Föreläsning 4 Funktionell programmering, lambdauttryck

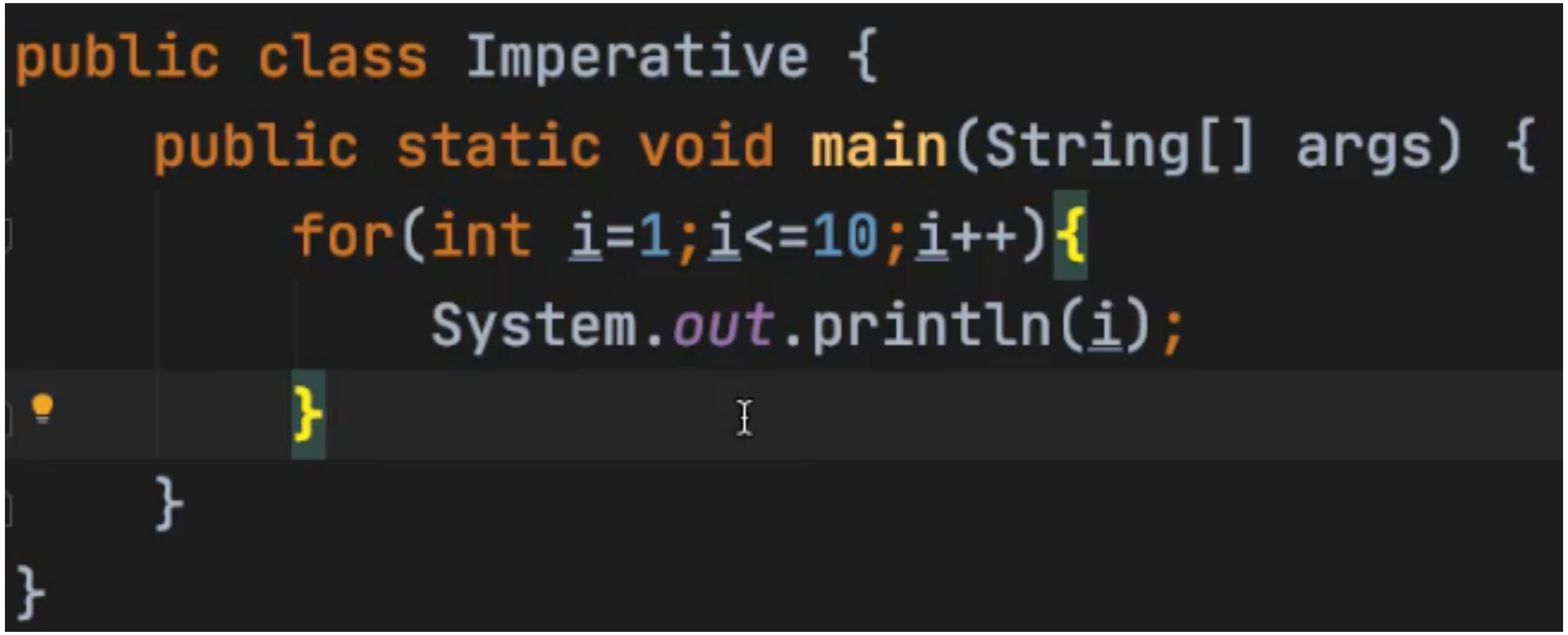
Avancerad Javaprogrammering

Utbildare: Mikael Olsson

Funktionell programmering

- Programmeringssätt, har inte med språk att göra (lite som objektorientering)
- Kodblock kan existera oberoende av klasser eller objekt
- En funktion kan tilldelas en variabel
- En funktion kan skickas som parameter
- En funktion kan returneras från en annan funktion
- Vanligt i JavaScript, allt vanligare i Java

Imperativ



```java
public class Imperative {
    public static void main(String[] args) {
        for(int i=1;i<=10;i++){
            System.out.println(i);
        }
    }
}
```

