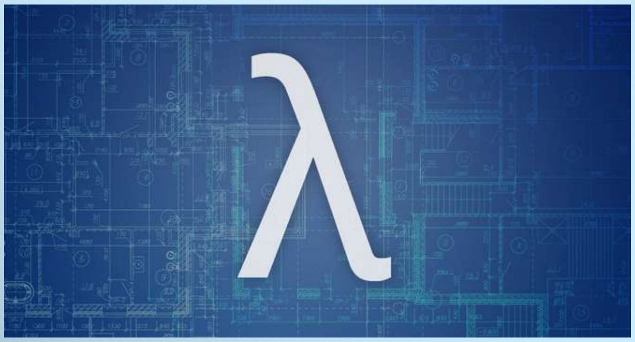
Lambda Expression & Functional Programming Part 2



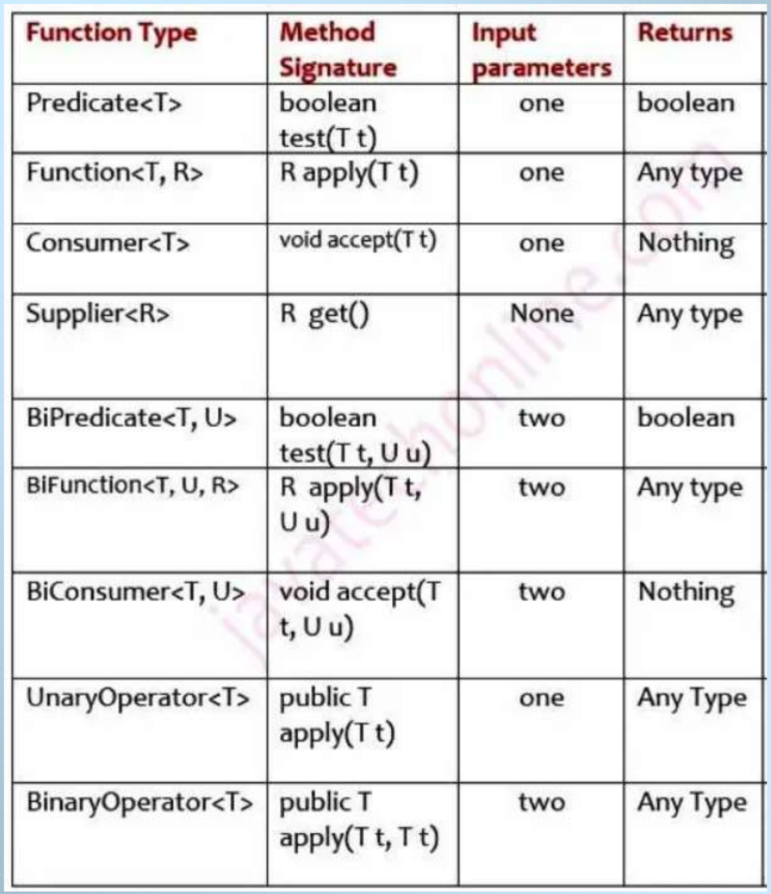
Objectives

- At the end of this topic, you should be able to
  - Explain the use of predefined functional interfaces.
  - Describe method references

Predefined Functional Interfaces

- Java has several built-in functional interfaces, which can be found in the java.util.function package
- Rather than defining your own, these interfaces can be used for your lambda expressions
- There are four main kinds of functional interfaces which can be applied in multiple situations:
  - Predicate
  - Function
  - Consumer
  - Supplier

Predefined Functional Interfaces

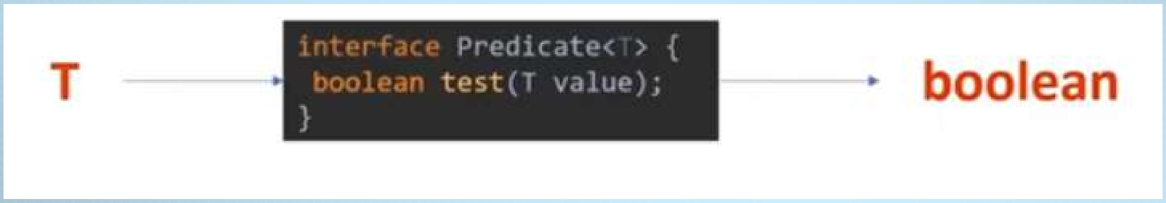


Function Type	Method Signature	Input parameters	Returns
Predicate<T>	boolean test(T t)	one	boolean
Function<T, R>	R apply(T t)	one	Any type
Consumer<T>	void accept(T t)	one	Nothing
Supplier<R>	R get()	None	Any type
BiPredicate<T, U>	boolean test(T t, U u)	two	boolean
BiFunction<T, U, R>	R apply(T t, U u)	two	Any type
BiConsumer<T, U>	void accept(T t, U u)	two	Nothing
UnaryOperator<T>	public T apply(T t)	one	Any Type
BinaryOperator<T>	public T apply(T t, T t)	two	Any Type

javatechonline.com

Predicate<T>:

