

# Genéricos en JAVA

# Introducción

- En JAVA se denomina “Genéricos” a la capacidad del lenguaje de **definir** y **usar tipos** (clases e interfaces) y **métodos genéricos**.
- Los **tipos genéricos** difieren de los “regulares” en que contienen **tipos de datos** como **parámetros formales**.
- Los **tipos** y los **métodos genéricos** se incorporan en JAVA 5.0 para proveer **chequeo de tipos en compilación en colecciones**.
- No pueden declararse genéricos los tipos enumerativos, las clases anónimas y subclasses de excepciones.

```
public class LinkedList <E> extends AbstractSequentialList <E> implements List<E>, Queue<E>, Cloneable, Serializable
```

**LinkedList es un tipo Genérico**

**E es un parámetro formal que denota un tipo de dato**

**Los elementos que se almacenan en la lista encadenada son del tipo desconocido E**

Con tipos genéricos podemos definir: `LinkedList<String>` y `LinkedList<Integer>`

- Una **clase genérica** tiene el **mismo comportamiento** para todos sus posibles tipos de parámetros.
- Los **tipos parametrizados** se forman a partir de los **tipos genéricos** al asignarle **tipos reales a los parámetros formales**:

`LinkedList<E>`, `Comparator<T>`      **Tipos Genéricos**

`LinkedList<String> listaStr;`      **Lista de Strings**

`LinkedList<Integer> listaInt = new LinkedList<Integer>();`      **Lista de Enteros**

`Comparator<String> compara;`      **Comparador de Strings**

**Tipos  
Parametrizados**

- El **framework de colecciones** del paquete **java.util** es **genérico** a partir de Java 5.0

# ¿Qué problemas resuelven los “Genéricos”?

En colecciones se **evitan los errores en ejecución** causados por el uso de *casting*.

```
List list = new ArrayList(); // Lista que contiene Strings
```

```
list.add("abc");  
list.add(new Integer(5));
```

El compilador devuelve una advertencia “unchecked call”

**Sin genéricos**

```
for(Object obj : list){
```

```
String str=(String) obj;
```

```
}
```

El casting dispara el siguiente error en ejecución: “ClassCastException”

Los **Genéricos** ofrecen una **mejora para el sistema de tipos**:

- permite operar sobre objetos de múltiples tipos.
- provee **seguridad** en compilación pudiendo detectar *errores* en compilación.

Los programas que usan genéricos son **seguros**, reusables, es un **código limpio**.

La **inserción errónea genera un mensaje de error en compilación** que indica exactamente qué es lo que está mal.

```
List<String> list1 = new ArrayList<>();
```

```
list1.add("abc");
```

```
list1.add(new Integer(5)); // error de compilación
```

```
//“error: incompatible types: Integer cannot be converted”
```

```
for(String str : list1){ ← No es necesario hacer casting
```

```
System.out.print(str);
```

```
}
```

**Con genéricos**

# Tipos Genéricos y Tipos Parametrizados

Un **tipo genérico** es un **tipo de datos con parámetros formales que denotan tipos de datos**.

Un **tipo parametrizado** es una **instanciación de un tipo genérico con argumentos que son tipos reales**.

```
public interface Collection<E> extends Iterable<E> {  
    int size();  
    boolean isEmpty();  
    boolean contains(Object o);  
    Iterator<E> iterator();  
    Object[] toArray();  
    <T> T[] toArray(T[] a);  
    boolean add(E e);  
    boolean remove(Object o);  
    boolean containsAll(Collection<?> c);  
    boolean addAll(Collection<? extends E> c);  
    boolean removeAll(Collection<?> c);  
    boolean retainAll(Collection<?> c);  
    void clear();  
    boolean equals(Object o);  
    int hashCode();  
}
```

## INTERFACE GENÉRICA

Un **tipo genérico** es un tipo de datos que **tiene uno o más parámetros formales** que representan tipos de datos.

Cuando el tipo genérico es instanciado o declarado, se reemplazan los parámetros formales por argumentos que representan tipos reales.

La **interface Collection** tiene un parámetro formal **E** que indica un tipo de datos. En la declaración de una colección específica, **E** es reemplazado por un tipo real.

