  
        super(startCad, startSpeed, StartGear);  
        seatHeight = startHeight;  
    }  
}
```



constructor de la clase padre

- Las subclases se definen usando la palabra clave **extends**
- Se pueden añadir atributos, métodos y redefinir métodos

# Polimorfismo. Inclusión

Algunas consideraciones de la herencia en Java (1/3)

## Motivación

## Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

**Inclusión**

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

## Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

## Otros paradigmas

Basado en interacción

## Bibliografía

- En Java, la base de cualquier jerarquía es la clase `Object`.
- Si una clase se declara como `final`, no se puede heredar de ella
- Java solo tiene herencia simple
- A una variable de la superclase se le puede asignar una referencia a cualquier subclase derivada de dicha superclase, pero **no** al contrario.

## Ejemplo de asignación válida

```
Bicycle b;  
MountainBike m = new MountainBike(75, 90, 25, 8);  
b = m
```

# Polimorfismo. Inclusión

## Algunas consideraciones de la herencia en Java (2/3)

### Motivación

### Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

**Inclusión**

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

### Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

### Otros paradigmas

Basado en interacción

### Bibliografía

- En Java se usan calificadores delante de los atributos y métodos para establecer qué variables de instancia y métodos de los objetos de una clase son visibles
  - Private:** ningún miembro o atributo `private` de la superclase es visible en las subclases u otras clases.

*Si se usa para atributos de clase, deberán definirse métodos que accedan a dichos atributos*

- Protected:** los miembros `protected` de la superclase son visibles en todas las subclases y en el propio paquete pero no visibles desde otros paquetes.
- Public:** los miembros `public` son visibles desde cualquier otra clase.
- Default:** los miembros con visibilidad `default` son visibles desde cualquier clase que esté en el mismo paquete.



## Motivación

## Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

**Inclusión**

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

## Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

## Otros paradigmas

Basado en interacción

## Bibliografía

## Polimorfismo. Inclusión

Algunas consideraciones de la herencia en Java (3/3)

	Clase	Paquete	Subclase	Otros
Public	Sí	Sí	Sí	Sí
Private	Sí	No	No	No
Protected	Sí	Sí	Sí	No
Default	Sí	Sí	No	No

Cuadro: Visibilidad en Java

# Polimorfismo. Inclusión

## Ejemplo de redefinición de método heredado en Java (1/2)

### Motivación

### Conceptos

Tipos y sistemas  
de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

**Inclusión**

Reflexión

Procedimientos y  
control de flujo

Paso de  
parámetros

Alcance de las  
variables

Gestión de  
memoria

### Paradigmas de programación

Imperativo

Declarativo  
OO

Concurrente

### Otros paradigmas

Basado en  
interacción

### Bibliografía

```
public class Employee {  
    String name;  
    int nEmployee, salary;  
    static private int counter = 0;  
    public Employee (String name, int salary){  
        this.name = name;  
        this.salary = salary;  
        nEmployee = ++counter;  
    }  
    public void increaseSalary(int wageRaise){  
        salary += (int) (salary*wageRaise/100);  
    }  
    public String toString(){  
        return "Num. Employee " + nEmployee +  
            " Name: " + name + " Salary: " + salary;  
    }  
}
```

# Polimorfismo. Inclusión

## Ejemplo de redefinición de método heredado en Java (2/2)

### Motivación

### Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

**Inclusión**

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

### Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

### Otros paradigmas

Basado en interacción

### Bibliografía

```
public class Executive extends Employee{
    int budget;
    public Executive(String name, int salary){
        super(name, salary);
    }
    void assignBudget(int b){
        budget = b;
    }
    public String toString(){
        String s = super.toString();
        s = s + " Budget: " + budget;
        return s;
    }
}
```

### Ejemplo de uso:

```
Executive boss = new Executive("Thomas Turner", 1000);
boss.assignBudget(1500);
boss.increaseSalary(5);
```

## Motivación

## Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

**Inclusión**

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

## Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

## Otros paradigmas

Basado en interacción

## Bibliografía

# Polimorfismo. Inclusión

## Herencia en Java: Clases abstractas

- Una clase abstracta es la que se declara como `abstract`
  - Si una clase tiene un método `abstract` es obligatorio que la clase sea `abstract`.
  - Para los métodos declarados `abstract` no se da implementación.
  - Una clase abstracta no puede tener instancias.
- Todas las subclases que hereden de una clase abstracta, si ellas no son abstractas tendrán que redefinir los métodos abstractos dándoles una implementación.

# Polimorfismo. Inclusión

## Ejemplo de uso de clases abstractas en Java (1/2)

### Motivación

### Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

**Inclusión**

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

### Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

### Otros paradigmas

Basado en interacción

### Bibliografía

```
public abstract class Shape {  
    private float x, y; // Position of the shape  
    public Shape (float initX, float initY){  
        x = initX; y = initY;  
    }  
    public void move(float incX, float incY){  
        x = x+incX; y = y+incY;  
    }  
    public float getX(){ return x; }  
    public float getY(){ return y; }  
    public abstract float perimeter();  
    public abstract float area();  
}
```

## Motivación

## Conceptos

Tipos y sistemas  
de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

**Inclusión**

Reflexión

Procedimientos y  
control de flujoPaso de  
parámetrosAlcance de las  
variablesGestión de  
memoriaParadigmas  
de programación

Imperativo

Declarativo

OO

Concurrente

Otros  
paradigmasBasado en  
interacción

## Bibliografía

## Polimorfismo. Inclusión

## Ejemplo de uso de clases abstractas en Java (2/2)

```

public class Square extends Shape {
    private float side;
    public Square (float initX, float initY, float initSide){
        super(initX,initY); // Call to super constructor
        side = initSide;
    }
    public float perimeter(){ return 4*side; }
    public float area(){ return side*side; }
}

public class Circle extends Shape {
    private float radius;
    public Circle(float initX, float initY, float initRadius){
        super(initX,initY); // Call to super constructor
        radius = initRadius;
    }
    public float perimeter(){ return 2*pi*radius; }
    public float area(){ return pi*radius*radius; }
}

```

## Motivación

## Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

**Inclusión**

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

## Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

## Otros paradigmas

Basado en interacción

## Bibliografía

## Polimorfismo. Inclusión

## Ejemplo de uso de clases abstractas en Java (2/2)

```

public class Square extends Shape {
    private float side;
    public Square (float initX, float initY, float initSide){
        super(initX,initY); // Call to super constructor
        side = initSide;
    }
    public float perimeter(){ return 4*side; }
    public float area(){ return side*side; }
}

public class Circle extends Shape {
    private float radius;
    public Circle(float initX, float initY, float initRadius){
        super(initX,initY);
        radius = initRadius;
    }
    public float perimeter(){ return 2*pi*radius; }
    public float area(){ return pi*radius*radius; }
}

```

Y si quisiéramos extender este ejemplo con una forma nueva como el triángulo, ¿qué hay que hacer?

## Motivación

## Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

**Inclusión**

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

## Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

## Otros paradigmas

Basado en interacción

## Bibliografía

## Polimorfismo. Inclusión

## Herencia en Java: Interfaces

Una interfaz **declara** los atributos y operaciones que deben definirse en las clases que implementan (**implements**) dicha interfaz

```
public interface MyInterface {
    public int method1(...);
    ...}

public class MyClass implements MyInterface {
    public int method1(...) {...}
    ...}
```

- Los métodos de una interfaz pueden ser abstractos, estáticos y default (incluyen una implementación por defecto). Su visibilidad puede ser “public”, “private” y “default”, pero no “protected”.
- Los atributos son estáticos y finales (constantes)
- Una clase puede implementar varias interfaces
- Las interfaces pueden heredar de otras interfaces (**extends**)



## Motivación

## Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

**Inclusión**

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

## Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

## Otros paradigmas

Basado en interacción

## Bibliografía

# Cuestión: Inclusión y genericidad

Dadas la siguiente definición de clases:

```
class Shape { /*...*/ }  
class Circle extends Shape { /*...*/ }  
class Rectangle extends Shape { /*...*/ }  
class Node<T> { /*...*/ }
```

¿Compila sin error el siguiente fragmento de código? ¿Por qué?

