**Advanced Java Features**

**Generics**

Generics in Java is one of the powerful feature which enhances Type Safety and hence reduces the need of Type casting.

**Type Safety**

Reduces the need of ClasscastException and other type related errors.

package AdvancedJavaFeatures;  
  
import java.util.ArrayList;  
import java.util.List;  
import java.util.Objects;  
  
public class GenericDemo  
{  
 public static void main(String args[])  
 {  
 // without generics  
 List l1 = new ArrayList<>();  
 l1.add("Java");  
 l1.add("dhvhds");  
 // l1.add(1234);  
 for(Object s:l1)  
 {  
 System.*out*.println((String) s);  
 }  
  
 // with generics  
 List<String> list = new ArrayList<>();  
 list.add("Ram");  
 list.add("dcscgs");  
 // list.add(12333);  
 list.add("dshcdsvc");  
  
 }  
}

**Advantages**

* Compile Time  Type Checking
* Reduces Type casting
* Code Resuability

**Generic Class**

package AdvancedJavaFeatures;  
  
class Box<T>  
{  
 private T t;  
  
 public T getT() {  
 return t;  
 }  
  
 public void setT(T t) {  
 this.t = t;  
 }  
}  
  
public class GenericClass  
{  
 public static void main(String args[])  
 {  
 // Object of Box class with string data  
 Box<String> obj = new Box<>();  
 obj.setT("Java");  
 System.*out*.println(obj.getT());  
  
  
 // Object of Box class with Integer data  
 Box<Integer> obj1 = new Box<>();  
 obj1.setT(1234);  
 System.*out*.println(obj1.getT());  
 }  
}

**Generic Method**

package AdvancedJavaFeatures;  
  
public class GenericMethod  
{  
 public static <T> void show(T[] x)  
 {  
 for(T val:x)  
 System.*out*.print(val+" ");  
 }  
 public static void main(String[] args)  
 {  
 String arr[] = {"sgf","hscdsj","hc","qw6qw"};  
 Integer p [] = {1,4,7,8,0,3};  
 *show*(p);  
 System.*out*.println(" ");  
 *show*(arr);   
 }  
}

**Bounded Type Parameters:** We can restrict the type that can be sent as type arguments using Bounded Type.

package AdvancedJavaFeatures;  
  
public class BoundedType  
{  
 public <T extends Number> void display(T num)  
 {  
 System.*out*.println(num);  
 }  
 public static void main(String args[])  
 {  
 BoundedType obj = new BoundedType();  
 obj.display(123890.45789); // double value  
 obj.display(12345); // int  
 obj.display((float)12.45); // float  
 obj.display(12);// int  
 }  
}

**Wild Cards**

Wild Cards are used to represent unknown type.

---- Unbounded wildcard -- <?>

----Bounded wildcard -- <? extends T> and <? super T>

 package AdvancedJavaFeatures;  
  
import java.util.ArrayList;  
import java.util.List;  
  
public class WildCard  
{  
 public static void show(List<?> l1)  
 {  
 for(Object ob : l1)  
 System.*out*.println(ob);  
 }  
 public static void main(String args[])  
 {  
 List<String> l11 = new ArrayList();  
 l11.add("hvds");  
 l11.add("sdhgcds");  
 List<Integer> l12 = new ArrayList<>();  
 l12.add(111);  
 l12.add(12345);  
 *show*(l11);  
 *show*(l12);  
 }  
}

**Java Reflection and Annotations**

Java Reflection is a mechanisim that allows to introspect variables, methods , constructors and classes at runtime without knowing their names at compile time.

