Tema 4
Polimorfismo

Programación II

Alicia Garrido Alenda

Departamento de Lenguajes y Sistemas Informáticos
Universidad de Alicante

Introducción

- El polimorfismo es el mecanismo mediante el que los lenguajes orientados a objetos implementan el concepto de **polisemia** del mundo real:
  - Un único nombre para muchos significados en función del contexto en el que se utilice.

Conceptos previos: signatura

- Signatura de un método:
  - Nombre del método y descripción del tipo de sus argumentos, su orden y el tipo devuelto por el método.
  - Notación:

    nombre_método(argumentos) $\rightarrow$ ***tipo_devuelto***

  - Ejemplos:

```
public static double max(double x,double y)
```

max(double,double) $\rightarrow$ *double*

```
public boolean agrega(Tarea nueva)
```

agrega(Tarea) $\rightarrow$ *boolean*

Conceptos previos: ámbitos

- Ámbito de un nombre: porción del programa en la que se puede usar.
- Por ejemplo:

  `public int alimenta(String alimento,String lugar)`

  Las variables `alimento` y `lugar` sólo pueden ser utilizadas dentro del método `alimenta`.
- Ámbitos activos: puede haber varios simultáneamente ya que las clases, los cuerpos de los métodos o cualquier bloque de código define un ámbito.

```
public class Par{
 private double x,y;
 private static int base;
 public double eleva(boolean orden){
  // ambitos activos:
  // CLASE: variables de clase y variables de
       instancia
  // METODO: argumentos, variables locales(decl. en el
       metodo)
  if (...){
    double temporal;
  // LOCAL: variables locales al bloque del if
 }}}
```

Conceptos previos: sistema de tipos (I)

- El sistema de tipos de un lenguaje asocia a cada expresión un tipo para intentar evitar errores en el código. Para ello proporciona:
  - Mecanismos para definir tipos y asociarlos a expresiones.
    ```
    public class Uno{...} //def. tipo Uno en java
    Uno objeto=new Uno(...); //objeto es de tipo Uno
    ```
  - Un conjunto de reglas para determinar la equivalencia o compatibilidad entre tipos.
    ```
    String objeto="una cadena";
    double x=10;
    int i=10.5; // Error en java: perdida de precision
    char caracter='a';
    i=caracter; // Ok en java
    ```


Conceptos previos: sistema de tipos (II)

- En función del mecanismo que asocia tipos y expresiones tenemos:
  - Sistema de tipos estático: el enlace se realiza en tiempo de compilación y las variables siempre tienen asociado un tipo.
    ```
    float dato; //(C) dato se define como un float
    ```
  - Sistema de tipos dinámico: el enlace se realiza en tiempo de ejecución y el tipo se asocia a los valores de las variables, y no a las variables en sí.
    ```
    my $mia; #(Perl) mia es una variable, sin tipo concreto
    $mia="hola\n";print $mia; # escribe en pantalla hola
    $mia=1;  $mia+=2.5;
    print "$mia\n"; # escribe en pantalla 3.5
    ```


Conceptos previos: sistema de tipos (III)

- En función de las reglas de compatibilidad entre tipos tenemos:
  - Sistema de tipos fuerte: las reglas de conversión implícita entre tipos son muy estrictas.
    ```
    int a=1;
    boolean b=true;
    System.out.println(a+b); // Error en java
    ```
  - Sistema de tipos débil: el lenguaje permite la conversión implícita entre tipos.
    ```
    int a=1;
    bool b=true;
    cout<<a+b<<endl; // Ok en C++
    ```
- Los términos fuerte/débil son relativos: un lenguaje puede tener un sistema de tipos más fuerte/débil que otro.

Conceptos previos: sistema de tipos (IV)

- El sistema de tipos de un lenguaje determina su soporte del enlace dinámico:
  - **Lenguajes procedimentales**: habitualmente con sistemas de tipos estáticos, fuertes y en general no soportan enlace dinámico, ya que el tipo de toda expresión se conoce en tiempo de compilación. Por ejemplo C, Basic, ...
  - **Lenguajes orientados a objetos**:
    - **Con sistema de tipos estático**: sólo soportan enlace dinámico dentro de la jerarquía de tipos a la que pertenece una expresión. Por ejemplo C++, Java, C#, ...
    - **Con sistema de tipos dinámico**: soportan enlace dinámico. Por ejemplo Python, Ruby ...

