{JG} JavaGuru

INTRODUCTION TO JAVA

Java 1.0

LAMBDAS & FUNCTIONS

Lesson # 14

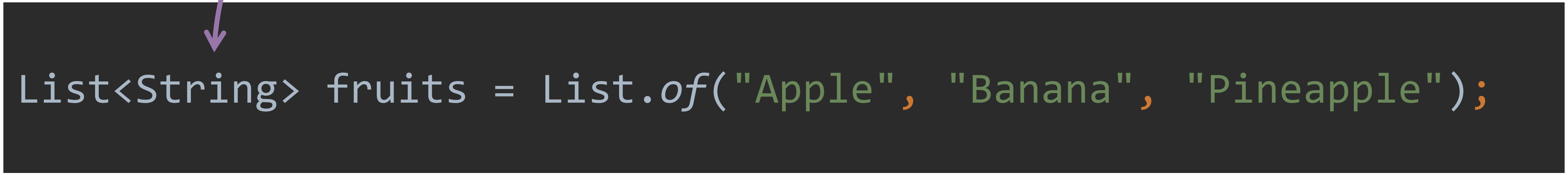
GENERICS

GENERICS IN JAVA

- Java 5 introduced the concept of Generics or parametrized types
- Generics provide the ability to write general or generic code which is independent of a particular type
- Generics provide compile-time type checking and remove the risk of ClassCastException that was common while working with collection classes

YOU ALREADY KNOW…

A type specification for
a generic List



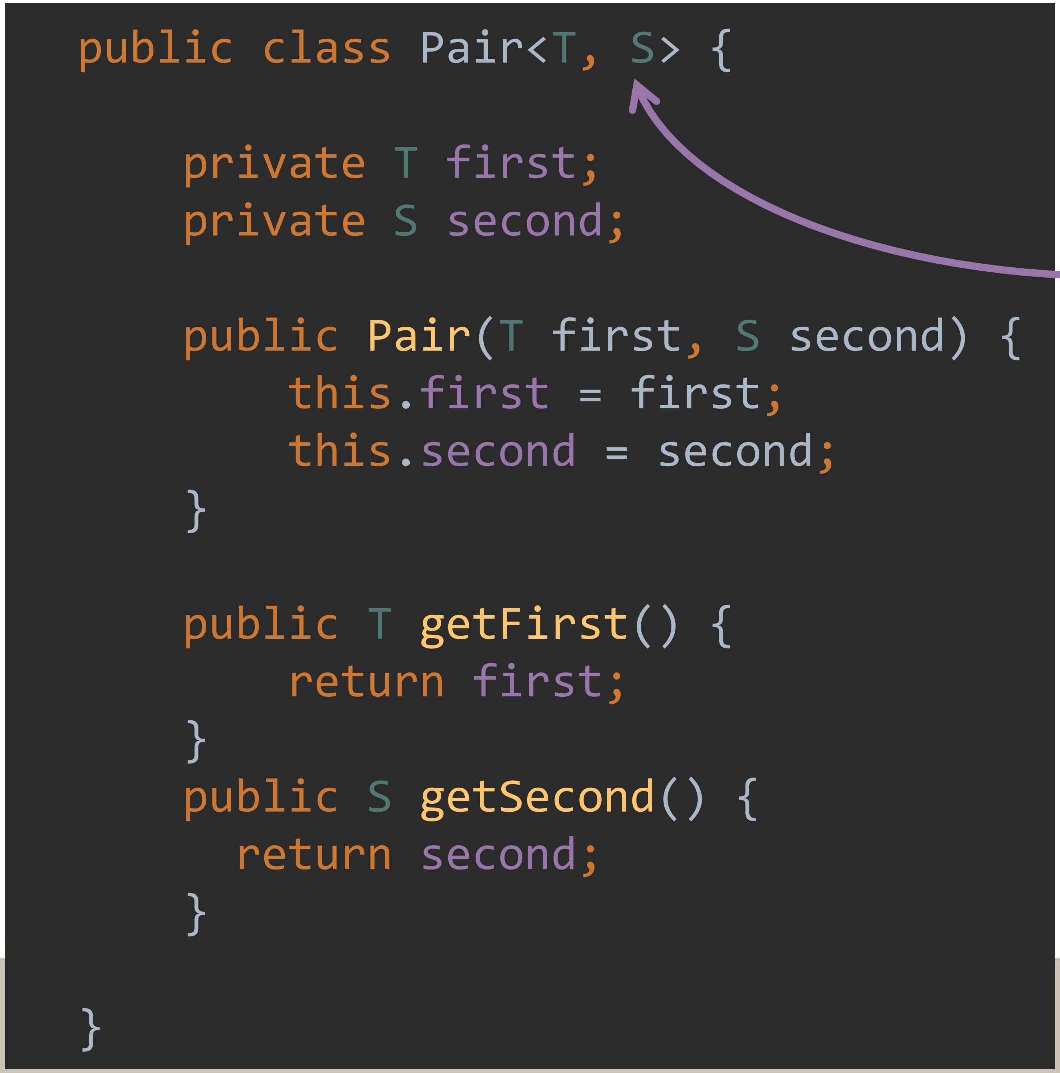
```java
List<String> fruits = List.of("Apple", "Banana", "Pineapple");
```

GENERICS USE CASES

- Here are the three common use patterns of generics that I think are worth knowing (examples below)…

1. Using a generic class, like using List<String>
2. Writing generic code with a simple <T> or <?> type parameter
3. Writing generic code with a <T extends Foo> type parameter

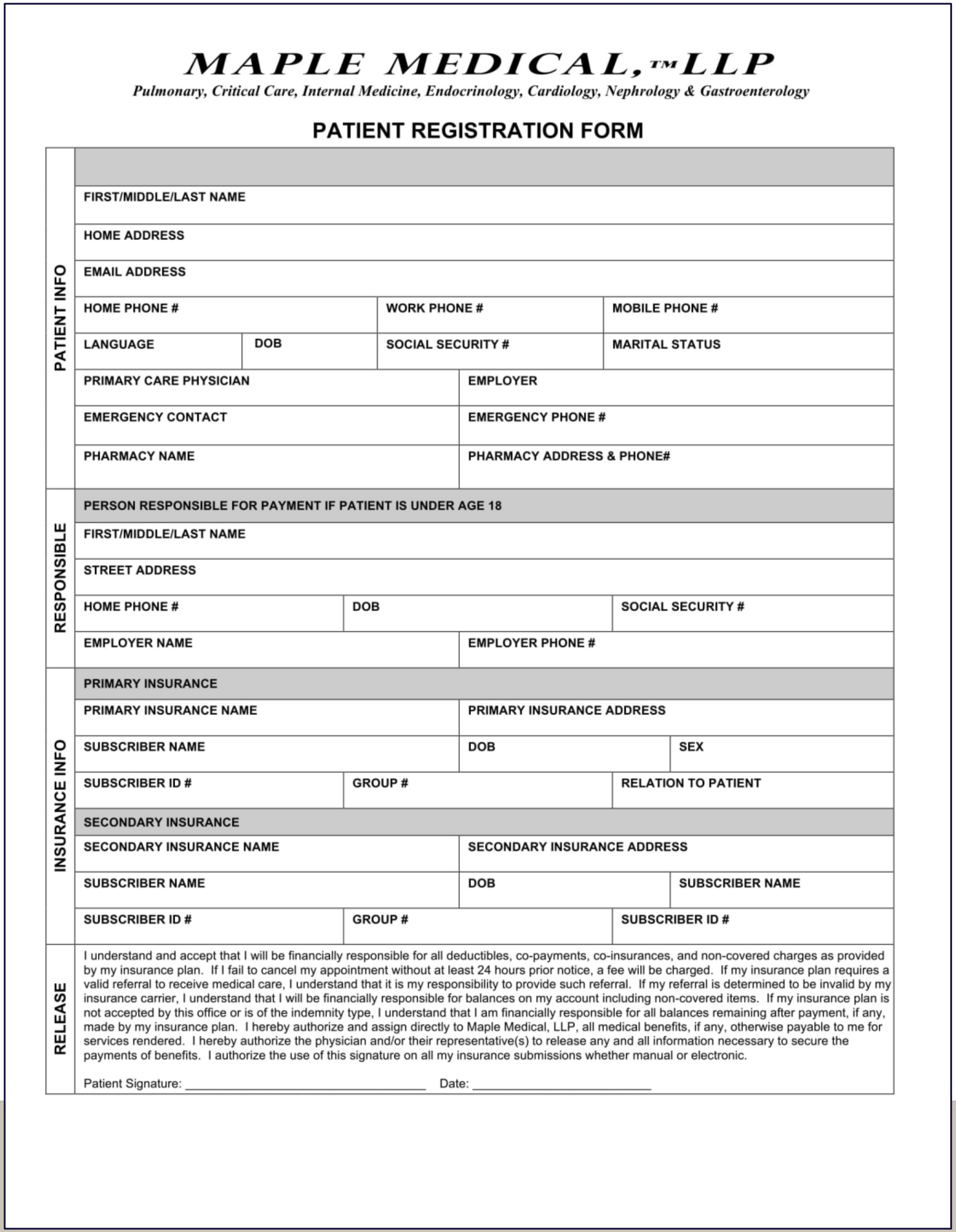
GENERIC CLASS EXAMPLE



```java
public class Pair<T, S> {

    private T first;
    private S second;

    public Pair(T first, S second) {
        this.first = first;
        this.second = second;
    }

    public T getFirst() {
        return first;
    }
    public S getSecond() {
      return second;
    }

}
```

