Java8 Features

1. Lambda Expression
2. Functional Interfaces
3. Default methods in interfaces
4. Static methods in interfaces
5. Pre-define Functional Interfaces
    I. Predicate
    II. Function
    III. Consumer
    IV. Supplier, etc.
6. Method reference and Constructor reference by double colon(::) operator
7. Stream API
8. Date and Time API (also known as Joda API because it is develop by joda.org)
9. Optional class
10. Nashron JavaScript Engine, etc.

Objective of java8:

- Simplify the programing.
- To enable functional programming in java by using lambda expression.
- To enable programming or parallel processing in java.

Lambda Expression

- LISP is the first programming language that uses lambda expression.
- Language that uses lambda expression are: -
    - C
    - C++
    - Objective C
    - C#
    - Dot net
    - SACLA
    - RUBY
    - Python
    - JAVA(uses from jdk 1.8)

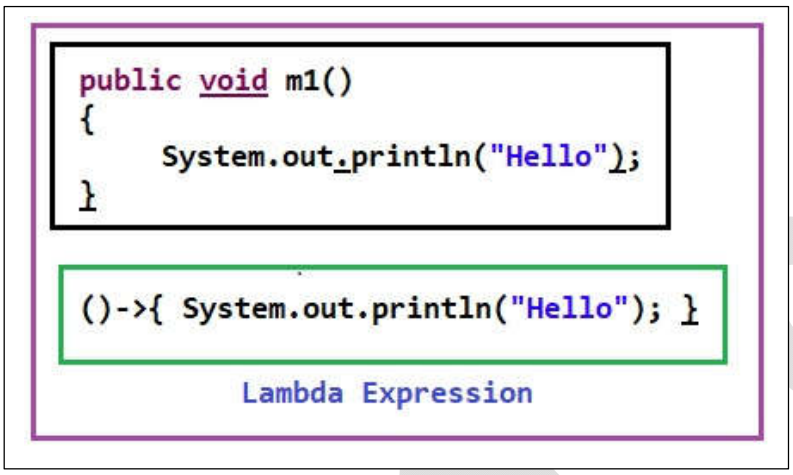
Benefits of Lambda Expression

- To enable functional programming in java.
- To write more readable, maintainable and clean concise code.
- To use API's very easily and effectively.
- To enable parallel processing.

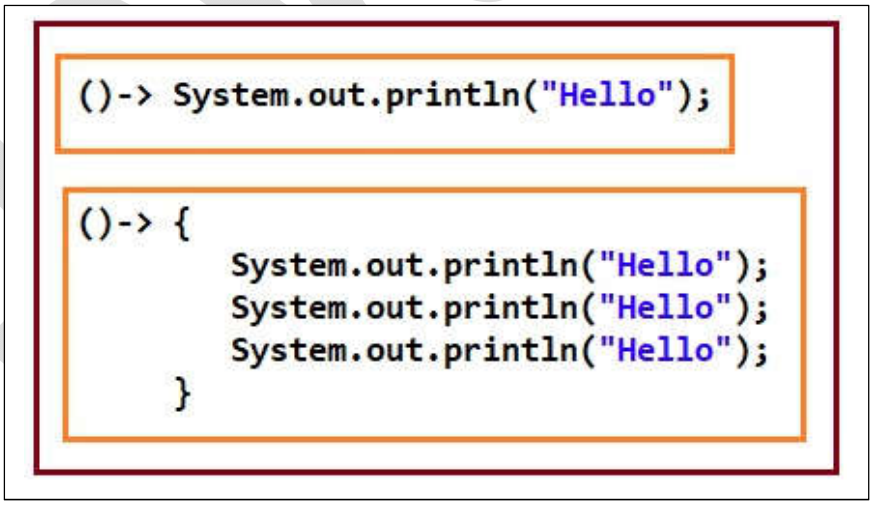
What is Lambda Expression?

- It is an Anonymous(nameless) function.
- A function without name , without return type and without modifiers is known as Anonymous function.
- It is represented by arrow(->).

How to write Lambda Expression?



- If in lambda expression only one line then we can remove the braces, because curly braces are optional, if there is more than one line of code then we can use curly braces.



- If there is parameter, then compiler get the type automatically we need not to write the data type.

```
public void m1( int a, int b)
{
      System.out.println(a+b);
}
```

```
//(int a,int b)->{ System.out.println(a+b); }
(a,b)->{ System.out.println(a+b); }
```

- Return statement with lambda expression

```
public int square( int n )
{
return n*n;
}


//lambda expression
( int n)->{ return n*n };
```

- If function is returning something then no need to write return statement, compiler can guess this is returning something. But with curly braces we have to write return keyword and without curly braces no need write return keyword.

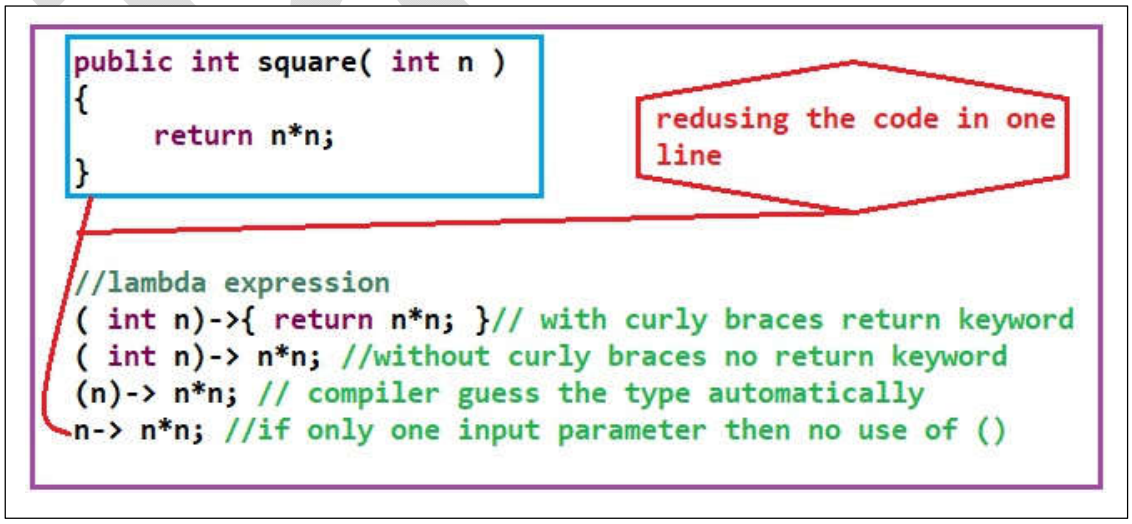
```
//lambda expression
( int n)->n*n;
```

- We know compiler guess the type automatically we can remove the data type also.

```
//lambda expression
(n)->n*n;
```

- If only one input parameter is used then no need to use of parenthesis (), in zero argument and more than one argument parenthesis are mandatory.

```
//lambda expression
n->n*n;
```



Example:

```
//normal method
   public int m1( String s )
{
      return s.length();
}


//lambda expression
  s->s.length();
```

Functional interfaces

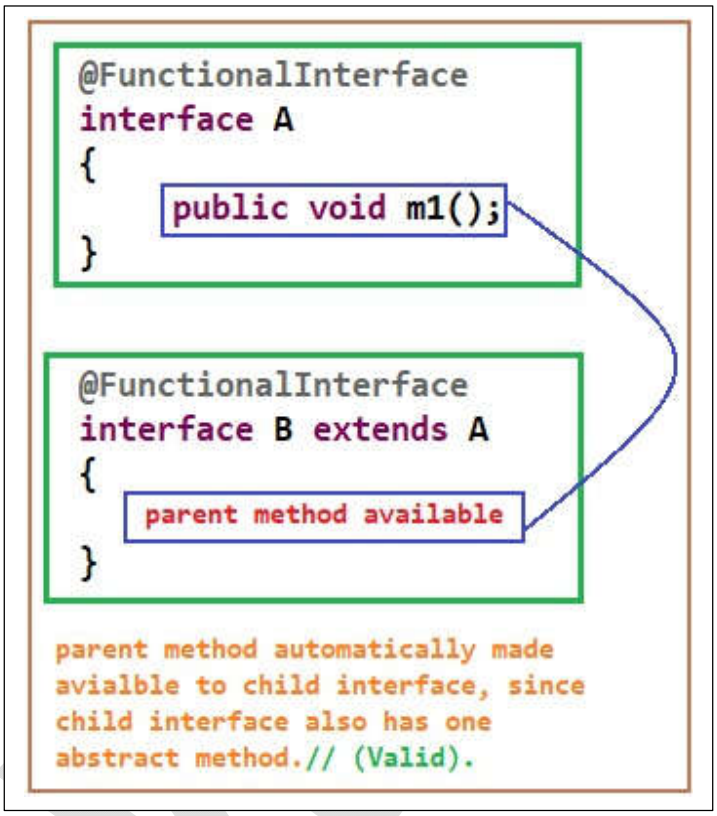
- With the help of functional interface we can call the lambda expression.
- Functional interfaces are those interfaces which has only one abstract method is known as functional interface.
- So if we want to invoke any lambda expression we use functional interface.
- Some functional interfaces with their methods are:-
  - Runnable➔run()
  - Callable➔call()
  - Comparable➔compareTo()
  - ActionListener➔actionPerformed()
- Functional interface can have exactly only one abstract method but it can have any number of **default and static method.**

```
interface Intef
{
      public void m1();
      default void m2()
      {
            //body
      }
      public static void m3()
      {
            //body
      }
}
```

