Tema 1. Introducció (Part 1)

Llenguatges, Tecnologies i Paradigmes de Programació (LTP)

DSIC, ETSInf

DSIC
DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

etsinf
Escuela Técnica Superior de Ingeniería Informática

Objectius del tema

- Conèixer l'evolució dels llenguatges de programació (LP) i quins han sigut les seues aportacions més importants quant a l'impacte en el disseny d'altres llenguatges.
- Entendre els principals paradigmes de programació disponibles avui en dia i les seues característiques principals.
- Comprendre els diferents mecanismes d'abstracció (genericidat, herència i modularització) i pas de paràmetres.
- Identificar aspectes fonamentals dels LP: abast estàtic/dinàmic, gestió de memòria.
- Entendre els criteris que permeten triar el paradigma/llenguatge de programació més adequat en funció de l'aplicació, envergadura i metodologia de programació.
- Entendre les característiques dels LP en relació al model subjacent (paradigma) i als seus components fonamentals (sistemes de tipus i classes, model d'execució, abstraccions).
- Entendre les implicacions dels recursos expressius d'un LP quant a la seua implementació.

Una història que va començar en 1950

ANYS 50:

- Temps programador barat, màquines cares:

  keep the machine busy

- Quan no es programava directament el hardware, el programa es compilava a mà per a obtenir la màxima eficiència, per a un hardware concret:

  connexió directa entre llenguatge i hardware

ACTUALITAT:

- Temps programador car, màquines barates:

  keep the programmer busy

- El programa es construeix per a ser eficient i es compila automàticament per a generar codi portable que siga, alhora, eficient:

  connexió directa entre disseny del programa i llenguatges: objectes, concurrència, etc.

Ensenyament dels LP

Tres aproximacions

1. Programació com un ofici
2. Programació com una branca de les matemàtiques
3. Programació en termes de conceptes

1.Programació com un ofici

- S'estudia en un paradigma únic i amb un únic llenguatge
- Pot ser contraproduent. Per exemple, aprendre a manipular llistes en certs llenguatges pot portar a la conclusió errònia que el maneig de llistes és sempre tan complicat i costós:

1.Programació com un ofici

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);
}
```

Eixida

```
[(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
```

Ús

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

2. Programació com una branca de les matemàtiques

- O bé s'estudia en un llenguatge *ideal*, restringit (Dijkstra) o el resultat és massa teòric, allunyat de la pràctica.

2. Programació com una branca de les matemàtiques

Exemple: verificació formal (d'un programa d'una línia)

El programa

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

La prova

Partim de l'expressió (*Hoare triple*)
$\{x \leq 10\}$ `while (x<10) x:=x+1` $\{x=10\}$
La condició del bucle és $x<10$. Usem el invariant de bucle $x \leq 10$ i amb aquestes assumpcions podem provar l'expressió
$\{x<10 \wedge x \leq 10\}$ `x:=x+1` $\{x \leq 10\}$
Aquesta expressió es deriva formalment de les regles de la lògica de Floyd-Hoare, però també pot justificar-se de forma intuïtiva: *La computació comença en un estat on es compleix* $x<10 \wedge x \leq 10$*, la qual cosa és equivalent a dir que* $x<10$*. La computació afig 1 a* x*, per la qual cosa tenim que* $x \leq 10$ *és cert (en el domini dels enters)*
Sota aquesta premissa, la regla per al bucle while ens permet traure la conclusió
$\{x \leq 10\}$ `while (x<10) x:=x+1` $\{\neg(x<10) \wedge x \leq 10\}$
I podem veure que la postcondició d'aquesta expressió és lògicament equivalent a $x=10$.

3. Programació en termes de conceptes

- S'estudia un conjunt de conceptes semántics i estructures d'implementació en termes dels quals es descriuen de forma natural diferents llenguatges i les seues implementacions

3. Programació en termes de conceptes

Un llenguatge de programació pot combinar característiques de diferents blocs

Llenguatge funcional

(+) Polimorfisme
(+) Estrategies
(+) Ordre superior

Llenguatge lògic

(+) No determinisme
(+) Variables lógiques
(+) Unificació

Llenguatge *kernel*

(+) Abstracció de dades
(+) Recursió
(+) ...

Llenguatge imperatiu

(+) Estados explícits
(+) Modularidat
(+) Components

Llenguatge OO

(+) Classes
(+) Herència

Llenguatge *dataflow*

(+) Concurrencia

Conceptes essencials

Destaquem els següents conceptes:

- Tipus i sistemes de tipus
- Polimorfisme
- Reflexió
- Pas de paràmetres
- Àmbit de les variables
- Gestió de memòria

Tipus i sistemes de tipus

Un **tipus** representa el conjunt de valors que pot adoptar una variable o expressió. Els tipus:

- Ajuden a detectar **errors** de programació

  *Solament els programes que utilitzen les expressions segons el tipus que tenen aplicant les funcions permeses són **legals***

- Ajuden a **estructurar** la informació

  Els tipus poden veure's com a col·leccions de valors que comparteixen certes propietats

- Ajuden a manejar estructures de **dades**

  Els tipus indiquen com utilitzar les estructures de dades que comparteixen el mateix tipus mitjançant certes operacions

Tipus i sistemes de tipus

Llenguatges tipificats

- En els llenguatges **tipificats**, les variables tenen un tipus associat (e.g., C, C++, C#, Haskell, Java, Maude).
- Els llenguatges que no restringeixen el rang de valors que poden adoptar les variables són **no tipificats** (e.g., Lisp, Prolog).
    - També pot entendre's que tots els valors tenen un tipus únic o universal

Tipus i sistemes de tipus

Llenguatges tipificats

El sistema de tipus estableix què tipus d'associació de variables és possible:

- El **valor** associat a la variable ha de tenir el tipus d'aquesta (e.g., C, Haskell)
- El **valor** associat a la variable pot ser d'altres tipus *compatibles* relacionats amb el tipus de la variable (e.g., C++, C#, Java).
- De forma ortogonal, a més, la persistència del tipus del valor pot canviar:
  - Estático: el tipus no canvia durant l'execució
  - Dinámico: el tipus pot canviar en canviar el valor associat

Tipus i sistemes de tipus

Llenguatges tipificats

Llenguatges tipificats	=	Expressions de programa	+	Sistemes de tipus

- En els llenguatges amb tipificació **explícita**, els tipus formen part de la sintaxi.
- En els llenguatges amb tipificació **implícita**, els tipus **no** formen part de la sintaxi.

Tipus i sistemes de tipus

Examples de tipificación

- Llenguatge **no tipificat**: Prolog

