Indice

Variables Polimórficas (Polimorfismo de asignación)

- Una variable polimórfica es aquélla que puede referenciar más de un tipo de objeto
  - Puede mantener valores de distintos tipos en distintos momentos de 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.
  - En Java: las referencias a objetos.
  - En C++: punteros o referencias a clases polimórficas

Variables Polimórficas.

Clase polimórfica

- En Java, por defecto todas las clases son polimórficas.
- En C++, clase con al menos un método virtual.
  - Podemos indicar que no se pueden crear clases derivadas con final:
    - `final class ClaseNoDerivable { … }`

    El efecto es que las referencias de tipo *ClaseNoDerivable* ya no son polimórficas: sólo pueden referenciar objetos de tipo *ClaseNoDerivable*.

Variables Polimórficas.

- **Variables polimórficas simples**
  - `Figura2D img; // Puntero a clase base polimórfica que en realidad apuntará a`
    `// objetos de clases derivadas (Circulos, Cuadrados,...)`
- **Variables receptoras: `this` y `super`**
  - En un método, hacen referencia al receptor del mensaje.
  - En cada clase representan un objeto de un tipo distinto.

(en otros lenguajes recibe otros nombres, como 'self')

Variables Polimórficas

- **Downcasting** (polimorfismo inverso):
  - Conversión de una referencia a clase base a referencia a clase derivada.
  - Implica 'deshacer' el ppio. de sustitución.
  - Tipos
    - **Estático** (en tiempo de compilación)
    - **Dinámico** (en tiempo de ejecución)

    C++ soporta ambos tipos
    En Java es siempre dinámico.

Variables Polimórficas

- **Downcasting dinámico**
  - Se comprueba en tiempo de ejecución que la conversión es posible
  - En Java, sólo permitido dentro de jerarquías de herencia
  - Si no es posible se lanza *ClassCastException*

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

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

// Downcasting no seguro
Base[] x = { new Base(), new Derivada() };
Derivada y = (Derivada)x[1]; // Downcasting OK
y = (Derivada)x[0]; // ClassCastException thrown
y.g();
```

Variables Polimórficas

- **Downcasting seguro y RTTI**
  - **RTTI: Run Time Type Information**
    - Mecanismo que proporciona información sobre tipos en tiempo de ejecución
    - Permite averiguar y utilizar información acerca de los tipos de los objetos mientras el programa se está ejecutando.
    - En particular, podemos identificar subtipos a partir de referencias al tipo base: downcasting seguro

Variables Polimórficas

- **RTTI: La clase Class**
  - Es una metaclase cuyas instancias representan clases
  - Cada clase tiene asociado un objeto Class

```
class MiClase {}

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

Literal de clase: es el objeto Class que representa a MiClase

Variables Polimórficas

- **RTTI: La clase Class**

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


// Downcasting seguro
Animal a = new PastorBelga();
if (a.getClass() == PastorBelga.class) // cierto
{  PastorBelga pb = (PastorBelga)a; }
if (a.getClass() == Perro.class) // falso
{  Perro p = (Perro)a; }
```

Variables Polimórficas

- **RTTI: instanceof**
  - **Instrucción que devuelve cierto si el objeto referenciado es del tipo indicado**

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

// Downcasting seguro
Animal a = new PastorBelga();
if (a instanceof PastorBelga) // cierto
{  PastorBelga pb = (PastorBelga)a; }
if (a instanceof Perro) // cierto
{  Perro p = (Perro)a; }
```

