

**FACULTAD DE INGENIERÍA**

**ESCUELA ACADÉMICA DE INGIENIERÍA DE SISTEMAS**

**EXPERIENCIA CURRICULAR DE**

**METODOLOGIA DE LA PROGRAMACIÓN**

**TÍTULO**

**MANEJO DE DATOS GENÉRICOS EN JAVA**

**AUTORES**

BONILLA YPANAQUE, Julio Cesar

DAMIAN SUAREZ, Alexander Milton

HUAMANI VARGAS, Alejandro Edwin

PAREDES ALMONACID, Luis Antonio

YUPANQUI LOZANO, Juan Nelson

**PROFESOR**

ERIC GUSTAVO CORONEL CASTILLO

**LIMA, PERÚ**

**2017**

**ÍNDICE**

pág.

[INTRODUCCIÓN 3](#_Toc498588106)

[CAPÍTULO I 5](#_Toc498588106)

[1.1 Concepto 6](#_Toc498588108)

[1.2 Historia 8](#_Toc498588114)

[CAPÍTULO II 9](#_Toc498588123)

[2.1 Tipos de datos genericos 10](#_Toc498588125)

[2.2 Restricciones 14](#_Toc498588126)

[CONCLUSIONES 15](#_Toc498588148)

**INTRODUCCION**

Antes de Java 5 y de la existencia de genéricos, los programadores java que utilizaban colecciones, se enfrentaban a un temible peligro.

Ellos podían, por ejemplo, crear una lista de Strings como la siguiente:

List miLista = new ArrayList( );

miLista.add(“Hola”);

miLista.add(“Galaxia”);

miLista.add(new Integer(2001));

podían, incluso, hacer un casting al obtener los elementos de esa lista:

String s = (String) miLista.get(0);

String st = (String) miLista.get(1);

String str = (String) miLista.get(2);

¡Pero nada de eso evitaba que al ejecutarse el programa se produjera un error! ¿Por qué? ¡Porque el tercer elemento no es un String, sino un Integer, pero nada impedía añadirlo a la lista! Por suerte, los genéricos vinieron al rescate de estos desdichados programadores y nunca más se produjeron errores de este tipo.

Bueno, vale, puede que la historia no fuese exactamente así, pero nos sirve para explicar en qué consisten los genéricos y su función. Desde luego los genéricos no se utilizan sólo con las colecciones, pero es cierto que es en ellas donde te los vas a encontrar mayoritariamente.

Los genéricos permiten asignar parámetros a las clases, interfaces, métodos..., de forma que sólo admitan los tipos de objetos que tú quieras. **No** se pueden utilizar con primitivos, pero si con las clases que se corresponden con ellos. Por eso no puedes hacer un genérico tipo int, pero sí un Integer. Veamos el ejemplo anterior utilizando genéricos:

List <String> miLista = new ArrayList <String>();

//al poner <String> estamos indicando que la lista sólo acepta Strings

miLista.add(“Hola”); //un String, perfecto

miLista.add(“Galaxia”); //otro String, muy bien

miLista.add(new Integer(2001)); //¡error de compilación, no es un String!

Evidentemente, es mucho más sencillo solucionar un error de compilación que un error de ejecución. Además, no hace falta ningún casting para obtener los elementos de la lista:

String s = miLista.get(0);

String st = miLista.get(1);

//el compilador ya sabe que es un String, no hace falta casting.

Para conseguir esto, basta poner entre los signos < >, llamados diamante, el tipo al que queremos que pertenezcan los elementos: String, Integer o lo que sea. Además, para facilitar más las cosas, a partir de Java 7 la primera línea de este código se escribiría así:

List <String> miLista = new ArrayList<>();

Porque el compilador ya infiere de qué tipo son los elementos de la lista.

**CAPÍTULO I**

1. **concepto**

Los genéricos sirven para permitir que métodos o clases puedan operar con distintos tipos de objetos proporcionando seguridad en tiempo de compilación. Esto nos permite crear código más legible y mucho más robusto.

