# MP485 - Programación

# Generics en Java

IES ANTÓN LOSADA DIÉGUEZ

Marcia Fernández Estévez

Marzo 2022

# Indice

## Introducción

Los *generics* fueron introducidos en la versión 5 de Java en 2004.

Tambíen se conocen como “Polimorfismo paramétrico o de tipos”.
Son importantes ya que permiten al compilador informar de muchos errores de compilación que hasta el momento solo se descubrirían en tiempo de ejecución, al mismo tiempo permiten eliminar los *cast. Esto* reduce la repetición y aumenta la legibilidad del código. Los errores por *cast* inválido son especialmente problemáticos de *debuggear* ya que el error se suele producir en un sitio alejado del de la causa.

**Los *generics* permiten usar tipos para parametrizar las clases, interfaces y métodos al definirlas. La característica más importante de los genéricos es que el código puede reutilizarse.**

Los beneficios son:

- Comprobación de tipos, más fuerte en tiempo de compilación.
- Eliminación de *casts,* aumentando la legibilidad del código.
- Posibilidad de implementar algoritmos genéricos, con tipado seguro.

- Una **variable de tipo** es un identificador no calificado. Las variables de tipo se introducen mediante declaraciones de clases genéricas, declaraciones de interfaz genéricas, declaraciones de métodos genéricos y declaraciones de constructores genéricos.
- Una **clase** es genérica si declara una o más variables de tipo. Estas variables de tipo se conocen como los parámetros de tipo de la clase. Define una o más variables de tipo que actúan como parámetros. Una declaración de clase genérica define un conjunto de tipos parametrizados, uno para cada posible invocación de la sección de parámetros de tipo. **Todos estos tipos parametrizados comparten la misma clase en tiempo de ejecución.**
- Una **interfaz** es genérica si declara una o más variables de tipo. Estas variables de tipo se conocen como los **parámetros de tipo de la interfaz**. Define una o más variables de tipo que actúan como parámetros. Una declaración de interfaz genérica define un conjunto de tipos, uno para cada posible invocación de la sección de parámetros de tipo. Todos los tipos parametrizados comparten la misma interfaz en tiempo de ejecución.
- Un **método** es genérico si declara una o más variables de tipo. Estas variables de tipo se conocen como los **parámetros de tipo formal del método**. La forma de la lista de parámetros de tipo formal es idéntica a la lista de parámetros de tipo de una clase o interfaz.
- Un **constructor** puede declararse como genérico, independientemente de si la clase en la que se declara el constructor es genérica. Un constructor es genérico si declara una o más variables de tipo. Estas variables de tipo se conocen como los **parámetros de tipo formal del constructor.** La forma de la lista de parámetros de tipo formal es idéntica a la de una lista de parámetros de tipo de una clase o interfaz genérica.

## Clase genérica

Una clase genérica puede tener múltiples argumentos de tipos y los argumentos pueden ser a su vez tipos genéricos.

**Después del nombre de la clase se puede indicar la lista de parámetros de tipos con el formato <T1, T2, T3, ...>.**

```java
public class Box<T> {
  private T t;

  public T get() { return t; }
  public void set(T t) { this.t = t; }
}
```

Box.java

```java
public interface Pair<K, V> {
  public K getKey();
  public V getValue();
}
```

Pair.java

```java
public class OrderedPair<K, V> implements Pair<K, V> {

  private K key;
  private V value;

  public OrderedPair(K key, V value) {
    this.key = key;
    this.value = value;
  }

  public K getKey() { return key; }
  public V getValue() { return value; }
}
```

OrderedPair.java

## Convención nombres de parámetros

Por convención, estos son  los nombres de los parámetros de tipo usados:

- E: elemento. Muy usado en colecciones.
- K: clave o llave. Usado en mapas
- N: número.
- T: tipo. Representa una clase.
- V: valor. También usado en mapas.
- S, U, V etc: para segundos, terceros y cuartos tipos.

En el momento de la instanciación de un tipo genérico indicaremos el argumento para el tipo, en este caso *Box* contendrá una referencia a un tipo *Integer*. Con Java 7 se puede usar el operador *diamond* (diamante) y el compilador inferirá el tipo según su definición para mayor claridad en el código.