La **instanciación de un tipo genérico** se denomina **tipo parametrizado**.

```
Collection <String> col=new LinkedList<String>();  
List <String> list= new ArrayList<>(); // a partir de JAVA 7  
Collection <? extends Number> col=new LinkedList<Integer>();
```

## TIPO PARAMETRIZADO

# Declaración de tipos Genéricos

En la definición de los **tipos genéricos** la sección correspondiente a los parámetros continúa al nombre de la clase o interface. Es una lista separada por comas y delimitada por los símbolos <>.

```
package genericos.definicion;  
public class ParOrdenado <X,Y> {  
    private X a;  
    private Y b;  
    public ParOrdenado (X a, Y b){  
        this.a=a;  
        this.b=b;  
    }  
    public X getA() { return a; }  
    public void setA(X a) { this.a = a; }  
    public Y getB() { return b; }  
    public void setB(Y b) { this.b = b; }  
}
```

El alcance de los identificadores **X** e **Y** es toda la clase **ParOrdenado**.

En el ejemplo, **X** e **Y** son usados en la declaración de variables de instancia y como argumentos y tipos de retorno de los métodos de instancia.

Los parámetros formales (tipos de datos) pueden declararse con cotas. Las cotas proveen acceso a métodos del tipo definido en el parámetro formal (que es desconocido) .

En el ejemplo de la clase **ParOrdenado** no invocamos a ningún método sobre los tipos desconocidos X e Y, es por esta razón que los 2 tipos son sin cotas.

# Tipos Parametrizados Concretos

Para usar un **tipo genérico** se deben especificar los **argumentos** que reemplazarán a los **parámetros formales**.

Los argumentos son referencias a **tipos concretos** como String, Long, Date, etc.

```
package genericos.definicion;
import java.awt.Color;
public class TestParOrdenado {
public static void main(String[] args){
```

```
    ParOrdenado<String, Long> par = new ParOrdenado<>("hola", 23L);
```

```
    System.out.println("(" + par.getA() + ", " + par.getB() + ")");
```

```
    ParOrdenado<String, Color> nombreColor = new ParOrdenado<>("Rojo", Color.RED);
```

```
    System.out.println("(" + nombreColor.getA() + ", " + nombreColor.getB() + ")");
```


```
    ParOrdenado<Double, Double> coordenadas = new ParOrdenado<>(17.3, 42.8);
```

```
    System.out.println("(" + coordenadas.getA() + ", " + coordenadas.getB() + ")"); (hola, 23)
```

```
}
```

```
}
```

Autoboxing: el 23 es  
automáticamente convertido  
a Long



(Rojo, java.awt.Color[r=255,g=0,b=0])

(17.3, 42.8)

La **instanciación** de ParOrdenado<String, Long>, ParOrdenado<String, Color>, ParOrdenado<Double, Double> son **tipos parametrizados concretos** y se usan como un tipo regular. Podemos usar **tipos parametrizados** como **argumentos de métodos**, para **declarar variables** y en la expresión **new** para crear un objeto.

# Tipos Parametrizados con Comodines

Son **instanciaciones de tipos genéricos** que contienen al menos un comodín (?) como tipo de argumento. Un comodín es una construcción sintáctica “?” que denota la familia de “todos los tipos”.

**No son tipos concretos**, no pueden usarse en la sentencia new.

```
static int cantElemEnComun(Set<?> s1, Set<?> s2) {  
    int result = 0;  
    for (Object o1 : s1)  
        if (s2.contains(o1)) result++;  
    return result;  
}
```

**Tipos parametrizados comodines sin cota**

El ? indica un conjunto de “algún tipo desconocido”.

Es el **conjunto parametrizado más general**, capaz de contener cualquier tipo de elemento. No interesa cuál es el parámetro del tipo real. Puede ser Set<String>, Set<A>, Set<Long>, etc.

**Set<?>** representa la flia de todos los conjuntos de “cualquier tipo”. No nos interesa el tipo real, es un conjunto que puede contener cualquier tipo.

Se pueden usar para **declarar variables o parámetros de métodos**, como s1 y s2, pero no en una sentencia **new** para crear objetos.

Como no sabemos nada del tipo de los elementos del Set, **solo se puede leer y tratar los objetos leídos como instancias de Object**.

**Los tipos parametrizados con comodines sin cota son útiles en situaciones en las que no es necesario conocer nada sobre el tipo del argumento** ¿Cuál es la diferencia entre Set<?> y Set?

# Tipos Parametrizados con Comodines

Un comodín con una cota superior “? extends T” es la familia de todos los tipos que son **subtipos** de T. T es la **cota superior**.

