Generics,Enum,Annotations

By:

Sujata Batra

Generics

- Java Generics programming is introduced in J2SE 5 to deal with type-safe objects.
- Generics enable types (classes and interfaces) to be parameters when defining classes, interfaces and methods.

Benefits of Generics

- 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 errors, which can be difficult to find.
- Elimination of casts.
  - The following code snippet without generics requires casting:
    - List list = new ArrayList();
      list.add("hello");
      String s = (String) list.get(0);
      When re-written to use generics, the code does not require casting:
      List<String> list = new ArrayList<String>();
      list.add("hello");
      String s = list.get(0); // no cast
- 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.

Generic Types

- A class that can refer to any type is known as generic class.
- A generic type is a generic class or interface that is parameterized over types.
- 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.

Example

```
public class Box<T> {
    // T stands for "Type"
    private T t;

    public void set(T t) { this.t = t; }
    public T get() { return t; }
}
```

Type Parameter Naming Conventions

- Type parameter names are single, uppercase letters to make it easily distinguishable from java variables.
- 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

Generic Methods

- Generic methods are methods that introduce their own type parameters.
- Static and non-static generic methods are allowed, as well as generic class constructors.

```
public class TestGenerics4{

   public static < E > void printArray(E[] elements) {
        for ( E element : elements){
           System.out.println(element );
        }
        System.out.println();
    }
    public static void main( String args[] ) {
        Integer[] intArray = { 10, 20, 30, 40, 50 };
        Character[] charArray = { ‘S', ‘U', ‘J', 'A', 'T',‘A' };

        System.out.println( "Printing Integer Array" );
        printArray( intArray  );

       System.out.println( "Printing Character Array" );
        printArray( charArray );
    }
}
```

Bounded Type Parameters

- To restrict the types that can be used as type arguments in a parameterized type.
- To declare a bounded type parameter, list the type parameter's name, followed by the extends keyword, followed by its upper bound.

```
<T extends B1>

public class NaturalNumber<T extends Integer> {

      private T n;

      public NaturalNumber(T n)  { this.n = n; }

      public boolean isEven() {
         return n.intValue() % 2 == 0;
    }

    // ...
}
```

```
public class Box<T> {

    private T t;

    public void set(T t) {
        this.t = t;
    }

    public T get() {
        return t;
    }

    public <U extends Number> void inspect(U u){
        System.out.println("T: " + t.getClass().getName());
        System.out.println("U: " + u.getClass().getName());
    }

    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<Integer>();
        integerBox.set(new Integer(10));
        integerBox.inspect("some text"); // error: this is
still String!
    }
}
```

Multiple Bounds

- a type parameter can have multiple bounds:
- A type variable with multiple bounds is a subtype of all the types listed in the bound. If one of the bounds is a class, it must be specified first. For example:

```
class A { /* ... */ }
interface B { /* ... */ }
interface C { /* ... */ }

class D <T extends A & B & C> { /* ... */ }
```

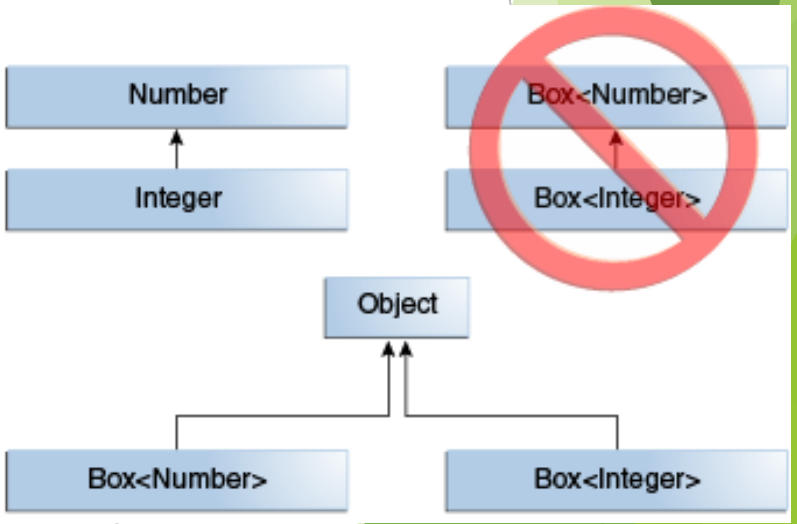
If bound A is not specified first, you get a compile-time error:

```
class D <T extends B & A & C> { /* ... */ }  // compile-time error
```