```
Node<Circle> nc = new Node<Circle>();  
Node<Shape> ns = nc;
```

Motivación

Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

**Inclusión**

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

Otros paradigmas

Basado en interacción

Bibliografía

# Cuestión: Inclusión y genericidad

Responde a las siguientes cuestiones:

- ¿Puede una interfaz heredar de una clase?
- ¿Se pueden crear instancias de las clases interfaz?

## Motivación

## Conceptos

Tipos y sistemas  
de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

## Reflexión

Procedimientos y  
control de flujoPaso de  
parámetrosAlcance de las  
variablesGestión de  
memoriaParadigmas  
de programación

Imperativo

Declarativo

OO

Concurrente

Otros  
paradigmasBasado en  
interacción

## Bibliografía

## Qué es la reflexión

Cuando te miras en un espejo puedes:

- ver tu reflejo y
  - reaccionar ante lo que ves
- 
- En los **lenguajes de programación**, la reflexión es **la infraestructura** que, durante su ejecución, permite a un programa:
    - ver su propia estructura y
    - manipularse a sí mismo
  - La reflexión se introdujo con el lenguaje LISP y está presente también en algunos lenguajes de script.

Permite, por ejemplo, definir programas capaces de monitorizar su propia ejecución y modificarse, en tiempo de ejecución, para adaptarse dinámicamente a distintas situaciones

## Motivación

## Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

## Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

## Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

## Otros paradigmas

Basado en interacción

## Bibliografía

## Reflexión

## Lenguajes con reflexión

- Se caracterizan porque las propias instrucciones del lenguaje son tratadas como valores de un tipo de datos específico; En los lenguajes sin reflexión se ven como simples cadenas de caracteres
- Los lenguajes con reflexión pueden verse como **metalenguajes** del propio lenguaje.

*Se llama metalenguaje a aquél con el que podemos escribir metaprogramas (programas que manipulan programas como compiladores, analizadores, etc.)*

# Reflexión

Debe usarse con cautela

- Mal usada puede afectar
  - al rendimiento del sistema ya que suele ser costosa

*Si puede hacerse sin usar reflexión, no la uses*

- a la seguridad ya que puede exponer información comprometida sobre el código

*La reflexión rompe la abstracción, con reflexión puede accederse a atributos y métodos privados, etc.*

- Es una característica avanzada pero no complicada, especialmente en lenguajes funcionales, gracias a la dualidad natural entre datos y programas (homoiconicidad).

Tipos y sistemas  
de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

**Reflexión**Procedimientos y  
control de flujoPaso de  
parámetrosAlcance de las  
variablesGestión de  
memoria

Imperativo

Declarativo

OO

Concurrente

Basado en  
interacción

# Reflexión

## La reflexión en Java

- En Java la reflexión se usa mediante la biblioteca `java.lang.reflect`

La biblioteca proporciona clases para representar de forma estructurada información de las clases, variables, métodos, etc.

- Se puede inspeccionar clases, interfaces, atributos y métodos sin conocer los nombres de los mismos en tiempo de compilación. Por ejemplo podemos
  - leer de teclado un `String` con el que poder crear un objeto con ese nombre, o invocar un método con ese nombre,
  - leer todos los métodos consultores (get) o modificadores (set) de una clase,
  - acceder a atributos y métodos privados de una clase.

## Motivación

## Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

## Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

## Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

## Otros paradigmas

Basado en interacción

## Bibliografía

```
import java.lang.reflect.*;

public class MyClass {...}

...
MyClass myClassObj = new MyClass();
// get the class information:
Class<? extends MyClass> objMyClassInfo =
    myClassObj.getClass();

// get the fields:
Field[] allDeclaredVars = objMyClassInfo.getDeclaredFields();
// travel the fields:
for (Field variable : allDeclaredVars) {
    System.out.println("Name of GLOBAL VARIABLE: " +
        variable.getName());
}
```

## Otros métodos definidos en la clase Class:

```
Constructor[] getConstructors();
Field[]        getDeclaredFields();
Method[]       getDeclaredMethods();

...
```

# Procedimientos y control de flujo

## Motivación

## Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

**Procedimientos y control de flujo**

Paso de parámetros

Alcance de las variables

Gestión de memoria

## Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

## Otros paradigmas

Basado en interacción

## Bibliografía

Existen algunos conceptos relacionados con el control de flujo de los programas y la definición y llamada a procedimientos.

- **Paso de parámetros.** Cuando se hace una llamada a un método o función hay un cambio de contexto que puede hacerse de distintas formas. Veremos las principales.
- **Ámbito de las variables.** Es necesario determinar si un objeto o variable es visible en un momento determinado de la ejecución y este cálculo puede hacerse de forma estática o bien de forma dinámica.



# Paso de parámetros

## Motivación

## Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

**Paso de parámetros**

Alcance de las variables

Gestión de memoria

## Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

## Otros paradigmas

Basado en interacción

## Bibliografía

Uno de los mecanismos de abstracción básicos es organizar las tareas de un programa definiendo funciones, métodos o procedimientos que resuelven subtarefas. Así, en un momento determinado se puede **invocar** a dichos procedimientos.

- LLAMADA:  $f(e_1, \dots, e_n)$  con  $e_1, \dots, e_n$  expresiones.
  - al ejecutarse la llamada, el flujo de control pasará al cuerpo de la función  $f$  y, una vez éste termine, volverá al flujo desde el que se hizo la llamada.
  - $e_1, \dots, e_n$  son los llamados **parámetros de entrada/reales** (*actual parameters*)
- DECLARACIÓN:  $f(x_1, \dots, x_n)$  con  $x_1, \dots, x_n$  variables.
  - $x_1, \dots, x_n$  son los llamados **parámetros formales** (*formal parameters*)
  - Los parámetros formales son variables locales al cuerpo de la función declarada

# Paso de parámetros

## Motivación

## Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

**Paso de parámetros**

Alcance de las variables

Gestión de memoria

## Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

## Otros paradigmas

Basado en interacción

## Bibliografía

Existen distintos tipos de paso de parámetros

- Paso por valor (*call by value*)
- Paso por referencia (*call by reference/call by address*)
- Paso por necesidad (*call by need*)

Existen más modalidades de paso de parámetros pero éstas son las más frecuentes en los lenguajes de programación

## Motivación

## Conceptos

Tipos y sistemas  
de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y  
control de flujo**Paso de  
parámetros**Alcance de las  
variablesGestión de  
memoriaParadigmas  
de programación

Imperativo

Declarativo

OO

Concurrente

Otros  
paradigmasBasado en  
interacción

## Bibliografía

## Paso de parámetros

## Paso por valor

Se calculan los valores  $v_i$  de los parámetros de entrada  $e_i$  en la llamada y se copian en los parámetros formales  $x_i$

- en el cuerpo de la función se trabaja sobre una referencia a memoria diferente.

```
void inc(int v)
{
    v = v + v;
}

...
int a = 10;
inc(a);
```

# Paso de parámetros

## Paso por valor

### Motivación

### Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

**Paso de parámetros**

Alcance de las variables

Gestión de memoria

### Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

### Otros paradigmas

Basado en interacción

### Bibliografía

Se calculan los valores  $v_i$  de los parámetros de entrada  $e_i$  en la llamada y se copian en los parámetros formales  $x_i$

- en el cuerpo de la función se trabaja sobre una referencia a memoria diferente.