Un comodín con una cota inferior “? super T” es la familia de todos los tipos que son **supertipos** de T. T es la **cota inferior**.

Los **tipos parametrizados con comodines acotados** son útiles en situaciones en las que es necesario contar con un **conocimiento parcial sobre el tipo de argumento** de los tipos parametrizados.

```
List<? extends Number> l;
```

La familia de todos los tipos de listas cuyos elementos son **subtipos** de Number.

```
Comparable<? super String> s;
```

La familia de todas las instancias de la interface Comparable para tipos que son **supertipos** de String.

```
public final class Byte extends Number implements Comparable<Byte>
public final class Double extends Number implements Comparable<Double>
public final class Float extends Number implements Comparable<Float>
public final class Integer extends Number implements Comparable<Integer>
public final class Long extends Number implements Comparable<Long>
public final class Short extends Number implements Comparable<Short>
```

**Subtipos de Number**

```
String, Object, CharSequence, Serializable y Comparable<String>
```

**Supertipos de String**



# Tipos Parametrizados

## Ejemplos

Suma de los números de una lista de números:

```
public static double sum(List<Number> list){  
    double sum = 0;  
    for(Number n : list){  
        sum += n.doubleValue();  
    }  
    return sum;  
}
```

Los **tipos parametrizados concretos** **son invariantes** por lo tanto List<Double> y List<Integer> no están relacionados con List<Number>, no tienen relación de subtipo ni supertipo y el método sum() no se puede usar.

Los **tipos parametrizados con comodines acotados** ofrecen mayor **flexibilidad** que los tipos invariantes.

```
public static double sum(List<? extends Number> list){  
    double sum = 0;  
    for(Number n : list){  
        sum += n.doubleValue();  
    }  
    return sum;  
}
```

```
public static void main(String[] args) {  
    List<Integer> ints = new ArrayList<>();  
    ints.add(3); ints.add(5); ints.add(10);  
    double sum = sum(ints);  
    System.out.println("Suma de ints="+sum);  
}
```

# Tipos Parametrizados con Comodines

## Resumen

Un tipo parametrizado con comodín no es un tipo concreto.

Pueden declararse variables del tipo parametrizado con comodín, pero no pueden crearse objetos con el operador **new**. En ese sentido son similares a las interfaces.

Las variables de tipo parametrizado con comodín hacen referencia a un objeto perteneciente a la familia de tipos que el tipo parametrizado comodín denota.

```
List<?> col= new ArrayList<String>();
```

**Lista de algún tipo**

```
List<? extends Number> lista=new ArrayList<Long>();
```

**Lista de subtipos de Number**

```
ParOrdenado<String, ?> par=new ParOrdenado<String, String>();
```

**Par ordenada con abscisa de algún tipo**

```
List<? extends Number> l = new ArrayList<String>();
```

**ERROR!!!!**

String no es subtipo de Number y consecuentemente **ArrayList<String>** no pertenece a la familia de tipos denotados por **List<? extends Number>**.

**List<?>** denota una lista de elementos de algún tipo, desconocido. Es una lista de “sólo lectura”.

# Reglas de subtipos para genéricos

Los **tipos parametrizados** forman una jerarquía de tipos basada en el tipo base NO en el tipo de los argumentos. Los **tipos parametrizados son invariantes**.

```
public interface List<E> extends Collection<E> {}  
public class ArrayList<E> extends AbstractList<E> implements List<E> {}
```

```
List<Integer> listita = new ArrayList<>();
```

```
List<Integer> li = listita;
```

```
Collection<Integer> c = listita;
```

```
ArrayList<Number> n = listita;
```

```
List<Object> o = listita;
```

```
List li = listita;
```

¿Cuáles asignaciones cumplen las reglas de subtipo?

Un `ArrayList<Integer>` es un `List<Integer>`, un `Collection<Integer>` y un `List`, pero NO es un `ArrayList<Number>` ni un `List<Object>`.

```
List<Integer> li = new ArrayList<>();
```

```
li.add(123);
```

```
List<Number> lo = li;
```

```
Number nro = lo.get(0);
```

```
lo.add(3.14);
```

```
Integer i = li.get(1);
```

¿Es válido este código?

