PROGRAMACIÓN GENÉRICA

Programación genérica

- **Objetivo:**
  - ✓ escribir algoritmos genéricos, independientes de las clases concretas de los datos (objetos) empleados.
- **Ejemplos:**
  - ✓ Algoritmos típicos: Ordenación, búsqueda,....
  - ✓ Estructuras de datos (contenedores): Listas, colas, vectores,...
- **1ª aproximación: Cualquier objeto es `Object` → describir los algoritmos en función de esta clase.**

Ejemplo: CeldaMemoria

Permite almacenar un objeto de cualquier clase

```
class CeldaMemoria {

    public Object leer () {
        return ( _valor );
    }

    public void escribir (Object x) {
        _valor = x;
    }

    private Object _valor;
}
```


Ejemplo: Test CeldaMemoria

```
class testcelda {

    public static void main (String[] args) {

        CeldaMemoria m = new CeldaMemoria ();

        m.escribir ( new Integer ( 10 ) );

        System.out.print ( "El contenido es: ");
        System.out.println ( ((Integer) m.leer()).intValue() );

    }

}
```

Ejemplo: Test CeldaMemoria

```
class testcelda {

    public static void main (String[] args) {

        CeldaMemoria m = new CeldaMemoria ();

        m.escribir ( new Integer ( 10 ) );

        System.out.print ( "El contenido es: ");
        System.out.println ( ((Integer) m.leer()).intValue() );

    }

}
```

Es transformado automáticamente a `Object` "upcasting"

Ejemplo: Test CeldaMemoria

```
class testcelda {

    public static void main (String[] args) {

        CeldaMemoria m = new CeldaMemoria ();

        m.escribir ( new Integer ( 10 ) );

        System.out.print ( "El contenido es: ");
        System.out.println ( ((Integer) m.leer()).intValue() );

    }

}
```

Es transformado automáticamente a `Object` "upcasting"

Devuelve `Object`, es preciso transformarlo a `Integer`. Esto NO es automático "downcasting"

Ejemplo: Test CeldaMemoria (2)

```
class testcelda {

    public static void main (String[] args) {

        CeldaMemoria m = new CeldaMemoria ();

        m.escribir ( new Integer ( 10 ) );

        System.out.print ( "El contenido es: ");
        System.out.println ( ((Integer) m.leer()).intValue() );

        m.escribir ( "Esto es un String" );

        System.out.print ( "El contenido es: ");
        System.out.println ( (String) m.leer() );

    }

}
```

Hay que interpretar adecuadamente el `Object` extraído.

Progr. Genérica asume requisitos

- **Cualquier estructura/algoritmo genérico asume requisitos que deben cumplir las clases a las que pertenezcan los datos.**
  - ✓ Estos requisitos pueden ser más o menos restrictivos.

```
class CeldaMemoria {

    public Object leer () {
        return ( _valor );
    }

    public void escribir (Object x) {
        _valor = x;
    }

    private Object _valor;
}
```

`Object` puede ser devuelto por un método

Se pueden asignar referencias

Progr. Genérica asume requisitos (2)

```
class Utilidades {

   public static int maximo ( int a, int b ) {
      if ( a > b ) return ( a );
      else return ( b );
   }

}
```

Permite calcular el máximo de 2 números enteros.

Progr. Genérica asume requisitos (3)

```
class Utilidades {

   public static int maximo ( int a, int b ) {
      if ( a > b ) return ( a );
      else return ( b );
   }

}
```

Permite calcular el máximo de 2 números enteros.

Intentamos generalizarlo para poder calcular el máximo de 2 Objetos.

```
class Utilidades {

   public static Object maximo ( Object a, Object b ) {
      if ( a > b ) return ( a );
      else return ( b );
   }

}
```

¿Es factible?

Progr. Genérica asume requisitos (4)

```
class Utilidades {

   public static int maximo ( int a, int b ) {
      if ( a > b ) return ( a );
      else return ( b );
   }

}
```

NO

```
class Utilidades {

   public static Object maximo ( Object a, Object b ) {
      if ( a > b ) return ( a );
      else return ( b );
   }

}
```

Se asume que dos `Object` se pueden comparar con el operador `>`, pero esto es falso.
El operador `>` NO está definido para esta clase.

