PROG-3

SUBTOPIC 5

POLYMORPHISM

Cristina Cachero, Pedro J. Ponce de León

English version by Juan Antonio Pérez

Version 20111024

Department of Computer Languages and Systems

Universitat d'Alacant
Universidad de Alicante

Subtopic 5
Objectives

- Fully understanding the concept of polymorphism.
- Knowing and applying the different types of polymorphism.
- Understanding the concept of static and dynamic binding in OO languages.
- Understanding the relation between polymorphism and inheritance in strongly typed languages.
- Understanding how polymorphism contributes to more extensible and maintainable systems.

Contents

1. Motivation and definitions
   - Signatures
   - Scopes
   - Type system
1. Polymorphism and reuse
   - Definition
   - Forms of polymorphism
1. Overloading
   - Overloading based on scopes
   - Overloading based on type signatures
   - Alternatives to overloading
1. Polymorphism in the context of inheritance relationships
   - Redefinition
   - Shadowing
   - Overriding
1. Polymorphic variable
   - The receiver variable
   - Downcasting
   - Pure polymorphism
1. Generics
   - Generic methods
   - Template classes
   - Inheritance in generic classes

1. Motivation

- Main goal of the OOP
  - To mimic the way problems are solved in the real world.
- Polymorphism is how OO languages implement the concept of polysemy:
  - A single word with multiple meanings.
  - Context is used to disambiguate.

Signature

- **Method type signature**:
  - It usually includes the method name, and the number, types and order of its parameters. Return types may be considered to be a part of the method signature as well.
    - Notation: ***<parameters> → <return type>***
    - The name of the method and the class are omitted.
  - Examples

```
double power (double base, int exp)
```

- ***double*** x ***int → double***

```
double distanciaA(Posicion p)
```

- ***Posicion → double***

1. Definitions: **Scope**

- **Name scope**:
  - When applied to a variable identifier, the (textual) portion of a program in which references to the identifier denote the particular variable.
  - Example:

    ```
    double power (double base, int exp)
    ```

    - Variable *base* can only be used inside method power
  - **Active scopes**: multiple scopes may be active simultaneously.
    - The following example shows various scopes:

```
class A {
  private int x,y;
  public void f() {
      // Active scopes:
      // GLOBAL
      // CLASS (instance and class attributes)
      // METHOD (parameters, local variables)
      if (…) {
        String s;
        // LOCAL scope (local var.)
      }
}
```

1. Definitions:
Scope: **namespaces**

- Namespace: abstract container or environment created to hold a logical grouping (classes, methods, objects…) of unique identifiers or symbols (i.e., names).
- An identifier defined in a namespace is associated only with that namespace.
    - Java: packages

Circulo.java

```
package Graficos;

class Circulo {…}
```

Rectangulo.java

```
package Graficos;

class Rectangulo {…}
```

1. Definitions
Scope: **namespaces**

- C++: `namespace`

Graficos.h
(grouped declarations)

```
namespace Graficos {
  class Circulo {…};
  class Rectangulo {…};
  class Lienzo {…};
  …
}
```

Circulo.h
(each class in its own .h)

```
namespace Graficos {
  class Circulo {…};
}
```

Rectangulo.h

```
namespace Graficos {
  class Rectangulo {…};
}
```

1. Definitions
Scope: **namespaces**

- Java: **`import instruction`**

```
class Main {
  public static void main(String args[]) {
    Graficos.Circulo c;
    c.pintar(System.out);
  }
}
```

```
import Graficos.*;
class Main {
  public static void main(String args[]) {
    Circulo c;
    c.pintar(System.out);
  }
}
```

1. Definitions
Scope: **namespaces**

- C++: **using clause**

```
#include “Graficos.h”
int main() {
  Graficos::Circulo c;
  Graficos::Rectangulo r;
  c.setRadio(4);
  double a = r.getArea();
}
```

```
#include “Graficos.h”
using Graficos::Circulo;
int main() {
  Circulo c;
  Graficos::Rectangulo r;
  c.setRadio(4);
  double a = r.getArea();
}
```