¿Qué es el polimorfismo?

Polimorfismo

Capacidad de una entidad de referenciar distintos elementos en distintos instantes de tiempo.

- Veremos cuatro formas de polimorfismo:
  - Variables polimórficas
  - Sobrecarga
  - Sobreescritura
  - Genericidad

Tipos de polimorfismo: Variables polimórficas

Variables polimórficas *(Polimorfismo de asignación)*

- Lo vimos en el tema de herencia.
- Variable que se declara con un tipo pero que referencia en realidad un valor de un tipo distinto (relacionado mediante herencia).

- Por ejemplo:

```
...
Ciudadano ejemplar=new Guerrero("Balkar",10,35.2,
                        {"boleadoras","bayesta"});
...
```


Tipos de polimorfismo: Sobrecarga

Sobrecarga (*overloading* o *polimorfismo ad hoc*)

- Un sólo nombre de método y muchas implementaciones distintas.
- Los métodos sobrecargados normalmente se distinguen en tiempo de compilación debido a que tienen distintos parámetros.

- Por ejemplo:

```
public class Agenda_Dinamica{
...
public boolean agrega(Tarea nueva){
...
  intro=lista.add(nueva);
...
}
public boolean agrega(Fecha limite,String desc){
...
  nueva=new Tarea(limite,desc);
...
}
... }
```


Tipos de polimorfismo: Sobreescritura

Sobreescritura (*overriding* o *polimorfismo de inclusión*)

- Lo vimos en el tema de herencia.
- Tipo especial de sobrecarga que sucede en las relaciones de herencia en métodos con enlace dinámico.
- Los métodos sobrecargados, definidos en la superclase, son refinados o reemplazados en las subclases.

- Por ejemplo:

```
public class Ciudadano{
...
public int alimenta(String come,String lg){...}
...
}
public class Guerrero extends Ciudadano{
...
public int alimenta(String come,String lg){...}
...
}
```

Tipos de polimorfismo: Genericidad

Genericidad *(Plantillas)*

- Clases o métodos parametrizados (con elementos por definir).
- Forma de crear herramientas de propósito general (clases, métodos) y especializarlas para situaciones específicas.

- Por ejemplo:

```
Agenda<Tarea> pendientes;
Agenda<Cliente> clientela;
Agenda<Persona> conocidos;
```


Variables polimórficas (I)

- Una variable polimórfica es aquella que puede referenciar más de un tipo de objeto.
- Puede tener valores de distintos tipos en distintos momentos de la ejecución del programa.
- En un lenguaje con sistema de tipos dinámico todas las variables son potencialmente polimórficas.
- En un lenguaje con sistema de tipos estático la variable polimórfica es la materialización del principio de sustitución.

Variables polimórficas (II)

- Variable polimórfica simple: cualquier variable declarada de una superclase en una jerarquía de herencia.

```
Padre variable; // variable de tipo Padre que en realidad
                // contendrá objetos de sus subclases
```

- Variable receptora: **this**
  - En un método hace referencia al receptor del mensaje.
  - En cada clase representa un objeto de un tipo distinto.

Variables polimórficas (III)

Run Time Type Information (RTTI)

Información acerca de tipos polimórficos que el compilador genera en el programa y que puede utilizarse durante la ejecución del mismo para identificar el tipo dinámico de una referencia.

- **`instanceof`** (Java)

```
public class Persona{ ...
public void Ocupaciones(double t, String d){...}
}
public class Estudiante extends Persona{...
public void Ocupaciones(double t, String d){...}
}
...
Estudiante e=new Estudiante(...);
Persona pr=e;
if (pr instanceof Estudiante)
{/* sabemos que pr es un estudiante */}
```

Variables polimórficas (III)

Run Time Type Information (RTTI)

Información acerca de tipos polimórficos que el compilador genera en el programa y que puede utilizarse durante la ejecución del mismo para identificar el tipo dinámico de una referencia.