Generic types T and S act as a placeholder for actual types

LIKE A BLANK FORM



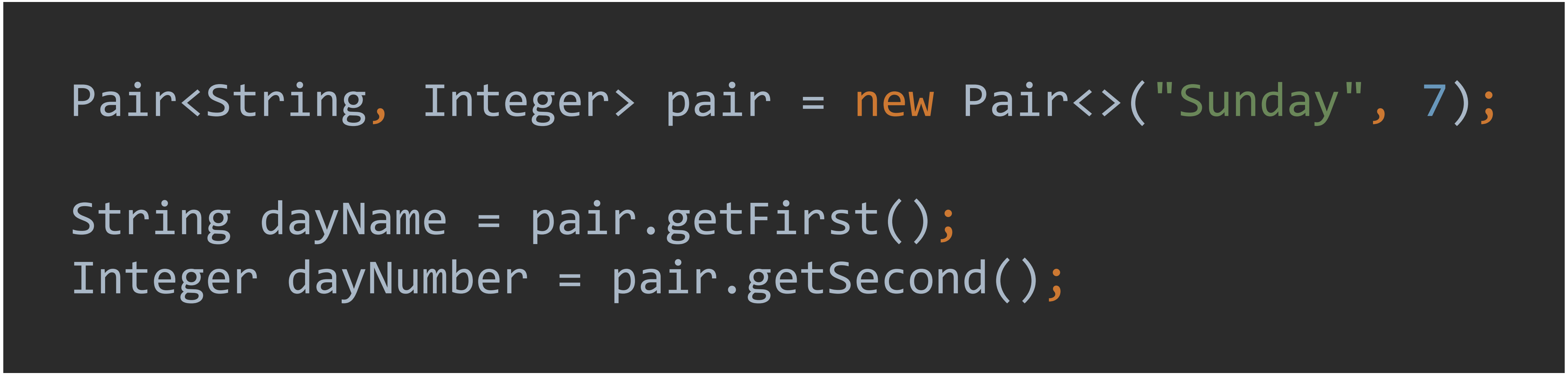
MAPLE MEDICAL,™ LLP

Pulmonary, Critical Care, Internal Medicine, Endocrinology, Cardiology, Nephrology & Gastroenterology

PATIENT REGISTRATION FORM

PATIENT INFO				
	FIRST/MIDDLE/LAST NAME			
	HOME ADDRESS			
	EMAIL ADDRESS			
	HOME PHONE #	WORK PHONE #	MOBILE PHONE #	
	LANGUAGE	DOB	SOCIAL SECURITY #	MARITAL STATUS
	PRIMARY CARE PHYSICIAN	EMPLOYER		
	EMERGENCY CONTACT	EMERGENCY PHONE #		
	PHARMACY NAME	PHARMACY ADDRESS & PHONE#		
RESPONSIBLE	PERSON RESPONSIBLE FOR PAYMENT IF PATIENT IS UNDER AGE 18			
	FIRST/MIDDLE/LAST NAME			
	STREET ADDRESS			
	HOME PHONE #	DOB	SOCIAL SECURITY #	
	EMPLOYER NAME	EMPLOYER PHONE #		
INSURANCE INFO	PRIMARY INSURANCE			
	PRIMARY INSURANCE NAME	PRIMARY INSURANCE ADDRESS		
	SUBSCRIBER NAME	DOB	SEX	
	SUBSCRIBER ID #	GROUP #	RELATION TO PATIENT	
	SECONDARY INSURANCE			
	SECONDARY INSURANCE NAME	SECONDARY INSURANCE ADDRESS		
	SUBSCRIBER NAME	DOB	SUBSCRIBER NAME	
	SUBSCRIBER ID #	GROUP #	SUBSCRIBER ID #	
RELEASE	I understand and accept that I will be financially responsible for all deductibles, co-payments, co-insurances, and non-covered charges as provided by my insurance plan. If I fail to cancel my appointment without at least 24 hours prior notice, a fee will be charged. If my insurance plan requires a valid referral to receive medical care, I understand that it is my responsibility to provide such referral. If my referral is determined to be invalid by my insurance carrier, I understand that I will be financially responsible for balances on my account including non-covered items. If my insurance plan is not accepted by this office or is of the indemnity type, I understand that I am financially responsible for all balances remaining after payment, if any, made by my insurance plan. I hereby authorize and assign directly to Maple Medical, LLP, all medical benefits, if any, otherwise payable to me for services rendered. I hereby authorize the physician and/or their representative(s) to release any and all information necessary to secure the payments of benefits. I authorize the use of this signature on all my insurance submissions whether manual or electronic. Patient Signature: ______________________ Date: ________________			

GENERIC CLASS USAGE



```
Pair<String, Integer> pair = new Pair<>("Sunday", 7);

String dayName = pair.getFirst();
Integer dayNumber = pair.getSecond();
```

NESTED CLASSES

NESTED CLASS

- In Java, it is possible to define a class within another class – such classes are known as nested classes

- Nested classes enable you to logically group classes only used in one place

- Thus, this increases the use of encapsulation and creates more readable and maintainable code

