PROG-3

SUBTOPIC 6 – PART 2

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

Contents

The polymorphic variable (assignment polymorphism)

- A polymorphic variable is a variable that can reference more than one type of object.
  - It can maintain values of different types during the course of execution.
- In a dynamically typed language all variables are potentially polymorphic.
- In a statically typed language the polymorphic variable is the embodiment of the principle of substitution.
  - Java: object references.
  - C++: pointers or references to polymorphic classes

The polymorphic variable

Polymorphic classes

- In Java, all classes are polymorphic by default.
- In C++, they need to include at least one virtual method.

  - Java uses the reserved word `final` to indicate that no derived classes can be created:
    - **`final`** `class ClaseNoDerivable { … }`

  This implies that references of type *ClaseNoDerivable* are no longer polymorphic, as they can only reference objects of type *ClaseNoDerivable*.

The polymorphic variable

- **Simple polymorphic variables**
  - `Figura2D img;` // Reference to the polymorphic base class; it will point
    // actually to objects of the derived classes
    // (Circulos, Cuadrados,...)

- **Receiver variables: `this` *and* `super`**
  - In a method, they reference the receiver of the message.
  - In each class, they represent an object of a different type.

(this pseudo-variable goes by a number of different names in different languages, such as 'self' or 'this')

The polymorphic variable

- **Downcasting** (reverse polymorphism):
  - The polymorphic variable, while declared as a parent class, actually holds a value derived from a child class (conversion).
  - In a sense, it means undoing the process of substitution.
  - Forms
    - **Static** (compile-time)
    - **Dynamic** (run-time)

  C++ supports both

  In Java downcasting is always dynamic

The polymorphic variable

- **Dynamic downcasting**
  - The feasibility of the conversion is checked at run time
  - In Java, it is only allowed in the context of inheritance hierarchies
  - When downcasting is not possible, exception *ClassCastException is thrown*

```
class Base {
  public void f() {}
}

class Derivada extends Base {
  public void f() {}
  public void g() {}
}

// Unsafe downcasting
Base[] x = { new Base(), new Derivada() };
Derivada y = (Derivada)x[1]; // Downcasting OK
y = (Derivada)x[0]; // ClassCastException thrown
y.g();
```

The polymorphic variable

- **Safe downcasting and RTTI**

  - **RTTI: Run Time Type Information**
    - Capability of some OO languages to determine the type of an object at runtime
    - Specifically, it allows to determine subtypes from references to the base type: <u>safe downcasting</u>
  - Notice that upcasting is always safe
  - “Traditional” RTTI assumes that you have all the types available at compile time and run time; on the other hand, the “reflection” mechanism allows you to discover class information solely at run time.

The polymorphic variable

- **RTTI: Class class and objects**
  - Class: metaclass whose instances represent classes
  - Each class has its own associated object of type Class in memory

```
class MiClase {}

MiClase c = new MiClase();
Class clase = MiClase.class; // class literal
clase = c.getClass(); // idem
```

Class literal: a second way to produce the reference to the Class object

The polymorphic variable

- **RTTI: Class objects**

```
class Animal {}
class Perro extends Animal {
  public void ladrar() {}
}
class PastorBelga extends Perro {}


// Safe downcasting
Animal a = new PastorBelga();
if (a.getClass() == PastorBelga.class) // true
{  PastorBelga pb = (PastorBelga)a; }
if (a.getClass() == Perro.class) // false
{  Perro p = (Perro)a; }
```

The polymorphic variable

- **RTTI: instanceof**
  - The keyword **instanceof** tells you if an object is an instance of a particular type

```
class Animal {}
class Perro extends Animal {
  public void ladrar() {}
}
class PastorBelga extends Perro {}

// Safe downcasting
Animal a = new PastorBelga();
if (a instanceof PastorBelga) // true
{  PastorBelga pb = (PastorBelga)a; }
if (a instanceof Perro) // true
{  Perro p = (Perro)a; }
```

The polymorphic variable

- **RTTI:**
  - **Class.isInstance(): dynamic `instanceof`**
    - `instanceof` requieres that the name of the target class is known at compile time
    - Use Class.isInstance if it is unknown

```
// Safe downcasting
Animal a = new Perro();
Animal b = new PastorBelga();

Class clasePerro = a.getClass();
if (clasePerro.isInstance(b)) { // true
  Perro p = (Perro)b; // safe
  p.ladrar();
}
```

- **Pure polymorphism** or **polymorphic method**:

    - One function can be used with a variety of arguments, because some of the parameters are polymorphic variables

