Tecnología y Paradigmas de la programación (TPP): José Quiroga Álvarez

80% attendance for labs

Theory (40%)
- Parcial tipotest (10%)
- Final (30%) ≥4

Lab (60%)
- First part (30%) ≥3
- Second part (30%) ≥3

≥5

Get min of 3 values (x,y,z):

```
if X<Y && X<Z
  ret X
ret Y-Z<0 ? Y : Z
```

```
f min(p1,p2){
  ret p1-p2<0 ? p1,p2
}
result = min(a, min(b,c))
```

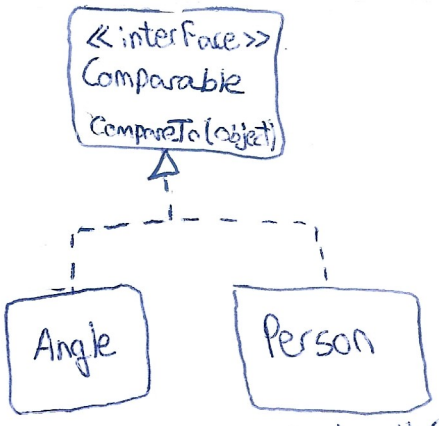
Topic 1: Characteristics of a programing languages

Impesative vs Declarative

↓ Cómo tiene que hacer

↓ Qué tiene que devolves

Simple Sort Algorithm Ex:




```
void Sort(Comparable[] vector){
  For(int i=0; i<vector.length-1; i++)
    For(int j=vector.length-1; j>i; j--)
      if(vector[i].compareTo(vector[j])){
        int temp = vector[i];
        vector[i] = vector[j];
        vector[j] = temp;
      }
}
```

Maintainable code

- Polymorphism: Usar una interfaz (sin código) y cuando llamas al método con dynamic binding se llama al código de la clase del objeto (runtime)
- Inheritance: Extends traspasa todos los métodos del padre a los hijos (Ahorra código)

Delegate Types:

Func<T1, T2> Method that returns T2

Action<T1, ...> Method that returns void

Predicate<T1 ...> Method that returns a boolean

delegate (Person p) { return p.Age >= 18; }

=

(Person P) => { return p.Age >= 18; }

Lambda Expressions:

$F(x,y) = x^2 - 2y + 1$

=

$(\lambda x.\lambda y.\ x^2 - 2y + 1)$

$F(2,4) = \ldots$

=

$(\lambda x.\lambda y.\ x^2 - 2y + 1)\ (2)\ (4)$ — β-reduction

=

$(\lambda y.\ x^2 - 2y + 1\ [x := 2])\ (4)$

α conversion: Used to avoid different bound variables to have the same name before function application

Ex: Double = $(\lambda x.\ x + x)$

ApplyTwice: $(\lambda F.\lambda x.\ F(Fx))$

Compute twice double of n: $(\lambda F.\lambda x.\ F(Fx))(\lambda x.\ x+x)\ (N)$

$(\lambda x.(\lambda y.\ y+y)((\lambda y.\ y+y)\ x))\ (N)$

$(\lambda y.\ y+y)\ ((\lambda y.\ y+y)\ N)$

$((\lambda y.\ y+y)\ N + (\lambda y.\ y+y)\ N)$

$N+N + N+N$

$4N$

Tecnología y Paradigmas de la Programación (TPP):

- Translator / compiler / interpreter: T = translates programs from one language to another language
  C = translator, translates from high-level to low-level machine code
  I = Executes a program in a given programing language.