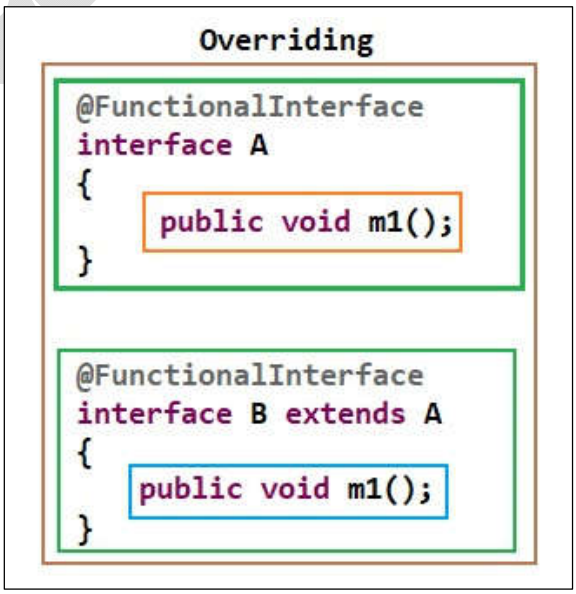
- @FunctionalInterface annotation is used to identify the functional interface. Since writing annotation is optional but if before any interface we write @FunctionalInterface annotation and we not write any abstract method in that interface then immediately compiler gives an error as **"no abstract method found in interface interface_name".**

Functional Interface with respect to inheritance

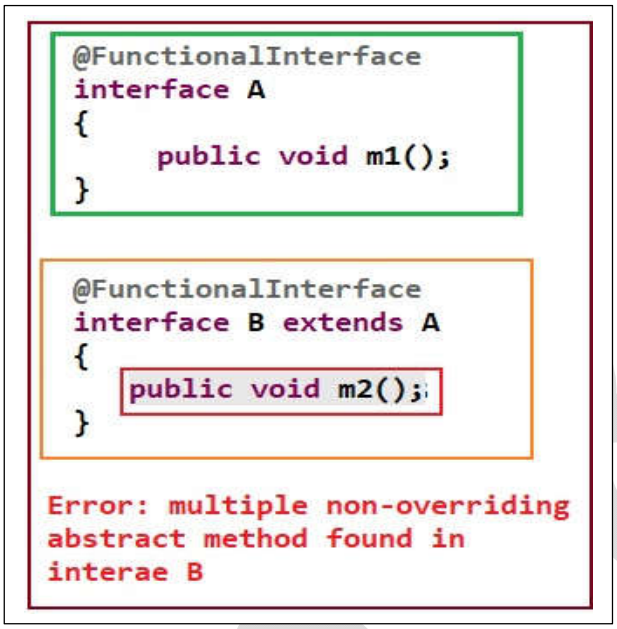
- Suppose a parent interface and parent interface has one abstract method and one child interface extends that interface and that child has no abstract method then automatically parent method made available to child interface.



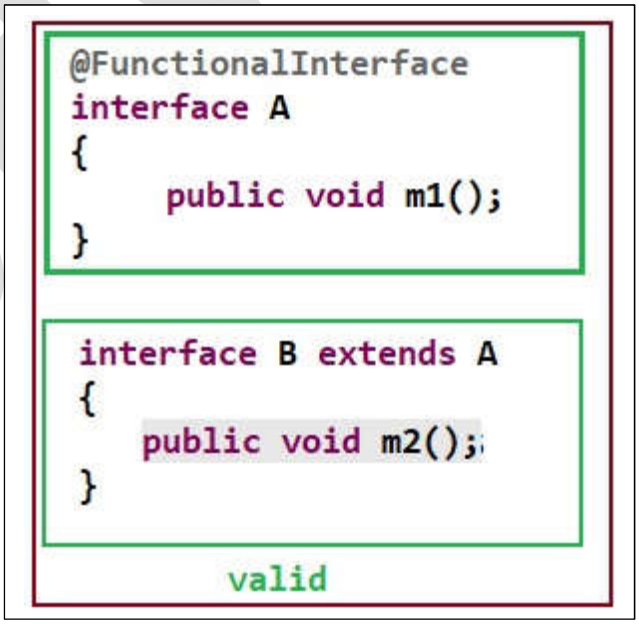
- Overriding of abstract method in case of functional interface is possible. Because at that time only one abstract method for child is available.



- When parent interface has an abstract method and child has another abstract method then it will give an error **"multiple non-overriding abstract method found in interface child_interface_name".**



- If parent and child both are functional interface then child should not have any other abstract method instead it has overridden abstract method.
- If parent is functional interface and child is a normal interface then child interface has any number of abstract method.



Lambda Expression with Functional Interface

- If an interface has an abstract method any child class implements the interface then child class has mandatory to override the abstract method of interface and provide implementation. Doing this make the code to lengthy, with the help of lambda expression we can minimize the code and make it readable.

Example: without lambda Expression

```
@FunctionalInterface
interface A
{
	public void m1();
}

class Test implements A
{
	public void m1()
	{
		System.out.println("Hello...!");
	}
}

class Demo
{
	public static void main(String[] args)
	{
		A obj= new Test();
		obj.m1();
	}
}
```

Example: with lambda Expression

```
@FunctionalInterface
interface A
{
	public void m1();
}

class Demo
{
	public static void main(String[] args)
	{
		A obj= ()-> System.out.println("Hello...!");

		obj.m1();
	}
}
```

```java
@FunctionalInterface
interface A
{
	public void m1();
}

class Demo
{
	public static void main(String[] args)
	{
		A obj= ()-> System.out.println("Hello...!");

		obj.m1();
	}
}
```

Lines of code is reduce by using lambda expression.

- Java.util.function package has many functional Interface which supports our program.
- No .class file is generated for any lambda expression. Because for anonymous class no .class file is generated. Since lambda expression written inside anonymous class.

Multithreading with Lambda Expression

- In two ways we can define a thread
  - By implementing Runnable interface
  - By Extending the thread class
- We can use lambda expression with multithreading concept, to reduce line of code.

By using Runnable interface without lambda Expression

```java
package multithreading;
class MyRunnable implements Runnable
{
	public void run()
	{
		for (int i = 0; i < 10; i++)
		{
			System.out.println(i+" Child Thread");
		}
	}
}

public class Program1
{
	public static void main(String[] args)
	{
```

```
                MyRunnable m=new MyRunnable();
                Thread t= new Thread(m);
                t.start();
                for (int i = 0; i < 10; i++)
                {
                        System.out.println(i+" Main Thread");
                }
        }
}
```

Runnable interface with Lambda Expression

```
package multithreading;
public class Program2
{
        public static void main(String[] args)
        {
                Runnable r=()->{
                        for (int i = 0; i < 10; i++) {
                                System.out.println(i+" Cild Thread");
                        }
                };
                Thread t= new Thread(r);
                t.start();
                for (int i = 0; i < 10; i++)
                {
                        System.out.println(i+" Main Thread");
                }
        }
}
```

Lambda Expression With Collection Interface

Normal collection program with sorting

```
package collection;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;

class MyComparator implements Comparator<Integer>
{
        @Override
        public int compare(Integer o1, Integer o2)
        {
        //return (o1<o2)?-1:(o1>o2)?+1:0;
                if(o1<o2)
                {
                        return -1;
                }
```

```
                else if(o1>o2)
                {
                        return +1;
                }
                else
                {
                        return 0;
                }
        }
}

public class Program1
{
        public static void main(String[] args)
        {
                ArrayList<Integer> l= new ArrayList<Integer>();
                l.add(20);
                l.add(10);
                l.add(25);
                l.add(5);
                l.add(30);
                l.add(0);
                l.add(15);
                System.out.println(l);
                Collections.sort(l,new MyComparator());
                System.out.println(l);
        }
}
```

Coding with Lambda Expression

```
package collection;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
public class Program2
{
        public static void main(String[] args)
        {
                ArrayList<Integer> l= new ArrayList<Integer>();
                l.add(20);
                l.add(10);
                l.add(25);
                l.add(5);
                l.add(30);
                l.add(0);
                l.add(15);
                System.out.println(l);
                //lambda expression
                Comparator<Integer> c=(o1,o2)->(o1<o2)?-1:(o1>o2)?+1:0;
                Collections.sort(l,c);
                System.out.println("sorted");
```

```
                System.out.println(l);
            }
        }
```

