LEARN PROGRAMMING IN JAVA™

Learning OO Java™ as a Beginner – Functional Programming

COURSE AGENDA

- What is an Object
- The Basics
- Operators
- Control Flow
- Implementation Hiding
- Reuse
- Interfaces
- Polymorphism
- Collections
- Functional Programming
- Streams
- Exceptions
- Enums

INSTRUCTOR – HUGO SCAVINO

- 30 Years of IT Experience
- Using Java since the beta
- Taught Java and OOP in USA, UK, and France to Fortune 500
- Senior Software Architect with
  - DTCC
  - Penske
  - HSBC
  - Government Agencies

https://www.linkedin.com/in/hugoscavino/

REQUIRED SOFTWARE

- JDK™
  - JDK 8 or Newer (Open Source, Amazon, or Oracle)
  - The beginner course focuses on core language constructs; feel free to use 11, 17, 21, etc.

- IDE
  - IntelliJ Community Edition (Free) or Enterprise (Paid)
  - While you can use any IDE with Java, the course and labs are explicitly made with IntelliJ in mind

RECOMMENDED TEXTS

- Java 5 Book
  - Thinking in Java – 4th Edition - Free – PDF - GitHub
  - Thinking in Java – 4th Edition – Hard Cover - Amazon

- Recommended Java 8 Book
  - Bruce Eckel on Java 8
    - Contains references to newer Java syntax
- #1 Best Seller in Beginner's Guide to Java Programming
  - Head First Java: A Brain-Friendly Guide 3rd Edition

TOPIC DESCRIPTION

Functional Programming: (Java 8 above) Functional programming (FP) in Java refers to a programming paradigm treating computation as the evaluation of mathematical functions and avoids changing state and mutable data. Java introduced FP through lambda expressions, the Stream API, and other constructs.

Key Characteristics:

- First Class Functions
- Immutability
- Declarative Programming
- Pure Functions
- Higher-Order Functions

FUNCTIONAL PROGRAMMING ADVANTAGES

- Conciseness: Code is cleaner
- Parallelism: `Easier to write parallel code with Stream API`
- Reduced Side Effects: Pure functions make code more predictable and easier to debug
- Reusability: Functional constructs like the lambdas and stream promote reusable logic

Although Java is not a purely functional language (mutability and state changes are supported), these functional programming features allow developers to write cleaner, more efficient, and expressive code.

KEY ADVANTAGES

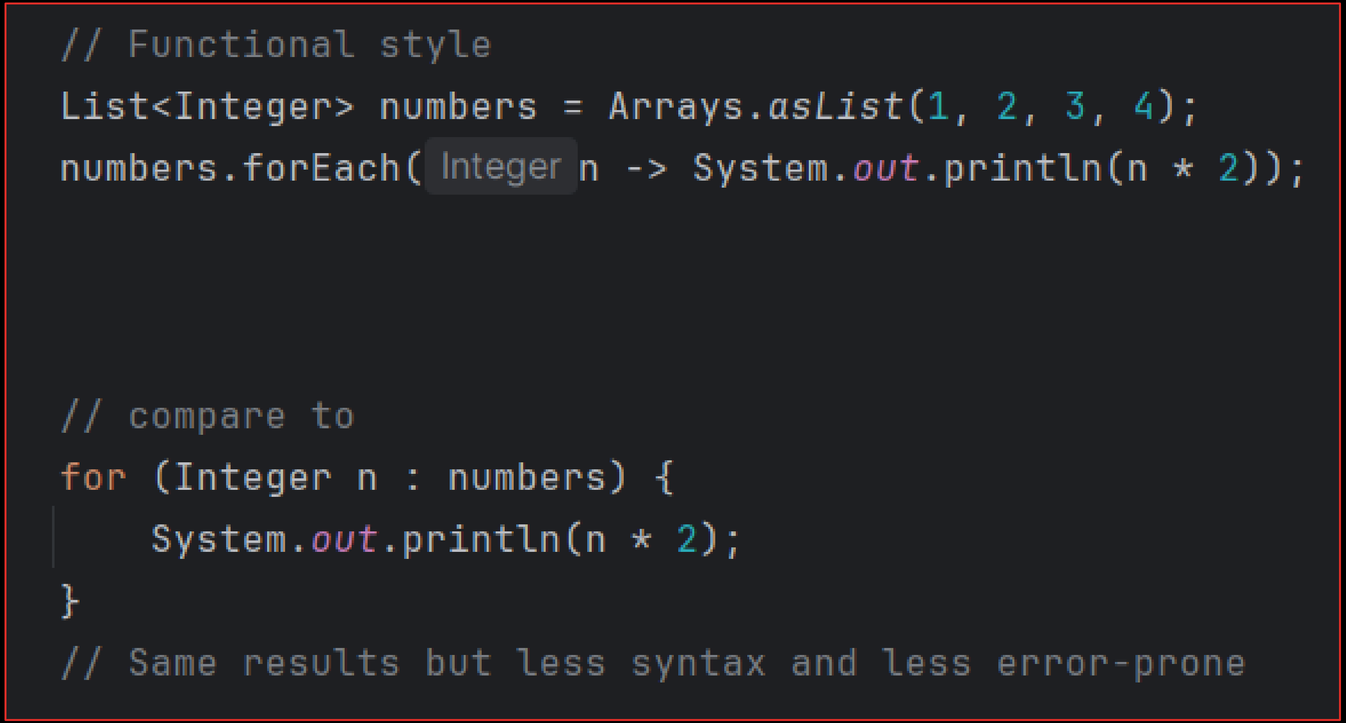
Functional Programming: Simplified and expressive operations using lambda expressions and streams.
Immutability: Simplified creation of immutable collections.
Efficiency: Improved performance and cleaner code through new utilities.
Parallelism: Easy parallel processing with `parallelStream`().

LAMBDA EXPRESSIONS

Lambda Expressions: Allow concise function-like behavior. You write code with the least amount of syntax.

The minimalist syntax objective can be maddening at times. Programming in this style takes time to appreciate and even more to excel.

