Diamond Operator Enhancements

This enhancement is as the part of Milling Project Coin (JEP 213).

Before understanding this enhancement, we should aware Generics concept, which has been introduced in java 1.5 version.

The main objectives of Generics are:

1. To provide Type Safety
2. To resolve Type Casting Problems.

Case 1: Type-Safety

Arrays are always type safe. i.e we can give the guarantee for the type of elements present inside array. For example if our programming requirement is to hold String type of objects, it is recommended to use String array. For the string array we can add only string type of objects. By mistake if we are trying to add any other type we will get compile time error.

Eg:

```
1)  String[] s = new String[100];
2)  s[0] = "Durga";
3)  s[1] = "Pavan";
4)  s[2] = new Integer(10);//error: incompatible types: Integer cannot be converted to String
```

String[] can contains only String type of elements. Hence, we can always give guarantee for the type of elements present inside array. Due to this arrays are safe to use with respect to type. Hence arrays are type safe.

But collections are not type safe that is we can't give any guarantee for the type of elements present inside collection. For example if our programming requirement is to hold only string type of objects, and if we choose ArrayList, by mistake if we are trying to add any other type we won't get any compile time error but the program may fail at runtime.

```
1)  ArrayList l = new ArrayList();
2)  l.add("Durga");
3)  l.add("Pavan");
4)  l.add(new Integer(10));
5)  ....
6)  String name1=(String)l.get(0);
7)  String name2=(String)l.get(1);
8)  String name3=(String)l.get(2);//RE: java.lang.ClassCastException
```

Hence we can't give any guarantee for the type of elements present inside collections. Due to this collections are not safe to use with respect to type, i.e collections are not Type-Safe.

Case 2: Type-Casting

In the case of array at the time of retrieval it is not required to perform any type casting.

```
1) String[] s = new String[100];
2) s[0] = "Durga";
3) s[1] = "Pavan";
4) ...
5) String name1=s[0];//Type casting is not required.
```

But in the case of collection at the time of retrieval compulsory we should perform type casting otherwise we will get compile time error.

Eg:

```
1) ArrayList l = new ArrayList();
2) l.add("Durga");
3) l.add("Pavan");
4) ....
5) String name1=l.get(0);//error: incompatible types: Object cannot be converted to String
6) String name1=(String)l.get(0);//valid
```

Hence in Collections Type Casting is bigger headache.

To overcome the above problems of collections(type-safety, type casting),Java people introduced Generics concept in 1.5version.Hence the main objectives of Generics are:

1. **To provide Type Safety to the collections.**
2. **To resolve Type casting problems.**

Example for Generic Collection:

To hold only string type of objects, we can create a generic version of ArrayList as follows.

```
ArrayList<String> l = new ArrayList<String>();
```

Here String is called Parameter Type.

For this ArrayList we can add only string type of objects. By mistake if we are trying to add any other type then we will get compile time error.

```
1) l.add("Durga");//valid
2) l.add("Ravi");//valid
3) l.add(new Integer(10));//error: no suitable method found for add(Integer)
```

Hence, through generics we are getting type safety.

At the time of retrieval it is not required to perform any type casting we can assign elements directly to string type variables.

String name1 = l.get(0);//valid and Type Casting is not required.

Hence, through generic syntax we can resolve type casting problems.

Difference between Generic and Non-Generic Collections:

ArrayList l = new ArrayList();	ArrayList<String> l = new ArrayList<String>();
It is Non Generic version of ArrayList	It is Generic version of ArrayList
For this ArrayList we can add any type of object and hence it is not Type-Safe.	For this ArrayList we can add only String type of Objects. By mistake if we are trying to add any other type,we will get compile time error.
At the time of retrieval compulsory we should perform Type casting.	At the time of retrieval, we are not required to perform Type casting.

Java 7 Diamond Operator (<>):

Diamond Operator '<>' was introduced in JDK 7 under project Coin.
The main objective of Diamond Operator is to instantiate generic classes very easily.

Prior to Java 7, Programmer compulsory should explicitly include the Type of generic class in the Type Parameter of the constructor.

```
ArrayList<String> l = new ArrayList<String>();
```

Whenever we are using Diamond Operator, then the compiler will consider the type automatically based on context, Which is also known as Type inference. We are not required to specify Type Parameter of the Constructor explicitly.

```
ArrayList<String> l = new ArrayList<>();
```

Hence the main advantage of Diamond Operator is we are not required to speicfy the type parameter in the constructor explicitly,length of the code will be reduced and readability will be improved.