1. Definitions
Scope: **namespaces**

- C++: **`using namespace`** `clause`

```
#include "Graficos.h"
int main() {
  Graficos::Circulo c;
  Graficos::Rectangulo r;
  c.setRadio(4);
  double a = r.getArea();
}
```

```
#include "Graficos.h"
using namespace Graficos;
int main() {
  Circulo c;
  Rectangulo r;
  c.setRadio(4);
  double a = r.getArea();
}
```

1. Definitions:
Type system

- A type system associates a type with each computed value. By examining the flow of these values, a type system attempts to ensure or prove that no type errors can occur. For that, a type system provides:
  - Mechanisms for defining data types and assigning types to variables and expressions.

```
class A {} // type definition in Java/C++
A objeto; // objeto's type is A
```

  - A set of rules for determining type equivalence or compatibility.

```
String s = “una cadena”;
int a = 10;
long b = 100;
a = s; // ERROR in Java/C++, types ‘String’ and ‘int’ are not
  compatible
b = a; // OK in Java/C++
```

1. Definitions:
Type system

- The process of verifying and enforcing the constraints of types (type checking) may occur either at compile-time (a static check) or run-time (a dynamic check):

  - **Early or static typing**
    - Typing is performed at compile-time. Variables always have an associated type.

```
String s; // (Java/C++) ‘s’ is defined as a string.
```

  - **Late or dynamic typing**
    - Typing is performed at run-time. Types are assigned to values, not to variables.

```
my $a; //(Perl) ‘a’ is a variable
$a = 1; //  ‘a’ is an integer…
$a = “POO”; // … an now a string
```

1. Definitions: **Type system**

- According to the rules for type compatibility, type systems can be:
  - Strong typing:
    - Implicit conversion rules are very strict:
      ```
      int a=1;
      bool b=true;
      a=b; // ERROR
      ```
  - Weak typing:
    - Language allows for most implicit conversions among types.
      ```
      int a=1;
      bool b=true;
      a=b; // OK
      ```

Note: ‘strong’ and ‘weak’ are relative terms: we will usually focus on whether a particular language has a stronger/weaker type system than another.

1. Definitions: **Type system**

- The type system of a language determines how it supports dynamic binding:

  - **Procedural languages**: they usually use static and strong type systems, and usually do not support dynamic binding: the type of every expression is known at compile time.
    - C, Fortran, BASIC

  - **Object oriented languages**:
    - Static typing (C++, Java, C#, Objective-C, Pascal…)
      - They only support dynamic binding inside the type hierarchy a expression (identifier or code fragment) belongs to.

    - Dynamic typing (Javascript, PHP, Python, Ruby,…)
      - They (obviously) support dynamic binding.

Contents

1. Motivation and definitions
   - Signatures
   - Scopes
   - Type system
1. Polymorphism and reuse
   - Definition
   - Forms of polymorphism
1. Overloading
   - Overloading based on scopes
   - Overloading based on type signatures
   - Alternatives to overloading
1. Polymorphism in the context of inheritance relationships
   - Redefinition
   - Shadowing
   - Overriding
1. Polymorphic variable
   - The receiver variable
   - Downcasting
   - Pure polymorphism
1. Generics
   - Generic methods
   - Template classes
   - Inheritance in generic classes

2. Polymorphism
Definition

- At heart, the term means there is one name and many different meanings. But names are used for a variety of purposes and meanings can be defined in a number of different ways.

  - We will study four different forms of polymorphism; each of them allows for a different way of **software reuse**:

    - Overloading
    - Overriding
    - Polymorphic variable
    - Generics

Forms of polymorphism

- **Overloading** (Ad hoc polymorphism)
  - A single method name has several alternative implementations.
  - Typically overloaded method names are distinguished at compile time based on their type signatures.

```
Factura.imprimir()
Factura.imprimir(int numCopias)
ListaCompra.imprimir()
```

- **Overriding** (Inclusion polymorphism)
  - A special case of overloading (but with identical signatures) which occurs within the context of the parent/child class relationship and dynamic binding.
  - Methods defined in base classes are refined or replaced in the derived classes.