Variables Polimórficas

- **RTTI:**
  - **Class.isInstance(): `instanceof` dinámico**
    - `instanceof` necesita conocer el nombre de la clase objetivo en tiempo de compilación
    - ¿y si no lo conozco?

```
// Downcasting seguro
Animal a = new Perro();
Animal b = new PastorBelga();

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

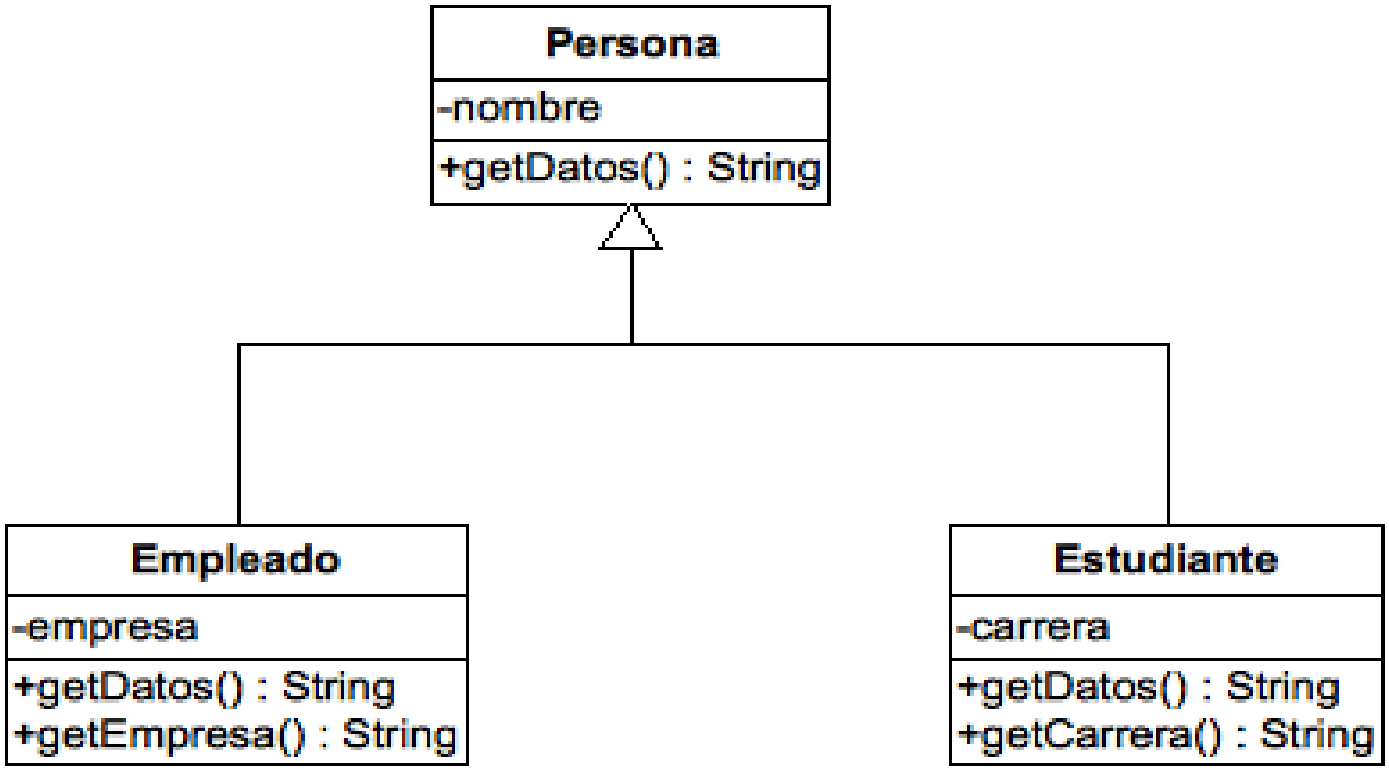
Variables Polimórficas. Polimorfismo puro

- Método con **polimorfismo puro** o **método polimórfico**
  - Alguno de sus argumentos es una variable polimórfica:
    - Un solo método puede ser utilizado con un número potencialmente ilimitado de tipos distintos de argumento.

- Ejemplo de polimorfismo puro

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

void f(Base obj) { // Método polimórfico
  // Aquí puedo usar sólo la interfaz de Base para manipular obj
  // Pero obj puede ser de tipo Base, Derivada1, Derivada2,…
}

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

Ejemplo: Uso de polimorfismo y jerarquía de tipos



Ejemplo: Uso de polimorfismo y jerarquía de tipos

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

Ejemplo: Uso de polimorfismo y jerarquía de tipos

```
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;
}
```

Refinamiento

Refinamiento

Ejemplo: Uso de polimorfismo y jerarquía de tipos

```
// código cliente

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

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

¿Qué ocurre?

