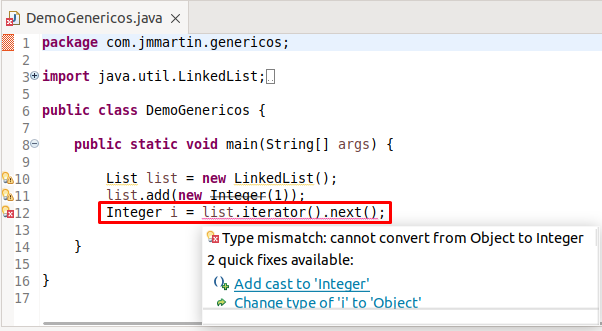
1. Java Entorno Servidor

# 1.1. Genéricos

Se introducen en Java 5 y dotan al lenguaje de:

* una mejor comprobación de tipos en tiempo de compilación
* y, al mismo tiempo, eliminan los *casteos* al usar las colecciones.

En el ejemplo siguiente, se produce un error de compilación al intentar establecer una salida de una lista a un tipo entero, aún cuando el valor almacenado en la lista se corresponda con la referencia.



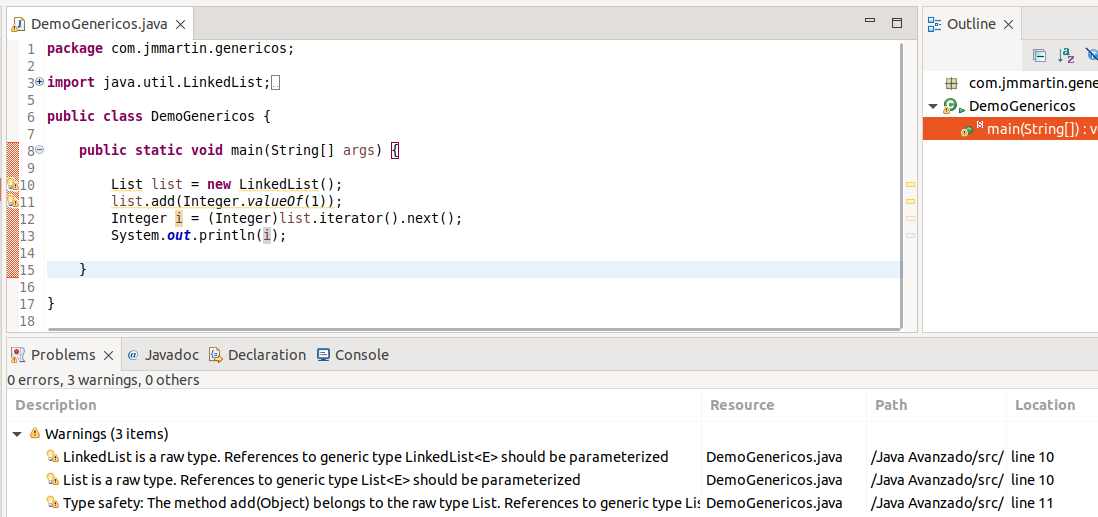


|  |
| --- |
| **import** java.util.LinkedList; **import** java.util.List;  **public** **class** **DemoGenericos** {   **public** **static** **void** **main**(String[] args) {    List list = **new** LinkedList();  list.add(**new** Integer(1));  Integer i = list.iterator().next();  }   } |

|  |
| --- |
| Esto se producía porque en las colecciones Java antes de la versión 5, se devolvía la clase de objeto más general de Java: Object, el padre de todos los objetos en la estructura jerárquica de clase de Java. |

## 1.1.1. *Casteo* de colecciones

Para solucionarlo sin utilizar tienes que utilizar un *cast*:

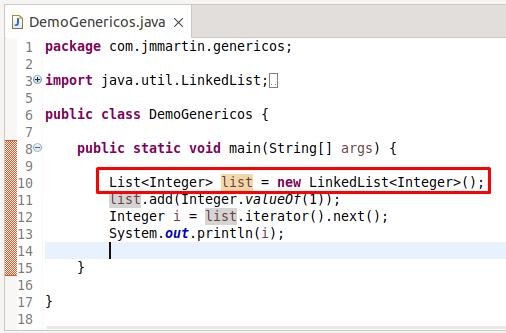


|  |
| --- |
| **import** java.util.LinkedList; **import** java.util.List;  **public** **class** **DemoGenericos** {   **public** **static** **void** **main**(String[] args) {    List list = **new** LinkedList();  list.add(Integer.valueOf(1));   Integer i = (Integer)list.iterator().next();  System.out.println(i);    }   } |

El compilador de Java indica mediante unos avisos que se utiliza una colección *LinkedList* sin parametrización de genéricos, directamente en *crudo* sobre Object.

