Java 8 New Features

java 7 – July 28th 2011
2 Years 7 Months 18 Days

Java 8 - March 18th 2014

Java 9 - September 22nd 2016

Java 10 - 2018

After Java 1.5version, Java 8 is the next major version.
Before Java 8, sun people gave importance only for objects but in 1.8version oracle people gave the importance for functional aspects of programming to bring its benefits to Java.ie it doesn't mean Java is functional oriented programming language.

Java 8 New Features:

1) **Lambda Expression**
2) **Functional Interfaces**
3) **Default methods**
4) **Predicates**
5) **Functions**
6) **Double colon operator (::)**
7) **Stream API**
8) **Date and Time API**

Etc…..

Lambda (λ) Expression

☀ Lambda calculus is a big change in mathematical world which has been introduced in 1930. Because of benefits of Lambda calculus slowly this concepts started using in programming world. "LISP" is the first programming which uses Lambda Expression.

☀ The other languages which uses lambda expressions are:
- C#.Net
- C Objective
- C
- C++
- Python
- Ruby etc.

and finally in Java also.

☀ The Main Objective of Lambda Expression is to bring benefits of functional programming into Java.

What is Lambda Expression (λ):

- Lambda Expression is just an anonymous (nameless) function. That means the function which doesn't have the name, return type and access modifiers.
- Lambda Expression also known as anonymous functions or closures.

Ex: 1

```
public void m1() {
    sop("hello");
}
```

```
() → {
        sop("hello");
    }
() → { sop("hello"); }
() → sop("hello");
```

Ex:2

```
public void add(inta, int b) {
    sop(a+b);
}
```

```
(inta, int b) → sop(a+b);
```

- If the type of the parameter can be decided by compiler automatically based on the context then we can remove types also.
- The above Lambda expression we can rewrite as (a,b) → sop (a+b);

Ex: 3

```
public String str(String str) {
        return str;
}
```

(String str) → return str;

↓

(str) → str;

Conclusions:

1) A lambda expression can have zero or more number of parameters (arguments).

   Ex:
   () → sop("hello");
   (int a) → sop(a);
   (inta, int b) → return a+b;

2) Usually we can specify type of parameter. If the compiler expects the type based on the context then we can remove type. i.e., programmer is not required.

   Ex:
   (inta, int b) → sop(a+b);

   ↓

   (a,b) → sop(a+b);

3) If multiple parameters present then these parameters should be separated with comma (,).

4) If zero number of parameters available then we have to use empty parameter [like ()].

   Ex: () → sop("hello");

5) If only one parameter is available and if the compiler can expect the type then we can remove the type and parenthesis also.

   Ex:
   (int a) → sop(a);

   ↓

   (a)→ sop(a);

   ↓

   A → sop(a);

6) Similar to method body lambda expression body also can contain multiple statements. If more than one statements present then we have to enclose inside within curly braces. If one statement present then curly braces are optional.

7) Once we write lambda expression we can call that expression just like a method, for this functional interfaces are required.

Functional Interfaces

If an interface contain only one abstract method, such type of interfaces are called functional interfaces and the method is called functional method or single abstract method (SAM).

Ex:

1) Runnable → It contains only run() method
2) Comparable → It contains only compareTo() method
3) ActionListener → It contains only actionPerformed()
4) Callable → It contains only call() method

Inside functional interface in addition to single Abstract method (SAM) we write any number of default and static methods.

Ex:

```
1)  interface Interf {
2)          public abstract void m1();
3)          default void m2() {
4)                  System.out.println ("hello");
5)          }
6)  }
```

In Java 8, Sun Micro System introduced @Functional Interface annotation to specify that the interface is Functional Interface.

Ex:

```
@Functional Interface
        Interface Interf {
            public void m1();
    }
```

This code compiles without any compilation errors.

Inside Functional Interface we can take only one abstract method, if we take more than one abstract method then compiler raise an error message that is called we will get compilation error.

Ex:

```
@Functional Interface {
        public void m1();
        public void m2();
}
```

This code gives compilation error.

Inside Functional Interface we have to take exactly only one abstract method.If we are not declaring that abstract method then compiler gives an error message.

Ex:

```
@Functional Interface {
        interface Interface {
}
```

compilation error

Functional Interface with respect to Inheritance:

If an interface extends Functional Interface and child interface doesn't contain any abstract method then child interface is also Functional Interface

Ex:

```
1)  @Functional Interface
2)  interface A {
3)          public void methodOne();
4)  }
5)  @Functional Interface
6)  Interface B extends A {
7)  }
```

In the child interface we can define exactly same parent interface abstract method.

