In Collection f/w we have 2 utility classes are there:

java.util.Arrays : it is used for normal array
java.util.Collections : it is used for Collection f/w related classes

--both classes provides lots of static methods to perform utility operations.

java.util.Arrays:

```
int[] arr= {10,20,30,40,50};

System.out.println(Arrays.toString(arr));
```

ex2:

```
int[] arr= {10,2,15,12,5};

System.out.println(Arrays.toString(arr));

Arrays.sort(arr);

System.out.println(Arrays.toString(arr));
```

ex2:

```
int[] arr= {10,2,15,12,5};


Arrays.sort(arr);

System.out.println(Arrays.toString(arr));


int index= Arrays.binarySearch(arr, 7);

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

ex4:

```
List<String> cities= Arrays.asList("delhi","mumbai","chennai","kolkata");
```

java.util.Collections:
==========

ex1:

```
        List<String> colors=
Arrays.asList("blue","red","purple","white","orange","blue","blue");


                int result= Collections.frequency(colors, "blue");

                System.out.println(result);//3
```

ex2:

```
List<String> colors= Arrays.asList("blue","red","purple","white","orange","blue","blue");


                Collections.sort(colors);

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

ex3:

```
                List<Student> students = new ArrayList<>();

                students.add(new Student(10, "n1", 780));
                students.add(new Student(8, "n5", 680));
                students.add(new Student(12, "n3", 480));
                students.add(new Student(14, "n2", 880));
                students.add(new Student(6, "n4", 580));


                Collections.sort(students, new StudentRollComp());

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

Functional programming:
==========

--this concept is introduced in java 1.8 v.

--in this type of paradigm, a function is treated as a value, (we can assign the entire function in a variable, or we can pass a function to another function parameter or we can return a function from another function).

int x = 10;

--the main adv of a FP is less coding, polymorphic and easy to understand.

--to achive the the FP in java we need a "Functional interface".

Functional interface in java:
==========
--an interface which has only one abstract method is called a FI.

--A FI may have n number of static and default methods.

--A FI may have some data members (variables) also.

--A FI can have Object class related methods also.

example:

Intr.java:

```
package com.masai;

@FunctionalInterface
public interface Intr {

	int x=10;
	public abstract void fun1();



	public abstract String toString();
```

}

@FunctionalInterface annotation make sure that we have a valid FI.

Some of the predefined FI in java:
==========

java.lang.Comparable : public int compareTo(Object obj);

java.util.Comparator : public int compare(Object obj1, Object obj2);

java.lang.Iterable : public Iterator iterator();

java.lang.Runnable : public void run();

****Note: with the help of FI we achive FP in java using Lamda expression.**

example:

Demo.java:

```
package com.masai;

public class Demo {

	public static void main(String[] args) {

		//using External class
		Intr i1= new IntrImpl();
		i1.sayHello("Ram");

		//using Annonymous Inner class
		Intr i2 = new Intr() {

			@Override
			public void sayHello(String name) {
				System.out.println("Welcome Using Annonymous inner class
"+name);
```

```
                }

        };

        i2.sayHello("Ram");

        //using Lambda expression
        Intr i3= n -> System.out.println("Welcome Using LE :"+n);

        i3.sayHello("Ram");


    }

}
```

Rules to use LE:
==========

--LE is a implementation of functional interface (using LE we can provide the implementation of a FI in much more easier way)

LE comprises 3 things:

1. parameters : (here data type is optional, and variable name could be anything), If only one parameter is there then small bracket is also optional.

2. lambda operator : ->

3. method body: if we write only one statement inside the implementation body then curly bracket {} is also optional.

Note: LE does not consider the FI method name.

example: longest implementation of above Intr interface

```
        Intr i1 = (String name) -> {

            System.out.println("Inside LE :"+name);

        };
```

example: shor implementation:

```
Intr i1 = n -> System.out.println("Inside LE :"+n);
```

example2:

Intr.java:

```
package com.masai;

@FunctionalInterface
public interface Intr {

	void sayHello(Student student);

}
```

Demo.java:

```
package com.masai;

public class Demo {

	public static void main(String[] args) {


		Intr i2 = s -> {

			System.out.println("Roll is :"+s.getRoll());
			System.out.println("Name is :"+s.getName());
			System.out.println("Marks is :"+s.getMarks());

		};


		i2.sayHello(new Student(10, "Rahul", 700));
```

```
	}

}
```

LE with return type:
==========

--if inside the method body only one statement is there then {} is optional, and return keyword is not allowed.

example:

Intr.java:

```
package com.masai;

@FunctionalInterface
public interface Intr {

	String sayHello(String name);

}
```

Demo.java:

```
package com.masai;

public class Demo {

	public static void main(String[] args) {

		Intr i2 = name -> "Welcome :" + name.toLowerCase();

		System.out.println(i2.sayHello("Amit"));

	}

}
```

example3:

Intr.java

```
package com.masai;

@FunctionalInterface
public interface Intr {

	Student getStudent(int roll, String name, int marks);

}
```

Demo.java:

```
package com.masai;

public class Demo {

	public static void main(String[] args) {


		Intr i2 = (roll,name,marks) -> new Student(roll, name.toUpperCase(),
marks+100);

		Student s= i2.getStudent(10, "Ravi", 500);

		System.out.println(s);



	}

}
```

Method reference:
==========

--it is the simplified form of (short-cut) of LE:
==========

--it is represented using :: double colon symbol.

--instead of creating a LE with all the details, with the help of MR we can refer an existing method of any class to the functional interface variable.

Note:

--we can take a reference of a non-static method using object::methodName

--we can take a reference of a static method using ClassName::methodName

--we can take a reference of a constructor also using ClassName::new

example:

Intr.java:

```
package com.masai;

@FunctionalInterface
public interface Intr {

	void sayHello(String message);

}
```

Demo.java:

```
package com.masai;

public class Demo {
```

```
public static void fun1(String s) {

        System.out.println("inside static fun1 of Demo ");
        System.out.println("The value of s is :"+s);

}


public void fun2(String s) {

        System.out.println("inside non-static fun2 of Demo ");
        System.out.println("The value of s is :"+s);

}


public Demo(String s) {

        System.out.println("inside Constructor of Demo ");
        System.out.println("The value of s is :"+s);


}


public static void main(String[] args) {


        Intr i1 = Demo::fun1; //refering to static method


        Intr i2 = new Demo("amit")::fun2; // refering to non-static method


        Intr i3 = Demo::new; //refering to the constructor

        i1.sayHello("Welcome to MR");
        i2.sayHello("Welcome to MR");
        i3.sayHello("Welcome to MR");



}
```

```
}
```

example 2:

Intr.java:

```
package com.masai;

@FunctionalInterface
public interface Intr {

	int getTheNumber(String number);

}
```

Demo.java:

```
package com.masai;

public class Demo {

	public static void main(String[] args) {

		Intr i1 = number ->	Integer.parseInt(number); //using LE

		Intr i2= Integer::parseInt;// refering to the parseInt method of Integer class



		System.out.println(i1.getTheNumber("100")+50);

	}

}
```

example 3:

Intr.java:

```
----------
```

```
package com.masai;

@FunctionalInterface
public interface Intr {

	void printSomeThing(String s);

}
```

Demo.java:

```
---------------
```

```
package com.masai;

public class Demo {


	public static void main(String[] args) {

		Intr i1 = s -> System.out.println(s); //using LE

		i1.printSomeThing("Hello");

		Intr i2 = System.out::println; //refering println method of PrintStream object

		i2.printSomeThing("Welcome");

	}

}
```