NESTED CLASS CHARACTERISTICS

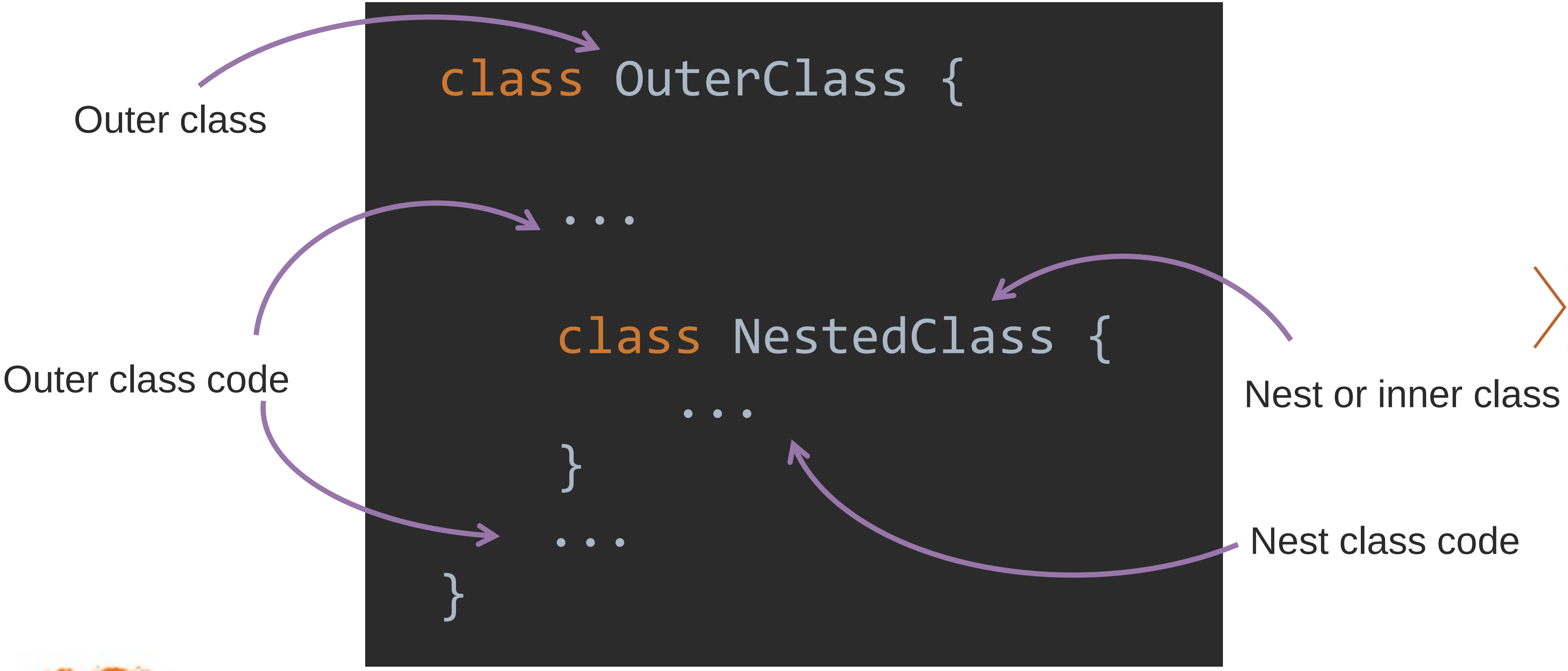
- The scope of a nested class is bounded by the scope of its enclosing class
- A nested class has access to the members, including private members, of the class in which it is nested
- An enclosing class does not have access to the members of the nested class
- A nested class is also a member of its enclosing class
- As a member of its enclosing class, a nested class can be declared private, public, protected, or package-private (default)

NESTED CLASS CATEGORIES

- Nested classes are divided into two categories:

  1. Static nested class: Nested classes that are declared static are called static nested classes
  2. Inner class: An inner class is a non-static nested class

NESTED CLASS SYNAX



```
class OuterClass {

    ...

    class NestedClass {
        ...
    }
    ...
}
```

