Java Generics

JDK 5 introduces generics, which supports abstraction over types (or parameterized types) on **classes** and **methods**.

The class or method designers can be generic about types in the definition, while the users are to provide the specific types **(actual type)** during the object instantiation or method invocation.

Java Generics

You are certainly familiar with passing arguments into methods. You place the arguments inside the round bracket **()** and pass them into the method.

In generics, instead of passing arguments, we pass type information inside the angle brackets **<>.**

The primary usage of generics is to abstract over types for the **Collection Framework.**

Auto-Boxing/Unboxing between Primitives and their Wrapper Objects (JDK 5)

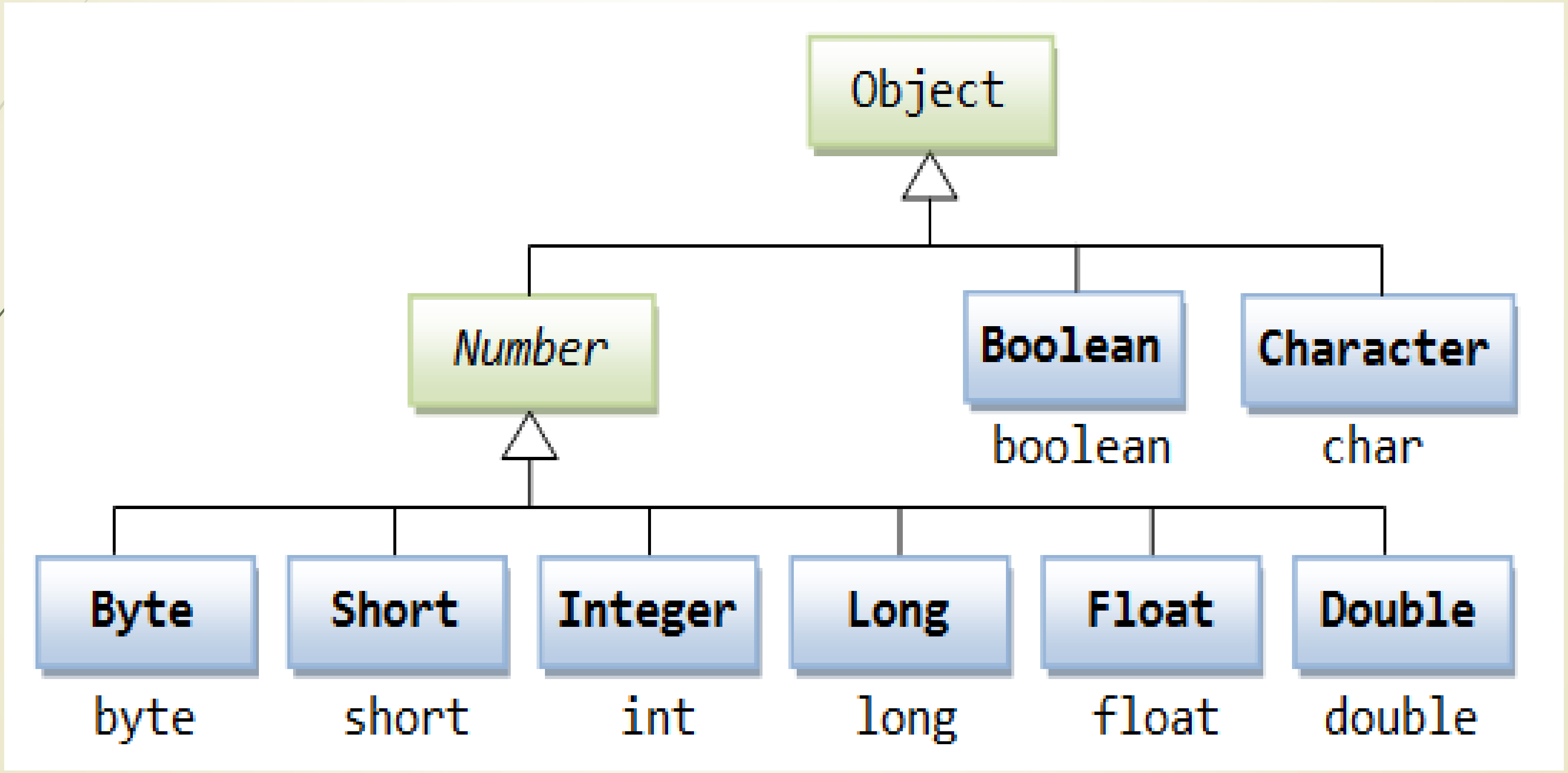
A Java Collection (such as List and Set) contains only objects.