LAMBDA EXAMPLE



```
// Functional style
List<Integer> numbers = Arrays.asList(1, 2, 3, 4);
numbers.forEach( Integer n -> System.out.println(n * 2));



// compare to
for (Integer n : numbers) {
    System.out.println(n * 2);
}
// Same results but less syntax and less error-prone
```

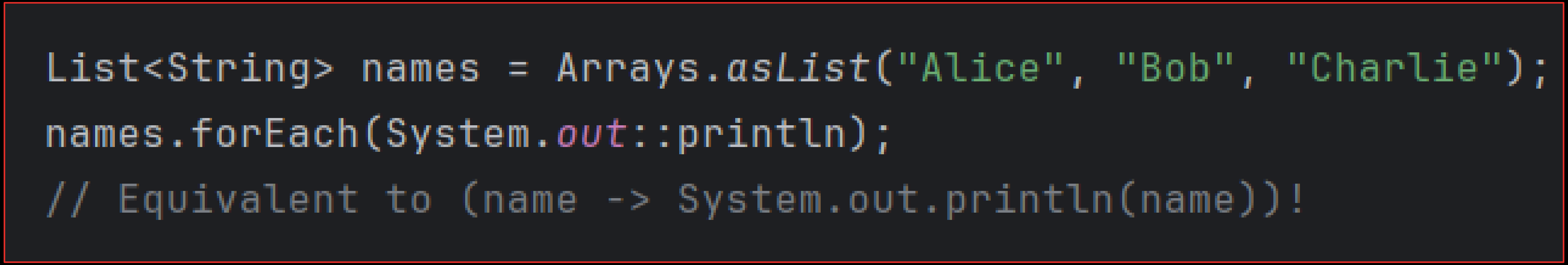
This is the Java lambda expression, distinguished by the arrow separating the argument ("`n`") and function body.

To the right of the arrow is
the expression that is returned from the lambda.

This is `System.out.println`, which accepts the "`n`" as the parameter.

Compare t he same functionality without lambdas.

METHOD REFERENCES



```java
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
names.forEach(System.out::println);
// Equivalent to (name -> System.out.println(name))!
```

A shorthand for lambdas where a method can be referenced by name. `System.out.println(name)` has been replaced with

- `System.out::println`
- The redundant "`name`" has been removed!

BASIC SYNTAX

1. The arguments
2. Followed by the “->” aka “produces”
3. Followed by the “method body”
    1. If there is a single argument (e.g., “n” or “name” from our previous example), the argument can be skipped
    2. Best practice is to use parentheses around the arguments
    3. When there are no arguments, YOU MUST use parentheses to indicate empty arguments
    4. Multiple arguments require a parenthesized argument list

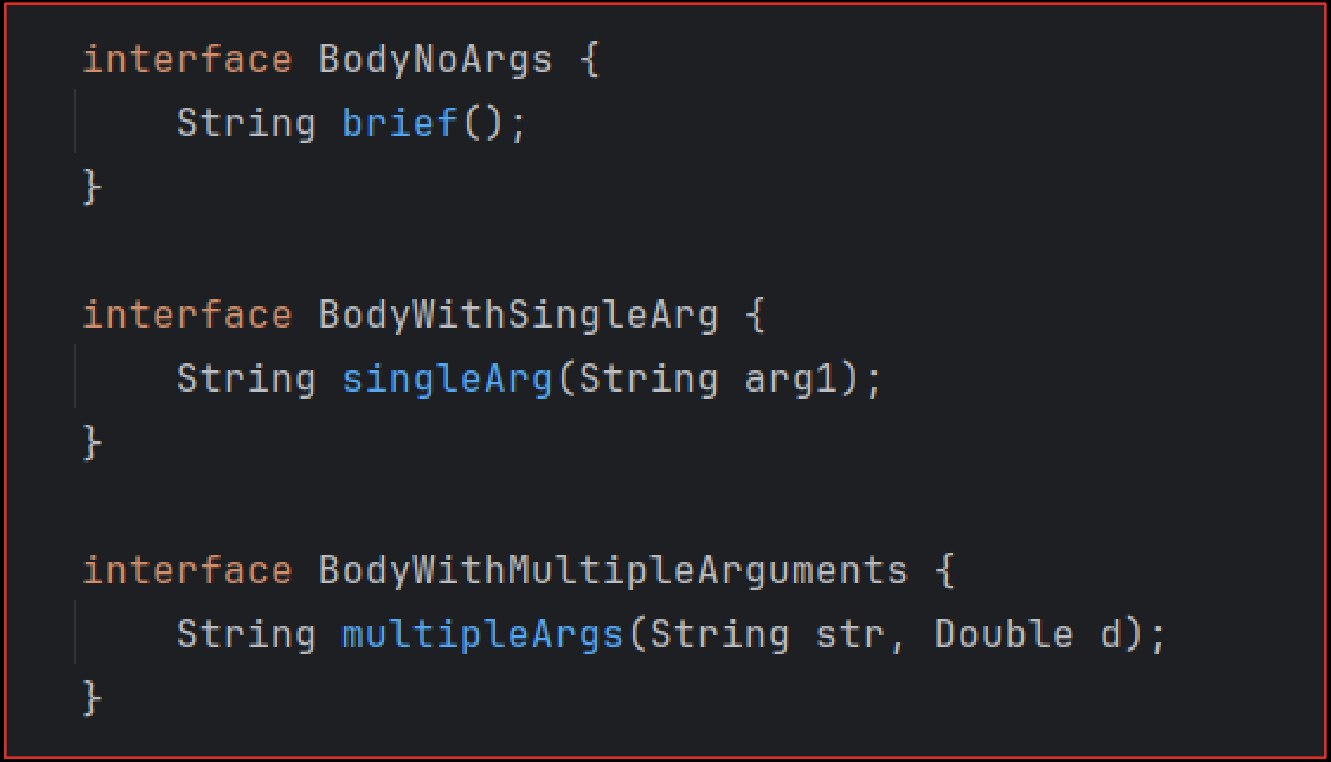
BASIC SYNTAX

4. …
   When the lambda expressions are in a single line, the "`return`" keyword is illegal. Yes, brevity!

5. When multiple lines are required in the method body, these lines must be wrapped by curly braces "`{…}`"

   In this case, revert to using the "`return`" keyword.