Stream

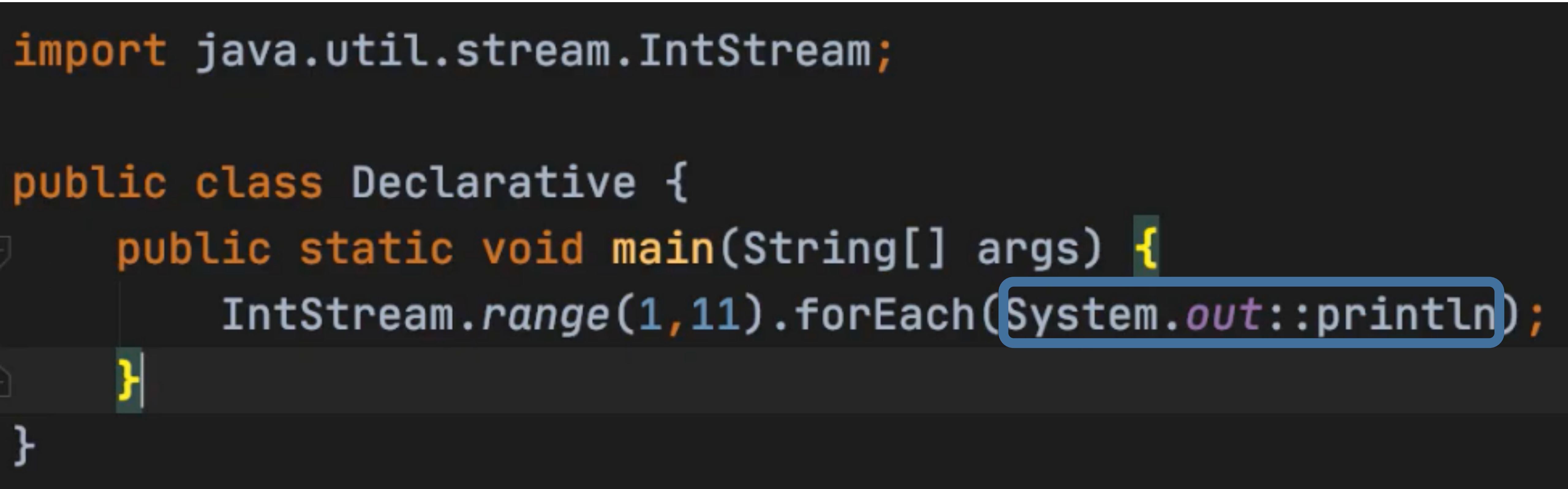
- ” A sequence of elements supporting sequential and parallel aggregate operations.”
- https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html

IntStream

- ”A sequence of primitive int-valued elements supporting sequential and parallel aggregate operations. This is the int primitive specialization of Stream.”
- https://docs.oracle.com/javase/8/docs/api/java/util/stream/IntStream.html

Deklarativ

- :: betyder att vi kommer att anropa metoden med en parameter, men att vi inte namnger parametern.



```java
import java.util.stream.IntStream;

public class Declarative {
    public static void main(String[] args) {
        IntStream.range(1,11).forEach(System.out::println);
    }
}
```

Deklarativ

- :: kallas ”methodreferens” och det är ”syntaktiskt socker”, ett enklare sätt att skriva ett annat uttryck.
  - `numbers.forEach(System.out::println);`
- I detta fall står detta för samma uttryck som:
  - `numbers.forEach(x -> System.out.println(x));`
- Vad innebär då detta uttryck?

Lambdauttryck

- Uttryck som skapar funktioner.

-
  ```
  function (a, b) {
    return a + b
  };
  ```
- // Lambda:
- `a, b -> a + b;`
- // Eller
- `(a, b) -> { return a + b };`
- // Om vi bara har en parameter kan vi skippa parenteserna
- `a -> a;`

Lambda-funktioner

- Introduced in Java SE 8, lambda expressions are a way to create single-method classes in your code in a much less cumbersome manner than anonymous classes/local classes
  - Like with anonymous classes, these work best when you only need the method in one place
  - One way to think about lambda expressions is as anonymous methods
- A great time to use for lambda expressions are functional interfaces – interfaces with one method (e.g. many of the event listeners)
  - An interface with static and default methods is still considered a functional interface if it only has one abstract method (only one REQUIRED method override)
- In addition to “regular” functional interfaces that perform specific functions, Java now includes a wide variety of general functional interfaces which are ideal for lambda expressions

