Generics in Java

Generic Classes

A generic class declaration looks like a non-generic class declaration, except that the class name is followed by a type parameter section.

The type parameter section of a generic class can have one or more type parameters separated by commas. These classes are known as parameterized classes or parameterized types because they accept one or more parameters.

Syntax:

```
public class Box<T> {
   private T t;
}
```

- **Box** – Box is a generic class.
- **T** – The generic type parameter passed to generic class. It can take any Object.
- **t** – Instance of generic type T.

Naming Conventions

By convention, type parameter names are named as single, uppercase letters so that a type parameter can be distinguished easily with an ordinary class or interface name. Following is the list of commonly used type parameter names –

- **E** – Element (mainly used by Java Collections framework)
- **K** – Key (mainly used to represent key of a map)
- **V** – Value (mainly used to represent value of a map)
- **N** – Number (represents numbers)
- **T** – Type (represents first generic type parameter)
- **S, U, V, etc** – 2nd, 3rd, 4th Types

Type Inference

Type inference represents the Java compiler's ability to look at a method invocation and its corresponding declaration to check and determine the type argument(s). The

inference algorithm checks the types of the arguments and, if available, assigned type is returned. Inference algorithms tries to find a specific type which can fullfil all type parameters.

Compiler generates unchecked conversion warning in-case type inference is not used.

Syntax:

```
Box<Integer> integerBox = new Box<>();
```

- **Box** – Box is a generic class.
- **<>** – The diamond operator denotes type inference.

Using diamond operator, compiler determines the type of the parameter. This operator is available from Java SE 7 version onwards.

Generic Methods

You can write a single generic method declaration that can be called with arguments of different types. Based on the types of the arguments passed to the generic method, the compiler handles each method call appropriately. Following are the rules to define Generic Methods –

- All generic method declarations have a type parameter section delimited by angle brackets (< and >) that precedes the method's return type (< E > in the next example).
- Each type parameter section contains one or more type parameters separated by commas. A type parameter, also known as a type variable, is an identifier that specifies a generic type name.
- The type parameters can be used to declare the return type and act as placeholders for the types of the arguments passed to the generic method, which are known as actual type arguments.
- A generic method's body is declared like that of any other method. Note that type parameters can represent only reference types, not primitive types (like int, double and char).

```
public static <E> void printArray( E[] inputArray ) {
      // Display array elements
      for(E element : inputArray) {
         System.out.printf("%s ", element);
      }
      System.out.println();
   }
```

Multiple Type Parameters

A Generic class can have multiple type parameters. Following example will showcase above mentioned concept.

```
public class Box<S,T> {
   private T t;
   private S s;
}
```

Parameterized Types

A Generic class can have parameterized types where a type parameter can be substituted with a parameterized type. Parameterized Types are types that take other types as parameters. Eg - Collection, ArrayList, etc.

```
public class Box<S,T> {
   ...
}
...
Box<Integer, List<String>> box = new Box<Integer, List<String>>(); ,
...
```

Raw Types

A raw type is an object of a generic class or interface if its type arguments are not passed during its creation.

```
Box rawBox = new Box();
```

Bounded Type Parameters

There may be times when you'll want to restrict the kinds of types that are allowed to be passed to a type parameter. For example, a method that operates on numbers might only want to accept instances of Number or its subclasses. This is what bounded type parameters are for.

To declare a bounded type parameter, list the type parameter's name, followed by the extends keyword, followed by its upper bound.

Single Bound:

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

Multiple Bounds:

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

- **maximum** – maximum is a generic method.
- **T** – The generic type parameter passed to generic method. It can take any Object.

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.

Calling eg.:

```
maximum( 6.6, 8.8, 7.7 )
```

Collections Framework Examples

Java has provided generic support in Collections Framework Interfaces like List, Set, Map, etc.

List

```
List<T> list = new ArrayList<T>();
```

- **list** – object of List interface.
- **T** – The generic type parameter passed during List declaration.

The T is a type parameter passed to the generic interface List and its implementation class ArrayList.

Set

```
Set<T> set = new HashSet<T>();
```

- **set** – object of Set Interface.