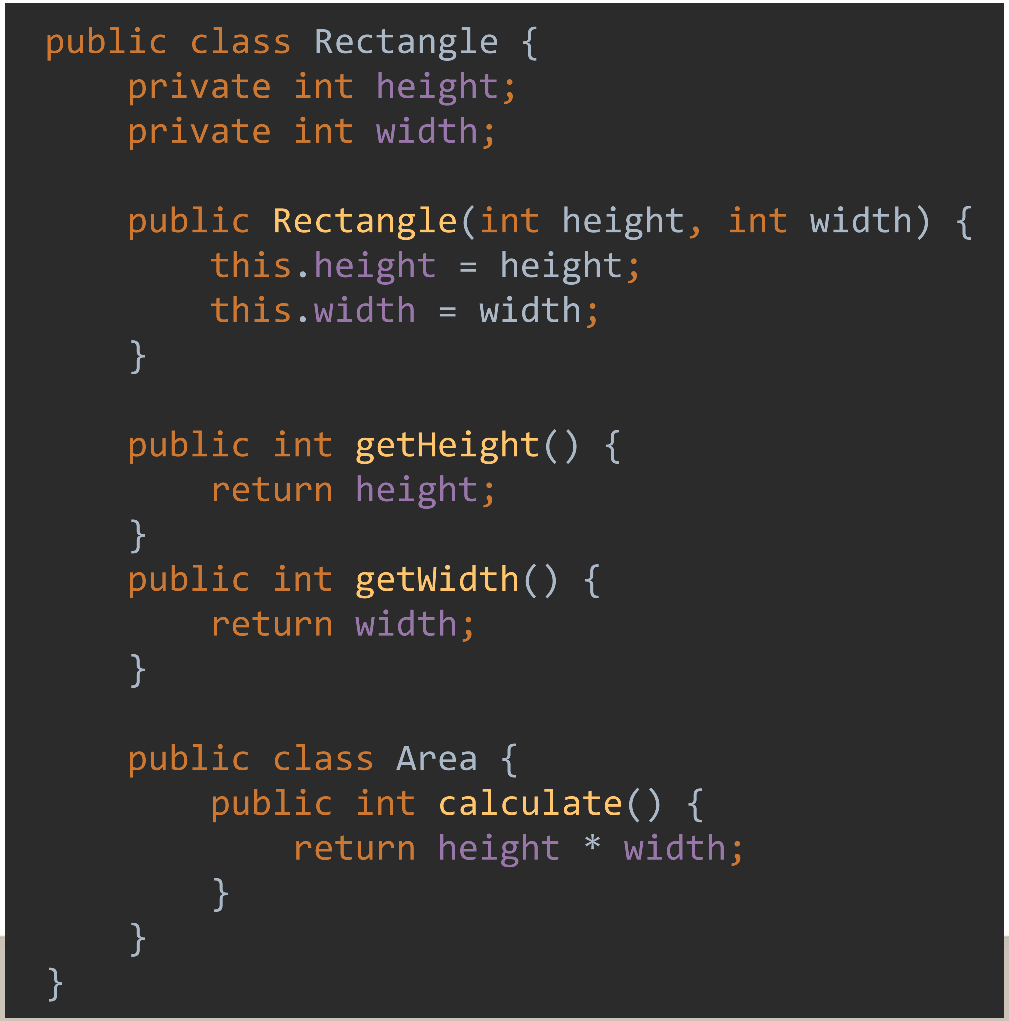
Outer class

Outer class code

Nest or inner class

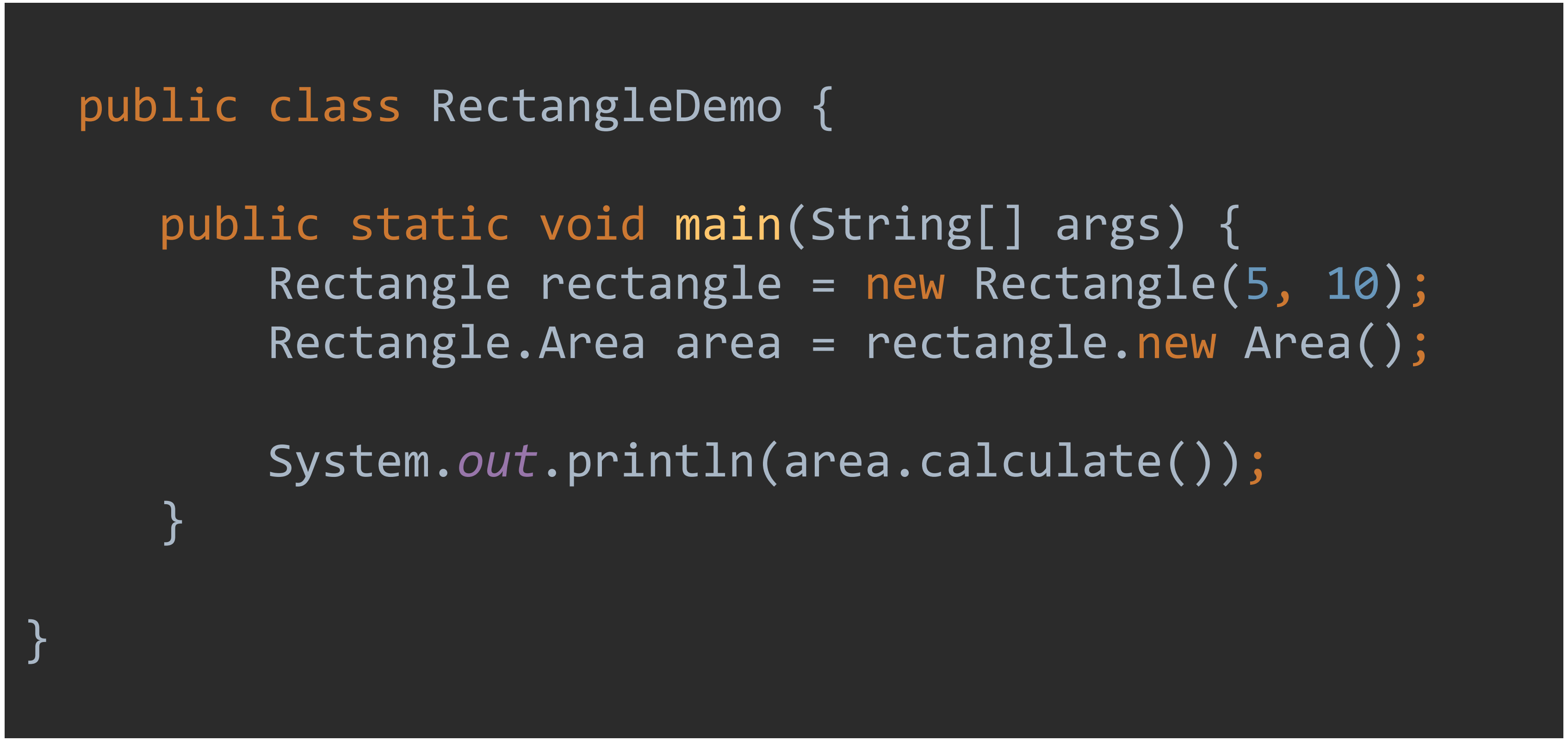
Nest class code

NESTED CLASS EXAMPLE



```java
public class Rectangle {
    private int height;
    private int width;

    public Rectangle(int height, int width) {
        this.height = height;
        this.width = width;
    }

    public int getHeight() {
        return height;
    }
    public int getWidth() {
        return width;
    }

    public class Area {
        public int calculate() {
            return height * width;
        }
    }
}
```

NESTED CLASS USAGE



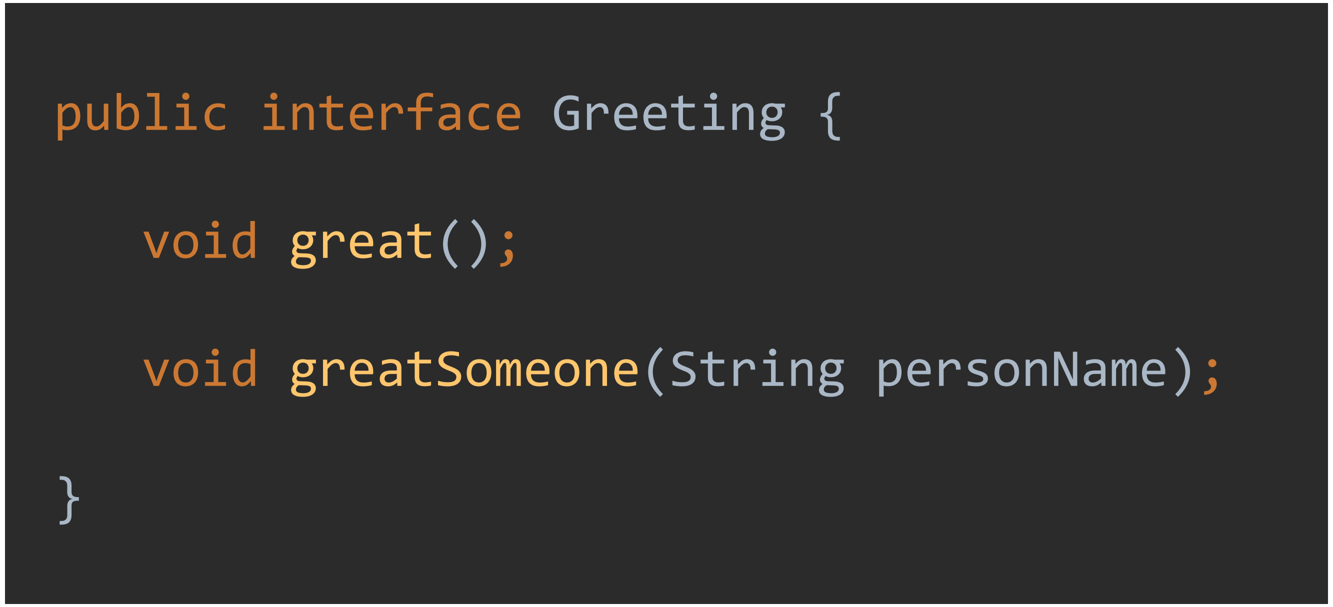
```
  public class RectangleDemo {

     public static void main(String[] args) {
         Rectangle rectangle = new Rectangle(5, 10);
         Rectangle.Area area = rectangle.new Area();

         System.out.println(area.calculate());
     }

}
```

ANONYMOUS CLASS

- Anonymous class is a nested class without a name and for which only a single object is created