Podemos usar cualquiera de estas dos maneras, prefiriendo usar el operador *diamond* por ser más clara.

```
1    Box<Integer> integerBox1 = new Box<Integer>();
2    Box<Integer> integerBox2 = new Box<>();
3    OrderedPair<String, Integer> p1 = new OrderedPair<>("Even", 8);
```

Instantation.java

Para mantener la compatibilidad con versiones anteriores a Java 5 los tipos genéricos que al usarse no indican argumentos de tipo se denominan *raw*. El compilador indicará una advertencia como un uso potencialmente peligroso ya que no podrá validar los tipos.

```
1    Box rawBox = new Box();
```

Raw.java

Además de las clases los métodos también pueden tener su propia definición de tipos genéricos.

```
1    public static <K, V> boolean compare(Pair<K, V> p1,  Pair<K, V> p2) {
2      return p1.getKey().equals(p2.getKey()) && p1.getValue().equals(p2.getValue());
3    }
```

Method.java

La sintaxis completa de uso sería:

```
1    Pair<Integer, String> p1 = new OrderedPair<>(1, "apple");
2    Pair<Integer, String> p2 = new OrderedPair<>(2, "pear");
3    boolean same = Util.<Integer, String>compare(p1, p2);
```

MethodUsage.java

Aunque puede abreviarse ya que el compilador puede inferir los tipos:

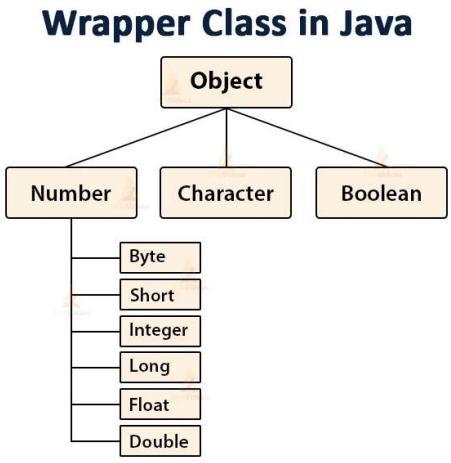
```
1    boolean same = Util.compare(p1, p2);
2
```

MethodUsageInference.java

## Bounded type - Limitación de tipos - “Tipos encerrados”

A veces, queremos limitar los tipos que pueden ser usados empleando lo que se denomina *bounded type*. Con <U extends Number> el tipo *U* debe extender la clase *Number*.

```
1    public class BoxBounds<T> {
2
3      private T t;
4
5      public void set(T t) {
6        this.t = t;
7      }
8
9      public T get() {
10       return t;
11     }
12
13     public <U extends Number> void inspect(U u){
14       System.out.println("T: " + t.getClass().getName());
15       System.out.println("U: " + u.getClass().getName());
16     }
17
18     public static void main(String[] args) {
19       Box<Integer> integerBox = new Box<Integer>();
20       integerBox.set(new Integer(10));
21       integerBox.inspect("some text"); // error: this is still String!
22     }
23   }
```




BoxBounds.java

Una clase puede tener múltiples limitaciones, si una es una clase debe ser la primera y el resto de argumentos interfaces.

```
<T extends B1 & B2 & B3>

Class A { /* ... */ }
interface B { /* ... */ }
interface C { /* ... */ }

class D <T extends A & B & C> { /* ... */ }
```

Bounds.java

## Tipos “es un”

En Java un tipo puede ser asignado a otro mientras el primero sea compatible con el segundo, es decir tengan una «relación es un». Una referencia de *Object* puede referenciar una instancia de *Integer* (un *Integer* es un *Object*).

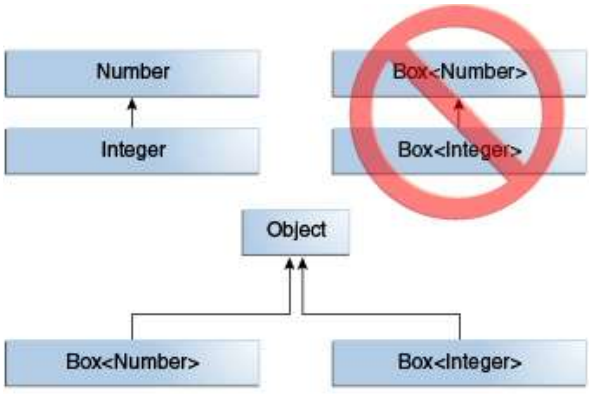
```
1  Object object = new Object();
2  Integer integer = new Integer(10);
3  object = integer;
```