Revisión del ejemplo (1)

```
class Utilidades {

   public static MI_CLASE maximo ( MI_CLASE a, MI_CLASE b ) {

      if ( a.mayorQue(b) ) return ( a );

      else return ( b );
   }

}
```

El requisito para que el método funcione es que `MI_CLASE` posea un método con la signatura:

```
public boolean mayorQue (MI_CLASE x)
```

Interfaces

- **En Java la `interface` es la clase más abstracta que existe. Consiste sólo en:**
  - ✓ Métodos públicos y abstractos.
  - ✓ Atributos públicos y finales (Constantes).
- **Se dice que una clase implementa un interfaz si define TODOS los métodos abstractos de la interfaz.**

```
class miClase implements miInterfaz
```

- **Interfaz:**
  - ✓ Especifica requisitos que debe cumplir una clase que lo implemente.
  - ✓ No incluye ningún detalle de implementación.

Especificación de un interfaz

```
interface Comparable {

   boolean mayorQue ( Comparable x );

}
```

No hace falta indicar que el método es `public` y `abstract`.
No puede ser de otra manera.

Establece el requisito de que se defina el método:

```
public boolean mayorQue (Comparable x)
```

en cualquier clase que desee cumplir (implementar) este interfaz.

Revisión del ejemplo (2)

```
class Utilidades {

   public static Comparable maximo ( Comparable a, Comparable b ) {

      if ( a.mayorQue(b) ) return ( a );

      else return ( b );
   }

}
```

Este requisito se cumple siempre en cualquier clase que implemente `Comparable`.

Implementación de interfaces

```
class Complejo implements Comparable {

//Constructores
    public Complejo (double x, double y) {...}
    public Complejo () {...}

//Métodos propios del tipo Complejo
    public double modulo () {...}
    public double real () {...}
    public double imaginario () {...}

//Redefiniciones de Object
    public String toString () {...}
    public boolean equals ( Object x ) {...}

//Implementacion del interfaz Comparable
    public boolean mayorQue ( Comparable arg ) {
        if ( this.modulo() > ((Complejo) arg).modulo() ) return ( true );
        else return ( false );
    }

//Implementacion: privado
    private double _p_real;
    private double _p_imag;
}
```

Test de la clase `Complejo`

```
class testcomplejo {

    public static void main (String[] args) {

        Complejo a = new Complejo (5.4, -67.0);
        Complejo b = new Complejo ();

        System.out.println ( "a = " + a + " , " + a.modulo());
        System.out.println ( "b = " + b + " , " + b.modulo());

        System.out.println ( "Maximo: " + Utilidades.maximo ( a, b ) );
    }
}
```

```
a = 5.4-67.0i , 67.21725968826756
b = 0.0 , 0.0
Maximo: 5.4-67.0i
```

Salida de la ejecución del test

Utilización del algoritmo genérico

Ampliación del interfaz `Comparable`

```
interface Comparable {
        boolean mayorQue ( Comparable x );
        boolean menorQue ( Comparable x );
        boolean igualQue ( Comparable x );
}
```

Amplia los requisitos a cumplir, a cambio flexibiliza su utilización.

```
class Utilidades {

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

    public static Comparable minimo ( Comparable a, Comparable b ) {
        if ( a.menorQue(b) ) return ( a );
        else return ( b );
    }
}
```

```
class Utilidades {

    public static Comparable maximo ( Comparable a, Comparable b ) {
        // Igual que antes
    }

    public static Comparable minimo ( Comparable a, Comparable b ) {
        // Igual que antes
    }

    public static void ordenacionSeleccion ( Comparable[] datos ) {
        Comparable aux;
        int min;

        for (int i = 0; i < (datos.length-1); i++) {
            min = i;
            for (int j = i+1; j < datos.length; j++) //busca minimo
                if ( datos[j].menorQue(datos[min]) ) min = j;
            if ( min != i ) { //coloca minimo en i
                aux = datos[min];
                datos[min] = datos[i];
                datos[i] = aux;
            }
        }
    }
}
```

Cualquier array de objetos Comparables.

Revisión de la clase `Complejo`

```
class Complejo implements Comparable {
//Constructores
    public Complejo (double x, double y) {...}
    public Complejo () {...}
//Métodos propios del tipo Complejo
    public double modulo () {...}
    public double real () {...}
    public double imaginario () {...}
//Redefiniciones de Object
    public String toString () {...}
    public boolean equals ( Object x ) {...}

//Implementacion del interfaz Comparable
    public boolean mayorQue ( Comparable arg )
    {//igual que antes}

    public boolean menorQue ( Comparable arg ) {
        if ( this.modulo() < ((Complejo) arg).modulo() ) return ( true );
        else return ( false );
    }

    public boolean igualQue ( Comparable arg ) {
        return ( this.equals( arg ) );
    }
//private
}
```