Ex:

```
1)  @Functional Interface
2)  interface A {
3)          public void methodOne();
4)  }
5)  @Functional Interface
6)  interface B extends A {
7)          public void methodOne();
8)  }
```

No Compile Time Error

In the child interface we can't define any new abstract methods otherwise child interface won't be Functional Interface and if we are trying to use @Functional Interface annotation then compiler gives an error message.

```
1)  @Functional Interface {
2)  interface A {
3)        public void methodOne();
4)  }
5)  @Functional Interface
6)  interface B extends A {
7)        public void methodTwo();
8)  }
```

Compiletime Error

Ex:

```
@Functional Interface
interface A {
      public void methodOne();
}
interface B extends A {
       public void methodTwo();
}
```

No compile time error

This's Normal interface so that code compiles without error

In the above example in both parent & child interface we can write any number of default methods and there are no restrictions. Restrictions are applicable only for abstract methods.

Functional Interface Vs Lambda Expressions:

Once we write Lambda expressions to invoke it's functionality, then Functional Interface is required.
We can use Functional Interface reference to refer Lambda Expression.
Where ever Functional Interface concept is applicable there we can use Lambda Expressions

Ex:1 **Without Lambda Expression**

```
1)  interface  Interf {
2)        public void methodOne() {}
3)        public class Demo implements Interface {
4)            public void methodOne() {
5)                  System.out.println("method one execution");
6)            }
7)            public class Test {
8)                public static void main(String[] args) {
9)                      Interfi = new Demo();
10)                     i.methodOne();
11)               }
12) }
```

Above code With Lambda expression

```
1)  interface Interf {
2)        public void methodOne() {}
3)        class Test {
4)             public static void main(String[] args) {
5)                  Interfi = () → System.out.println("MethodOne Execution");
6)                  i.methodOne();
7)              }
8)  }
```

Without Lambda Expression

```
1)  interface Interf {
2)         public void sum(inta,int b);
3)  }
4)  class Demo implements Interf {
5)          public void sum(inta,int b) {
6)                System.out.println("The sum:"+(a+b));
7)          }
8)  }
9)  public class Test {
10)         public static void main(String[] args) {
11)                Interfi = new Demo();
12)                i.sum(20,5);
13)         }
14) }
```

Above code With Lambda Expression

```
1)  interface Interf {
2)       public void sum(inta, int b);
3)  }
4)  class Test {
5)        public static void main(String[] args) {
6)             Interfi = (a,b) → System.out.println("The Sum:" +(a+b));
7)             i.sum(5,10);
8)        }
9)  }
```

Without Lambda Expressions

```
1)  interface Interf {
2)          publicint square(int x);
3)  }
4)  class Demo implements Interf {
5)              public int square(int x) {
6)                      return x*x;  OR  (int x) → x*x
7)              }
8)  }
9)  class Test {
10)         public static void main(String[] args) {
11)             Interfi = new Demo();
12)             System.out.println("The Square of 7 is: " +i.square(7));
13)         }
14) }
```

Above code with Lambda Expression

```
1)  interface Interf {
2)          public int square(int x);
3)  }
4)  class Test {
5)          public static void main(String[] args) {
6)                  Interfi = x → x*x;
7)                  System.out.println("The Square of 5 is:"+i.square(5));
8)          }
9)  }
```

Without Lambda expression

```
1)  class MyRunnable implements Runnable {
2)              public void main() {
3)                      for(int i=0; i<10; i++) {
4)                              System.out.println("Child Thread");
5)                      }
6)              }
7)  }
8)  class ThreadDemo {
9)          public static void main(String[] args) {
10)                     Runnable r = new myRunnable();
11)                     Thread t = new Thread(r);
12)                     t.start();
13)                     for(int i=0; i<10; i++) {
14)                             System.out.println("Main Thread")
15)                     }
16)         }
```

```
17) }
```

With Lambda expression

```
1)  class ThreadDemo {
2)        public static void main(String[] args) {
3)              Runnable r = () → {
4)                    for(int i=0; i<10; i++) {
5)                          System.out.println("Child Thread");
6)                    }
7)              };
8)              Thread t = new Thread(r);
9)              t.start();
10)             for(i=0; i<10; i++) {
11)                   System.out.println("Main Thread");
12)             }
13)       }
14) }
```

Anonymous inner classes vs Lambda Expressions

Wherever we are using anonymous inner classes there may be a chance of using Lambda expression to reduce length of the code and to resolve complexity.