```
void inc(int v)
{
    v = v + v;
}
...
int a = 10;
inc(a);
```

$a = 10$

## Motivación

## Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

**Paso de parámetros**

Alcance de las variables

Gestión de memoria

## Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

## Otros paradigmas

Basado en interacción

## Bibliografía

## Paso de parámetros

## Paso por valor

Se calculan los valores  $v_i$  de los parámetros de entrada  $e_i$  en la llamada y se copian en los parámetros formales  $x_i$

- en el cuerpo de la función se trabaja sobre una referencia a memoria diferente.

```
void inc(int v)
{
    v = v + v;
}
...
```

```
int a = 10;
```

```
inc(a);
```

```
a = 10
```

## En la llamada:

Se copia el valor 10 en el parámetro formal  $v$

# Paso de parámetros

## Paso por valor

### Motivación

### Conceptos

Tipos y sistemas  
de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y  
control de flujo

**Paso de  
parámetros**

Alcance de las  
variables

Gestión de  
memoria

### Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

### Otros paradigmas

Basado en  
interacción

### Bibliografía

Se calculan los valores  $v_i$  de los parámetros de entrada  $e_i$  en la llamada y se copian en los parámetros formales  $x_i$

- en el cuerpo de la función se trabaja sobre una referencia a memoria diferente.

```
void inc(int v)
```

```
{
```

```
    v = v + v;
```

```
}
```

```
...
```

```
int a = 10;
```

```
inc(a);
```

$v = 10$

$a = 10$

# Paso de parámetros

## Paso por valor

### Motivación

### Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

**Paso de parámetros**

Alcance de las variables

Gestión de memoria

### Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

### Otros paradigmas

Basado en interacción

### Bibliografía

Se calculan los valores  $v_i$  de los parámetros de entrada  $e_i$  en la llamada y se copian en los parámetros formales  $x_i$

- en el cuerpo de la función se trabaja sobre una referencia a memoria diferente.

```
void inc(int v)
{
    v = v + v;
}

...
int a = 10;
inc(a);
```

$v = 20$

$a = 10$

# Paso de parámetros

## Paso por valor

### Motivación

### Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

**Paso de parámetros**

Alcance de las variables

Gestión de memoria

### Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

### Otros paradigmas

Basado en interacción

### Bibliografía

Se calculan los valores  $v_i$  de los parámetros de entrada  $e_i$  en la llamada y se copian en los parámetros formales  $x_i$

- en el cuerpo de la función se trabaja sobre una referencia a memoria diferente.

```
void inc(int v)
{
    v = v + v;
}
...
int a = 10;
inc(a);
```

$a = 10$

- La variable  $a$  NO se modifica porque se trabaja con una copia en la función `inc`.



# Paso de parámetros

## Paso por referencia

### Motivación

### Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

**Paso de parámetros**

Alcance de las variables

Gestión de memoria

### Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

### Otros paradigmas

Basado en interacción

### Bibliografía

Se pasa **la referencia** a memoria, por lo que el cuerpo de la función trabaja sobre el mismo objeto en memoria

- Para los parámetros de entrada  $e_i$  que **no** sean una variable, funciona como el paso por valor
- Cuando  $e_i$  es una variable (e.g.,  $y_i$ ), las asignaciones realizadas sobre el parámetro formal  $x_i$  alteran también el valor asociado a  $y_i$

```
void inc(int v)
{
    v = v + v;
}

...
int a = 10;
inc(a);
```

## Motivación

## Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

**Paso de parámetros**

Alcance de las variables

Gestión de memoria

## Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

## Otros paradigmas

Basado en interacción

## Bibliografía

## Paso de parámetros

## Paso por referencia

Se pasa **la referencia** a memoria, por lo que el cuerpo de la función trabaja sobre el mismo objeto en memoria

```
void inc(int v)
{
    v = v + v;
}

...
int a = 10;
inc(a);
```

a = 10

## Motivación

## Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

**Paso de parámetros**

Alcance de las variables

Gestión de memoria

## Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

## Otros paradigmas

Basado en interacción

## Bibliografía

## Paso de parámetros

## Paso por referencia

Se pasa **la referencia** a memoria, por lo que el cuerpo de la función trabaja sobre el mismo objeto en memoria

```
void inc(int v)
{
    v = v + v;
}
```

...

```
int a = 10;
```

```
inc(a);
```

```
a = 10
```

## En la llamada:

El parámetro formal  $v$  recibe la dirección de memoria de  $a$

## Motivación

## Conceptos

Tipos y sistemas  
de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y  
control de flujo**Paso de  
parámetros**Alcance de las  
variablesGestión de  
memoriaParadigmas  
de programación

Imperativo

Declarativo

OO

Concurrente

Otros  
paradigmasBasado en  
interacción

## Bibliografía

## Paso de parámetros

## Paso por referencia

Se pasa **la referencia** a memoria, por lo que el cuerpo de la función trabaja sobre el mismo objeto en memoria

```
void inc(int v)
```

```
{
```

**v = 10**

```
    v = v + v;
```

```
}
```

```
...
```

```
int a = 10;
```

```
inc(a);
```

**a = 10**

## Motivación

## Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

**Paso de parámetros**

Alcance de las variables

Gestión de memoria

## Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

## Otros paradigmas

Basado en interacción

## Bibliografía

## Paso de parámetros

## Paso por referencia

Se pasa **la referencia** a memoria, por lo que el cuerpo de la función trabaja sobre el mismo objeto en memoria

```
void inc(int v)
{
    v = v + v;
}

...
int a = 10;
inc(a);
```

v = 20

a = 20

## Motivación

## Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

**Paso de parámetros**

Alcance de las variables

Gestión de memoria

## Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

## Otros paradigmas

Basado en interacción

## Bibliografía

## Paso de parámetros

## Paso por referencia

Se pasa **la referencia** a memoria, por lo que el cuerpo de la función trabaja sobre el mismo objeto en memoria

```
void inc(int v)
{
    v = v + v;
}

...
int a = 10;
inc(a);
```

**a = 20**

- La variable *a* **SÍ** se modifica porque se trabaja sobre la misma dirección de memoria.

## Motivación

## Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

**Paso de parámetros**

Alcance de las variables

Gestión de memoria

## Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

## Otros paradigmas

Basado en interacción

## Bibliografía

# Paso de parámetros

## Paso por necesidad

- Cuando se pasan expresiones, **no se evalúan** hasta que se usan en el cuerpo de la función
- Mecanismo usado en algunos lenguajes funcionales

## Ejemplo

Dada la siguiente función que devuelve el segundo argumento multiplicado por dos

```
sel2nd x y = 2*y
```

si la invocamos con la expresión `sel2nd (2*3) 5`, con paso por necesidad, no se calcularía el valor de la expresión `2*3` al no utilizarse en el cuerpo de la función.

Motivación

Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

**Paso de parámetros**

Alcance de las variables

Gestión de memoria

Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

Otros paradigmas

Basado en interacción

Bibliografía

# Paso de parámetros

## Algunas consideraciones

- En paso por valor, si se pasa una expresión, ésta se evalúa para copiar el valor resultante (a diferencia del paso por necesidad)
- En paso por referencia, si se pasa una expresión también se evalúa y se pasa el valor resultante.



Motivación

Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

**Alcance de las variables**

Gestión de memoria

Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

Otros paradigmas

Basado en interacción

Bibliografía

# Alcance (ámbito) de las variables (1/2)

- Una variable es un *nombre* que se utiliza para acceder a una posición de memoria
- No todos los nombres (de variables, funciones, constantes, etc.) están accesibles durante toda la ejecución, aunque existan en el programa
- El ámbito o alcance de un nombre es la porción del código donde ese nombre es visible (su valor asociado puede ser consultado/modificado).

# Alcance (ámbito) de las variables (2/2)

## Motivación

## Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

**Alcance de las variables**

Gestión de memoria

## Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

## Otros paradigmas

Basado en interacción

## Bibliografía

- El momento en el que se hace el enlace (la asociación) es lo que se llama tiempo de enlazado.
  - Con **alcance estático**, se define en *tiempo de compilación*
  - Con **alcance dinámico**, se define en *tiempo de ejecución*
- Todos los lenguajes de programación modernos usan el alcance estático.
- Cada lenguaje de programación establece una forma de determinar el alcance de los elementos.
  - En Java por ejemplo se usan los atributos `private`, `public`, `protected` y el sistema de jerarquía de paquetes y clases.

# Alcance de las variables

## Ejemplo de cálculo del ámbito de las variables (1/2)

### Motivación

### Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

### Paradigmas de programación

Imperativo

Declarativo  
OO

Concurrente

### Otros paradigmas

Basado en interacción

### Bibliografía

