# Lambda Expressions

**Week 4 Presentation 1**

Lambda is a letter in the Greek alphabet that was used by the mathematician Alonzo Church in his study of computable functions. His lambda notation makes it possible to define a function without giving it a name.

For example, you might think that the notation $x^2$ is a perfectly good way of representing a function that squares a number, but in fact, it's an expression that represents the result of squaring $x$, which leaves open the question of what $x$ represents.

A METHOD OR FUNCTION IS JUST binary numbers (representing instructions) stored somewhere in the computer's memory. Considered as a long string of zeros and ones, a subroutine doesn't seem all that different from a data value such as, for example, as an integer, a string, or an array, which is also represented as a string of zeros and ones in memory. We are used to thinking of subroutines and data as very different things, but inside the computer, a subroutine is just another kind of data. Some programming languages make it possible to work with a subroutine as a kind of data value. In Java 8, that ability was added to Java in the form of something called lambda expressions.

We can define a function with *x* as a dummy parameter:

```
static double square( double x ) {
        return x*x;
}
```

but to do that, we had to name the function *square*, and that function becomes a permanent part of the program—which is overkill if we just want to use the function once.

$$\lambda x . x^2$$

Alonzo Church introduced the notation *lambda*$(x) . x^2$ to represent "the function of $x$ that is given by $x^2$"

# Java lambda expressions

```
x -> x*x
```

$\lambda x . x^2$

Means: "the function of $x$ that is given by $x^2$"

# Java lambda expressions

```
x -> x*x
```

The operator –> is what makes this a lambda expression

# Java lambda expressions

`x -> x*x`

`sqrt = x -> x*x`

~~`sqrt(42)`~~

The operator –> is what makes this a lambda expression

```java
// to sort a list of apples in inventory based on their weight

// Before:
Collections.sort( inventory, new Comparator<Apple>() {
    public int compare (Apple a1, Apple a2) {
        return a1.getWeight().compareTo( a2.getWeight() );
    }
});

// it is equivalent to
Comparator<Apple> byWeight = new Comparator<Apple>() {
    public int compare (Apple a1, Apple a2) {
        return a1.getWeight().compareTo( a2.getWeight() );
    }
};
Collections.sort( inventory, byWeight );

// After (with lambda expressions):
Comparator<Apple> byWeight =
    (Apple a1, Apple a2) -> a1.getWeight().compareTo( a2.getWeight() );
Collections.sort( inventory, byWeight );

// or
Collections.sort( inventory,
    (Apple a1, Apple a2) -> a1.getWeight().compareTo( a2.getWeight() )
);
```

# Functional Interfaces

To know how a subroutine can be legally used, you need to know its name, how many parameters it requires, their types, and the return type of the subroutine.

A functional interface specifies this information about one subroutine. A functional interface is similar to a class, and it can be defined in a .java file, just like a class. However, its content is just a specification for a single subroutine.

A functional interface is an interface that contains only one abstract method.

They can have only one functionality to exhibit. From Java 8 onwards, lambda expressions can be used to represent the instance of a functional interface.

**Java comes with a number of built in functional interfaces that are suitable for many situations…**

```
public interface FunctionR2R {
    double valueAt( double x );
}
```

This code would be in a file named *FunctionR2R.java*. It specifies a function named *valueAt* with one parameter of type double and a return type of double.

```
public interface ArrayProcessor {
    void process( String[] array, int count );
}
```

This one is called ArrayProcessor and specifies a method process that takes a String[] array and an int

*Some different ways of writing lambda expressions*

```java
Runnable noArguments = () -> System.out.println("Hello World");

ActionListener oneArgument = event -> System.out.println("button clicked");

Runnable multiStatement = () -> {
   System.out.print("Hello");
   System.out.println(" World");
};

BinaryOperator<Long> add = (x, y) -> x + y;

BinaryOperator<Long> addExplicit = (Long x, Long y) -> x + y;
```