Sorting Employee Details based on Employee number

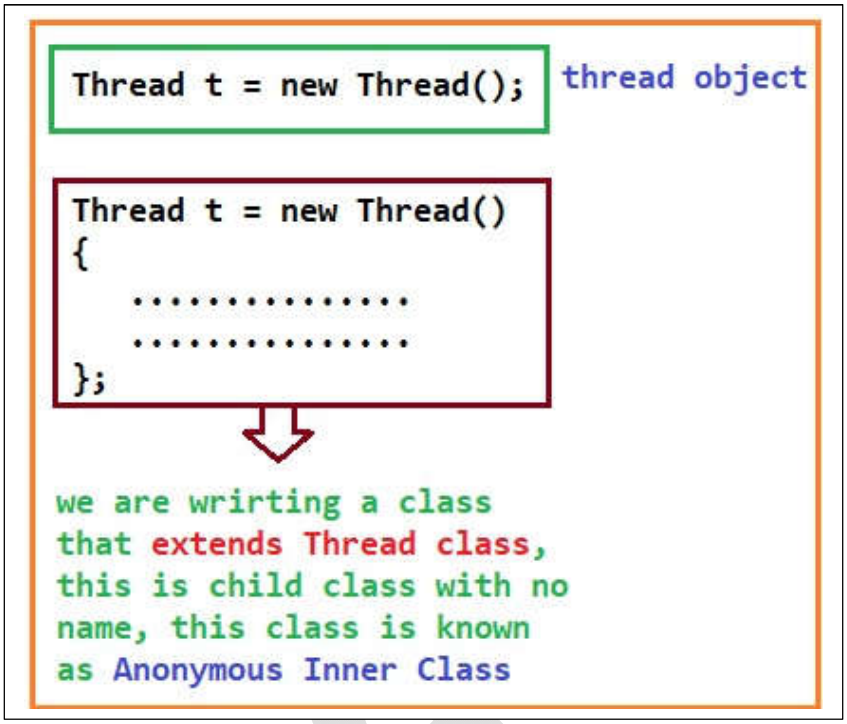
```
package collection;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;

class EmployeeDetails
{
        public String name;
        public int empno;
         public EmployeeDetails(String name, int empno)
         {
                this.empno =empno;
                this.name=name;
        }
         public String toString()
         {
                return name+":"+empno;

         }
}

public class Employee
{
        public static void main(String[] args) {
                ArrayList<EmployeeDetails> a= new
ArrayList<EmployeeDetails>();
                a.add(new EmployeeDetails("Ram", 123));
                a.add(new EmployeeDetails("Avinash", 453));
                a.add(new EmployeeDetails("Neha", 0651));
                a.add(new EmployeeDetails("Prabhjot", 726));
                a.add(new EmployeeDetails("Farooq", 786));
                a.add(new EmployeeDetails("Rinky", 2));
                System.out.println("Employee details");
                System.out.println(a);

                Comparator<EmployeeDetails> c=(e1,e2)->(e1.empno<e2.empno)?-
1:(e1.empno>e2.empno)?+1:0;
                Collections.sort(a,c);
                System.out.println(a);
        }

}
```

Sorting based on Alphabetical order

```
package collection;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;

class EmployeeDetails
{
	public String name;
	public int empno;
	 public EmployeeDetails(String name, int empno)
	 {
		this.empno =empno;
		this.name=name;
	}
	 public String toString()
	 {
		return name+":"+empno;

	 }
}

public class Employee
{
	public static void main(String[] args) {
		ArrayList<EmployeeDetails> a= new
ArrayList<EmployeeDetails>();
		a.add(new EmployeeDetails("Ram", 123));
		a.add(new EmployeeDetails("Avinash", 453));
		a.add(new EmployeeDetails("Neha", 0651));
		a.add(new EmployeeDetails("Prabhjot", 726));
		a.add(new EmployeeDetails("Farooq", 786));
		a.add(new EmployeeDetails("Rinky", 2));
		System.out.println("Employee details");
		System.out.println(a);
		//Comparator<EmployeeDetails> c=(e1,e2)-
>(e1.empno<e2.empno)?-1:(e1.empno>e2.empno)?+1:0;

		Collections.sort(a,(e1,e2)->e1.name.compareTo(e2.name));
		System.out.println(a);
	}

}
```

Anonymous Inner classes V/s Lambda Expression

- The class without having name is called anonymous inner class.



- **This anonymous class is nothing but an implementation class with no name.**

```
package AnonymousInnerClass;
public class Program1
{
	public static void main(String[] args)
	{
		//Anonymous inner class
		Runnable r= ()->
		{
			for (int i = 0; i <10; i++) {
			System.out.println("child thread");

			}
		};
		Thread t = new Thread(r);
		t.start();
		for (int i = 0; i <10; i++) {
			System.out.println("main thread");

		}
	}
}
```

- Anonymous inner class and lambda expression are not same, because lambda expression works only for functional interface means only for that interfaces which has only one public abstract method but for more than lambda expression not worked.
- We can write any number of anonymous inner class for depends on how many abstract methods are present in an interface.
- Anonymous inner class is more powerful than lambda expression.

```
package AnonymousInnerClass;
interface A
{
	void m1();
	void m2();
	}
public class Main
{
	public static void main(String[] args)
	{
		A a=new A() {
			public void m1(){
				System.out.println("this is m1");
			}
			public void m2(){
				System.out.println("this is m2");
			}
		};
		a.m1();
		a.m2();
	}
}
```

- If anonymous inner class implements an interface that contains single abstract method then only we can replace that anonymous inner class with lambda expression.
- Anonymous inner class can extend a normal class.
- Anonymous inner class can extend an abstract class.
- Anonymous inner class can implement an interface which contains any number of abstract method.
- Lambda Expression can implement an interface which contains a single abstract method (Functional Interface).
- Anonymous inner class never be equal to Lambda Expression.

```
package org.ram.java8Features.lambdaExpression;
import java.util.Scanner;
interface DemoInterface{
	public void method1();
	public void methodWithStingParameter(String message);
	public int methodWithIntegerReturnType();
public boolean methodWithParameterAndReturnType(int number);
}
```

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

		DemoInterface demoInterface= new DemoInterface() {

	@Override
	public void methodWithStingParameter(String message) {
	System.out.println("This is method with string parameter
message: "+message);
			}

	@Override
public boolean methodWithParameterAndReturnType(int number) {
				System.out.println("Method with boolean
return type with respect to integer parameter. "+number);
				return false;
			}

@Override
public int methodWithIntegerReturnType() {
System.out.println("Method with integer return type based on
operation");
	return 0;
			}

@Override
public void method1() {
System.out.println("This is sample method for Annonymous ineer
class method.");
			}
		};

demoInterface.method1();
demoInterface.methodWithIntegerReturnType();
demoInterface.methodWithParameterAndReturnType(1);
demoInterface.methodWithStingParameter("Hello Annonymous.");
	}
}
```

OutPut

```
This is sample method for Annonymous ineer class method.
Method with integer return type based on operation
Method with boolean return type with respect to integer parameter. 1
This is method with string parameter message: Hello Annonymous.
```

Default Methods in Interface

- Until jdk1.7 every method present inside an interface is always public and abstract.
- From jdk1.8 default and static methods are allowed inside an interface.
- From jdk1.9 private methods are allowed inside an interface.
- Every variable present inside an interface are public, static and final variable. Since no enhancement is done with respect to variables in versions of java.
- Default method are also known as Virtual Extension Method or also known as Defender Method.
- This is called defender method because it is not going to affect the implementation classes, when there is a default method is added in to the interface. The rights to child class either they want to implement that method or not, it's their choice.
- Child class can override the default method, but while overriding mandatory to use public access specifier.
- So default keyword we only write inside the interface not inside the class.
- Object class methods can't implement as default method inside an interface, it gives compile time error. Because object class method already made available to every child class in java, so implementation class of interface has by default also has Object class method. That's why java don't support defining object class method inside an interface.

Objective of Default method

- Without affecting implementation classes if we want to add a new method to the interface.

Why the word default?

- Because this method having default implementation.

How to solve diamond access problem with respect to "default" method in interface.

- When two interfaces has same default method with their body, this lead to diamond problem in java.
- When two interfaces has same method name with their bodies, then we have to explicitly define that method (override) inside the child class, by doing this we can solve the diamond problem.

Coding part of diamond access problem solution in interface

```
package DefaultMethod.DiamondProblem;

public interface IFirst
{
	default void show()
	{
		System.out.println("this is interface show1");
	}
}
```

```
package DefaultMethod.DiamondProblem;

public interface ISecond
{
	default void show()
	{
		System.out.println("this is interface show2");
	}
}
```