It cannot holds primitives (such as **int** and **double**).

On the other hand, arrays can hold primitives and objects, but they are not resizable.

To put a primitive into a **Collection** (such as **ArrayList**), you have to wrap the primitive into an object using the corresponding primitive wrapper class as shown in image (Next Slide)

Auto-Boxing/Unboxing between Primitives and their Wrapper Objects (JDK 5)



Generics in Java

WITHOUT GENERICS

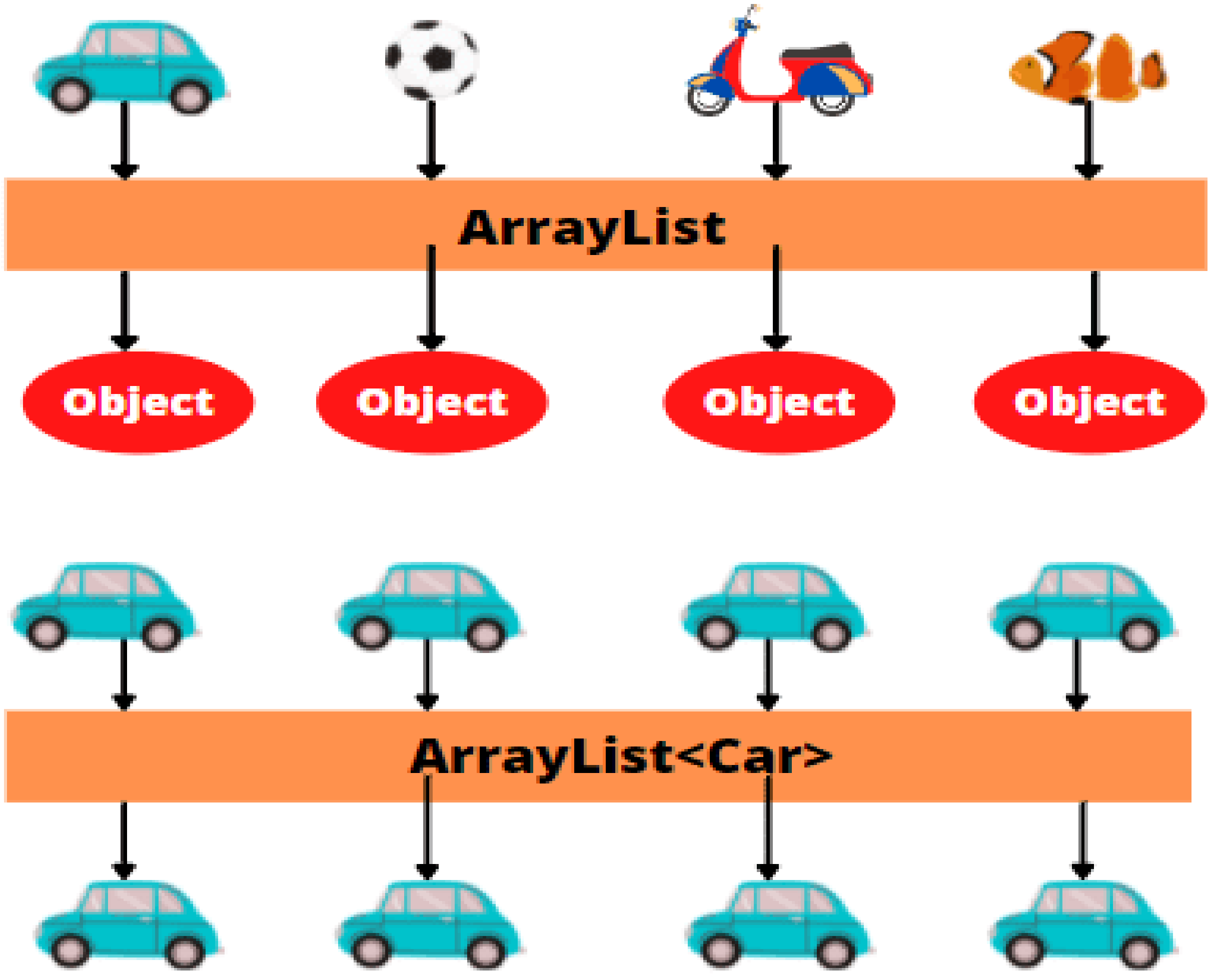
Objects go IN as a reference to Car, Football, Scooter, and Fish objects