```
objecte(clau).
objecte(pilota).
cosa(X) <- objecte(X).
```

La variable `X` no té tipus associat.

- Llenguatge amb **tipificació explícita**: Java

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

Totes les variables han de ser declarades, i en la declaració ha d'especificar-se el seu tipus explícitament

- Llenguatge amb **tipificació implícita**: Haskell

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

El sistema de tipus infereix automàticament el tipus de la variable `x`

Tipus i sistemes de tipus

Per a definir el tipus de les variables o expressions usem un *llenguatge d'expressions de tipus*.

Exemple de llenguatge d'expressions de tipus

- Tipus bàsics o primitius: `Bool`, `Char`, `Int`, ...
- Variables de tipus: `a`, `b`, `c`, ...
- Constructors de tipus:
  - $\rightarrow$ per a definir funcions,
  - $\times$ per a definir parells,
  - `[]` per a definir llistes
  - ...
- Regles de construcció de les expressions:

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

Tipus i sistemes de tipus

Tipus monomòrfics y tipus polimòrfics

- Els tipus en l'expressió dels quals de tipus **no** apareix cap variable de tipus es denominen monotipus o tipus monomòrfics.
- Els tipus en l'expressió dels quals de tipus apareix alguna variable de tipus es denominen politipus o tipus polimòrfics.
- Un tipus polimòrfic representa un conjunt infinit de monotips

Tipus i sistemes de tipus

Examples de expressions de tipus

- Expressió de tipus predefinit. Tipus bàsics `Bool`, `Int`, ...

`Bool` és el tipus dels valors booleans `True` i `False`

- Expressió de tipus funcional.

`Int` $\rightarrow$ `Int` és el tipus de la funció `fact` vista abans, que retorna el factorial d'un nombre.

- Expressió de tipus parametritzat.

`[a]` $\rightarrow$ `Int` és el tipus de la funció `length`, que calcula la longitud d'una llista.

Tipus i sistemes de tipus

Examples de expressions de tipus

- Expressió de tipus predefinit. Tipus bàsics `Bool`, `Int`, ...

`Bool` és el tipus dels valors booleans `True` i `False`

- Expressió de tipus monomòrfics

`Int` $\rightarrow$ `Int` és el tipus de la funció `fact` vista abans, que retorna el factorial d'un nombre.

- Expressió de tipus polimòrfic parametritzat.

`[a]` $\rightarrow$ `Int` és el tipus de la funció `length`, que calcula la longitud d'una llista.

variable de tipus

constructor de tipus

Polimorfisme

- És una característica dels llenguatges que permet manejar valors de diferents tipus usant una interfície uniforme
- S'aplica tant a funcions com a tipus:
  - Una funció pot ser polimòrfica pel que fa a un o varis dels seus arguments.

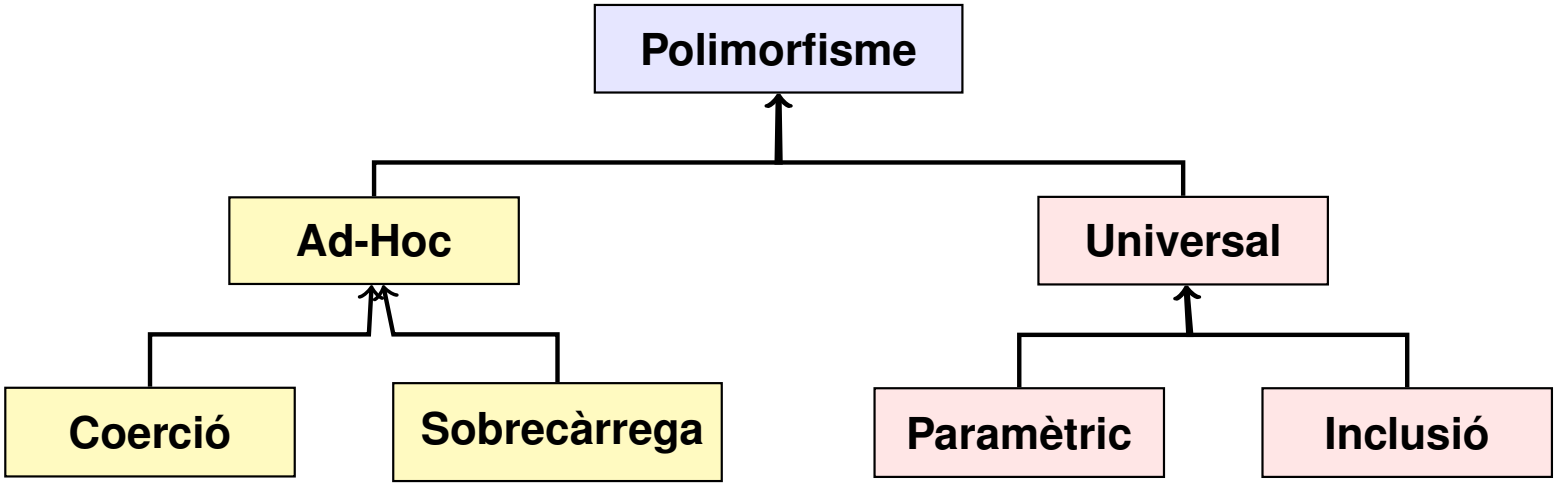
    La summa (+) pot aplicar-se a valors de diferents tipus com a sencers, reals, . . .

  - Un tipus de dades pot ser polimòrfic pel que fa als tipus dels elements que conté.

    Una llista amb elements d'un tipus arbitrari és un tipus polimòrfic

Polimorfisme

Tipus de polimorfisme



- Ad-hoc o aparent: treballa sobre un nombre finit de tipus no relacionats
  - Sobrecàrrega
  - Coerció
- Universal o vertader: treballa sobre un nombre potencialment infinit de tipus *amb certa estructura comuna*
  - paramètric (genericitat)
  - d'inclusió (herència)

Polimorfisme. Sobrecàrrega

Polimorfisme Ad-Hoc: Sobrecàrrega

- **Sobrecàrrega**: existència de diferents funcions amb el mateix nom.
  - Els operadors aritmètics $+$, $-$, $*$, $/$, ... solen estar sobrecarregats:

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

    corresponen a diferents usos de $+$
  - L'operador $+$ no pot rebre el politipus

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

    perquè significaria dotar de significat (i implementació) a *la suma* de caràcters, funcions, llestes, etc., la qual cosa pot interessar-nos o no

Polimorfisme. Sobrecàrrega

Example de sobrecàrrega en Java (1)

En Java, la sobrecàrrega de mètodes es realitza canviant el tipus dels paràmetres:

```
/* overloaded methods */

int myAdd(int x,int y, int z){
...
}

double myAdd(double x, double y, double z){
...
}
```

Polimorfisme. Sobrecàrrega

Example de sobrecàrrega 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());
  }
  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 té per què haver-hi coincidència quant al nombre ni quant al tipus dels paràmetres/resultat

Polimorfisme. Coerció

Polimorfisme Ad-Hoc: Coerció

- **Coerció**: conversió (implícita o explícita) de valors d'un tipus a un altre.
- Quan és implícita sol fer-se usant una jerarquia de tipus o de la seua representació.

Per exemple, en la majoria de llenguatges, per als arguments dels operadors aritmètics existeix coerció entre valors sencers i reals

- Alguns llenguatges permeten forçar una coerció explícita.

  - Llenguatges de la família de C (sentència *Cast*)
  - En Java és possible transformar:
    - una variable primitiva d'un tipus bàsic a un altre
    - un objecte d'una classe a una superclase

Polimorfisme. Coerció

Example de Coerció en Java

Conversió implícita en Java:

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

Conversió 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)
```

Polimorfisme. Genericitat

Polimorfisme Universal: Genericitat

- **Genericitat/Paramètric**: la definició d'una funció o la declaració d'una classe presenta una estructura que és comuna a un nombre potencialment infinit de tipus

  - En Haskell podem definir i usar tipus genèrics i funcions genèriques
  - En Java podem definir i usar classes genèriques i mètodes genèrics

Polimorfisme. Genericitat

Example de Genericitat en Haskell

Usant **un tipus genèric** (amb variables de tipus), podem definir una estructura de dades per a representar i manipular les entrades (de qualsevol tipus) d'un *diccionari*:

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

getKey :: Entry k v -> k
getKey (x,y) = x

getValue :: Entry k v -> v
getValue (x,y) = y
```

Amb una **funció genèrica** podem calcular la longitud d'una llista els elements de la qual són de qualsevol tipus:

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

Polimorfisme. Genericitat

Example de Genericitat en Java (1/2)

Podem usar una classe genèrica (amb paràmetres) per a definir una entrada d'un *diccionari*:

```
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;
  }
}
```

Podem definir un **mètode genèric** per a calcular la longitud d'un array de *qualsevol* tipus:

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

Polimorfisme. Genericitat

Example de Genericitat en Java (2/2)

Exemple d'ús d'entrades d'un diccionari:


```
Entry<Integer,String> elem1 = new Entry<Integer,String>;

elem1(3,"Programming");
System.out.println(elem1.getValue());
```

Exemple d'ús del mètode genèric per a la longitud d'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));
```

Polimorfisme. Genericitat

Algunes consideracions de la genericitat Java

- Una classe genèrica és una classe convencional, llevat que dins de la seua declaració utilitza una **variable de tipus** (paràmetre), que serà definit quan siga utilitzat.
- Dins d'una classe genèrica es poden utilitzar altres classes genèriques
- Una classe genèrica pot tenir diversos paràmetres

Polimorfisme. Inclusió

Polimorfisme Universal: Inclusió/Herència

- **Inclusió o Herència**: la definició d'una funció treballa sobre tipus que estan relacionats seguint una jerarquia d'inclusió.
- En l'orientació a objectes l'herència és el mecanisme més utilitzat per a permetre la **reutilització** i **extensibilitat**.

L'herència organitza les classes en una estructura jeràrquica formant ***jerarquies de classes***

Polimorfisme. Inclusió

Polimorfisme Universal: Inclusió/Herència

IDEA:

Una classe B heretarà d'una classe A quan volem que B tinga l'estructura i comportaments de la classe A. A més podrem

- afegir nous atributs a B
- afegir nous mètodes a B

I depenent del llenguatge podrem

- redefinir mètodes heretats
- heretar de diverses classes (en Java solament podem heretar d'una classe)

Polimorfisme. Inclusió

Example de Herència en Java (1/2)

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

Polimorfisme. Inclusió

Example de Herència en Java (2/2)

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

- Les subclasses es defineixen usant la paraula clau **extends**
- Es poden afegir atributs, mètodes i redefinir mètodes

Polimorfisme. Inclusió

Example de Herència en Java (2/2)

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

constructor de la classe pare

- Les subclasses es defineixen usant la paraula clau **extends**
- Es poden afegir atributs, mètodes i redefinir mètodes

Polimorfisme. Inclusió

Algunes consideracions de la herència Java (1/2)

- En Java, la base de qualsevol jerarquia és la classe `Object`.
- Si una classe es declara com `final`, no es pot heretar d'ella
- Java solament té herència simple
- A una variable de la superclase se li pot assignar una referència a qualsevol subclasse derivada d'aquesta superclase, però **no** al contrari.

Exemple d'assignació vàlida

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

Polimorfisme. Inclusió

Algunes consideracions de la herència en Java (2/2)

- En Java s'usen qualificadors davant dels atributs i mètodes per a establir què variables d'instància i mètodes dels objectes d'una classe són visibles
  - **Private**: cap membre o atribut `private` de la superclase és visible en les subclasses o altres classes.

    Si s'usa per a atributs de classe, hauran de definir-se mètodes que accedisquen a aquests atributs

  - **Protected**: els membres `protected` de la superclasse són visibles en la subclasse, però no visibles per a l'exterior.
  - **Public**: els membres públics de la superclasse segueixen sent públics en la subclasse.

Polimorfisme. Inclusió

Example de redefinició de mètode heretat 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;
  }
}
```

Polimorfisme. Inclusió

Example de redefinició de mètode heretat en Java (2/2)

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

Exemple d'ús:

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

Polimorfisme. Inclusió

Herència en Java: Classes abstractes

- Una classe abstracta és la que es declara com `abstract`
  - Si una classe té un mètode `abstract` és obligatori que la classe siga `abstract`.
  - Per als mètodes declarats `abstract` no es dóna implementació.
  - Una classe abstracta no pot tenir instàncies.
- Totes les subclasses que hereten d'una classe abstracta, si elles no són abstractes hauran de redefinir els mètodes abstractes donant-los una implementació.

Polimorfisme. Inclusió

Example d'ús de classes abstractes en Java (1/2)

```
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();
}
```

Polimorfisme. Inclusió

Example d'ús de classes abstractes 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 Cicle 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; }
}
```

