Generics

Generics

- The term generic means parameterized types.
- Through the use of generics, it is possible to create classes, interfaces and methods that will work in a type safe manner with various kinds of object.
- A class, interface, or method that operates on a parameterized type is called generic, as in generic class or generic method
- Object references can be used to operate on various types of objects. The problem is that they could not do so with type safety.
- When java code is compiled, all generic type information is removed (erased, known as **Erasure**) i.e. **actually replacing type parameters with their bound type which is Object if no explicit bound is specified.**
- no type info i.e. T exists at run time.
- **Generics work only with objects and not with primitive type (int, float, char etc.)**

Generics

Syntax: **class class-name <type parameter list>**

```
Ex.     class Gen <T>
{
        T ob;

        Gen(T ob)
        {
                this.ob = ob;
        }

        T getOb()
        {
                return ob;
        }
}
```

NOTE: class<T> Gen { //code } **//Error** as type parameter must come after class name.

Generics

```
class GenDemo {
	public static void main(String args[])
	{
		// Type safety : try to pass any value other than int or Integer like 100.0
		Gen <Integer> iOb = new Gen <Integer> (100);
		System.out.println(iOb.getOb());

		Gen <String> strOb = new Gen <String> ("Welcome");
		System.out.println(strOb.getOb());

		int v = (Integer)iOb.getOb();		// No typecasting is required
		System.out.println(v);

		//iOb = strOb;				// Type safety

	}
}
```

Generics

Note:
Gen <Integer> iOb = new Gen <> (100); // Works
Gen <> iOb = new Gen <Integer> (100); // error: illegal start of type

Generics provides type safety and avoids explicit casting i.e. performs implicit casting.

Generic types differ based on their type arguments so reference of one specific version of a generic type is not type compatible with another version of same generic type.

Ex. Gen<Integer> intOb = new Gen<Integer>();
Gen<String> strOb = new Gen<String>();
intOb = strOb; **// Error,** because they are references to different types.
//error: incompatible types: Gen<String> cannot be converted to Gen<Integer>

Generics

Non-generic Demo Program

```
class NonGen
{
	Object ob;
	NonGen(Object ob)
	{
		this.ob = ob;
	}

	void  setOb(Object ob)
	{
		this.ob = ob;
	}

	Object getOb()
	{
		return ob;
	}
}
```

Generics

Non-generic Demo Program Contd...

```
class NonGenDemo
{
	public static void main(String args[])
	{
		NonGen  iOb = new NonGen  (100);  // Type safety : try to pass any value other than int or Integer like 100.0
		System.out.println(iOb.getOb());

		NonGen strOb = new NonGen ("Welcome");
		System.out.println(strOb.getOb());

		int v = (Integer)iOb.getOb();		// typecasting is required
		System.out.println(v);

		iOb = strOb;				// No Type safety. Compiler does not show any error.
		//v = (Integer)iOb.getOb();		// Runtime error
	}
}
```

Generics

Upper Bound

// upper bound(s): Number - Type can be either Number or any child class of Number like Integer, Doublec etc.

class Gen <T extends Number> { ..... }

Generics

Generic Methods

The type parameters are declared before the return type of the method.
Methods can be static or non-static.
Ex. static <T, V extends T> boolean isInArr(T x, V[ ] y) {

We can not apply type variable with static data member.
No static member can use a type parameter declared by the enclosing class.

Ex.

```
class Demo <T>
{
        static T ob;              // Error
        static void showOb( ) { } // Error
        T getOb( ) { }            // use with non-static OK
        void setOb( T ob) { }  // use with non-static OK

}
```

Generics

Generic Methods

Although we can not declare static members that uses a type parameter declared by the enclosing class, we can declare static generic methods, which define their own type parameters.

Ex.

```
class Gen <T>
{
        T ob;
        static <T2> void showOb( T2 ob) // OK
        {
                System.out.println(ob);
        }
}
```

A non-generic class can have generic functions/methods

A non-generic class can have generic constructor.

Ex. <T extends Number> Gen(T obj) {

Generics

Generic Methods

Generic or non-generic class can contain generic functions which might use the type parameter of the enclosing class or may use different type parameter.

Ex.

```
class Gen<T>
{
        <T2> void doTask ( T2 ob)  // new type parameter T2 is used here which is different from T.
        {
                // code
        }
}
```

Generics

Erasure

Example code of generic with String type:

```
Gen <String> strOb = new Gen<String>();
```

Because the type argument is String, String is substituted for T inside Gen and it creates conceptually a String version of Gen.

But that is just conceptually!

Generic type info does not exist at run time.
All type parameters are erased during compilation. At run time, only raw types actually exist.

Actually Java compiler does not create different versions of Gen or of any other generic class. Instead the compiler removes all generic type information, substituting the necessary casts, to make your code behave as if a specific version of Gen was created. So, there is only one version of Gen that actually exists in the program. This process of removing generic type information is called **erasure**.

Generics

Erasure

When java code is compiled, all generic type information is removed (erased) i.e. actually replacing type parameters with their bound type which is Object if no explicit bound is specified.