Supongamos las siguientes dos clases:

class A()

{

}

class B()

{

}

Imaginemos que queremos crear una colección que guarde instancias de la clase A. Podríamos hacerlo así:

ArrayList c = new ArrayList();

c.add(new A());

c.add(new A());

c.add(new A());

Sin embargo, esto no nos garantiza que la colección solo tenga elementos de ese tipo. Podemos añadir instancias de otras clases sin problema…

ArrayList c = new ArrayList();

c.add(new A());

c.add(new A());

c.add(new A());

c.add(new B());

c.add(new B());

Esto podría solucionarse añadiendo otra capa de abstracción que se encargue de controlar los elementos que vamos añadiendo, pero eso no garantiza necesariamente un 100% de fiabilidad. El programador, a la hora de recuperar un objeto de una colección, no puede saber con seguridad a que clase pertenece.

Fijémonos en este pequeño snippet:

ArrayList c = new ArrayList();

A clase\_a;

c.add(new A());

c.add(new B()); // Fijémonos que podemos añadir elementos de cualquier clase a la colección

clase\_a = (A) c.get(1); // El programador intenta recuperar elementos de la colección en la que cree que solo hay instancias de la clase (A)

¿Como usamos los genéricos?

Reconoceremos el uso de genéricos en una clase, interfaz o método cuando veamos un solo carácter en mayúsculas encerrado en el operador “diamond”. Por ejemplo:

Como ejemplo, veamos la declaración de la clase ArrayList:

public class ArrayList extends AbstractList implements List, RandomAccess, Cloneable, Serializable

El operador después de ArrayList significa que la clase está implementando genéricos. Si nos fijamos en sus herencias e implementaciones, veremos que AbstractList y List también usan genéricos.

Para usar una clase que implemente genéricos, solo deberemos inicializar y declarar esa clase usando el operador diamante con la clase deseada. ¿Confuso?, veamos un ejemplo.

ArrayList str = new ArrayList(); // Un ArrayList que solo permite Strings

Como podéis ver, usamos el operador diamante con la clase String de la siguiente forma: De esta forma, le decimos a Java que queremos un ArrayList que solo acepte instancias de la clase, si intentamos añadir otra cosa, tendremos un bonito error en tiempo de compilación.

Una vez entendida la funcionalidad de los genéricos, podemos decir que su sintaxis es similar (ignorando el operador diamante) a la de la declaración de parámetros de cualquier método, ya que una misma clase o método puede usar todos los genéricos que necesite. Veamos la declaración de la interfaz Map:

public interface Map

Como veis, utiliza dos genéricos. Esto es que, a la hora de inicializarlo o declararlo deberemos usar dos nombres de Clase en el interior del operador diamante. La funcionalidad, evidentemente, dependerá de la clase. Para que nos entendamos mediante la práctica, voy a poner un ejemplo de que clase que implementa la interfaz Map y, por tanto, usa dos genéricos. Esta clase es la clase HashMap.

Declaración de la clase, para que sepamos que hacemos:

public class HashMap extends AbstractMap implements Cloneable, Serializable

Ejemplo de clase que usa dos genéricos:

HashMap h = new HashMap();

h.put(new Integer(1), "uno");

h.put(new Integer(2), "dos");

h.put("tres", 3); // Esto dará un error, ya que "tres",3 no se corresponde con la declaración que hemos hecho

En el ejemplo se crea un HashMap (un array asociativo) que relaciona un objeto Integer con un String. En concreto, relaciono un número con su representación lexicográfica. Si intentamos crear una relación llave -> valor que no sea Integer -> String, tendremos un error de compilación, como se ve en la línea 4.

# **Historia**

Hace ya más de una década que en Java 5 se introdujeron los generics para dotar al lenguaje de una mejor comprobación de tipos en tiempo de compilación y al mismo tiempo eliminar los cast que hasta entonces eran necesarios al usar las colecciones. Dada la lentitud de adopción que suele haber en la plataforma Java en los grandes entornos empresariales puede que aún no los hayamos usado extensamente o tengamos alguna duda en su uso. Hay unos cuantos conceptos sobre los generics que son convenientes conocer.