Polimorfisme. Inclusió

Example d'ús de classes abstractes 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 Cicle extends Shape {
  private float radius;
  public Circle(float initX, float initY, float initRadius){
    super(initX,init
    radius = initRad
  }
  public float perime
  public float area(){ return pi*radius*radius; }
}
```

I si volguérem estendre aquest Example amb una forma nova com el triangle, què cal fer?

Questió: Inclusió y genericitat

Donades la següent definició de classes:

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

Compila sense error el següent fragment de codi? Per què?

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

Reflexió

Què és la reflexió

Quan et mires en un espill pots:

- veure el teu reflex i
- reaccionar davant el que veus

- En els llenguatges de programació, la reflexió és **la infraestructura** que, durant la seua execució, permet a un programa:
  - veure la seua pròpia estructura i
  - manipular-se a si mateix
- La reflexió es va introduir amb el llenguatge LISP i està present en alguns llenguatges de script.

Permet, per exemple, definir programes capaços de monitorizar la seua pròpia execució i modificar-se, en temps d'execució, per a adaptar-se dinàmicament a diferents situacions

Reflexió

Llenguatges amb reflexió

- Es caracteritzen perquè les pròpies instruccions del llenguatge són tractades com a valors d'un tipus de dades específic; En els llenguatges sense reflexió es veuen com a simples cadenes de caràcters
- Els llenguatges amb reflexió poden veure's com metallenguatges del propi llenguatge.

> *Es diu metallenguatge a aquell amb el qual podem escriure metaprogramas (programes que manipulen programes com a compiladors, analitzadors, etc.)*

Reflexió

Ha d'usar-se amb cautela

- Malament usada pot afectar
  - al rendimient del sistema ja que sol ser costosa

    Si pot fer-se sense usar reflexió, no la uses
  - a la seguretat ja que pot exposar informació compromesa sobre el codi

    La reflexió trenca l'abstracció, amb reflexió pot accedir-se a atributs i mètodes privats, etc.
- És una característica avançada però no complicada, especialment en llenguatges funcionals, gràcies a la dualitat natural entre dades i programes (homoiconicitat).

Reflexió

La Reflexió en Java

- En Java la reflexió s'usa mitjançant la biblioteca java.lang.reflect

La biblioteca proporciona classes per a representar de forma estructurada informació de les classes, variables, mètodes, etc.

- Es pot inspeccionar classes, interfícies, atributs i mètodes sense conèixer els noms dels mateixos en temps de compilació. Per exemple podem

- obtenir i mostrar durant l'execució del programa el nom de totes les instàncies de la classe que s'han creat en temps d'execució.
- llegir de teclat un `String` amb el qual poder crear un objecte amb aqueix nom, o invocar un mètode amb aqueix nom.

Reflexió

Example d'us de Reflexió en Java

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

public class MyClass {...}

...
Class 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);
}
```

Altres mètodes definits en la classe `Class`:

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

Procedimients i control de flux

Existeixen alguns conceptes relacionats amb el control de flux dels programes i la definició i cridada a procediments.

- **Pas de paràmetres**. Quan es fa una crida a un mètode o funció hi ha un canvi de context que pot fer-se de diferents formes. Veurem les principals.
- **Àmbit de les variables**. És necessari determinar si un objecte o variable és visible en un moment determinat de l'execució i aquest càlcul pot fer-se de forma estàtica o bé de forma dinàmica.

Pas de paràmetres

Un dels mecanisme d'abstracció bàsic és organitzar les tasques d'un programa definint funcions, mètodes o procediments que resolen subtasques. Així, en un moment determinat es pot invocar a aquests procediments.

- CRIDA: $f(e_1, \ldots, e_n)$ amb $e_1, \ldots, e_n$ expressions.
  - en executar-se la crida, el flux de control passarà al cos de la funció f i, una vegada aquest acabe, tornarà al flux des del qual es va fer la crida.
  - $e_1, \ldots, e_n$ són els anomenats paràmetres d'entrada/reals (*actual parameters*)
- DECLARACIÓ: $f(x_1, \ldots, x_n)$ amb $x_1, \ldots, x_n$ variables.
  - $x_1, \ldots, x_n$ són els anomenats paràmetres formals (*formal parameters*)
  - Els paràmetres formals són variables locals al cos de la funció declarada

Pas de paràmetres

Es distingeixen tres tipus de pas de paràmetres

- Pas per valor (*call by value*)
- Pas per referència (*call by reference/call by address*)
- Pas per necessitat (*call by need*)

Existeixen més modalitats de pas de paràmetres però aquestes tres són les més freqüents en els llenguatges de programació

Pas de paràmetres

Pas per valor

Es calculen els valors v_i dels paràmetres d'entrada e_i en la crida i es copien en els paràmetres formals x_i

- en el cos de la funció es treballa sobre una referència a memòria diferent.

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

Pas de paràmetres

Pas per valor

Es calculen els valors v_i dels paràmetres d'entrada e_i en la crida i es copien en els paràmetres formals x_i

- en el cos de la funció es treballa sobre una referència a memòria diferent.

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

Pas de paràmetres

Pas per valor

Es calculen els valors v_i dels paràmetres d'entrada e_i en la crida i es copien en els paràmetres formals x_i

- en el cos de la funció es treballa sobre una referència a memòria diferent.

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

En la crida:

Es copia el valor `10` en el paràmetre formal `v`

Pas de paràmetres

Pas per valor

Es calculen els valors v_i dels paràmetres d'entrada e_i en la crida i es copien en els paràmetres formals x_i

- en el cos de la funció es treballa sobre una referència a memòria diferent.

