Generics

https://www.digitalocean.com/community/tutorials/java-generics-example-method-class-interface

Outlines

- Generics in Java
- Java Generic Class
- Java Generic Type Convension
- Java Generic Method
- Java Generic Interface
- Java Generics Upper Bounded Wildcard

Generics in Java

```
List list = new ArrayList();
list.add("abc");
list.add(new Integer(5)); //OK

for(Object obj : list){
        //type casting leading to ClassCastException at runtime
    String str=(String) obj;
}
```

Above code compiles fine but throws ClassCastException at runtime because we are trying to cast Object in the list to String whereas one of the element is of type Integer. After Java 5, we use collection classes like below.

```
List<String> list1 = new ArrayList<String>(); // java 7 ? List<String> list1 = new ArrayList<>();
list1.add("abc");
//list1.add(new Integer(5)); //compiler error

for(String str : list1){
     //no type casting needed, avoids ClassCastException
}
```

```
public static void main(String[] args) {
    method1(new A());
    method1(new B());
}

static <T> void method1(T obj) {
    System.out.println(obj);
}

public class B {
    @Override
    public String toString() {
        return "B";
    }
}
```

Java Generic Class

```
public class GenericsTypeOld {

	private Object t;

	public Object get() {
		return t;
	}

	public void set(Object t) {
		this.t = t;
	}

	public static void main(String args[]){
		GenericsTypeOld type = new GenericsTypeOld();
		type.set("Pankaj");
		String str = (String) type.get(); //type casting, error prone and can cause ClassCastE
	}
}
```

```
public class GenericsType<T> {

	private T t;

	public T get(){
		return this.t;
	}

	public void set(T t1){
		this.t=t1;
	}

	public static void main(String args[]){
		GenericsType<String> type = new GenericsType<>();
		type.set("Pankaj"); //valid

		GenericsType type1 = new GenericsType(); //raw type
		type1.set("Pankaj"); //valid
		type1.set(10); //valid and autoboxing support
	}
}
```

Java Generic Type Convension

Java Generic Type Naming convention helps us understanding code easily and having a naming convention is one of the best practices of Java programming language. So generics also comes with its own naming conventions. Usually, type parameter names are single, uppercase letters to make it easily distinguishable from java variables. The most commonly used type parameter names are:

- E - Element (used extensively by the Java Collections Framework, for example ArrayList, Set etc.)
- K - Key (Used in Map)
- N - Number
- T - Type
- V - Value (Used in Map)
- S,U,V etc. - 2nd, 3rd, 4th types

Java Generic Method

Comparable interface is a great example of Generics in interfaces and it's written as:

```
package java.lang;
import java.util.*;

public interface Comparable<T> {
    public int compareTo(T o);
}
```

```
public class GenericsMethods {

        //Java Generic Method
        public static <T> boolean isEqual(GenericsType<T> g1, GenericsType<T> g2){
                return g1.get().equals(g2.get());
        }

        public static void main(String args[]){
                GenericsType<String> g1 = new GenericsType<>();
                g1.set("Pankaj");

                GenericsType<String> g2 = new GenericsType<>();
                g2.set("Pankaj");

                boolean isEqual = GenericsMethods.<String>isEqual(g1, g2);
                //above statement can be written simply as
                isEqual = GenericsMethods.isEqual(g1, g2);
                //This feature, known as type inference, allows you to invoke a generic method as an
                //Compiler will infer the type that is needed
        }
}
```

```
public class GenericMethodTest {
   // generic method printArray
   public static < E > void printArray( E[] inputArray ) {
      // Display array elements
      for(E element : inputArray) {
         System.out.printf("%s ", element);
      }
      System.out.println();
   }

   public static void main(String args[]) {
      // Create arrays of Integer, Double and Character
      Integer[] intArray = { 1, 2, 3, 4, 5 };
      Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };
      Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };

      System.out.println("Array integerArray contains:");
      printArray(intArray);   // pass an Integer array

      System.out.println("\nArray doubleArray contains:");
      printArray(doubleArray);   // pass a Double array

      System.out.println("\nArray characterArray contains:");
      printArray(charArray);   // pass a Character array
   }
}
```