# Reglas de subtipos para genéricos

Los **tipos parametrizados** forman una jerarquía de tipos basada en el tipo base NO en el tipo de los argumentos. Los **tipos parametrizados son invariantes**.

```
public interface List<E> extends Collection<E> {}
```

```
public class ArrayList<E> extends AbstractList<E> implements List<E> {}
```

```
List<Integer> listita = new ArrayList<>();
```

```
List<Integer> li = listita;
```

```
Collection<Integer> c = listita;
```

```
ArrayList<Number> n = listita;
```

```
List<Object> o = listita;
```

```
List li = listita;
```

Un `ArrayList<Integer>` es un `List<Integer>`, un `Collection<Integer>` y un `List`, pero NO es un `ArrayList<Number>` ni un `List<Object>`.

`List<Integer>` no es un subtipo de `List<Number>`

```
List<Integer> li = new ArrayList<>();
```

```
li.add(123);
```

```
List<Number> lo = li;
```

```
Number nro = lo.get(0);
```

```
lo.add(3.14);
```

```
Integer i = li.get(1);
```

No compila `List<Number> lo=li;` NO ES POSIBLE CONVERTIR DE `List<Integer>` a `List<Number>`

Si asumimos que compila:

- podríamos recuperar elementos de la lista como `Number` en vez de como `Integer`:

`Number nro = lo.get(0);`

- podríamos agregar un objeto `Double`: `lo.add(3.14);`

- la línea `li.get(1);` daría error de *casting*, porque no puedo *castear* un `Double` a un `Integer`

# Tipos Parametrizados con Comodines Acotados - Ejemplos

```
interface Collection<E> {  
    public boolean addAll(Collection<? extends E> c);  
}
```

`public interface List<E> extends Collection<E> {}`  
**addAll()** agrega todos los elementos de una colección en otra colección

"? extends E": está permitido agregar a una colección de tipos E elementos de otra colección que son subtipo de E.

El parámetro **c** de **addAll()** produce elementos para la colección, por lo tanto los elementos de **c** deben ser subtipos de E.

```
package genericos;  
import java.util.ArrayList;  
import java.util.Arrays;  
import java.util.List;  
public class TestComodinConExtends {  
    public static void main(String[] args) {  
        List<Number> nums = new ArrayList<>();  
        List<Integer> ints = Arrays.asList(1, 2);  
        List<Double> dbls = Arrays.asList(2.78, 3.14);  
        nums.addAll(ints);  
        nums.addAll(dbls);  
    }  
}
```

- **nums** es de tipo **List<Number>** que es subtipo de **Collection<Number>**
- **ints** es de tipo **List<Integer>** que es subtipo de **Collection<? extends Number>**
- **dbls** es de tipo **List<Double>** que es subtipo de **Collection<? extends Number>**.
- **E** toma el valor **Number**.

Si el parámetro de **addAll()** estuviese escrito sin comodines es decir **Collection<E>**: ¿Podría agregarse a **nums** una lista de enteros y de números decimales?

NO!! Solamente estaría permitido agregar listas que estuviesen explícitamente declaradas como listas de **Number**.

# ¿Qué está disponible a través de variables de tipo parametrizado con comodín?

```
public static void printList(PrintStream out, List<?> lista) {  
    // lista.add("hola");  
    for(int i=0, n=lista.size(); i < n; i++) {  
        if (i > 0) out.println(", ");  
        Object o = lista.get(i);  
        out.print(o.toString());  
    }  
}
```

El método **printList()** imprime todos los elementos de la lista pasada como parámetro.

```
interface List<E> {  
    boolean add (E element);  
    E get(int index)  
}
```

- El método **add(E e)** de List<E> acepta un argumento del tipo especificado por el parámetro **E**. En nuestro ejemplo el tipo es “?” (desconocido) por lo tanto el compilador no puede asegurar que el objeto pasado como parámetro al método add() es del tipo esperado por el método. **Por lo tanto, List<?> es de sólo lectura, no es posible invocar a los métodos add( ), set() y addAll( ) de List. Código Seguro.**
- El método **E get(int)** de List<E> devuelve un valor que es del mismo tipo que el parámetro **E**. En nuestro ejemplo el tipo es “?” (desconocido) por lo tanto el método get() puede ser invocado y el resultado puede asignarse a un variable de tipo Object (sabemos que será un objeto).
- **Conclusión:** los tipos parametrizados con comodines son de solo lectura.
- Si en lugar de List<?> usamos List, ¿ocurre lo mismo? ¿cuál es la diferencia?