Ex: With anonymous inner class

```
1)  class Test {
2)      public static void main(String[] args) {
3)          Thread t = new Thread(new Runnable() {
4)                  public void run() {
5)                      for(int i=0; i<10; i++) {
6)                          System.out.println("Child Thread");
7)                      }
8)                  }
9)          });
10)         t.start();
11)         for(int i=0; i<10; i++)
12)             System.out.println("Main thread");
13)     }
14) }
```

With Lambda expression

```
1)  class Test {
2)        public static void main(String[] args) {
3)                Thread t = new Thread(() → {
4)                                                for(int i=0; i<10; i++) {
5)                                                    System.out.println("Child Thread");
6)                                                }
7)                });
8)                t.start();
9)                for(int i=0; i<10; i++) {
10)                   System.out.println("Main Thread");
11)               }
12)       }
13) }
```

What are the advantages of Lambda expression?

- We can reduce length of the code so that readability of the code will be improved.
- We can resolve complexity of anonymous inner classes.
- We can provide Lambda expression in the place of object.
- We can pass lambda expression as argument to methods.

Note:

- Anonymous inner class can extend concrete class, can extend abstract class, can implement interface with any number of methods but
- Lambda expression can implement an interface with only single abstract method (Functional Interface).
- Hence if anonymous inner class implements Functional Interface in that particular case only we can replace with lambda expressions. Hence wherever anonymous inner class concept is there, it may not possible to replace with Lambda expressions.
- Anonymous inner class! = Lambda Expression

- Inside anonymous inner class we can declare instance variables.
- Inside anonymous inner class "this" always refers current inner class object(anonymous inner class) but not related outer class object

Ex:

- Inside lambda expression we can't declare instance variables.
- Whatever the variables declare inside lambda expression are simply acts as local variables
- Within lambda expression 'this" keyword represents current outer class object reference (that is current enclosing class reference in which we declare lambda expression)

Ex:

```
1)  interface Interf {
2)          public void m1();
3)  }
4)  class Test {
5)          int x = 777;
6)          public void m2() {
7)                  Interfi = ()→ {
8)                          int x = 888;
9)                          System.out.println(x); 888
10)                         System.out.println(this.x); 777
11)                 };
12)                 i.m1();
13)         }
14)         public static void main(String[] args) {
15)                 Test t = new Test();
16)                 t.m2();
17)         }
18) }
```

- From lambda expression we can access enclosing class variables and enclosing method variables directly.
- The local variables referenced from lambda expression are implicitly final and hence we can't perform re-assignment for those local variables otherwise we get compile time error

Ex:

```
1)  interface Interf {
2)          public void m1();
3)  }
4)  class Test {
5)          int x = 10;
6)          public void m2() {
7)                  int y = 20;
8)                  Interfi = () → {
9)                          System.out.println(x); 10
10)                         System.out.println(y); 20
11)                         x = 888;
12)                         y = 999; //CE
13)                 };
14)                 i.m1();
15)                 y = 777;
16)         }
17)         public static void main(String[] args) {
18)                 Test t = new Test();
19)                 t.m2();
```

```
20)         }
21) }
```

Differences between anonymous inner classes and Lambda expression

Anonymous Inner class	Lambda Expression
It's a class without name	It's a method without name (anonymous function)
Anonymous inner class can extend Abstract and concrete classes	lambda expression can't extend Abstract and concrete classes
Anonymous inner class can implement An interface that contains any number of Abstract methods	lambda expression can implement an Interface which contains single abstract method (Functional Interface)
Inside anonymous inner class we can Declare instance variables.	Inside lambda expression we can't Declare instance variables, whatever the variables declared are simply acts as local variables.
Anonymous inner classes can be Instantiated	lambda expressions can't be instantiated
Inside anonymous inner class "this" Always refers current anonymous Inner class object but not outer class Object.	Inside lambda expression "this" Always refers current outer class object. That is enclosing class object.
Anonymous inner class is the best choice If we want to handle multiple methods.	Lambda expression is the best Choice if we want to handle interface With single abstract method (Functional Interface).
In the case of anonymous inner class At the time of compilation a separate Dot class file will be generated (outerclass$1.class)	At the time of compilation no dot Class file will be generated for Lambda expression. It simply converts in to private method outer class.
Memory allocated on demand Whenever we are creating an object	Reside in permanent memory of JVM (Method Area).