And come OUT as a reference of type Object.

WITH GENERICS

Objects go IN as a reference to only Car objects

And come OUT as a reference of type Car.



Generic in Java

Generics in Java is a mechanism that allows writing code for one type **(say T)** that is applicable for all types of data, instead of writing separate classes for each type.

An element such as a class, interface, or method that works on parameterized type is called generic in java.

Note that whenever a type parameter T is being declared, it must be specified within angle brackets. Since **Myclass** uses a type parameter, **Myclass** is a generic class, which is also called parameterized type.

The best example of generic is the collection classes. For example, we declare **ArrayList** with type parameter like **ArrayList<String>**, **ArrayList<Integer>**, **ArrayList<Person>,** etc.

Features of Generics in Java

Java Generics adds type safety to the code. Using generics, we can hold only a single type of object. It does not allow to store other types of objects.

Generics allows dynamic binding.

It helps to detect and fix errors at compile time rather than runtime. A generic class or method permits to specify a type of object that the class or method can work with.

Generics helps to simplify the code by reducing ambiguity and eliminating the need for explicit typecasting.

Features of Generics in Java

Java permits to define generic classes, interfaces, and methods from Java 1.5 version. Several classes and interfaces in the Java API were modified using generics.

Java also allows declaring a generic constructor.

Java Generics work only with objects. i.e. While creating an object of a generic type, the type parameter must be a class type. It cannot be primitive types such as int, float, char, etc.

For example:

Myclass<int> obj = new Myclass<int>(20);   // Error, cannot use primitive type.

Features of Generics in Java

A generic type can have more than one type parameter. To define two or more type parameters in generic type, simply use a comma-separated list. For example:

```
class Myclass<T, V>
{
   T obj1;
   V obj2;
}
```

Advantages of Generics in Java

There are several advantages of using generics in java that are as follows:

1. The main advantage of generics is reusability. We can reuse the same code for different data types.
2. Another advantage of generics is type safety.
3. Using generics, we can eliminate typecasting.
4. It helps to avoid **ClassCastException** at runtime.

Syntax to Declare Generic Class in Java

The general syntax to declare a generic class in Java using a generic parameter **(type parameter) <T>** is as follows:

```
class class-name<T>
{
   // class code.
}
Or, class class-name<type-parameter-list>      {
    // class code
}
```

For example:

```
class Myclass<T>
{
    // class code
}
```

Syntax to Declare Generic Class in Java

The general syntax for the declaration of a simple generic class having two parameters **T** and **V** is as follows:

```
class Myclass<T, V>
{
  // class code.
}
```

How to Create an Object of Generic Class in Java?

The general syntax to create an object of generic class in Java is as follows:

```
class-name<type-arg-list>  var-name = new  class-name<type-arg-list>(cons-arg-list);
```

For example:

```
// creating a generic type object of Integer type.
Myclass<Integer> obj = new Myclass<Integer>();
```

Java Diamond Syntax

We can just leave it to the Java compiler to understand the types from the type declaration.

For example, the diamond syntax to create a generic class instance in java is as follows:

```
Myclass<String>  obj  =  new  Myclass<>();
```

Generic Method in Java

Similar to generic classes, we can also create generic methods. A method that takes generic type parameters is called generic method in Java.

A generic type can be applied for the static method. We can define a generic method by putting generic type parameter **<T>** before the method return type and immediately after the keyword static.

The general syntax to declare a generic method in Java is as follows:

```
public static <T> void display(T[ ]  list)
{
  // method code;
}
```

How to Call Generic Method in Java?