STANDARD FUNCTIONAL INTERFACES

- Found in `java.util.function` – a few examples
  - `Predicate<T>` - contains a predicate function (boolean function) with one argument
  - `Function<T, R>` - represents a function that accepts an argument of type `T` and produces a result of type `R`
  - `IntConsumer` – a function that accepts a single integer argument and returns no result
  - `LongSupplier` – a function that accepts no arguments and returns a long
  - `BinaryOperator<T>` - a function that accepts two arguments of type `T` and produces a result of the same type
  - `BiFunction<T, U, R>` - a function that accepts two arguments (of type `T` and `U`) and returns a result of type `R`

SYNTAX OF LAMBDA EXPRESSIONS

- The basic format of a lambda expression is:

```
parameters -> body
```

- The parameters are a comma-separated list enclosed in parentheses. However, if there is only one parameter you can leave off the parentheses.
  - You also do not have to specify the types of the parameters (sometimes it is necessary if the compiler cannot figure out the type)
  - You can also have lambdas with no parameters
  - So, you may have lambda expressions which look like the following:

```
p -> body
() -> body
(a, b) -> body
(Class c) -> body
(stringVariable, array, intVar, otherVar) -> body
```

- Between the parameters and the body you have the arrow token, which consists of a dash followed by the greater than sign: ->

SYNTAX OF LAMBDA EXPRESSIONS

- The body of a lambda expression contains either a single expression or statement block
- For a single expression that is not a return statement, you just place the expression to the right of the arrow token and the lambda expression evaluates the expression and returns its value (or, if it is a void expression, returns nothing)
  - So, for example, this statement returns true if the `String s` is all lowercase and shorter than 200 characters:

    ```
    s -> s.equals(s.toLowerCase()) && s.length() < 200
    ```

    and this statement would return nothing, but would print out the `String s`

    ```
    s -> System.out.println(s)
    ```
  - Notice that there is no semi-colon required when the lambda contains a single expression only

SYNTAX OF LAMBDA EXPRESSIONS

- If the body consists of a statement block rather than a single expression, you must enclose it in curly braces {}, just like any other statement block
  - These can have return statements or not as determined by the situation
- So, the following is a valid lambda expression that takes 2 `int`s and returns an `int`:

```
(x, y) -> {
     System.out.println("x = " + x);
     System.out.println("y = " + y);
     z = x + y;
     System.out.println("z = " + z);
     return z;
}
```

LAMBDAS VS ANONYMOUS CLASSES

- As you can see from the syntax, lambda expressions are functionally similar to an anonymous class
  - In fact, every time you can use a lambda expression, you COULD use an anonymous class
- The main benefit of a lambda expression is the concise syntax, as you can leave a lot out that would be required for an anonymous class
  - Parameter types, new operator, interface type, and method name are all found in every anonymous class but not lambda expressions
- There are some drawbacks to using a lambda expression, however
  - Anonymous classes work with any class or interface (no matter how many methods), but lambda expressions can only have exactly one method
  - Anonymous classes can contain member data, lambda expressions cannot
  - Anonymous classes can have their own (non-inherited) methods, lambda expressions cannot

TARGET TYPING

- Lambda expressions are typed based on a combination of their parameters and return type, as well as what interface type the compiler is expected
  - The type the compiler is expecting is called the *target type*
  - This means that the same lambda expression could be used in multiple places, but have different types
- You can only use lambda expressions where the compiler can determine the target type:
  - Variable declarations
  - Assignments
  - Return statements
  - Array initializers
  - Method arguments
  - Lambda expression bodies
  - Conditional expressions
  - Casting expressions

TARGET TYPING, EXAMPLE

- As an example, suppose you have the following methods:

```
public void doSomething1(List<Integer> list, IntConsumer consumer)

public void doSomething2(List<Integer> list, Consumer<Integer> consumer)

public void doSomething3(List<String> list, Consumer<String> consumer)
```

- Because the second parameter in all three methods consists of a functional interface that has a single method that takes some object and is of void return type, you can actually use the same lambda expression for all three parameters and because of target typing, they end up having different types.
- In this call, the second parameter is of type `IntegerConsumer` and the parameter `x` is an `int`:

```
doSomething1(list, x -> System.out.println(x));
```