```
package DefaultMethod.DiamondProblem;
public class MainClass implements IFirst,ISecond
{
	//defining the method
	public void show()
	{
		System.out.println("this is child");
	}

	public static void main(String[] args)
	{
		MainClass obj= new MainClass();
		obj.show();
	}
}
```

Output→ this is child

- If we want to access the code of a default method from any specific interface then you have to write interface name followed by dot operator after that super keyword followed by dot operator and then after method name inside the override method.

Syntax: interface_name.super.default_method();

```
package DefaultMethod.DiamondProblem;
public class MainClass implements IFirst,ISecond
{
	//defining the method
	public void show()
	{
		IFirst.super.show()
	}

	public static void main(String[] args)
	{
		MainClass obj= new MainClass();
		obj.show();
	}
}
```

Output→ this is interface show1

Class and interface with same method name

- When a class and interface have same name and their implementation are different then in that case object of child class give the priority of method which is present inside the class not the method which is present inside an interface. Because class has more power than interface (This concept is known as Third Rule In Java).

➢ **Coding part of Class and interface with same method name.**

```
package DefaultMethod.ClassAndInterface;

public class ClassA
{
	public void show()
	{
		System.out.println("this is class show");
	}
}
```

```
package DefaultMethod.ClassAndInterface;

public interface IFirst
{
default void show()
{
System.out.println("this is interface show");
}

}
```

```
package DefaultMethod.ClassAndInterface;
public class MainClass extends ClassA implements IFirst
{
public static void main(String[] args)
{
MainClass obj = new MainClass();
obj.show();
}
}
```

```
Output→this is class show
```

Static methods in Interface

- As we know for class we have to create an object , and for interface we don't create an object.
- Static methods are not related to object. So without creating object we can call the static methods.

- So if we have only static method so going for class better to go for interface, because class is very costly.
- So from jdk1.8 onwards interfaces also has static methods because static methods are not related to objects.
- Static method are by default not available to implementation class.

How to call Static method in implementation class

- As we know Static method are by default not available to implementation class.
- So there is only one way to call the static method of interface is that, i.e with interface name followed by dot operator and then name of static method.

Syntax: Interface_name.staticMethod()

```
interface A
{
	public static void m1()
	{
		System.out.println("interface static method");
	}
}
 class Program1 implements A
{
	public static void main(String[] args)
	{
		A.m1();// correct
		m1()//not possible
		Program1.m1();//not possible
		Program1 p = new Program1();
		p.m1()//not possible
	}
}
```

- we can call static method of interface, we can also call inside a non implementation class also.

```
interface A
{
	public static void m1()
	{
		System.out.println("interface static method");
	}
}
 class Program1
{
	public static void main(String[] args)
	{
		A.m1();// correct
		m1()//not possible
		Program1.m1();//not possible
		Program1 p = new Program1();
		p.m1()//not possible
	}}
```

Interface and Main Method(`public static void main(String[] args)`)

- from jdk1.8 onward it is possible to write main method inside an interface.

```
public interface Interface2
{
   public static void main(String[] args)
   {
         System.out.println("Hello.. This is Interface main method");
   }
}
```

Output→ Hello.. This is Interface main method

Why Static methods in interface from jdk1.8

- To general define utility method we used to static methods inside interface. Because every thing is static no need of object, since class is costly.

Predefined Functional Interface:

- These Predefined Functional Interfaces helps in lambda expression.
- These Predefined Functional Interfaces present **in java.util.function package.**
- Some Predefined Functional Interfaces are:-
    1. Predicate
    2. Function
    3. Consumer
    4. Supplier
- Two argument Predefined Functional Interfaces
    1. BiPredicate
    2. BiFunction
    3. BiConsumer
- Primitive Functional Interfaces
    1. IntPredicate
    2. IntFunction
    3. IntConsumer

    ……

Predicate

- Predicate is predefined Functional Interfaces contains only one abstract method.
- The return type is always Boolean, because it always used for conditional check.
- Predicate Functional Interfaces contains only test() method.
- Present inside **java.util.function** package.
- For predicate Input type of predicate varies from different-different scenario, but return type is always Boolean.
- Whenever conditional check is required then we go for Predicate concept with lambda expression.

- Syntax: public abstract boolean test (Test t);

Interface Predicate<T>{

Public Boolean test(T t);

}

```
public boolean test(Integer I)
	{
		if(I%2==0)
		{return true;}
		else {return false}
	}

	//lambda expression for above function
	(Integer I)->I%2==0


//Predicate Program

public static void main(String[] args)
	{
		Predicate<Integer> p1= i->i%2==0;
		System.out.println(p1.test(10)); //true
		System.out.println(p1.test(15));//false
		System.out.println(p1.test(16));//true
	}
```

Program2

```
public static void main(String[] args)
    {
	String[] s= {"Sangeeta","Avinash","Farooq","Anuj","Rinky"};
	Predicate<String > p= s1->s1.length()>5;
	for (String s11 : s)
	{
		if(p.test(s11))
		{
			System.out.print(s11);
		}
	}
    }
```

Output

Sangeeta Avinash Farooq

Program-3

```
package Methods.Predicate;
import java.util.ArrayList;
import java.util.function.Predicate;
public class Employee
{
	String name;
	double salary;

	public Employee(String name, double salary)
	{
		super();
		this.name = name;
		this.salary = salary;
	}

	public static void main(String[] args)
	{
		ArrayList<Employee> l= new ArrayList<Employee>();
		l.add (new Employee ("Rinky", 12000));
		l.add (new Employee ("Farooq", 10000));
		l.add (new Employee ("Abhijeet", 9000));
		l.add (new Employee ("Anuj", 8000));
		l.add (new Employee ("Neha", 11500));

		Predicate<Employee> p=e->e.salary>9000;
		for (Employee e1 : l)
		{
			if(p.test(e1))
	System.out.println(e1.name+" salary is "+e1.salary);
		}
	}
}
```

Output
Rinky salary is 12000.0
Farooq salary is 10000.0
Neha salary is 11500.0

Advantage of Predicate

- Inside predicate we can write n number of condition.

Predicate Joining

- We can combine multiple predicate together to check complex condition.
- P1.and(P2)→both condition are satisfied
- P1.or(P2)→either one condition are satisfied
- P1.negate()→opposite of P1.

Program

```
/*
 * 1.Predicate-1 checks whether number is even or not
 * 2.Predicate-2 check whether number is greater than 20*/

package Methods.Predicate;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.function.Predicate;

public class TwoPredicateInOne
{
	public static void main(String[] args)
	{
		int[] x= {1,8,3,9,22,44,5,45,36,78,12};

Predicate<Integer> p1=i->i%2==0;//checks whether number is even or not
Predicate<Integer> p2=i->i>20;//heck whether number is greater than 20

		for (int j : x)
		{
			if(p1.and(p2).test(j))
				System.out.print(j+",");
		}

	}
}
```

Function

- It is a predefined functional interface.
- This method take two parameter, one is input and one return type. Return type is anything does not depend on input type.
  - Function<InputType, ReturnType>
- Function contain apply() method.
- It is used to execute the business logic.
- Present inside java.util.function package.

Q.Input type is integer and return type is integer

```
//square of number
	public static void main(String[] args)
		{
			Function<Integer, Integer> f=i->i*i;
			System.out.println(f.apply(5));
	}
```

Q.Input type is any String and return type is length of String

```
public static void main(String[] args)
	{
		Function<String, Integer> f=s->s.length();
```

```
            System.out.println("Length of String is: "+f.apply("Ram"));
      }
```

Output:→ Length of String is: 3

```
Q.//convert given string into upper case
public static void main(String[] args)
      {
            Function<String, String> f=s->s.toUpperCase();
            System.out.println(f.apply("hello world"));//HELLO WORLD
      }

//find grade of student a/c to marks obtained
package student;
import java.util.ArrayList;
import java.util.function.Function;
public class GradeOfStudent
{
      String name;
      int marks;
      public GradeOfStudent(String name, int marks)
      {
            super();
            this.name = name;
            this.marks = marks;
      }
      public static void main(String[] args)
      {
            Function<GradeOfStudent, String>f=s->{
                  int marks=s.marks;
                  String grade="";
                  if(marks>=80) grade="A[Distinction]";
                  else if(marks>=60) grade="B[First Class]";
                  else if(marks>=50) grade="C[Second Class]";
                  else if(marks>=35) grade="D[Third Class]";
                  else grade="E[Fail]";
                  return grade;
            };
            ArrayList<GradeOfStudent> a= new ArrayList<GradeOfStudent>();
            a.add(new GradeOfStudent("Rinky", 66));
            a.add(new GradeOfStudent("Farooq", 96));
            a.add(new GradeOfStudent("Prabht", 72));
            a.add(new GradeOfStudent("Sangea", 45));

System.out.println("Name\t\tMarks\t\tGrade");
System.out.println("-----------------------------------------");
            for (GradeOfStudent s : a)
            {
                  System.out.print(s.name);
                  System.out.print("\t\t"+s.marks);
                  System.out.print("\t\t"+f.apply(s));
                  System.out.println();
```