To call a generic method in Java, prefix the method name with the actual type in angle brackets. The general syntax to call a generic method is as follows:

```
class-name.<T>method-name(argument-list);
```

For example:

```
Myclass.<Integer>display(arr);    // Here, arr is the array reference.
```

or simply invoke it as follows:

```
display(arr);
display(strings);
```

Points to remember:

1. A generic class represents that is **type-safe.**

2. A generic class and a generic method can handle any type of data.

3. Generics can work only with objects of any class. They cannot work on **primitive data types.**

4. A generic class and a generic method eliminate the need of re-writing code every time there is a change in a data type.

5. Java compiler constructs a non-generic version of the class by putting the specified data type in a generic class. This is called **erasure**.

Points to remember:

6. By using generic types, we can easily eliminate type casting in many cases.

7. All the classes of **java.util** package has been revised using generics.

8. We cannot create an object of generic type parameter. For example,

9. class Myclass<T> // Here, T is generic parameter.

```
T  obj  =  new  T();        // Invalid.
```

Generic Constructor in Java

Generic Constructor in Java

When we define a generic type parameter to the constructor of a class, it is called **generic constructor in Java.**

We can use a generic constructor for both **generic** and **non-generic classes**.

```
// General Syntax:
// A generic class.
class Myclass<X>
{
// Generic constructor.
   <T> Myclass(T  t)
   {
      // constructor code.
   }
public static void main(String[ ] args)
{
  // Instantiate the class.
     Myclass<Integer>  obj  =  new Myclass<>(" ");
  }
}
```

Generic Interface in Java

Similar to generic class, when an interface is defined using generic type parameter, it is called generic interface in Java.

When a generic interface is implemented by either a generic class or non-generic class, the type argument must be specified.

The generic interface has main two advantages that are as:

1. **A generic interface can be implemented for different types of data.**
2. **It allows placing constraints (i.e. bounds) on the types of data for which interface can be implemented.**

Generic Interface in Java

Syntax Declaration of **Generic Interface**

The general syntax to declare a generic interface in Java is as follows:

```
interface  interface-name<T>
{
   void  method-name(T  t);      // public abstract method.
}
```

In the above syntax, **<T>** is called a **generic** type parameter that specifies any data type used in the interface. An interface can also have more than one type parameter separated by a **comma**.

Generic Interface in Java

The general syntax to write an implementation **class** that implements the **interface**, as follows:

```
class  class-name<T>  implements  interface-name<T>
{
  public void  method-name(T  t)
{
 // Write code for this method as per requirement.
 }
}
```

Restrictions on generic (Important points to remember)

1. It is not possible to create an instance of a type parameter. The compiler doesn't know what type of object to create, their **T** is simply a **placeholder**.

2. It is illegal for the **static members(variables, methods)** to use a type parameter declared by the enclosing class.

3. We cannot instantiate an array whose element type is a type parameter. There is no way for the compiler to know what type of array to actually create.

4. **Primitive data types** are not used with the generic types.

5. We cannot create **generic exception classes** and cannot extend **throwable** (which is superior to all **exception classes** in the exception class hierarchy).

Bounded Type Parameter

Type parameter with on or more bounds

Syntax:

```
<TypeParameter  extends bound1 & bound2 & …>
{
     …..
}
```

Bounded Type Parameter (Conti..)

Type parameter with on or more bounds

Example:

```
 class GenericDemo <T extends List>
{
      ……
}
GenericDemo<List> Gd = new GenericDemo<>(); // valid
GenericDemo<ArrayList> Gd = new GenericDemo<>();  // valid
GenericDemo<LinkedList> Gd = new GenericDemo<>();  // valid
GenericDemo<Collection> Gd = new GenericDemo<>();  // not valid
```

Bounded Type Parameter (Conti..)

Can access methods defined by bounds

`class GenericDemo<T>`
`{`
 	
`void  methodName(T list)`
`   {`
`      int  l =  list.size();`
`   }`
`}`
 	
`// compile time error`
	`class GenericDemo<T  extends  List>`
	`{`
	
	`void  methodName(T  list)`
	`   {`
	`      int  l =  list.size();`
	`   }`
	`}`
	
	`// No error`

Bounded Type Parameter (Conti..)