TARGET TYPING, EXAMPLE

- In this call, the second parameter is of type `Consumer<Integer>` and the parameter `x` is an `Integer`:

```
doSomething2(list, x -> System.out.println(x));
```

- In this call, the second parameter is of type `Consumer<String>` and the parameter `x` is a `String`:

```
doSomething3(list, x -> System.out.println(x));
```

LOCAL VARIABLES/SCOPING ISSUES

- Like local classes, a lambda expression can only access local variables and parameters of the block enclosing it that are final or effectively final
- However, unlike local classes, the lambda expression does not create a new level of scope.
- As an example, this will cause an error (a has already been declared):

```
int a = 5;

int b = 17;

IntConsumer consumer = c -> {

    int a = c;

    System.out.println(a);

}
```

LOCAL VARIABLES/SCOPING ISSUES

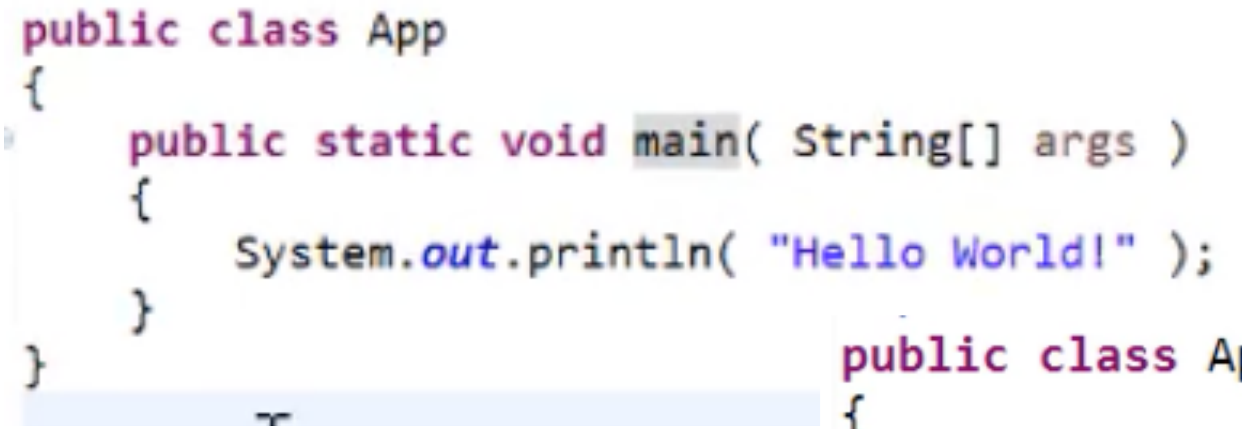
- This will also cause the same error:

```
int a = 5;
int b = 17;
//Note: It is the parameter ‘a’ causing the issue here
IntConsumer consumer = a -> {
    int d = a;
    System.out.println(d)
}
```