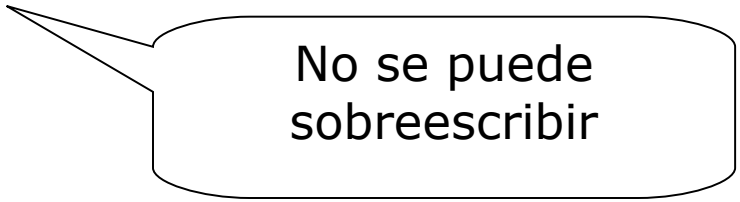
Ejemplo: Uso de polimorfismo y jerarquía de tipos

```
//código cliente

  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() );
```

¿Qué salida dará este programa?
¿Se produce un enlazado estático o dinámico?

Ejemplo: Uso de polimorfismo y jerarquía de tipos

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



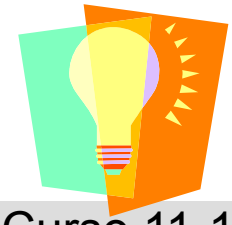
Ejemplo: Uso de polimorfismo y jerarquía de tipos

```
//código cliente

  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() );
```

¿Qué salida dará este programa?
¿Se produce un enlazado estático o dinámico?

Ejemplo: Uso de polimorfismo y jerarquía de tipos

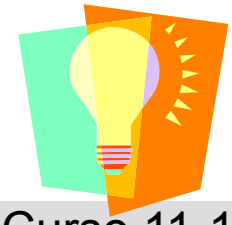
```
// código cliente 1

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

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



¿Qué sucede en ambos casos?

Ejemplo: Uso de polimorfismo y jerarquía de tipos

```
// código cliente 2
    Persona desc = new Persona("Carlos");
    if (desc instanceof Empleado) {
      Empleado emp = (Empleado)desc;
      System.out.println( emp.getEmpresa() );
    }
```



¿Qué sucede en ambos casos?
¿Se produce un enlazado estático o dinámico?

Polimorfismo
Implementación interna en Java

- Los métodos con enlace dinámico son algo menos eficientes que las funciones normales.
  - Cada clase no abstracta en Java dispone de un vector de punteros a métodos llamado *Tabla de métodos*. Cada puntero corresponde a un método de instancia con enlace dinámico, y apunta a su implementación más conveniente (la de la propia clase o, en caso de no existir, la del ancestro más cercano que la tenga definida)
  - Cada objeto de la clase tiene un puntero oculto a esa *tabla de métodos*.

Polimorfismo
Ventajas

- El polimorfismo hace posible que un usuario pueda añadir nuevas clases a una jerarquía sin modificar o recompilar el código escrito en términos de la clase base.

- Permite programar a nivel de clase base utilizando objetos de clases derivadas (posiblemente no definidas aún): Técnica base de las librerías/frameworks (UD 8)

Indice

Genericidad
Motivación

- La genericidad es otro tipo de polimorfismo
- Para ilustrar la idea de la genericidad se propone un ejemplo:
  - Suponed que queremos implementar una función *máximo,* donde los parámetros pueden ser de distinto tipo

Genericidad
Motivación

- Solución: usar 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);
```

Genericidad
Motivación

- Maximo() está restringido a clases que implementen el interface Comparable.
- ¿Y si queremos algo todavía más general, que funcione con cualquier tipo de objeto? Por ejemplo, una clase Lista que pueda contener cualquier tipo de dato, ya sean tipos primitivos u objetos.
  - Necesitaríamos una función genérica

Genericidad
DEFINICION

- *Propiedad que permite definir una clase o una función sin tener que especificar el tipo de todos o algunos de sus miembros o argumentos.*

  - Su utilidad principal es la de agrupar variables cuyo tipo base no está predeterminado (p. ej., listas, colas, pilas etc. de objetos genéricos: Java Collection Framework).