# ¿Qué está disponible a través de variables de tipo parametrizado comodín?

```
List<Integer> li = new ArrayList<>();  
li.add(123);  
List<? extends Number> lo = li;  
Number nro = lo.get(0);  
lo.add(3.14);  
Integer i = li.get(1);
```

→ ¿Se puede hacer?

- Se puede asignar `li` a `lo` porque `li` es de tipo `List<Integer>` y es subtipo de `List<? extends Number>`.
- No se puede agregar un `Double` a `lo` (`List<? extends Number>`) dado que podría ser una lista de algún otro subtipo de `Number`.

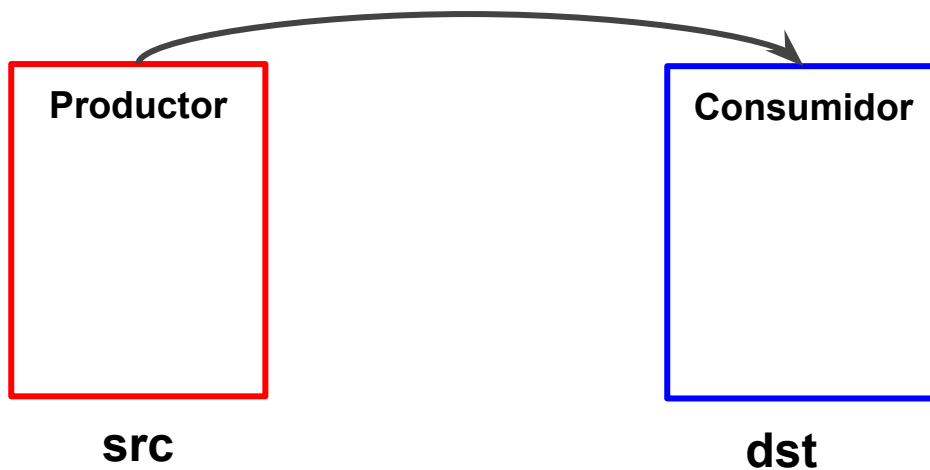
En general: si una variable referencia a una colección que declara contener elementos de tipo `<? extends E>` es posible recuperar elementos de la estructura pero **NO** es posible agregar elementos a la estructura. **Es una variable de solo lectura.**

# Tipos Parametrizados con Comodines Acotados - Ejemplos

```
public class Collections{  
    public static <T> void copy( List<? super T> dst , List<? extends T> src) {  
        for (int i = 0; i < src.size(); i++) { dst.set(i, src.get(i)); }  
    }  
}
```

El método **copy()** de la clase **Collections** copia elementos desde una **lista fuente (src)** a una **destino (dst)**

La **lista destino** tiene que ser capaz de **guardar los elementos de la lista fuente**.



Paradigma productor-consumidor:

productor-extends, consumidor-super

- src **produce** para dst y dst **consume** de src.
- Los elementos de src deben ser subtipos de los de dst.
- Los elementos de dst deben ser supertipos de los de src.

**List<? extends T>** (solo lectura)

**List<? super T>**

**copy:** es un método genérico, acepta argumentos de tipo List<? super T> y List<? extends T>, devuelve void y se aplica a cualquier tipo T.

**List<? super T> dst:** la lista destino puede contener elementos de cualquier tipo que sea supertipo de T.

**List<? extends T> src:** la lista fuente puede contener elementos de cualquier tipo que sea subtipo de T.



# Tipos Parametrizados con Comodines Acotados - Ejemplos

```
public class Collections{  
    public static <T> void copy( List<? super T> dst , List<? extends T> src) {  
        for (int i = 0; i < src.size(); i++) { dst.set(i, src.get(i)); }  
    }  
}
```

Object



T



Integer

```
package genericos;  
import java.util.*;  
public class TestMetodosGenericos {  
    public static void main(String[] args) {  
        List<Object> objs = Arrays.asList(2, 3.14, "hola");  
        List<Integer> ints = Arrays.asList(5, 6);  
        Collections.copy(objs, ints);  
    }  
    Collections.<Number>copy(objs, ints);
```

Integer es subtipo de T y Object es supertipo de T. El compilador elige dentro de los posibles, por ejemplo **Number**

```
List<String> strs = Arrays.asList("2", "hola", "3.14" );  
List<Integer> ints2 = Arrays.asList(5, 6);  
Collections.copy(strs, ints2);
```