Default Methods

- Until 1.7 version onwards inside interface we can take only public abstract methods and public static final variables (every method present inside interface is always public and abstract whether we are declaring or not).
- Every variable declared inside interface is always public static final whether we are declaring or not.
- But from 1.8 version onwards in addition to these, we can declare default concrete methods also inside interface, which are also known as defender methods.
- We can declare default method with the keyword "default" as follows

```
1)  default void m1(){
2)  System.out.println ("Default Method");
3)  }
```

- Interface default methods are by-default available to all implementation classes. Based on requirement implementation class can use these default methods directly or can override.

Ex:

```
1)  interface Interf {
2)          default void m1() {
3)                  System.out.println("Default Method");
4)          }
5)  }
6)  class Test implements Interf {
7)          public static void main(String[] args) {
8)                  Test t = new Test();
9)                  t.m1();
10)         }
11) }
```

- Default methods also known as defender methods or virtual extension methods.
- The main advantage of default methods is without effecting implementation classes we can add new functionality to the interface (backward compatibility).

Note: We can't override object class methods as default methods inside interface otherwise we get compile time error.

Ex:

```
1)  interface Interf {
2)        default inthashCode() {
3)              return 10;
4)        }
5)  }
```

CompileTimeError

Reason: **Object class methods are by-default available to every Java class hence it's not required to bring through default methods.**

Default method vs multiple inheritance

Two interfaces can contain default method with same signature then there may be a chance of ambiguity problem (diamond problem) to the implementation class. To overcome this problem compulsory we should override default method in the implementation class otherwise we get compile time error.

```
1)  Eg 1:
2)  interface Left {
3)        default void m1() {
4)              System.out.println("Left Default Method");
5)        }
6)  }
7)
8)  Eg 2:
9)  interface Right {
10)       default void m1() {
11)             System.out.println("Right Default Method");
12)       }
13) }
14)
15) Eg 3:
16) class Test implements Left, Right {}
```

How to override default method in the implementation class?

In the implementation class we can provide complete new implementation or we can call any interface method as follows.
interfacename.super.m1();

Ex:

```
1)  class Test implements Left, Right {
2)          public void m1() {
3)                System.out.println("Test Class Method"); OR Left.super.m1();
4)          }
5)          public static void main(String[] args) {
6)                  Test t = new Test();
7)                  t.m1();
8)          }
9)  }
```

Differences between interface with default methods and abstract class

Even though we can add concrete methods in the form of default methods to the interface, it won't be equal to abstract class.

Interface with Default Methods	Abstract Class
Inside interface every variable is Always public static final and there is No chance of instance variables	Inside abstract class there may be a Chance of instance variables which Are required to the child class.
Interface never talks about state of Object.	Abstract class can talk about state of Object.
Inside interface we can't declare Constructors.	Inside abstract class we can declare Constructors.
Inside interface we can't declare Instance and static blocks.	Inside abstract class we can declare Instance and static blocks.
Functional interface with default Methods Can refer lambda expression.	Abstract class can't refer lambda Expressions.
Inside interface we can't override Object class methods.	Inside abstract class we can override Object class methods.

Interface with default method != abstract class

Static methods inside interface:

- From 1.8 version onwards in addition to default methods we can write static methods also inside interface to define utility functions.
- Interface static methods by-default not available to the implementation classes hence by using implementation class reference we can't call interface static
- methods. We should call interface static methods by using interface name.

Ex:

```
1)  interface Interf {
2)          public static void sum(int a, int b) {
3)                  System.out.println("The Sum:"+(a+b));
4)          }
5)  }
6)  class Test implements Interf {
7)           public static void main(String[] args) {
8)                  Test t = new Test();
9)                  t.sum(10, 20); //CE
10)                 Test.sum(10, 20); //CE
11)                 Interf.sum(10, 20);
12)         }
13) }
```

- As interface static methods by default not available to the implementation class, overriding concept is not applicable.
- Based on our requirement we can define exactly same method in the implementation class, it's valid but not overriding.

Ex:1

```
1)  interface Interf {
2)          public static void m1() {}
3)  }
4)  class Test implements Interf {
5)           public static void m1() {}
6)  }
```

It's valid but not overriding

Ex:2

```
1)  interface Interf {
2)        public static void m1() {}
3)  }
4)  class Test implements Interf {
5)        public void m1() {}
6)  }
```

This's valid but not overriding

Ex3:

```
1)  class P {
2)       private void m1() {}
3)  }
4)  class C extends P {
5)        public void m1() {}
6)  }
```

This's valid but not overriding

From 1.8 version onwards we can write main() method inside interface and hence we can run interface directly from the command prompt.