- **`typeid`** (C++)

```
class Persona{ public:
...
virtual void Ocupaciones(double t, string d); };
class Estudiante: public Persona{ public:
...
public void Ocupaciones(double t, string d); };
...
Estudiante* e=new Estudiante(...);
Persona* pr = e;
if (typeid(pr) == typeid(Estudiante))
{/* sabemos que pr apunta a un estudiante */}
```


Variables polimórficas: polimorfismo puro

Método con polimorfismo puro *(método polimórfico)*

- Alguno de sus argumentos es una variable polimórfica.
- Un sólo método puede ser utilizado con un número potencialmente ilimitado de tipos distintos de argumentos.

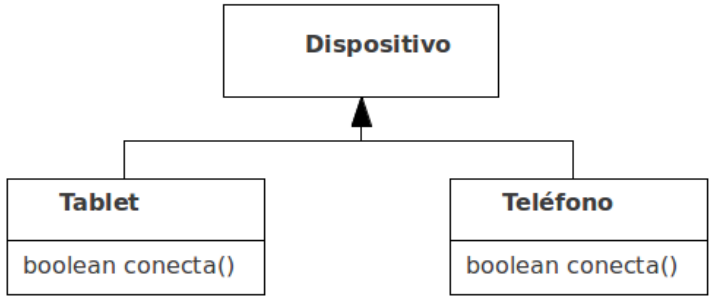
```
public class Persona{...}
public class Estudiante extends Persona{...}
public class Trabajador extends Persona{...}
public class City{
public static void vaAlCine(Persona cinefilo){
 // método polimórfico
 //Solo accesible la interfaz de Persona, pero cinefilo puede
 //ser Persona, Estudiante o Trabajador
}
public static void main(String[] args){
 Persona hbt=new Estudiante(...);
 vaAlCine(hbt);
}
}
```


Sobrecarga (*overloading* o *polimorfismo ad hoc*)

- Un mismo nombre de mensaje está asociado a varias implementaciones.
- La sobrecarga se realiza en **tiempo de compilación** en función de la signatura completa del mensaje.
- Podemos distinguir dos tipos de sobrecarga:
  - **Basada en ámbito**: métodos con diferentes ámbitos de definición, independientemente de su signatura.
  - **Basada en signatura**: métodos con diferentes signaturas en el mismo ámbito.

Sobrecarga basada en ámbito

- Distintos ámbitos implican que el mismo nombre de método puede aparecer en ellos sin ambigüedad.
- La signatura puede ser exactamente igual.
- Por ejemplo:



- Tablet y Teléfono, ¿son ámbitos distintos?
- ¿Y Dispositivo y Tablet?

Sobrecarga basada en signatura

- Métodos en el mismo ámbito pueden compartir el mismo nombre siempre que difieran en número, orden y/o tipo de los argumentos que requieren (el tipo devuelto no se tiene en cuenta).
- Java y C++ permiten esta sobrecarga de manera implícita siempre que la selección del método requerido por el usuario pueda establecerse de manera no ambigua en tiempo de compilación.
- Por ello la signatura no puede distinguirse solo por el tipo de retorno.

```
int f(){...};
String f(){...};
cout <<f(); //??
```


Sobrecarga basada en signatura

- Si usamos sobrecarga basada en la signatura, ¿qué sucede cuando los tipos son diferentes pero relacionados por la herencia?

```
public class Padre{...}
public class Hijo extends Padre{...}
public class Main{
 public static void prueba(Padre p){
   System.out.println("Padre");
   p.muestra();}
 public static void prueba(Hijo h){
   System.out.println("Hijo");
   h.muestra();}
 public static void main(...){
  Padre objeto=null;
  Hijo sub=new Hijo();
  ...
  if (leido==1) objeto=new Padre();
  else objeto=new Hijo();
  prueba(objeto);  // que muestra?
  prueba(sub);
}
}
```


Sobrecarga basada en signatura

- No todos los LOO permiten la sobrecarga:
  - Permite sobrecarga de métodos y operadores: C++, C#
  - Permite sobrecarga de métodos pero no de operadores: Java
  - Permite sobrecarga de operadores pero no de métodos: Eiffel

