GENERICS<> IN JAVA

Prepared by github.com/ali-rnp

I WHAT IS GENERIC ?

Generics means parameterized types introduced in java5 .

The idea is to allow type (Integer, String, ... etc., and user-defined types) to be a parameter to methods, classes, and interfaces.

Using Generics, it is possible to create classes that work with different data types.

II ADVANTAGE OF JAVA GENERICS

- **Type-safety**
  We can hold only a single type of objects in generics.
  It doesn't allow to store other objects.
- **Type casting is not required**
  There is no need to typecast the object.
- **Compile-Time Checking**
  It is checked at compile time so problem will not occur at runtime.
  The good programming strategy says it is far better to handle the problem at compile time than runtime.
- **Generics Promotes Code Reusability**
  With the help of generics in Java, we can write code that will work with different types of data.

III HOW TO DEFINE

IN CLASS OR INTERFACE

IN FIELD

```java
package me.preacher;

public class GenericClass<T> {
    private T type;  3 usages

    public GenericClass(T type) {
        this.type = type;
    }

    public T getType() {
        return type;
    }

    public void setType(T type) {
        this.type = type;
    }
}
```

```java
package me.preacher;

public class GenericField {
    private <T> type;
    private T t;
}
```

```
Unexpected token :4
Cannot resolve symbol 'type' :4
Identifier expected :4
Cannot resolve symbol 'T' :5
```

IN METHODE

Sometimes we don't want the whole class to be parameterized, in that case, we can create java generics method.

```java
package me.preacher;

public class GenericMethod<E> {  no usages

    private E e;  2 usages

    public <T> void print(T t){  no usages
        System.out.println(t.toString());
    }

    public <T> E getE (T t){  no usages
        return this.e;
    }

    public E getE(){  no usages
        return this.e;
    }
}
```

```java
package me.preacher;
public class Main {

    public static <T> T genericPrinter(T t) {
        System.out.println(t.toString());
        return t;
    }

}
```

IV WHY WE USE GENERICS ? WHY WE DON'T USE OBJECTS ?

The Object is the superclass of all other classes, and Object reference can refer to any object.

These features lack type safety.

Generics add that type of safety feature.

```java
package me.preacher;

public class UseGenerics<T> {
    private final T[] array;

    public UseGenerics(int capacity) {
        array = (T[]) new Object[capacity];
    }

    public void set(T t, int index){
        array[index] = t;
    }

    public T get(int index){
        return array[index];
    }
}
```

```java
package me.preacher;

public class UseObjects {
    private final Object[] objectArray;

    public UseObjects(int capacity) {
        objectArray = new Object[capacity];
    }

    public void set(Object object, int index){
        objectArray[index] = object;
    }

    public Object get(int index){
        return objectArray[index];
    }
}
```

```java
package me.preacher;

public class Main {
    public static void main(String[] args) {
        UseObjects stringObjects = new UseObjects( capacity: 10);
        stringObjects.set("Name", 0);
        stringObjects.set(2,1); // no compile error

        UseGenerics<String> stringGenerics = new UseGenerics<>
                ( capacity: 10);
        stringGenerics.set("Name",  0);
        stringGenerics.set(2, 1);// compile error

        // ...var
        Object o = stringObjects.get(1);
        String s = stringGenerics.get(1);

        // get objects from array
        Long o1 = (Long) stringObjects.get(0); // no compile error
        // >
        // |__> exception will be on runtime
        Long s2 = stringGenerics.get(0); // compile error
    }
}
```

```
'set(java.lang.String, int)' in 'me.preacher.UseGenerics' cannot be applied to '(int, int)' :11
Incompatible types. Found: 'java.lang.String', required: 'java.lang.Long' :21
```

V NAMING CONVENTION GENERICS

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

VI GENERICS AND PRIMITIVE DATA TYPES

A LITTLE TOPIC

WRAPPER CLASSES IN JAVA

The wrapper class in Java provides the mechanism to convert primitive into object and object into primitive.

Primitive Type	Wrapper class
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

USE OF WRAPPER CLASSES IN JAVA

Java is an object-oriented programming language, so we need to deal with objects many times like in Collections, Serialization, Synchronization, etc.