Ex:

```
1)  interface Interf {
2)        public static void main(String[] args) {
3)              System.out.println("Interface Main Method");
4)        }
5)  }
```

At the command prompt:
Javac Interf.Java
JavaInterf

Predicates

- A predicate is a function with a single argument and returns boolean value.
- To implement predicate functions in Java, Oracle people introduced Predicate interface in 1.8 version (i.e.,Predicate<T>).
- Predicate interface present in *Java.util.function* package.
- It's a functional interface and it contains only one method i.e., test()

Ex:

```
interface Predicate<T> {
    public boolean test(T t);
}
```

As predicate is a functional interface and hence it can refers lambda expression

Ex:1 Write a predicate to check whether the given integer is greater than 10 or not.

Ex:

```
public boolean test(Integer I) {
        if (I >10) {
            return true;
        }
        else {
            return false;
        }
}
```

```
(Integer I) → {
            if(I > 10)
                return true;
            else
                return false;
        }
```

I → (I>10);

predicate<Integer> p = I →(I >10);
System.out.println (p.test(100)); true
System.out.println (p.test(7)); false

Program:

```
1)  import Java.util.function;
2)  class Test {
3)        public static void main(String[] args) {
4)            predicate<Integer> p = I → (i>10);
5)            System.out.println(p.test(100));
6)            System.out.println(p.test(7));
7)            System.out.println(p.test(true)); //CE
8)        }
9)  }
```

1 Write a predicate to check the length of given string is greater than 3 or not.
Predicate<String> p = s → (s.length() > 3);
System.out.println (p.test("rvkb")); true
System.out.println (p.test("rk")); false

#-2 write a predicate to check whether the given collection is empty or not.
Predicate<collection> p = c → c.isEmpty();

Predicate joining

It's possible to join predicates into a single predicate by using the following methods.
and()
or()
negate()
these are exactly same as logical AND ,OR complement operators

Ex:

```
1)  import Java.util.function.*;
2)  class test {
3)        public static void main(string[] args) {
4)            int[] x = {0, 5, 10, 15, 20, 25, 30};
5)            predicate<integer> p1 = i->i>10;
6)            predicate<integer> p2=i -> i%2==0;
7)            System.out.println("The Numbers Greater Than 10:");
8)            m1(p1, x);
9)            System.out.println("The Even Numbers Are:");
```

```
10)             m1(p2, x);
11)             System.out.println("The Numbers Not Greater Than 10:");
12)             m1(p1.negate(), x);
13)             System.out.println("The  Numbers Greater Than 10 And Even Are:â€");
14)             m1(p1.and(p2), x);
15)             System.out.println("The Numbers Greater Than 10 OR Even:â€");
16)             m1(p1.or(p2), x);
17)         }
18)          public static void m1(predicate<integer>p, int[] x) {
19)                 for(int x1:x) {
20)                     if(p.test(x1))
21)                         System.out.println(x1);
22)                     }
23)                 }
24) }
```

Functions

- **Functions are exactly same as predicates except that functions can return any type of result but function should (can) return only one value and that value can be any type as per our requirement.**
- **To implement functions oracle people introduced Function interface in 1.8version.**
- **Function interface present in *Java.util.function* package.**
- **Functional interface contains only one method i.e., apply()**

```
interface function(T,R) {
        public R apply(T t);
}
```

Assignment: **Write a function to find length of given input string.**

Ex:

```
1)  import Java.util.function.*;
2)  class Test {
3)          public static void main(String[] args) {
4)                  Function<String, Integer> f = s ->s.length();
5)                  System.out.println(f.apply("Durga"));
6)                  System.out.println(f.apply("Soft"));
7)          }
8)  }
```

Note: **Function is a functional interface and hence it can refer lambda expression.**

Differences between predicate and function

Predicate	Function
To implement conditional checks We should go for predicate	To perform certain operation And to return some result we Should go for function.
Predicate can take one type Parameter which represents Input argument type. Predicate<T>	Function can take 2 type Parameters. First one represent Input argument type and Second one represent return Type. Function<T,R>
Predicate interface defines only one method called test()	Function interface defines only one Method called apply().
public boolean test(T t)	public R apply(T t)
Predicate can return only boolean value.	Function can return any type of value

Note: Predicate is a boolean valued function and(), or(), negate() are default methods present inside Predicate interface.