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

Pas de paràmetres

Pas per valor

Es calculen els valors v_i dels paràmetres d'entrada e_i en la crida i es copien en els paràmetres formals x_i

- en el cos de la funció es treballa sobre una referència a memòria diferent.

```
void inc(int v)
{
      v = v + v;
}                                   v = 20
...
int a = 10;
inc(a);                             a = 10
```

Pas de paràmetres

Pas per valor

Es calculen els valors v_i dels paràmetres d'entrada e_i en la crida i es copien en els paràmetres formals x_i

- en el cos de la funció es treballa sobre una referència a memòria diferent.

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

- La variable `a` NO es modifica perquè es treballa amb una còpia en la funció `inc`.

Pas de paràmetres

Pas per referència

Es passa la referència a memòria, per la qual cosa el cos de la funció treballa sobre el mateix objecte en memòria

- Per als paràmetres d'entrada e_i que **no** siguen una variable, funciona com el pas per valor
- Quan e_i és una variable (i.g., y_i), les assignacions realitzades sobre el paràmetre formal x_i alteren també el valor associat a y_i

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

Pas de paràmetres

Pas per referència

Es passa la referència a memòria, per la qual cosa el cos de la funció treballa sobre el mateix objecte en memòria

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

Pas de paràmetres

Pas per referència

Es passa la referència a memòria, per la qual cosa el cos de la funció treballa sobre el mateix objecte en memòria

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

En la crida:

El paràmetre formal `v` rep l'adreça de memòria de `a`

Pas de paràmetres

Pas per referència

Es passa la referència a memòria, per la qual cosa el cos de la funció treballa sobre el mateix objecte en memòria

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

Pas de paràmetres

Pas per referència

Es passa la referència a memòria, per la qual cosa el cos de la funció treballa sobre el mateix objecte en memòria

```
void inc(int v)
{
      v = v + v;
}                                          v = 20
...
int a = 10;
inc(a);                                    a = 20
```

Pas de paràmetres

Pas per referència

Es passa la referència a memòria, per la qual cosa el cos de la funció treballa sobre el mateix objecte en memòria

```
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.

Pas de paràmetres

Pas per necessitat

- Quan es passen expressions, no s'avaluen fins que s'usen en el cos de la funció

```
void inc(int v)
{
      v = v + v;
}
...
int a = 10;
inc(a+1);
```

Pas de paràmetres

Pas per necessitat

- Quan es passen expressions, no s'avaluen fins que s'usen en el cos de la funció

```
void inc(int v)
{
      v = v + v;
}
...
int a = 10;                                    a = 10
inc(a+1);
```

Pas de paràmetres

Pas per necessitat

- Quan es passen expressions, no s'avaluen fins que s'usen en el cos de la funció

```
void inc(int v)
{
      v = v + v;
}
...
int a = 10;
inc(a+1);                                    a = 10
```

En la crida:

S'usa una referència a una còpia local d'expressió sense avaluar, és a dir `v=a+1`.

Pas de paràmetres

Pas per necessitat

- Quan es passen expressions, no s'avaluen fins que s'usen en el cos de la funció

```
void inc(int v)
{                                          v = a + 1
      v = (a+1) + (a+1);
}
...
int a = 10;
inc(a+1);                                  a = 10
```

Durant l'execució:

Si necessita l'expressió, l'avalua.

Pas de paràmetres

Pas per necessitat

- Quan es passen expressions, no s'avaluen fins que s'usen en el cos de la funció

```
void inc(int v)
{
      v = (a+1) + (a+1);
}                                          v = 22
...
int a = 10;
inc(a+1);                                  a = 10
```

Durant l'execució:

Si necessita l'expressió, l'avalua.

Pas de paràmetres

Pas per necessitat

- Quan es passen expressions, no s'avaluen fins que s'usen en el cos de la funció

```
void inc(int v)
{
      v = (a+1) + (a+1);
}
...
int a = 10;
inc(a+1);                                    a = 10
```

Pas de paràmetres

Pas per necessitat

- Quan es passen expressions, no s'avaluen fins que s'usen en el cos de la funció

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

- La variable `a` NO es modifica, ja que s'ha usat una còpia local.
- Normalment, gràcies a la *memoization*, `a+1` solament s'avalua una vegada.

Pas de paràmetres

Algunes consideracions

- En pas per valor, si es passa una expressió, aquesta s'avalua per a copiar el valor resultant (a diferència del pas per necessitat)
- En pas per referència, si es passa una expressió també s'avalua i es passa el valor resultant.

Abast (àmbit) de les variables

- Una variable és un *nom* que s'utilitza per a accedir a una posició de memòria
- No tots els noms (de variables, funcions, constants, etc.) estan accessibles durant tota l'execució, encara que existisquen en el programa
- L'àmbit o abast d'un nom és la porció del codi on aquest nom és visible (el seu valor associat pot ser consultat/modificat).
- El moment en el qual es fa l'enllaç (l'associació) és el que es diu temps d'enllaçat.
  - Amb abast estàtic, es defineix en *temps de compilació*
  - Amb abast dinàmic, es defineix en *temps d'execució*

Abast de les variables

Example de càlcul del àmbit de les variables (1/2)