Generics, Inheritance, and Subtypes

- For class types, subtyping works. You can assign a derived type object to its base type reference. For generic type parameters, however, subtyping does not work. You cannot assign a derived generic type parameter to a base type parameter.
- You can perform a generic type invocation, e.g
  Box<Number> box = new Box<Number>();
  box.add(new Integer(10)); // OK
  box.add(new Double(10.1)); // OK

- Now consider the following method:
  public void boxTest(Box<Number> n) { /* ... */ }
  What type of argument does it accept?
- It accepts a single argument whose type is Box<Number>
- You are not allowed to pass in Box<Integer> or Box<Double>.



Bounded Wild Card

- **The question mark (?), called the wildcard, represents an unknown type.**

- **Wildcard Arguments With An Unknown Type -> “?”**
  - can hold any type of objects.
- **Bound -> " ? extends Type "**
  - stands for the family of all types that are subtypes of Type

- **Lower Bound -> " ? super Type "**
  - stands for the family of all types that are super types of Type.
  - ‘super’ clause is used to specify the lower bound for only wildcard arguments. It does not work with bounded types.

- used as arguments for instantiation of generic types.

- useful in situations where only partial knowledge about the type argument of a parameterized type is needed,

Bounded

```
public class Example_Super {

public static void display(List<? super Sub2> obj)
{
obj.add(new Sub2());
obj.get(0);
//obj.set(0, obj);
}

public static void show(List<? extends Parent > obj)
{
//obj.add(new Sub2());
//obj.set(0,new Sub2());
obj.get(0);

}
```

Bounded-Test Class

```
public static void main(String[] args) {

ArrayList<Parent> p1 = new ArrayList<Parent>();

display(p1);
ArrayList<Sub1> p2 = new ArrayList<Sub1>();
display(p2);

ArrayList<Sub2> p3 = new ArrayList<Sub2>();
display(p3);

show(p3);
}
```

Methods with Generic Wild Card (?)

- **The Classes**

```
class Person
{
String name;
String address;
……..
…….
}

class Employee extends Person
{

private int empid;
private String desig;
……..
……
}
```

The Method and Test Class

```
public  static void  displayItems(ArrayList<? extends Person> list)
   {
      list.add(new Person("ramesh","chennai"));

     for(Person mylist:list)
     System.out.println(mylist);
   }

public static void main(String[] args) {

     ArrayList< Person> alis = new ArrayList<Person>();
     alis.add(ge1);
     alis.add(ge2);
     displayItems(alis);
   }
```

Not Allowed , compile time Exception

Type Erasure

- Generics were introduced to the Java language to provide tighter type checks at compile time and to support generic programming. To implement generics, the Java compiler applies type erasure to:

  - Replace all type parameters in generic types with their bounds or Object if the type parameters are unbounded. The produced bytecode, therefore, contains only ordinary classes, interfaces, and methods.
  - Insert type casts if necessary to preserve type safety.

```
class GenericClassOne<T>
{
    T t;     //T will be replaced by java.lang.Object when compiled
}

class GenericClassTwo<T extends Number>
{
    T t;     //T will be replaced by java.lang.Number when compiled
}
```

This is how the two classes look after compilation

```
class GenericClassOne extends java.lang.Object
{
    java.lang.Object t;
}

class GenericClassTwo extends java.lang.Object
{
    java.lang.Number t;
}
```

- As type parameters are erased after compilation. They don't exist at run time. That's why you can't instantiate a type parameter. It gives compile time error.

```
class GenericClass<T>
{
    T t = new T();       //Compile time error

    <V> void genericMethod()
    {
        V v = new V();    //Compile time error
    }
}
```

Restrictions on Generics

- Cannot Instantiate Generic Types with Primitive Types.