Method and Constructor references by using ::(double colon) operator

- Functional Interface method can be mapped to our specified method by using :: (double colon) operator. This is called method reference.
- Our specified method can be either static method or instance method.
- Functional Interface method and our specified method should have same argument types, except this the remaining things like
- returntype, methodname, modifiersetc are not required to match.

Syntax:

if our specified method is static method

Classname::methodName

if the method is instance method

Objref::methodName

Functional Interface can refer lambda expression and Functional Interface can also refer method reference. Hence lambda expression can be replaced with method reference. Hence method reference is alternative syntax to lambda expression.

Ex: With Lambda Expression

```
1)  class Test {
2)      public static void main(String[] args) {
3)              Runnable r = () → {
4)                                  for(int i=0; i<=10; i++) {
5)                                                  System.out.println("Child Thread");
6)                                  }
7)                                 };
8)              Thread t = new Thread(r);
9)              t.start();
10)             for(int i=0; i<=10; i++) {
11)                     System.out.println("Main Thread");
12)             }
13)     }
14) }
```

With Method Reference

```
1)  class Test {
2)      public static void m1() {
3)          for(int i=0; i<=10; i++) {
4)              System.out.println("Child Thread");
5)          }
```

```
6)   }
7)   public static void main(String[] args) {
8)              Runnable r = Test:: m1;
9)              Thread t = new Thread(r);
10)             t.start();
11)             for(int i=0; i<=10; i++) {
12)                   System.out.println("Main Thread");
13)             }
14) }
```

In the above example Runnable interface run() method referring to Test class static method m1().
Method reference to Instance method:

Ex:

```
1)   interface Interf {
2)         public void m1(int i);
3)    }
4)   class Test {
5)          public void m2(int i) {
6)                System.out.println("From Method Reference:"+i);
7)          }
8)          public static void main(String[] args) {
9)               Interf  f = I ->sop("From Lambda Expression:"+i);
10)              f.m1(10);
11)              Test t = new Test();
12)              Interf i1 = t::m2;
13)              i1.m1(20);
14)        }
15) }
```

In the above example functional interface method m1() referring to Test class instance method m2().
The main advantage of method reference is we can use already existing code to implement functional interfaces (code reusability).

Constructor References

We can use :: (double colon)operator to refer constructors also

Syntax: classname :: new

Ex:

Interf f = sample :: new;
functional interface f referring sample class constructor

Ex:

```
1)  class Sample {
2)          private String s;
3)          Sample(String s) {
4)                  this.s = s;
5)                  System.out.println("Constructor Executed:"+s);
6)           }
7)  }
8)  interface Interf {
9)          public Sample get(String s);
10) }
11) class Test {
12)         public static void main(String[] args) {
13)                 Interf f = s -> new Sample(s);
14)                 f.get("From Lambda Expression");
15)                 Interf f1 = Sample :: new;
16)                 f1.get("From Constructor Reference");
17)         }
18) }
```

Note: In method and constructor references compulsory the argument types must be matched.

Streams

To process objects of the collection, in 1.8 version Streams concept introduced.

What is the differences between Java.util.streams and Java.io streams?

java.util streams meant for processing objects from the collection. Ie, it represents a stream of objects from the collection but Java.io streams meant for processing binary and character data with respect to file. i.e it represents stream of binary data or character data from the file .hence Java.io streams and Java.util streams both are different.

What is the difference between collection and stream?

- **If we want to represent a group of individual objects as a single entity then We should go for collection.**
- **If we want to process a group of objects from the collection then we should go for streams.**
- **We can create a stream object to the collection by using stream() method of Collection interface. stream() method is a default method added to the Collection in 1.8 version.**

default Stream stream()

Ex: Stream s = c.stream();

- **Stream is an interface present in java.util.stream. Once we got the stream, by using that we can process objects of that collection.**

- **We can process the objects in the following 2 phases**

1.Configuration
2.Processing

1) Configuration:

We can configure either by using filter mechanism or by using map mechanism.

Filtering:

We can configure a filter to filter elements from the collection based on some boolean condition by using filter()method of Stream interface.

```
public Stream filter(Predicate<T> t)
```

here (Predicate<T > t) can be a boolean valued function/lambda expression

Ex:

```
Stream s = c.stream();
Stream s1 = s.filter(i -> i%2==0);
```

Hence to filter elements of collection based on some Boolean condition we should go for filter() method.

Mapping:

If we want to create a separate new object, for every object present in the collection based on our requirement then we should go for map() method of Stream interface.

```
public Stream map (Function f);
```

It can be lambda expression also

Ex:

```
Stream s = c.stream();
Stream s1 = s.map(i-> i+10);
```

Once we performed configuration we can process objects by using several methods.