**NO ES APLICABLE**

[5, 6, hola]

Productor  
(src)

ints

Consumidor  
(dst)

objs

```
public static <E> Set<E> union(Set<? extends E> s1, Set<? extends E> s2)
```

S1 y s2 son ambos productores, sus elementos se guardarán en el conjunto receptor del método union()

# Métodos Genéricos

De la misma manera que los tipos (clases e interfaces) los **métodos** pueden definirse **genéricos**, es decir pueden ser parametrizados por uno o más tipos de datos.

```
public static <T> int countOccurrences(T[] list, T itemToCount) {  
    int count = 0;  
    if (itemToCount == null) {  
        for (T listItem : list)  
            if (listItem == null)  
                count++;  
    } else {  
        for (T listItem : list)  
            if (itemToCount.equals(listItem))  
                count++;  
    }  
    return count;  
}
```

**countOccurrences()** cuenta la cantidad de **ocurrencias** del elemento **itemToCount** en el arreglo genérico **list** y se aplica a **cualquier tipo T**

Declarar métodos genéricos es similar a declarar tipos genéricos pero el alcance del tipo como parámetro está limitado al método.

Los métodos genéricos expresan dependencias entre los tipos de los argumentos y/o el tipo de retorno del método.

```
class Collections {  
    public static <T> void copy(List<T> dest, List<? extends T> src) { }  
}
```

Cuando se usa un método genérico no hay una mención explícita al tipo que sustituirá al tipo del parámetro formal. El compilador infiere el tipo a partir de los parámetros reales. **Inferencia de tipos**

**En nuestro caso como list es un arreglo de Number, entonces T es un Number.**

**countOccurrences (arrNumber,10);**

```
Number arrNumber[]=new Number[5];  
arrNumber[0]=10.9; arrNumber[1]=5L;  
arrNumber[2]=15; arrNumber[3]=10;  
arrNumber[4]=10;
```

# Métodos Genéricos

```
public static <T extends Comparable <? superT>> T max(Collection<? extends T> coll)
```

**max** devuelve el **mayor valor de una colección de elementos de un tipo desconocido T**, de acuerdo al orden natural de sus elementos.

Aquí el parámetro **coll produce** números que cumplen relaciones de orden.

Los **Comparables** y **Comparator** siempre son **consumidores**.

Los métodos genéricos se invocan de la forma usual, los argumentos que representan a los tipos concretos no necesitan ser explicitados, son **inferidos automáticamente**.

```
import java.util.*;
public class TestMetodos {
    public static void main(String[] args) {
        List<String> strList=new ArrayList<>();
        strList.add("hola");
        strList.add("chau");
        strList.add("hi");
        strList.add("bye");
        System.out.println(Collections.max(strList));
    }
}
```

<T extends Comparable <? superT>> T

Cualquier tipo T que sea comparable es decir que permita comparar objetos entre sí.

**El compilador invoca automáticamente al método max() usando como argumento el tipo String.**

**El compilador infiere automáticamente el tipo del argumento.**

# Tipos Parametrizados con Comodines

## Algunos tips

Si se usan correctamente los **tipos comodines** son **invisibles** para los usuarios de una clase.

Los **métodos aceptan los parámetros** que deben aceptar y **rechazan** los que deberían rechazar.

Si el usuario de una clase tiene que pensar en tipos de comodines, probablemente haya algo mal con la API.

# Ejemplos

```
public class TestListaParametrizada {  
    public static void main(String[] args) {  
        List<String> listaPalabras = new ArrayList<>();  
        //listaPalabras.add(args); // ERROR DE COMPILACIÓN!!!  
        for(String arg : args)  
            listaPalabras.add(arg);  
        String unaPalabra = listaPalabras.get(0); // CASTING AUTOMÁTICO!!!  
    }  
}
```

SIN TIPOS GENÉRICOS un error accidental en la inserción causaría una falla de EJECUCIÓN

Map es un tipo genérico del framework de colecciones con 2 parámetros que representan tipos de datos: uno representa el tipo de las claves y el otro el tipo del valor de cada clave.