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



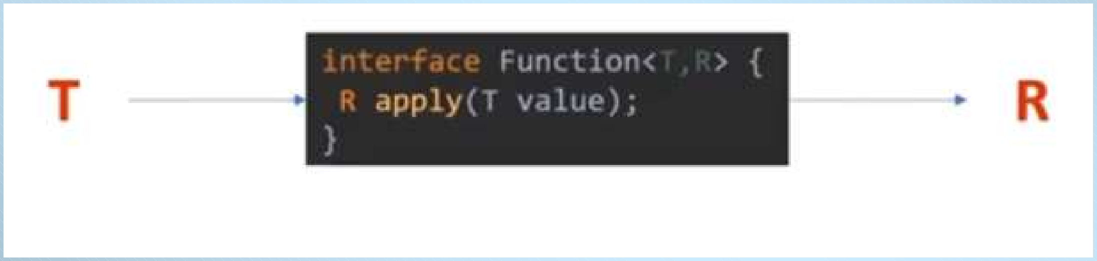
```
interface Predicate<T> {
 boolean test(T value);
}
```

Example:

```
Predicate<Integer> p = (i) -> (i > -10) && (i < 10);
System.out.println(p.test(9));
```

Function<T,R>:

```
interface Function<T,R> {
        R apply(T t);
}
```



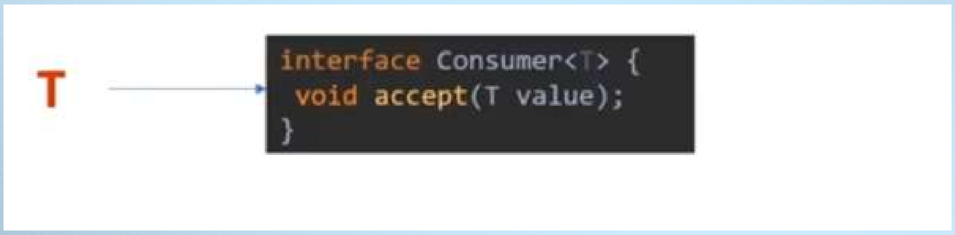
```
interface Function<T,R> {
 R apply(T value);
}
```

Example:

```
Function<String, Integer> f = s -> s.length();
System.out.println(f.apply(“Hello!"));
```

Consumer<T>:

```
interface Consumer<T> {
        void accept(T t);
}
```



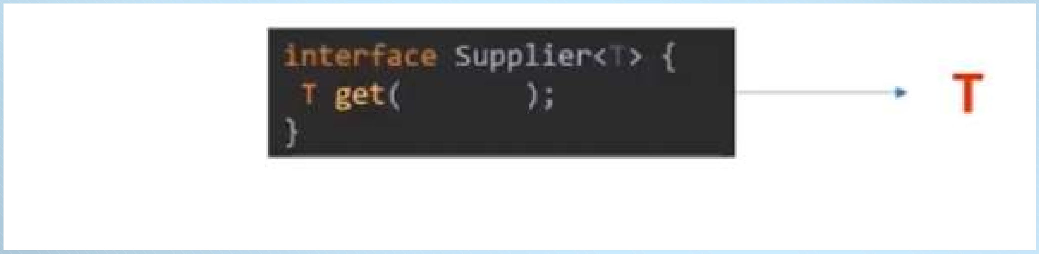
```
interface Consumer<T> {
 void accept(T value);
}
```

Example:

```
Consumer<String> c = s -> System.out.println(s);
c.accept("I only consume data!");
```

Supplier<R>:

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



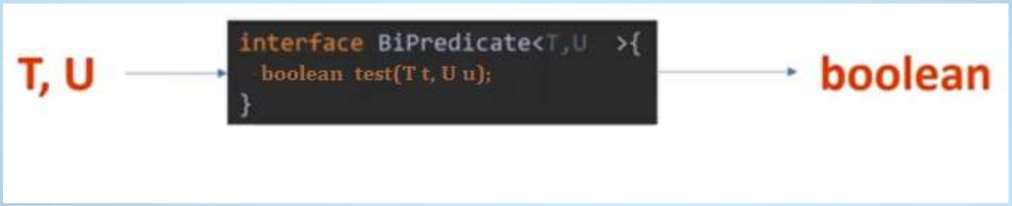
```
interface Supplier<T> {
 T get(        );
}
```

