Overview of Predined Functional interfaces :

1) Predicate<T> : public abstract boolean test(T x)
2) Consumer<T> : public abstract void accept(T x)
3) Function<T,R> : public abstract R apply(T x)
4) Supplier<T> : public abstract T get()
5) BiPredicate<T,U> : public abstract boolean test(T x, U y)
6) BiConsumer<T,U> : public abstract void accept(T x, U y)
7) BiFunction<T,U,R> : public abstract R apply(T x, U y)
8) UnaryOperator<T> : public abstract T apply(T x)
9) BinaryOperator<T,T> : public abstract T apply(T x, T y)

BiPredicate<T,U> functional interface :

It is a predefined functional interface available in java.util.function sub package.

It is a functional interface in Java that represents a predicate (a boolean-valued function) OF TWO ARGUMENTS.

The BiPredicate interface has method named test, which takes two parameters and returns a boolean value, basically this BiPredicate is same with the Predicate, instead, it takes 2 arguments for the metod test.

```
@FunctionalInterface
public interface BiPredicate<T, U>
{
    boolean test(T t, U u);
}
```

Type Parameters:

T - the type of the first argument to the predicate
U - the type of the second argument the predicate
Note : return type is boolean.

```
import java.util.function.*;
public class Lambda11
{
    public static void main(String[] args)
    {
        BiPredicate<String, Integer> filter = (x, y) ->
        {
            return x.length() == y;
        };

        boolean result = filter.test("Ravi", 4);
        System.out.println(result);

        result = filter.test("Hyderabad", 10);
        System.out.println(result);
    }
}
```

```
import java.util.function.BiPredicate;

public class Lambda12
{
    public static void main(String[] args)
    {
      // BiPredicate to check if the sum of two integers is even
      BiPredicate<Integer, Integer> isSumEven = (a, b) -> (a + b) % 2 == 0;

        System.out.println(isSumEven.test(2, 3));
        System.out.println(isSumEven.test(5, 7));
    }
}
```

BiConsumer<T, U> functional interface :

It is a predefined functional interface available in java.util.function sub package.

It is a functional interface in Java that represents an operation that accepts two input arguments and returns no result.

It takes a method named accept, which takes two parameters and performs an action without returning any result.

```
@FunctionalInterface
public interface BiConsumer<T, U>
{
    void accept(T t, U u);
}
```

```
import java.util.function.BiConsumer;

public class Lambda13
{
    public static void main(String[] args)
    {
        BiConsumer<Integer, String> updateVariables = (num, str) ->
        {
            num = num * 2;
            str = str.toUpperCase();
            System.out.println("Updated values: " + num + ", " + str);
        };

        int number = 15;
        String text = "hii";

        updateVariables.accept(number, text);

        System.out.println("Original values: " + number + ", " + text);
    }
}
```

BiFunction<T, U, R> Functional interface :

It is a predefined functional interface available in java.util.function sub package.

It is a functional interface in Java that represents a function that accepts two arguments and produces a result R.

The BiFunction interface has a method named apply that takes two arguments and returns a result of type R.

```
@FunctionalInterface
public interface BiFunction<T, U, R>
{
  R apply(T t, U u);
}
```

```
import java.util.function.BiFunction;

public class Lambda14
{
  public static void main(String[] args)
  {
      // BiFunction to concatenate two strings
      BiFunction<String, String, String> concatenateStrings = (str1, str2) -> str1 + str2;

        String result = concatenateStrings.apply("Hello", " Java");
        System.out.println(result);

      // BiFunction to find the length two strings
    BiFunction<String, String, Integer> calculateLength = (str1, str2) -> str1.length() + str2.length();

        Integer result = calculateLength.apply("Hello", "Java");
        System.out.println(result);
    }
}
```

UnaryOperator<T> :

It is a predefined functional interface available in java.util.function sub package.

It is a functional interface in Java that represents an operation on a single operand that produces a result of the same type as its operand. This is a specialization of Function for the case where the operand and result are of the same type.

It has a single type parameter, T, which represents both the operand type and the result type.

```
@FunctionalInterface
public interface UnaryOperator<T> extends Function<T,T>
{
public abstract T apply(T x);
}
```

```
import java.util.function.*;
public class Lambda15
{
    public static void main(String[] args)
    {
        UnaryOperator<Integer> square = x -> x*x;
        System.out.println(square.apply(5));

        UnaryOperator<String> concat = str -> str.concat("base");
        System.out.println(concat.apply("Data"));
    }
}
```

BinaryOperator<T>

It is a predefined functional interface available in java.util.function sub package.

It is a functional interface in Java that represents an operation upon two operands of the same type, producing a result of the same type as the operands.

This is a specialization of BiFunction for the case where the operands and the result are all of the same type.

It has two parameters of same type, T, which represents both the operand types and the result type.

```
@FunctionalInterface
public interface BinaryOperator<T> extends BiFunction<T,T,T>
{
  public abstract T apply(T x, T y);
}
```

```
import java.util.function.*;
public class Lambda16
{
    public static void main(String[] args)
    {
        BinaryOperator<Integer> add = (a, b) -> a + b;
        System.out.println(add.apply(3, 5));
    }
}
```

Can we use all different types of variables in the lambda body ?

Yes, we can use all different types of variables like static field, non static field and local variable in the lambda body.

The local variable must be final or effectively final otherwise we will get compilation error.

```
@FunctionalInterface
interface Drawable
{
    void draw();
}

public class Main
{
    public static void main(String[] args)
    {
        String shapeType = "Rectangle";
        //shapeType = "Circle";

        Drawable d1 = ()->
        {
            System.out.println("Drawing : "+shapeType);
        };

        d1.draw();
    }
}
```

Does an interface extends a class ?

* In Java, We have a hierarchy of Java classes, An interface cannot extend a class. An interface can extend another interface.

* Whenever we define an interface in java and If the interface does not contain any super interface then at the time of compilation automatically java compiler will add all the public non final methods of Object class as an abstract method to each and every interface.

* In order to support upcasting concept, Java compiler will add all these public non final methods as an abstract methods

```
package com.ravi.object_class_methods;

interface Drawable
{
    void draw();
}

public class InterfaceDemo1
{
    public static void main(String[] args)
    {
        Drawable d1 = null;
        d1.draw();
        d1.equals(null);
        d1.toString();
    }
}
```

```
package com.ravi.object_class_methods;

@FunctionalInterface
interface Printable
{
    void print();
    public boolean equals(Object obj);
    public String toString();
    public int hashCode();
}

public class InterfaceDemo2 {

    public static void main(String[] args) {
        // TODO Auto-generated method stub

    }

}
```

```
package com.ravi.object_class_methods;

@FunctionalInterface
interface Movable
{
    void move();
    public boolean equals(Object obj);
    public String toString();
    public int hashCode();
}

class Move extends Object implements Movable
{
    @Override
    public void move()
    {
        System.out.println("Moving");
    }
}

public class InterfaceDemo3
{
    public static void main(String[] args)
    {
        Movable m1 = new Move();
        System.out.println(m1.toString());

    }

}
```