Test de la clase `Complejo` (2)

```
class testcomplejo {
    public static void main (String[] args) {
        Complejo[] vector = new Complejo[4];

        for (int i=0; i<4; i++)
            vector[i] = new Complejo ( 10-i, 0 );

        for (int i=0; i<4; i++)
            System.out.print (vector[i] + " ");
        System.out.println();

        Utilidades.ordenacionSeleccion ( vector );

        for (int i=0; i<4; i++)
            System.out.print (vector[i] + " ");
        System.out.println();
    }
}
```

Utilización del algoritmo genérico

```
10.0 9.0 8.0 7.0
7.0 8.0 9.0 10.0
```

Salida de la ejecución del test

Ejemplo: Estructura genérica PILA

- **Objetivo:**
  - ✓ Construir un tipo *`Pila`* que sea independiente del tipo de datos que se almacenen.
- **Hay diversas formas de implementar una Pila:**
  - ✓ Lo que define al tipo no es su implementación, son sus operaciones.
- **Podemos definir un *interfaz* al cual se deba ajustar cualquier *implementación* que se realice.**

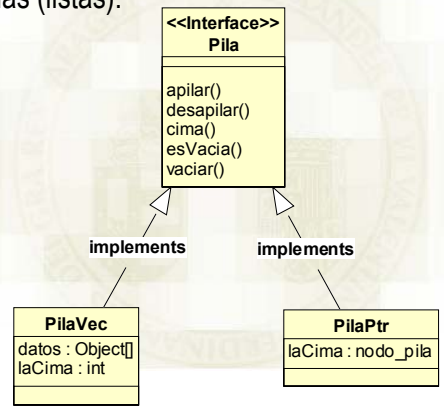
Interfaz PILA

```
interface Pila {

   void apilar ( Object x );

   void desapilar ();

   Object cima ();

   boolean esVacia ();

   void vaciar ();

}
```

Requisitos que debe cumplir cualquier implementación de una Pila

Implementaciones de PILA

- **Existen 2 posibilidades típicas:**
  - ✓ Usar un array para almacenar los datos.
  - ✓ Manejar estructuras enlazadas (listas).



Pila con Vectores

```
class PilaVec implements Pila {

   private Object[] datos;
   private int laCima;
   private static final int CAPACIDAD_MAX = 100;

   public PilaVec () {
      datos = new Object [ CAPACIDAD_MAX ];
      laCima = -1;
   }

   public void apilar ( Object x ) {
      if ( laCima < ( datos.length - 1 ) ) {
         laCima++;
         datos[laCima] = x;
      }
      else System.err.println ("Error: Desbordamiento.");
   }

   public void desapilar () {
      if ( this.esVacia() )
         System.err.println ("Error: Desbordamiento inferior.");
      else
         laCima--;
   }                                                    //continúa...
```


Pila con Vectores (2)

```
   public Object cima () {
      if ( this.esVacia() ) {
         System.err.println ("Error: Desbordamiento inferior.");
         return ( new Object() );
      }
      else
         return ( datos[laCima] );
   }

   public boolean esVacia () {
      if ( laCima == -1 ) return ( true );
      else return ( false );
   }

   public void vaciar () {
      laCima = -1;
   }

}  //fin Clase
```

Pila enlazada

```
class Nodo_Pila {

   private Object elDato;
   private Nodo_Pila elSiguiente;

   public Nodo_Pila ( Object x, Nodo_Pila sig ) {
      elDato = x;
      elSiguiente = sig;
   }

   public Nodo_Pila ( Object x ) {
      this ( x, null );
   }

   public Object dato () {
      return ( elDato );
   }

   public Nodo_Pila siguiente () {
      return ( elSiguiente );
   }

}
```

Los elementos que van a constituir la estructura.

Pila enlazada (2)

```
class PilaPtr implements Pila {

   private Nodo_Pila laCima; //referencia al elemento accesible.

   public PilaPtr () {
      laCima = null;
   }

   public void apilar ( Object x ) {
      laCima = new Nodo_Pila ( x, laCima );
   }

   public void desapilar () {
      if ( this.esVacia() )
         System.err.println ("Error: Desbordamiento inferior.");
      else
         laCima = laCima.siguiente();
   }
```

//continúa...