**Accessing Class Object( First create object of “Class”)**

1. getClass() from the object of a class
2. Class.forName() – this method  with full name of the class.
3. .class

package AdvancedJavaFeatures;  
  
import java.lang.reflect.Modifier;  
  
class Fruit  
{  
  
}  
class Apple extends Fruit  
{  
 public void colour()  
 {  
 System.*out*.println("Apple -- Red In Colour");  
 }  
}  
  
public class Reflection  
{  
 public static void main(String args[])  
 {  
 Apple obj = new Apple();  
 Class ob = obj.getClass();  
 String st = ob.getName();  
 System.*out*.println("Name of the Class :"+ st);  
 int modifier = ob.getModifiers();  
 String s1 = Modifier.*toString*(modifier);  
 System.*out*.println("Modifier :"+ s1);  
 Class sc = ob.getSuperclass();  
 System.*out*.println("Super Class : "+ sc.getName());  
 }  
}

**Reflection Method**

**Method ---**Is a class and it has various methods that can be used to get the info about the methods present in a class.

package AdvancedJavaFeatures;  
  
import java.lang.reflect.Method;  
import java.lang.reflect.Modifier;  
  
class Orange  
{  
 public void colour()  
 {  
 System.*out*.println("Orange In Colour");  
 }  
 private void taste()  
 {  
 System.*out*.println("Tangy in Taste");  
 }  
}  
  
public class ReflectionMethod  
{  
 public static void main(String args[])  
 {  
 Orange ob = new Orange();  
 Class obj = ob.getClass();  
 Method[] m = obj.getDeclaredMethods();  
 // get method name  
 for(Method m1: m)  
 {  
 System.*out*.println("Method Name :" + m1.getName());  
 //get return type  
 System.*out*.println("Return Type : " + m1.getReturnType());  
 }  
 int modifier = obj.getModifiers();  
 System.*out*.println("Modifiers : "+ Modifier.*toString*(modifier));  
 }  
}

**Annotations**

Are powerful feature that allows programers to provide metadata about their code.

It provides hint to the compiler at runtime .

Annotations can be applied to Classes, methods , Interfaces, variables and parameters.

**Annotations**

---------- Built In

---------- Custom( Userdefined)

**BuiltIn (**Use to provide hint to the compiler about the code.)

**Standard** - @Override, @FunctionalInterface, @SupressWarnings.

**MetaAnnotation ( -**@Target, @Inherited, @Documented.)

**@ Target -**  @Target annotation is a meta-annotation, i.e., it can only be used to annotate other annotations. It takes Element Type enumeration as its only argument. ElementType enumeration is a constant which specifies the type of the program element declaration (class, interface, constructor, etc.) to which the annotation can be applied.

**Custom Annotation**

Define an Interface with @interface annotation

It can be defined as methods with no parameters and a return type.

 package AdvancedJavaFeatures;  
  
import java.lang.annotation.Retention;  
import java.lang.annotation.RetentionPolicy;  
@Retention(RetentionPolicy.*RUNTIME*)  
  
@interface LogExecute  
{  
 String msg() default "Executed Time";  
}  
  
public class CustomAnnotations  
{  
 @LogExecute(msg ="my method execution time")  
 public void method1()  
 {  
  
 }  
  
}

**Lambda Expressions and Functional Programming**

Lambda Expressions – Is an anonymous function (function without name) and with parameters and body.

An -> (arrow) symbol is used.

Syntax:

() -> expression.

(parameters) -> { body }

@FunctionalInterface  
interface Operate<T>  
{  
 //abstract method  
 T calculate(T a, T b);  
  
}  
class Operations  
{  
 public static void main(String args[])  
 {  
 Operate<Integer> rf = (a,b)-> a+b;  
 System.*out*.println( rf.calculate(25,35));  
 Operate<Double> rf1 = (a,b)-> a+b;  
 System.*out*.println( rf1.calculate(25.789,35.567));  
 Operate<String> rf3 = (a,b)-> a+b;  
 System.*out*.println( rf3.calculate("356","456"));  
 }  
}

**ForEach Example**

List<String> l1 = new ArrayList<>();  
l1.add("sdhjs");  
l1.add("hcgdshc");  
l1.add("dewtd");  
  
l1.forEach(p ->{System.out.println(p);});

Lambda Expressions represents an instance of **Functional Interface.**

**Functional Interface (FI)**

Is an Interface which have only one abstract method but it can have any number of default static methods.