```
1  program ambits;
2  type
3    TArray : array [1..3]
4             of integer;
5  var
6    a: TArray;
7  procedure un;
8    procedure dos;
9      a : TArray;
10   begin {* dos *}
11     a[1] := 1;
12     a[2] := 2;
13     a[3] := 3;
14     canvia(1, 2);
15     writeln(a[1],' ',a[2],' ',
                        a[3]);
16   end {* dos *};
17   procedure canvia(i,j:integer);
18   var aux : integer;
19   begin {* canvia *}
20     aux := a[i];
21     a[i] := a[j];
22     a[j] := aux;
23   end {* canvia *};
24 begin {* un *}
25   a[1] := 0;
26   a[2] := 0;
27   a[3] := 0;
28   dos;
29 end {* un *};
30 begin {* ambits *}
31   u;
32 end {* ambits *}
```

Abast de les variables

Example de càlcul del àmbit de les variables (1/2)

```
1  program ambits;
2  type
3    TArray : array [1..3]
4             of integer;
5  var
6    a: TArray;
7  procedure un;
8    procedure dos;
9      a : TArray;
10   begin {* dos *}
11     a[1] := 1;
12     a[2] := 2;
13     a[3] := 3;
14     canvia(1, 2);
15     writeln(a[1],' ',a[2],' ',
                        a[3]);
16   end {* dos *};
17   procedure canvia(i,j:integer)
18   var aux : integer;
19   begin {* canvia *}
20     aux := a[i];
21     a[i] := a[j];
22     a[j] := aux;
23   end {* canvia *};
24 begin {* un *}
25   a[1] := 0;
26   a[2] := 0;
27   a[3] := 0;
28   dos;
29 end {* un *};
30 begin {* ambits *}
31   u;
32 end {* ambits *}
```

Quins són els valors que emmagatzema l'array `a` al final de l'execució? Què s'imprimeix per pantalla en la sentència `writeln` del codi?

Abast de les variables

Example de càlcul del àmbit de les variables (2/2)

Considerant abast estàtic...

Quins són els valors que emmagatzema el array `a` al final de l'execució? Què s'imprimeix per pantalla en la sentència `writeln` del codi?

En **tiemps de compilació** l'enllaç és:

- En el cos de la funció `un` (línies 25 a 27), la variable `a` està enllaçada amb la variable global de la línia 6 (`un` no té declaració de variables locals).
- En el cos de la funció `dos` (línies 11 a 15), la variable `a` està enllaçada amb la variable local `a` definida en la línia 9, ja que les variables locals amb el mateix nom que les globals oculten a aquestes últimes.
- En el cos de la funció `canvia` (línies 20 a 22), la variable `a` està enllaçada amb la variable global de la línia 6 (el procediment `canvia` està definit en l'àmbit de `un`, igual que `dos`).

Abast de les variables

Example de càlcul del àmbit de les variables (2/2)

Considerant abast estàtic...

Quins són els valors que emmagatzema el array `a` al final de l'execució? Què s'imprimeix per pantalla en la sentència `writeln` del codi?

Por lo tanto:

- En el cuerpo principal del programa (línea 31) se hace una llamada al procedimiento `uno` .
- Els valors del array global s'inicialitzen als valors 0, 0 i 0 (línies 25 a 27)
- Es diu a `dos`, que inicialitza un array local amb valors 1, 2 i 3 el que no modifica el array global (línies 11 a 13)
- La crida a `canvia` canvia els valors del array global, quedant 0, 0 i 0, la qual cosa no modifica el array local de `dos`.
- S'imprimeix per pantalla els valors del array local (1, 2 i 3)

Abast de les variables

Considerant abast dinàmic...

¿Quins són els valors que emmagatzema el array `a` al final de l'execució?
¿Què s'imprimeix per pantalla en la sentència `writeln` del codi?

En **tiempo de ejecución** l'enllaç és:

- En el cos de la funció `un` (línies 25 a 27), la variable `a` està enllaçada amb la variable global de la línia 6 (`un` no té declaració de variables locals).
- En el cos de la funció `dos` (línies 11 a 15), la variable `a` està enllaçada amb la variable local `a` definida en la línia 9.
- En el cos de la funció `canvia` (línies 20 a 22), com la crida a aquesta funció ocorre en l'àmbit de `dos` i `dos` té una variable local `a`, la variable `a` del cos de `canvia` s'enllaçada amb la variable local de la línia 9.

Abast de les variables

Considerant abast dinàmic...

¿Quins són els valors que emmagatzema el array `a` al final de l'execució?
¿Què s'imprimeix per pantalla en la sentència `writeln` del codi?

Por lo tanto:

- En el cos principal del programa (línia 31) es fa una trucada al procediment `un` .
- Els valors del array global s'inicialitzen als valors 0, 0 i 0 (línies 25 a 27)
- Es crida a `dos`, que inicialitza un array local amb valors 1, 2 i 3, la qual cosa no modifica el array global (línies 11 a 13)
- La crida a `canvia` canvia els valors del array local, quedant 2, 1 i 3, la qual cosa no modifica el array global.
- S'imprimeix per pantalla els valors del array local (2, 1 y 3).