Valid bounds:	Invalid bounds
1. class 2. interface 3. enum 4. Parametrized Type	array primitive
Example of parametrized Type < T extends Comparable<T> >	
Note: for both class and interface we will use extends keyword not implements keyword for the interface. It means every type parameter bound with class and interface	

Specifics of Bound type parameters

- ✓ **Type argument must be subtype of all bounds**

```
class GenericDemo <T extends List & Serializable >

    GenericDemo<List>  test = new GenericDemo<>(); // Error

    GenericDemo<ArrayList> test new GenericDemo<>(); // valid
```

Note: ArrayList class is a subtype of **List** as well as **Serializable** both. So, it is valid

Specifics of Bound type parameters (Cont..)

- ✓ If class is one of the bounds, it must be first
- ✓ First bound is class, means remaining must be interfaces
- ✓ Final class & enums, type argument is bound itself

Generic methods and wild-card character (?) :

methodOne(ArrayList<String> l):

This method is applicable for **ArrayList** of only String type.

Example:

```
l.add("A");
l.add(null);
l.add(10);    //(invalid)
```

Within the method we can add only String type of objects and null to the List.

Generic methods and wild-card character (?) (Conti..)

methodOne(ArrayList<?> l):

We can use this method for **ArrayList** of any type but within the method we can't add anything to the List except null.

Example:

```
l.add(null);  //(valid)
l.add("A");  //(invalid)
l.add(10);   //(invalid)
```

Generic methods and wild-card character (?) (Conti..)

- **methodOne(ArrayList<? Extends x> l):**

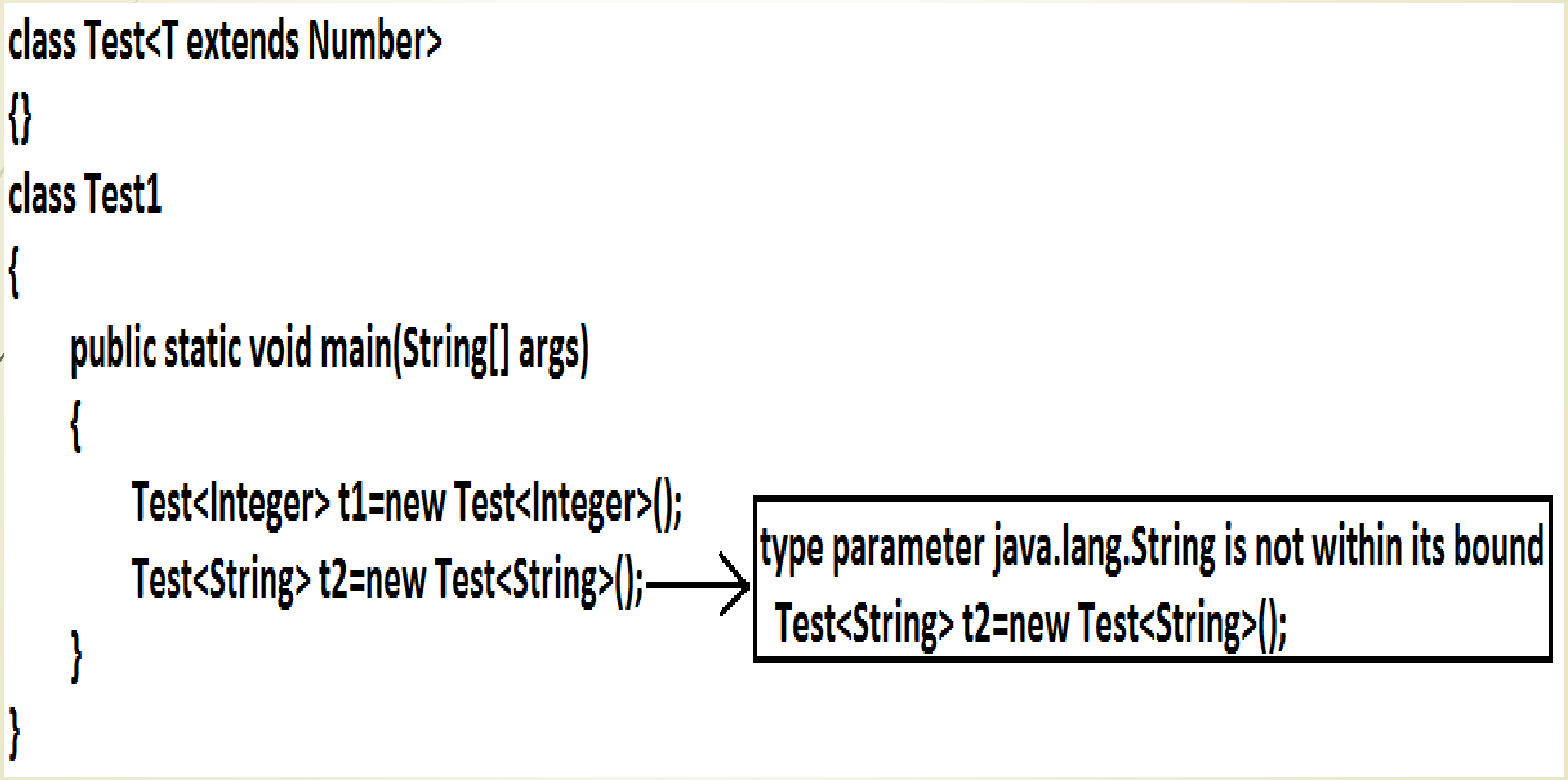
- If x is a class then this method is applicable for **ArrayList** of either x type or its **child** classes.
- If x is an interface then this method is applicable for **ArrayList** of either x type or its **implementation** classes.
- In this case also within the method we can't add anything to the **List** except **null**.