```
}}}
```

```
OutPut
Name        Marks       Grade
*-------------------------------------------
Rinky       66          B[First Class]
Farooq      96          A[Distinction]
Prabht      72          B[First Class]
Sangea      45          D[Third Class]
```

```
//show student whose marks is greater than 60
package student;
import java.util.ArrayList;
import java.util.function.*;
public class MarksGreaterThan60
{
	String name; int marks;
	public MarksGreaterThan60(String name, int marks)
	{
		super();
		this.name = name; this.marks = marks;
	}
	public static void main(String[] args)
	{
		Function<MarksGreaterThan60, String>f=s->{
			int marks=s.marks;
			String grade="";
			if(marks>=80) grade="A[Distinction]";
			else if(marks>=60) grade="B[First Class]";
			else if(marks>=50) grade="C[Second Class]";
			else if(marks>=35) grade="D[Third Class]";
			else grade="E[Fail]";
			return grade;
		};
		ArrayList<MarksGreaterThan60> a= new
ArrayList<MarksGreaterThan60>();
		a.add(new MarksGreaterThan60("Rinky", 66));
		a.add(new MarksGreaterThan60("Farooq", 96));
		a.add(new MarksGreaterThan60("Prabht", 72));
		a.add(new MarksGreaterThan60("Sangea", 45));

		Predicate<MarksGreaterThan60> p=s->s.marks>60;
		System.out.println("Name\t\tMarks\t\tGrade");
System.out.println("-------------------------------------------");
		for (MarksGreaterThan60 s : a)
		{
			if(p.test(s))
			{
				System.out.print(s.name);
				System.out.print("\t\t"+s.marks);
				System.out.print("\t\t"+f.apply(s));
```

```
                    System.out.println();
                }}}}
```

Output

```
Name        Marks       Grade
-----------------------------------------------
Rinky       66          B[First Class]
Farooq      96          A[Distinction]
Prabht      72          B[First Class]
```

Function Chaining

- F1.andThen(F2).apply(i)
  - First F1 followed by F2
- F1.composed(F2).apply(i)
  - First F2 and then F1

```
Function<Integer, Integer> f1=i->i*2;//2*2=4
Function<Integer, Integer> f2=i->i*i*i;//4*4*4
System.out.println(f1.andThen(f2).apply(2));//64
System.out.println(f1.compose(f2).apply(2));//16
```

Consumer

- It takes input, but never return anything, it just print the input type.
- Just like a customer it only consume.
- It has accept() method which takes a parameter as input type.
  - public void accept(T t)
- **Whenever printing of object is required we use consumer functional interface**

```
Consumer<String> c=s->System.out.println(s);
c.accept("Ram");
c.accept("Avinash");
c.accept("Abhijeet");
```

Output

```
Ram
Avinash
Abhijeet
```

```
import java.util.function.Consumer;
import java.util.function.Function;

public class Student
{
      String name; int marks;

      public Student(String name, int marks) {
            super();
```

```
		this.name = name;
		this.marks = marks;
	}
	public static void main(String[] args)
	{
		Function<Student, String> f=s->
		{
			int m=s.marks;
			String grade="";
			if(m>=80) grade="A[Distinction]";
			else if(m>=60) grade="B[First Class]";
			else if(m>=50) grade="C[Second Class]";
			else if(m>=35) grade="D[Third Class]";
			else grade="E[Fail]";
			return grade;
		};

		Consumer<Student> c=s->{
			System.out.println("Student Name: "+s.name);
			System.out.println("Student marks: "+s.marks);
			System.out.println("Student grade: "+f.apply(s));
	System.out.println("=====================================");
		};
		Student[]s= {
				new Student("Anuj", 56),
				new Student("Farooq", 78),
				new Student("Prabjot",60),
				new Student("Sangeeta", 98),
				new Student("Neha", 45),
				new Student("Avinash", 83),
				new Student("Rinky",71),
				new Student("Amrender",66),
		};
		for (Student student : s)
		{
			c.accept(student);
		}
	}
}
```

OutPut

```
Student Name: Anuj
Student marks: 56
Student grade: C[Second Class]
=====================================
Student Name: Farooq
Student marks: 78
Student grade: B[First Class]
=====================================
Student Name: Prabjot
Student marks: 60
Student grade: B[First Class]
=====================================
Student Name: Neha
Student marks: 45
Student grade: D[Third Class]
=====================================
Student Name: Avinash
Student marks: 83
Student grade: A[Distinction]
=====================================
Student Name: Rinky
Student marks: 71
Student grade: B[First Class]
=====================================
Student Name: Amrender
Student marks: 66
Student grade: B[First Class]
=====================================
```

```
Student Name: Sangeeta
Student marks: 98
Student grade: A[Distinction]
=====================================
```

Supplier

- Just supply my required object and it won't take any input.
- It has only one method that is get().

```
interface Supplier<R>{
      public R get();
}
```

```
package org.ram.java8Features.supplier;
import java.util.Date;
import java.util.function.Supplier;

public class SupplierDate {
      public static void main(String[] args) {
            Supplier<Date> s=()->new Date();
            System.out.println(s.get());
      }
}
```

Output:

```
Tue Jul 20 07:28:27 IST 2021
```

```
package org.ram.java8Features.supplier;
import java.util.function.Supplier;

public class SupplierOTP {
      public static void main(String[] args) {
            Supplier<String> s1=()->{String otp="";
                  for(int i=0;i<7;i++) {
                        otp+=(int)(Math.random()*10);
                  }
                  return otp;
            };
            System.out.println("OTP is: "+s1.get());
      }
}
```

Note: Predicate(test()), Function(apply()) and Consumer(accept()) always take one input. Supplier(get()) not take any input.

Two Input argumnets

- Normal Pre-Define Functional Interfaces take only one input accept Supplier Interface.
- For two input we use BiPredicate,Bifunction,BiConsumer.

BiPredicate

- Normal predicate can take only one input argument and perform some conditional check.
- Some time our program required we have to take 2 input arguments and perfirm some conditional check, for this requirement we should go for Bipredicate.
- BiPredicate is exactly same as Predicate except that it will take 2 input arguments.

```
interface BiPredicate<T1,T2>{
	public boolean test(T1 t1,T2 t2);
	//remaining default and static methods are same
}
```

```
package org.ram.java8Features;
import java.util.function.BiPredicate;

public class BiPredicateTest{
	public static void main(String[] args) {
		BiPredicate<Integer, Integer> bp=(a,b)->(a+b)%2==0;
		System.out.println(bp.test(10, 30)); // true
		System.out.println(bp.test(10, 31)); //false
	}
}
```

BiFunction

Interface BiFunction<T,U,R>

Type Parameters:

T - the type of the first argument to the function

U - the type of the second argument to the function

R - the type of the result of the function

```
interface BiFunction<T,U,R>{
	public R apply(T t,U u);
	//remaining default method andThen()
}
```

```
package org.ram.java8Features.twoInputs;
import java.util.ArrayList;
import java.util.function.BiFunction;

class Employee{
	int eno;
	String name;
	public Employee(int eno, String name) {
		super();
		this.eno = eno;
		this.name = name;
```

```
	}
}

public class BiFunctionTest {
	public static void main(String[] args) {
		ArrayList<Employee> l=new ArrayList<>();
		BiFunction<Integer, String, Employee> bf=(eno,name)->new
Employee(eno, name);
		l.add(bf.apply(100, "abc"));
		l.add(bf.apply(200, "xyz"));
		l.add(bf.apply(300, "mnp"));
		l.add(bf.apply(400, "pqr"));
		for(Employee e:l)
			System.out.println(e.eno+","+e.name);
	}
}
```

Output

```
100,abc
200,xyz
300,mnp
400,pqr
```

BiConsumer

Interface BiConsumer<T,U>

- **Type Parameters:**

  T - the type of the first argument to the operation

  U - the type of the second argument to the operation

Functional Interface:

This is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference.

@FunctionalInterface
public interface **BiConsumer<T,U>**

Represents an operation that accepts two input arguments and returns no result. This is the two-arity specialization of Consumer. Unlike most other functional interfaces, BiConsumer is expected to operate via side-effects.

This is a functional interface whose functional method is accept(Object, Object).

```
package org.ram.java8Features.twoInputs;
import java.util.ArrayList;
import java.util.function.BiConsumer;

class Employee1{
	double salary;
	String name;
	public Employee1(double salary, String name) {
		super();
		this.salary = salary;
		this.name = name;

	}
}

public class BiConsumerTest {
	public static void main(String[] args) {
		ArrayList<Employee1> l=new ArrayList<>();
		populate(l);
		BiConsumer<Employee1, Double>c=(e,d)-
>e.salary=e.salary+d;
		for(Employee1 e:l) {
			c.accept(e, (double) 500);
		}
		for(Employee1 e1:l) {
			System.out.println(e1.name+","+e1.salary);
		}
		}

	private static void populate(ArrayList<Employee1> l) {
		l.add(new Employee1(100,"abc"));
		l.add(new Employee1(200,"pqr"));
		l.add(new Employee1(300,"mnp"));
		l.add(new Employee1(400,"xyz"));
	}
}
```

Output

abc,600.0
pqr,700.0
mnp,800.0
xyz,900.0

Pimitive Version Of Predicate Interface

- **DoublePredicate**
- **LongPredicate**
- **IntPredicate** : It always take some int value.