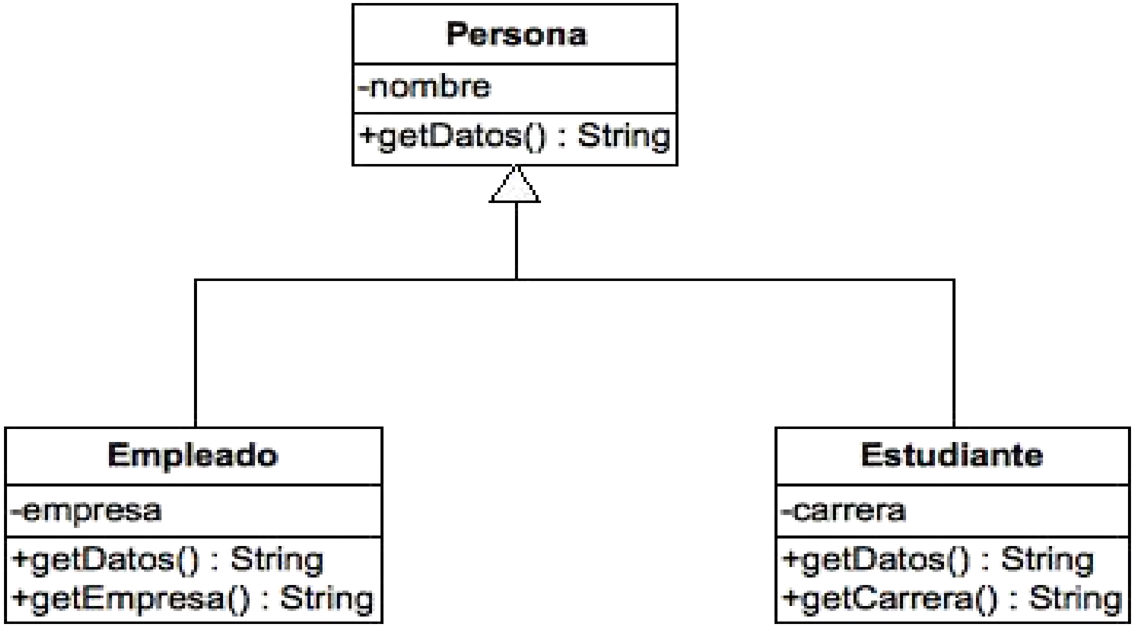
- Example of pure polymorphism:

```
class Base { ... }
class Derivada1 extends Base { … }
class Derivada2 extends Base { … }

void f(Base obj) { // Polymorphic method
  // Here, only the interface of Base can be used to handle obj,
  // but obj can be of type Base, Derivada1, Derivada2,…
}

public static void main(String args[]) {
  Derivada1 objeto = new Derivada1();
  f(objeto); // OK
}
```

Example: polymorphism and type hierarchies



Example: polymorphism and type hierarchies

```
class Persona {
  public Persona(String n)      {nombre=n;}
  public String getDatos() {return nombre;}
   ...
  private String nombre;
}
```

Example: polymorphism and type hierarchies

```
class Empleado extends Persona {
  public Empleado(String n,String e)
  {  super(n); empresa=e }
  public String getDatos()
  { return super.getDatos()+“trabaja en " + empresa; }
        ...
  private String empresa;
}
class Estudiante extends Persona {
  public Estudiante(String n,String c)
  { super(n); carrera=c }
  public String getDatos()
  { return super.getDatos() + " estudia " + carrera; }
        ...
  private String carrera;
}
```

Refinement

Refinement

Example: polymorphism and type hierarchies

```
// client code

    Empleado empleado =
      new Empleado(“Carlos”,”Lavandería”);
    Persona pers =
      new Persona(“Juan”);

    empleado = pers;
    System.out.println( empleado.getDatos() );
```

What happens with this code?

Example: polymorphism and type hierarchies

```
// client code

  Empleado emp = new Empleado("Carlos", "lavanderia");
  Estudiante est = new Estudiante("Juan",”Derecho");
  Persona pers;
  pers = emp;
  System.out.println( emp.getDatos() );
  System.out.println( est.getDatos() );
  System.out.println( pers.getDatos() );
```

What is the output of this program?
Is static or dynamic binding being used here?

Example: polymorphism and type hierarchies

```
class Persona {
  public Persona(String n)      {nombre=n;}
     public final String getDatos() {
      return (nombre);
    }
   ...
  private String nombre;
}
```

Avoids overriding

Example: polymorphism and type hierarchies

```
// client code

  Empleado emp = new Empleado("Carlos", "lavanderia");
  Estudiante est = new Estudiante("Juan",”Derecho");
  Persona pers;
  pers = emp;
  System.out.println( emp.getDatos() );
  System.out.println( est.getDatos() );
  System.out.println( pers.getDatos() );
```

What is the output of this program?
Is static or dynamic binding being used here?