Generic methods and wild-card character (?) (Conti..)

methodOne(ArrayList<? super x> l):

- ✓ If x is a class then this method is applicable for **ArrayList** of either x type or its **super** classes.
- ✓ If x is an **interface** then this method is applicable for **ArrayList** of either x type or **super** classes of **implementation** class of x.
- ✓ But within the method we can add x type objects and null to the List.

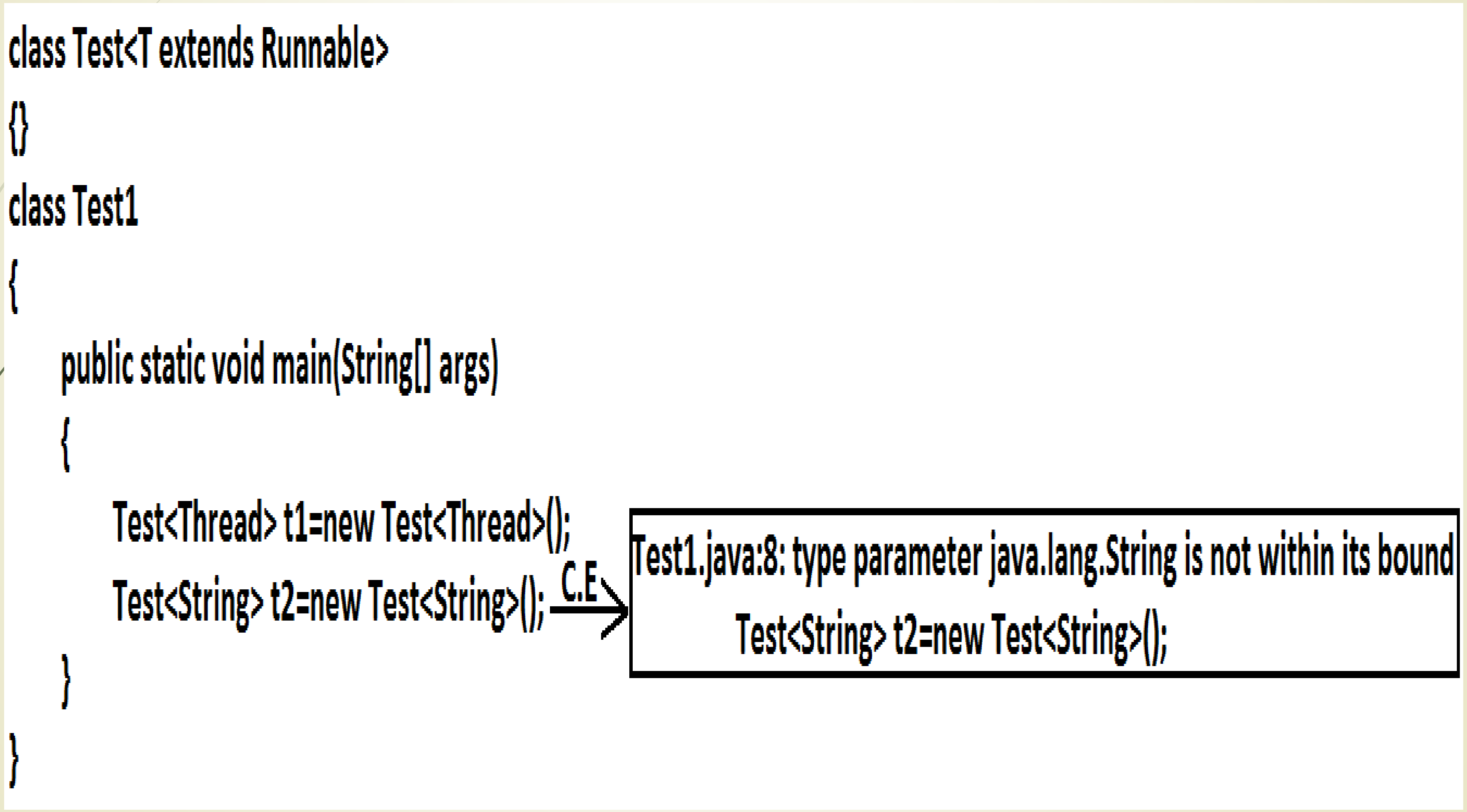
Example 1:



```
class Test<T extends Number>
{}
class Test1
{
    public static void main(String[] args)
    {
        Test<Integer> t1=new Test<Integer>();
        Test<String> t2=new Test<String>();
    }
}
```

```
type parameter java.lang.String is not within its bound
 Test<String> t2=new Test<String>();
```