IsA.java

Sin embargo, en el caso de los *generics*,

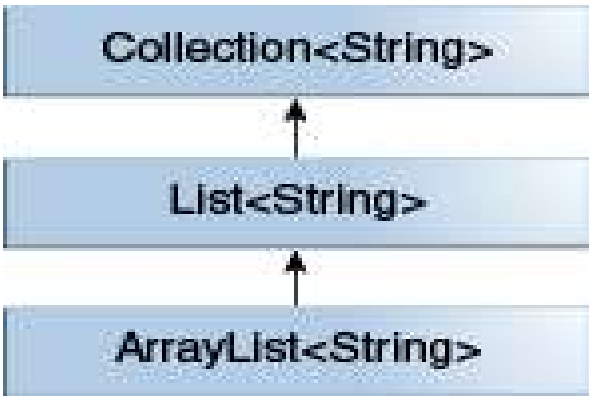
¿una referencia de *Box<Number>* puede aceptar una instancia *Box<Integer>* or *Box<Double>* aun siendo *Integer* y *Double* subtipos de *Number*?.

**La respuesta es no, ya que *Box<Integer>* y *Box<Double>* en Java no son subtipos de *Box<Number>***. La jerarquía de tipos es la siguiente:



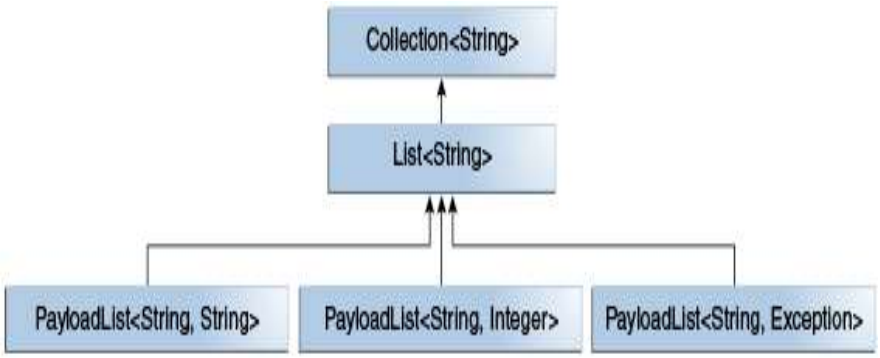

Los tipos genéricos pueden extenderse o implementarse y mientras no se cambie el tipo del argumento la «relación es un» se preserva.

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




```
1  interface PayloadList<E,P> extends List<E> {
2    void setPayload(int index, P val);
3    ...
4  }
5
6  PayloadList<String,String>
7  PayloadList<String,Integer>
8  PayloadList<String,Exception>
```

PayloadList.java



## Wildcard - Comodín

En los *generics* un parámetro para un tipo *?* se denomina *wildcard* siendo este un tipo desconocido.

Se usan para imponer restricciones:

- ☐ `<? extends Base>` admite cualquier tipo que extienda de la clase Base
- ☐ `<? super Base>` admite cualquier tipo que sea supertipo de la clase Base
- ☐ `<?>` admite cualquier tipo, sin nombrarlo

Son usados para reducir las restricciones de un tipo de modo que un método pueda funcionar con una lista de *List<Integer>*, *List<Double>* y *List<Number>*. El término *List<Number>* es más restrictivo que *List<? extends Number>* porque el primero solo acepta una lista de *Number* y el segundo una lista de *Number* o de sus subtipos. *List<? extends Number>* es un *upper bounded wildcard*.

```
1  public static void process(List<? extends Number> list) { /* ... */ }
2
```

BoundedWildcard.java

Se puede definir una lista de un tipo desconocido, *List<?>*, en casos en los que:

- La funcionalidad se puede implementar usando un tipo *Object*.
- Cuando el código usa métodos que no dependen del tipo de parámetro. Por ejemplo, *List.size* o *List.clear*.

