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

DSIC
DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

etsinf
Escuela Técnica Superior de Ingeniería Informática

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

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

# Objetivos del tema

- 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.

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 historia que empezó en 1950

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

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. Programación como un oficio

- 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 $x=10$.

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

# 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

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

# Conceptos esenciales

Destacamos los siguientes conceptos:

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

# Tipos y sistemas de tipos

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*

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

# Tipos y sistemas de tipos

## Lenguajes tipificados

- 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

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

# Tipos y sistemas de tipos

## Lenguajes tipificados

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

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

# Tipos y sistemas de tipos

## Lenguajes tipificados

| Lenguajes tipificados | = | Expresiones de programa | + | 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.

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

# Tipos y sistemas de tipos

## Ejemplos de tipificación

- 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`

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

# Tipos y sistemas de tipos

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:

$\tau$ `::=` `Bool` | `Char` | `Int` | $\cdots$ | $\tau \rightarrow \tau$ | $\tau \times \tau$ | $[\tau]$

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

# 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

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

# Tipos y sistemas de tipos

## Ejemplo de expresiones de tipos

- 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 `fac` 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.

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

Otr pa

Basado en interacción

Bibliografía

# Tipos y sistemas de tipos

## Ejemplo de expresiones de tipos

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

tipo polimórfico

- Expresión de tipo parametrizado.

`[a]` $\rightarrow$ `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

# Polimorfismo

- 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

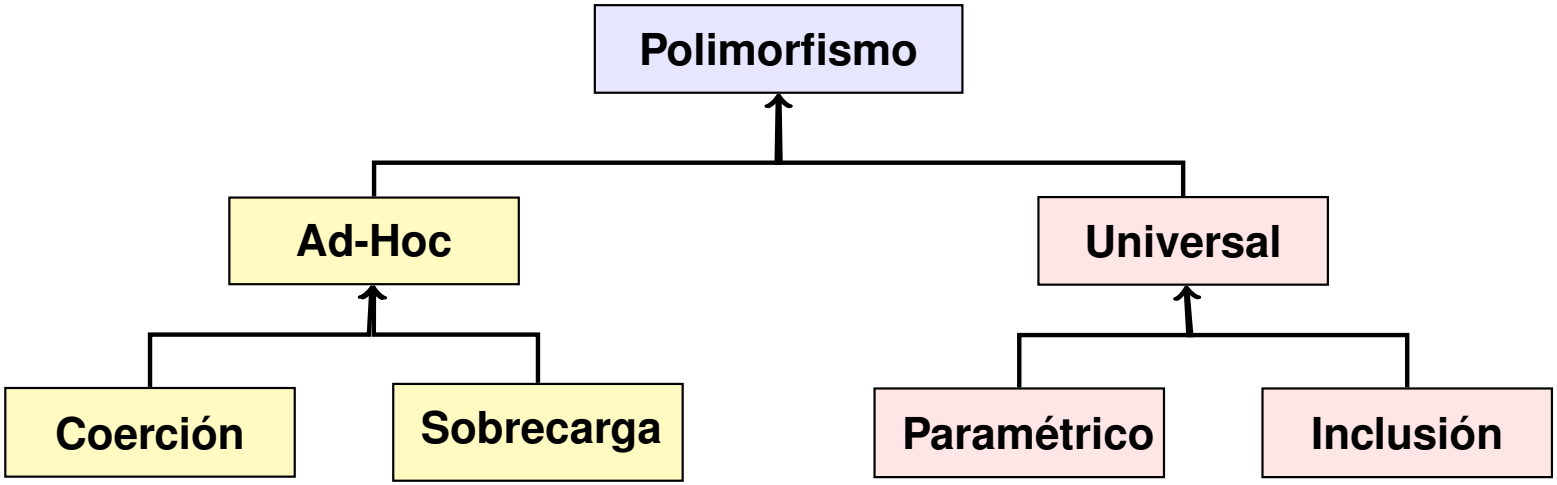
Otros paradigmas

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)

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. Sobrecarga

## Polimorfismo Ad-Hoc: Sobrecarga

- **Sobrecarga**: existencia de distintas funciones con el mismo nombre.
  - Los operadores aritméticos $+$, $-$, $*$, $/$, ... suelen estar sobrecargados:

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

    corresponden a distintos usos de $+$
  - El operador $+$ no puede recibir el politipo

    ```
    (+) :: a -> a -> 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

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

## Ejemplo de coerción en Java

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);
```

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. Genericidad

## Polimorfismo Universal: Genericidad

- **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

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. Genericidad

## Ejemplo de genericidad en Haskell

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] -> Int
length [] = 0
length (x:xs) = 1 + (length xs)
```

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. Genericidad

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

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

```
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){
  ...
  }
```

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. Genericidad

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

Ejemplo de uso de entradas de un diccionario:


```
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));
```

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. Genericidad

## Algunas consideraciones de la genericidad en Java

- 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

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

## Polimorfismo Universal: Inclusión/Herencia

- **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***

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

## 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 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 (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 startGear){
    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

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 (1/3)

- 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;
```

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 (2/3)

- 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 qu*
*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

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 redefinición de método heredado en Java (1/2)

```
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)

```
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.

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 (1/2)

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

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 double side;
  public Square(double initX, double initY, double initSide){
    super(initX, initY); // Call to super constructor
    side = initSide;
  }
  public double perimeter(){ return 4 * side; }
  public double area(){ return side * side; }
}

public class Circle extends Shape {
  private double radius;
  public Circle(double initX, double initY,
                double initRadius){
    super(initX, initY); // Call to super constructor
    radius = initRadius;
  }
  public double perimeter(){ return 2 * Math.PI * radius; }
  public double area(){ return Math.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 double side;
  public Square(double initX, double initY, double initSide){
    super(initX, initY); // Call to super constructor
    side = initSide;
  }
  public double perimeter(){ return 4 * side; }
  public double area(){ return side * side; }
}

public class Circle extends Shape {
  private double radius;
  public Circle(double initX, double initY,
                double initRadius){
    super(initX, ini
    radius = initRad
  }
  public double peri                                    }
  public double area(){ return Math.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 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

## 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.)*

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

## Debe usarse con cautela

- Mal usada puede afectar
  - al rendimiendo 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).

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

## 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

# Reflexión

## Ejemplo de uso de reflexión en Java

```
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();
...
```

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

# Procedimientos y control de flujo

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.

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

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

- LLAMADA: $f(e_1, \ldots, e_n)$ con $e_1, \ldots, e_n$ 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.
  - $e_1, \ldots, e_n$ son los llamados parámetros de entrada/reales (*actual parameters*)
- DECLARACIÓN: $f(x_1, \ldots, x_n)$ con $x_1, \ldots, x_n$ variables.
  - $x_1, \ldots, x_n$ son los llamados parámetros formales (*formal parameters*)
  - Los parámetros formales son variables locales al cuerpo de la función declarada

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

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 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);
```

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;                                    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 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`

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 = 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 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;
}                                          v = 20
...
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
```

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

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

- 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;                                    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
```

## 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 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 = 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;
}                                          v = 20
...
int a = 10;
inc(a);                                    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).

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 (2/2)

- 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.

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 de las variables

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

```
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 *}
```

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 de las variables

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

```
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 *}
```

¿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?

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 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`).

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 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)

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

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 tienen 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)

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

## 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

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 *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)

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 *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