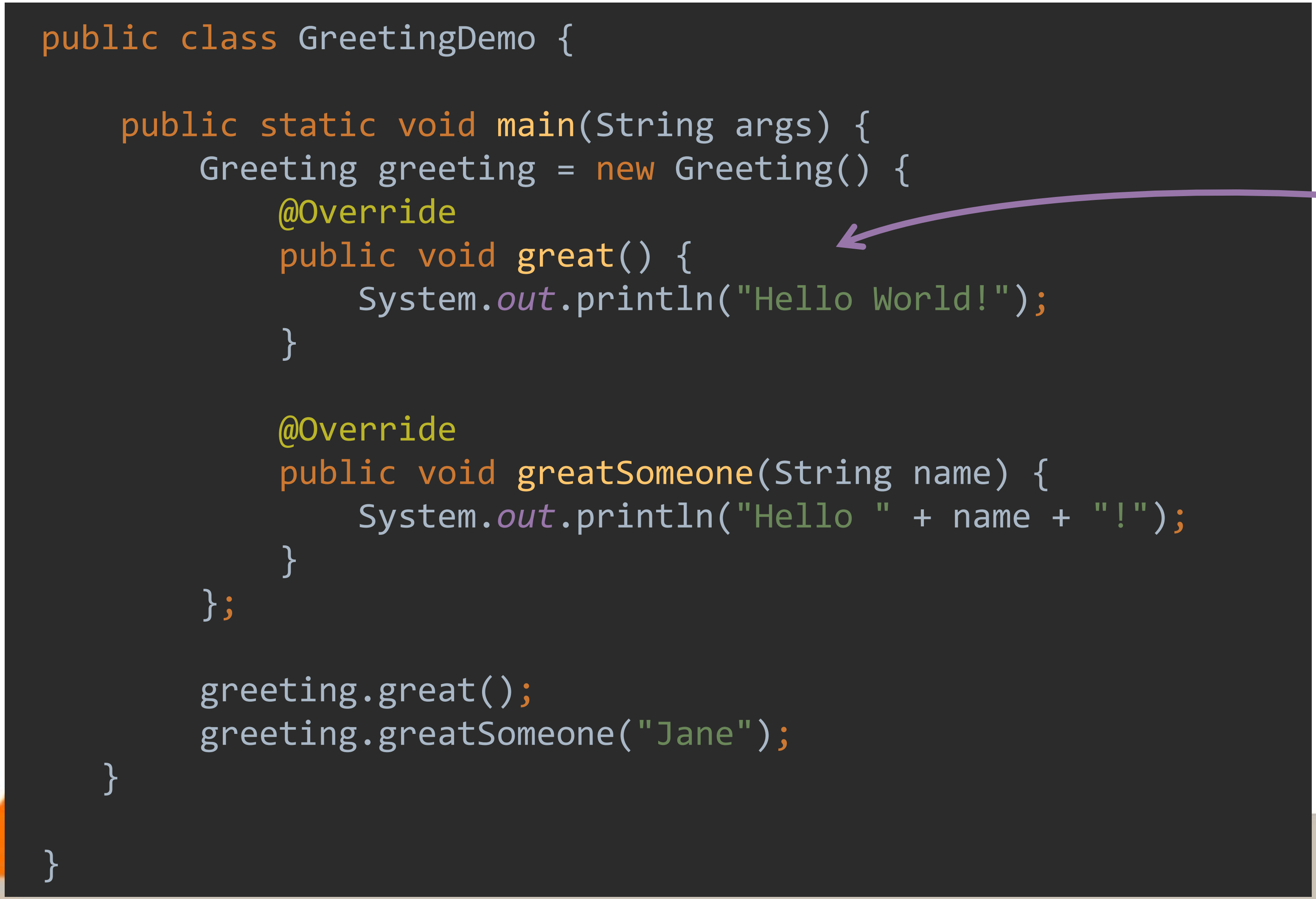
- An anonymous inner class can be useful when making an instance of an object with certain “extras,” such as overriding methods of a class or interface, without having actually to subclass a class

Anonymous Class Example



```
public interface Greeting {

   void great();

   void greatSomeone(String personName);

}
```

ANONYMOUS CLASS EXAMPLE



```java
public class GreetingDemo {

    public static void main(String args) {
        Greeting greeting = new Greeting() {
            @Override
            public void great() {
                System.out.println("Hello World!");
            }

            @Override
            public void greatSomeone(String name) {
                System.out.println("Hello " + name + "!");
            }
        };

        greeting.great();
        greeting.greatSomeone("Jane");
    }

}
```

Anonymous class

FUNCTIONAL PROGRAMMING

FUNCTIONAL PROGRAMMING

- Functional programming is centered around building software composed of functions, similar to procedural programming
- Functional programming is a declarative programming paradigm where programs are created by applying sequential functions rather than statements
- Functions can be:
  - stored in a variable
  - passed as an argument
  - returned from a function

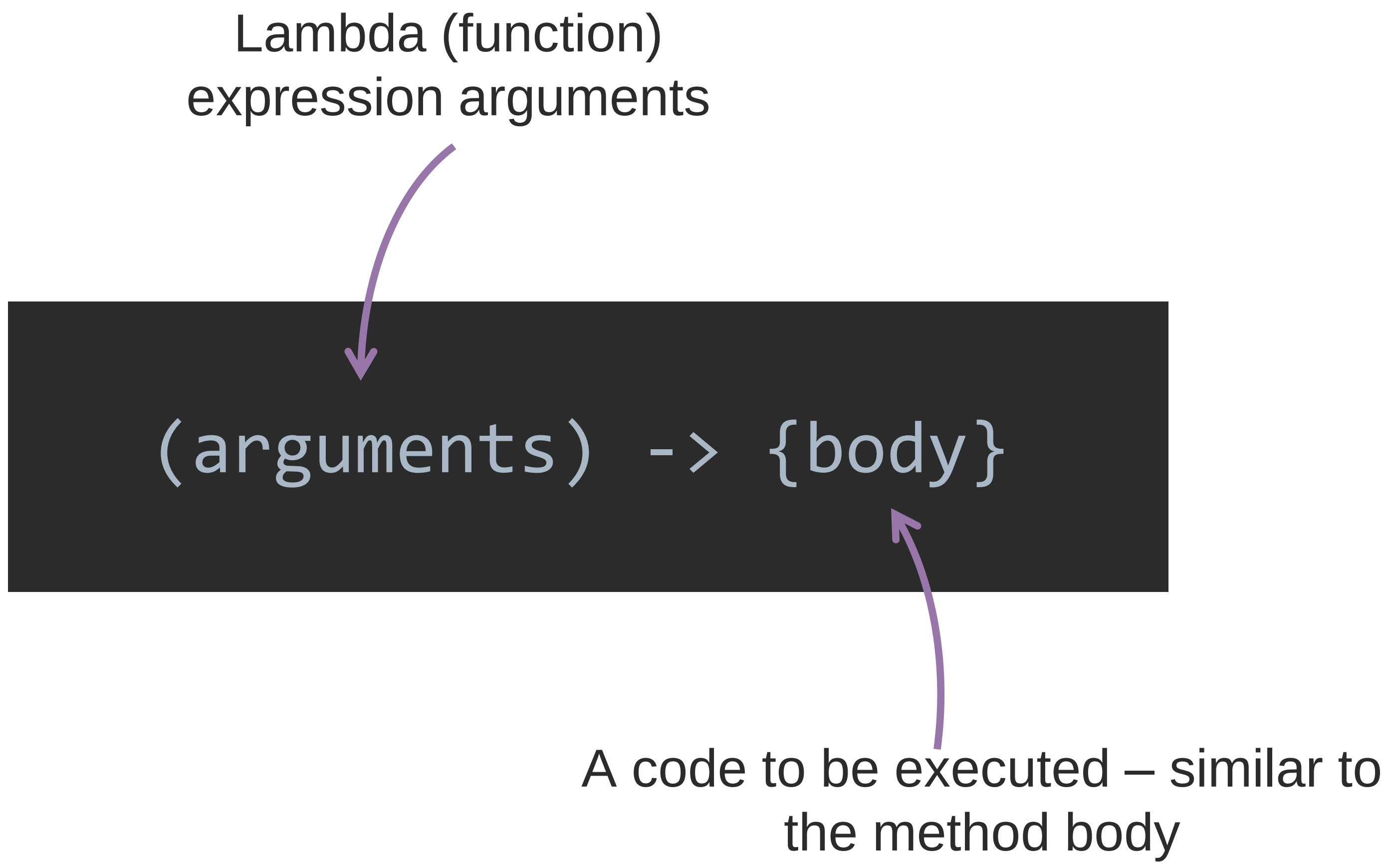
LAMBDA EXPRESSION

- Lambda expression is a new and vital feature of Java that was included in Java 8

- It provides a clear and concise way to represent one method interface using an expression.