Ex. 1. //Here T is bound by Object itself

```
class Gen <T>
	{
		T obj;
		Gen(T o)
		{
			Obj = o;
		}

		T getObj()
		{
			return ob;
		}

		//code
	}
```

Generics

Erasure

Compiler will convert the above code as below:

```
Class Gen extends java.lang.Object
{
        java.lang.Object ob;
        Gen(java.lang.Object ob)
        {
                Obj = o;
        }

        java.lang.Object getObj()
        {
                return ob;
        }
        //code
}
```

Generics

Erasure

Ex. 2. // Here T is bound by String

```
class Gen <T extends String>
        {
                T obj;
                Gen(T o)
                {
                        Obj = o;
                }

                T getObj()
                {
                        return ob;
                }

                //code
        }
```

Generics

Erasure

Compiler will convert the above code as below:

```
Class Gen extends java.lang.Object
{
        java.lang.String ob;
        Gen(java.lang.String ob)
        {
                Obj = o;
        }

        java.lang.String getObj()
        {
                return ob;
        }
        //code
}
```

Generics

Erasure

Now Java compiler itself performs the typecasting wherever required to ensure the type safety.

Ex.
Gen<Integer> intOb = new Gen<Integer>(99);
int x = intOb.getObj();

Would be compiled like as below:

Gen intOb = new Gen(99);
int x = (Integer) intOb.getObj();

Some noticeable points:
intOb.getClass().getName(); // Output : Gen
strOb.getClass.getName(); // Output : Gen

Because of erasure at run time only raw types actually exist.

Generics

Bounded Types (Type Parameter)

To create an upper bound that declares the superclass from which all type arguments must be derived. This is accomplished through the use of extends.

<T extends superclass>

It means, T can be replaced by superclass or subclass of superclass. So, here superclass is an inclusive upper limit.

[Example with **<T extends Number>** and use Integer, Double, String] . **With String will give compilation error.**

Generics

Using Wildcard Arguments

Ex.

```
class Stats <T>
{
        boolean sameAvg(Stats<T> obj)
        {
                //Function code
        }
}
```

// Works only with Stats objects whose type is similar to invoking object i.e. if invoking obj is of type Stat<Integer> then obj type must also be Stats<Integer> and not Stats<Double>.

Solution:

```
Boolean sameAvg(Stats<?> obj)
{
        //Function Code
}
```

//Works with all valid Stats objects irrespective of invoking object's type.

Example: Comparing averages of Integer and Double objects.

Generics

Bounded WildCard

To establish upper bound with wildcard

It can be bound using the extend keyword just as discussed earlier.

<? extends superclass>

In this case only classes that are subclasses of superclass will be accepted. Here superclass is **inclusive.**

Ex. Boolean sameAvg(Stats<? extends Number> obj)

```
{
        //Function Code
}
```

Meaning sameAvg will take any valid Stat object as argument provided it's type should be either Number or any subclass of Number class.

Generics

To establish **lower bound** with wildcard

<? super subclass>

In this case only classes that are superclasses of subclass will be accepted. Here subclass is **exclusive.** It (super) can not be used at class level.

Ex. class Gen<T super FileInputStream> ; **//Will give error.**

Generics

Generic Interfaces
We can create generic interfaces as we create generic classes as below.
Interface GenInterface <T extends Number> { ……. }

Class implementing interface must establish the same bound as in interface, if the class is also generic.
Ex. class MyClass <T extends Number> implements GenInterface<T>

Remember, there is no need to specify the bound again in the implements clause.
So,
class MyClass <T extends GenInterface<T>> implements MinMax<T extends GenInterface<T>> // This declaration is wrong.

In general, if a class implements a generic interface then it can be either generic or non-generic but in the case of non-generic i.e. when class **implements a specific type of the generic interface**, the class must pass that specific type to the interface.

Ex. class MyClass implements GenInterface<T> { **//Wrong, compiler will report error.**
class MyClass implements GenInterface<Integer> { **//OK**

Generics

Generic Interface Demo Program

```
interface GenInterface <T extends Number>
{
	void calAvg();
}

class Stats<T extends Number> implements GenInterface <T>
{
	T nums[];
	Stats( T nums[])
	{
		this.nums =  nums;
	}

	public void calAvg()
	{
		double sum = 0.0;
		double avg = 0.0;
```

Generics

Generic Interface Demo Program Contd...

```
   for (int i = 0; i < nums.length; i++)
                {
                        sum = sum + nums[i].doubleValue();
                }
           avg = sum / nums.length;
           System.out.println(avg);
        }
}

class GenInterfaceDemo
{
        public static void main(String args[])
        {
                Integer intArr [] = {3,5,8,7,6};
                Double doubleArr [] = {5.6, 8.4, 3.2, 4.8, 5.6};
                Stats<Integer> s1 = new Stats <Integer> (intArr);
                Stats<Double> s2 = new Stats <Double> (doubleArr);
                s1.calAvg();
                        s2.calAvg();

        }
}
```

Generics

Generic Class Hierarchies

Ex. class Gen2<T> extends Gen<T> {

Subclass must need to specify the type parameters (either generic or specific type) required by the superclass.