Forms of polymorphism

- **Polymorphic variables** (Assignment polymorphism)
  - A variable is declared as one type and holds a value of a different type.

```
Figura2D fig = new Circulo();
```

- **Generics** (templates)
  - A generic function or class is parametrized by a type.
  - By leaving the type unspecified, to be filled in later, a generic allows the function or class to be used in a wider range of situations.

```
Lista<Cliente> clientes;
Lista<Articulo> articulos;
Lista<Alumno> alumnos;
```

Contents

1. Motivation and definitions
   - Signatures
   - Scopes
   - Type system
1. Polymorphism and reuse
   - Definition
   - Forms of polymorphism
1. Overloading
   - Overloading based on scopes
   - Overloading based on type signatures
   - Alternatives to overloading
1. Polymorphism in the context of inheritance relationships
   - Redefinition
   - Shadowing
   - Overriding
1. Polymorphic variable
   - The receiver variable
   - Downcasting
   - Pure polymorphism
1. Generics
   - Generic methods
   - Template classes
   - Inheritance in generic classes

3. Overloading, ad hoc polymorphism

- A single message name is associated to multiple implementations.
- Overloading is performed (unlike overriding) at **compile time** (early binding) depending on the type signature of the message used by the call.

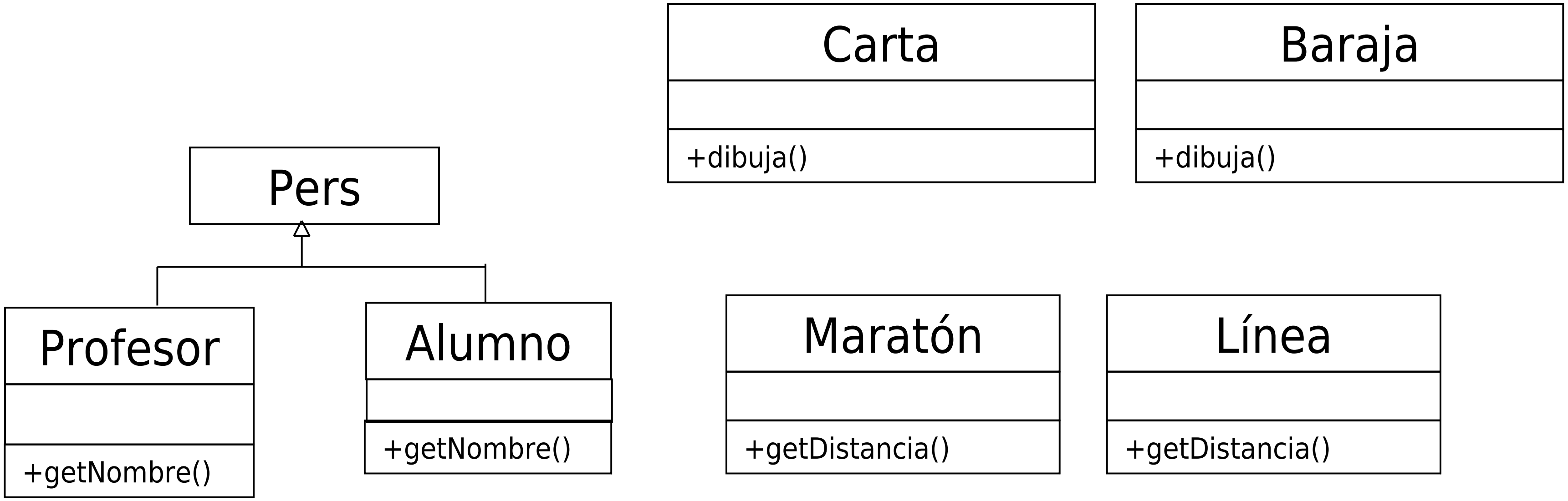
- Two broad categories of overloading:

  - **Scope-based**: methods having **different scopes**, regardless of their type signatures.
    - E.g. toString() method in Java.