Gestió de Memòria

Es refereix als diferents mètodes i operacions que s'encarreguen d'obtenir la màxima utilitat de la memòria, organitzant els processos i programes que s'executen en el sistema operatiu de manera tal que s'optimitze l'espai disponible.

Influeix en les decisions de disseny d'un llenguatge

A voltes els llenguatges contenen característiques o restriccions que solament poden explicar-se pel desig dels dissenyadors d'usar una tècnica o una altra de gestió de memòria

Gestió de Memòria

Necessitat de gestionar la memòria

Elements amb requeriments d'emmagatzematge durant l'execució dels programes:

- Codi del programa traduït
- Informació temporal durant l'avaluació d'expressions i en el pas de paràmetres (e.g., en les crides a funcions els valors actuals tenen que avaluar-se i emmagatzemar-se fins a completar la llista de paràmetres)
- Crides a subprogrames i operacions de tornada
- *Buffers* per a les operacions d'entrada i eixida
- Operacions de inserció i destrucció d'estructures de dades en l'execució del programa (e.g., `new` en Java o `dispose` en Pascal)
- Operacions de inserció is borrat de componentes en estructures de dades (e.g., la funció *push* de Perl per a afegir un element a un array)

Gestió de Memòria

Tipus de gestió de memòria

Tipus d'assignació de l'emmagatzematge

Estàtic

Es calcula i assigna en temps de compilació

- Eficient però incompatible amb recursión o estructures de dades dinàmiques

Dinàmic

Es calcula i assigna en temps d'execució

- en pila
- en *un heap*

Gestió de Memòria

Emmagatzematge estàtic

Emmagatzematge calculat en temps de compilació que roman fix durant l'execució del programa.
Se sol usar amb:

- variables globals
- programa compilat (instruccions en llenguatge màquina)
- variables locals a un subprograma
- constants numèriques i cadenes de caràcters
- taules produïdes pels compiladors i usades per a operacions d'ajuda en temps d'execució (e.g., comprovació dinàmica de tipus, depuració, . . .).

Gestió de Memòria

Emmagatzematge estàtic

Emmagatzematge calculat en temps de compilació que roman fix durant l'execució del programa.
Se sol usar amb:

- variables globals
- programa compilat (instruccions en llenguatge màquina)
- variables locals a un subprograma
- constants numèriques i cadenes de caràcters
- taules produïdes pels compiladors i usades per a operacions d'ajuda en temps d'execució (e.g., comprovació dinàmica de tipus, depuració, . . .).

És eficient però incompatible amb recursión i amb estructures de dades, la grandària de les quals depèn de dades d'entrada o dades computades durant l'execució del programa

Gestió de Memòria

Emmagatzematg dinàmic: en pila

És la tècnica més simple per a manejar els registres d'activació en les trucades a funcions/procedimients durant l'execució del programa (hi ha prou amb un punter al cim de la pila)

Emmagatzematge en pila

- a l'inici de l'execució s'assigna un bloc seqüencial en memòria com a espai d'emmagatzematge lliure,
- quan es requereix espai d'emmagatzematge (hi ha una crida), aquest es pren del bloc començant des del final de l'últim espai assignat (secuencialment)
- una vegada acabada la crida, l'espai s'allibera en ordre invers al que va ser assignat, per la qual cosa l'espai lliure sempre està en el cim de la pila

Gestió de Memòria

Emmagatzematg dinàmic: en *heap*

Un heap és una regió d'emmagatzematge en la qual els blocs de memòria s'assignen i alliberen en *moments arbitraris*

- L'emmagatzematge en *heap* és necessari quan el llenguatge permet estructures de dades (e.g., conjunts o llistes) la grandària de les quals pot canviar en temps d'execució.
- Els subblocs assignats poden ser del mateix grandària sempre o de grandària variable
- La desasignació pot ser
  - explícita (ex. C, C++, Pascal)
  - implícita (quan l'element assignat ja no és assolible per cap variable del programa)

Gestió de Memòria

Emmagatzematg dinàmic: en *heap*

Un heap és una regió d'emmagatzematge en la qual els blocs de memòria s'assignen i alliberen en *moments arbitraris*

- L'emmagatzematge en *heap* és necessari quan el llenguatge permet estructures de dades (e.g., conjunts o llistes) la grandària de les quals pot canviar en temps d'execució.
- Els subblocs assignats poden ser del mateix grandària sempre o de grandària variable
- La desasignació pot ser
  - explícita (ex. C, C++, Pascal)
  - implícita (quan l'element assignat ja no és assolible per cap variable del programa)
- *Garbage collector*: mecanisme del llenguatge que identifica els elements inassolibles i desasigna la memòria que ocupen, la qual passa a estar lliure