Pila enlazada (3)

```
   public Object cima () {
      if ( this.esVacia() ) {
         System.err.println ("Error: Desbordamiento inferior.");
         return ( new Object() );
      }
      else
         return ( laCima.dato() );
   }

   public boolean esVacia () {
      if ( laCima == null ) return ( true );
      else return ( false );
   }

   public void vaciar () {
      laCima = null;
   }

}  //fin Clase
```


Test de las Pilas

```
  class test_pila {
     public static void main (String[] args) {
        Pila mi_pila;
        mi_pila = new PilaVec();
        for (int veces = 0; veces < 2; veces++) {
           for (int i = 1; i < 11; i++ )
              mi_pila.apilar ( new Integer (i) ); //int NO es Object
           imprimir ( mi_pila );
           mi_pila = new PilaPtr();
        }
     }

     private static void imprimir ( Pila p ) {
        System.out.print ("Contenido de la pila ");
        if ( p instanceof PilaVec )
           System.out.println ("(con vectores):");
        else
           System.out.println ("(enlazada):");
        while ( ! p.esVacia() ) {
           System.out.print ( p.cima() + "," );
           p.desapilar ();
        }
        System.out.println ();
     }
  }
```

Test de las Pilas

```
class test_pila {
   public static void main (String[] args) {
      Pila mi_pila;
      mi_pila = new PilaVec();
      for (int veces = 0; veces < 2; veces++) {
         for (int i = 1; i < 11; i++ )
            mi_pila.apilar ( new Integer (i) ); //int NO es Object
         imprimir ( mi_pila );
         mi_pila = new PilaPtr();
      }
   }

   private static void imprimir ( Pila p ) {
      System.out.print ("Contenido de la pila ");
      if ( p instanceof PilaVec )
         System.out.println ("(con vectores):");
      else
         System.out.println ("(enlazada):");
      while ( ! p.esVacia() ) {
         System.out.print ( p.cima() + "," );
         p.desapilar ();
      }
      System.out.println ();
   }
}
```

```
Contenido de la pila (con vectores):
10,9,8,7,6,5,4,3,2,1,
Contenido de la pila (enlazada):
10,9,8,7,6,5,4,3,2,1,
```


Problemas de la programación genérica basada en *Object*

- Para recuperar la información el usuario debe conocer el tipo de dato almacenado.

```
List lista = new LinkedList();
lista.add(new Integer(4));
lista.add(new Integer(8));
Iterator it = lista.iterator();

int suma=0;
while (it.hasNext())
   suma = suma + ((Integer)it.next()).intValue();
System.out.println(suma);
```

Problemas de la programación genérica basada en *Object*

- Para recuperar la información el usuario debe conocer el tipo de dato almacenado.

```
List lista = new LinkedList();
lista.add(new Integer(4));
lista.add(new Integer(8));
Iterator it = lista.iterator();

int suma=0;
while (it.hasNext())
   suma = suma + ((Integer)it.next()).intValue();
System.out.println(suma);
```

Al recuperar hay que realizar el *cast* al tipo apropiado.

Problemas de la programación genérica basada en *Object*

- No hay forma de especificar que se va a crear un contenedor de un determinado tipo.

Se están almacenando tipos diferentes.

```
List lista = new LinkedList();
lista.add(new Integer(4));
lista.add("Esto es una cadena");
Iterator it = lista.iterator();

int suma=0;
while (it.hasNext())
   suma = suma + ((Integer)it.next()).intValue();
System.out.println(suma);
```

Problemas de la programación genérica basada en *Object*

- No hay forma de especificar que se va a crear un contenedor de un determinado tipo.

Se están almacenando tipos diferentes.

```
List lista = new LinkedList();
lista.add(new Integer(4));
lista.add("Esto es una cadena");
Iterator it = lista.iterator();

int suma=0;
while (it.hasNext())
   suma = suma + ((Integer)it.next()).intValue();
System.out.println(suma);
```

```
ClassCastException: java.lang.String
  at prueba.main(prueba.java:12)
  at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
  at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
  at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
  at java.lang.reflect.Method.invoke(Unknown Source)
```


Plantillas (sólo en la versión 1.5 de Java)

- Solución → Tipos parametrizados (Plantillas o *Templates*)
- Al crear el contenedor se especifica qué tipo de datos va a almacenar.
- Sintaxis:

```
Contenedor<Tipo> ref = new Contenedor<Tipo>(params);
```

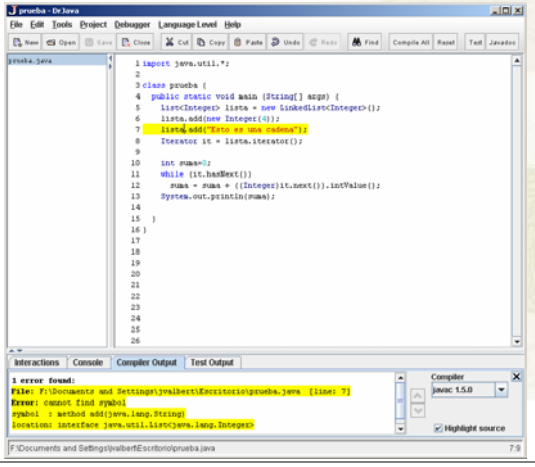
- Ejemplo: crear una `LinkedList` (clase de `java.util`) para almacenar `Integer`:

```
List<Integer> lista = new LinkedList<Integer>();
```

Especificamos que vamos a almacenar `Integer`.