Example 2:

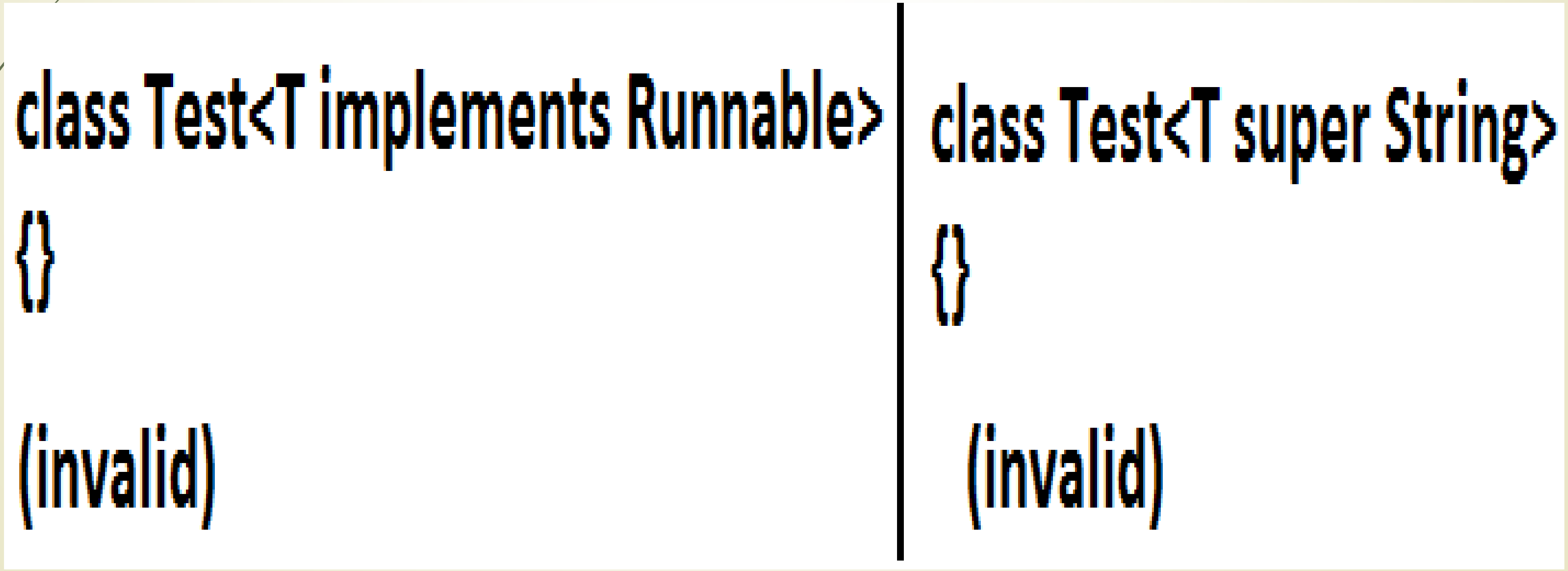


```
class Test<T extends Runnable>
{}
class Test1
{
    public static void main(String[] args)
    {
        Test<Thread> t1=new Test<Thread>();
        Test<String> t2=new Test<String>(); C.E
    }
}
```

```
Test1.java:8: type parameter java.lang.String is not within its bound
        Test<String> t2=new Test<String>();
```