Example: polymorphism and type hierarchies

```java
// client code 1

   Empleado uno= new Empleado("Carlos", "lavanderia");
   Persona desc = uno;
   System.out.println( desc.getEmpresa() );

// client code 2
   Persona desc = new Persona("Carlos");
   Empleado emp = (Empleado)desc;
   System.out.println( emp.getEmpresa() );
```

What happens in both cases?

Example: polymorphism and type hierarchies

```
// client code 3
   Persona desc = new Persona("Carlos");
   if (desc instanceof Empleado) {
     Empleado emp = (Empleado)desc;
     System.out.println( emp.getEmpresa() );
   }
```

What happens here?
Is static or dynamic binding being used here?

Polymorphism
Internal implementation in Java

- Methods with dynamic binding are less efficient that normal methods.
  - Every non-abstract class in Java contains a vector of method pointers called the *method table*. Each pointer corresponds to an instance method with dynamic binding and points to the most convenient implementation (the one in the class or, in case it does not exist, the one defined in the nearest ancestor class).
  - Every object in Java contains a hidden reference to the method table.

Polymorphism
Advantages

- Polymorphism allows for adding new classes to an existing hierarchy without the need to modify or even recompile the code already implemented in terms of the base class.

- Polymorphism allows programmers to write code at the base class level which could use objects of derived (possibly not implemented yet) classes: see libraries/frameworks in a later topic

Contents

Generics
Motivation

- **Genericity** is another form of polymorphism
- To illustrate the main ideas behind generics, let's consider the following example:
  - Implementation of a single *maximo* function with different parameter types.

- Solution: use interfaces

```
class Comparable { boolean mayorQue(Object); }
class A implements Comparable { … }
class B implements Comparable { … }
```

```
Comparable maximo(Comparable a, Comparable b) {
    if (a.mayorQue(b))
        return a;
    else
        return b;
}

A a1 = new A(), a2 = new A();
B b1 = new B(), b2 = new B();
A mayorA = maximo (a1,a2);
B mayorB = maximo (b1,b2);
```

Generics
Motivation

- Maximo() is restricted to classes implementing interface Comparable.
- If a more general mechanism is required, a generic function will be needed; for instance, to implement a class Lista which could contain any type of data (primitive or object)

Generics
Definition

- *Generics provide a way to define a (parameterized) class or a function without giving the type of every member or parameter*
  - Its main purpose is to group objects in collections when the type of these objects is not explicitly defined (e.g., lists, stacks, queues, etc. of generic objects: Java Collection Framework).
  - To create an instance of a generic class, the programmer associates a type to the parameter of the generic.
  - C++ supports generics (templates) since the middle of the 80's. Java supports them since version 1.5.

Generics in Java:
Generics

- Two forms of generics:
  - **Generic methods**: they are useful to implement functions with arbitrary-type arguments.
  - **Generic classes**: they are useful to implement *container classes.*

A single generic argument

```
public <T> void imprimeDos(T a, T b)
{
  System.out.println(
     "Primero: "   + a.toString() +
     " y Segundo:" + b.toString() );
}
```

```
Cuenta a = new Cuenta(),
Cuenta b = new Cuenta();
imprimeDos(a,b);
```

Argument type inference is performed by the compiler

Generics
Generic methods

More than a generic parameter

```
public <T,U> void imprimeDos(T a, U b)
{
  System.out.println(
     "Primero: "   + a.toString() +
     " y Segundo:" + b.toString() );
}


 Cuenta c = new Cuenta(),
 Perro p = new Perro();
 imprimeDos(c,p);
```

Generics
Generic classes in Java

- Let's study the implementation of a generic class *vector.*

Generics

Example: **template class in C++**

```
class vector<T> { // generic argument: T
    private T v[];
    public vector(int tam)
    { v = new T[tam]; }
    T get(int i)
    { return v[i]; }
}
```

Generics
Using a generic class

- Object creation:

```
vector<int> vi = new vector<int>(10);
vector<Animal> va = new vector<Animal>(30);
```

The name of a type must be indicated when instantiating the template class.

Subtypes of Animal can also be stored in 'va'.

Generics

Example: generic object stack

- Let's implement now a generic stack...