READ MORE
---> https://www.javatpoint.com/wrapper-class-in-java

BOXING,UNBOXING

The automatic conversion of primitive data types into its equivalent Wrapper type is known as BOXING and opposite operation is known as UNBOXING.

This is the new feature of Java5.

So java programmer doesn't need to write the conversion code.

- Boxing
- Unboxing

```java
package me.preacher;

public class Topic {
    public static void main(String[] args) {
        Integer integer = 5; // Boxing
        int i = integer; // Unboxing
    }
}
```

A restriction of generics in Java is that the type parameter cannot be a primitive type.

```java
package me.preacher;

public class PrimitiveTypeGenerics {
    GenericClass<int> integers = new GenericClass<>(); // compile error  no usages
}
```

```
Type argument cannot be of primitive type :4
'GenericClass(java.lang.Object)' in 'me.preacher.GenericClass' cannot be applied to '()' :4
```

VII JAVA GENERICS BOUNDED TYPE PARAMETERS

Suppose we want to restrict the type of objects that can be used in the parameterized type

IN CLASS OR INTERFACE

```java
package me.preacher;

public class GenericInheritance <T extends Number>{
    private final T[] array;  3 usages

    public GenericInheritance(int capacity) {
        array = (T[]) new Object[capacity];
    }

    public void set(T t, int index){
        array[index] = t;
    }

    public T get(int index){
        return array[index];
    }
}
```

```java
package me.preacher;

public class Main {
    public static void main(String[] args) {
        GenericInheritance<Long> longs = new GenericInheritance<>( capacity: 10); // no compile error
        GenericInheritance<Integer> integers = new GenericInheritance<>( capacity: 10); // no compile error

        GenericInheritance<String> strings = new GenericInheritance<>( capacity: 10); // compile error
        // because String does not extend Number
    }
}
```

IN METHOD

```java
package me.preacher;
public class Main {

    public static <T extends Number> T genericPrinter(T t) {
        System.out.println(t.toString());
        return t;
    }

    public static void main(String[] args) {
        Integer integer = genericPrinter( t: 25);
        System.out.println(genericPrinter( t: "Preacher")); // compile error
    }
}
```

```
'genericPrinter(T)' in 'me.preacher.Main' cannot be applied to '(java.lang.String)' :11
```

VIII MULTIPLE GENERIC TYPE

- IN CALSS
- IN METHOD

//TODO

IX WILDCARD S IN JAVA

Question mark (?) is the wildcard in generics and represent an unknown type.

The wildcard can be used as the type of a parameter, field, or local variable and sometimes as a return type.

We can't use wildcards while invoking a generic method or instantiating a generic class.

UPPER BOUNDED WILDCARD

Upper bounded wildcards are used to relax the restriction on the type of variable in a method.

Suppose we want to write a method that will return the sum of numbers in the list

WITHOUT WILDCARD

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

Now the problem with above implementation is that it won't work with List of Integers or Doubles because we know that List<Integer> and List<Double> are not related, this is when an upper bounded wildcard is helpful.

We use generics wildcard with extends keyword and the upper bound class or interface that will allow us to pass argument of upper bound or it's subclasses types

WITH WILDCARD

```java
package me.preacher;

import java.util.ArrayList;
import java.util.List;

public class Wildcards {

    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<>(List.of(1, 2, 3));
        double sum = sum(ints);
        System.out.println("Sum of ints= "+ sum);
    }
}
```

UNBOUNDED WILDCARD

Sometimes we have a situation where we want our generic method to be working with all types, in this case, an unbounded wildcard can be used.

```java
package me.preacher;

import java.util.List;

public class Wildcards {

    public static void printData(List<?> list){
        for(Object obj : list){
            System.out.println(obj);
        }
    }

}
```

LOWER BOUNDED WILDCARD

```java
package me.preacher;

import java.util.ArrayList;
import java.util.List;

public class Wildcards {

    public static void main(String[] args) {
        List<Number> numbers = new ArrayList<>();
        List<Object> objects = new ArrayList<>();
        List<Long> longs = new ArrayList<>();
        printer(objects);
        printer(numbers);
        printer(longs); // compile error
    }

    public static void printer(List<? super Integer> list){
        System.out.println(list);
    }
}
```