STEP 1 – THE ARGUMENTS

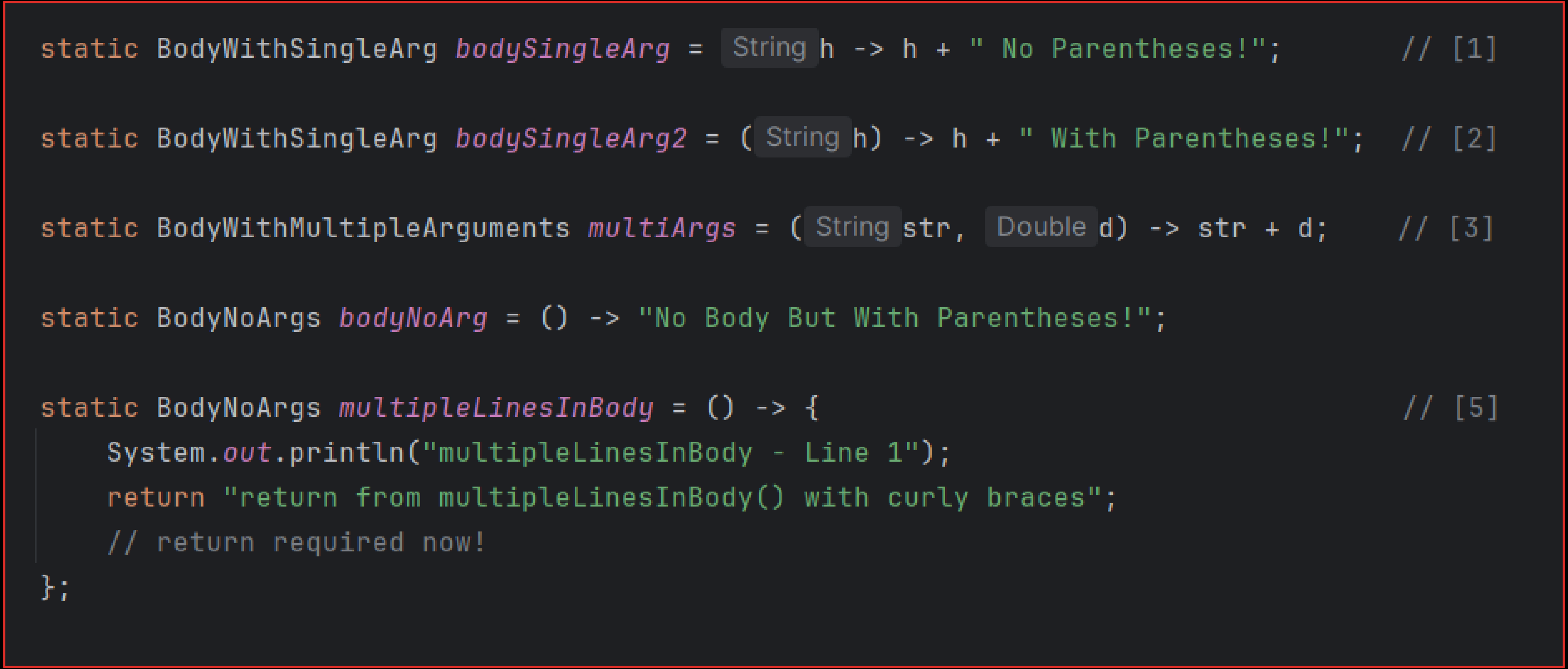


```
interface BodyNoArgs {
    String brief();
}

interface BodyWithSingleArg {
    String singleArg(String arg1);
}

interface BodyWithMultipleArguments {
    String multipleArgs(String str, Double d);
}
```

Assume three interfaces, each with a single method.

Each method has a different number of arguments to demonstrate lambda expression syntax.

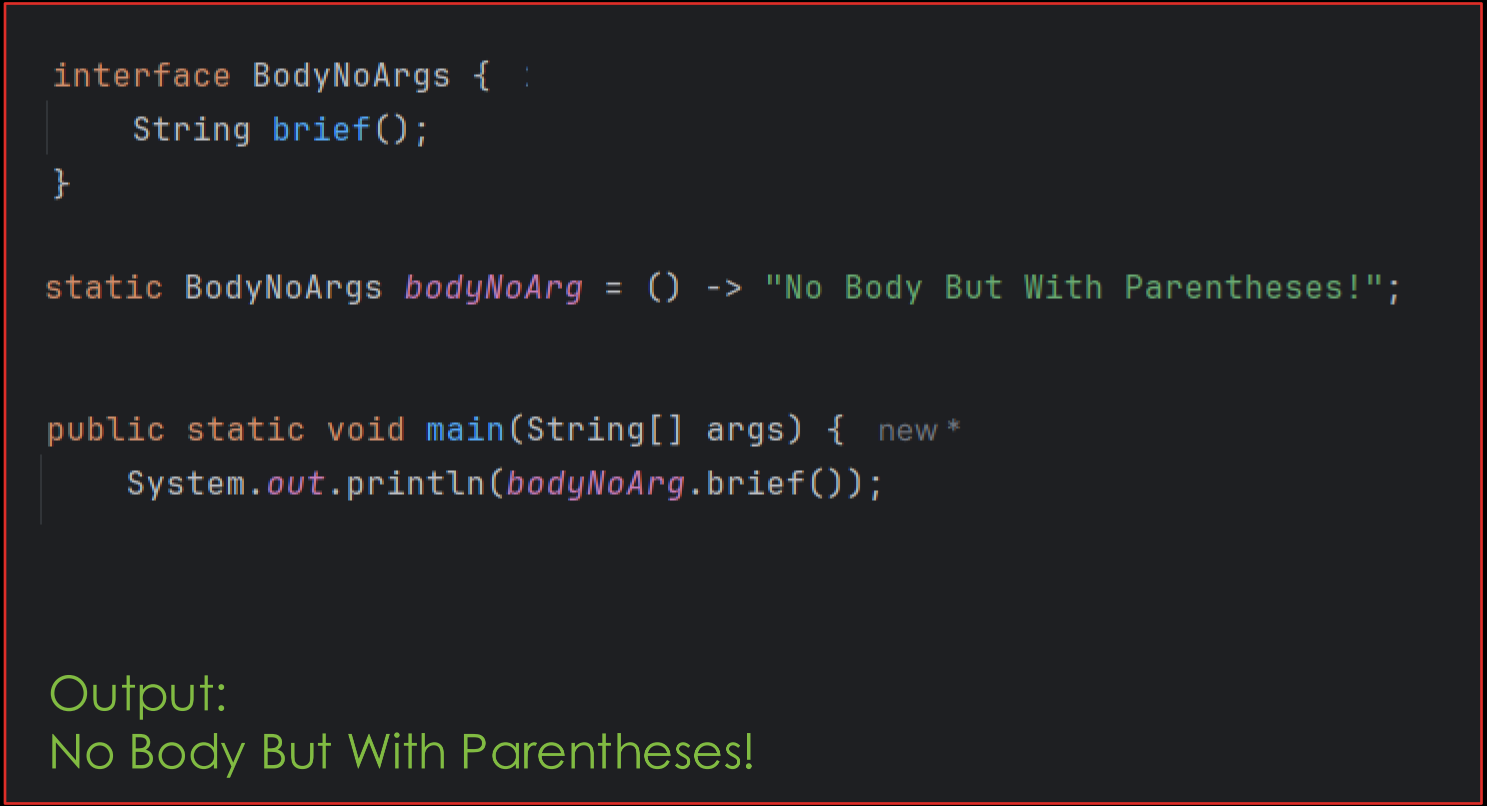
STEP 2, 3 –> PRODUCES



```
static BodyWithSingleArg bodySingleArg = String h -> h + " No Parentheses!";         // [1]

static BodyWithSingleArg bodySingleArg2 = ( String h) -> h + " With Parentheses!";  // [2]

static BodyWithMultipleArguments multiArgs = ( String str,  Double d) -> str + d;    // [3]

static BodyNoArgs bodyNoArg = () -> "No Body But With Parentheses!";

static BodyNoArgs multipleLinesInBody = () -> {                                     // [5]
    System.out.println("multipleLinesInBody - Line 1");
    return "return from multipleLinesInBody() with curly braces";
    // return required now!
};
```