- The Lambda expression is used to provide the implementation of a functional interface

LAMBDA EXPRESSION SYNTAX



Lambda (function) expression arguments

```
(arguments) -> {body}
```

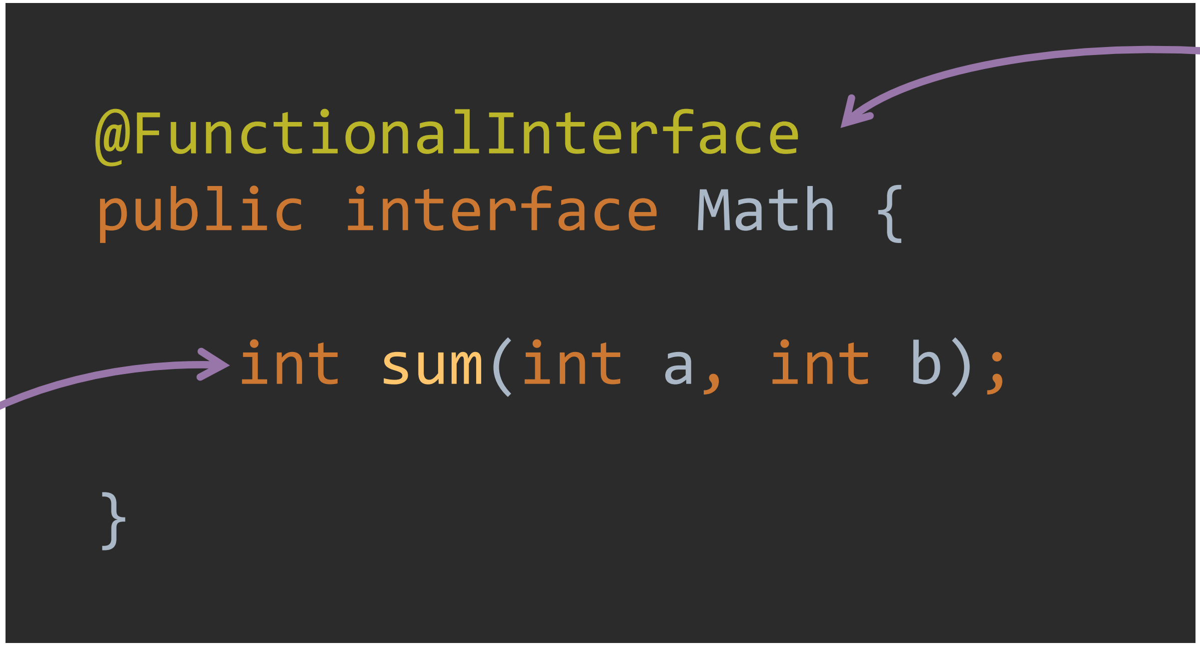
A code to be executed – similar to the method body

FUNCTIONAL INTERFACE

- An interface that has only one abstract method is called a functional interface

- Java provides an annotation @FunctionalInterface, which is used to declare a functional interface

FUNCTIONAL INTERFACE EXAMPLE

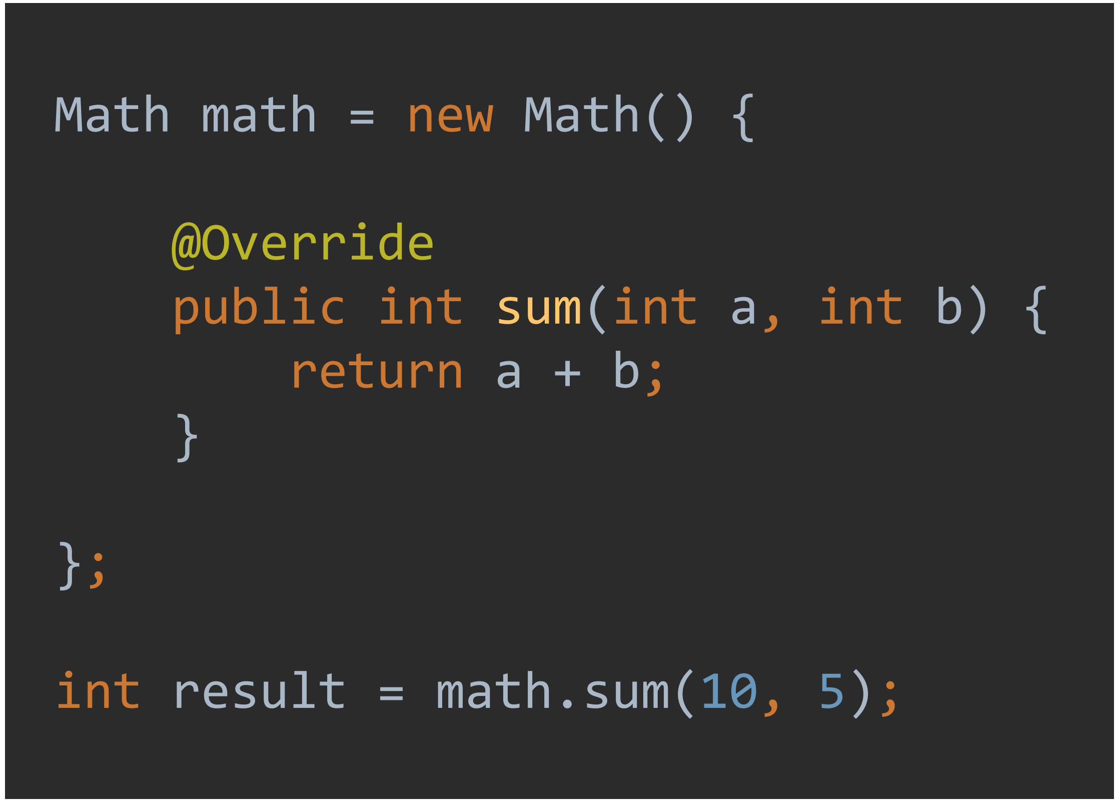


```
@FunctionalInterface
public interface Math {

    int sum(int a, int b);

}
```