  - **Signature-based**: methods having **different signatures** visible in the same scope

Overloading based on **scopes**

- The same name can appear in different scopes with no ambiguity.
- Type signatures can be the same.



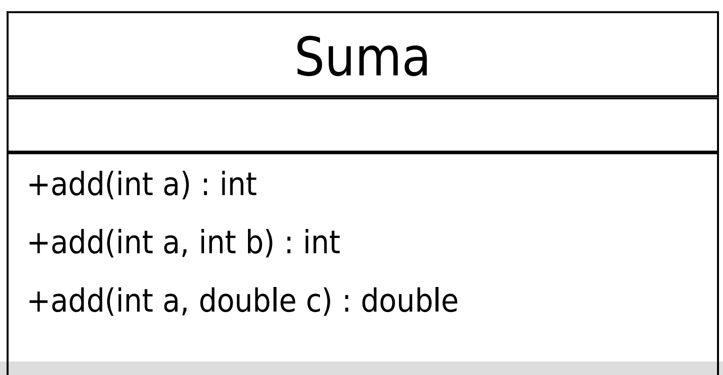
- Do contents of Profesor and Alumno belong to the same scope?
- What about Pers and Profesor?

Overloading based on **type signatures**

- Methods in the same scope are allowed to share a name and are disambiguated by the <u>number</u>, <u>order</u>, and, in statically typed languages, the <u>type of arguments</u> they require (return type is not considered).

  - C++ and Java permit this form of overloading with the only requirement that the selection of the routine intended by the user can be unambiguously determined at **<u>compile time</u>**.
    - Therefore, methods are not permitted to differ only in their return types.

```
int f() {}
string f() {}
System.out.println( f() ); // ???
```



Overloading based on **type signatures**

- Homework: if overloading based on type signatures is considered and methods have static binding, what happens when two types are different but related by means of inheritance?

```
class Base{…}
class Derivada extends Base {…}
class Cliente {
   public static void Test (Base b)
     {System.out.println("Base");}
  public static void Test (Derivada d) // overloading
     {System.out.println("Derivada");}
   public static void main(String args[]){
     Base obj;
     if (...) obj = new Base();
     else obj = new Derivada(); // substit.
     Test (obj); // which method is invoked?
 }
```

Overloading based on **type signatures**

- **Some languages do not permit overloading** (or at least some forms of it):
  - C++ supports method and operator overloading.
  - Java, Python, and Perl support only method overloading.
  - Eiffel supports only operator overloading.

Operator overloading in C++

- **Operator overloading** is a special form of overloading based on type signatures.

- It is claimed to be useful because it allows the developer to program using notation "closer to the target domain".

- Overloading operator @ in C++:

  `<return type>` **`operator@`**`(<args>)`

- Operators must be overloaded whenever we want to use them with our own types.

  - Operators defined by default: assignment operator (=) and reference operator (&)

Operator overloading in C++

- Operator overloading prohibits changing
  - <u>*Precedence*</u> (which operator must be evaluated first)
  - <u>*Associativity*</u> a=b=c → a=(b=c)
  - <u>*Arity*</u> (unary, binary...)

- New operators cannot be created.

- Operators for predefined types cannot be overloaded.

- Some operators are not overloadable: "**.**", "**.***", "**::**", **sizeof, "? :"**

Operator overloading in C++

- Operator overloading may be done by using member or non-member methods.

  - <u>Member functions</u>: left operand (in a binary operator) must be an object (or a reference to an object) of the class.

