Genericidad. Que una función o clase pueda trabajar o contener objetos de varios tipos.
Ejemplo de uso.

Suponed que queremos implementar una función máximo, donde los parámetros pueden ser de distinto tipo.

Solución 1. Interfaces

```
interface Comparable {
        boolean mayorQue(Object);
}
class A implements Comparable {
boolean mayorQue(Object){
        if(…) return true;
        else return false;
        }
 }
class B implements Comparable {
        boolean mayorQue(Object){
        if(…) return true;
        else return false;
        }
}
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);
```

Limitaciones: solo podré usar la función si implemento el interface Comparable (gracias al polimorfismo).

Solución 2. Funciones genéricas.

```
public <T> T maximo(T a, T b){
        if(a.mayor(b)){ return a;}
        else return b;
}
```

Limitaciones: solo podré usar el método si tiene el método mayor implementado.

Introducción.

- **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)

Centro Privado de Estudios Universitarios AulaDoce Conceptos Básicos POO Jose Manuel Baldó Pujante

Métodos genéricos.

```
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);
```

```
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);
```

Clase genérica.

```
class vector<T> {
        private T v[];
        public vector(int tam)
        { v = new T[tam]; }
        T get(int i)
        { return v[i]; }
}
vector<int> vi = new vector<int>(10);
vector<Animal> va = new
vector<Animal>(30);
//podemos almacenar cualquier tipo
//de datos derivado de Animal
```

```
class Pila<T> {
  public Pila(int nelem){
        if (nelem <= limite)
                info = new T[nelementos];
        else
                info = new T[limite];
        cima = 0;
        }
  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]);
  }
  private T info[];
  private int cima;
  private static final int limite=30;
}
```

```
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();
```

Derivar de 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);
```

Clase derivada NO genérica:

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

La clase derivada ya no tiene ningún parámetro genérico.
Instanciamos el tipo genérico.

En JAVA no existe relación entre dos clases genéricas aunque los tipos que contengan sean derivados.

```
class Uno {}
class Dos extends Uno {}
ArrayList<Uno> u = new
ArrayList<Uno>();
ArrayList<Dos> d = new ArrayList<Dos>();
u = d; // Error: incompatible types
```

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.

Interfaces genéricas.
Podemos crear interfaces genéricas.

```
interface nombre-interfaz<T1, T2, T3...>{
  T1 metodo1(T1 t);
  T2 metodo2(T2 t);…
}
```

DEFINICIÓN

```
class nombre-clase<T1, T2, T3...>
 implements nombre-interfaz<T1, T2, T3,..>{
  T1 metodo1(T1 t){
  // conversiones no necesarias para acceder a
// los métodos de mi nombre-clase
 }
  T2 metodo2(T2 t){ // implementación}…
}
```

IMPLEMENTACIÓN

Antes…

```
interface nombre-interfaz{
  Object metodo1(Object t);
  Object metodo2(Object t);…
}
```

DEFINICIÓN

```
class nombre-clase
 implements nombre-interfaz{
  Object metodo1(Object t){
  // conversiones necesarias para acceder a
  // los métodos de mi nombre-clase
  }
  Object metodo2(Object t){}…
}
```

IMPLEMENTACIÓN

Ejemplo genérico:

```
interface Contenedora{
  boolean contiene(Object valor);
}
```

El contrato hace que tu clase contenga el método contiene, para poder consultar si tu clase contiene un elemento concreto.

```
class Verificadora<T> implements Contenedora<T>
{       private T[] datos;
        public  Verificadora(T[] x){
                datos = x;
 }
public boolean contiene(T dato) {
        for(T valor: datos)
                if(valor.equals(dato))
                        return true;
         return false;
  }
}
```

La limitiación existente es que las funcione que usemos con los templates deben estar definidas en object.
La interfaz Contenedora no limita el tipo de dato con que trabajará, puede ser Integer, Double, Float o inclusive String, cualquier Object en general, por lo tanto podemos utilizar el método equals de esta clase (Object), tal como lo hace la clase verificadora en el método contiene().

```
public class Main {
public static void main(String[] args) {
                Integer[] x = {0,1,2,3,4};
                Verificadora<Integer>  ver = new Verificadora<Integer>(x);
                if(ver.contiene(2)){
                        System.out.println("Si tiene 2");
                }else{
                        System.out.println("NO tiene 2");
                }
        }
}
```

Implementaciones

OK: class Verificadora<T> implements Contenedora<T>{…}

NO OK: class Verificadora implements Contenedora<T>{…}

OK: class Verificadora implements Contenedora<String>{…}

Limitar el tipo de dato con que trabajará la interfaz

```
class Verificadora<T extends Number>
implements Contenedora<T extends Number>{ ERROR

interface Contenedora<T extends Number>{
  boolean contiene(T valor);
}

class Verificadora<T extends Number>
        implements Contenedora<T>{ OK
```