Generics

Example: generic object stack

```
class Pila<T> {
    public Pila(int nelem) { … }
    public     void apilar (T elem) { … }
    public     void imprimir() { … }

    private    T info[];
    private    int cima;
    private    static final int limite=30;
}
```

Generics

Example: generic object stack

```
public Pila(int nelem){
  if (nelem<=limite)
    info=new T[nelementos];
  else
    info=new T[limite];
  cima=0;
}
```

Generics

Example: generic object stack

```
void apilar(T elem) {
    if (cima<info.length)
        info[cima++]=elem;
}

void imprimir() {
    for (int i=0; i < cima; i++ )
       System.out.println(info[i]);
}
```

```
Pila<Cuenta> pCuentas = new Pila<Cuenta>(6);
Cuenta c1 = new Cuenta("Cristina",20000,5);
Cuenta c2 = new Cuenta("Antonio",10000,3);
pCuentas.apilar(c1);
pCuentas.apilar(c2);
pCuentas.imprimir();
```

```
Pila<Animal> panim = new Pila<Animal>(8);
panim.apilar(new Perro());
panim.apilar(new Gato());
panim.imprimir();
```

Homework: implement a generic object list.

Generics

Inheritance with generic classes

- Generic classes can be derived from other generic classes:

Derived generic class:

```
class DoblePila<T> extends Pila<T>
{
  public void apilar2(T a, T b) {...}
}
```

- Class doblePila is also generic:

```
DoblePila<float> dp =  new DoblePila(10);
```

- Obviously, non-generic classes can be derived from a generic class:

Non-generic derived class:

```
class monton extends public Pila<int>
{
  public void amontonar(int a) {...}
}
```

- ‘monton’ is a normal class with no generic parameter.

Inheritance with generic classes

- In Java, no special relationship is established between objects instantiated from the same generic class, even in the case that the types are related through inheritance:

```
class Uno {}
class Dos extends Uno {}


    ArrayList<Uno> u = new ArrayList<Uno>();
    ArrayList<Dos> d = new ArrayList<Dos>();

    u = d; // Error: incompatible types
```

Generics
Type erasure

- However,

```
ArrayList<Integer> v = new ArrayList<Integer>();
ArrayList<String> w = new ArrayList<String>();
System.out.println(
  v.getClass() == w.getClass() );
// prints 'true'
//v = w; // Error: incompatible types
```

Type erasure: Java generics are checked at compile-time for type-correctness. The generic type information is then removed in a process called type erasure. For example, List<Integer> will be converted to the non-generic type List, which can contain arbitrary objects. The compile-time check guarantees that the resulting code is type-correct.

As a result of type erasure, type parameters cannot be determined at run-time.

Generics
Generic interfaces

- Interfaces can be generic as well:

```
interface Factoria<T>
{ T newObject(); }

class FactoriaDeAnimales implements Factoria<Animal>
{

  Animal newObject() {
    if (…) return new Perro();
    else if (…) return new Gato();
    else return new ProgramadorDeJava();
  }
}
```

Generics

Constrained genericity

- *In principle, only those methods defined in Object can be used inside a generic class.*
- *Restricted generics allow to indicate that the generic types belong to a particular point in the class hierarchy*
  - *As a result, the interface of the particular class can be used.*

```
class Perrera<T extends Perro> {
  public acoger(T p) { jaula.add(p); }
  public alimentar() {
    for (T p : jaula)
      if (p.ladrar()) p.alimentar();
  }
  private ArrayList<T> jaula =  new ArrayList<T>;
}
```

Generics
Wildcards

```
public class NonCovariantGenerics {
  // Compile Error: incompatible types:
  List<Fruit> flist = new ArrayList<Apple>();
} ///:~
```

- A list of apples IS NOT a list of fruits. Lists of fruits are not a supertype of lists of apples.
- **Subtype wildcards (upper bound)**

```
  List<? Extends Fruit> flist =
    new ArrayList<Apple>();
  flist.add(new Apple()); // ERROR
  flist.get(0); // returns a fruit
```

- ***Base type wildcars (lower bound):***

```
List<? super Apple> flist =
  New ArrayList<Apple>();
flist.add(new Apple()); // OK
flist.add(new Fruit()); // ERROR
flist.get(0); // returns an Apple
```

More information on Java wildcards: here (and here)

Bibliography

- Cachero et. al.
  - ***Introducción a la programación orientada a Objetos***
    - Ch. 4
- T. Budd.
  - ***An Introduction to Object-oriented Programming, 3rd ed.***
    - Ch. 11,12,14-18; ch. 9: case study in C#
- Scott Meyers
  - ***Effective C++. Third Edition***
    - Ch. 6 y 7: some sections on polymorphism in inheritance contexts and genericity.