## 1.1.2 Uso de genéricos en colecciones

La forma correcta de utilizar estas colecciones será mediante la parametrización con genéricos:

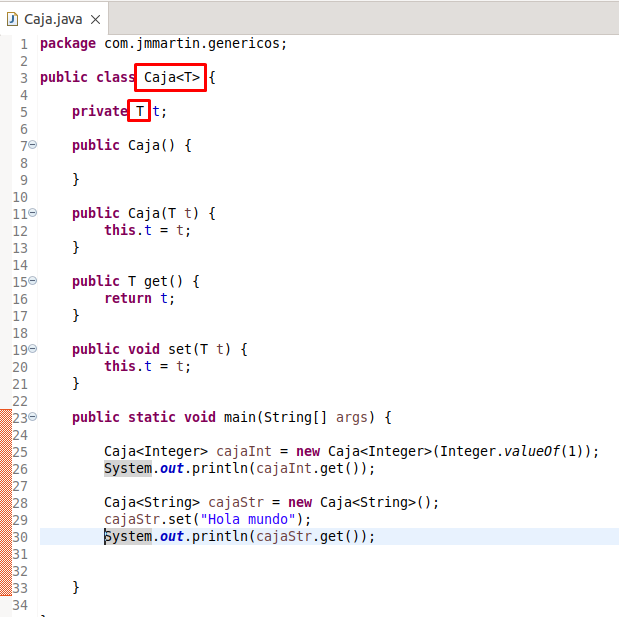


|  |
| --- |
| package org.iesbelen.genericos;  **import** java.util.LinkedList; **import** java.util.List;  **public** **class** **DemoGenericos** {   **public** **static** **void** **main**(String[] args) {    List<Integer> list = **new** LinkedList<Integer>();  list.add(Integer.valueOf(1));   Integer i = list.iterator().next();  System.out.println(i);    }   } |

La declaración del tipo genérico ahorra el *casteo* cada vez que se accede a la colección.

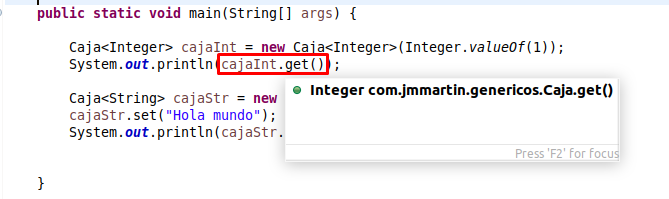
## 1.1.3 Clase genérica

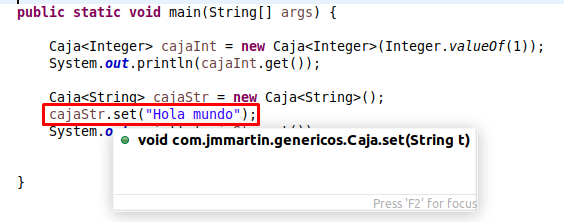
Una clase también puede tener la parametrización con genéricos. Veamos un ejemplo:



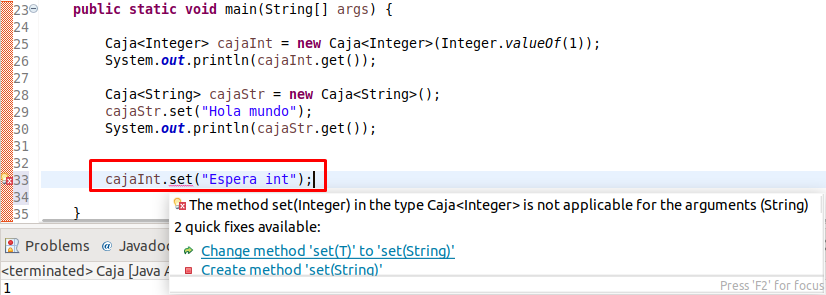
|  |
| --- |
| **public** **class** **Caja**<**T**> {    **private** T t;    **public** Caja() {    }    **public** Caja(T t) {  this.t = t;  }   **public** T **get**() {  **return** t;  }   **public** void **set**(T t) {  this.t = t;  }    **public** **static** void main(**String**[] args) {    Caja<Integer> cajaInt = **new** **Caja**<Integer>(Integer.valueOf(1));  System.out.println(cajaInt.**get**());    Caja<**String**> cajaStr = **new** **Caja**<**String**>();  cajaStr.**set**("Hola mundo");  System.out.println(cajaStr.**get**());   } } |

Si comprobamos los tipos devueltos por los métodos *get* y *set* en los objetos *cajaInt* y *cajaStr* vemos cómo actúa la parametrización de la clase por genérico preparando para la compilación el tipo.