- **T** – The generic type parameter passed during Set declaration.

The T is a type parameter passed to the generic interface Set and its implementation class HashSet.

Map

```
Map<T> set = new HashMap<T>();
```

- **set** – object of Map Interface.
- **T** – The generic type parameter passed during Map declaration.

The T is a type parameter passed to the generic interface Map and its implementation class HashMap.

Generics Wild Cards

The question mark (?), represents the wildcard, stands for unknown type in generics.

Upper Bounded Wildcards

There may be times when you'll want to restrict the kinds of types that are allowed to be passed to a type parameter. For example, a method that operates on numbers might only want to accept instances of Number or its subclasses.

To declare a upper bounded Wildcard parameter, list the ?, followed by the extends keyword, followed by its upper bound.

```
public static double sum(List<? extends Number> numberlist) {
      ...
   }
```

Unbounded Wildcards

There may be times when any object can be used when a method can be implemented using functionality provided in the Object class or When the code is independent of the type parameter.

To declare a Unbounded Wildcard parameter, list the ? only.

```
public static void printAll(List<?> list) {
    ...
   }
```

Lower Bounded Wildcards

There may be times when you'll want to restrict the kinds of types that are allowed to be passed to a type parameter. For example, a method that operates on numbers might only want to accept instances of Integer or its superclasses like Number.

To declare a lower bounded Wildcard parameter, list the ?, followed by the super keyword, followed by its lower bound.

```
public static void addCat(List<? super Cat> catList) {
      ...
   }
...
//You can add list of Cat or Animal (super class of the Cat class)
addCat(animalList);
addCat(catList);
```

Type Erasure

Generics are used for tighter type checks at compile time and to provide a generic programming. To implement generic behaviour, java compiler apply type erasure. Type erasure is a process in which compiler replaces a generic parameter with actual class or bridge method. In type erasure, compiler ensures that no extra classes are created and there is no runtime overhead.

Type Erasure rules:

- Replace type parameters in generic type with their bound if bounded type parameters are used.
- Replace type parameters in generic type with Object if unbounded type parameters are used.
- Insert type casts to preserve type safety.
- Generate bridge methods to keep polymorphism in extended generic types.

Restrictions on Generics

No Primitive Types - Using generics, primitive types can not be passed as type parameters.

```
Box<int> intBox = new Box<int>() //Error
```

NOTE: Use Wrappers like Integer instead.

No Instance - A type parameter cannot be used to instantiate its object inside a method.

```
public static <T> void add(Box<T> box) //Error
```

NOTE: To achieve such functionality, reflection can be used.

No Static field - Using generics, type parameters are not allowed to be static. As static variable is shared among object so compiler can not determine which type to used.

```
class Box<T> {
   private static T t; //Error
}
```

No Cast - Casting to a parameterized type is not allowed unless it is parameterized by unbounded wildcards.

```
Box<Integer> integerBox = new Box<Integer>();
Box<Number> numberBox = new Box<Number>();
integerBox = (Box<Integer>)numberBox; //Error: Cannot cast from Box<
```

NOTE: To achieve the same, unbounded wildcards can be used.

No instanceOf - Because compiler uses type erasure, the runtime does not keep track of type parameters, so at runtime difference between Box and Box cannot be verified using instanceOf operator.

```
... integerBox instanceof Box<Integer> ...
```

No Array - Arrays of parameterized types are not allowed. Because compiler uses type erasure, the type parameter is replaced with Object and user can add any type of object to the array. And at runtime, code will not able to throw ArrayStoreException.

```
Object[] stringBoxes = new Box<String>[]; //Error
```

No Exception - A generic class is not allowed to extend the Throwable class directly or indirectly.

```
//The generic class Box<T> may not subclass java.lang.Throwable
class Box<T> extends Exception {}
class Box1<T> extends Throwable {}
```

A method is not allowed to catch an instance of a type parameter.

```
... catch (T e) ...
```

No Overload - A class is not allowed to have two overloaded methods that can have the same signature after type erasure.

```
...
public void print(List<String> stringList) { }  // Error
public void print(List<Integer> integerList) { }
```