```

1 program ambitos;
2 type
3   TArray : array [1..3]
4             of integer;
5 var
6   a: TArray;
7 procedure uno;
8 procedure dos;
9   a : TArray;
10 begin { * dos * }
11   a[1] := 1;
12   a[2] := 2;
13   a[3] := 3;
14   cambia(1, 2);
15   writeln(a[1], ' ', a[2], ' ', a[3]);
16 end { * dos * };

```

```

17 procedure cambia(i, j:integer)
18   var aux : integer;
19 begin { * cambia * }
20   aux := a[i];
21   a[i] := a[j];
22   a[j] := aux;
23 end { * cambia * };
24 begin { * uno * }
25   a[1] := 0;
26   a[2] := 0;
27   a[3] := 0;
28 dos;
29 end { * uno * };
30 begin { * ambitos * }
31   uno;
32 end { * ambitos * }

```

# Alcance de las variables

## Ejemplo de cálculo del ámbito de las variables (1/2)

### Motivación

### Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

### Paradigmas de programación

Imperativo

Declarativo  
OO

Concurrente

### Otros paradigmas

Basado en interacción

### Bibliografía

```

1 program ambitos;
2 type
3   TArray : array [1..3]
4             of integer;
5 var
6   a : TArray;
7 procedure uno;
8 procedure dos;
9   a : TArray;
10 begin { * dos * }
11   a[1] := 1;
12   a[2] := 2;
13   a[3] := 3;
14   cambia(1, 2);
15   writeln(a[1], ' ', a[2], ' ',
16             a[3]);
16 end { * dos * };

```