Eg 2:

```
List<Map<String,Integer>> l = new ArrayList<Map<String,Integer>>();
```

can be writtern with Diamond operator as follows

```
List<Map<String,Integer>> l = new ArrayList<>();
```

But until Java 8 version we cannot apply diamond operator for Anonymous Generic classes. But in Java 9, we can use.

Usage of Diamond Operator for Anonymous Classes:

In JDK 9, Usage of Diamond Operator extended to Anonymous classes also.

Anonymous class:

Sometimes we can declare classes without having the name, such type of nameless classes are called Anonymous Classes.

Eg 1:

```
1)  Thread t = new Thread()
2)  {
3)  };
```

We are creating a child class that extends Thread class without name(Anonymous class) and we are creating object for that child class.

Eg 2:

```
1)  Runnable r = new Runnable()
2)  {
3)  };
```

We are creating an implementation class for Runnable interface without name(Anonymous class) and we are creating object for that implementation class.

Eg 3:

```
1)  ArrayList<String> l = new ArrayList<String>()
2)  {
3)  };
```

We are creating a child class that extends ArrayList class without name(Anonymous class) and we are creating object for that child class.

From JDK 9 onwards we can use Diamond Operator for Anonymous Classes also.

```
1)  ArrayList<String> l = new ArrayList<>()
2)  {
3)  };
```

It is valid in Java 9 but invalid in Java 8.

Demo Program - 1: To demonstrate usage of diamond operator for Anonymous Class

```
1)  class MyGenClass<T>
2)  {
3)     T obj;
4)     public MyGenClass(T obj)
5)     {
6)        this.obj = obj;
7)     }
8)
9)     public T getObj()
10)    {
11)       return obj;
12)    }
13)    public void process()
14)    {
15)       System.out.println("Processing obj...");
16)    }
17) }
18) public class Test
19) {
20)    public static void main(String[] args)
21)    {
22)          MyGenClass<String> c1 = new MyGenClass<String>("Durga")
23)          {
24)             public void process()
25)             {
26)               System.out.println("Processing... " + getObj());
27)             }
28)          };
29)          c1.process();
30)
31)          MyGenClass<String> c2 = new MyGenClass<>("Pavan")
32)          {
33)             public void process()
34)             {
35)               System.out.println("Processing... " + getObj());
36)             }
37)          };
38)          c2.process();
39)    }
40) }
```

Output:

```
Processing... Durga
Processing... Pavan
```

If we compile the above program according to Java 1.8 version,then we will get compile time error

```
D:\durga_classes>javac -source 1.8 Test.java
error: cannot infer type arguments for MyGenClass<T>
                MyGenClass<String> c2 = new MyGenClass<>("Pavan")
                                        ^
reason: cannot use '<>' with anonymous inner classes in -source 1.8
  (use -source 9 or higher to enable '<>' with anonymous inner classes)
```

Demo Program - 2: To demonstrate usage of diamond operator for Anonymous Class

```
1)  import java.util.Iterator;
2)  import java.util.NoSuchElementException;
3)  public class DiamondOperatorDemo
4)  {
5)    public static void main(String[] args)
6)    {
7)      String[] animals = { "Dog", "Cat", "Rat", "Tiger","Elephant" };
8)      Iterator<String> iter = new Iterator<>()
9)       {
10)        int i = 0;
11)       public boolean hasNext()
12)       {
13)         return i < animals.length;
14)       }
15)       public String next()
16)       {
17)         if (!hasNext())
18)           throw new NoSuchElementException();
19)         return animals[i++];
20)       }
21)     };
22)     while (iter.hasNext())
23)     {
24)       System.out.println(iter.next());
25)     }
26)   }
27) }
```

Output:

```
Dog
Cat
Rat
Tiger
Elephant
```

Note - 1:

```
1)  ArrayList<String> preJava7 = new ArrayList<String>();
2)  ArrayList<String> java7 = new ArrayList<>();
3)  ArrayList<String> java9 = new ArrayList<>()
4)  {
5)  };
```

Note - 2:

Be Ready for Partial Diamond Operator in the next versions of Java, as the part of Project "Amber". Open JDK people already working on this.

Eg: new Test<String,_>();