After the “->” (aka finger, produces) is the method body

NO ARGUMENT

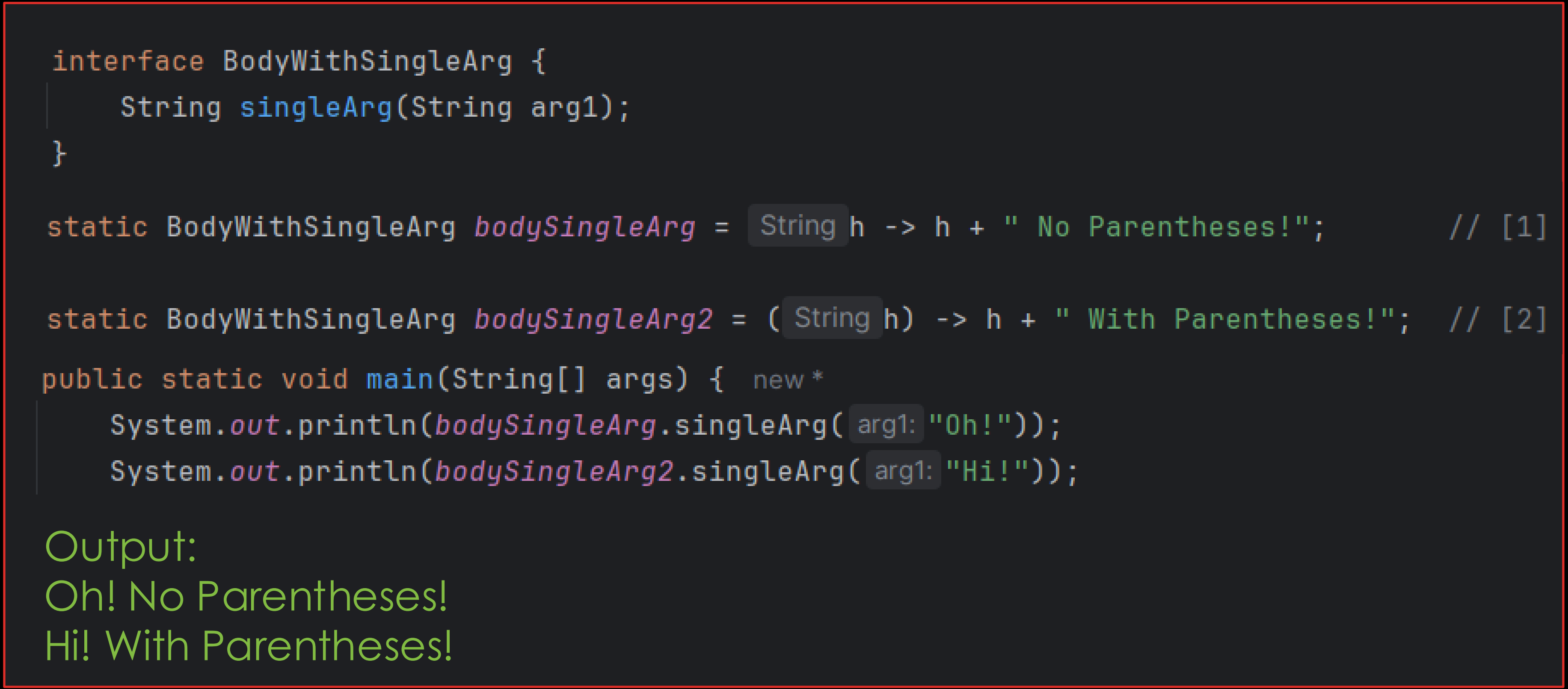


```
interface BodyNoArgs {
    String brief();
}

static BodyNoArgs bodyNoArg = () -> "No Body But With Parentheses!";


public static void main(String[] args) {  new *
    System.out.println(bodyNoArg.brief());
```

Output:
No Body But With Parentheses!

A particular case with a single argument can be produced without the parentheses.