Functional Interface annotation

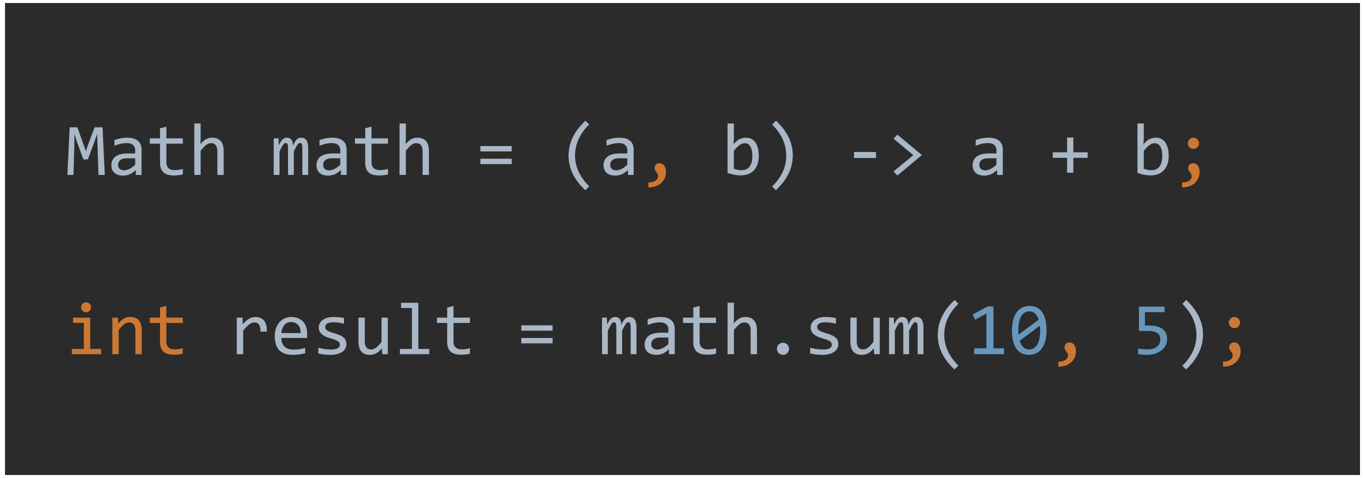
Has one and only one method

IMPLEMENTATION AS ANONYMOUS CLASS



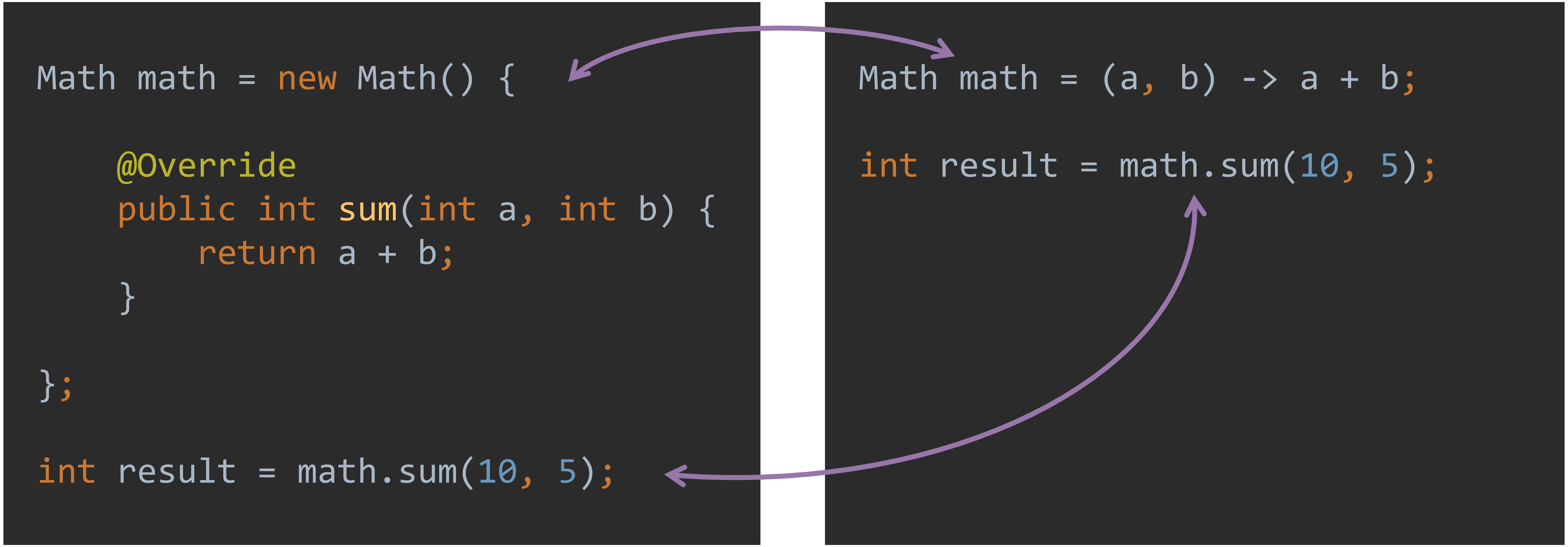
```java
Math math = new Math() {

    @Override
    public int sum(int a, int b) {
        return a + b;
    }

};

int result = math.sum(10, 5);
```

IMPLEMENTATION AS LAMBDA EXPRESSION



```
Math math = (a, b) -> a + b;

int result = math.sum(10, 5);
```

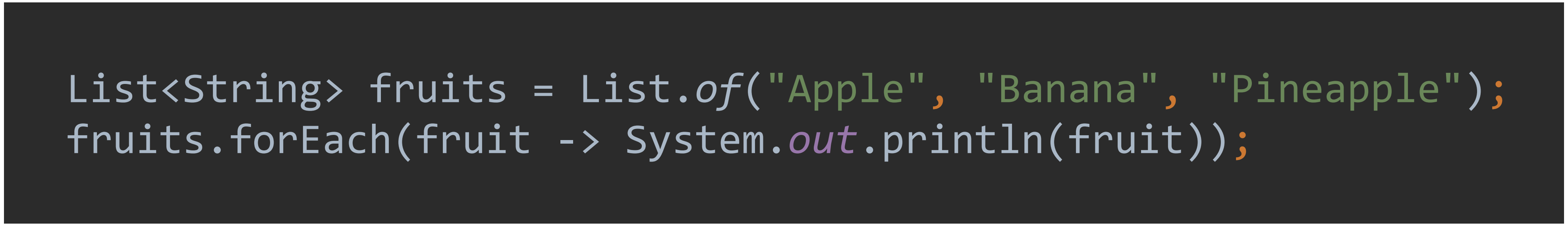
IMPLEMENTATION SIDE BY SIDE



```java
Math math = new Math() {

    @Override
    public int sum(int a, int b) {
        return a + b;
    }

};

int result = math.sum(10, 5);
```

```java
Math math = (a, b) -> a + b;

int result = math.sum(10, 5);
```

FOR EACH USING LAMBDA EXPRESSION

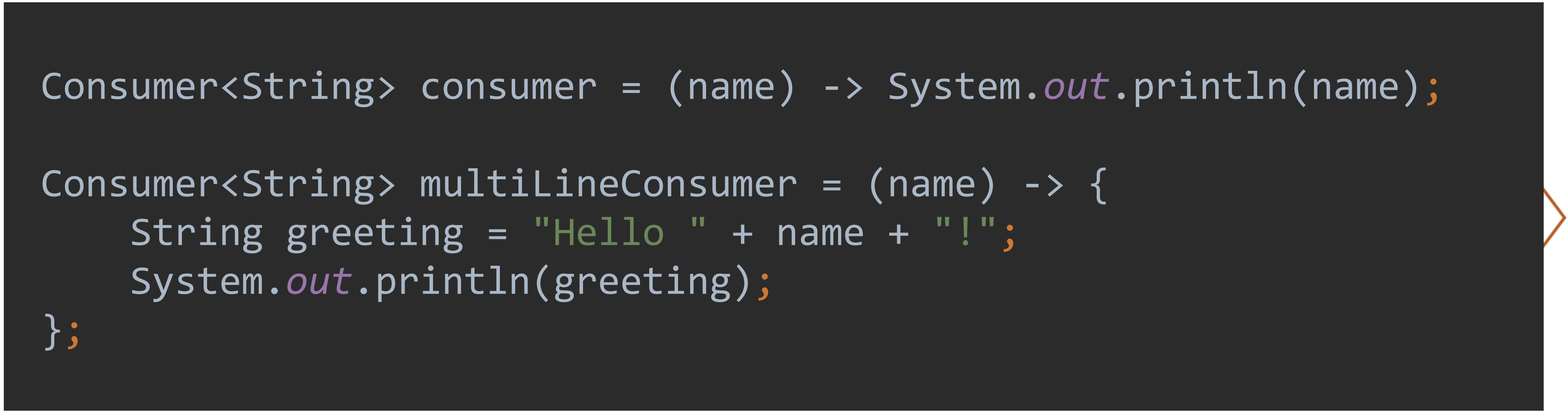


```java
List<String> fruits = List.of("Apple", "Banana", "Pineapple");
fruits.forEach(fruit -> System.out.println(fruit));
```