Ex.

class Gen2<T,V> extends Gen<T> { // Gen2 is also generic and it has another type also V

class Gen2 extends Gen<Integer> { // Integer-specific version of Gen. Gen2 is non-generic and it passes specific type i.e. Integer to Gen type parameter

Note: If superclass has any constructor explicitly defined, the same constructor should be defined in the subclass otherwise it will give compilation error.

It is perfectly acceptable for a non-generic class to be the superclass of a generic subclass

Generics

Demo Program: Non-generic subclass with a generic superclass

```
abstract class Gen <T extends Number>
{
	T nums[];

	Gen(T nums[])
	{
		this.nums = nums;
	}
	public double getAvg()
	{
		double sum = 0.0;
		double avg = 0.0;

		for (int i = 0; i < nums.length; i++)
		{
			sum = sum + nums[i].doubleValue();
		}
		avg = sum / nums.length;
		System.out.println(avg);
		return avg;
	}
}
```

Generics

Demo Program: Non-generic subclass with a generic superclass

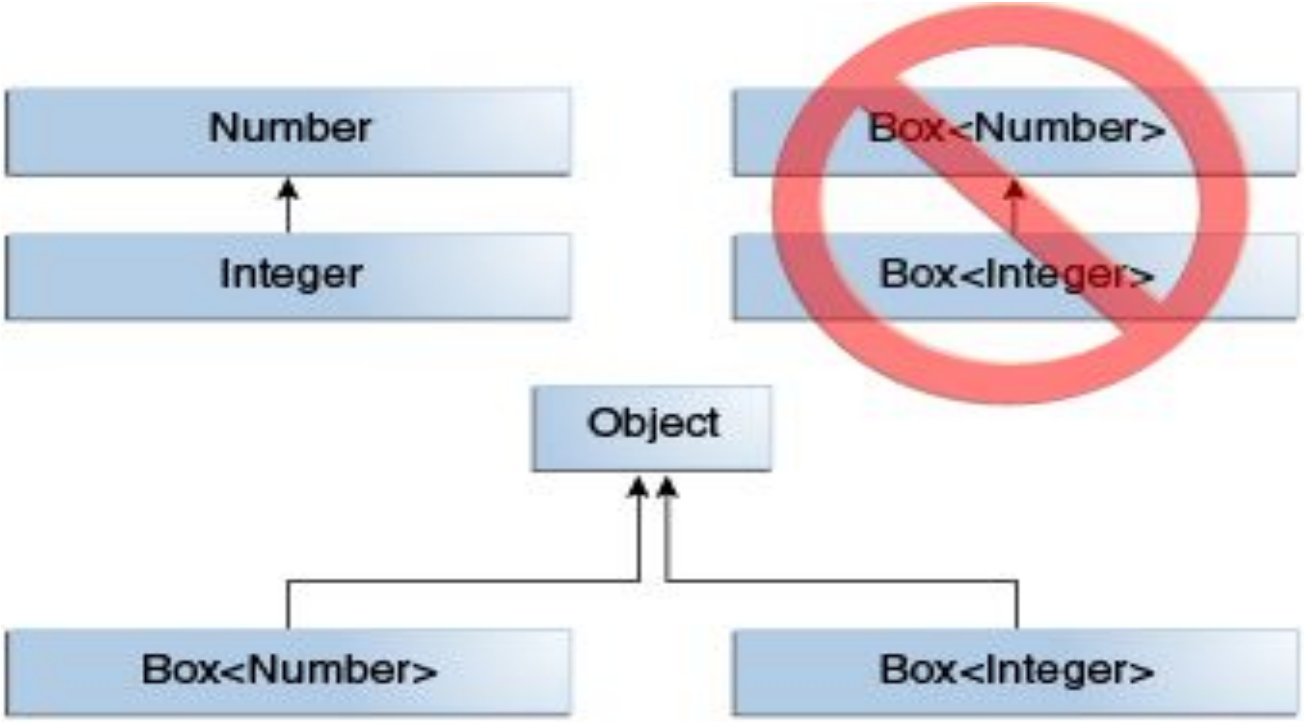
```
class Stats extends Gen <Integer>
{

	Stats(Integer nums[])
	{
		super(nums);
	}
}

class GenDemo5
{
	public static void main(String args[])
	{
		Integer intArr [] = {3,5,8,7,6};
			Stats s1 = new Stats (intArr);
		s1.getAvg();

	}
}
```

Generics


Number
Integer
Box<Number>
Box<Integer>
Object
Box<Number>
Box<Integer>

Generics

Some Generic Restrictions

Type parameter can't be instantiated.

T ob = new T() ; // Illegal. Where T is a type parameter

No static members (class and methods) can use a type parameter declared by the enclosing class.

We can not instantiate an array whose base type is a type parameter.

We can not create an array of type specific generic references.

Ex.

```
Class Gen < T extends Number>
{
        T vals [ ];
        Gen(T [ ] nums)
        {
        vals = new T[10]; // Wrong, can't create an array of T. Reason is no type info i.e. T exists at run time.
        vals = nums // OK. Assigning reference is ok.
        }
}
```

A generic class can not extend Throwable. Creating generic exception classes is not allowed.