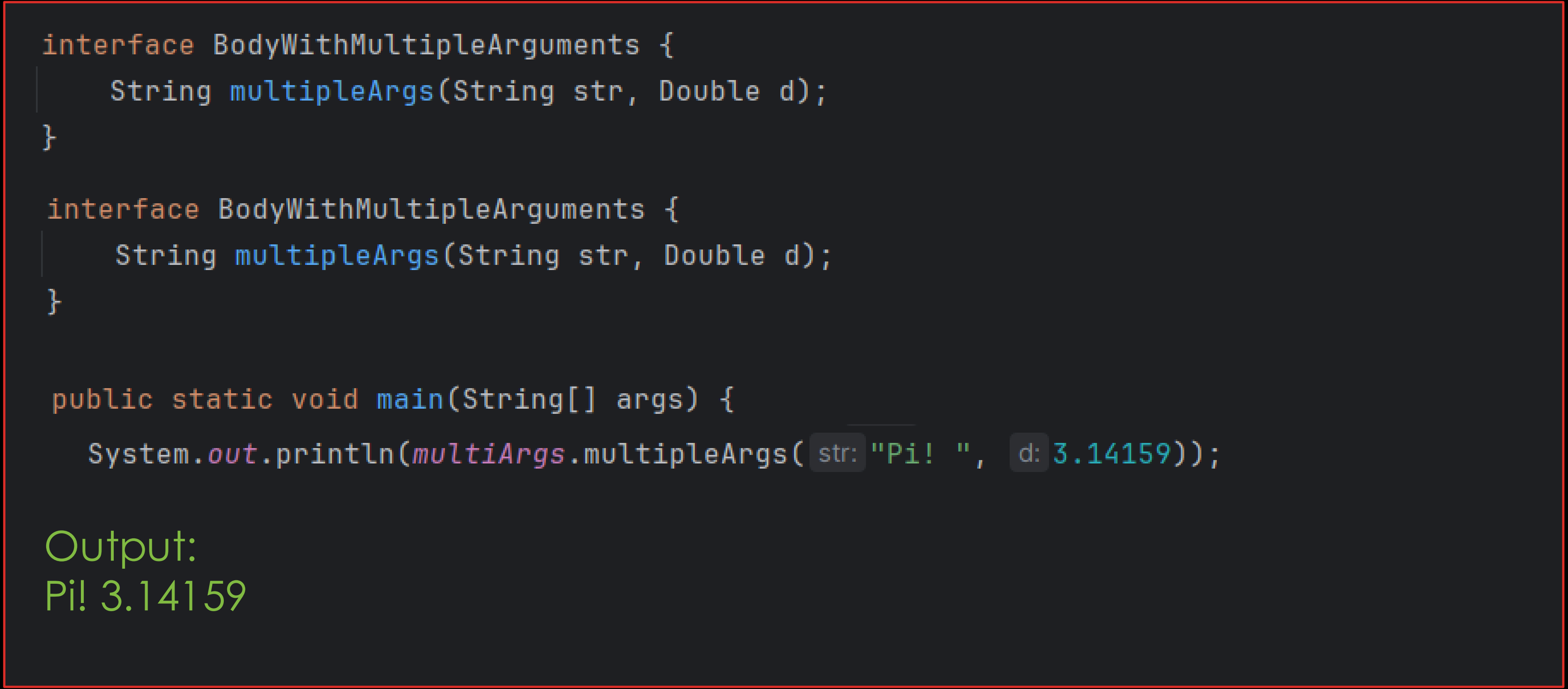
SINGLE ARGUMENT



```java
interface BodyWithSingleArg {
    String singleArg(String arg1);
}

static BodyWithSingleArg bodySingleArg = String h -> h + " No Parentheses!";         // [1]

static BodyWithSingleArg bodySingleArg2 = ( String h) -> h + " With Parentheses!";   // [2]

public static void main(String[] args) {  new *
    System.out.println(bodySingleArg.singleArg( arg1: "Oh!"));
    System.out.println(bodySingleArg2.singleArg( arg1: "Hi!"));
```

```
Output:
Oh! No Parentheses!
Hi! With Parentheses!
```

A particular case with a single argument can be produced without the parentheses.



```
interface BodyWithMultipleArguments {
    String multipleArgs(String str, Double d);
}
```

```
interface BodyWithMultipleArguments {
    String multipleArgs(String str, Double d);
}
```

```
public static void main(String[] args) {
   System.out.println(multiArgs.multipleArgs( str: "Pi! ",  d: 3.14159));
```

Output:
Pi! 3.14159

A particular case with a single argument can be produced without the parentheses.

METHOD REFERENCES

- A Method reference is a class name OR
- An object name, followed by “::” and then
- The name of the method

It is a shorthand notation of a lambda expression to call a method.

It enables referring to methods by their names, improving code readability and conciseness.

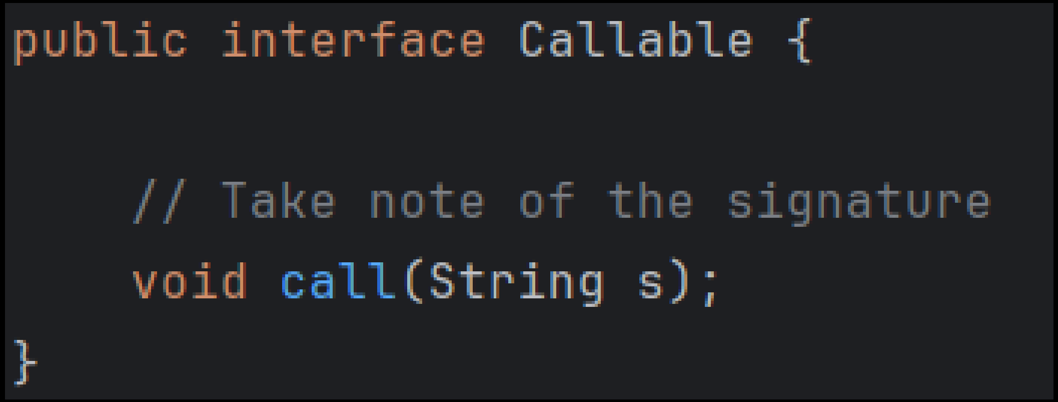
KEY POINTS : METHOD REFERENCES

Key Points:

- Method references can be used wherever a functional interface is expected.
- They are typically more compact and readable than the equivalent lambda expressions.
- Denoted by the :: operator.

METHOD REFERENCE EXAMPLE

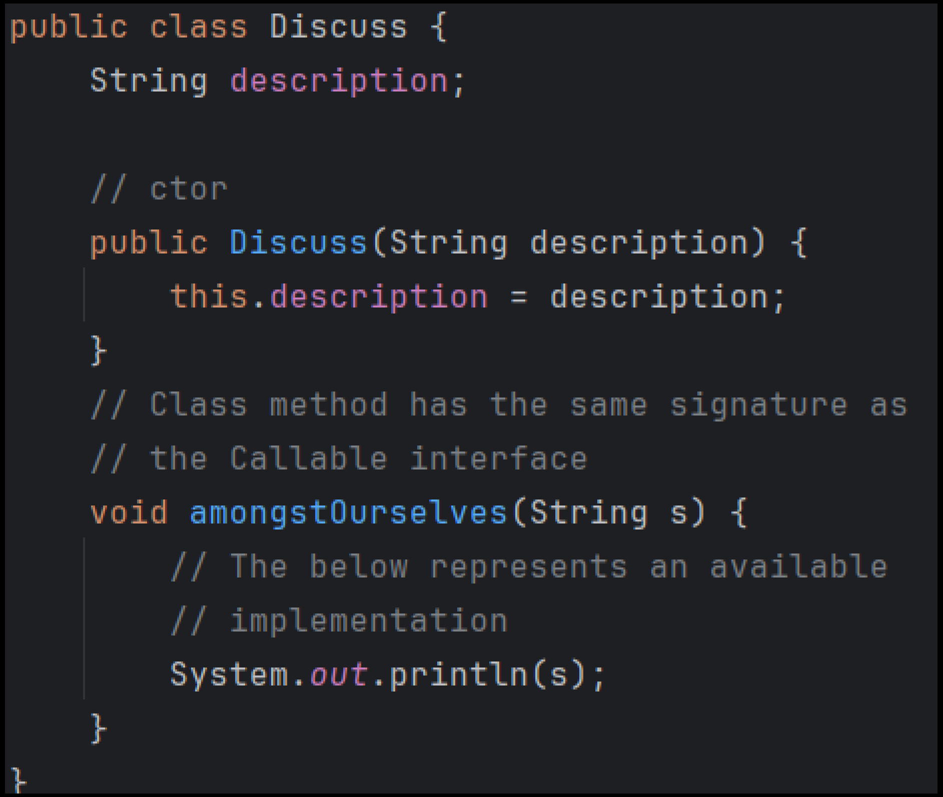
Start with the Callable interface with one void method that takes one parameter of type String



```java
public interface Callable {

    // Take note of the signature
    void call(String s);
}
```

