The Function interface

On this slide, I'm showing four of the most common interfaces in this category.

Each has a return type, shown here as either T, or R, which stands for result, meaning a result is expected for any of these.

In addition to Function and BiFunction, there is also UnaryOperator and BinaryOperator.

You can think of the UnaryOperator as a Function Interface, but where the argument type is the same as the result type.

Interface Name	Method Signature	Interface Name	Method Signature
`Function<T,R>`	`R apply(T t)`	`UnaryOperator<T>`	`T apply(T t)`
`BiFunction<T,U,R>`	`R apply(T t, U u)`	`BinaryOperator<T>`	`T apply(T t1, T t2)`

The Function interface

The Binary Operator is a BiFunction interface, where both arguments have the same type, as does the result, which is why the result is shown as T, and not R.

This reminds us that the result will be the same type as the arguments to the methods.

I've also included the type parameters with each interface name on this slide, because I wanted you to see that the result, for a Function or BiFunction, is declared as the last type argument.

For UnaryOperator and BinaryOperator, there is only one type argument declared, because the types of the arguments and results, will be the same.

Interface Name	Method Signature	Interface Name	Method Signature
`Function<T,R>`	`R apply(T t)`	`UnaryOperator<T>`	`T apply(T t)`
`BiFunction<T,U,R>`	`R apply(T t, U u)`	`BinaryOperator<T>`	`T apply(T t1, T t2)`

A Function Interface Lambda Expression Example

On this slide, I'm showing an example of a lambda expression which targets a Function interface.

This lambda expression takes a String, s, and splits that String on commas, returning an array of String.

In this case, the argument type, T, is a String, and the result, R, is an array of String.

To demonstrate how to declare a variable of this type, I'm showing a variable declaration as well, for this specific example.

Example Lambda Expression for Function	**Function Method**	**Variable Declaration for this example**
`s -> s.split(",");`	`R apply(T t)`	`Function<String, String[]> f1;`

The Supplier Interface

The supplier interface takes no arguments but returns an instance of some type, T.

Interface Name	**Method Signature**
Supplier	`T get()`

You can think of this as kind of like a factory method code.

It will produce an instance of some object.

However, this doesn't have to be a new or distinct result returned.

A Supplier Lambda Expression Example

In the example I'm showing you on this slide, I'm using the Random class to generate a random Integer.

This method takes no argument, but lambda expressions can use final or effectively final variables in their expressions, which I'm demonstrating here.

The variable random is an example of a variable, from the enclosing code.

Example Lambda Expression for Consumer
`() -> random.nextInt(1, 100)`

Valid Lambda Declarations for different number of arguments

This slide shows the many varieties of declaring a parameter type in a lambda expression.

Parentheses are required in all but the one case, where the functional method has a single argument, and you don't specify a type, or use var.

Arguments in Functional Method	**Valid lambda syntax**
None	`() -> statement`

One	`s -> statement`
	`(s)  -> statement`
	`(var s) -> statement`
	`(String s) -> statement`

Two
• When using var, all arguments must use var.
• When specifying explicit types, all arguments must specify explicit types.
	`(s, t) -> statement`
	
	`(var s, var t) -> statement`
	
	`(String s, List t) -> statement`

Valid Lambda Declarations for different number of arguments

When using var as the type, every argument must use var.

When specifying explicit types, every argument must include a specific type.

Arguments in Functional Method	**Valid lambda syntax**
None	`() -> statement`

One	`s -> statement`
	`(s)  -> statement`
	`(var s) -> statement`
	`(String s) -> statement`

Two
• When using var, all arguments must use var.
• When specifying explicit types, all arguments must specify explicit types.
	`(s, t) -> statement`
	
	`(var s, var t) -> statement`
	
	`(String s, List t) -> statement`