```

17 procedure cambia(i, j: integer)
18 var aux : integer;
19 begin { * cambia * }
20   aux := a[i];
21   a[i] := a[j];
22   a[j] := aux;
23 end { * cambia * };
24 begin { * uno * }
25   a[1] := 0;
26   a[2] := 0;
27   a[3] := 0;
28 dos;
29 end { * uno * };
30 begin { * ambitos * }
31 uno;
32 end { * ambitos * }

```

¿Cuáles son los valores que almacena el array `a` al final de la ejecución? ¿Qué se imprime por pantalla en la sentencia `writeln` del código?

# Alcance de las variables

## Ejemplo de cálculo del ámbito de las variables (2/2)

### Considerando alcance estático...

¿Cuáles son los valores que almacena el array `a` al final de la ejecución?  
 ¿Qué se imprime por pantalla en la sentencia `writeln` del código?

En **tiempo de compilación** el enlace es:

- En el cuerpo de la función `uno` (líneas 25 a 27), la variable `a` está enlazada con la variable global de la línea 6 (`uno` no tiene declaración de variables locales).
- En el cuerpo de la función `dos` (líneas 11 a 15), la variable `a` está enlazada con la variable local `a` definida en la línea 9, ya que las variables locales con el mismo nombre que las globales ocultan a estas últimas.
- En el cuerpo de la función `cambia` (líneas 20 a 22), la variable `a` está enlazada con la variable global de la línea 6 (el procedimiento `cambia` está definido en el ámbito de `uno`, igual que `dos`).

# Alcance de las variables

## Ejemplo de cálculo del ámbito de las variables (2/2)

### Considerando alcance estático. . .

¿Cuáles son los valores que almacena el array `a` al final de la ejecución?  
 ¿Qué se imprime por pantalla en la sentencia `writeln` del código?

Por lo tanto:

- En el cuerpo principal del programa (línea 31) se hace una llamada al procedimiento `uno` .
- Los valores del array global se inicializan a los valores 0, 0 y 0 (líneas 25 a 27)
- Se llama a `dos`, que inicializa un array local con valores 1, 2 y 3 lo que no modifica el array global (líneas 11 a 13)
- La llamada a `cambia` cambia los valores del array global, quedando 0, 0 y 0, lo que no modifica el array local de `dos`.
- Se imprime por pantalla los valores del array local (1, 2 y 3)

# Gestión de Memoria

## Motivación

## Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

**Gestión de memoria**

## Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

## Otros paradigmas

Basado en interacción

## Bibliografía

Se refiere a los distintos métodos y operaciones que se encargan de obtener la **máxima utilidad de la memoria**, organizando los procesos y programas que se ejecutan en el sistema operativo de manera tal que se optimice el espacio disponible.

## Influye en las decisiones de diseño de un lenguaje

A veces los lenguajes contienen características o restricciones que solo pueden explicarse por el deseo de los diseñadores de usar una técnica u otra de gestión de memoria

## Motivación

## Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

## Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

## Otros paradigmas

Basado en interacción

## Bibliografía

## Gestión de Memoria

## Necesidad de gestionar la memoria

Elementos con requerimientos de almacenamiento durante la ejecución de los programas:

- Código del **programa traducido**
- **Información temporal** durante la evaluación de expresiones y en el paso de parámetros (e.g., en las llamadas a funciones los valores actuales que evaluarse y almacenarse hasta completar la lista de parámetros)
- **Llamadas** a subprogramas y operaciones de **retorno**
- **Buffers** para las operaciones de entrada y salida
- Operaciones de **inserción y destrucción de estructuras de datos** en la ejecución del programa (e.g., `new` en Java or `dispose` en Pascal)
- Operaciones de **inserción y borrado de componentes** en estructuras de datos (e.g., la función *push* de Perl para añadir un elemento a un array)



# Gestión de Memoria

## Tipos de gestión de memoria

### Tipos de asignación del almacenamiento

#### Estático

Se calcula y asigna en tiempo de compilación

- Eficiente pero incompatible con recursión o estructuras de datos dinámicas

#### Dinámico

Se calcula y asigna en tiempo de ejecución

- en pila
- en un *heap*

## Motivación

## Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

## Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

## Otros paradigmas

Basado en interacción

## Bibliografía

## Gestión de Memoria

## Almacenamiento estático

Almacenamiento calculado en tiempo de compilación que permanece fijo durante la ejecución del programa.

Se suele usar con:

- **variables globales**
- **programa compilado** (instrucciones en lenguaje máquina)
- **variables locales** a un subprograma cuyo **valor no cambia** en las diferentes llamadas
- **constantes** numéricas y cadenas de caracteres
- **tablas** producidas por los compiladores y usadas para operaciones de ayuda en tiempo de ejecución (e.g., comprobación dinámica de tipos, depuración, ...).

Motivación

Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

Otros paradigmas

Basado en interacción

Bibliografía

# Gestión de Memoria

## Almacenamiento estático

Almacenamiento calculado en tiempo de compilación que permanece fijo durante la ejecución del programa.

Se suele usar con:

- **variables globales**
- **programa compilado** (instrucciones en lenguaje máquina)
- **variables locales** a un subprograma cuyo **valor no cambia** en las diferentes llamadas
- **constantes** numéricas y cadenas de caracteres
- **tablas** producidas por los compiladores y usadas para operaciones de ayuda en tiempo de ejecución (e.g., comprobación dinámica de tipos, depuración, ...).

Es eficiente pero incompatible con recursión y con estructuras de datos cuyo tamaño depende de datos de entrada o datos computados durante la ejecución del programa

## Motivación

## Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

**Gestión de memoria****Paradigmas de programación**

Imperativo

Declarativo

OO

Concurrente

**Otros paradigmas**

Basado en interacción

## Bibliografía

# Gestión de Memoria

## Almacenamiento dinámico: en pila

Es la técnica más simple **para manejar los registros de activación en las llamadas a funciones/procedimientos** durante la ejecución del programa (basta un puntero a la cima de la pila)

## Almacenamiento en pila

- al inicio de la ejecución se asigna un bloque secuencial en memoria como espacio de almacenamiento libre,
- cuando se requiere espacio de almacenamiento (hay una llamada), éste se toma del bloque comenzando desde el final del último espacio asignado (**secuencialmente**)
- una vez terminada la llamada, el espacio **se libera en orden inverso** al que fue asignado, por lo que el espacio libre siempre está en la cima de la pila

# Gestión de Memoria

## Almacenamiento dinámico: en *heap*

Un **heap** es una región de almacenamiento en la que los bloques de memoria se asignan y liberan en *momentos arbitrarios*

- El almacenamiento en *heap* es necesario cuando el lenguaje permite estructuras de datos (e.g., conjuntos o listas) cuyo tamaño puede cambiar en tiempo de ejecución.
- Los subbloques asignados pueden ser del **mismo tamaño siempre o de tamaño variable**
- La desasignación puede ser
  - explícita (ej. C, C++, Pascal)
  - implícita (cuando el elemento asignado ya no es alcanzable por ninguna variable del programa)

# Gestión de Memoria

## Almacenamiento dinámico: en *heap*

Un **heap** es una región de almacenamiento en la que los bloques de memoria se asignan y liberan en *momentos arbitrarios*

- El almacenamiento en *heap* es necesario cuando el lenguaje permite estructuras de datos (e.g., conjuntos o listas) cuyo tamaño puede cambiar en tiempo de ejecución.
- Los subbloques asignados pueden ser del **mismo tamaño siempre o de tamaño variable**
- La desasignación puede ser
  - explícita (ej. C, C++, Pascal)
  - implícita (cuando el elemento asignado ya no es alcanzable por ninguna variable del programa)
- **Garbage collector**: mecanismo del lenguaje que identifica los elementos inalcanzables y desasigna la memoria que ocupan, la cual pasa a estar libre

Motivación

Conceptos

Tipos y sistemas  
de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y  
control de flujo

Paso de  
parámetros

Alcance de las  
variables

Gestión de  
memoria

Paradigmas  
de programación

Imperativo

Declarativo

OO

Concurrente

Otros  
paradigmas

Basado en  
interacción

Bibliografía

# Tema 1. Introducción (Parte II)

## Lenguajes, Tecnologías y Paradigmas de Programación (LTP)

DSIC, ETSInf



# Paradigmas de Programación

## Factores de éxito de un LP

### Motivación

### Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

### Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

### Otros paradigmas

Basado en interacción

### Bibliografía

- **Potencia expresiva**: para generar código claro, conciso y fácil de mantener
- **Fácil** de aprender
- **Portable** y con garantías para la seguridad
- Soportado por **múltiples plataformas** y herramientas de desarrollo
- Respaldo **económico**
- Fácil **migración** de aplicaciones escritas en otros lenguajes (C++ → Java)
- Múltiples **bibliotecas** para gran variedad de aplicaciones
- Disponibilidad de descarga de **código abierto** escrito en el lenguaje



# Paradigmas de programación

## Motivación

## Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

## Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

## Otros paradigmas

Basado en interacción

## Bibliografía

## Definición de paradigma de programación

Modelo básico de diseño y desarrollo de programas que proporciona un conjunto de métodos y técnicas para producir programas con unas directrices específicas (estilo y forma de plantear la solución al problema)

### Principales paradigmas:

- Imperativo
- Declarativo
  - funcional
  - lógico
- Orientado a objetos
- Concurrente

Existen también los llamados paradigmas *emergentes*

# Paradigma imperativo

## Motivación

### Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

### Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

### Otros paradigmas

Basado en interacción

### Bibliografía

Describe la programación como una secuencia de instrucciones o comandos que cambian el estado del programa.

- Establece el **cómo** proceder → **algoritmo**
- El concepto básico es el **estado de la máquina**, el cual se define por los valores de las variables involucradas y que se almacenan en la memoria
- Las instrucciones suelen ser secuenciales y el programa consiste en **construir la secuencia de estados** de la máquina que conduce a la solución
- Este modelo está muy vinculado a la arquitectura de la máquina convencional (*Von Neumann*)
- Programa estructurado en bloques y módulos
- Eficiente, difícil de modificar y verificar, con **efectos laterales**

# Paradigma imperativo

## Ejemplo en Pascal

### Motivación

### Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

### Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

### Otros paradigmas

Basado en interacción

### Bibliografía

Función **length** en Pascal:

```
function length (l : list): integer
var
    b : boolean;
    aux : list;
begin
    b := is_empty(l);
    case b of
        true : length := 0;
        false : begin
            aux := tail(l);
            length := 1+length(aux);
        end;
    end;
end;
```

## Motivación

## Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

## Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

## Otros paradigmas

Basado en interacción

## Bibliografía

# Paradigma imperativo

Características: Efectos laterales

Puede ocurrir que dos llamadas a función con los mismos argumentos den resultados diferentes

```
program proof;  
var  
    flag : boolean;  
function f (x : integer) : integer;  
begin  
    flag := not flag;  
    if flag then f := x else f := x+1;  
end;  
begin  
    flag := false;  
    write(f(1));  
    write(f(1));  
end
```

variable global

f cambia el valor de la variable global

## Motivación

## Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

## Paradigmas de programación

## Imperativo

Declarativo

OO

Concurrente

## Otros paradigmas

Basado en interacción

## Bibliografía

# Paradigma imperativo

Características: Efectos laterales

Puede ocurrir que dos llamadas a función con los mismos argumentos den resultados diferentes

```
program proof;
var
    flag : boolean;
function f (x : integer) : integer;
begin
    flag := not flag;
    if flag then f := x else f := x+1;
end;
begin
    flag := false;
    write(f(1));
    write(f(1));
end
```

Salida del programa:

```
> proof
1
2
```

# Programación imperativa

## Características

### Motivación

### Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

### Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

### Otros paradigmas

Basado en interacción

### Bibliografía

- Pone el énfasis en el **cómo** resolver un problema
- Las sentencias de los programas se ejecutan en el orden en que están escritas y dicho **orden de ejecución** es crucial
- **Asignación destructiva** (el valor asignado a una variable destruye el valor anterior de dicha variable) → efectos laterales que oscurecen el código
- El **control** es responsabilidad del programador
- Más **complejo** de lo que parece (así lo demuestra la complejidad de sus definiciones semánticas o la dificultad de las técnicas asociadas, e.g., de verificación formal)
- **Difícil de paralelizar**
- Los programadores están mejor dispuestos a sacrificar las características avanzadas a cambio de poder obtener mayor velocidad de ejecución

## Paradigma declarativo

## Motivación

## Conceptos

## Tipos y sistemas de tipos

## Polimorfismo

### Sobrecarga

### Coerción

### Genericidad

## Inclusión

### Reflexión

### Procedimientos y control de flujo

### Paso de parámetros

### Alcance de las variables

## Gestión de memoria

## Paradigmas de programación

Imperativo

### Declarativo

00

Concurrente

## Otros paradigmas

Basado en  
interacción

## Bibliografía

Describe las propiedades de la solución buscada, dejando indeterminado el algoritmo (conjunto de instrucciones) usado para encontrar esa solución

- Responde a la idea propuesta por Kowalski

*PROGRAMA = LÓGICA + CONTROL*

- Lógica: se relaciona con el establecimiento del **Qué**
- Control: se relaciona con el establecimiento del **Cómo**
- El programador se centra en **aspectos lógicos de la solución** y deja los aspectos de control al sistema
- Fácil de verificar y modificar, conciso y claro

# Paradigma declarativo

## Motivación

## Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

## Paradigmas de programación

Imperativo

**Declarativo**

OO

Concurrente

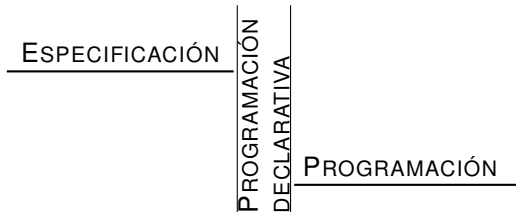
## Otros paradigmas

Basado en interacción

## Bibliografía

Un programa declarativo puede entenderse como una **especificación ejecutable**.

Leng. declarativo = Lenguaje de **ESPECIFICACIÓN** (ejecutable)  
Lenguaje de **PROGRAMACIÓN** (alto nivel)





## Motivación

## Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

## Paradigmas de programación

Imperativo

**Declarativo**

OO

Concurrente

## Otros paradigmas

Basado en interacción

## Bibliografía

# Paradigma declarativo

## Especificación vs programación

### Especificación: Definición de función matemática

$$\text{fib}(0) = 1$$
$$\text{fib}(1) = 1$$
$$\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$$

# Paradigma declarativo

## Especificación vs programación

### Motivación

### Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

### Paradigmas de programación

Imperativo

**Declarativo**  
OO

Concurrente

### Otros paradigmas

Basado en interacción

### Bibliografía

## Especificación: Definición de función matemática

$$\text{fib}(0) = 1$$

$$\text{fib}(1) = 1$$

$$\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$$

## Programa (dos versiones):

### Directamente la especificación:

$$\text{fib}(0) = 1$$

$$\text{fib}(1) = 1$$

$$\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$$

### Variante optimizada con acumulador

$$\text{fib}(0) = 1$$

$$\text{fib}(1) = 1$$

$$\text{fib}(n) = \text{fib\_aux}(1, 1, n)$$

$$\text{fib\_aux}(x, y, 0) = x$$

$$\text{fib\_aux}(x, y, n) = \text{fib\_aux}(y, x+y, n-1)$$

# Paradigma declarativo

## Motivación

## Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

## Paradigmas de programación

Imperativo

**Declarativo**

OO

Concurrente

## Otros paradigmas

Basado en interacción

## Bibliografía

- **Paradigma funcional** (basado en el  $\lambda$ -cálculo)
  - definición de estructuras de datos y funciones que manipulan las estructuras mediante ecuaciones
  - **polimorfismo**
  - orden superior
- **Paradigma lógico** (basado en la lógica de primer orden)
  - definición de relaciones mediante reglas:
 

*Si  $C1$  y  $C2$  y ...  $Cn$ , entonces  $A$*   
*escrito  $A \leftarrow C1, C2, \dots Cn$*
  - variables lógicas
  - indeterminismo

## Motivación

## Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

## Paradigmas de programación

Imperativo

**Declarativo**

OO

Concurrente

## Otros paradigmas

Basado en interacción

## Bibliografía

## Paradigma declarativo

## Ejemplo en Haskell y Prolog

La función `length` de una lista:

## En Haskell

```
data list a = [] | a:list a
```

```
length [] = 0
```

```
length (x:xs) = (length xs) + 1
```

## En Prolog

```
length([],0).
```

```
length([X|Xs],N) :- length(Xs,M), N is M + 1.
```

# Programación declarativa

## Características

### Motivación

### Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

### Paradigmas de programación

Imperativo

**Declarativo**

OO

Concurrente

### Otros paradigmas

Basado en interacción

### Bibliografía

- Expresa **qué** es la solución a un problema
- El **orden** de las sentencias y expresiones **no tiene por qué afectar a la semántica** del programa
- Una **expresión denota un valor** independiente del contexto (**transparencia referencial**)
- Nivel más alto de programación
  - semántica más sencilla
  - control automático
  - más fácil de paralelizar
  - mejor mantenimiento
  - mayor potencia expresiva
  - menor tamaño del código
  - mayor productividad
- Eficiencia comparable a la de lenguajes como Java.
- La curva de aprendizaje es más lenta cuando se aprendió a programar en un paradigma más convencional
- Las *impurezas* de sistemas reales son difíciles de modelar de manera declarativa

## Paradigma imperativo

## Bibliografía

## Referencias a memoria

# LÓGICA

como lenguaje de programación

## PROGRAMA

## Especificación de un problema

## INSTRUCCIONES

## Fórmulas lógicas

## MODELO DE COMPUTACIÓN

## Máquina de inferências

## VARIABLES

## Variables lógicas

## Motivación

## Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

## Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

## Otros paradigmas

Basado en interacción

## Bibliografía

# Paradigma imperativo vs Paradigma declarativo

Un ejemplo

## ¿Qué hace este programa imperativo?

```

void f(int a[], int lo, hi){
    int h, l, p, t;

    if (lo<hi) {
        l = lo;
        h = hi;
        p = a[hi];
        do {
            while ((l<h) &&
                (a[l] <= p))
                l = l+1;
            while ((h>l) &&
                (a[h] >= p))
                h = h-1;
            if (l<h) {
                t = a[l];
                a[l] = a[h];
                a[h] = t;
            }
            a[hi] = a[l];
            a[l] = p;
            f(a, lo, l-1);
            f(a, l+1, hi);
        }
    }
}

```



## Motivación

## Conceptos

Tipos y sistemas  
de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y  
control de flujoPaso de  
parámetrosAlcance de las  
variablesGestión de  
memoriaParadigmas  
de programación

Imperativo

**Declarativo**

OO

Concurrente

Otros  
paradigmasBasado en  
interacción

## Bibliografía

# Paradigma imperativo vs Paradigma declarativo

Un ejemplo

## ¿Qué hace este programa declarativo?

```
f :: Ord a => [a] -> [a]

f [] = []
f (p:xs) = (f lesser) ++ [p] ++ (f greater)
  where
    lesser = filter (< p) xs
    greater = filter (>= p) xs
```

## Motivación

## Conceptos

Tipos y sistemas  
de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y  
control de flujoPaso de  
parámetrosAlcance de las  
variablesGestión de  
memoriaParadigmas  
de programación

Imperativo

**Declarativo**

OO

Concurrente

Otros  
paradigmasBasado en  
interacción

## Bibliografía

# Paradigma imperativo vs Paradigma declarativo

Un ejemplo

## ¿Qué hace este programa declarativo?