Plantillas (Java 1.5)

- Ahora el compilador detecta el error, ya que se está añadiendo un `String` a una `LinkedList` cuyo tipo base es `Integer`:



Declaración de un tipo parametrizado (Java 1.5)

- Redefinimos la clase `CeldaMemoria`* para almacenar objetos de un determinado tipo, que depende del parámetro de la clase:

Parámetro del tipo

```
class CeldaMemoria<E> {
    public E leer () {
        return ( _valor );
    }
    public void escribir (E x) {
        _valor = x;
    }
    private E _valor;
}
```

**comparar esto con lo que aparece en la diapositiva 3*

Uso de un tipo parametrizado (Java 1.5)

- Veamos cómo se puede utilizar ahora `CeldaMemoria`

```
class testcelda2{
   public static void main(String[] args){
      CeldaMemoria<Integer> cm = new CeldaMemoria<Integer>();

      cm.escribir(new Integer(4));
      int i = cm.leer();
      System.out.println(i);

      // Si se descomenta la siguiente linea
      // cm.escribir("Esto es una cadena");
      // Se produce un error de compilacion

   }
}
```

No hacen falta conversiones de tipos

Herencia y tipos parametrizados

- Deseamos hacer una clase de utilidad que muestre el contenido de `CeldaMemoria`, sea cual sea el tipo de dato que está almacenando:

  ✓ Opción 1:

```
class Utilidad{

   public static void muestra(CeldaMemoria<Object> c){

      System.out.println(c.leer());

   }
}
```

Herencia y tipos parametrizados

- Deseamos hacer una clase de utilidad que muestre el contenido de `CeldaMemoria`, sea cual sea el tipo de dato que está almacenando:

  ✓ Opción 1:

```
class Utilidad{

   public static void muestra(CeldaMemoria<Object> c){

      System.out.println(c.leer());

   }
}
```

Se está asumiendo que cualquier `CeldaMemoria<Tipo>` es un `CeldaMemoria<Object>`

Herencia y tipos parametrizados

- Deseamos hacer una clase de utilidad que muestre el contenido de `CeldaMemoria`, sea cual sea el tipo de dato que está almacenando:

  ✓ Opción 1:

```
class Utilidad{

   public static void muestra(CeldaMemoria<Object> c){

      System.out.println(c.leer());

   }
}
```

FALSO

Se está asumiendo que cualquier `CeldaMemoria<Tipo>` es un `CeldaMemoria<Object>`

Herencia y tipos parametrizados

- **NO se cumple que `CeldaMemoria<Tipo>` sea un subtipo de `CeldaMemoria<Object>`**
  - ✓ Esto evita errores o una utilización incorrecta de las clases.
  - ✓ Ejemplo:

```
CeldaMemoria<Integer> original = new CeldaMemoria<Integer>();

CeldaMemoria<Object> alias = original;

alias.escribir(new String("Esto es una cadena"));

System.out.println(original.leer().intValue());
```

Si esto fuera posible, se podría almacenar cualquier objeto en `original` a través de `alias`.

Herencia y tipos parametrizados

- ✓ Opción 2: el argumento es del tipo **`CeldaMemoria<?>`**

```
class Utilidad{

   public static void muestra(CeldaMemoria<?> c){

      System.out.println(c.leer());

   }
}
```

- **El supertipo de cualquier `CeldaMemoria<Tipo>` es `CeldaMemoria<?>` (leído `CeldaMemoria` de tipo desconocido).**

Tipos parametrizados vs. Object

- Cuando declaramos que una referencia *c* pertenece al tipo *CeldaMemoria<String>*, eso nos dice algo sobre *c* que siempre es cierto en cualquier condición de uso y que además, el compilador de Java lo garantiza.

```
class CeldaMemoria<E> {
//...
    private E _valor;
}
```

- Cuando se realiza un cambio de tipo (*cast*), eso nos dice algo que el programador piensa que es cierto en un punto determinado del código y la comprobación de si el programador está en lo cierto la realiza la máquina virtual de Java sólo en tiempo de ejecución.

```
class CeldaMemoria {
//...
    private Object _valor;
}
```