```
package org.ram.java8Features.primitiveFunctional;
import java.util.function.IntPredicate;
public class IntPredicateTest {
public static void main(String[] args) {
IntPredicate p=i->i%2==0;
System.out.println(p.test(5));
System.out.println(p.test(50));
}}
```

Pimitive Version Of Functional Interface

- **DoubleFunction**: Can take input type as double and return type can be any type.
- **IntFunction**: Can take int type as input and return type can be any type.
- **LongFunction**: Can take long type as input and return type can be any type.
- **DoubleToIntFunction**: Input type-> double, Return type-> int, method name->applyAsInt(double value).
- **DoubleToLongFunction**: Input type-> double, Return type-> long, method name-> applyAsLong(double value).
- **IntToDouble**: method name-> applyAsDouble(int value).
- **IntToLong**: method name-> applyAsLong(int value).
- **ToIntFunction**: return type-> any thing, input type->int
- **ToDoubleFunction**: return type-> any thing, input type->double
- **ToIntBiFunction**: input type->any thing, return type->int
- **ToLongBiFunction**: input type->any thing, return type->long
- **ToDoubleBiFunction**: input type->any thing, return type->double

Pimitive Version Of Consumer Interface

- **IntConsumer** : void accept(int value)
- **LongConsumer**:
- **DoubleConsumer**:
- **ObjDoubleConsumer**: void accept(T t, double value)
- **ObjLongConsumer**:
- **ObjeDoubleConsumer**:

Pimitive Version Of Supplier Interface

- **BooleanSupplier:** getAsBoolean()
- **IntSupplier:** getAsInt()
- **LongSupplier:** getAsLong()
- **DoubleSupplier:** getAsDouble()

UnaryOperator

- UnaryOperator is child of function.
- If input and output are same type then we should go for UnaryOperator.
- Primitive version of UnaryOperator is:
    - IntUnaryOperator : public int applyAsInt(int)
    - LongUnaryOperator : public int applyAsInt(long)
    - DoubleUnaryOperator: public int applyAsInt(double)

```
package org.ram.java8Features.unaryOperator;

import java.util.function.IntUnaryOperator;
import java.util.function.UnaryOperator;

public class Test1 {
	public static void main(String[] args) {
		//UnaryOperator<Integer> u=i->i*i;
//		System.out.println("Square of 4 is: "+u.apply(4));
		IntUnaryOperator u=i->i*i;
		System.out.println("Square of 4 is: "+u.applyAsInt(4));
	}
}
```

Output:

```
Square of 4 is: 16
```

BinaryOperator:

- Child of BiFunction<T,T,T>
- BinaryOperator<T>

```
package org.ram.java8Features.binaryOperator;
import java.util.function.BiFunction;
import java.util.function.BinaryOperator;
public class Test
{
	public static void main(String[] args) {
//		BiFunction<String, String, String> f=(s1,s2)->s1+s2;
//	System.out.println(f.apply("Hello ", "World"));//Hello World
//		BinaryOperator<String> b=(s1,s2)->s1+s2;
//		System.out.println(b.apply("Hello ", "World"));
	}
}
```

Output

```
Hello World
```

Primitive types of Binary Operator

- **IntBinaryOperator**

```
package org.ram.java8Features.binaryOperator;

import java.util.function.BiFunction;
import java.util.function.BinaryOperator;
import java.util.function.IntBinaryOperator;

public class Test
{
   public static void main(String[] args) {
//       BiFunction<String, String, String> f=(s1,s2)->s1+s2;
//       System.out.println(f.apply("Hello ", "World"));//Hello World

//       BinaryOperator<String> b=(s1,s2)->s1+s2;
//       System.out.println(b.apply("Hello ", "World"));

         IntBinaryOperator b=(a,a1)->a+a1;
         System.out.println(b.applyAsInt(4, 5)); //9


   }
}
```

- **LongBinaryOperator**
- **DoubleBinaryOperator**

Method and Constructor Reference:

Method Reference

- This is alternative of lambda expression.
- Where ever lambda expression is required used Method reference.
- Biggest advantage of doing this is code reusability.
- Functional interface method can be maped to our specified method by using double colon :: operator.
- For method we reference we can use static as well as instance method anyone is fine.
- If static method Syntax is ClassName::MethodName;
- If static method Syntax is ObjectReference::MethodName;
- In method reference argument type must be same. It never worry about return type and access modifier.
- If implementation is available go for method reference.
- If implementation is not available then go for lambda expression.

```
//code without Method reference
package org.ram.java8Features.methodAndObjectReference;
public class Test {
      public static void main(String[] args) {
            Runnable runnable=()-> {
                        for (int i = 0; i < 3; i++) {
```

```
                        System.out.println("Child Thread.");
                }
            };
            Thread t = new Thread(runnable);
            t.start();
            for (int i = 0; i < 3; i++) {
                System.out.println("Main Thread.");
            }
        }
}
```

Output

```
Main Thread
Main Thread
Main Thread
Child Thread
Child Thread
Child Thread
```

```
//code with Method reference
package org.ram.java8Features.methodAndObjectReference;
public class MethodReference1 {
    public static void methodReference() {
        for (int i = 0; i < 3; i++) {
            System.out.println("Child Thread");
        }
    }
    public static void main(String[] args) {
//        Runnable r=()->{
//            for (int i = 0; i < 3; i++) {
//                System.out.println("Child Thread");
//            }
//        };
        Runnable r=MethodReference1::methodReference;//method
reference
        Thread t= new Thread(r);
        t.start();
        for (int i = 0; i < 3; i++) {
            System.out.println("Main Thread");
        }
    }
}
```

Output

```
Main Thread
Main Thread
Main Thread
Child Thread
Child Thread
Child Thread
```

```
//method reference with return type
package org.ram.java8Features.methodAndObjectReference;
interface Intrf{
	public void add(int firstNumber, int secondNumber);
}
public class MethodReference2 {
	public int sum(int number1, int number2) {
		System.out.println("Sum is: "+(number1+number2));
		return number1+number2;
	}
	public static void main(String[] args) {
		MethodReference2 m=new MethodReference2();
		Intrf i=m::sum;
		i.add(100, 200);
	}
}
```

Output

```
Sum is: 300
```

Constructor Reference

- We use Constructor Reference when a functional interface method returns an object, then we use Constructor Reference.
- If the method has some argument then constructor also has argument.
- This is alternative of lambda expression.
- Where ever lambda expression is required used Method reference.
- Biggest advantage of doing this is code reusability.

```
package org.ram.java8Features.constructorReference;

class Sample{
	public Sample() {
		System.out.println("Sample class constructor executed.");
	}
}

interface Intrf{
	public Sample get();
}

public class Test {
	public static void main(String[] args) {
		Intrf i=Sample::new;
		Sample s=i.get();
	}
}
```

Output -> `Sample class constructor executed.`

```
package org.ram.java8Features.constructorReference;
class Sample{
	public Sample() {
		System.out.println("Sample class constructor executed.");
	}
	public Sample(String s) {
		System.out.println("Sample class Parameterized
constructor executed.");
	}
}

interface Intrf{
	//public Sample get();
	public Sample get(String s);
}

public class Test {
	public static void main(String[] args) {
		Intrf i=Sample::new;
		Sample s=i.get("Hello");
	}
}
```

Output

```
Sample class Parameterized constructor executed.
```

```
package org.ram.java8Features.constructorReference;
class Student{
	int roll;
	String name;
	int marks;
	int age;
	public Student(int roll, String name, int marks, int age) {
		super();
		this.roll = roll;
		this.name = name;
		this.marks = marks;
		this.age = age;
	}
}
//class Demo implements Intrfe{
//	@Override
//	public Student get(int roll, String name, int marks, int age) {
//		Student s=new Student(1, "abc", 56, 12);
//		return s;
//	}


interface Intrfe{
	public Student get(int roll,String name, int marks,int age);
```