Example:

```
Supplier<Integer> s = () -> new Random().nextInt(10);
System.out.println(s.get());
```

BiPredicate<T,U>:

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



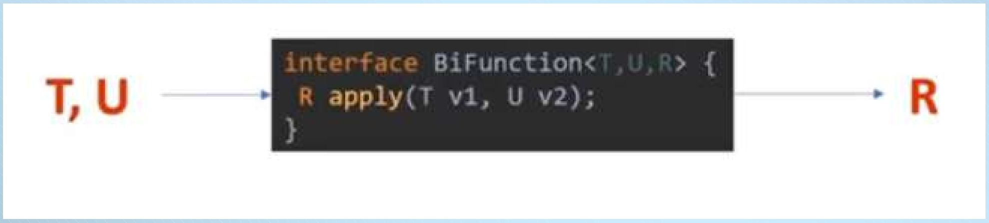
```
interface BiPredicate<T,U >{
  boolean test(T t, U u);
}
```

Example:

```
BiPredicate<Integer,Integer> bp = (i, j) -> (i + j) % 2 == 0;
System.out.println(bp.test(5,6));
```

BiFunction<T,U,R>:

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



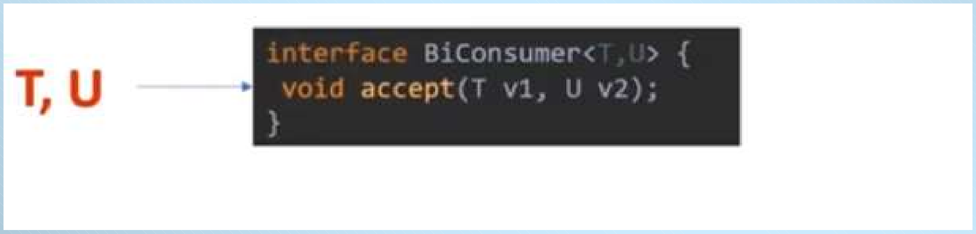
```
interface BiFunction<T,U,R> {
 R apply(T v1, U v2);
}
```

Example:

```
BiFunction<String, String, Integer> bf = (i,j) -> i.length()+j.length();
System.out.println(bf.apply("Hello","World!"));
```

BiConsumer<T,U>:

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

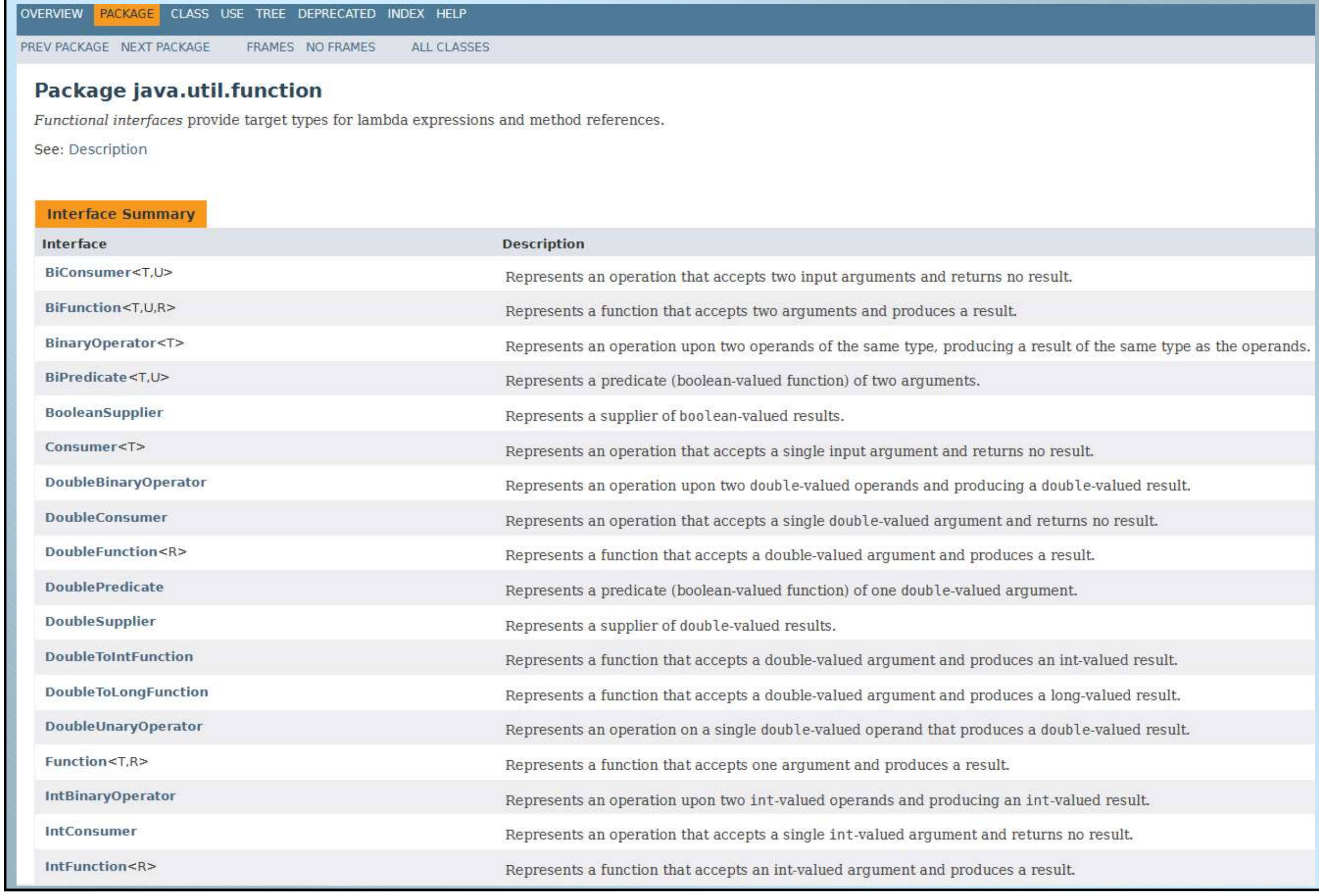


```
interface BiConsumer<T,U> {
 void accept(T v1, U v2);
}
```