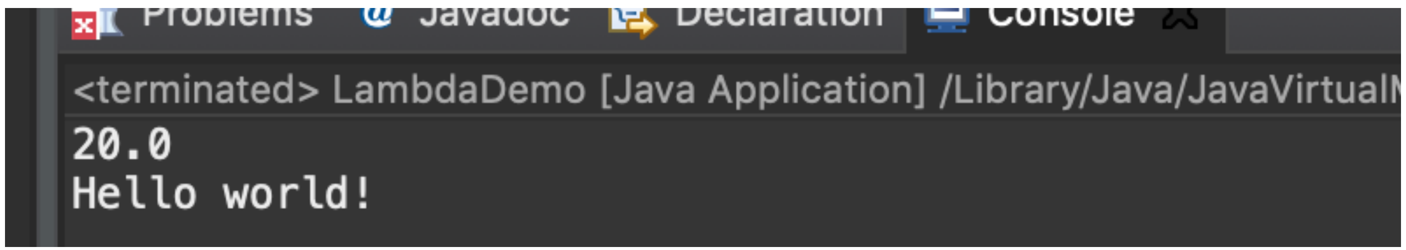
# Built in Functional Interfaces

The name of a functional interface is a **type**, just as *String* and double are types.

Here are are some examples

```
// interface binary operator has two arguments and
// requires to implement a method .apply()
BinaryOperator<Double> add = (x,y) -> {return x+y;};
System.out.println(add.apply(10.0, 10.0));

// another useful interface is the runnable interface which
// takes no arguments but runs the statements provided
// it has the method to implement .run()
Runnable noArgs = () -> System.out.println("Hello world!");
noArgs.run()
```



`java.util.function.Function`

## Interface Function<T,R>

```
// the interface
public interface Function<T,R> {
        public <R> apply(T parameter);
}

// We can implement with a class
public class AddOne implements Function<Long, Long> {
@Override  // more on annotations later
public Long apply(Long aLong) {
        return aLong + 1;
        }
}

// Or with a lambda expression
Function<Long, Long> adder = (value) -> value + 1; Long
resultLambda = adder.apply((long) 8);
System.out.println("resultLambda = " + resultLambda);
```

The `Function` interface represents a function (method) that takes a single parameter of type T and returns a single value of type R.

**Type Parameters:**

T - the type of the input to the function

R - the type of the result of the function

```
// in which case to use instantiate the
// interface (and return 2):
LambdaDemo x = new LambdaDemo();
Function<Long, Long> adder = x.new
AddOne();
Long result = adder.apply((long) 1);
System.out.println("result = " + result);
```

*Table 2-1. Important functional interfaces in Java*

| Interface name | Arguments | Returns | Example |
|---|---|---|---|
| Predicate<T> | T | boolean | Has this album been released yet? |
| Consumer<T> | T | void | Printing out a value |
| Function<T,R> | T | R | Get the name from an Artist object |
| Supplier<T> | None | T | A factory method |
| UnaryOperator<T> | T | T | Logical not (!) |
| BinaryOperator<T> | (T, T) | T | Multiplying two numbers (*) |

*Type inference*

```
Predicate<Integer> atLeast5 = x -> x > 5;
```

*The predicate interface in code, generating a boolean from an Object*

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


*A more complex type inference example*

```
BinaryOperator<Long> addLongs = (x, y) -> x + y;
```

```
BinaryOperator<Double> add = (x,y) -> x+y;
```

*Code doesn't compile due to missing generics*

```
BinaryOperator add = (x, y) -> x + y;
```

This code results in the following error message:

```
Operator '&#x002B;' cannot be applied to java.lang.Object, java.lang.Object.
```

# Why use lambda expressions

- Mainly in cases where we are interested in using the behaviors (i.e. methods) of an interface rather than the rest of a class

- For example we have seen the comparator object in the last class

- With a "Comparator" object we usually just want the compareTo() method…

# Making Static Methods into Lambdas

Lambdas Made from Static Methods

```
IntFunction<String> intToString = Integer::toString;
ToIntFunction<String> parseInt = Integer::valueOf;
```

# Making Constructors into Lambdas

Lambdas Made from a Constructor

```
Function<String,BigInteger> newBigInt = BigInteger::new;
```

# Method Reference Operator ::

## Making Static Methods into Lambdas

Lambdas Made from Static Methods

```
IntFunction<String> intToString = Integer::toString;
ToIntFunction<String> parseInt = Integer::valueOf;
```

## Making Constructors into Lambdas

Lambdas Made from a Constructor

```
Function<String,BigInteger> newBigInt = BigInteger::new;
```