Los generics fueron introducidos en la versión 5 de Java en 2004 junto con otras muchas novedades suponiendo en su historia una de las mayores modificaciones o al mismo nivel de las novedades introducidas con Java 8 más recientemente al lenguaje Java. Los generics 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 simplificando, reduciendo la repetición y aumentando la legibilidad el 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.

**CAPÍTULO II**

# **Los tipos genéricos de Java**

Los genéricos permiten usar tipos para parametrizar las clases, interfaces y métodos al definirlas. 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 tapado seguro.

Un tipo usando genéricos tiene el siguiente aspecto, por ejemplo usando una clase Boxcontenedor de una referencia a un tipo no determinado en la definición de la clase pero que lo será en su uso. 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.

Según las convenciones los nombres de los parámetros de tipo usados comúnmente son los siguientes:

E: elemento de una colección.

K: clave.

N: número.

T: tipo.

V: valor.

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 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.

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.

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

|  |
| --- |
|  |
|  |
|  |

La sintaxis completa de uso sería:

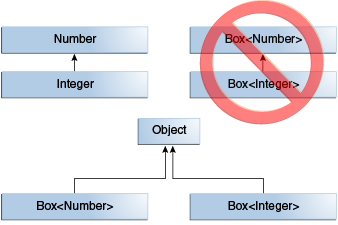
Aunque puede abreviarse ya que el compilador puede inferir los tipos:

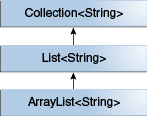
A veces querremos limitar los tipos que pueden ser usados empleando lo que se denomina bounded type. Con <U extends Number> el tipo U debe extender la clase Number.

|  |
| --- |
|  |
|  |
|  |

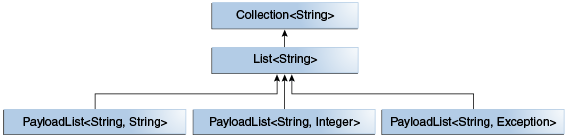
Una clase puede tener múltiples limitaciones, si una es una clase debe ser la primera y el resto de argumentos interfaces.

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).

[](https://picodotdev.github.io/blog-bitix/assets/images/posts/2016/138/generics-subtypeRelationship.gif)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 Doublesubtipos 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:

[](https://picodotdev.github.io/blog-bitix/assets/images/posts/2016/138/generics-sampleHierarchy.gif)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>.

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

[](https://picodotdev.github.io/blog-bitix/assets/images/posts/2016/138/generics-payloadListHierarchy.gif)

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

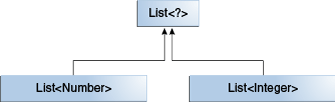
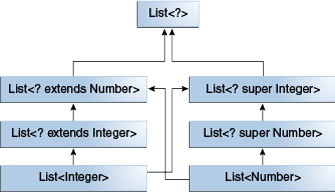
Se puede definir una lista de un tipo desconocido, List<?>, en casos en los que:

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.

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 wildcads podemos crearlas.

[](https://picodotdev.github.io/blog-bitix/assets/images/posts/2016/138/generics-listParent.gif)[](https://picodotdev.github.io/blog-bitix/assets/images/posts/2016/138/generics-wildcardSubtyping.gif)

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 supe tipo de U.

En caso de que la variable pueda ser usando 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 tapado seguro.

Generar métodos puente para preservar el polimorfismo en generics en los que son extendidos.

Un tipo non reifiable 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.

1. **Restricciones:**

Los generics tienen 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 tengan la misma firma que otro después del type erasure.

**Conclusiones**

1. los genéricos con java permite asignar parámetros a las clases, interfaces, etc. Que sirve para darle seguridad al programador que no habrá ningún problema al momento de compilar, son diferentes modos en cada categoría de java, sin utilizar eso.

2.-Los generics en un tiempo fueron el boom en 2004 en el lenguaje java, es mucha ayuda en la hora de programar ya que puede detectar errores de compilación y de cast dando una mayor facilidad de entender el código,

4.-Los generics siempre tendrán algunas restricciones al momento de su uso o de su ejecución, esas restricciones se podría decir que son necesarias para una buena ejecución g