2) Processing

processing by collect() method
Processing by count()method
Processing by sorted()method
Processing by min() and max() methods
forEach() method
toArray() method
Stream.of()method

Processing by collect() method

This method collects the elements from the stream and adding to the specified to the collection indicated (specified) by argument.

Ex 1: To collect only even numbers from the array list

Approach-1: Without Streams

```
1)  import Java.util.*;
2)  class Test {
3)        public static void main(String[] args) {
4)              ArrayList<Integer> l1 = new ArrayList<Integer>();
5)              for(int i=0; i<=10; i++) {
6)                  l1.add(i);
7)              }
8)              System.out.println(l1);
9)              ArrayList<Integer> l2 = new ArrayList<Integer>();
10)             for(Integer i:l1) {
11)                   if(i%2 == 0)
12)                       l2.add(i);
13)             }
14)             System.out.println(l2);
15)       }
16) }
```

Approach-2: With Streams

```
1)  import Java.util.*;
2)  import Java.util.stream.*;
3)  class Test {
4)        public static void main(String[] args) {
5)              ArrayList<Integer> l1 = new ArrayList<Integer>();
6)              for(inti=0; i<=10; i++) {
7)                    l1.add(i);
8)              }
9)              System.out.println(l1);
10)             List<Integer> l2 = l1.stream().filter(i -> i%2==0).collect(Collectors.toList());
11)             System.out.println(l2);
12)       }
13) }
```

Ex: Program for map() and collect() Method

```
1)  import Java.util.*;
2)  import Java.util.stream.*;
3)  class Test {
4)        public static void main(String[] args) {
5)                ArrayList<String> l = new ArrayList<String>();
6)                l.add("rvk"); l.add("rk"); l.add("rkv"); l.add("rvki"); l.add("rvkir");
7)                System.out.println(l);
8)                List<String> l2 = l.Stream().map(s ->s.toUpperCase()).collect(Collectors.toList());
9)                System.out.println(l2);
10)       }
11) }
```

II.Processing by count()method

This method returns number of elements present in the stream.

```
public long count()
```

Ex:

```
long count = l.stream().filter(s ->s.length()==5).count();
sop("The number of 5 length strings is:"+count);
```

III.Processing by sorted()method

If we sort the elements present inside stream then we should go for sorted() method.
the sorting can either default natural sorting order or customized sorting order specified by comparator.
sorted()- default natural sorting order
sorted(Comparator c)-customized sorting order.

Ex:

```
List<String> l3=l.stream().sorted().collect(Collectors.toList());
sop("according to default natural sorting order:"+l3);

List<String> l4=l.stream().sorted((s1,s2) -> -s1.compareTo(s2)).collect(Collectors.toList());
sop("according to customized sorting order:"+l4);
```

IV.Processing by min() and max() methods

min(Comparator c)
returns minimum value according to specified comparator.

max(Comparator c)
returns maximum value according to specified comparator

Ex:

```
String min=l.stream().min((s1,s2) -> s1.compareTo(s2)).get();
sop("minimum value is:"+min);

String max=l.stream().max((s1,s2) -> s1.compareTo(s2)).get();
sop("maximum value is:"+max);
```

V.forEach() method

This method will not return anything.
This method will take lambda expression as argument and apply that lambda expression for each element present in the stream.

Ex:

```
l.stream().forEach(s->sop(s));
l3.stream().forEach(System.out:: println);
```

Ex:

```
1)  import Java.util.*;
2)  import Java.util.stream.*;
3)  class Test1 {
4)          public static void main(String[] args) {
5)                  ArrayList<Integer> l1 = new ArrayaList<Integer>();
6)                  l1.add(0); l1.add(15); l1.add(10); l1.add(5); l1.add(30); l1.add(25); l1.add(20);
7)                  System.out.println(l1);
8)                  ArrayList<Integer> l2=l1.stream().map(i-> i+10).collect(Collectors.toList());
9)                  System.out.println(l2);
10)                 long count = l1.stream().filter(i->i%2==0).count();
11)                 System.out.println(count);
12)                 List<Integer> l3=l1.stream().sorted().collect(Collectors.toList());
13)                 System.out.println(l3);
14)                 Comparator<Integer> comp=(i1,i2)->i1.compareTo(i2);
15)                 List<Integer> l4=l1.stream().sorted(comp).collect(Collectors.toList());
16)                 System.out.println(l4);
17)                 Integer min=l1.stream().min(comp).get();
18)                 System.out.println(min);
19)                 Integer max=l1.stream().max(comp).get();
20)                 System.out.println(max);
```

```
21)                 l3.stream().forEach(i->sop(i));
22)                 l3.stream().forEach(System.out:: println);
23)
24)         }
25) }
```

VI.toArray() method

We can use toArray() method to copy elements present in the stream into specified array

```
Integer[] ir = l1.stream().toArray(Integer[] :: new);
for(Integer i: ir) {
        sop(i);
}
```

VII.Stream.of()method

We can also apply a stream for group of values and for arrays.