    - Example: overloading + in class Complejo:

```
Complejo Complejo::operator+(const Complejo&)
…
Complejo c1(1,2), c2(2,-1);
c1+c2; // c1.operator+(c2);
c1+c2+c3; // c1.operator+(c2).operator+(c3)
```

Operator overloading in C++

- <u>Non-member functions</u>:

  - Useful when the left operand does not belong to the class.

Example: overloading operators << and >> in class Complejo:

```
ostream& operator<<(ostream&, const Complejo&);
istream& operator>>(istream&, Complejo&);
…
Complejo c;
cout << c; // operator<<(cout,c)
cin >> c; //  operator>>(cin,c)
```

Alternatives to overloading: Functions with a variable number of arguments

- **Polyadic functions**
  - Functions with a variable number of arguments
  - Supported by many languages:
    - E.g., printf in C and C++

- If the maximum number of parameters is known, C++ allows to declare optional parameters with default values:

```
int sum (int e1, int e2, int e3=0, int e4=0);
```

Alternatives to overloading: Functions with a variable number of arguments

- **Polyadic methods in Java**

```
void f(Object... args)
{  for (Object obj : args) {…} }
// args is treated as an array

f(“A”, new A(), new Float(10.0));
f();

void g(int a, int... resto) {...}

g(3,”A”,”B”); g(3);
```

- Polyadicity makes overloading more complex. It must be used carefully.

Coercion and conversion

- **COERCION**
  - A value of one type is IMPLICITLY converted into one of a different type by the compiler.
    - E.g., implicit coercion between reals and integers in C++/Java.

```
double f(double x) {...}
f(3); // coercion from int to real
```

  - In OOL, the principle of substitutability introduces a variant of coercion not available in conventional languages.
    - `// substit. principle (coercion between pointers)`

```
class B extends A {…}
B pb = new B();
A pa = pb;
```

Alternatives to overloading: Coercion and conversion

- **CONVERSION**
  - Explicit change of type, usually referred as **cast**.
  - Conversion operators:
    - Example:

```
double  x; int i;
x= i + x; // COERCION
x= (double)i + x; // CONVERSION
```

    - Java supports:
      - Conversion between scalar types (except for booleans)
      - Between types related through inheritance (upcasting, downcasting)
      - Additional conversions with specific methods:
        - `Integer.valueOf("15.4");`

Alternatives to overloading: Coercion and conversion

- **CONVERSION under C++**
  - Conversion operators (cast) in C++:
    - From an external type to the type defined by the class:
      - Constructor that takes a single argument (of the external type).
    - From the type defined by the class to a different type:
      - Implementation of a conversion operator.
      - ```
        class Fraccion{
            private: int num, den;
            public : operator double() {
              return (numerador()/(double)denominador());
            }
        };
        Fraccion f; double d = f * 3.14;
        ```
        ```

Contents

1. Motivation and definitions
   - Signatures
   - Scopes
   - Type system
1. Polymorphism and reuse
   - Definition
   - Forms of polymorphism
1. Overloading
   - Overloading based on scopes
   - Overloading based on type signatures
   - Alternatives to overloading
1. Polymorphism in the context of inheritance relationships
   - Redefinition
   - Shadowing
   - Overriding
1. Polymorphic variable
   - The receiver variable
   - Downcasting
   - Pure polymorphism
1. Generics
   - Generic methods
   - Template classes
   - Inheritance in generic classes

Overloading in the context of inheritance

- DEFAULT BINDING TIME
  - JAVA
    - <u>Dynamic binding</u> for public and protected instance methods.
    - <u>Static binding</u> for private and class methods, and attributes.
  - C++
    - <u>Static binding</u> for all the properties (instance methods, class methods, attributes).