It supports Lambda Expressions

FI are used with Lambda Expressions, which represents an instance of the FI in a concise way.

------ User Defined FI

------ BuiltIn FI (present in java.util package)(Consumer, Function, Predicate,Supplier)

**Consumer**

It is a Functional Interface which takes a single argument and does not return any result.

Interface Consumer<T>

{

void accept(T t);

}

Example:

 package AdvancedJavaFeatures;  
  
import java.util.ArrayList;  
import java.util.Arrays;  
import java.util.List;  
import java.util.function.Consumer;  
  
public class ConsumerFunctionalInterface {  
 public static void main(String[] args) {  
 // Example 1: Doubling the numbers in the list  
 List<Integer> l1 = new ArrayList<>(Arrays.*asList*(11, 22, 33, 44, 55));  
 System.*out*.println("List with original data " + l1);  
  
 // using consumer functional interface  
 Consumer<List<Integer>> doubler = list -> {  
 for (int i = 0; i < list.size(); i++) {  
 list.set(i, list.get(i) \* 2);  
 }  
 };  
  
 // calling the accept method of consumer functional interface  
 doubler.accept(l1);  
 System.*out*.println("List after applying Consumer Accept method (Double Number) " + l1);  
  
 // Example 2: Converting strings to uppercase  
 List<String> l2 = new ArrayList<>(Arrays.*asList*("day14", "consumer", "functional", "interface", "example"));  
 System.*out*.println("List with original data " + l2);  
  
 // using consumer functional interface  
 Consumer<List<String>> converter = list -> {  
 for (int i = 0; i < list.size(); i++) {  
 list.set(i, list.get(i).toUpperCase());  
 }  
 };  
  
 // calling the accept method of consumer functional interface  
 converter.accept(l2);  
 System.*out*.println("List after applying Consumer Accept method (To Uppercase) " + l2);  
 }  
}

**Supplier**

The supplier interface returns the supplier of results.

It has one method get().

It does not take any argument but returns result.

Interface Supplier<T>

{

T get();

}

Example:

package AdvancedJavaFeatures;  
  
import java.util.function.Supplier;  
  
public class SupplierFunctionalInterface  
{  
 public static void main(String args[])  
 {  
 Supplier<Double> result = Math::*random*;  
 System.*out*.println(result.get());  
 }  
}

**Predicate**

It’s a Functional Interface which takes one argument and returns a Boolean value.

It has one method test(T t).

Interface Predicate<T>

{

boolean test(T t);

}

Example:

package AdvancedJavaFeatures;  
  
import java.util.Arrays;  
import java.util.List;  
import java.util.function.Predicate;  
import java.util.stream.Collectors;  
  
public class PredicateFunctionalInterface {  
 public static void main(String args[])  
 {  
 List<String> li = Arrays.*asList*("Ram", "Shyam","Sita","Sam");  
// predicate to filter the strings starts with 'S'.  
 Predicate<String> withS = l1 -> l1.startsWith("S");  
 List<String> names = li.stream().filter(withS).collect(Collectors.*toList*());  
 System.*out*.println("Names starts with 'S'");  
 System.*out*.println(names);  
// predicate to filter the strings with length greater than 3  
 Predicate<String> length = l2 -> l2.length()>2;  
 List<String> length3 = li.stream().filter(length).collect(Collectors.*toList*());  
 System.*out*.println("Names greater than length 2");  
 System.*out*.println(length3);  
 }  
}

**Function**

It’s a Functional interface which takes one argument and returns result.

It is mainly used for data transformation.

Function<T,R>

Example:

package AdvancedJavaFeatures;  
  
import java.util.Arrays;  
import java.util.List;  
import java.util.function.Function;  
import java.util.stream.Collectors;  
  
public class FunctionFunctionalInterface  
{  
 public static void main(String args[])  
 {  
 List<String> li = Arrays.*asList*("Hi", "Hello"," How ","Oh ");  
 System.*out*.println("Greetings");  
 System.*out*.println(li);  
 Function<String,String> adding = s -> s+" !";  
 List<String> greetings = li.stream().map(adding).collect(Collectors.*toList*());  
 System.*out*.println("Greetings with Exclamation");  
 System.*out*.println(greetings);  
 }  
}

**BiPredicate**

It takes two arguments and returns Boolean value.