```
}
public class StudentTest {
	public static void main(String[] args) {
		Intrfe i=(roll,name,marks,age)->new Student(1, "abc", 56,
120);
		//Intrfe i=Student::new;
	}
}
```

Stream API

- If we want to represent a group of objects as a single entity then we go for collection.
- If we want to process objects from the collection then we go for stream.
- Stream API is new feature available from java8 .
- Stream API is available in java.util.stream package.

How to create a stream

- To call the stream() from collection interface.

How to configure the stream?

- By using filter() and map() we can configure the stream.
- filter() is used to filter the record .
    - filter(Predicate)// Predicate return boolean value.
- map() :- for every object if you want perform some function and want some result object then we go for map().
    - map(Function) //Function is used to perform business logic

Where Stream is applicable?

- Wherever collection is present stream is applicable.

collect()

The collect() method of Stream class can be used to accumulate elements of any Stream into a Collection. In Java 8, you will often write code which converts a Collection like a List or Set to Stream and then applies some logic using functional programming methods like the filter, map, flatMap and then converts the result back to the Collection like a List, Set, Map, or ConcurrentMap in Java. In this last part, the collect() method of Stream helps. It allows you to accumulate the result into a choice for containers you want like a list, set, or a map.

Programmers often confuse that the collect() method belongs to the Collector class but that's not true. It is defined in Stream class and that's why you can call it on Stream after doing any filtering or mapping. It accepts a Collector to accumulate elements of Stream into a specified Collection.

The Collector class provides different methods like toList(), toSet(), toMap(),
and toConcurrentMap() to collect the result of Stream into List, Set, Map, and ConcurrentMap in Java.

It also provides a special toCollection() method which can be used to collect Stream elements into a specified Collection like ArrayList, Vector, LinkedList, or HashSet.

It's also a terminal operation which means after calling this method on Stream, you cannot call any other method on Stream.

Count()

- This method is used to count the number of elements present in stream.
- Stream.count() method simply returns the number of elements in the stream with or without filter condition is applied.

```
//checking how many students failed
package ArrayList;
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;

import com.sun.net.httpserver.Filter;
public class AddGraceMarks
{
	public static void main(String[] args)
	{
		ArrayList<Integer> marks= new ArrayList<Integer>();
		marks.add(56);
		marks.add(89);
		marks.add(26);
		marks.add(45);
		marks.add(15);
		marks.add(78);
		System.out.println(marks);
		List<Integer> l=
		marks.stream().map(i->i+5)
		.collect(Collectors.toList());
		//Map() :- for every object if you want perform some function
		//and want some result object then we go fro map().
		long nuoOfFailedStudent=
		marks.stream().filter(m->m<35).count();
	System.out.println("number of failed student: "+nuoOfFailedStudent);
		System.out.println(l);
	}
}
```

sorted()

- This method is used to sort the elements in natural sorting order.
- By default sorting order is ascending order.
- For customize sorting we can use comparator interface.

Natural Sorting order code by using sorted()

```
ArrayList<Integer> marks= new ArrayList<Integer>();
            marks.add(56);
            marks.add(89);
            marks.add(26);
            marks.add(45);
            marks.add(15);
            marks.add(78);
            System.out.println(marks);
List<Integer>l=marks.stream().sorted().collect(Collectors.toList());
            System.out.println(l);
```

Output

```
[56, 89, 26, 45, 15, 78]
[15, 26, 45, 56, 78, 89]
```

Customize Sorting order code by using comparator interface

```
ArrayList<Integer> marks= new ArrayList<Integer>();
            marks.add(56);
            marks.add(89);
            marks.add(26);
            marks.add(45);
            marks.add(15);
            marks.add(78);
            System.out.println(marks);
            //code for customize sorting order in decending order
            //(m1,m2)->(m1<m2)?1:(m1>m2)?-1:0→comparator code
            List<Integer>l=
      marks.stream()
.sorted((m1,m2)->(m1<m2)?1:(m1>m2)?-
1:0).collect(Collectors.toList());
            System.out.println(l);
```

Output:

```
[56, 89, 26, 45, 15, 78]
[89, 78, 56, 45, 26, 15]
```

Note:

- Internally in default sorting order compareTo() is invoked.

```
      ArrayList<Integer> marks= new ArrayList<Integer>();
                  marks.add(56);
                  marks.add(89);
                  marks.add(26);
```

```
                marks.add(45);
                marks.add(15);
                marks.add(78);
                System.out.println(marks);
                List<Integer>l=marks.stream().sorted((m1,m2)-
>m1.compareTo(m2)).collect(Collectors.toList());
                System.out.println(l);
```

Output
[56, 89, 26, 45, 15, 78]
[15, 26, 45, 56, 78, 89]

- **Just by adding –(minus) sign we can reverse the default natural sorting order.**

```
List<Integer>l=marks.stream().sorted((m1,m2)->-
m1.compareTo(m2)).collect(Collectors.toList());
```

Output:
[56, 89, 26, 45, 15, 78]
[89, 78, 56, 45, 26, 15]

- For string also default natural sorting order is alphabetical order.

```
ArrayList<String> name= new ArrayList<String>();
        name.add("Farooq");
        name.add("Rinky");
        name.add("Neha");
        name.add("Sangeeta");
        name.add("Avinash");
        name.add("Abhijeet");
        name.add("Prabjot");
        System.out.println(name);

        List<String>l=name.stream()
        .sorted()
        .collect(Collectors.toList());

        System.out.println(l);
```

OutPut:
[Farooq, Rinky, Neha, Sangeeta, Avinash, Abhijeet, Prabjot]
[Abhijeet, Avinash, Farooq, Neha, Prabjot, Rinky, Sangeeta]

- Decreasing order

```
List<String>l=name.stream().sorted((n1,n2)-
>n2.compareTo(n1)).collect(Collectors.toList());
```

OutPut:
[Farooq, Rinky, Neha, Sangeeta, Avinash, Abhijeet, Prabjot]
[Sangeeta, Rinky, Prabjot, Neha, Farooq, Avinash, Abhijeet]

Sorting based on length of String.

```
package org.ram.java8Features.streamApi;
import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;
import java.util.stream.Collectors;

public class CompareStringLengthWise
{
	public static void main(String[] args)
	{
		List<String> name=new ArrayList<>();
		name.add("ravi");
		name.add("manish");
		name.add("rinky");
		name.add("akanksha");
		name.add("vivek");
		name.add("parmod");
		System.out.println(name);

		Comparator<String>c=(a,b)->{
			int l1=a.length();
			int l2=b.length();
			if(l1<l2) return -1;
			else if (l1>l2) return +1;
			else return a.compareTo(b);
		};

		List<String>
sort=name.stream().sorted(c).collect(Collectors.toList());
		System.out.println(sort);
	}
}
```

Output
[ravi, manish, rinky, akanksha, vivek, parmod]
[ravi, rinky, vivek, manish, parmod, akanksha]

min() and max()

- These methods present inside stream.
- According to the sorting order first and last element is min element and max element.
- Without comparator object we can't be able to find min() and max() element.

```
ArrayList<Integer> marks= new ArrayList<Integer>();
marks.add(56);
marks.add(89);
marks.add(26);
marks.add(45);
marks.add(15);
marks.add(78);
```

```
                System.out.println(marks);
                      Integer min=marks.stream().min((m1,m2)-
    >m1.compareTo(m2)).get();
                System.out.println(min);//15
                Integer max=marks.stream().max((m1,m2)-
  >m1.compareTo(m2)).get();
                System.out.println(max);//89
```

forEach()

- This method is used to process the each element from stream.

```
ArrayList<Integer> marks= new ArrayList<Integer>();
      marks.add(56);
      marks.add(89);
      marks.add(26);
      marks.add(45);
      marks.add(15);
      marks.add(78);

      System.out.println(marks);
marks.stream().forEach(System.out::println);
```

Program2

```
      ArrayList<Integer> marks= new ArrayList<Integer>();
      marks.add(56);
      marks.add(89);
      marks.add(26);
      marks.add(45);
      marks.add(15);
      marks.add(78);
      System.out.println(marks);
      Consumer<Integer>c=i->
      {
            System.out.println("The square of "+i+" is: "+i*i);
      };

            marks.stream().forEach(c);
```

toArray()

- toArray() is used to convert stream of object into array.

```
ArrayList<Integer> marks= new ArrayList<Integer>();
      marks.add(56);
      marks.add(89);
      marks.add(26);
      marks.add(45);
      marks.add(15);
      marks.add(78);
      System.out.println(marks);
      Integer[] i = marks.stream().toArray(Integer[]::new);
      for (Integer integer : i) {
      System.out.println(integer);  }
```

Stream.of()

- Where ever group of values are there then we can go for stream concept.

