**Generics in Java**

Generics was added in Java 5 to provide **compile-time type checking** and removing risk of ClassCastException that was common while working with collection classes. The whole collection framework was re-written to use generics for type-safety. Let’s see how generics help us using collection classes safely.

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

}

Notice that at the time of list creation, we have specified that the type of elements in the list will be String. So if we try to add any other type of object in the list, the program will throw compile time error. Also notice that in for loop, we don’t need type casting of the element in the list, hence removing the ClassCastException at runtime.

**Java Generic Class**

We can define our own classes with generics type. A generic type is a class or interface that is parameterized over types. We use angle brackets (<>) to specify the type parameter.

To understand the benefit, lets say we have a simple class as:

package generics;

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 ClassCastException

}

}

Notice that while using this class, we have to use type casting and it can produce ClassCastException at runtime. Now we will use java generic class to rewrite the same class as shown below.

package generics;

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

}

}

Notice the use of GenericsType class in the main method. We don’t need to do type-casting and we can remove ClassCastException at runtime. If we don’t provide the type at the time of creation, compiler will produce a warning that “GenericsType is a raw type. References to generic type GenericsType<T> should be parameterized”. When we don’t provide type, the type becomes Object and hence it’s allowing both String and Integer objects but we should always try to avoid this because we will have to use type casting while working on raw type that can produce runtime errors.

**Java Generic Interface**

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);

}

In similar way, we can create generic interfaces in java. We can also have multiple type parameters as in Map interface. Again we can provide parameterized value to a parameterized type also, for example new HashMap<String, List<String>>(); is valid.

**Java Generic Type**

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 it’s 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**

Sometimes we don’t want whole class to be parameterized, in that case we can create java generics method. Since constructor is a special kind of method, we can use generics type in constructors too.

Here is a class showing example of java generic method.

package generics;

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 ordinary method, without specifying a type between angle brackets.

//Compiler will infer the type that is needed

}

}

Notice the *isEqual* method signature showing syntax to use generics type in methods. Also notice how to use these methods in our java program. We can specify type while calling these methods or we can invoke them like a normal method. Java compiler is smart enough to determine the type of variable to be used, this facility is called as **type inference**.

**Java Generics Bounded Type Parameters**

Suppose we want to restrict the type of objects that can be used in the parameterized type, for example in a method that compares two objects and we want to make sure that the accepted objects are Comparables. To declare a bounded type parameter, list the type parameter’s name, followed by the extends keyword, followed by its upper bound, similar like below method.

public static <T extends Comparable<T>> int compare(T t1, T t2){

return t1.compareTo(t2);

}

The invocation of these methods is similar to unbounded method except that if we will try to use any class that is not Comparable, it will throw compile time error.

Bounded type parameters can be used with methods as well as classes and interfaces.

Java Generics supports multiple bounds also, i.e <T extends A & B & C>. In this case A can be an interface or class. If A is class then B and C should be interfaces. We can’t have more than one class in multiple bounds.

**Java Generics and Inheritance**

We know that [Java inheritance](http://www.journaldev.com/644/inheritance-in-java-example) allows us to assign a variable A to another variable B if A is subclass of B. So we might think that any generic type of A can be assigned to generic type of B, but it’s not the case. Lets see this with a simple program.

package generics;

public class GenericsInheritance {

public static void main(String[] args) {

String str = "abc";

Object obj = new Object();

obj=str; // works because String is-a Object, inheritance in java

MyClass<String> myClass1 = new MyClass<String>();

MyClass<Object> myClass2 = new MyClass<Object>();

//myClass2=myClass1; // compilation error since MyClass<String> is not a MyClass<Object>

obj = myClass1; // MyClass<T> parent is Object

}

public static class MyClass<T>{}

}

We are not allowed to assign MyClass<String> variable to MyClass<Object> variable because they are not related, in fact MyClass<T> parent is Object.

**Java Generic Classes and Subtyping**

We can subtype a generic class or interface by extending or implementing it. The relationship between the type parameters of one class or interface and the type parameters of another are determined by the extends and implements clauses.

For example, ArrayList<E> implements List<E> that extends Collection<E>, so ArrayList<String> is a subtype of List<String> and List<String> is subtype of Collection<String>.

The subtyping relationship is preserved as long as we don’t change the type argument, below shows an example of multiple type parameters.

interface MyList<E,T> extends List<E>{

}

The subtypes of List<String> can be MyList<String,Object>, MyList<String,Integer> and so on.

**Java Generics Wildcards**

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. In following sections, we will learn about upper bounded wildcards, lower bounded wildcards, and wildcard capture.

**Java Generics 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, so our implementation will be something like this.

public static 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 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.

The above implementation can be modified like below program.

package generics;

import java.util.ArrayList;

import java.util.List;

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;

}

}

It’s similar like writing our code in terms of interface, in above method we can use all the methods of upper bound class Number. Note that with upper bounded list, we are not allowed to add any object to the list except null. If we will try to add an element to the list inside the sum method, the program won’t compile.

**Java Generics Unbounded Wildcard**

Sometimes we have a situation where we want our generic method to be working with all types, in this case unbounded wildcard can be used. Its same as using <? extends Object>.

public static void printData(List<?> list){

for(Object obj : list){

System.out.print(obj + "::");

}

}

We can provide List<String> or List<Integer> or any other type of Object list argument to the *printData*method. Similar to upper bound list, we are not allowed to add anything to the list.

**Java Generics Lower bounded Wildcard**

Suppose we want to add Integers to a list of integers in a method, we can keep the argument type as List<Integer> but it will be tied up with Integers whereas List<Number> and List<Object> can also hold integers, so we can use lower bound wildcard to achieve this. We use generics wildcard (?) with **super**keyword and lower bound class to achieve this.

We can pass lower bound or any super type of lower bound as an argument in this case, java compiler allows to add lower bound object types to the list.

public static void addIntegers(List<? super Integer> list){

list.add(new Integer(50));

}

**Subtyping using Generics Wildcard**

List<? extends Integer> intList = new ArrayList<>();

List<? extends Number> numList = intList; // OK. List<? extends Integer> is a subtype of List<? extends Number>