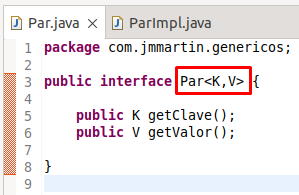


Si se intenta establecer en el objeto *cajaInt* un tipo distinto del parámetro indicado en la instanciación del objeto <Integer> se nos presentará un error de compilación:



## 1.1.4 Clase con múltiples tipos genéricos

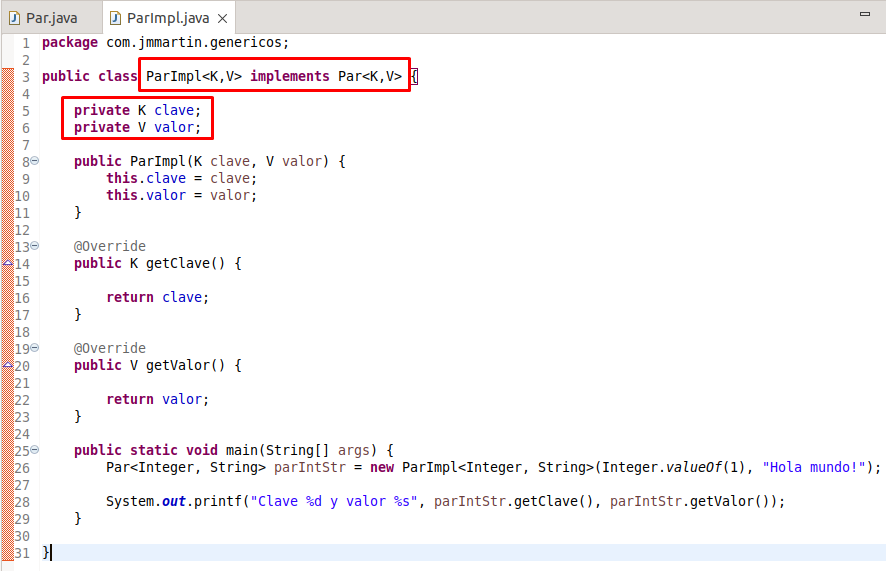
Una clase puede tener múltiples tipos genéricos. En este caso la parametrización por genéricos va en una interfaz:



|  |
| --- |
| public interface Par<K,V> {    public K getClave();  public V getValor(); } |

Si se crea una clase que implementa la interfaz anterior:

implementa los metodos de la interface



Esa clase arrastrará los parámetros del interfaz a la clase que la implementa.

|  |
| --- |
| **public** **class** **ParImpl**<**K**,**V**> **implements** **Par**<**K**,**V**> {   **private** K clave;  **private** V valor;    **public** **ParImpl**(K clave, V valor) {  **this**.clave = clave;  **this**.valor = valor;  }    **@Override**  **public** K **getClave**() {   **return** clave;  }   **@Override**  **public** V **getValor**() {   **return** valor;  }   **public** **static** **void** **main**(String[] args) {  Par<Integer, String> parIntStr = **new** ParImpl<Integer, String>(Integer.valueOf(1), "Hola mundo!");    System.out.printf("Clave %d y valor %s", parIntStr.getClave(), parIntStr.getValor());  } } |

## 1.1.5 Convención de parámetro

E: elemento de una colección.

K: clave.

N: número.