Alternativas a la sobrecarga: lista variable de argumentos (I)

- Especificar un método que reciba un número variable de parámetros en lugar de especificar el mismo método para cada lista de parámetros.
- Por ejemplo:

```
public class Agenda_Dinamica{
...
public boolean agrega(Tarea nueva){...}
public boolean agrega(Fecha limite,String desc){...}
...
}
```

Alternativas a la sobrecarga: lista variable de argumentos (II)

- Se puede hacer:

```
public class Agenda_Dinamica{
...
public boolean agrega(Object... args){
Fecha f=null;String d=null;
 for(Object obj : args){
  if (obj instanceof Tarea)
   lista.add((Tarea)obj);
  if (obj instanceof Fecha)
   f=(Fecha)obj;
  if (obj instanceof String){
   d=(String)obj;
   Tarea nueva=new Tarea(f,d);
   lista.add(nueva);
  }
 }
}
...
}
```


Alternativas a la sobrecarga: Coerción y conversión (I)

- En determinadas ocasiones la sobrecarga se puede sustituir por una operación semánticamente diferente: la **coerción**.
- Un valor de un tipo se convierte de manera implícita en un valor de otro tipo distinto:
  - Coerción implícita entre caracteres y enteros:

```
boolean vocal(int j){...}
boolean flag=vocal('a'); //coerción de carácter a entero
```


Alternativas a la sobrecarga: Coerción y conversión (II)

- Un valor de un tipo se convierte de manera implícita en un valor de otro tipo distinto:
  - El principio de sustitución en los LOO introduce una forma de coerción que no existe en los lenguajes convencionales:

```
public class Padre{...}
public class Hijo extends Padre{...}
public class Main{
 public static void prueba(Padre p){
   p.muestra();}
 public static void main(...){
  Padre objeto;
  ...
  if (leido==1) objeto=new Padre();
  else objeto=new Hijo();
  prueba(objeto);  //ppio. sustitucion
 }
}
```


Alternativas a la sobrecarga: Coerción y conversión (III)

- Cuando el cambio de tipo es solicitado de manera explícita por el programador se trata de una **conversión**.
- El operador utilizado para realizar este tipo de operación se denomina CAST:
  - Conversión explícita entre enteros y reales:

```
double x=2.3;
int i;
i=(int) x; //conversión de real a entero
```

  - La conversión se utiliza en las jerarquías de herencia para realizar el *downcasting*:

```
class B extends A{...}
A objA=new B();
B objB=(B) objA;
```

Sobrecarga en jerarquías de herencia

Shadowing

Métodos en la superclase y en una subclase con el mismo nombre, la misma signatura y enlace estático.

Redefinición

Métodos en la superclase y en una subclase con el mismo nombre, distinta signatura y enlace estático.

Sobreescritura

Métodos en la superclase y en una subclase con el mismo nombre, la misma signatura y enlace dinámico.

Shadowing

- Las signaturas del método son idénticas en la superclase y en la subclase.
- Implica refinamiento/reemplazo del método en la subclase.
- El método a invocar se decide en tiempo de compilación.
- No existe en Java.

Redefinición

- La subclase define un método con el mismo nombre que uno existente en la superclase pero con distintos parámetros.
- Dos formas de resolverlo en LOO:
  - Modelo ***merge*** (Java): los diferentes significados que se encuentran en todos los ámbitos actualmente activos se unen para formar una sola colección de métodos.
  - Modelo ***jerárquico*** (C++): una redefinición en la subclase oculta el acceso directo a otras definiciones en la superclase.

