Why we need Generics in Java?

Errors are integral part of coding. Some errors occur at compile time and some errors occur at run time. Errors which occur at compile time can be easily identified and can be removed. But, run time errors occur when an application is running in real time. If they happen, they cause abrupt termination of an application.

**ClassCastException is also such an exception which happens only at run time. It occurs when data of one type can not be casted to another type**. You will never get a single clue about this exception during compilation. Look at the below code which throws ClassCastException at run time. But, you will never be get notified about this exception at compile time.

public class GenericsInJava

{

    public static void main(String[] args)

    {

        ArrayList list = new ArrayList();

        list.add("JAVA");

        list.add(123);

        for (Object object : list)

        {

**//Below statement throws ClassCastException at run time**

            String str = (String) object;       //Type casting

            System.out.println(str);

        }

    }

}

In this example, ‘list’ contains elements of String type as well as int type. When you try to cast it’s elements to string type in the for loop, element of string type is casted without throwing errors but element of int type throws ClassCastException.

**You can avoid ClassCastException by using generics in your code. The above example can be re-written using generics like below.**

public class GenericsInJava

{

    public static void main(String[] args)

    {

**ArrayList<String> list = new ArrayList<String>();**

        list.add("JAVA");

    //  list.add(123);       Compile time error

        for (String str : list)

        {

            //No type casting needed. ClasscastException Never occurs

            System.out.println(str);

        }

    }

}

Now, ‘list’ is declared so that it can hold only string type. If you try to add elements of different type, it gives compile time error. Therefore, ClassCastException never occurs while executing the for loop.

Generics are introduced in Java 5 to provide the type checking at compile time. If you use generics, you need not to perform the type casting explicitly. Java compiler applies strong type checking if you use generics in your code and shows errors if the code violates the type safety. Thus removing the risk of ClassCastException.

Therefore, To write the type safety code and to remove the risk of ClassCastException at run time, we need generics.

**Generics are used to check the type compatibility at the compile time and hence removing the chances of occuring ClassCastException at run time**

**How to define Generic class:**

class Class\_Name<T1, T2, T3 ... Tn>

{

    //Generic Type or Parameterized type

}

Where T1, T2, T3 … Tn (T stands for Type) enclosed within angle brackets (<>) are called **type parameters** and class ‘**Class\_Name**‘ is called generic type or parameterized type.

Now, let’s try to define one generic class based on the above format.

class GenericClass<T>

{

    T t;

    public GenericClass(T t)

    {

        this.t = t;

    }

    public void setT(T t)

    {

        this.t = t;

    }

    public T getT()

    {

        return t;

    }

}

While creating an instance to the above generic class, you can pass any class type as a type parameter and that class type replaces generic ‘T’ for that object. For example, if you pass String type as a type parameter then String will be the type of variable ‘t’. If you pass Integer as type parameter than Integer will be the type of variable ‘t’.

In the other words, when you pass a type while creating an object to the generic class, that object works only with that type. For example, If you pass String type while creating an object to the above generic class then that object works only with String type. That means setT() method takes String type as an argument and getT() method returns String type. If you pass any other type to setT() method, it gives compile time error. Hence, strictly checking type casting during compilation.

public class GenericsInJava

{

    public static void main(String[] args)

    {

        GenericClass<String> gen1 = new GenericClass<String>("It must be string");

        gen1.setT("Value Changed");        //Passing String to setT() method

        String s = gen1.getT();              //getT() method returning string

        gen1.setT(new Integer(123));      //Compile time error. You can't pass Integer type to setT() method now

        gen1.setT(new Double(23.56));    //Compile time error. You can't pass Double type to setT() method now

    }

}

If you create an object by using Integer type as a type parameter then that object works only with Integer type.

public class GenericsInJava

{

    public static void main(String[] args)

    {

        GenericClass<Integer> gen1 = new GenericClass<Integer>(new Integer(123));

        gen1.setT(456);             //Passing Integer type to setT() method

        Integer I = gen1.getT();      //getT() method returning Integer type

        gen1.setT(new String("123"));      //Compile time error. You can't pass String type to setT() method now

        gen1.setT(new Double(23.56));    //Compile time error. You can't pass Double type to setT() method now

