09 - Polymorphism

Greek word that mean "having several forms". In this context, a "form" is a type (primitive/user defined).

The polymorphic elements are:

- Functions (operators, methods): with the same name can take as parameters many types;
- Types (parametric data types, type constructors, generics...): ;

Classification of Polymorphism

There are 2 main categories of polymorphism. Mind that they are not mutual exclusive: we can have both at the same time.

The full schema is the following ($\rightarrow$ means that points to an already existing node of the tree):

- Ad hoc:
  - Overloading;
  - Overriding;
- Universal:
  - Coercion;
  - Inclusion:
    - $\rightarrow$ Overriding;
    - $\rightarrow$ Bounded;
  - Parametric:
    - Implicit;
    - Explicit:
      - Bounded:
        - Covariant;
        - Invariant;
        - Contravariant;

Ad Hoc Polymorphism

With ad hoc polymorphism the same function name denotes different algorithms (different code). The code to execute is determined by the actual types.

Overloading

Concept present in all languages, at least built in for arithmetic operators such as `+`, `*`, ...

- E.g. Java: `+` is the sum for numbers but also the concatenation of strings;

A language can support it for:

- Functions (Java, C++);
- Primitive operators (C++, Haskell);

The code to execute is determined by the type of the arguments, thus:

- Early binding in statically typed languages;

- Dynamic Binding in dynamically typed languages;

Example

Let's say that we want to implement: $f(x) = x^2$

C Language

```language-c
// no support for overloading:
// different names for the same logic
int intSqr(int x){
        return x * x;
}
double doubleSqr(double x){
        return x * x
}
```

Java, C++

```language-java
// overloading: the implementation to
// execute is decided based on the type of pars
int sqr(int x){
        return x * x;
}
double sqr(double x){
        return x * x
}
```

Haskell
Haskell Polymorphism > Type Classes

Overriding

A method `m()` of a class `A` can be redefined in a subclass `B` of `A`

Example:

```language-java
A a = new B(); //legal
a.m(); // the overridden method in B is invoked
```

Universal Polymorphism

With universal polymorphism there is only one algorithm: a single (universal) solution that is applied to different objects. The call of the algorithm is type independent: it is the algorithm that can handle different types of arguments.

Coercion

It is the automatic conversion of an object to a different type. It is opposed to casting, which is explicit.

It is usually done when the conversion cannot provoke any loss of information (no harm is done). Otherwise, casting is required, as the compiler can't take the liberty of destroying potentially useful info.

Example:

```language-c
int x = 5;
double dy = 3.14;
```

```
// coercion, "implicit casting", no info loss
double dx = x;

//casting, potentially dangerous info loss
int y = (int) dy;
```

Coercion can be used for polymorphism but it is a degenerate and uninteresting case:

```language-c
double sqrt(double x){...}

// The parameter is an int, which is coerced to double
double d = sqrt(5);
```

Inclusion

Inclusion polymorphism is also known as subtyping polymorphism, or just *inheritance*.

The polymorphism is ensured by the substitution principle: an object of a subtype (subclass) can be used in any context where an object of the supertype (superclass) is expected. That is because the subclass has *at least* all the fields and methods of the superclass

Java and C++ uses the substitution principle for classes: methods/functions with formal parameter of type T accept an actual parameter of type $S <: T$ (S subtype of T).

WARNING: even if we inherit a method it doesn't mean that we are always in this case of polymorphism, as we could override it.

Parametric

In Parametric polymorphism a function/type can be generic, operating on values of different types.

It can be:

- Implicit: the type of a polymorphic function/type is inferred based on its use. E.g.:

```language-haskell
-- The `length` function is polymorphic, as it can operate on lists of any type.
let length xs =
        case xs of
                [] -> 0
                x:xs' -> 1 + length xs'

-- Usage
length [1,2,3]         -- inferred type: Int
length ["foo", "bar"]  -- inferred type: Int
length ['a', 'b', 'c'] -- inferred type: Int
```

- Explicit: the type of a polymorphic function/type is given. E.g.:

```language-java
// Class `Box` is parameterized with a type variable `T`, which can be any type.
public class Box<T> {
  private T contents;

  public void set(T contents) {
    this.contents = contents;
  }

  public T get() {
```

```
    return contents;
  }
}

// Usage
Box<String> stringBox = new Box<>();
Box<Integer> intBox = new Box<>();
```

An example of **implicit** parametric polymorphism is Type Inference in Haskell.

Instead, 2 examples of **explicit** polymorphism are:

- C++ Templates;
- Java Generics.