Consider the following parameterized type:

```
class Pair<K, V> {

    private K key;
    private V value;

    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    // ...
}
```

When creating a Pair object, you cannot substitute a primitive type for the type parameter K or V:

```
Pair<int, char> p = new Pair<>(8, 'a');  // compile-time error
```

You can substitute only non-primitive types for the type parameters K and V:

```
Pair<Integer, Character> p = new Pair<>(8, 'a');
```

Restrictions on Generics (Continued)

- Cannot Create Instances of Type Parameters

For example, the following code causes a compile-time error:

```
public static <E> void append(List<E> list) {
    E elem = new E();  // compile-time error
    list.add(elem);
}
```

Restrictions on Generics (Continued)

- Cannot Declare Static Fields Whose Types are Type Parameters

A class's static field is a class-level variable shared by all non-static objects of the class. Hence, static fields of type parameters are not allowed. Consider the following class:

```
public class MobileDevice<T> {
    private static T os;

    // ...
}
```

Restrictions on Generics (Continued)

- Cannot Use Casts or instanceof with Parameterized Types

Because the Java compiler erases all type parameters in generic code, you cannot verify which parameterized type for a generic type is being used at runtime:

```
public static <E> void rtti(List<E> list) {
    if (list instanceof ArrayList<Integer>) {  // compile-time error
        // ...
    }
}
```

Restrictions on Generics (Continued)

- Cannot Create Arrays of Parameterized Types

You cannot create arrays of parameterized types. For example, the following code does not compile:

```
List<Integer>[] arrayOfLists = new List<Integer>[2];  // compile-time error
```

Restrictions on Generics (Continued)

- Cannot Create, Catch, or Throw Objects of Parameterized Types

A generic class cannot extend the Throwable class directly or indirectly. For example, the following classes will not compile:

```
// Extends Throwable indirectly
class MathException<T> extends Exception { /* ... */ }    // compile-time error

// Extends Throwable directly
class QueueFullException<T> extends Throwable { /* ... */ // compile-time error
```

A method cannot catch an instance of a type parameter:

```
public static <T extends Exception, J> void execute(List<J> jobs) {
    try {
        for (J job : jobs)
            // ...
    } catch (T e) {   // compile-time error
        // ...
    }
}
```

You can, however, use a type parameter in a throws clause:

```
class Parser<T extends Exception> {
    public void parse(File file) throws T {     // OK
        // ...
    }
}
```

Restrictions on Generics (Continued)

- Cannot Overload a Method Where the Formal Parameter Types of Each Overload Erase to the Same Raw Type

A class cannot have two overloaded methods that will have the same signature after type erasure.

```
public class Example {
    public void print(Set<String> strSet) { }
    public void print(Set<Integer> intSet) { }
}
```

The overloads would all share the same classfile representation and will generate a compile-time error.

Generics And Their Inheritance

- A generic class can extend a non-generic class.

```
class NonGenericClass
{
    //Non Generic Class
}

class GenericClass<T> extends NonGenericClass
{
   //Generic class extending non-generic class
}
```

Generics And Their Inheritance(Continued)

- Generic class can also extend another generic class.
  - sub class should have at least same type and same number of type parameters and at most can have any number and any type of parameters

```
class GenericSuperClass<T>
{
    //Generic super class with one type parameter
}
class GenericSubClass1<T> extends GenericSuperClass<T>
{
    //sub class with same type parameter
}

class GenericSubClass2<T, V> extends GenericSuperClass<T>
{
    //sub class with two type parameters
}
class GenericSubClass3<T1, T2> extends GenericSuperClass<T>
{
    //Compile time error, sub class having different type of parameters
}
```

Generics And Their Inheritance(Continued)

- When generic class extends another generic class
  - the type parameters are passed from sub class to super class

```
class GenericSuperClass<T>
{
    T t;

    public GenericSuperClass(T t)
    {
        this.t = t;
    }
}
class GenericSubClass<T> extends GenericSuperClass<T>
{
    public GenericSubClass(T t)
    {
        super(t);
    }
}
public class GenericsInJava
{
    public static void main(String[] args)
    {
        GenericSubClass<String> gen = new GenericSubClass<String>("I am string");

        System.out.println(gen.t);       //Output : I am string
    }
}
```

Generics And Their Inheritance(Continued)