    }

}

**Generics Work Only With Derived Types :**

**While creating an instance of generic class, you must pass only derived types. You can’t pass primitive types. If you pass primitive type, it gives compile time error. i.e generics works only with derived type.**

public class GenericsInJava

{

    public static void main(String[] args)

    {

        GenericClass<int> gen1 = new GenericClass<int>(123);   **//Error, can't use primitive type**

        GenericClass<float> gen2 = new GenericClass<float>(23.56);  **//Error, can't use primitive type**

    }

}

**Objects Of Same Generic Class Differ Based On Their Type Parameters :**

Objects of same generic class differ depending upon their type parameters. For example, object of above generic class created using String type is not compatible with an object of same class created using Integer type.

public class GenericsInJava

{

    public static void main(String[] args)

    {

        GenericClass<String> gen1 = new GenericClass<String>("Value Of t");

        GenericClass<Integer> gen2 = new GenericClass<Integer>(new Integer(20));

        gen1 = gen2;        //Error : Type mismatch

        gen2 = gen1;        //Error : Type mismatch

    }

}

**Generic Class With Two Type Parameters :**

Below is an example of a generic class with two type parameters.

class GenericClass<T1, T2>

{

    T1 t1;

    T2 t2;

    public GenericClass(T1 t1, T2 t2)

    {

        this.t1 = t1;

        this.t2 = t2;

    }

    public void setT1(T1 t1)

    {

        this.t1 = t1;

    }

    public T1 getT1()

    {

        return t1;

    }

    public void setT2(T2 t2)

    {

        this.t2 = t2;

    }

    public T2 getT2()

    {

        return t2;

    }

}

public class GenericsInJava

{

    public static void main(String[] args)

    {

**GenericClass<String, Integer> gen1 = new GenericClass<String, Integer>("Value of t1", new Integer(123));**

**GenericClass<Integer, String> gen2 = new GenericClass<Integer, String>(new Integer(123), "Value of t2");**

        System.out.println(gen1.getT1());       //Output : Value of t1

        System.out.println(gen1.getT2());       //Output : 123

        System.out.println(gen2.getT1());       //Output : 123

        System.out.println(gen2.getT2());       //Output : Value of t2

    }

}

You can pass your own type while creating an instance to the generic class. Here is an example for that.

class GenericClass<T>

{

    T t;

    public GenericClass(T t)

    {

        this.t = t;

    }

    public void setT(T t)

    {

        this.t = t;

    }

    public T getT()

    {

        return t;

    }

}

class A

{

    int i;

    public A(int i)

    {

        this.i = i;

    }

}

public class GenericsInJava

{

    public static void main(String[] args)

    {

        GenericClass<A> gen1 = new GenericClass<A>(new A(10));     //Passing A-type as type parameter

        GenericClass<A> gen2 = new GenericClass<A>(new A(20));     //Passing A-type as type parameter

        System.out.println(gen1.getT().i);    //Output : 10

        System.out.println(gen2.getT().i);    //Output : 20

    }

}

[**Rules To Follow While Implementing Generic Interfaces**](https://javaconceptoftheday.com/generic-interfaces-java/)

Like generic classes, you can also define generic interfaces. The same syntax used to define generic classes is also used to define generic interfaces. Here is an example of generic interface.

interface GenericInterface<T>

{

    void setT(T t);

    T getT();

}

While implementing generic interfaces you have to follow some rules. Below is the discussion of those rules.

Only generic classes can implement generic interfaces. Normal classes can’t implement generic interfaces. For example, above generic interface can be implemented as,

class GenericClass<T> implements GenericInterface<T>

{

}

Not like below. It gives compile time error.

class NormalClass implements GenericInterface<T>

{

     //Compile time error

}

Here is the full implementation of above generic interface.

class GenericClass<T> implements GenericInterface<T>

{

    T t;

    //Implementing setT() method

    @Override

    public void setT(T t)

    {

        this.t = t;

    }

    //Implementing getT() method

    @Override

    public T getT()

    {

        return t;

    }

}

A normal class can implement a generic interface if type parameter of generic interface is a wrapper class. For example, below implementation of GenericInterface is legal.

interface GenericInterface<Integer>

{

       //Generic interface with Integer as type parameter

}

class NormalClass implements GenericInterface<Integer>

{

       //Normal class implementing generic interface

}