Example:

BiPredicate<Integer, Integer> greater = (x,y) -> x>y;  
System.out.println("Is the number greater : "+greater.test(50,59));

**BiConsumer**

It takes two arguments and performs some action on them.

Example:

BiConsumer<List<Integer>, List<Integer>> compare = (l11, l2) -> {  
 if (l11.size() == l2.size()) {  
 System.*out*.println("True");  
  
 }  
 else {  
 System.*out*.println("False");  
 }  
  
};  
List<Integer> aa = Arrays.*asList*(2,3,4);  
List<Integer> ab = Arrays.*asList*(2,3);  
 compare.accept(aa,ab);  
}

**BiFunction**

It takes two arguments and produces the result.

Example:

BiFunction<Integer,Integer,Integer> add = (x,y)-> x+y;  
System.out.println(add.apply(50,40));  
BiFunction<Double,Double,Double> sum = (x,y)-> x+y;  
System.out.println(sum.apply(500.345,400345.67));

**Stream API**

Stream API is one of the java 8 features by using this we processes collections of objects.

**Stream:**

It is sequence of data that need to be processed.

Streams take input from different sources like Collections, arrays.

Stream uses internal iteration to process data.

**Stream Operations:**

**Intermediate Operation:**Transformation (filter, sorted, map)

**Terminal Operation**: count, forEach.

**Lazy Loading –**because the computation is done after invoking terminal operation.

**Stream Operations:**

filter(Predicate<T>)

map(Function<T,R>)

collect(Collector<T,A,R>)

sorted(Comparator<T>)

Example:

package AdvancedJavaFeatures;  
  
import java.util.Arrays;  
import java.util.List;  
import java.util.stream.Collectors;  
  
public class StreamAPI  
{  
 public static void main(String args[])  
 {  
 List<Integer> num = Arrays.*asList*(23,14,30,06);  
 List<Integer> even = num.stream().filter(n -> n%2==0).collect(Collectors.*toList*());  
 System.*out*.println("Even Numbers: "+even);  
 int total = even.stream().mapToInt(Integer::intValue).sum();  
 System.*out*.println("Total Value: "+total);  
 List<Double> num1 = Arrays.*asList*(23.34,14.67,30.67,6.89);  
 double totaldoubl = num1.stream().mapToDouble(Double::doubleValue).max().getAsDouble();  
 System.*out*.println("Max Value: "+totaldoubl);  
 }  
}

**map and flatMap**

map – one to one

1. It returns stream containing the transformed elements  in the same order as of the original stream.
2. This transformation is one to one that means each element produces one output.

flatMap

flatMap - one to many

1. This transformation is one to many that means each element produces multiple outputs that later flattened.
2. flatMap = map+flattened

Example:

package AdvancedJavaFeatures;  
  
import java.util.Arrays;  
import java.util.List;  
import java.util.stream.Collectors;  
  
public class Maps  
{  
 public static void main(String args[])  
 { // map  
 List<String> l1 = Arrays.*asList*("11","22","33","44");  
 List<Double> l3 = l1.stream().map(Double::*valueOf*).collect(Collectors.*toList*());  
 System.*out*.println ("Original List : "+l1);  
 System.*out*.println ("List after map operation : "+l3);  
 // flatMap  
 List<List<Integer>> li = Arrays.*asList*(Arrays.*asList*(11,12,13),Arrays.*asList*(24,34,44),Arrays.*asList*(23,14,06,30));  
 List<Integer> flat = li.stream().flatMap(a->a.stream()).toList();  
  
 System.*out*.println ("Original List : "+li);  
 System.*out*.println ("List after map operation : "+flat);  
 }  
}