***Generics*:**

Generic programming is a style of computer programming in which algorithms are written in terms of ***types to-be-specified-later*** that are then instantiated when needed for specific types provided as parameters. In java Generics were added by ***JDK version 5***.

A class, interface, or method that operates on a parameterized type is called *generic,* as in *generic class* or *generic method.*

Need of Generics

1. ***Stronger type checks at compile time:***
   A Java compiler applies strong type checking to generic code and issues errors if the code violates type safety. Fixing compile-time errors is easier than fixing runtime.

2. ***Elimination of type casts.***
   Without generics it requires code for type casting.

3. ***Enabling programmers to implement generic algorithms.***
   By using generics, programmers can implement generic algorithms that work on collections of different types, can be customized, and are type safe and easier to read.

Benefits of Generics

1. Generics provide type safety.
2. Generally perform better for storing and manipulating value types.

Generics Usage

Generics are one of the most Java language feature. Generics enable *types* (classes and interfaces) to be parameters when defining classes, interfaces and methods. ***Type parameters provide a way to re-use the same code with different data types.***

Basics of Generic Types

A *generic type* is a generic class or interface that is parameterized over types. The following Box class will be modified to demonstrate the concept.

A Simple Box Class

Begin by examining a non-generic Box class that operates on objects of any type. It needs only to provide two methods: set, which adds an object to the box, and get, which retrieves it:

```
public class Box {
private Object object;
public void set(Object object) { this.object = object; }
public Object get() { return object; }
}
```

Since its methods accept or return an Object, you are free to pass in whatever you want, provided that it is not one of the primitive types. There is no way to verify, at compile time, how the class is used. One part of the code may place an Integer in the box and expect to get Integers out of it, while another part of the code may mistakenly pass in a String, resulting in a runtime error.

A Generic Version of the Box Class

A *generic class* is defined with the following format:

```
class name<T1, T2, ..., Tn> { /* ... */ }
```

The type parameter section, delimited by angle brackets (<>), follows the class name. It specifies the *type parameters* (also called *type variables*) T1, T2, ..., and Tn.

To update the Box class to use generics, you create a *generic type declaration* by changing the code "public class Box" to "public class Box<T>". This introduces the type variable, T, that can be used anywhere inside the class.

With this change, the Box class becomes:

```
public class Box<T> {
// T stands for "Type"
private T t;
public void set(T t) { this.t = t; }
public T get() { return t; }
}
```

As you can see, all occurrences of Object are replaced by T. A type variable can be any **non-primitive** type you specify: any class type, any interface type, any array type, or even another type variable.
This same technique can be applied to create generic interfaces.

Type Parameter Naming Conventions

By convention, type parameter names are single, uppercase letters. This stands in sharp contrast to the variable naming conventions that you already know about, and with good reason: Without this convention, it would be difficult to tell the difference between a type variable and an ordinary class or interface name.
The most commonly used type parameter names are:

- E - Element (used extensively by the Java Collections Framework)
- K - Key
- N - Number
- T - Type
- V - Value
- S,U,V etc. - 2nd, 3rd, 4th types

Type Wildcards

Java Generic's wildcards is a mechanism in Java Generics aimed at making it possible to cast a collection of a certain class, In generic code, the question mark (?), called the *wildcard*, represents an unknown type. The wildcard can be used in a variety of situations: The wildcard is never used as a type argument for a generic method or a generic class.

Using Type Wildcards
The generic wildcards target two primary needs:

- Reading from a generic collection
- Inserting into a generic collection

There are three ways to define a collection (variable) using generic wildcards. These are:

```
List<?>          listUknown = new ArrayList<A>();
List<? extends A> listUknown = new ArrayList<A>();
List<? super   A> listUknown = new ArrayList<A>();
```

Consider the example

```
public class A { }
public class B extends A { }
public class C extends A { }
```

The Unknown Wildcard

List<?> means a list typed to an unknown type. This could be a List<A>, a List<B>, a List<String> etc.

The extends Wildcard Boundary

List<? extends A> means a List of objects that are instances of the class A, or subclasses of A (e.g. B and C).

The super Wildcard Boundary

List<? super A> means that the list is typed to either the A class, or a superclass of A.

When you know that the list is typed to either A, or a superclass of A, it is safe to insert instances of A or subclasses of A (e.g. B or C) into the list.

Note: why B and C:
Since both B and C extend A, if A had a superclass, B and C would also be instances of that superclass.

Generic Methods

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).

Following example illustrates how we can print array of different type using a single Generic method:

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

   public static void main( String args[] )
   {
        // Create arrays of Integer, Double and Character
        Integer[] intArray = { 1, 2, 3, 4, 5 };
        Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };
        Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };

        System.out.println( "Array integerArray contains:" );
        printArray( intArray  ); // pass an Integer array

        System.out.println( "\nArray doubleArray contains:" );
        printArray( doubleArray ); // pass a Double array

        System.out.println( "\nArray characterArray contains:" );
        printArray( charArray ); // pass a Character array
    }
}
```

This would produce the following result:

```
Array integerArray contains:
1 2 3 4 5 6

Array doubleArray contains:
1.1 2.2 3.3 4.4

Array characterArray contains:
H E L L O
```

Bound Types

There may be times when you want to restrict the types that can be used as type arguments in a parameterized type. 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.*

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

Following example illustrates how extends is used in a general sense to mean either "extends" (as in classes) or "implements" (as in interfaces). This example is Generic method to return the largest of three Comparable objects:

```
import java.io.*;
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( "Maxm 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" ) );
  }
}
```

This would produce the following result:

```
Maximum of 3, 4 and 5 is 5
Maximum of 6.6, 8.8 and 7.7 is 8.8

Maximum of pear, apple and orange is pear
```

Generic Constructors

It is also possible for constructors to be generic, even if their class is not. For example, consider
the following short program:

```
// Use a generic constructor.
class GenCons {
private double val;
<T extends Number> GenCons(T arg) {
val = arg.doubleValue();
}
void showval() {
System.out.println("val: " + val);
}
}
class GenConsDemo {
public static void main(String args[]) {
GenCons test = new GenCons(100);
GenCons test2 = new GenCons(123.5F);
test.showval();
test2.showval();
}
}
```

The output is shown here:

```
val: 100.0
val: 123.5
```