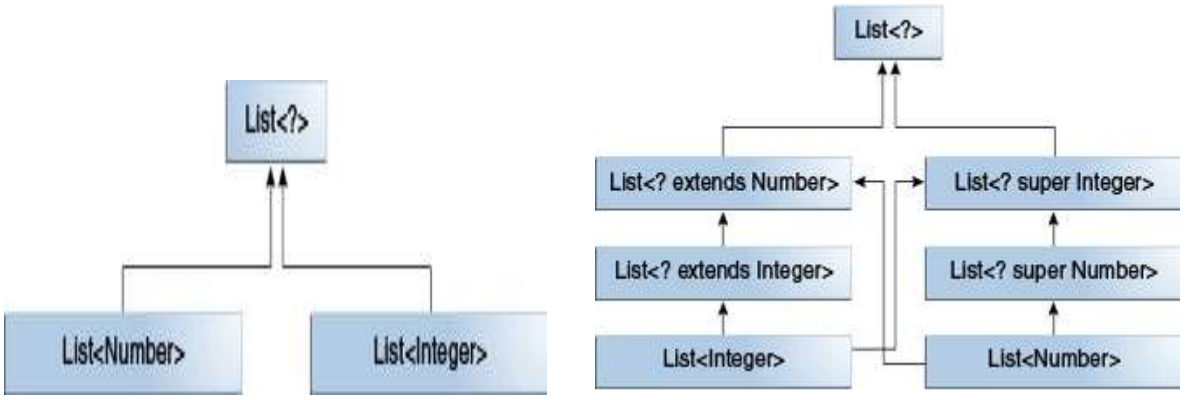
## *Lower Bounded Wildcard.*

Digamos que queremos definir un método que inserte objetos *Integer* en un *List*. Para mayor flexibilidad queremos que ese método pueda trabajar con cualquier tipo de lista que permita contener *Integer*, ya sea *List<Integer>*, *List<Number>* y *List<Object>*. Lo podemos conseguir definiendo *List<? super Integer>* que se conoce como *Lower Bounded Wildcard*.

Las clases genéricas no tienen relación alguna aunque sus tipos los tengan, pero usando *wildcards* podemos crearlas.

```
1    List<? extends Integer> intList = new ArrayList<>();
2    List<? extends Number>  numList = intList;
```

WildcardList.java



## Reglas uso

Uno de las mayores confusiones al usar generics es cuando usar *upper bounded wildcards* o cuando usar *lower bounded wildcards*. Podemos usar las siguientes reglas:

- Una variable *generic* que se usa como fuente de datos (*in*), por ejemplo *src* en
    - *<U> copy(List<? extends U> src, List<? super U> dest)* se define usando ***upper bounded wildcard*** con la palabra clave *extends*. De modo que la lista del parámetro *src* pueda ser una lista de un tipo *U* o de un subtipo de *U*.
- Un variable *generic* que se usa como destino de datos (*out*), por ejemplo *dest* en
    - *<U> copy(List<? extends U> src, List<? super U> dest)* se define usando ***lower bounded wildcard*** con la palabra clave *super*. De modo que la lista del parámetro *dest* pueda ser una lista de un tipo *U* o de un supertipo de *U*.
- En caso de que la variable pueda ser usada mediante métodos definidos en la clase *Object* se recomienda usar un ***unbounded wildcard (?)***.
- En caso de que la variable se necesite usar como fuente de datos y como destino (*in* y *out*) **no usar *wildcard***.

**Los *generics* son un mecanismo para proporcionar comprobaciones en tiempo de compilación**, sin embargo, el compilador aplica *type erasure* que implica:

- Reemplazar todos los tipos con sus *bounds* o por Object si son *unbounded*.
- Insertar *casts* para preservar el tipado seguro.

```java
// TypeErasure to Object
public class Node<T> {

  private T data;
  private Node<T> next;

  public Node(T data, Node<T> next) }
    this.data = data;
    this.next = next;
  }

  public T getData() { return data; }
  // ...
}

// Node type erased
public class Node {

  private Object data;
  private Node next;

  public Node(Object data, Node next) {
    this.data = data;
    this.next = next;
  }

  public Object getData() { return data; }
  // ...
}

// TypeErasure to Comparable
public class Node<T extends Comparable<T>> {

  private T data;
  private Node<T> next;

  public Node(T data, Node<T> next) {
    this.data = data;
    this.next = next;
  }

  public T getData() { return data; }
  // ...
}
```

```
48  // Node type erased
49  public class Node {
50
51    private Comparable data;
52    private Node next;
53
54    public Node(Comparable data, Node next) {
55      this.data = data;
56      this.next = next;
57    }
58
59    public Comparable getData() { return data; }
      // ...
    }
```

TypeErasure.java

## ¿Qué es el Type erasure?

En compilación se verifican los tipos de los parámetros y argumentos genéricos.