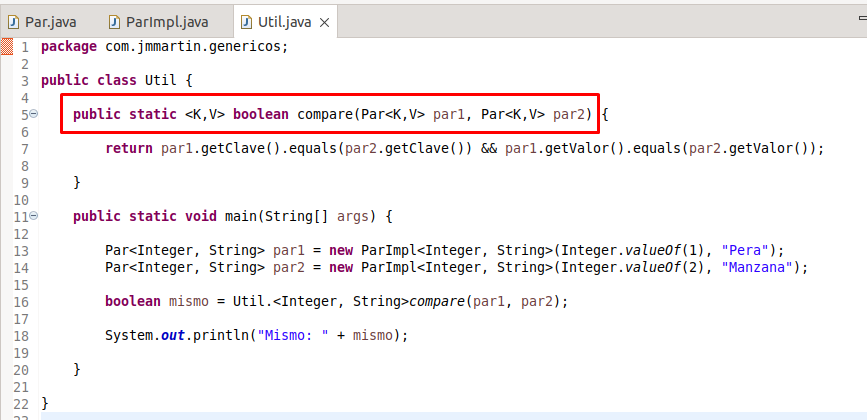
T: tipo.

V: valor.

S, U, V etc: para segundos, terceros y cuartos tipos.

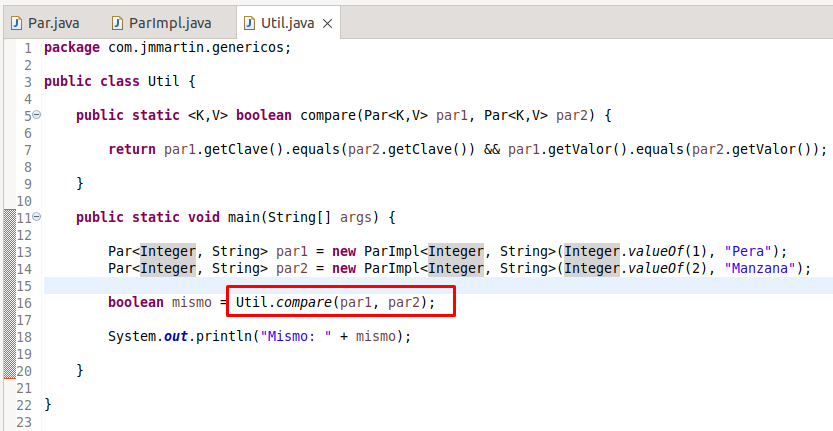
## 1.1.6 Tipos genéricos en métodos

Estos genéricos también se pueden utilizar en los métodos. En el caso del método estático compare tenemos una parametrización múltiple de genéricos:



|  |
| --- |
| **public** **class** **Util** {    **public** **static** <K,V> boolean compare(Par<K,V> par1, Par<K,V> par2) {    **return** par1.getClave().equals(par2.getClave()) && par1.getValor().equals(par2.getValor());    }    **public** **static** void main(**String**[] args) {    Par<Integer, **String**> par1 = **new** **ParImpl**<Integer, **String**>(Integer.valueOf(1), "Pera");  Par<Integer, **String**> par2 = **new** **ParImpl**<Integer, **String**>(Integer.valueOf(2), "Manzana");    boolean mismo = Util.<Integer, **String**>compare(par1, par2);    System.out.println("Mismo: " + mismo);    }  } |

Puesto que los parámetros podrán inferirse de los parámetros del método es común omitirlos al realizar la invocación:

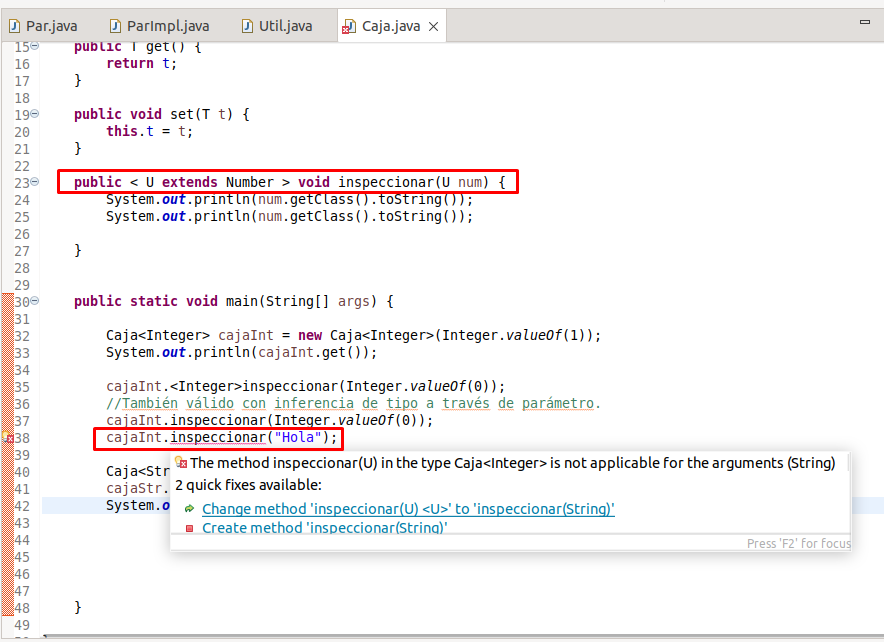


## 1.1.7 <T extends U> tipo conocido T que extiende U



Pero si no existe relación de herencia de T a U (U clase padre y T clase hija) se tendrá un error.