- Language Features: + Abstraction { • High-level (python/java/...) • Medium-level (C) • Low-level (assembly/machine code) }
  Human (High) ↓ Computer (Low)
  Scripting Language: To automate the execution of tasks by joining components
  Dynamic Language: You can create programs only using it
  + Domain { • General-purpose (C++, python, java) • DSL (Domain Specific Language) (SQL, R, Csound) }
  + Implementation (can be both) { • Compiled { • AOT (Ahead of Time) → Prior to the execution • JIT (Just in Time) → before its execution } • Interpreted }
  + Concurrency { • Single processor • In parallel }
  + Type Cheking { • Static (by the compiler) • Dynamic (at runtime) }
  + Source code represented { • Visually (scratch) • Textual (C#) }

- Paradigms: - Imperative (How to do)
  - Declarative (what to recieve)
  + Structural Procedural: Like following a recipe with if, while and functions.
  + Object Oriented: Abstracts programs as objects (data + methods) { • Class-based • Prototype-based }
  + Functional: Uses pure functions with inmutable data (lamba calculus)
  + Logic: Rules + axioms (Facts) + queries
  + Aspect Oriented: Modularizing (Spliting) the program (persistance, security, logging, ...)
  + Constraint Solving: Constraints (rules) and let the program find a solution
  + Real-Time: Guaranties a response between time constrains { • Hard (if unfulfilling time constrain crash the program) • Soft (" " recovery mechanism used) }
  + Event-Driven: Flow of the program determined by events (mouse, key, sensor)
  + Automata: Follows steps in different states.
  + Reactive Programing: Program reacts to changes in data (checks for updates)

• Obj. Oriented Paradigm:

+ Encapsulation: Hide details and controlling access by an interface
+ Properties (in C#): Access the abstract state of objects (can be read only / read and write)
+ Modularity: Partitioning a program into smaller subprograms
+ Coupling and Cohesion: * Coupling: How much modules depend on each other (less better)
  * Cohesion: How much parts inside a module connect (more better)
+ Method overloading: Creating several methods with the same name. (Different, param types, type, number)
+ Operator Overloading: Allows operators with different meanings depending on the param type.
+ Inheritance: The childs inherit the methods and fields of the father (code reutilization)
+ Polymorphism: Generalization mechanism to have several forms in a thing (a car is also a Vehicle)
+ Dynamic Binding: Specialization mechanism to decide at runtime which method should be called.
+ Multiple Inheritance: Allows inheriting from more than 1 super-class { • Name conflict • Repeated inheritance }
+ Exceptions: Objects holding relevant information about an exceptional situation
+ Assertions: Must be true in the correct execution of a program
+ Preconditions / Postconditions / Invariants
+ Generics: Allows writing methods / functions / ... for several different types (T)
+ Bounded Generics: Generics where T: SomeInterface, specifies more T
+ Type Inference: Automatic deduction (by compiler) of the type of an expression. (Using Var)

• Lamda Calculus: * Function definition (abstraction)
* Its application (invocation)

Smallest Universal Programming Language
(Turing Complete)

$\lambda x.\ M$
- variable
- expression

- Abstraction to lambda function:
$F(x) = x \rightarrow \lambda x.x$
$g(x) = x+x \rightarrow \lambda y.\ y+y$

- Bound / Free Variables: $\lambda x.y.x$
  - y → Free variable
  - x → Bounded Variable

+ Application (β-reduction)

$(\lambda x.M)\ N \xrightarrow{\beta} M[x:=N]$ or $M[N/x]$ (x is substituted by N)

- N → parameter
- x → free

+ α-conversions (Like change of name)

$\lambda x.x \xrightarrow{\alpha} \lambda y.y$

$\lambda x.\ x+y \xrightarrow{\alpha} \lambda z.\ z+y$

+ Halting-Problem

$(\lambda x.xx)(\lambda x.xx) \xRightarrow{\beta} (\lambda x.xx)(\lambda x.xx)$

+ Curry-Howard:

Logic	Type
⊃ (→)	Function
∧	Tuple (Product type)
∨	Union (Sum type)
true	Object
False	Void
∀	Generics

† You can store / return / pass a Function using delegates:
- Delegates can store 1 or more static / instance methods.
- High Order Function: Recieves a Function as a parameter

† Generic Delegate Types:
* Func <..., ..., T_R> → Method with (...) params and return type T_R
* Action [<...>] (optional) → Method that always returns void with or without params
* Predicate <T, ...> → Method that always returns bool with param types (T, ...)

† Anonimous delegates:

```
delegate (ParamType person) { return person.age >= 18; }
```

+ Transformation to lambda expresion:

```
(paramType p) => p.age >= 18;
```