METHOD REFERENCE EXAMPLE

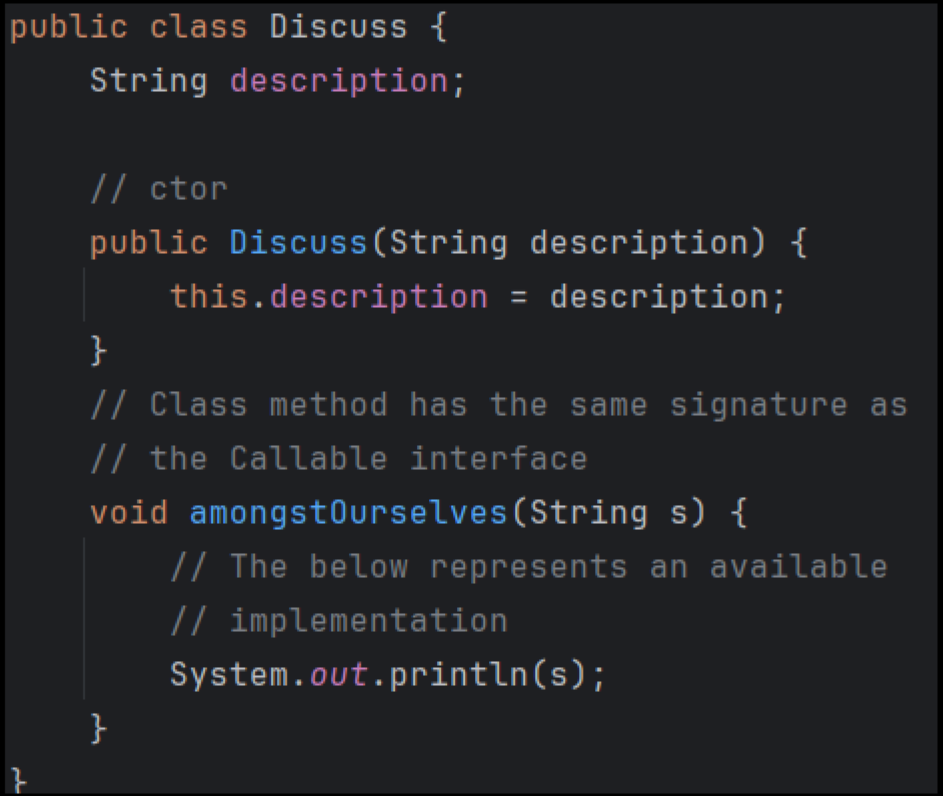
Add a class called Discuss with one void method that takes one parameter of type `String` and a ctor that takes a `String`



```java
public class Discuss {
    String description;

    // ctor
    public Discuss(String description) {
        this.description = description;
    }
    // Class method has the same signature as
    // the Callable interface
    void amongstOurselves(String s) {
        // The below represents an available
        // implementation
        System.out.println(s);
    }
}
```

METHOD REFERENCE EXAMPLE

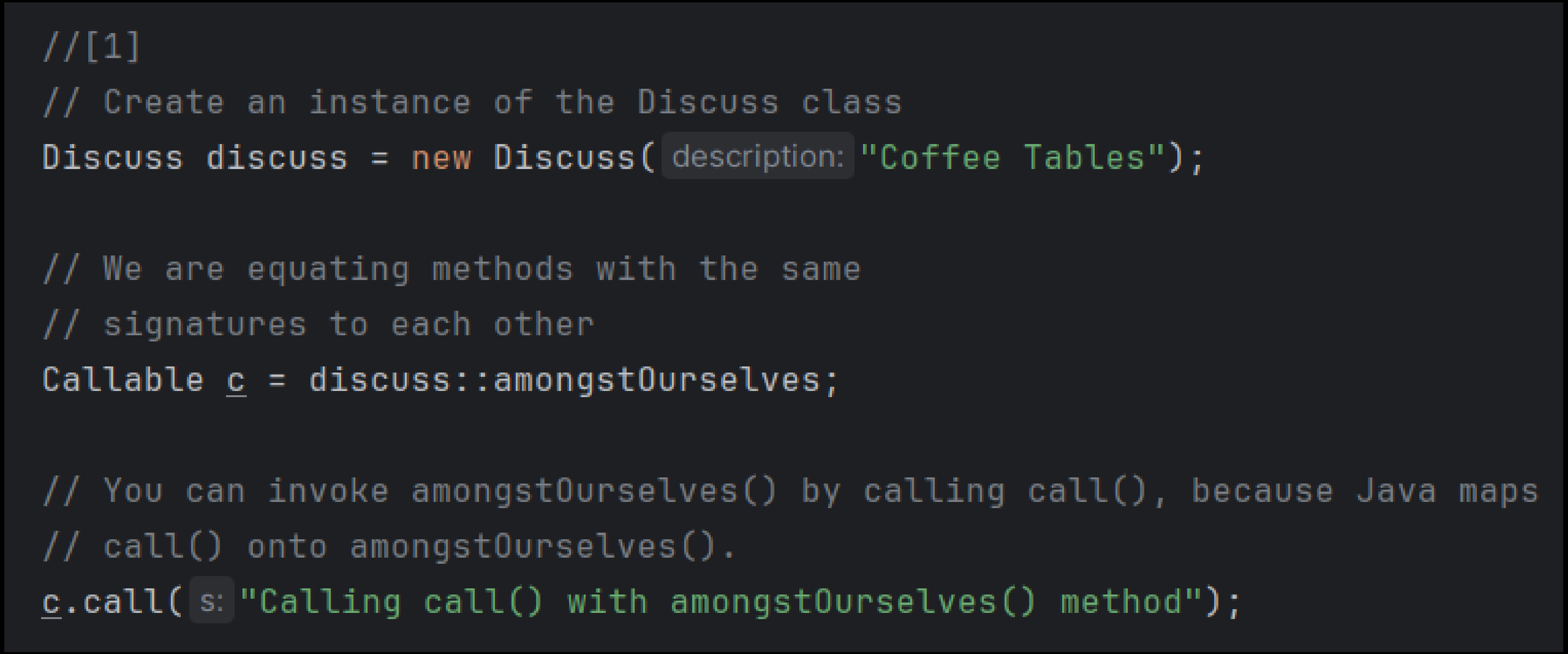
Add a class called Discuss with one void method that takes one parameter of type `String` and a ctor that takes a `String`



```java
public class Discuss {
    String description;

    // ctor
    public Discuss(String description) {
        this.description = description;
    }
    // Class method has the same signature as
    // the Callable interface
    void amongstOurselves(String s) {
        // The below represents an available
        // implementation
        System.out.println(s);
    }
}
```