```
f :: Ord a => [a] -> [a]

f [] = []
f (p:xs) = (f lesser) ++ [p] ++ (f greater)
  where
    lesser = filter (< p) xs
    greater = filter (>= p) xs
```

- Sin asignación de variables
- Sin índices de vector
- Sin gestión de memoria

# Paradigma orientado a objetos

## Motivación

## Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

## Paradigmas de programación

Imperativo

Declarativo

oo

Concurrente

## Otros paradigmas

Basado en interacción

## Bibliografía

Basado en la idea de encapsular en objetos estado y operaciones

- Objeto: estado + operaciones
- Concepto de *clase*, *instancia*, *subclases* y **herencia**
- Elementos fundamentales:
  - abstracción
  - encapsulamiento
  - modularidad
  - jerarquía

# Paradigma orientado a objetos

Ejemplo en Java

## Motivación

## Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

## Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

## Otros paradigmas

Basado en interacción

## Bibliografía

La clase *Circle* es una **abstracción** de la noción de *círculo*:

```
public class Circle {           // class name
    double radius;              // variables (state)
    String color;

    double getRadius() {...}    // methods (operations)
    double getArea() {...}
}
```

A partir de una clase se definen **instancias** que representan los objetos concretos (círculos con un radio y color específicos)

```
Circle c1, c2;
c1 = new Circle(2.0, "blue");
c2 = new Circle(3.0, "red");

Circle c3 = new Circle(1.5, "red");
```

# Paradigma concurrente

## Motivación

## Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

## Paradigmas de programación

Imperativo

Declarativo

OO

**Concurrente**

## Otros paradigmas

Basado en interacción

## Bibliografía

- Los lenguajes de programación concurrentes utilizan para programar la **ejecución simultánea de múltiples tareas**
- Las tareas pueden consistir en un **conjunto de procesos** creados por un único programa

Acceso concurrente en bases de datos, uso de recursos de un sistema operativo, etc.

- El inicio de la programación concurrente está en la invención de la **interrupción** a finales de los 50.
  - Interrupción: mecanismo hardware que interrumpe el programa en ejecución y hace que la unidad de proceso bifurque el control a una dirección dada, donde reside otro programa que tratará el evento asociado a la interrupción

## Motivación

## Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

## Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

## Otros paradigmas

Basado en interacción

## Bibliografía

## Paradigma concurrente

## Problemas asociados a la concurrencia

- **Corrupción de los datos** compartidos

Cuando dos procesos escriben concurrentemente en la pantalla puede producirse una mezcla incomprensible

- **Interbloqueos** entre procesos que comparten recursos

A necesita dos recursos compartidos (R1 y R2). Trata de obtener los recursos en exclusiva (para evitar corrupción de datos) solicitando R1 y luego R2. Mientras, B solicita R2 y luego R1. Cada uno obtiene un recurso, pero ninguno puede obtener el segundo

- **Inanición** de un proceso que no consigue un recurso dado.

Normalmente el SO organiza una cola de procesos para los recursos compartidos en función de la prioridad de dichos procesos. Dos procesos con alta prioridad podrían estar acaparando el recurso.

- **Indeterminismo** en el orden en el que se entrelazan las acciones de los distintos procesos.

Dificulta la depuración ya que los errores pueden depender de dicho orden

# Paradigma concurrente

## Conceptos propios: Primeras abstracciones (1/2)

### Motivación

### Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

### Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

### Otros paradigmas

Basado en interacción

### Bibliografía

- La manera primitiva de definir un lenguaje concurrente consistió en añadir a un lenguaje secuencial (Simula) primitivas del SO para la creación de procesos (corutinas)
  - Problema: bajo nivel y falta de portabilidad
- Dijkstra introdujo (1965-71) las primeras abstracciones.
  - **Programa concurrente:** conjunto de procesos secuenciales asíncronos que no hacen suposiciones sobre las velocidades relativas con las que progresan otros procesos
  - Introduce los **semáforos** como mecanismo de sincronización

## Motivación

## Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

## Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

## Otros paradigmas

Basado en interacción

## Bibliografía

## Paradigma concurrente

## Conceptos propios: Primeras abstracciones (2/2)

- Hoare introduce la noción de **región crítica** para evitar interbloqueos
  - gestionar las regiones críticas era ineficiente y poco modular
- En 1974 se introduce el concepto de **monitor** (inspirado en los TAD) para encapsular los recursos compartidos.
  - El primer lenguaje concurrente de alto nivel con monitores fue Pascal concurrente (1975), después incorporado a Modula-2.
- Surgen modelos, independientes de la arquitectura, que permiten el análisis de los programas concurrentes (CSP, CCS,  $\pi$ -cálculo, redes de petri, PVM)
  - estos modelos influyen en distintos lenguajes, por ejemplo CSP influyó en los canales de Occam y las llamadas remotas de ADA



## Motivación

## Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

## Paradigmas de programación

Imperativo

Declarativo

OO

**Concurrente**

## Otros paradigmas

Basado en interacción

## Bibliografía

# Paradigma concurrente

## Ejemplo en Java de definición de hilos (1/2)

En Java existen dos formas de crear hilos:

- usando herencia (`extends`)
- usando interfaces (`implements`)

### Usando herencia

Se define la clase `MyThread` heredando de la clase `Java Thread`

```
class MyThread extends Thread {  
    public void run () {  
        // cuerpo de la tarea a ejecutar  
        // por el thread  
    }  
}
```

## Motivación

## Conceptos

Tipos y sistemas  
de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y  
control de flujoPaso de  
parámetrosAlcance de las  
variablesGestión de  
memoriaParadigmas  
de programación

Imperativo

Declarativo  
OO

Concurrente

Otros  
paradigmasBasado en  
interacción

## Bibliografía

# Paradigma concurrente

## Ejemplo en Java de uso de hilos (2/2)

Se crea y usa una instancia de la clase `MyThread`