**public interface Map<K,V> {}**

```
public class TestMapParametrizado{  
    public static void main(String[] args) {  
        Map<String,Integer> tabla = new HashMap<>();  
        for(int i=0; i < args.length; i++)  
            tabla.put(args[i], i);  
        int posicion =tabla.get("hola"); // CASTING AUTOMÁTICO!!!  
    }  
}
```

Operaciones de boxing y unboxing permiten convertir automáticamente de tipos primitivos a clases wrapper

java TestMapParametrizado chau hola adiós

# Ventajas de usar Genéricos

## Detección temprana de errores

El compilador puede realizar más chequeos de tipos. Los errores son detectados tempranamente y reportados por el compilador en forma de mensajes de error en lugar de ser detectados en ejecución mediante excepciones.

```
package genericos;
import java.util.Date;
import java.util.LinkedList;
public class TestErrores {
    public static void main(String[] args) {
        List<String> list= new LinkedList<String>();
        list.add("hola");
        list.add(new Date());
    }
}
```

Es una lista homogénea de strings

El compilador chequea que la lista sólo contenga strings. En otro caso, el compilador lo rechaza

Con **tipos no-parametrizados** (LinkedList) es posible agregar diferentes tipos de elementos a la colección. La compilación será exitosa.

```
List list= new LinkedList ();
list.add("hola");
list.add(new Date());
```

# Ventajas de usar Genéricos

## Detección temprana de errores (Continuación)

```
package genericos;
import java.util.*;
public class TestErrores {
    public static void main(String[] args) {
        List<String> list= new LinkedList<String>();
        list.add("hola");
        String str=list.get(0);
    }
}
```

Los elementos se recuperan sin realizar *casting*. Está garantizado que la lista contiene strings.

Con **tipos no-parametrizados** (LinkedList) no hay conocimiento ni garantías respecto del tipo de los elementos que se recuperan. Todos los métodos retornan referencias a **Object** que deben ser *downcasteadas* al tipo real del elemento a recuperar.

```
List list= new LinkedList();
list.add("hola");
list.add(new Date());
String str=(String)list.get(0);
```

El *casting* podría causar errores en ejecución, **ClassCastException**, en caso que el elemento recuperado no sea un string

# Ventajas de usar Genéricos

## Seguridad de Tipos

En JAVA se considera que un programa es **seguro respecto al tipado** si compila sin errores ni advertencias y en ejecución NO dispara ningún **ClassCastException**.

Un programa bien formado permite que el compilador realice suficientes **chequeos de tipos basados en información estática** y que no ocurran errores inesperados de tipos en ejecución. **ClassCastException** sería un error inesperado de tipos que se produce en ejecución sin ninguna expresión de casting visible en el código fuente.



# Interoperabilidad con código *legacy*

## ¿Cómo se implementan en JAVA los tipos genéricos?

Mediante la **técnica llamada “borrado” o *erasure***: el compilador usa la información de los tipos genéricos y tipos parametrizados para compilar y luego elimina la información del TIPO.

**List<Integer>, List<String> y List<String>** son traducidas a **List**. El bytecode es el mismo que el de **List**.

**Después de la traducción por “borrado” desaparece toda la información del TIPO.**

- **Simplicidad:** hay una única implementación de List, no una versión de cada tipo.
- **Compatibilidad:** la misma librería puede ser accedida tanto por genéricos como por no-genéricos.
- **Evolución Fácil:** no es necesario cambiar todo el código a genéricos, es posible evolucionar el código a genéricos actualizando de a un paquete, clase o método.

Podremos mantener versiones del código fuente de nuestras librerías que usen tipos genéricos y otras que usen raw types, es decir sin tipos genéricos. Es posible invocar a un método diseñado con tipos parametrizados con tipos sin parametrizar y viceversa.

**Compatibilidad de Migración.**

# Compatibilidad entre tipos parametrizados y raw types

Es compatible la asignación entre un raw type y todas las instanciaciones de un tipo genérico.

La asignación de un tipo parametrizado a un raw type está permitida; la asignación de un raw type a uno parametrizado produce advertencias en compilación **“conversiones no chequeadas/seguridad de tipos”**.

Es importante tener en cuenta que se rompe la seguridad de tipos genéricos.

```
List<Integer> li = new ArrayList<Integer>(); // tipo parametrizado
List lo=new ArrayList (); // raw type
li.add(123);
lo=li;
Object nro = lo.get(0);
lo.add("hola"); // inserción de un tipo no permitido no se detecta en compilación
Integer i = li.get(1);
```

La advertencia indica que el compilador desconoce si el ArrayList que se está asignando contiene o no Integer

Este código compila, pero en ejecución dispara un error de casting: **ClassCastException**