Java Generic Interface

```java
//Generic interface definition
    interface DemoInterface<T1,T2>{
    T1 doSomeOperation(T1 t);
    T2 doReverseOperation(T2 t);
    }
//Class implementing generic interface
class DemoClass implements DemoInterface<String,Integer>{
    public Integer doSomeOperation(String t){
        //logic
    }
    public Integer doReverseOperation(String t){
        //logic
    }
}
```

```java
class GenericMethodDemo
{
// A Generic method
static <T> void printGeneric (T element)
    {
        System.out.println(element);
    }
   public static void main(String[] args)
        {
        // Calling generic method with Integer as argument
        printGeneric(42);

        // Calling generic method with String as argument
        printGeneric("The answer is 42");
    }
}
```

https://www.scaler.com/topics/java/generics-in-java/

Java Generic Multiple Type

```
class Box<T, S> {
   private T t;
   private S s;

   public void add(T t, S s) {
      this.t = t;
      this.s = s;
   }

   public T getFirst() {
      return t;
   }

   public S getSecond() {
      return s;
   }
}
```

```
public class GenericsTester {
   public static void main(String[] args) {
      Box<Integer, String> box = new Box<Integer, String>();
      box.add(Integer.valueOf(10),"Hello World");
      System.out.printf("Integer Value :%d\n", box.getFirst());
      System.out.printf("String Value :%s\n", box.getSecond());

      Box<String, String> box1 = new Box<String, String>();
      box1.add("Message","Hello World");
      System.out.printf("String Value :%s\n", box1.getFirst());
      System.out.printf("String Value :%s\n", box1.getSecond());
   }
}
```

https://www.tutorialspoint.com/java_generics/java_generics_multiple_type.htm

Java Generics Upper Bounded Wildcard

```
public class GenericsWildcards {

	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("Sum of ints="+sum);
	}

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

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

Generics FAQs

- Why do we use Generics in Java?
  - Generics provide strong compile-time type checking and reduces risk of ClassCastException and explicit casting of objects.
- What is T in Generics?
  - We use <T> to create a generic class, interface, and method. The T is replaced with the actual type when we use it.
- How does Generics work in Java?
  - Generic code ensures type safety. The compiler uses type-erasure to remove all type parameters at the compile time to reduce the overload at runtime.
- Additional Advantage -> Code Reuse

third-party tuples

Last Updated: April 12, 2023 By: Sajal Chakraborty Java Data Structure

In this *Java tuple tutorial*, we will learn about **Tuples** – a generic data structure and how we can use tuples in a Java program. Tuples, by default, are not part of the Java programming language as a data structure so we will use one nice third-party library javatuples for it.

1. What is a Tuple?

A tuple can be seen as an ***ordered collection of objects of different types***. These objects do not necessarily relate to each other in any way, but collectively they will have some meaning.

For example, `["Sajal Chakraborty," "IT Professional," 32]` can be a tuple where each value inside the tuple does not have any relation, but this whole set of values can have some meaning in the application.

Let's see some more *Java tuple examples*.

```
Sample Tuples

["Java", 1.8, "Windows"]

["Alex", 32, "New York", true]

[3, "Alexa", "howtodoinjava.com", 37000]
```

https://howtodoinjava.com/java/java-tuples/

Javatuples core classes

- `Unit` (one element)
- `Pair` (two elements)
- `Triplet` (three elements)
- `Quartet` (four elements)
- `Quintet` (five elements)
- `Sextet` (six elements)
- `Septet` (seven elements)
- `Octet` (eight elements)
- `Ennead` (nine elements)
- `Decade` (ten elements)

4.2. Getting Values from Tuple

4.2.1. `getValue()` Methods

We can get the values from the tuples by using its indexed **`getValueX()`** methods where **`'X'`** denotes the element position inside the tuple. For example, **`getValue0()`**, **`getValue1()`** etc.

```
Pair<String, Integer> pair = Pair.with("Sajal", 12);

System.out.println("Name : " + pair.getValue0());
System.out.println("Exp : " + pair.getValue1());
```

Program output.

```
Name : Sajal
Exp : 12
```

Please note that these ***getValue()*** **methods are type-safe**. It means the compiler already knows the method return type based on the element values used to initialize the tuple.