**Java Generic Class:**

**Basic Example:**

import java.util.ArrayList;

class GenericStack<T>

{

private ArrayList<T> container;

private int top;

GenericStack()

{

container=new ArrayList<>();

top=0;

}

public int size()

{

return top;

}

public void push(T item)

{

container.add(top++,item);

//add(int index,E item) is called

}

public T pop()throws Exception

{

if(top==0)

{

throw new Exception("The stack is empty");

}

return container.remove(--top);

}

public boolean empty()

{

return (top==0);

}

}

public class GenericTypeExample

{

public static void main(String args[])throws Exception

{

try

{

GenericStack<Integer> container=new GenericStack<>();

container.push(10);

System.out.println("10 is being pushed in stack");

container.push(20);

System.out.println("20 is being pushed in stack");

container.push(30);

System.out.println("30 is being pushed in stack");

System.out.println("the element is popped: "+container.pop());

System.out.println("the element is popped: "+container.pop());

System.out.println("the element is popped: "+container.pop());

System.out.println("the element is popped: "+container.pop());

}

catch(Exception e)

{

System.out.println("The exception thrown: "+e);

}

}

}

### **Java Generic Interface**

**Example From Existing Interfaces:**

package java.lang;  
import java.util.\*;  
public interface Comparable<T>

{  
 public int compareTo(T o);  
}

Now, Comparable interface which required in sort function is a great example of generic interface.

### **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 collection framework, for example ArrayList, Set etc.)
* K – Key (Used in Map)
* N – Number
* T – Type
* V – Value (Used in Map)

**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. **(is that related to multiple inheritance issue, again?)**

Also if there is a class and there are interfaces present as bounds, and if the class bound is not specified first, you will get a compilation error.

For instance, see the following example:

**Class A { /\* ... \*/ }  
interface B { /\* ... \*/ }  
interface C { /\* ... \*/ }  
  
class D <T extends A & B & C> { /\* ... \*/ }**

**If bound A is not specified first, you get a compile-time error:  
  
class D <T extends B & A & C> { /\* ... \*/ } // compile-time error**

**Example:**

public static <T extends Number & Comparable<T>> T maximum(T x, T y, T z)

What does that mean?

The T is a type parameter passed to the generic class Box and should be subtype of Number class and must implements Comparable interface. In case a class is passed as bound, it should be passed first before interface otherwise compile time error will occur.

(that’s why, there are rules. Like, in **class D <T extends A & B & C> ,**  only one can be class. And if the class is present, it must be A, neither B nor C)

**One good example is:**

**public class MaximumTest {  
 // determines the largest of three Comparable objects  
   
 public static <T extends Comparable<T>> T maximum(T x, T y, T z) {  
 T max = x; // assume x is initially the largest  
   
 if(y.compareTo(max) > 0) {  
 max = y; // y is the largest so far  
 }  
   
 if(z.compareTo(max) > 0) {  
 max = z; // z is the largest now   
 }  
 return max; // returns the largest object   
 }  
   
 public static void main(String args[]) {  
 System.out.printf("Max of %d, %d and %d is %d\n\n",   
 3, 4, 5, maximum( 3, 4, 5 ));  
  
 System.out.printf("Max of %.1f,%.1f and %.1f is %.1f\n\n",  
 6.6, 8.8, 7.7, maximum( 6.6, 8.8, 7.7 ));  
  
 System.out.printf("Max of %s, %s and %s is %s\n","pear",  
 "apple", "orange", maximum("pear", "apple", "orange"));  
 }  
}**

Now,   
 public static <T extends Comparable<T>> T maximum(T x, T y, T z)

It ensures that x,y and z are comparable objects, right?

**Java Generics and Inheritance:**

We know that Java inheritance 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.

**( we can do B b=new A();**

**//assigning the object of B to the reference of A. Since, A is subclass of B)**

**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>’s parent is Object  
 }  
   
 public static class MyClass<T>{}  
   
}**Now, str can be initialized to obj as both belong to the same inheritance tree and Object class is parent of String class.

**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>

{  
}

(Note that one interface can extends another interface. One interface cannot implement another interface)

**WildCards In Java Generic Programming:**

Question mark (?) is the wildcard in generics and represent an unknown type. The wildcard can be used as the type of a parameter (parameter to function), field (instance variables?), or local variable and sometimes as a return type.

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

**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, **Number** is parent class of AtomicInteger, AtomicLong, BigDecimal, BigInteger, Byte, Double, Float, Integer, Long, Short.

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.