```
package org.ram.java8Features.streamApi;

import java.util.ArrayList;

import java.util.stream.Stream;

public class StreamOf {

	public static void main(String[] args) {

		ArrayList<Integer> l= new ArrayList<>();

		l.add(0);

		l.add(5);

		l.add(10);

		l.add(15);

		l.add(20);

		l.add(25);

		System.out.println(l);

		Integer[] i=l.stream().toArray(Integer[]::new);

		Stream.of(i).forEach(System.out::println);

	}

}
```

Output

```
[0, 5, 10, 15, 20, 25]
0
5
10
15
20
25
```

```
package org.ram.java8Features.streamApi;
import java.util.stream.Stream;

public class StreamOf1
{
public static void main(String[] args)
{
	Stream s=Stream.of(11,22,111,333,44,2266177,444);
	s.forEach(System.out::println);
```

```
        Integer[] i= {10,80,20,60,40};
        Stream.of(i).forEach(System.out::println);
}
}
```

Output

```
11
22
111
333
44
2266177
444
10
80
20
60
40
```

Date and Time Api

- Until 1.7 to handel date and time we have Date class, Calender class, TimeStamep ,etc.. classes are available.
- These classes are not recommended to use.
- In java 1.8 new api is introduced to handel date and time.
- These concept is also known as JODA Time api. This api is developed by joda.org organisation.
- If we want represent current date and time then we used this.

```
package org.ram.java8Features.dateAndTime;
import java.time.LocalDate;
import java.time.LocalTime;

public class Program1 {
   public static void main(String[] args) {
         LocalDate date= LocalDate.now();
         System.out.println(date);

         LocalTime time = LocalTime.now();
         System.out.println(time);
   }
}
```

Output

```
2021-07-23
11:41:38.208426200
```

Extract Day Value, Month Value and Year Value from Date

```
package org.ram.java8Features.dateAndTime;
import java.time.LocalDate;

public class Program2 {
	public static void main(String[] args) {
		LocalDate date= LocalDate.now();
		System.out.println("Date is: "+date);

		int dd =date.getDayOfMonth();
		int mm=date.getMonthValue();
		int yy=date.getYear();

		System.out.println(dd+":"+mm+":"+yy);
	}
}
```

Output

Date is: 2021-07-23
23:7:2021

Get current system time.

```
package org.ram.java8Features.dateAndTime;
import java.time.LocalTime;

public class Program3 {
	public static void main(String[] args) {
		LocalTime time=LocalTime.now();

		System.out.println(time);

		int h=time.getHour();
		int m=time.getMinute();
		int s=time.getSecond();
		int ns=time.getNano();

		System.out.println(h+"-"+m+"-"+s+"-"+ns);

	}
}
```

Output

11:52:36.666539100
11-52-36-666539100

LocalDateTime

```
package org.ram.java8Features.dateAndTime;
import java.time.LocalDateTime;
public class Program4 {
	public static void main(String[] args) {
		LocalDateTime dt=LocalDateTime.now();
```

```
            System.out.println(dt);
            int dd=dt.getDayOfMonth();
            int mm=dt.getMonthValue();
            int yy=dt.getYear();
            System.out.println(dd+"-"+mm+"-"+yy);

            int s=dt.getSecond();
            int m=dt.getMinute();
            int h=dt.getHour();
            System.out.println(h+"/"+m+"/"+s);
      }
}
```

Output
2021-07-23T14:02:18.790875700
23-7-2021
14/2/18

Show Particular Date and Time

- LocalDateTime.of(yy,mm,dd,h,m,s,n);
- There is various overloaded of() is present to show date and time.

```
//particular date and time
package org.ram.java8Features.dateAndTime;

import java.time.LocalDateTime;
import java.time.Month;

public class OfMethod {
      public static void main(String[] args) {
            LocalDateTime  dt=LocalDateTime.of(1995, Month.MAY, 11, 19, 10,
15, 36);
            System.out.println(dt);
      }
}
```

Output
1995-05-11T19:10:15.000000036

plussDay(),plusMonths(),minusMonths, plusYears

- This is used to add or subtract days,month and year from a given date or year or month.
- Simillar methods are also available for time.

```
package org.ram.java8Features.dateAndTime;
import java.time.LocalDateTime;
class PlusMinus
{
      public static void main(String[] args) {
            LocalDateTime dt= LocalDateTime.now();
            LocalDateTime m=dt.plusMonths(85);
```

```
            LocalDateTime m1=dt.minusMonths(17);
            System.out.println(dt);
            System.out.println(m);
            System.out.println(m1);
      LocalDateTime addyear=dt.plusYears(2).plusMonths(6).plusDays(52);
      System.out.println(addyear);


      }
}
```

Period

- This is used to represent quantity of time.
- Like number of days, number of moths, number of years.

```
package org.ram.java8Features.dateAndTime.period;

import java.time.LocalDate;
import java.time.Period;

public class Age {
      public static void main(String[] args) {
            LocalDate birthday=LocalDate.of(1996,12,28);
            LocalDate today=LocalDate.now();

            Period p=Period.between(birthday, today);
            System.out.println("Your age is: "+p.getYears()+"Years,
"+p.getMonths()+"Months, "+p.getDays()+"Days  ");

            LocalDate death=LocalDate.of(1996+80,12,28);
            Period p1=Period.between(birthday, death);
            System.out.println("Your age is: "+p1.getYears()+"Years,
"+p1.getMonths()+"Months, "+p1.getDays()+"Days  ");


      }
}
```

Output

Your age is: 24Years, 6Months, 25Days
Your age is: 80Years, 0Months, 0Days

Year Class API

```
//check leap year
package org.ram.java8Features.dateAndTime.YearClass;
import java.time.Year;

public class CheckYear {
public static void main(String[] args) {
      Year y= Year.of(2210);
```

```
	if(y.isLeap())
		System.out.println("leap year");
	else
		System.out.println("not leap year");
}
}
```

ZoneId Class Api

```
package org.ram.java8Features.dateAndTime.zoneIdClass;
import java.time.ZoneId;
import java.time.ZonedDateTime;

public class CheckZoneId
{
	public static void main(String[] args) {

		//local zone id
		ZoneId id=ZoneId.systemDefault();
		System.out.println(id);
		ZonedDateTime zdt=ZonedDateTime.now();
		System.out.println(zdt);

		//america/ LA id
		ZoneId la=ZoneId.of("America/Los_Angeles");
		System.out.println(la);
		ZonedDateTime zdt1=ZonedDateTime.now(la);
		System.out.println(zdt1);
	}
}
```

Output
Asia/Calcutta
2021-07-23T16:49:00.691687900+05:30[Asia/Calcutta]
America/Los_Angeles
2021-07-23T04:19:00.697683900-07:00[America/Los_Angeles]

Java Optional Class
Java introduced a new class Optional in jdk8. It is a public final class and used to deal with NullPointerException in Java application. You must import java.util package to use this class. It provides methods which are used to check the presence of value for particular variable.

Optional Class Method

Sr.No.	Method & Description
1	**static <T> Optional<T> empty()** Returns an empty Optional instance.
2	**boolean equals(Object obj)** Indicates whether some other object is "equal to" this Optional.
3	**Optional<T> filter(Predicate<? super <T> predicate)** If a value is present and the value matches a given predicate, it returns an Optional describing the value, otherwise returns an empty Optional.
4	**<U> Optional<U> flatMap(Function<? super T,Optional<U>> mapper)** If a value is present, it applies the provided Optional-bearing mapping function to it, returns that result, otherwise returns an empty Optional.
5	**T get()** If a value is present in this Optional, returns the value, otherwise throws NoSuchElementException.
6	**int hashCode()** Returns the hash code value of the present value, if any, or 0 (zero) if no value is present.
7	**void ifPresent(Consumer<? super T> consumer)** If a value is present, it invokes the specified consumer with the value, otherwise does nothing.
8	**boolean isPresent()** Returns true if there is a value present, otherwise false.
9	**<U>Optional<U> map(Function<? super T,? extends U> mapper)** If a value is present, applies the provided mapping function to it, and if the result is non-null, returns an Optional describing the result.

10	**static <T> Optional<T> of(T value)** Returns an Optional with the specified present non-null value.
11	**static <T> Optional<T> ofNullable(T value)** Returns an Optional describing the specified value, if non-null, otherwise returns an empty Optional.
12	**T orElse(T other)** Returns the value if present, otherwise returns other.
13	**T orElseGet(Supplier<? extends T> other)** Returns the value if present, otherwise invokes other and returns the result of that invocation.
14	**<X extends Throwable> T orElseThrow(Supplier<? extends X> exceptionSupplier)** Returns the contained value, if present, otherwise throws an exception to be created by the provided supplier.
15	**String toString()** Returns a non-empty string representation of this Optional suitable for debugging.

- This class inherits methods from the following class – java.lang.Object

```
package org.ram.java8Features.optionalClass;
import java.util.Optional;

public class OptionalExmple {

	public static Optional<String> getName(){
//		String name="sachin";
		String name=null;
		return Optional.of(name);
	}

	public static void main(String[] args) {

		//String str=null;
		String str="java8";
```

```
//          if(str==null)
//          {
//                System.out.println("this is null object");
//          }
//          else {
//                System.out.println(str.length());
//          }

            Optional o=Optional.ofNullable(str);
            System.out.println(o.isPresent());//true
            System.out.println(o.get());//java
            System.out.println(o.orElse("no value"));

            Optional nameOptional= getName();
            System.out.println(nameOptional.orElse("null return"));
      }
}
```