Después se pierde toda la información sobre el tipo. Este procedimiento se lama "Type erasure".

Un tipo *non reifiable (no verificable)* son aquellos cuya información de tipo ha sido eliminada en tiempo de compilación por el *type erasure*, para la JVM no hay ninguna diferencia en tiempo de ejecución entre *List<String>* y *List<Number>*. No se crean nuevas clases para los tipos parametrizados de modo que no hay ninguna penalización en tiempo de ejecución. Una clase genérica al compilarla se transforma aplicando *type erasure*.

## Restricciones

Los *generics* tiene algunas restricciones:

- No se pueden instanciar tipos genéricos con tipos primitivos.
- No se pueden crear instancias de los parámetros de tipo.
- No se pueden declarar campos *static* cuyos tipos son parámetros de tipo.
- No se pueden usar *casts* o *instanceof* con tipos parametrizados.
- No se pueden crear arrays de tipos parametrizados.
- No se pueden crear, capturar o lanzar tipos parametrizados que extiendan de *Throwable*.
- No se puede sobrecargar un método que tenga la misma firma que otro después del *type erasure*.

## Ejemplo

Cuando operamos con distintos tipos de datos en los métodos es necesario especificar el tipo de datos que vamos a recibir. De esta manera, si no utilizamos una clase genérica tendríamos que construir el siguiente proyecto

```
package claseSinGenerica;

public class ClaseSinGenerica {

    public void classTypeInteger(Integer t) {
        System.out.println("El tipo de T es " + t);
        System.out.println("El tipo de T es " + t.getClass());
    }

    public void classTypeString(String t) {
        System.out.println("El tipo de T es " + t);
        System.out.println("El tipo de T es " + t.getClass());
    }

    public void classTypeDouble(Double t) {
        System.out.println("El tipo de T es " + t);
        System.out.println("El tipo de T es " + t.getClass());
    }

}
```

```
package claseSinGenerica;

public class MainClaseSinGenerico {

    public static void main(String[] args) {
        // Creamos una instancia de ClaseGenerica para Integer.
        System.out.println("Primera llamada");
        ClaseSinGenerica intObj = new ClaseSinGenerica();
        intObj.classTypeInteger(88);

        System.out.println("Segunda llamada");
        // Creamos una instancia de ClaseGenerica para String.
        ClaseSinGenerica strObj = new ClaseSinGenerica();
```

```
        strObj.classTypeString("Paso Una cadena");
        System.out.println("Tercera llamada");
        // Creamos una instancia de ClaseGenerica para Double.
        ClaseSinGenerica doubleObj = new ClaseSinGenerica();
        doubleObj.classTypeDouble(3.141617);
    }
}
```

```
package claseSinGenerica.claseConGenerica;

public class MainClaseGenerico {

      public static void main(String[] args) {
          // Creamos una instancia de ClaseGenerica para Integer.
          System.out.println("Primera llamada");
          ClaseGenerica<Integer> intObj = new ClaseGenerica();
          intObj.classType(88);

          System.out.println("Segunda llamada");
          // Creamos una instancia de ClaseGenerica para String.
          ClaseGenerica<String> strObj = new ClaseGenerica();
          strObj.classType("Paso Una cadena");

          System.out.println("Tercera llamada");
          // Creamos una instancia de ClaseGenerica para Double.
          ClaseGenerica<Double> doubleObj = new ClaseGenerica();
          doubleObj.classType(3.14);

      }

}
```

```
package claseSinGenerica.claseConGenerica;

class ClaseGenerica<T> {

    void classType(T d) {
        System.out.println("El tipo de T es " + d);
        System.out.println("El tipo de T es " + d.getClass());
    }

}
```