Redefinición

- Modelo ***merge*** (Java)

```
public class P{
  public void E(int a){
   System.out.println("P-"+a);
  }
}
```

```
class H extends P{
  public void E(int a,int b){
   System.out.println("H-"+(a+b));
  }
}
```

```
public class M{
  public static void main(String[] args){
   H h=new H();
   h.E(3); //Ok
  }
}
```

Redefinición

- Modelo ***jerárquico*** (C++)

```
class P{
  public: P(){};
   void E(int a);};
class H:public P{
  public: H():P(){};
   void E(int a,int b);};
```

```
#include "PH.h"
void P::E(int a){
   cout<<"P-"<<a<<endl;}
void H::E(int a,int b){
   cout<<"H-"<<a+b<<endl;}
```

```
#include "PH.h"
int main(){
   H* h=new H();
   h->E(3); //error compilacion
   h->P::E(3); //Ok
}
```


Sobreescritura

- Las signaturas del método son idénticas en la superclase y en la subclase.
- Implica refinamiento/reemplazo del método en la subclase.
- El método a invocar se decide en tiempo de ejecución en función del tipo dinámico del receptor del mensaje.
- Según el lenguaje:
  - Java: la propia existencia de un método con la misma signatura en la superclase y en una subclase indica sobreescritura excepto cuando el método es de clase *(static)*.
  - C++: es la superclase la que debe indicar explícitamente que un método tiene enlace dinámico y que puede ser sobreescrito con la palabra reservada `virtual`.
  - Object Pascal: la subclase debe indicar que sobreescribe un método con la palabra reservada `override`.
  - C#, Delphi Pascal: es necesario que tanto la superclase como la subclase lo indiquen.

Sobrecarga en jerarquías de herencia: resumen

- Es importante distinguir entre **sobreescritura**, **redefinición** y **shadowing**.
  - Sobreescritura: la signatura para el mensaje es la misma en la superclase y en la subclase, pero el método se enlaza con la llamada en función del tipo real del objeto receptor en tiempo de ejecución.
  - Redefinición: la subclase define un método con el mismo nombre que la superclase pero con distintos parámetros (número o tipos).
  - Shadowing: la signatura para el mensaje es la misma en la superclase y en la subclase, pero el método se enlaza en tiempo de compilación (en función del tipo declarado de la variable receptora).

Polimorfismo: ventajas

- El polimorfismo permite al usuario la posibilidad de añadir nuevas clases a una jerarquía sin modificar o recompilar el código escrito en términos de la superclase.
- Permite programar a nivel de superclase utilizando objetos de subclases (posiblemente aún no definidas): técnica base de las librerías/*frameworks*.

Genericidad

- La genericidad es un tipo de polimorfismo diferente.
- Consiste en la capacidad de definir clases *parametrizadas* con tipos de datos.
- Son útiles para la implementación de tipos de datos contenedores como las colecciones (listas, arrays, ...).
- La genericidad sólo tiene sentido en lenguajes con comprobación estática de tipos, como Java.
- La genericidad permite escribir código reutilizable.

Clase Contenedor

```
public class Contenedor<T> {

 private T objeto;

 public void setObjeto(T obj){
  objeto=obj;
 }

 public T getObjeto(){
  return objeto;
 }

}
```


Genericidad: parametrización

- La parametrización de una clase genérica se realiza en la declaración de una variable de esa clase y en la creación del objeto:

```
Contenedor<String> cS = new Contenedor<String>();
Contenedor<Fecha> cF = new Contenedor<Fecha>();
cS.setObjeto("Hola");
cF.setObjeto(new Fecha(23,04,2003));
```


Operaciones sobre tipos genéricos

- ¿Qué operaciones podemos realizar sobre tipos genéricos?
  - La **asignación** (=) y la **comparación de identidad** (== o !=).
  - Operaciones aplicables sobre cualquier objeto (métodos de la clase `Object`).
- **No es posible** construir objetos de un tipo genérico:

```
T objeto = new T(); //Error compilacion
```

- Es posible realizar más operaciones aplicando **genericidad restringida**.

Clase genérica

- Una clase genérica es una clase normal, salvo que dentro de su declaración utiliza un tipo de variable (parámetro), que se definirá cuando sea utilizada.
- Dentro de una clase genérica se pueden utilizar otras clases genéricas.
- Una clase genérica puede tener varios parámetros.

```
public class ContenedorDoble<T,K>{
 private Contenedor<T> cobjeto;
 private K valor;
 private String nombre;
 public K getValor(){return valor;}...}

ContenedorDoble<Fecha,Integer> cD = new ...;
```


Genericidad y tipos primitivos

- Las clases genéricas no pueden ser parametrizadas a tipos primitivos.
- Para resolver este problema Java tiene las llamadas clases envoltorio de los tipos primitivos:

  `Integer, Double, Character, Boolean, ...`

- El compilador transforma automáticamente tipos primitivos en clases envoltorio y viceversa.

```
Contenedor<Fecha> f=new Contenedor<Fecha>();
f.setObjeto(new Fecha(23,4,2003));
ContenedorDoble<Fecha,Integer> cD =
new ContenedorDoble<Fecha,Integer>(f,8,"Juan");
int entero=cD.getValor();
```


Genericidad restringida (I)

- Objetivo: limitar los tipos a los que puede ser parametrizada una clase genérica.
- Al restringir los tipos podemos invocar métodos de los objetos de tipo genérico.
- Una clase con genericidad restringida sólo puede ser parametrizada con tipos compatibles con el de la restricción (clase o interfaz).

Genericidad restringida: ejemplo

- La clase `Calendario` sólo puede ser parametrizada con tipos compatibles con `Fecha`.
- Podemos invocar los métodos de la clase `Fecha`.

```
import java.util.*;
public class Calendario<T extends Fecha>{
 private ArrayList<T> periodo;
 public Calendario(){periodo=new ArrayList<T>();}
 ...
 public void consulta(){
  for(int i=0;i<periodo.size();i++)
   if(periodo.get(i)!=null)
    periodo.get(i).muestra();
 }
}
public class Fecha{
 ...
 public void muestra(){
  System.out.println("Day: "+dia+
     " Month: "+mes+" Year: "+anyo);
}
}
```

Genericidad restringida (II)

- Una clase genérica puede estar restringida por varios tipos:

```
public class Calendario<T extends Fecha & Descripcion>
```

- De esta forma, las operaciones disponibles para objetos de tipo `T` son las de todos los tipos.
- Al igual que en la herencia sólo el primer tipo puede ser una clase, los demás deben ser interfaces.

Genericidad y herencia

- Una clase puede heredar de una clase genérica.
- La nueva clase puede:
  - Mantener la genericidad de la superclase:

```
public class Agenda<T> extends Calendario<T>
```

  - Restringir la genericidad de la superclase:

```
public class Agenda<T extends Fecha> extends
Calendario<T>
```

  - No ser genérica y especificar un tipo concreto:

```
public class Agenda extends Calendario<Tarea>
```


Genericidad y sistema de tipos

- Si dos clases tienen una relación de herencia, podemos utilizar el principio de sustitución:

```
public class Persona{...}
public class Estudiante extends Persona{...}
...
Estudiante modelo=new Estudiante(...);
Persona hbt=modelo; //Ok
```

- Si una clase genérica se instancia a clases relacionadas por herencia, dichos objetos de la clase genérica no cumplen el principio de sustitución:

```
public class Contenedor<T>{...}
...
Contenedor<Persona> poblacion;
Contenedor<Estudiante> alumnado;
...
poblacion=alumnado; //Error de compilacion
```


Genericidad y sistema de tipos

- Cuando se crea un objeto de una clase genérica y no se especifica el parámetro se asigna el tipo puro *(raw)* que se corresponde con:
  - La clase `Object`: cuando no hay genericidad restringida.
  - La clase a la cual se restringe: cuando hay genericidad restringida.
- Por ejemplo:

```
public class Contenedor<T>{...}
public class Calendario<T extends Fecha>{...}
Contenedor generico=new Contenedor();    // Object
Calendario mensual=new Calendario(lapso);     // Fecha
```

Genericidad y máquina virtual

- La máquina virtual no maneja objetos de tipo genérico.
- En **tiempo de ejecución** se pierde la información sobre el tipo utilizado para parametrizar la clase genérica.
- Con el operador `instanceof` sólo podemos preguntar por el nombre de la clase genérica, no por el nombre de la clase de los objetos que contiene.

```
//error de compilacion
if (poblacion instanceof Contenedor<Persona>) //error
...
if (poblacion instanceof Contenedor){ // Ok
 Object generico=poblacion.getObjeto();
// Podemos comprobar el tipo de objeto
 if (generico instanceof Persona) // de esta manera
...
}
```

lsi Departamento de Lenguajes y Sistemas Informáticos