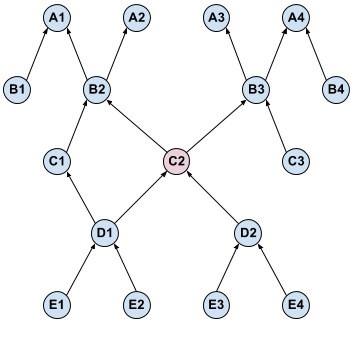
En el caso siguiente cuando el método *inspeccionar* recibe como parámetro un String que no tiene herencia de Number conforme a la definición del parámetro < U extends Number> dará un error de compilación.



|  |
| --- |
| No existe herencia en genéricos básicos.  Caja<Number> cajaInt = new Caja<Integer> //Da error de compilación    Solución: |

## 1.1.8 <? extends U>

Teniendo un esquema de herencia como el siguiente:



Supongamos que tenemos la siguiente declaración:

|  |
| --- |
| **List<?** extends C2> **list**; |

Con esto estás diciendo que *list* podrá hacer referencia a un objeto, por ejemplo, de tipo ArrayList cuyo tipo genérico es uno de los 7 subtipos de C2 (C2 incluido):

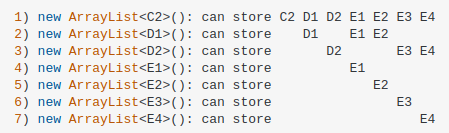
C2: new ArrayList<C2>();, (un objeto que puede almacenar C2 o subtipos) o

D1: new ArrayList<D1>();, (un objeto que puede almacenar D1 o subtipos) o

D2: new ArrayList<D2>();, (un objeto que puede almacenar D2 o subtipos) o…

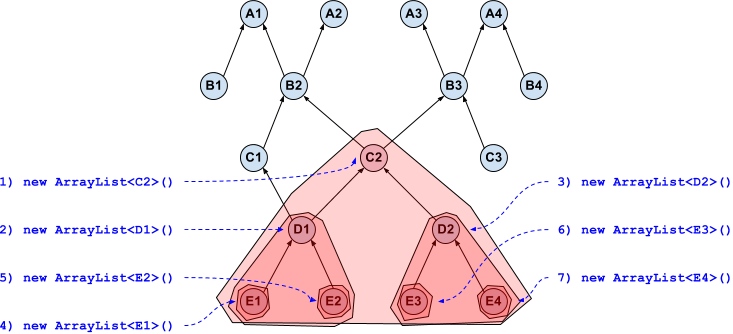
List<? extends C2> list = new ArrayList<B2>(); ERROR!!

y así. Siete casos diferentes:



Tenemos un conjunto de tipos "almacenables" para cada caso posible:

* 7 conjuntos (rojos)



Fíjate en los siguientes casos que **no** se podrían cumplir:

no puede añadir C2 list.add(**new** C2(){}); si se trata de un array de D1 list = new ArrayList<D1>()*;* porque C2 no es subtipo D1

no puede añadir D1 list.add(**new** D1(){}); si se trata de un array de D2 list = new ArrayList<D2>()*;*porque D2 no es subtipo D2

y así.