Overloading in the context of inheritance

- Changing default binding time
  - JAVA
    - Instance methods with static binding: not supported
      - Actually, a method declared as `final` in the root of an inheritance hierarchy behaves as if static binding was being used (it prohibits overriding).
        - `public final void doIt() {…}`
  - C++
    - Instance methods with dynamic binding:
      - `virtual void doIt();`

Overloading in the context of inheritance

- **Shadowing**: The type signatures of two methods in the parent and child classes are the same, and static binding is used:
  - Refinement/replacement in the derived class. The method to be invoked is chosen at compile time.
- **Redefinition**: The type signature in the child class differs from that given in the parent class (although the names are the same), and static binding is used.
- **Overriding**: see later.

Overloading in the context of inheritance

- Two approaches to **redefinition** in OOL:
  - ***Merge*** model (Java):
    - Method implementations found in all currently active scopes are merged into a single collection and the closest match from this list is executed.
  - ***Hierarchical*** model (C++):
    - Each currently active scope is examined in turn to find the matching method. A redefinition in the child class hides other definitions in the base class:

```
class Padre{
  public void ejemplo(int a){System.out.println("Padre");}
}
class Hija extends Padre{
  public void ejemplo (int a, int b){System.out.println("Hija";}
}


  Hija h;
  h.ejemplo(3); // OK in Java
                // but compiler error in C++
  h.Padre::ejemplo(3); // OK (C++)
```

Overloading in the context of inheritance
Overriding

- The name and the type signature are the same in both parent and child classes, and dynamic binding is used.
    - The method in the base class presents dynamic binding (declared as virtual in C++, for instance).
    - The (overridden) method in the child class may replace or refine the method in the parent class.
    - The method to be invoked is chosen at run time depending on the dynamic type of the receiver of the message.

Overriding

- Reimplementation in derived classes in Java always implies overriding as instance methods present dynamic binding by default.
  - In spite of this, the (optional) annotation *@Override* may be put in the child class to explicitly indicate that a method in the base class will be overridden. If a method marked with @Override fails to correctly override a method in one of its superclasses, the compiler generates an error.
- The reserved word '`final`' prohibits that a method is overridden.

```
class Base { public void f() {} }
class Derivada {
  @Override
  // Compiler error if 'void f()' is private or final, or
  // if it does not exist in the base class
  public void f() {}
}
```

Overriding

- Java:

```
class Padre {
  public int ejemplo(int a)
         {System.out.println("padre");}
  public final void f() {}
}
class Hija extends Padre {
  @Override // optional annotation (recommended)
  public int ejemplo (int a)
         {System.out.println("hija");}
  // public void f() { … }
 // ERROR, f() cannot be overridden
}

// client code
Padre p = new Hija(); // substitut.
p.ejemplo(10); // Hija.ejemplo(10) is executed
```

Overriding: covariance

- Covariant return types: the method in the child class changes the return type to a subtype of the type used in the parent class:

```
class A {…}
class B extends A {...}
class Base {
  A objA = new A();
  public A getA() { return objA; } }
class Derivada {
  B objB = new B();
  @Override
  public B getA() { return objB; } }

Base b = new Derivada();
A objetoA = b.getA(); // Upcasting.
// objetoA will point to b.objB
```

Overloading in the context of inheritance
Overriding

More languages:

C++: the method must be declared as virtual in the parent class (allowing for dynamic binding)

Smalltalk: similar to Java.

Object Pascal: the derived class must indicate that a method is being overridden: `procedure setAncho(Ancho: single);` **`override`**`;`

C#, Delphi Pascal: overriding must be indicated both in the parent and the child class. In C#:

Base class: `public` **`virtual`** `double Area() {...}`

Derived class: `public` **`override`** `double Area() {...}`

Summary

- It is important to distinguish among **overriding**, **shadowing**, and **redefinition**:
  - **Overriding**: the type signatures are the same in both parent and child classes, and dynamic binding is used to determine the method to be executed.
  - **Shadowing**: the type signatures are the same in both parent and child classes, and static binding is used to determine the method to be executed.
  - **Redefinition**: The type signature in the child class differs from that given in the parent class, but the same name is used for the two methods.