El resultado será

1. El tipo de T es java.lang.Integer
2. El tipo de T es java.lang.String
3. El tipo de T es java.lang.Double

## Resumen

| Generic Term | Meaning |
|---|---|
| `Set<E>` | Generic Type , E is called formal parameter |
| `Set<Integer>` | Parametrized type , `Integer` is actual parameter here |
| `<T extends Comparable>` | Bounded type parameter |
| `<T super Comparable>` | Bounded type parameter |
| `Set<?>` | Unbounded wildcard |
| `<? extends T>` | Bounded wildcard type |
| `<? Super T>` | Bounded wildcards |
| Set | Raw type |
| <T extends Comparable<T>> | Recursive type bound |

## Preguntas comunes

**1. ¿Qué son los genéricos en Java? ¿Cuáles son los beneficios de usar los genéricos?**

Aquellos con experiencia en desarrollo en Java 1.4 o anterior saben lo inconveniente que es almacenar objetos en colecciones y realizar conversiones de tipos antes de usarlos. Los genéricos evitan que eso suceda. **Proporciona seguridad de tipo en tiempo de compilación, asegurando que solo pueda colocar objetos del tipo correcto en la colección, evitando ClassCastException en tiempo de ejecución.**

**2. ¿Cómo funcionan los genéricos de Java? ¿Qué es el borrado de tipo?**

**Los genéricos se logran mediante el borrado de tipo. El compilador borra toda la información relacionada con el tipo en tiempo de compilación, por lo que no hay información relacionada con el tipo en tiempo de ejecución.** Por ejemplo, List <String> está representado por una sola List en tiempo de ejecución. El propósito de esto es garantizar la compatibilidad con la biblioteca de clases binarias desarrollada en versiones anteriores a Java 5. No puede acceder a los parámetros de tipo en tiempo de ejecución porque el compilador ha convertido el tipo genérico.

**3. ¿Cuáles son los comodines calificados y no calificados en genéricos?**

Los comodines calificados restringen el tipo.

- Hay **2 comodines calificados**:
    - uno es <? Extiende T>, que establece el límite superior del tipo al garantizar que el tipo debe ser una subclase de T,
    - y el otro es <? Super T>, lo que garantiza que el tipo debe ser el padre de T Clase para establecer el límite inferior del tipo. El tipo genérico debe inicializarse con el tipo dentro del límite; de lo contrario, se producirá un error de compilación.
- Por otro lado, <?> Representa un comodín **no calificado**, porque <?> Se puede reemplazar por cualquier tipo.

**4.¿Cuál es la diferencia entre List <? Extend T> y List <? Super T>?**

Relacionado con la pregunta anterior. Equivale a preguntar qué son comodines calificados y comodines no calificados. Estas dos declaraciones de Lista son ejemplos de comodines calificados: Lista <? Extiende T> puede aceptar cualquier Lista heredada de T, y Lista <? Super T> puede aceptar cualquier Lista compuesta de cualquier clase padre de T.

Por ejemplo, Lista <? Extiende Número> puede aceptar Lista <Entero o Lista <Flotante. Se puede encontrar más información en los enlaces que aparecen en este párrafo.

**5. ¿Escribir un programa genérico para implementar el almacenamiento en caché de LRU?**

LinkedHashMap se puede usar para implementar un caché LRU de tamaño fijo. Cuando el caché LRU está lleno, eliminará el par clave-valor más antiguo del caché. LinkedHashMap proporciona un método llamado removeEldestEntry (), que será llamado por put () y putAll () para eliminar el par clave-valor más antiguo..

6. ¿Se puede pasar la Lista a un método que acepte los parámetros de la Lista?

Para cualquier persona que no esté familiarizada con los genéricos, este tema de genéricos de Java parece confuso, porque a primera vista, String es un tipo de objeto, por lo que List debe usarse en Donde se necesita List, pero no lo es. Hacerlo provocará errores de compilación. Si lo piensa más, encontrará que Java tiene sentido hacer esto, porque List puede almacenar cualquier tipo de objeto, incluyendo String, Integer, etc., mientras que List solo se puede usar para almacenar Strings.

```
List<Object> objectList;
List<String> stringList;
objectList = stringList;  //compilation error incompatible types
```

**7. ¿Se pueden usar genéricos en Array?**

*La matriz en realidad no admite genéricos*, esta es la razón por la cual Joshua Bloch recomienda usar List en lugar de Array en el libro Effective Java, porque List puede proporcionar garantías de seguridad de tipo de tiempo de compilación, pero Array no puede.