- A generic class can extend only one generic class and one or more generic interfaces.
  - It's type parameters should be union of type parameters of generic class and generic interface(s).

```
class GenericSuperClass<T1>
{
    //Generic class with one type parameter
}

interface GenericInterface1<T1, T2>
{
    //Generic interface with two type parameters
}

interface GenericInterface2<T2, T3>
{
    //Generic interface with two type parameters
}

class GenericClass<T1,T2, T3> extends GenericSuperClass<T1> implements GenericInterface1<T1, T2>,
GenericInterface2<T2, T3>
{
    //Class having parameters of both the interfaces and super class
}
```

Generics And Their Inheritance(Continued)

- Non-generic class can’t extend generic class except of those generic classes which have already pre defined types as their type parameters.

```
{
    //Generic class with one type parameter
}

class NonGenericClass extends GenericSuperClass<T>
{
    //Compile time error, non-generic class can't extend generic class
}

class A
{
    //Pre defined class
}

class GenericSuperClass1<A>
{
    //Generic class with pre defined type 'A' as type parameter
}

class NonGenericClass1 extends GenericSuperClass1<A>
{
    //No compile time error, It is legal
}
```

Generics And Their Inheritance(Continued)

- Non-generic class can extend generic class by removing the type parameters. i.e as a raw type. But, it gives a warning.

```
class GenericClass<T>
{
    T t;

    public GenericClass(T t)
    {
        this.t = t;
    }
}

class NonGenericClass extends GenericClass       //Warning
{
    public NonGenericClass(String s)
    {
        super(s);          //Warning
    }
}

public class GenericsInJava
{
    public static void main(String[] args)
    {
        NonGenericClass nonGen = new NonGenericClass("I am String");

        System.out.println(nonGen.t);    //Output : I am String
    }
}
```

Generics And Their Inheritance(Continued)

- While extending a generic class having bounded type parameter, type parameter must be replaced by either upper bound or it's sub classes.

```
class GenericSuperClass<T extends Number>
{
    //Generic super class with bounded type parameter
}

class GenericSubClass1 extends GenericSuperClass<Number>
{
    //type parameter replaced by upper bound
}

class GenericSubClass2 extends GenericSuperClass<Integer>
{
    //type parameter replaced by sub class of upper bound
}

class GenericSubClass3 extends GenericSuperClass<T extends Number>
{
    //Compile time error
}
```

Generics And Their Inheritance(Continued)

- Generic methods of super class can be overrided in the sub class like normal methods.

```
class GenericClass
{
    <T> void genericMethod(T t)
    {
        System.out.println(1);
    }
}

class NonGenericClass extends GenericClass
{
    @Override
    <T> void genericMethod(T t)
    {
            System.out.println(2);
    }
}
```

```
public class GenericsInJava
{
    public static void main(String[] args)
    {
        new GenericClass().genericMethod("I am String");       //Output : 1

        new NonGenericClass().genericMethod("I am String");    //Output : 2
    }
}
```

Enums

- Enums are classes that extend **java.lang.Enum.**
- Each declared value is an instance of the enum.
- Enum values are **public**, **static** and **final.**
- Enums can be compared with == or equals().
- Enums are Comparable and Serializable
- The Enum type can be declared with a constructor, and each enum constant is declared with parameters to be passed to the constructor.

ENUM -Examples

```
enum language {JAVA,J2EE,FORTRAN}

public class Enum_Ex1 {

   enum dbprds {ORACLE,DB2,MYSQL}

       private dbprds database;
        private  language prglang;

        public static void main(String[] args) {

       Enum_Ex1  myObj = new Enum_Ex1();

     myObj.database = dbprds.ORACLE;
     myObj.prglang = language.JAVA;

     System.out.println(myObj.database);
     System.out.println(myObj.prglang);
   }

}
```

```
public class Enum_Ex2 {

      flashDrive usbDrives;

    enum flashDrive {SANDISK(1) ,SONY(2),PHILIPS(3);
    private int size;

    flashDrive(int size)
    {
     this.size=size;
    }
    public int getCap()
    {
       return this.size;
    }
     }
     public static void main(String[] args) {

     Enum_Ex2  myObj = new Enum_Ex2();

        myObj.usbDrives = flashDrive.SANDISK;

     System.out.println("Capacity :"+myObj.usbDrives.size+" Mb");
    } }
```

```
public class Enum_Ex5 {

   enum Servers {WEBLOGIC,WEBSPHERE,ORACLE,APACHE};

   public static void main(String[] args) {

     Servers myserver=Servers.WEBLOGIC;

     switch (myserver)
     {

     case WEBLOGIC:
          System.out.println("BEA");
          break;
     case WEBSPHERE:
          System.out.println("IBM");
          break;
     case APACHE:
          System.out.println("Open Source");
           break;
     }
```

```
        for(Servers s:Servers.values())
        {
            System.out.println(s);
        }

        myserver = Servers.APACHE;

        if(myserver==Servers.APACHE)
        {
            System.out.println("Good Choice for WEB Applications");
        }

        if(Servers.APACHE.equals(Servers.ORACLE))
            System.out.println("equals");
        else System.out.println("not equal");

        }

}
```