Example 3:

1. We can't define bounded types by using implements and super keyword.
2. But implements keyword purpose we can replace with extends keyword.



```
class Test<T implements Runnable>
{}

(invalid)
```

```
class Test<T super String>
{}

(invalid)
```

Which of the following declarations are valid?

1. ArrayList<String> l1 = new ArrayList<String>();
2. ArrayList<?> l2 = new ArrayList<String>();
3. ArrayList<?> l3 = new ArrayList<Integer>();
4. ArrayList<? extends Number> l4 = new ArrayList<Integer>();
5. ArrayList<? extends Number> l5 = new ArrayList<String>();
6. ArrayList<?> l6=new ArrayList<? extends Number>();
7. ArrayList<?> l7=new ArrayList<?>();

Conclusions :

- Generics concept is applicable only at compile time, at runtime there is no such type of concept. Hence the following declarations are equal.

```
ArrayList  l1  =  new   ArrayList<String>();
ArrayList  l2  =  new   ArrayList<Integer>();
ArrayList  l3  =  new   ArrayList();
```

// All are equal at runtime.

Which one is valid or invalid ?

HashMap<Integer, String> h=new HashMap<Integer, String>();

We can define bounded types even in combination also.

Example 1:

class Test <T extends Number & Runnable> {}(valid)

As the type parameter we can pass any type which extends Number class and implements Runnable interface.

Example 2:

class Test<T extends Number & Runnable & Comparable> { } // (valid)

Example 3:

class Test<T extends Number & String> {} // (invalid)

We can't extend more than one class at a time.

Which one is valid or invalid ?

Example 4:

class Test<T extends Runnable & Comparable> { } // (valid)

Example 5:

class Test<T extends Runnable & Number> { } // (invalid)

We have to take 1st class followed by interface.

Declaring type parameter at method level:

We have to declare just before return type.

Example:

1. public <T> void methodOne1(T t){}   //valid
2. public <T extends Number> void methodOne2(T t){}   //valid
3. public <T extends Number & Comparable> void methodOne3(T t){} //valid
4. public <T extends Number & Comparable & Runnable> void methodOne4(T t){} //valid
5. public <T extends Number & Thread> void methodOne(T t){}   //invalid
6. public <T extends Number & Thread> void methodOne(T t){} //valid
7. public <T extends Runnable & Number> void methodOne(T t){} //invalid
8. public <T extends Number & Runnable> void methodOne(T t){} //valid

Example 1:

```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        ArrayList l=new ArrayList<String>();
        l.add(10);
        l.add(10.5);
        l.add(true);
        System.out.println(l);    // [10, 10.5, true]
    }
}
```

Example 2:

```
import java.util.*;
class Test
{
	public void methodOne(ArrayList<String> l){}
	public void methodOne(ArrayList<Integer> l){}
}
```

Output:

Compile time error.

```
Test.java:4: name clash: methodOne(java.util.ArrayList<java.lang.String>)
 and methodOne(java.util.ArrayList<java.lang.Integer>)
   have the same erasure
	public void methodOne(ArrayList<String> l){}
```