Example:

```
BiConsumer<String,String> bc = (s1,s2) -> System.out.println(s1+" "+s2);
bc.accept("Hello", "World");
```

Predefined Functional Interfaces


OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP
PREV PACKAGE NEXT PACKAGE FRAMES NO FRAMES ALL CLASSES

Package java.util.function

Functional interfaces provide target types for lambda expressions and method references.

See: Description

Interface Summary

Interface	Description
BiConsumer<T,U>	Represents an operation that accepts two input arguments and returns no result.
BiFunction<T,U,R>	Represents a function that accepts two arguments and produces a result.
BinaryOperator<T>	Represents an operation upon two operands of the same type, producing a result of the same type as the operands.
BiPredicate<T,U>	Represents a predicate (boolean-valued function) of two arguments.
BooleanSupplier	Represents a supplier of boolean-valued results.
Consumer<T>	Represents an operation that accepts a single input argument and returns no result.
DoubleBinaryOperator	Represents an operation upon two double-valued operands and producing a double-valued result.
DoubleConsumer	Represents an operation that accepts a single double-valued argument and returns no result.
DoubleFunction<R>	Represents a function that accepts a double-valued argument and produces a result.
DoublePredicate	Represents a predicate (boolean-valued function) of one double-valued argument.
DoubleSupplier	Represents a supplier of double-valued results.
DoubleToIntFunction	Represents a function that accepts a double-valued argument and produces an int-valued result.
DoubleToLongFunction	Represents a function that accepts a double-valued argument and produces a long-valued result.
DoubleUnaryOperator	Represents an operation on a single double-valued operand that produces a double-valued result.
Function<T,R>	Represents a function that accepts one argument and produces a result.
IntBinaryOperator	Represents an operation upon two int-valued operands and producing an int-valued result.
IntConsumer	Represents an operation that accepts a single int-valued argument and returns no result.
IntFunction<R>	Represents a function that accepts an int-valued argument and produces a result.

Method References

- A shorthand way of writing lambda expressions that refer to an existing method or constructor
- Makes lambda expression simpler & more concise
- General Syntax:

  class/object :: method name

- Can be used to refer to:
  - Static method
  - Instance method of an object
  - Constructor

Reference to a static method:

Syntax:

class :: static method

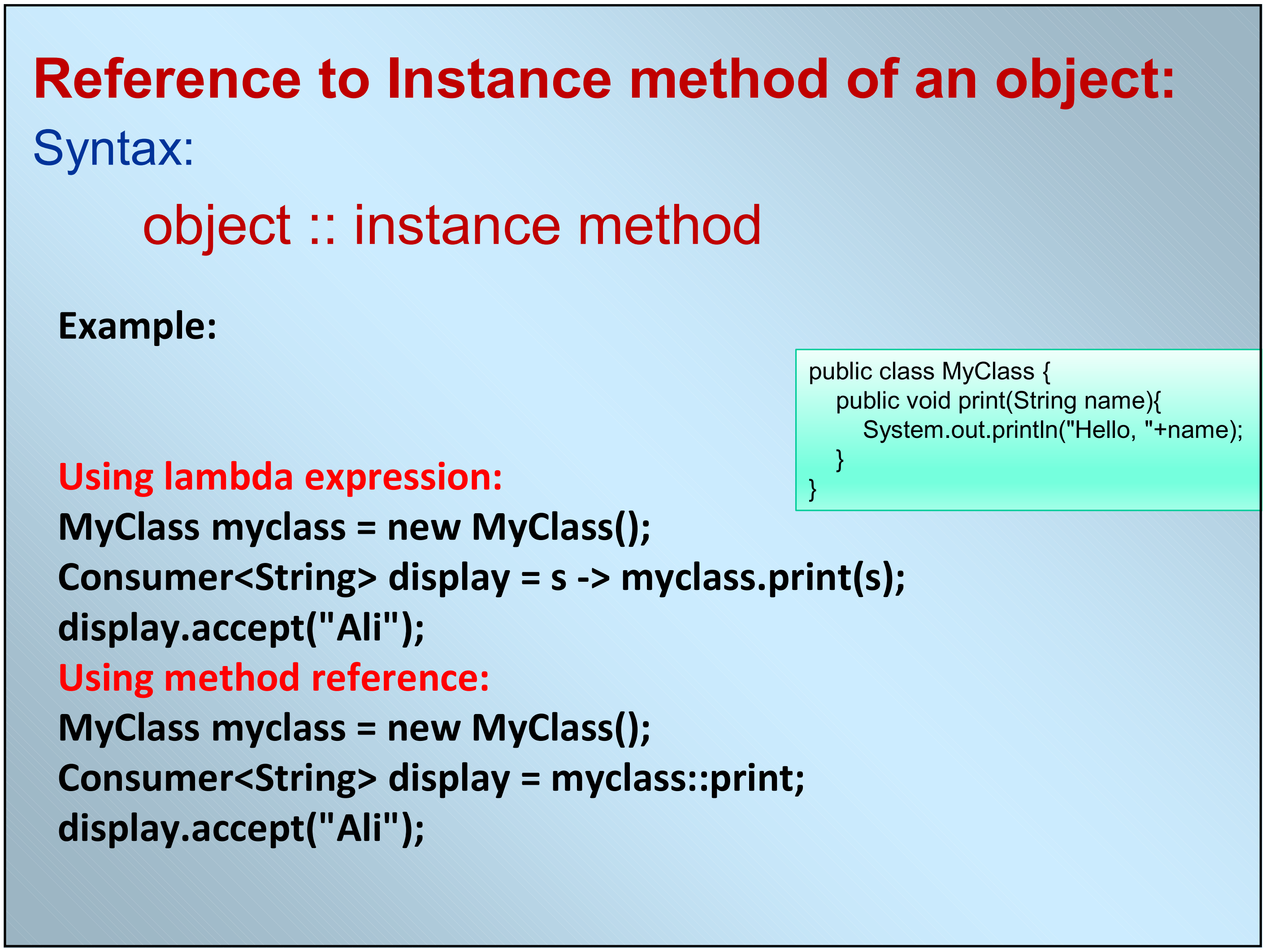
Example:

Using lambda expression:

```
Function<Integer,Double> squareRoot = n -> Math.sqrt(n);
System.out.println("Sqrt of 4 is "+squareRoot.apply(4));
```

Using method reference:

```
Function<Integer,Double> squareRoot = Math::sqrt;
System.out.println("Sqrt of 4 is "+squareRoot.apply(4));
```



Reference to Instance method of an object:

Syntax:

object :: instance method

Example:

```
public class MyClass {
    public void print(String name){
        System.out.println("Hello, "+name);
    }
}
```

Using lambda expression:

```
MyClass myclass = new MyClass();
Consumer<String> display = s -> myclass.print(s);
display.accept("Ali");
```

Using method reference:

```
MyClass myclass = new MyClass();
Consumer<String> display = myclass::print;
display.accept("Ali");
```

Reference to a Constructor:

Syntax:

class :: new

Example:

Using lambda expression:

```
BiFunction<Integer,String,Student> c = (n,m)->new Student(n,m);
Student s1 = c.apply(12345,"Ali");
```

Using method reference:

```
BiFunction<Integer,String,Student> c = Student::new;
Student s1 = c.apply(12345,"Ali");
```