(now, i don’t understand it. Hence, I will try to understand it by trial and error)

Now, I understand the meaning:

**import java.util.ArrayList;  
import java.util.List;  
  
public class GenericUpperWildCardExample  
{  
  
 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<Number> list){  
 double sum = 0;  
 for(Number n : list){  
 sum += n.doubleValue();  
 }  
 return sum;  
 }  
}**

Now, Since, java.lang.Integer and java.lang.Double are not related, we cannot pass a List<Integer> to the function since, in the function doubleValue is called.

This will generate compilation time error (yes, not runtime error. Because, function declaration finding (i.e. compiler will check whether the function doubleValue() is declared for Integer class ) is done in compile time. Hence, compilation error)

Now, using upper bound wildcard relax the restriction on the type of variable in a method.

**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;  
 }  
}**

Now, using upper bound wildcard we allow to pass argument of upper bound or it’s subclasses types.

**Few Misconceptions Regarding This: (and some more trial and errors to understand the actual meaning of upperbound wild card)**

**import java.util.ArrayList;  
import java.util.List;  
  
public class Test  
{  
  
 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<Number> list){  
 double sum = 0;  
 for(Number n : list){  
 Double x=(Double)n;  
 sum += x.doubleValue();  
 }  
 return sum;  
 }  
}**

This is not allowed.

This will give the following compilation error:

double sum = sum(ints);

^

required: List<Number>

found: List<Integer>

reason: actual argument List<Integer> cannot be converted to List<Number> by method invocation conversion

**The following is also not allowed:**

**import java.util.ArrayList;**

**import java.util.List;**

**public class Test**

**{**

**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);**

**\*/**

**List<Double> doubles=new ArrayList<>();**

**doubles.add(3.0);**

**doubles.add(5.0);**

**doubles.add(7.0);**

**double sum=sum(doubles);**

**System.out.println("Sum of doubles="+doubles);**

**}**

**public static double sum(List<Number> list){**

**double sum = 0;**

**for(Number n : list){**

**Double x=(Double)n;**

**sum += x.doubleValue();**

**}**

**return sum;**

**}**

**}**

This is also not allowed.

Test.java:18: error: method sum in class Test cannot be applied to given types;

double sum=sum(doubles);

^

required: List<Number>

found: List<Double>

That is why relaxation is allowed. Now, why **upperbound**? Because, a particular class’s all subclass are being allowed?

### **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.

Now, it is relatively easier.

**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));  
}**

A more proper example of this:

**import java.util.ArrayList;  
import java.util.List;  
  
public class GenericsTester {  
  
 public static void addCat(List<? super Cat> catList) {  
 catList.add(new RedCat());  
 System.out.println("Cat Added");  
 }  
  
 public static void main(String[] args) {  
 List<Animal> animalList= new ArrayList<Animal>();  
 List<Cat> catList= new ArrayList<Cat>();  
 List<RedCat> redCatList= new ArrayList<RedCat>();  
 List<Dog> dogList= new ArrayList<Dog>();  
  
 //add list of super class Animal of Cat class  
 addCat(animalList);  
  
 //add list of Cat class  
 addCat(catList);  
  
 //compile time error  
 //can not add list of subclass RedCat of Cat class  
 //addCat(redCatList);  
  
 //compile time error  
 //can not add list of subclass Dog of Superclass Animal of Cat class  
 //addCat.addMethod(dogList);   
 }  
}  
class Animal {}  
  
class Cat extends Animal {}  
  
class RedCat extends Cat {}  
  
class Dog extends Animal {}**

**Java Generics Type Erasure:**

**This is actually an advantage provided by java generics type. Necessary errors are not found in runtime, rather in would be found in compile time**

Generics in Java was added to provide type-checking at compile time and it has no use at run time, **so java compiler uses type erasure feature to remove all the generics type checking code in byte code and insert type-casting if necessary. (generics type checking is removed from bytecode. So that, compatibility is not checked at runtime. Insert type casting if necessary, why? So, it does not generate exception in runtime)** Type erasure ensures that no new classes are created for parameterized types **(did not understand it)**; consequently, generics incur no runtime overhead.

**public class Test<T extends Comparable<T>>**

**{  
 private T data;  
 private Test<T> next;  
 public Test(T d, Test<T> n)**

**{  
 this.data = d;  
 this.next = n;  
 }  
 public T getData() { return this.data; }  
}**

Now, T extends Comparable <T> what does that mean?

That means type parameter must support comparison with other instances of its own type, via the Comparable interface.

Now, here, the Java compiler replaces the bounded type parameter T with the first bound interface, Comparable, as below code:  
  
**public class Test {  
  
 private Comparable data;  
 private Test next;  
  
 public Node(Comparable d, Test n) {  
 this.data = d;  
 this.next = n;  
 }  
  
 public Comparable getData() { return data; }**

(remember) we can do things like T extends A & B & C

However, If A is class, B and C must be interface in this case.