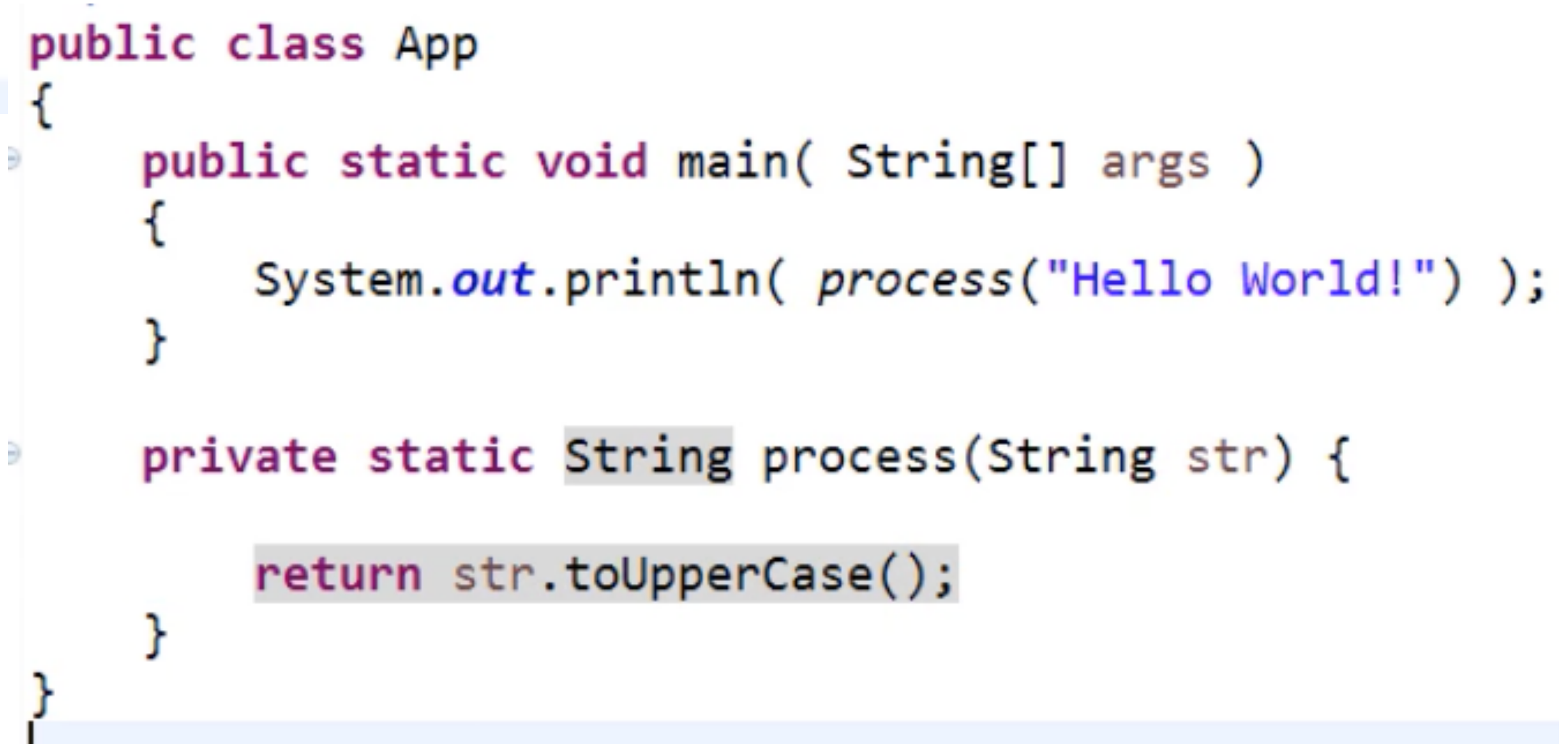
EXAMPLES

- This example contains a few simple lambda expressions and a method that takes one as a parameter
- This example shows different predicates being passed into the PrintSome function

Hur skickar vi kod som parametrar?