a
amongstOurselves

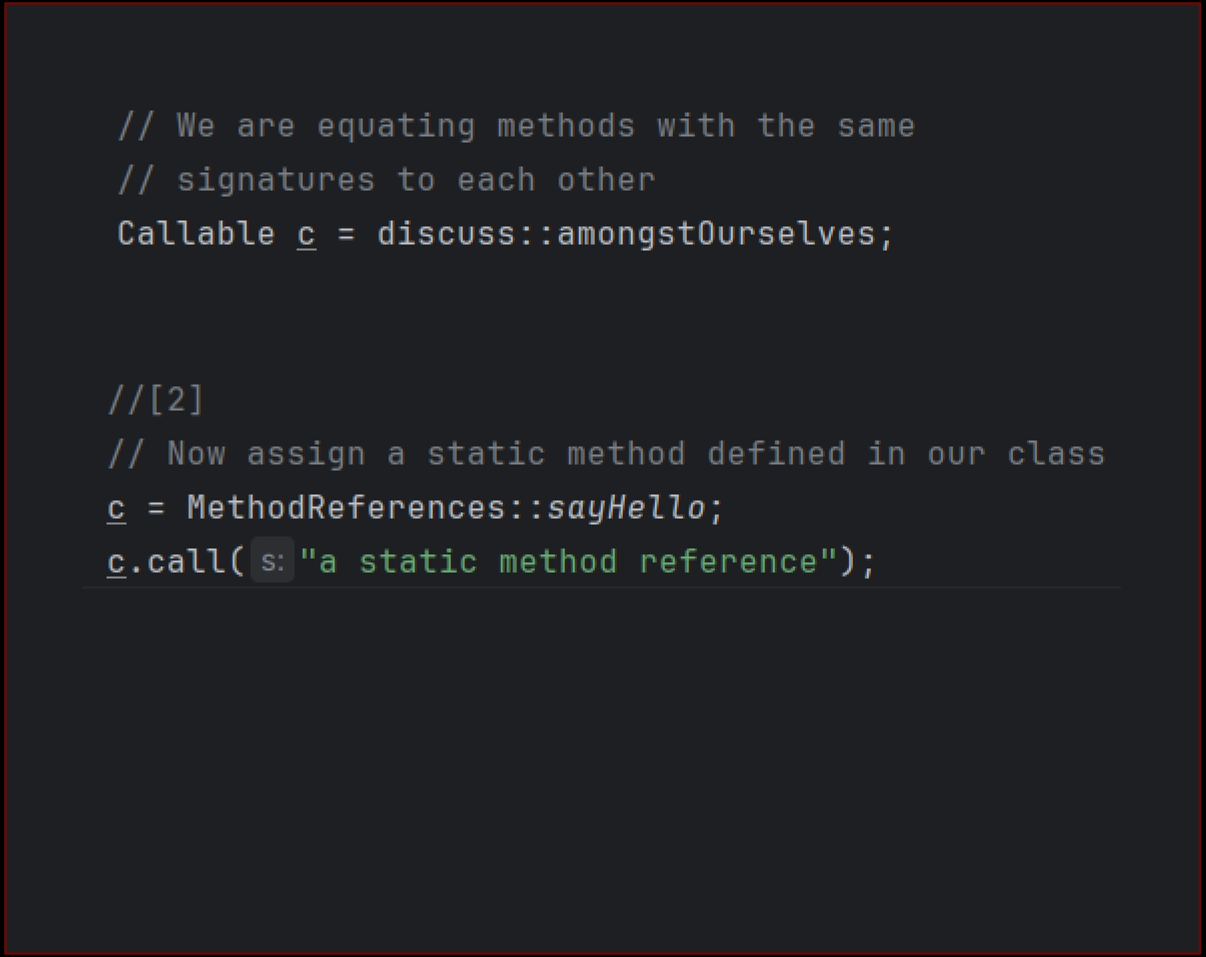
EQUATING METHODS SAME SIGNATURES



```java
//[1]
// Create an instance of the Discuss class
Discuss discuss = new Discuss( description: "Coffee Tables");

// We are equating methods with the same
// signatures to each other
Callable c = discuss::amongstOurselves;

// You can invoke amongstOurselves() by calling call(), because Java maps
// call() onto amongstOurselves().
c.call( s: "Calling call() with amongstOurselves() method");
```

You can invoke amongstOurselves () by calling call()
because Java maps call() onto amongstOurselves ().