Ex:

```
Stream s=Stream.of(99,999,9999,99999);
s.forEach(System.out:: println);

Double[] d={10.0,10.1,10.2,10.3};
Stream s1=Stream.of(d);
s1.forEach(System.out :: println);
```

Date and Time API: (Joda-Time API)

Until Java 1.7version the classes present in Java.util package to handle Date and Time (like Date, Calendar, TimeZoneetc) are not up to the mark with respect to convenience and performance.

To overcome this problem in the 1.8version oracle people introduced Joda-Time API. This API developed by joda.org and available in Java in the form of Java.time package.

 # program for to display System Date and time.

```
1)  import Java.time.*;
2)  public class DateTime {
3)     public static void main(String[] args) {
4)        LocalDate date = LocalDate.now();
5)        System.out.println(date);
6)        LocalTime time=LocalTime.now();
7)        System.out.println(time);
8)     }
9)  }
```

O/p:
2015-11-23
12:39:26:587

Once we get LocalDate object we can call the following methods on that object to retrieve Day,month and year values separately.

Ex:

```
1)  import Java.time.*;
2)  class Test {
3)        public static void main(String[] args) {
4)                LocalDate date = LocalDate.now();
5)                System.out.println(date);
6)                int dd = date.getDayOfMonth();
7)                int mm = date.getMonthValue();
8)                int yy = date.getYear();
9)                System.out.println(dd+"..."+mm+"..."+yy);
10)               System.out.printf("\n%d-%d-%d",dd,mm,yy);
11)   }
12) }
```

Once we get LocalTime object we can call the following methods on that object.

Ex:

```
1)  importJava.time.*;
2)  class Test {
3)     public static void main(String[] args)  {
4)        LocalTime time = LocalTime.now();
5)        int h = time.getHour();
6)        int m = time.getMinute();
7)        int s = time.getSecond();
8)        int n = time.getNano();
9)        System.out.printf("\n%d:%d:%d:%d",h,m,s,n);
10)    }
11) }
```

If we want to represent both Date and Time then we should go for LocalDateTime object.

```
LocalDateTimedt = LocalDateTime.now();
System.out.println(dt);
```

O/p: 2015-11-23T12:57:24.531

We can represent a particular Date and Time by using LocalDateTime object as follows.

Ex:

```
LocalDateTime dt1 = LocalDateTime.of(1995,Month.APRIL,28,12,45);
sop(dt1);
```

Ex:

```
LocalDateTime dt1=LocalDateTime.of(1995,04,28,12,45);
Sop(dt1);
Sop("After six months:"+dt.plusMonths(6));
Sop("Before six months:"+dt.minusMonths(6));
```

To Represent Zone:

ZoneId object can be used to represent Zone.

Ex:

```
1)  import Java.time.*;
2)  class ProgramOne {
3)        public static void main(String[] args) {
4)              ZoneId zone = ZoneId.systemDefault();
5)              System.out.println(zone);
6)        }
7)  }
```

We can create ZoneId for a particular zone as follows

Ex:

```
ZoneId la = ZoneId.of("America/Los_Angeles");
ZonedDateTimezt = ZonedDateTime.now(la);
System.out.println(zt);
```

Period Object:

Period object can be used to represent quantity of time

Ex:

```
LocalDate today = LocalDate.now();
LocalDate birthday = LocalDate.of(1989,06,15);
Period p = Period.between(birthday,today);
System.out.printf("age is %d year %d months %d days",p.getYears(),p.getMonths(),p.getDays());
```

write a program to check the given year is leap year or not

```
1)  import Java.time.*;
2)  public class Leapyear {
3)     int n = Integer.parseInt(args[0]);
4)     Year y = Year.of(n);
5)     if(y.isLeap())
6)           System.out.printf("%d is Leap year",n);
7)     else
8)           System.out.printf("%d is not Leap year",n);
9)  }
```