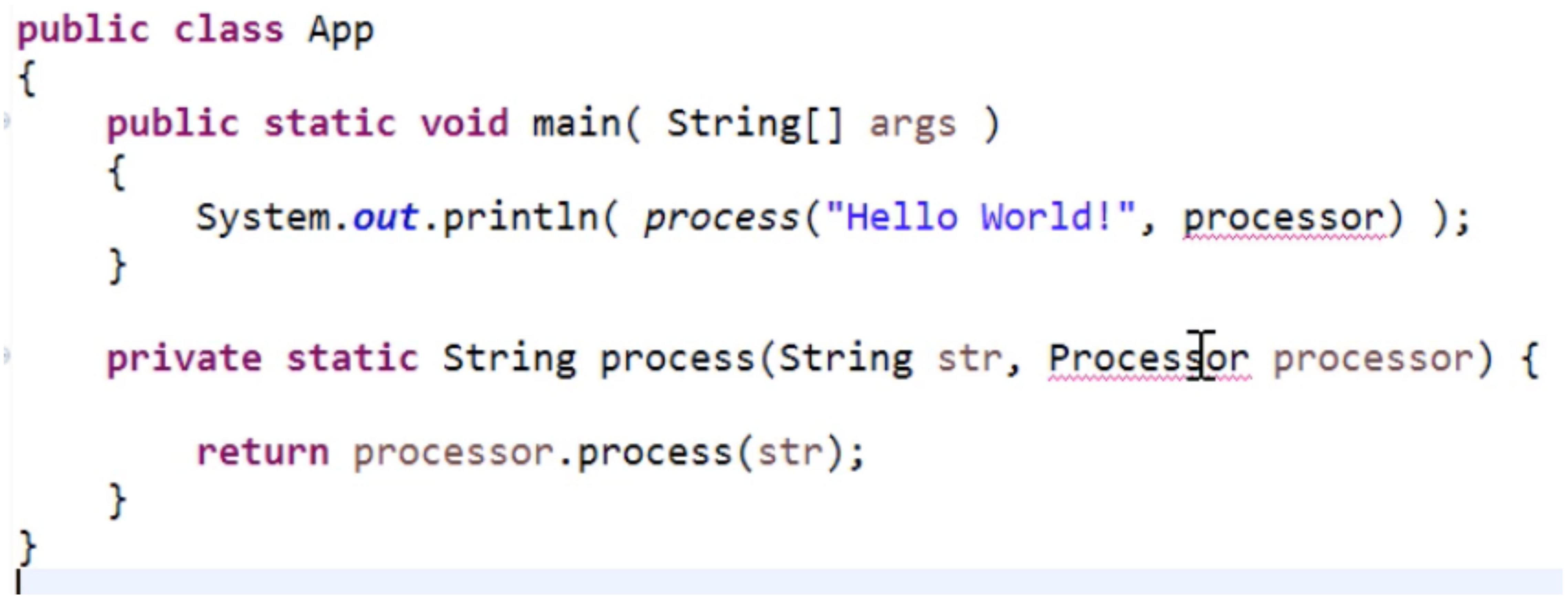


```
public class App
{
    public static void main( String[] args )
    {
        System.out.println( "Hello World!" );
    }
}
```



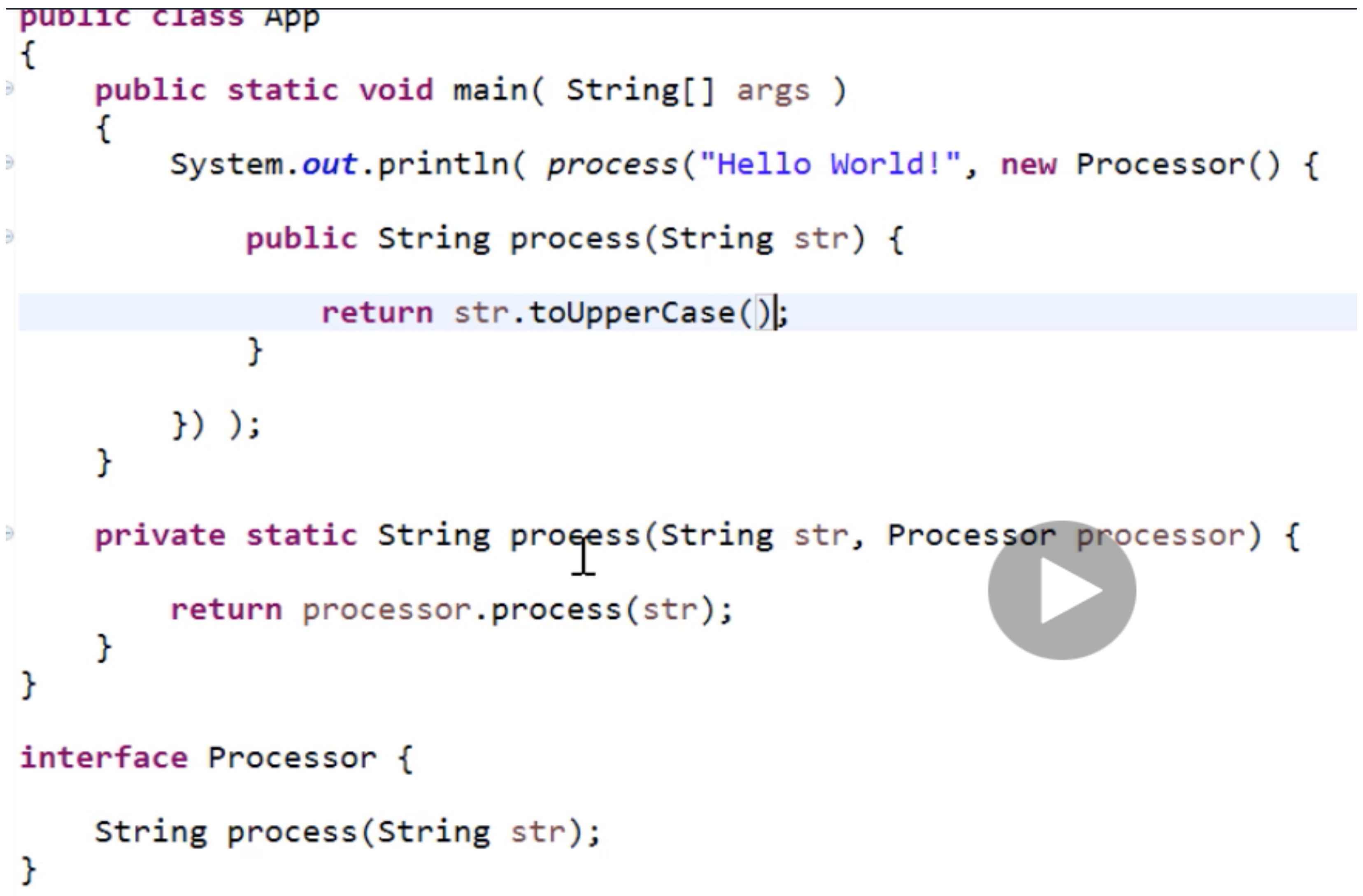
```
public class App
{
    public static void main( String[] args )
    {
        System.out.println( process("Hello World!") );
    }

    private static String process(String str) {

        return str.toUpperCase();
    }
}
```