```
MyThread t1 = new MyThread();  
t1.setPriority(5)  
t1.start();  
System.out.println("Puedo seguir con mis cosas");  
// ...
```

- El método `start` inicia la ejecución del hilo (e invocará al método `run`)
- La asignación de prioridad es opcional (entero entre 1 y 10, siendo 10 la mayor prioridad)
- El mensaje se mostrará por la salida independientemente de la ejecución del hilo arrancado

## Motivación

## Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

## Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

## Otros paradigmas

Basado en interacción

## Bibliografía

## Paradigma concurrente

## Algunas consideraciones de la concurrencia en Java

- Java soporta la programación concurrente de forma **nativa** (no mediante bibliotecas) gracias a la clase `Thread`
- Un **hilo** (*thread*) es un concepto similar al de proceso. La diferencia es que los hilos siempre dependen de un *programa padre* en cuanto a recursos para su ejecución.
  - Un proceso puede mantener su propio espacio de direcciones y entorno de ejecución
- El programador tiene funciones para (por ejemplo) crear, arrancar, abortar, priorizar, suspender o reanudar hilos
- La máquina virtual de java se encarga de organizar los hilos, pero es responsabilidad del programador evitar los problemas indeseados de la concurrencia (inanición, etc.)
- La comunicación es mediante **memoria compartida**. Como ayuda, cada objeto tiene implícitamente un bloqueo para cuando está siendo utilizado por un hilo.

# Programación paralela

## Motivación

## Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

## Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

## Otros paradigmas

Basado en interacción

## Bibliografía

## Objetivo:

Aceleración de algoritmos que consumen muchas horas de proceso dividiendo el tiempo de ejecución mediante el uso de varios procesadores, **distribución de los datos** y **reparto de la carga**.

- Con la aparición de los primeros microprocesadores (1975), los procesos pasaron a ejecutarse concurrentemente **en distintos procesadores**, por lo que dejaba de valer el principio de disponer de una memoria común.
  - surgen nuevas construcciones para la comunicación entre procesos, como el **paso de mensaje entre procesadores** *rendez-vous*.
- Primeros lenguajes paralelos: los secuenciales Fortran o C extendidos con bibliotecas de paso de mensajes dependientes del fabricante.

# Programación paralela vs Programación concurrente

## Motivación

## Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

## Paradigmas de programación

Imperativo

Declarativo

OO

**Concurrente**

## Otros paradigmas

Basado en interacción

## Bibliografía

	PARALELA	CONCURRENTE
OBJETIVO	Eficiencia: reparto de carga	Varios procesos interactúan simultáneamente
PROCESADORES	solo se concibe con varios	es compatible con uno
COMUNICACIÓN	paso de mensajes y/o memoria compartida	

## Paradigma basado en interacción

- El paradigma tradicional sigue la idea de *programación como cálculo en el modelo de Von Neumann*
  - un programa describe la secuencia de pasos necesarios para producir la salida a partir de una entrada
- En algunas áreas este modelo no se adapta bien: robótica, AI, aplicaciones orientadas a servicios, ...

Tiene más sentido la

**Programación como interacción:** las entradas se monitorizan y las salidas son acciones que se llevan a cabo dinámicamente (no hay un *resultado final*)

# Paradigma basado en interacción

## Programa interactivo

Es una comunidad de entidades (agentes, bases de datos, servicios de red, etc.) que interactúan siguiendo ciertas **reglas de interacción**

- Las reglas de interacción pueden estar restringidas por interfaces, protocolos y ciertas garantías del servicio (tiempo de respuesta, confidencialidad de datos, etc.)
- Instancias del modelo de programación interactiva:
  - **Programación conducida por eventos**
  - **Arquitectura cliente/servidor**
  - **Sistemas reactivos**
  - **Software basado en agentes**
  - **Sistemas empotrados**
- Usado en aplicaciones distribuidas, diseño de GUI, programación web, diseño incremental de programas (se refinan partes de un programa mientras está en ejecución)

## Motivación

## Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

## Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

## Otros paradigmas

Basado en interacción

## Bibliografía

# Paradigma basado en interacción

## Programación por eventos

- El flujo del programa está determinado por eventos

Eventos: señales de sensores o, más comúnmente, acciones de usuario en la interfaz, mensajes desde otros programas o procesos, ...

- La arquitectura típica de una aplicación dirigida por/basada en eventos (*event-driven/event-based*) consiste en un bucle principal dividido en dos secciones independientes:
  - 1 detección o selección de eventos (*event-detection*)
  - 2 manejo de los eventos (*event-handling*)
- En el caso de *software* empotrado, la primera sección reside en el *hardware* y se gestiona mediante interrupciones



## Paradigma basado en interacción

## Programación por eventos

La programación por eventos es una caracterización ortogonal a otros paradigmas:

- Se puede usar cualquier lenguaje de alto nivel para escribir programas siguiendo el estilo *event-driven*.
- Puede o no ser orientada a objetos
- No implica programación concurrente
- Requisitos:
  - poder detectar señales, interrupciones al procesador o eventos de la GUI
  - poder gestionar una cola de eventos para responder a los mismos

## Motivación

## Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

## Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

## Otros paradigmas

Basado en interacción

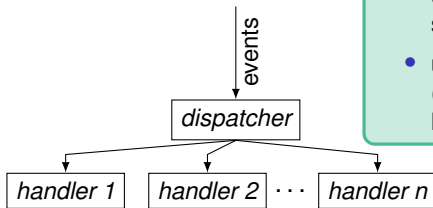
## Bibliografía

# Paradigma basado en interacción

## Programación por eventos

- Los *patrones de diseño* (en particular el patrón *event-handler*) suelen ser una ayuda que simplifica la tarea de programar este tipo de aplicaciones.

### El patrón *event-handler*



- un *dispatcher* que gestiona la secuencia de eventos
- un conjunto de manejadores (*handlers*) que implementan las acciones de respuesta

## Motivación

## Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

## Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

## Otros paradigmas

Basado en interacción

## Bibliografía

# Paradigma basado en interacción

Programación por eventos. Un ejemplo de *dispatcher*

```
do forever: // the event loop
  get an event from the input stream

  if event.type == EndOfEventStream
    quit // break out of event loop

  if event.type == ...:
    call the appropriate handler, passing it
    event information as an argument

  elseif event.type == ...:
    call the appropriate handler, passing it
    event information as an argument

  else: // unrecognized event type
    ignore the event, or raise an exception
```

**bucle principal**

**salida del bucle**

**selección de handler**

## Motivación

## Conceptos

Tipos y sistemas  
de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y  
control de flujoPaso de  
parámetrosAlcance de las  
variablesGestión de  
memoriaParadigmas  
de programación

Imperativo

Declarativo

OO

Concurrente

Otros  
paradigmasBasado en  
interacción

## Bibliografía

# Paradigma basado en interacción

## Consideraciones finales

La programación basada en eventos se usa masivamente en la programación de GUIs, principalmente debido a que la mayoría de herramientas de desarrollo comerciales disponen de **asistentes** para la definición asistida de este esquema

- Ventaja:
  - Simplifica la tarea del programador al proporcionar una implementación por defecto para el bucle principal y la gestión de la cola de eventos
- Desventajas:
  - promueve un modelo de interacción excesivamente simple
  - es difícil de extender
  - es propenso a errores ya que dificulta la gestión de recursos compartidos

# Otros paradigmas emergentes

## Motivación

## Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

## Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

## Otros paradigmas

Basado en interacción

## Bibliografía

- **BIO-COMPUTACIÓN:** Existen modelos de computación inspirados en la **biología**
  - utilizan conceptos y técnicas que se emplean en sistemas de la naturaleza como base para desarrollar nuevas técnicas de programación
- **COMPUTACIÓN CUÁNTICA:** reemplaza los circuitos clásicos por otros que utilizan puertas cuánticas (en vez de puertas lógicas)

# ¿A qué paradigma pertenecen los lenguajes?

## Motivación

## Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

## Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

## Otros paradigmas

Basado en interacción

## Bibliografía

La mayoría son multi-paradigma:

- **CoffeeScript** (2009): Es un lenguaje orientado a objetos, basado en prototipos, funcional e imperativo. CoffeeScript se compila a JavaScript.
- **Scala** (2003): Orientado a objetos, imperativo y funcional (usado por Twitter junto con Ruby).
- **Erlang** (1986): funcional y concurrente (usado por HP, Amazon, Ericsson, Facebook, ...)
- **Python** (1989): funcional (listas intensionales, abstracción lambda, fold, map) y orientado a objetos (herencia múltiple)

## Motivación

## Conceptos

Tipos y sistemas  
de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y  
control de flujoPaso de  
parámetrosAlcance de las  
variablesGestión de  
memoriaParadigmas  
de programación

Imperativo

Declarativo

OO

Concurrente

Otros  
paradigmasBasado en  
interacción

## Bibliografía

- Cortazar, Francisco. *Lenguajes de programación y procesadores*. Editorial Cera, 2012.
- Peña, Ricardo. *De Euclides a Java: historia de algoritmos y lenguajes de programación*, Editorial Nivola, 2006.
- Pratt, T.W.; Zelkowitz, M.V. *Programming Languages: design and implementation*, Prentice-Hall, 2001 (versión de 1998 en castellano)
- Scott, M.L. *Programming Language Pragmatics*, Morgan Kaufmann Publishers, 2008 (versión revisada).
- Schildt, Herbert. *Java. The Complete Reference*. Eight Edition. The McGraw-Hill eds. 2011

Motivación

Conceptos

Tipos y sistemas de tipos

Polimorfismo

Sobrecarga

Coerción

Genericidad

Inclusión

Reflexión

Procedimientos y control de flujo

Paso de parámetros

Alcance de las variables

Gestión de memoria

Paradigmas de programación

Imperativo

Declarativo

OO

Concurrente

Otros paradigmas

Basado en interacción

Bibliografía

# Bibliografía

## Aspectos de implementación

- “Programming Language Pragmatics”, M.L. Scott. (cap. 3)
- “Lenguajes de programación y procesadores”, Francisco Cortazar (cap. 1)
- “Programming Languages: design and implementation”, Pratt, T.W.; Zelkowitz, M.V. (cap. 9 y 10)