## 1.1.9. <? super U>

De igual forma podemos tener la siguiente referencia mediante la declaración super:

|  |
| --- |
| **List<?** super C2> **list**;  Dado un elemento cualquiera que herede de super C2 |

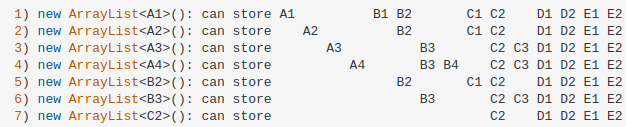
Estás diciendo que *list* podrá hacer referencia a un objeto, por ejemplo, de tipo ArrayList cuyo tipo genérico es uno de los 7 supertipos de C2 (C2 incluido):

A1: new ArrayList<A1>();, (un objeto que puede almacenar A1 o subtipos) o

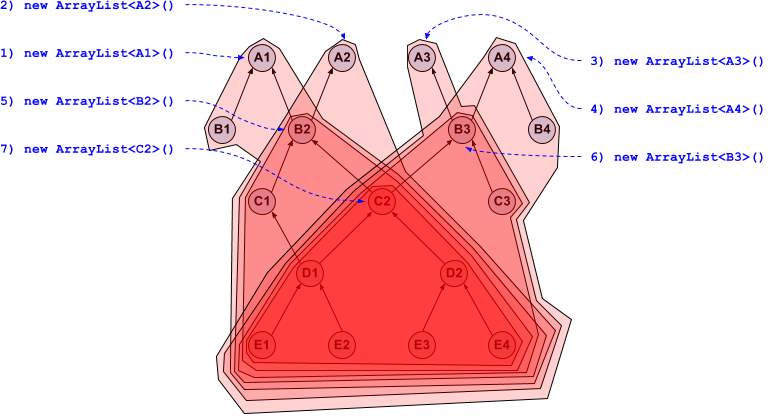
A2: new ArrayList<A2>();, (un objeto que puede almacenar A2 o subtipos) o

A3: new ArrayList<A3>();, (un objeto que puede almacenar A3 o subtipos) o...

y así. Siete casos diferentes:



Tenemos un conjunto de tipos "almacenables" para cada caso posible: 7 conjuntos (rojos) aquí representados gráficamente:

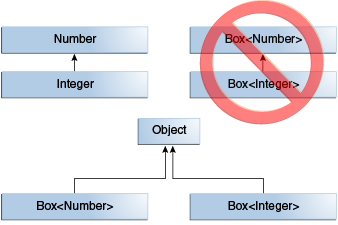


Fíjate en los siguientes casos que se **sí** podrían cumplir:

|  |
| --- |
| list.add(**new** C2(){}); |

list.add(**new** D1(){});

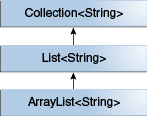
## 1.1.10.Herencia de tipos genéricos



Destacar que *Box<Integer>* y *Box<Double>* en Java no son subtipos de *Box<Number>*, aún cuando Integer si sea clase hija de Number.

Los genéricos pueden extenderse (herencia de clase) o implementarse (implementar una interfaz) y mientras no se cambie el tipo del argumento la «relación es un» se preserva.

De modo que *ArrayList<String>* es un subtipo de *List<String>* que a su vez es un subtipo de *Collection<String>*.



Es decir, se puede tener código como el siguiente:

|  |
| --- |
| **interface** **PayloadList**<**E**,**P**> **extends** **List**<**E**> {  **void** setPayload(int index, P val);  ... }  PayloadList<String,String> PayloadList<String,Integer> PayloadList<String,Exception> |

Contravarianza -> Consumir -> Super -> introducir datos.

Covarianza -> Producir -> Extends -> leer o sacar datos

Regla de los PECS de Java.

## 1.1.11 Ejercicios

1. Escribe una clase Pila genérica usando para ello un atributo del tipo LinkedList. La clase Pila tendrá los siguientes métodos: LIFO

* estaVacia(): devuelve true si la pila está vacía y false en caso contrario.
* extraer(): devuelve y elimina el primer elemento de la colección.
* primero(): devuelve el primer elemento de la colección
* aniadir(): añade un objeto por el extremo que corresponda.
* toString(): devuelve en forma de String la información de la colección.

1. NO Implementa una pila utilizando como atributos un array genérico y un entero que cuente el número de objetos insertados. La clase se debe llamar PilaArray y tiene los mismos métodos que la pila del ejercicio anterior.
2. Escribe una clase Matriz genérica con los siguientes métodos:

* constructor que recibe por parámetro el número de filas y columnas de la matriz.
* set() recibe por parámetro la fila, la columna y el elemento a insertar. El elemento es de tipo genérico. Este método inserta el elemento en la posición indicada.
* get() recibe por parámetro la fila y la columna. Devuelve el elemento en esa posición. El elemento devuelto es genérico.
* columnas() devuelve el número de columnas de la matriz.
* filas() devuelve el número de filas de las matriz.
* toString() devuelve en forma de String la información de la matriz.