EnumSet

- EnumSet is a special Set implementation, only applicable for Enums.
- Can only store instances of the single enum type.
- Adding an instance of different enum will result in compile time error, as EnumSet provide type-safety.

```
public class EnumSetDemo {
    private enum Color {
        RED(255, 0, 0), GREEN(0, 255, 0), BLUE(0, 0, 255);
        private int r;
        private int g;
        private int b;
        private Color(int r, int g, int b) {
            this.r = r;
            this.g = g;
            this.b = b;
        }
        public int getR() {
            return r;
            }
        public int getG() {
            return g;
        }
        public int getB() {
            return b;
        }
    }
```

```
public static void drawLine(Set<Color> colors) {
	System.out.println("Requested Colors to draw lines : " + colors);
		for (Color c : colors) {
			System.out.println("drawing line in color : " + c);
	}
}
public static void main(String args[]) {
	// this will draw line in yellow color
	EnumSet<Color> yellow = EnumSet.of(Color.RED, Color.GREEN);
	drawLine(yellow);
	// RED + GREEN + BLUE = WHITE
	EnumSet<Color> white = EnumSet.of(Color.RED, Color.GREEN, Color.BLUE);
	drawLine(white);
	// RED + BLUE = PINK
	EnumSet<Color> pink = EnumSet.of(Color.RED, Color.BLUE);
	drawLine(pink);
}
}
```

Annotation

- used to provide meta data for your Java code.
- do not directly affect the execution of your code.
- annotations were added to Java from Java 5

Types Of Annotations

- Marker Annotation
  - Marker Annotations are used to mark a declaration.
  - A marker annotation is a special kind of annotation.
  - A marker annotation contains no members.
- Single Member Annotation
  - Single-Member Annotations contains only one member.
  - Single-Member Annotations allow a shorthand form of specifying the value.
  - The name of the member must be value.

Types of Annotation(Continued)

- Normal/Full Annotation
  - Annotations with multiple elements
  - Elements can have default values
  - When annotations are used ,values are assigned using name/value pairs.

Java Annotation Purposes

- Compiler instructions
- Build-time instructions
- Runtime instructions

Built-in Java Annotations

- @Deprecated
  - The @Deprecated annotation is used to mark a class, method or field as deprecated, meaning it should no longer be used.
  - When you use the @Deprecated annotation, it is a good idea to also use the corresponding @deprecated JavaDoc symbol, and explain why the class, method or field is deprecated, and what the programmer should use instead.
  - E.g. @Deprecated

```
/**
@deprecated Use MyNewComponent instead.
*/
public class MyComponent {
… }
```

- **@Override**
  - The @Override Java annotation is used above methods that override methods in a superclass.
  - If the method does not match a method in the superclass, the compiler will give you an error.
  - The @Override annotation is not necessary in order to override a method in a superclass.

```
public class MySuperClass {


    public void doTheThing() {
        System.out.println("Do the thing");
    }
}


public class MySubClass extends MySuperClass{
    @Override
    public void doTheThing() {
        System.out.println("Do it differently");
    }
}
```

- @SuppressWarnings
  - The @SuppressWarnings annotation makes the compiler suppress warnings for a given method.
  - For instance, if a method calls a deprecated method, or makes an insecure type cast, the compiler may generate a warning.

```
@SuppressWarnings
public void methodWithWarning() {
…

}
```