Hur skickar vi kod som parametrar?



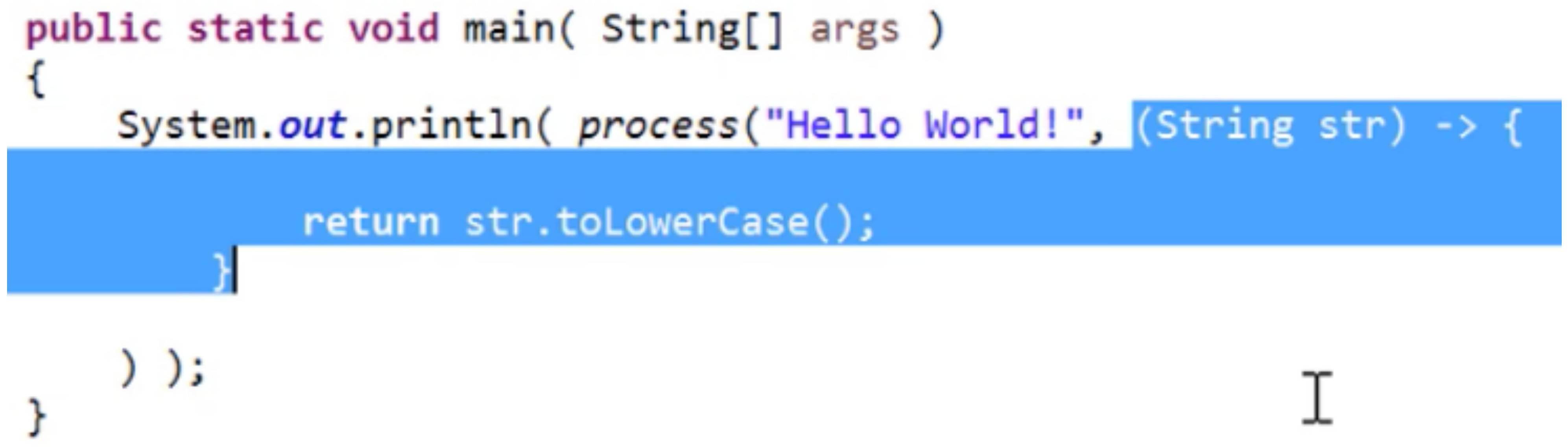
```
public class App
{
    public static void main( String[] args )
    {
        System.out.println( process("Hello World!", processor) );
    }

    private static String process(String str, Processor processor) {

        return processor.process(str);
    }
}
```

Hur skickar vi kod som parametrar?