1. NO Escribe una aplicación que:

* Cree una matriz de Integer de 4 filas y 2 columnas
* Rellénala con números consecutivos comenzando por el 1.
* Muestra por pantalla la matriz.
* Muestra por pantalla el contenido de la celda en la fila 1, columna 2.

1. NO Escribe una interfaz ColeccionSimpleGenerica , que como su propio nombre indica, es genérica, con los siguientes métodos:

* estaVacia(): devuelve true si la pila está vacía y false en caso contrario
* extraer(): devuelve y elimina el primer elemento de la colección.
* primero(): devuelve el primer elemento de la colección.
* aniadir(): añade un objeto por el extremo que corresponda.

1. Escribe una clase genérica ListaOrdenada con un tipo parametrizado E que sea Comparable… < E extends Comparable<E>>.... La clase debe tener lo siguiente:

* Un constructor
* void add(E o) - nota: cuando añado el elemento debería añadirse en el orden adecuado, recuerda que E tiene que implementar Comparable<E>
* E get(int index)
* int size()
* boolean isEmpty()
* boolean remove(E o)
* int indexOf(E o)
* String toString()

1. NO Escribe la interfaz Operable, con un tipo genérico E que implemente todas las operaciones aritméticas (+-\*/). Todos los métodos de esta interfaz reciben un objeto de tipo E y devuelven un objeto de tipo E. De esta manera, se opera el objeto que llame al método con el que se recibe por parámetro y se devuelve el resultado. Ambos, operandos y el resultado, son del mismo tipo.

Finalmente, en un Test crea una clase (interna) anónima que implemente esta interfaz.

APÉNDICE DE CÓDIGO DE PRUEBAS DE COVARIANZA Y CONTRAVARIANZA DE GENÉRICOS

Repo: https://github.com/toteabe/apendice\_genericos.git

package org.iesvdm;

import java.util.Collection;

import java.util.List;

public class Caja<T>{

private T t;

public Caja(T t) {

this.t = t;

}

public T getT() {

return t;

}

public void setT(T t) {

this.t = t;

}

public void reemplazaPorPrimeroEnColeccion(Collection<? extends T> coleccion) {

if (coleccion.iterator().hasNext()) {

T t1 = coleccion.iterator().next();

this.t = t1;

}

}

public static <T> Caja<T> convertirATipoT(Caja<? super T> cajaSuper) {

Caja<T> cajaT = (Caja<T>)cajaSuper;

return cajaT;

}

public <U> String escribirJunto(U u) {

return "" + this.t + u;

}

@Override

public String toString() {

return "Caja{" +

"t=" + t +

'}';

}

}

package org.iesvdm;

import java.util.List;

public class Util {

public static <T> void llenaConT(List<? super T> listSuperT, T t) {

listSuperT.add(t);

}

}

import org.iesvdm.Caja;

import org.iesvdm.Util;

import org.junit.jupiter.api.Assertions;

import org.junit.jupiter.api.Test;

import java.util.ArrayList;

import java.util.List;

public class GenericosTest {

@Test

public void testInstanceOfClassCaja() {

Caja<Integer> cajaInteger = new Caja<>(1);

Caja<String> cajaString = new Caja<>("UNO");

Assertions.assertTrue(cajaString instanceof Caja<?>);

Assertions.assertTrue(cajaInteger instanceof Caja);

Assertions.assertTrue(Caja.class.isInstance(cajaString));

Assertions.assertTrue(cajaString.getClass().isInstance(cajaInteger));

//Aunque sean instancias del mismo tipo para instanceof

//No pueden asignarse entre sí, debido a la comprobación en tiempo de compilación

//de los genéricos.

//cajaString = cajaInteger;

}

@Test

public void testMetodoGenericoCaja() {

Caja<Integer> cajaInteger = new Caja<>(1);

String junto = cajaInteger.<Double>escribirJunto(0.1d);

Assertions.assertEquals("10.1", junto);

}

@Test

public void testCOVARIANZAListExtendsNumber() {

//NO EXISTE VARIANZA EN LOS TIPOS DE GENÉRICOS

//No hay herencia en genéricos

//Produce error de compilación

//List<Number> listNumber = new ArrayList<Integer>();

//NOTA: LA VARIANZA ES EL MECANISMO DE POLIMORFISMO DE LA HERENCIA

// C extends B extends A

// C ------> B ------> A

// A a = new B();