  - Es el usuario el que indica el tipo de la variable cuando crea un objeto de esa clase.

  - En C++ esta característica apareció a finales de los 80. En Java, existe desde la versión 1.5.

Genericidad en Java:
Genéricos

- Dos tipos de genéricos:
  - **Métodos genéricos**: son útiles para implementar funciones que aceptan argumentos de tipo arbitrario.
  - **Clases genéricas**: su utilidad principal consiste en agrupar variables cuyo tipo no está predeterminado (*clases contenedoras*)

Un argumento genérico

```
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);
```

Inferencia de tipo de los argumentos: la realiza el compilador

Más de un argumento genérico

```
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);
```

Genericidad
Clases Genéricas en Java

- A continuación se plantea un ejemplo de una clase genérica *vector,* este vector va a contener elementos de tipo genérico, no se conocen a priori.

Genericidad
Ejemplo **Clase Genérica en C++**

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

Genericidad
Ejemplo de uso de una Clase Genérica

- Creación de objeto:

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

Hay que indicar el tipo de objeto al instanciar la clase.

En el caso de 'va', podemos almacenar ahí cualquier Animal o subtipo de Animal.

Genericidad
Ejemplo: Pila de Objetos Genérica

- El siguiente ejemplo es una pila que contendrá objetos de cualquier tipo, para ello la vamos a definir como una clase genérica. En esta pila podremos introducir nuevos elementos e imprimir el contenido de la misma.

Genericidad
Ejemplo: Pila de Objetos Genérica

```
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;
}
```

Genericidad
Ejemplo: Pila de Objetos Genérica

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

Genericidad
Ejemplo: Pila de Objetos Genérica

```
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]);
}
```

Genericidad
Ejemplo: Pila de Objetos Genérica

```
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();
```

De manera análoga, plantead una lista de objetos genérica.

Genericidad
Herencia en clases genéricas

- Se pueden **derivar clases genéricas** de otras clases genéricas:

  Clase derivada genérica:

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

- La clase doblePila es a su vez genérica:

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

- Se pueden **derivar clases no genéricas** de una genérica:

Clase derivada no genérica:

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

- 'monton' es una clase normal. No tiene ningún parámetro genérico.

Genericidad
Herencia en clases genéricas

- En Java, no existe relación alguna entre dos clases generadas desde la misma clase genérica, aunque los tipos estén relacionados por herencia:

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


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

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

Genericidad
Borrado de tipos

- Sin embargo,

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

Borrado de tipos: Java no guarda información RTTI sobre tipos genéricos. En tiempo de ejecución, sólo podemos asumir que los parámetros genéricos son de tipo Object.

En tiempo de compilación, obviamente, sí existe dicha información.

Genericidad
Interfaces genéricas

- Las interfaces también pueden ser genéricas

```
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();
  }
}
```

Genericidad
Genericidad restringida

- *El problema con la genericidad es que sólo podemos utilizar aquellos métodos que estén definidos en Object.*

- *La genericidad restringida permite indicar que los tipos genéricos pertenezcan a una determinada jerarquía de herencia*
  - *Esto permite utilizar la interfaz de la clase usada como raíz de la jerarquía en los métodos/clases genéricos.*

```
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>;
}
```

Genericidad
Comodines

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

- Una lista de manzanas NO ES una lista de frutas. Las listas de manzans no pueden contener cualquier tipo de fruta.

- **Comodines de subtipo**

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

- ***Comodines de tipo base***

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

Bibliografía

- Cachero et. al.
  - ***Introducción a la programación orientada a Objetos***
    - Capítulo 4
- T. Budd.
  - ***An Introduction to Object-oriented Programming, 3rd ed.***
    - Cap. 11,12,14-18; cap. 9: caso de estudio en C#
- Bruce Eckl
  - ***Piensa en Java***, 4ª edición.
    - Cap. 8, 14 y 15