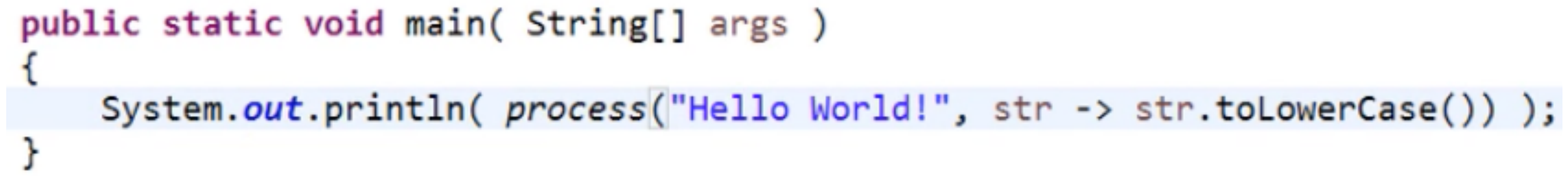


```
public class App
{
    public static void main( String[] args )
    {
        System.out.println( process("Hello World!", new Processor() {

            public String process(String str) {

                return str.toUpperCase();
            }

        }) );
    }

    private static String process(String str, Processor processor) {

        return processor.process(str);
    }
}

interface Processor {

    String process(String str);
}
```

Lambda-uttryck

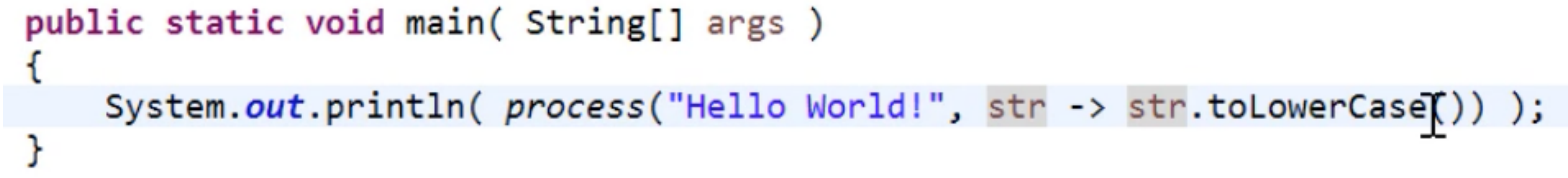


```
public static void main( String[] args )
{
    System.out.println( process("Hello World!", (String str) -> {

            return str.toLowerCase();
        }

    ) );
}
```

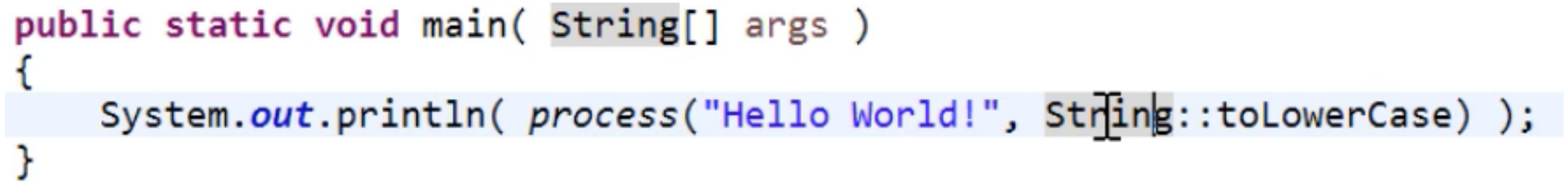


```
public static void main( String[] args )
{
    System.out.println( process("Hello World!", str -> str.toLowerCase()) );
}
```

Metodreferens



```
public static void main( String[] args )
{
    System.out.println( process("Hello World!", str -> str.toLowerCase()) );
}
```



```
public static void main( String[] args )
{
    System.out.println( process("Hello World!", String::toLowerCase) );
}
```

```
String::str.substring(i)
```

```
str, i -> str.substring(i)
```

Dagens uppgifter

- http://www.java-programming.info/tutorial/pdf/java/exercises/exercises-lambdas-1.pdf
- Extra:
  - https://code-exercises.com/programming/easy/20/average-value-java-8-lambdas-and-streams
  - https://code-exercises.com/programming/easy/21/convert-to-upper-case-java-8-lambdas-and-streams