// A a = new C();

// B b = new C();

// EN JAVA NO EXISTE VARIANZA DIRECTA EN LOS TIPOS GENÉRICOS

// Z<C> x-------> Z<B> x------> Z<A>

// SE HACE MEDIANTE LA COVARIANZA: ? extends T

List<Number> listNumber = new ArrayList<>();

Integer i = 1;

listNumber.add(i);

Assertions.assertEquals(1, listNumber.size());

// COVARIANZA: ? extends T

//No hay error de compilación por el uso de ? extends Number

//SE DICE QUE EL PARÁMETRO DEL TIPO ES COVARIANTE

// COVARIANTE

// |

// V

List<? extends Number> listExtNumber = new ArrayList<Integer>();

//Error de compilación

//listExtNumber.add(i);

//Pero sólo se puede añadir null, PARA NO CORROMPER LA MEMORIA, LISTAS DE UN SÓLO TIPO EN GENÉRICOS

listExtNumber.add(null);

//Debe existir previamente

List<Integer> listInteger = new ArrayList<>();

listInteger.add(i);

//Y asignando uno ya creado con elementos, sí se puede leer ? extends U (COVARIANTE)

// , en este caso U = Number

List<? extends Number> listExtNumber2 = listInteger;

for (Number number: listExtNumber2

) {

System.out.println(number);

System.out.println((Integer)number);

}

}

@Test

public void testCONTRAVARIANZAListSuperNumber() {

//CONTRAVARIANZA

// ? super T ==>

Integer i = 1;

//CONTRAVARIANTE

// |

// V

List<? super Number> listSupNumber = new ArrayList<Number>();

//CONTRAVARIANTE

// |

// V

List<? super Number> listSupNumber2 = new ArrayList<Object>();

//Object es super de Number --------------------------^

//SE DICE QUE EL PARÁMETRO DEL TIPO ES CONTRAVARIANTE

//Puedo añadir Integer

listSupNumber.add(i);

//Object es super de Number

//error de compilación al añadir object

//Object obj = new Object();

//listSupNumber.add(obj);

listSupNumber2.add(i);

//Object es super de Number

//error de compilación al añadir object

//Object obj = new Object();

//listSupNumber2.add(obj);

//Error de compilación

//for (Number number: listSupNumber

// ) {

// System.out.println(number);

//}

//Sólo se puede iterar a Object

for (Object obj: listSupNumber

) {

System.out.println(obj);

}

Assertions.assertTrue(true);

}

@Test

public void testProductor\_extendsT() {

Integer i = 1;

Caja<Number> cajaNumber = new Caja<>(i);

List<Short> listShort = new ArrayList<>();

Short s = 2;

listShort.add(s);

//Firma de productor extends T

// public void reemplazaPorPrimeroEnColeccion(Collection<? extends T> coleccion) {

cajaNumber.reemplazaPorPrimeroEnColeccion(listShort);

Assertions.assertEquals(s, cajaNumber.getT());

}

@Test

public void testConsumidor\_superT() {

Integer i = 1;

Caja<Number> cajaNumber = new Caja<>(i);

List<Short> listShort = new ArrayList<>();

Short s = 2;

listShort.add(s);

//Firma de productor extends T

// public void reemplazaPorPrimeroEnColeccion(Collection<? extends T> coleccion) {

cajaNumber.reemplazaPorPrimeroEnColeccion(listShort);

Assertions.assertEquals(s, cajaNumber.getT());

}

@Test

public void convertirATipoTTest() {

Integer i = 1;

Caja<Number> cajaNumber = new Caja<>(i);

Number number = cajaNumber.getT();

Assertions.assertTrue( cajaNumber.getT() instanceof Number);

// CONTRAVARIANTE

// |

// V

//public static <T> Caja<T> convertirATipoT(Caja<? super T> cajaSuper) {

Caja<Integer> cajaInteger = Caja.convertirATipoT(cajaNumber);

Integer iNumber = cajaInteger.getT();

Assertions.assertTrue(cajaInteger.getT() instanceof Integer);

}

@Test

public void llenaListSuperTTest() {

List<Number> listNumber = new ArrayList<>();

Integer i = 1;

// CONTRAVARIANTE

// |

// V

//public static <T> void llenaConT(List<? super T> listSuperT, T t) {

Util.llenaConT(listNumber, i);

Assertions.assertEquals(i, listNumber.iterator().next());

}

}