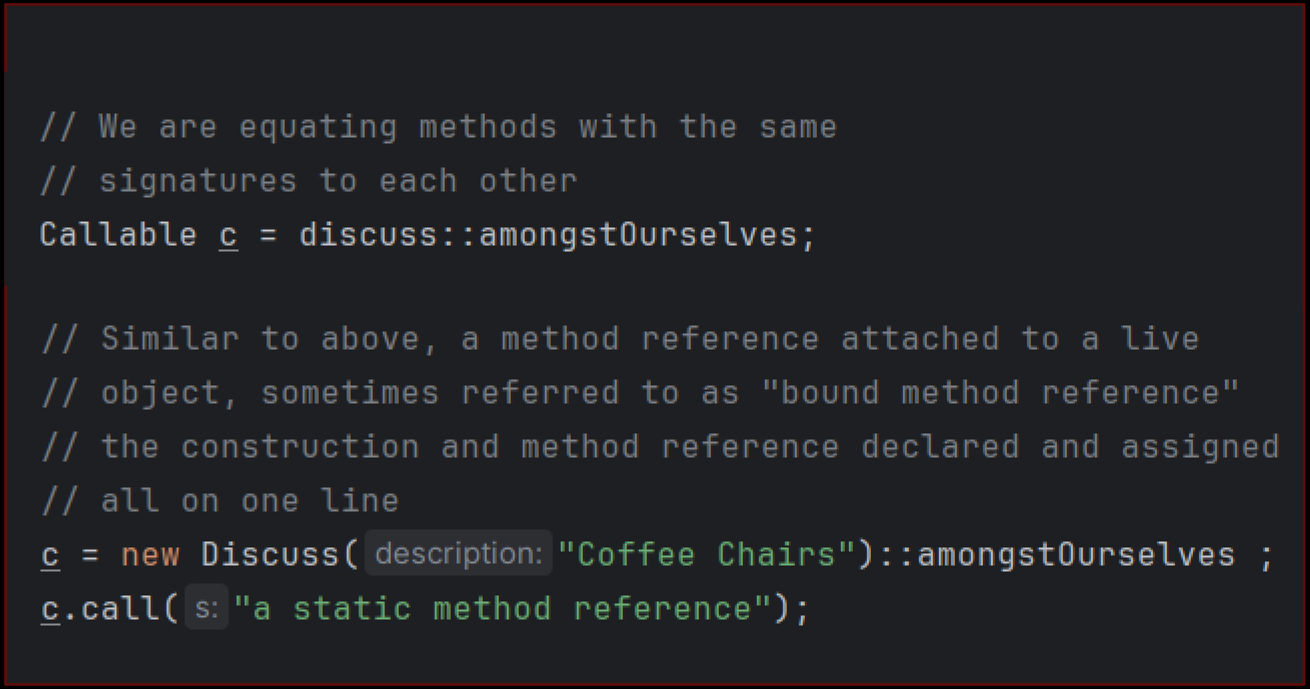
ASSIGN STATIC METHOD



```
// We are equating methods with the same
// signatures to each other
Callable c = discuss::amongstOurselves;


//[2]
// Now assign a static method defined in our class
c = MethodReferences::sayHello;
c.call( s: "a static method reference");
```

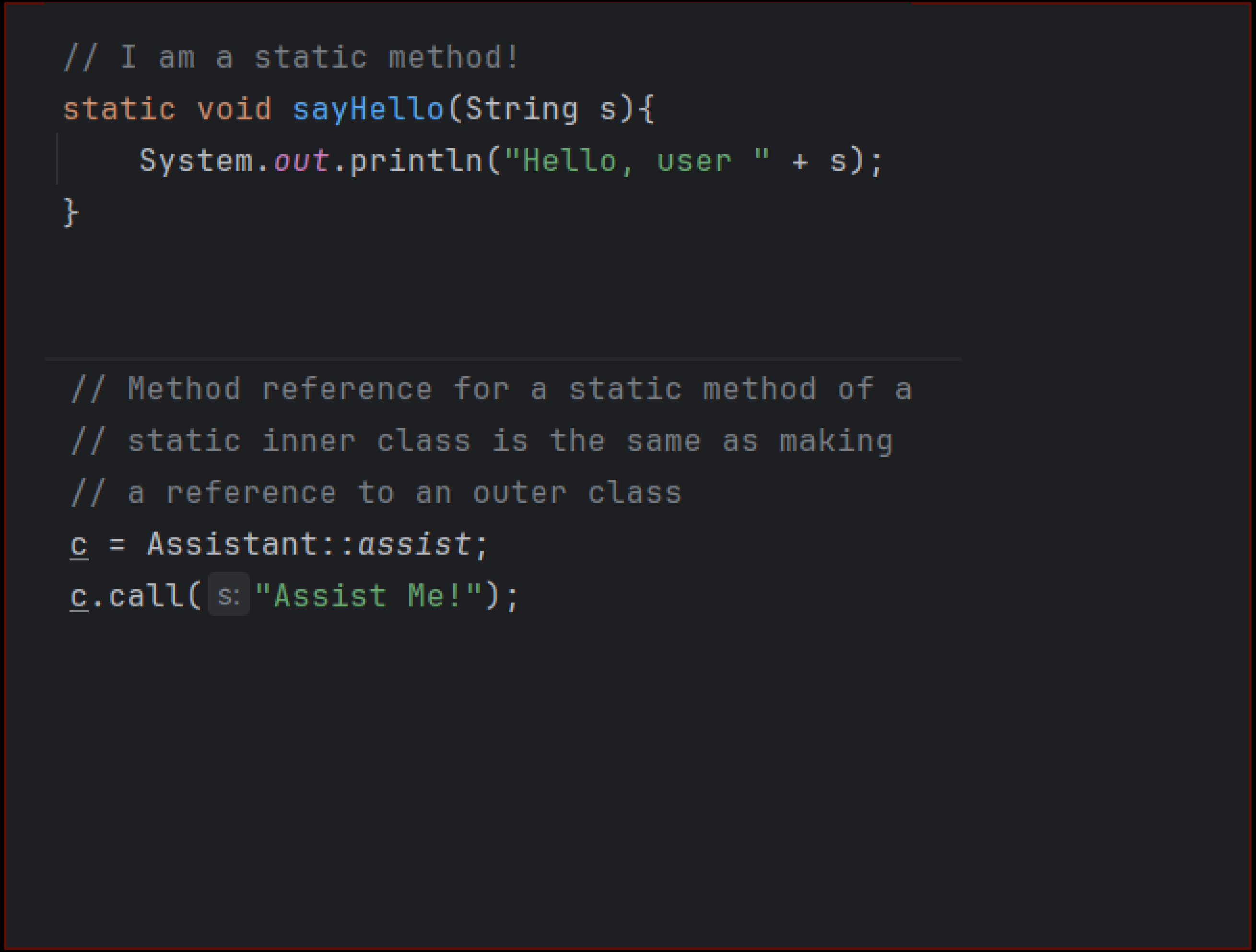
You can invoke sayHello() by calling call() because Java maps call() onto sayHello().

CONSTRUCTION AND ASSIGNMENT



```
// We are equating methods with the same
// signatures to each other
Callable c = discuss::amongstOurselves;

// Similar to above, a method reference attached to a live
// object, sometimes referred to as "bound method reference"
// the construction and method reference declared and assigned
// all on one line
c = new Discuss( description: "Coffee Chairs")::amongstOurselves ;
c.call( s: "a static method reference");
```

METHOD REFERENCE TO STATIC METHOD



```java
// I am a static method!
static void sayHello(String s){
    System.out.println("Hello, user " + s);
}
```

```java
// Method reference for a static method of a
// static inner class is the same as making
// a reference to an outer class
c = Assistant::assist;
c.call( s: "Assist Me!");
```

TOPIC SUMMARY

TOPIC SUMMARY - COLLECTIONS

Lambda expressions and method references do not make Java a functional language but support programming in a more functional style.

The paradigm is powerful and not required for day one. However, their use will only increase, and a topic that may be considered immediate today will likely become an introductory topic in the future.