CORE FUNCTIONAL INTERFACES

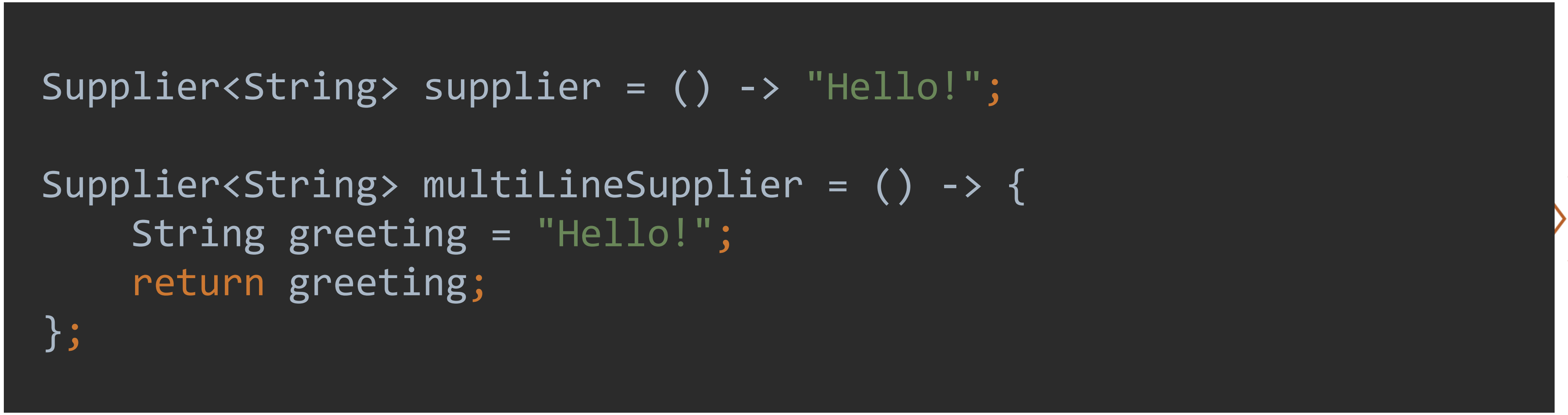
Interface Name	Purpose
Consumer	Represents an operation that accepts a single input argument and returns no result
Supplier	Represents an operation that supplies results
Function	Represents a function that accepts one argument and produces a result
Predicate	Represents a predicate (boolean-valued function) of one argument
BiFunction	Represents a function that accepts two arguments and produces a result. This is the two-arity specialization of Function.

CONSUMER EXAMPLES



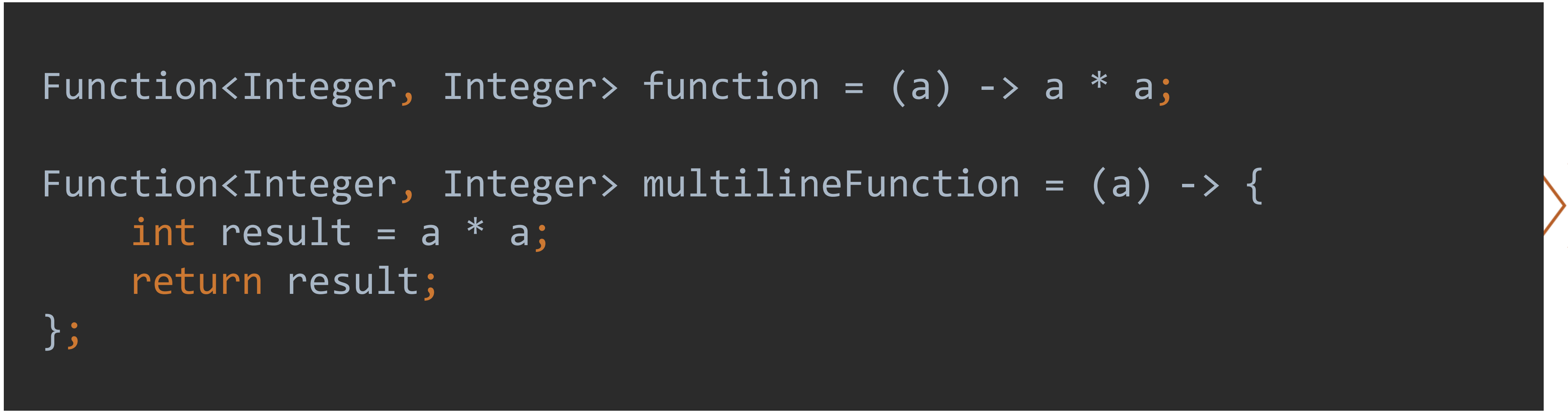
```java
Consumer<String> consumer = (name) -> System.out.println(name);

Consumer<String> multiLineConsumer = (name) -> {
    String greeting = "Hello " + name + "!";
    System.out.println(greeting);
};
```

SUPPLIER EXAMPLES



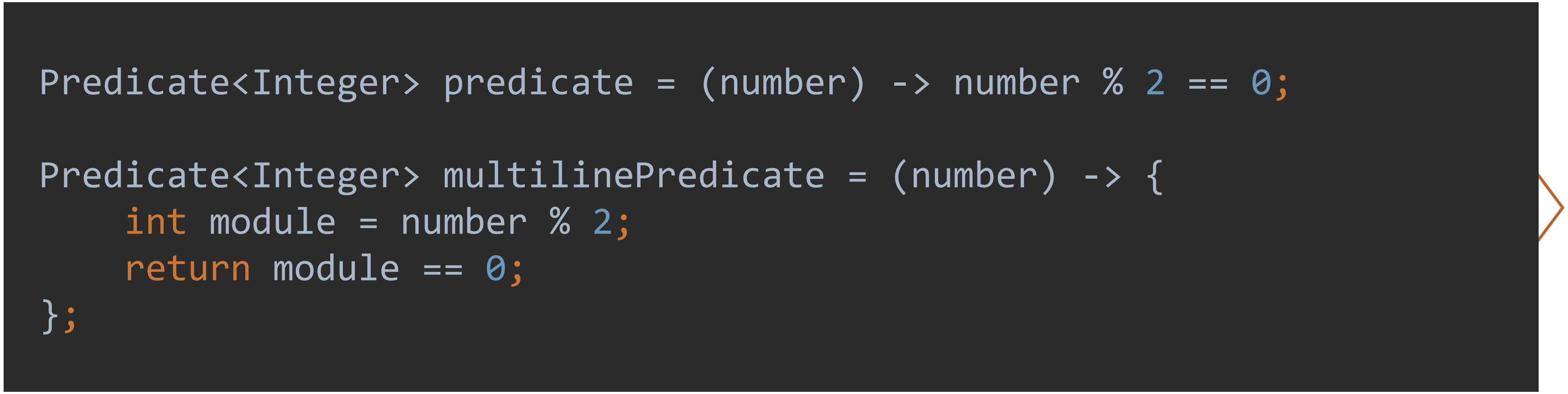
```java
Supplier<String> supplier = () -> "Hello!";

Supplier<String> multiLineSupplier = () -> {
    String greeting = "Hello!";
    return greeting;
};
```

FUNCTION EXAMPLES



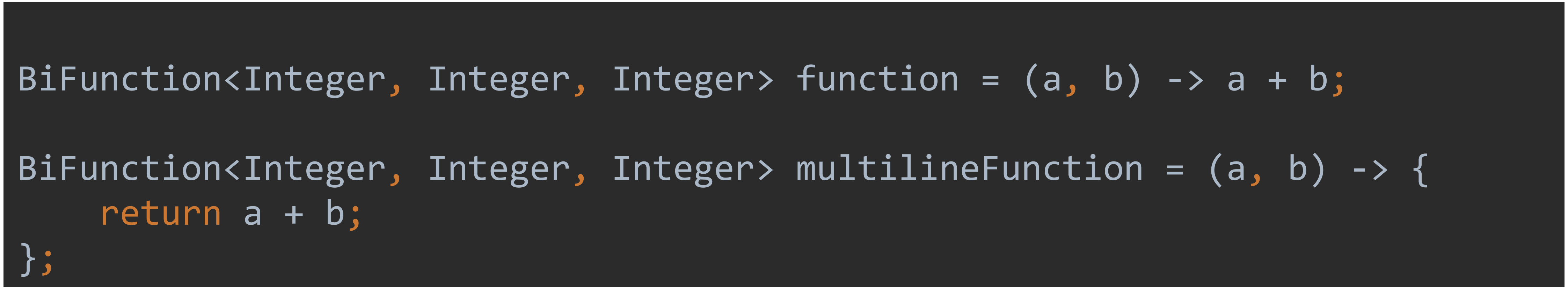
```java
Function<Integer, Integer> function = (a) -> a * a;

Function<Integer, Integer> multilineFunction = (a) -> {
    int result = a * a;
    return result;
};
```

PREDICATE EXAMPLES



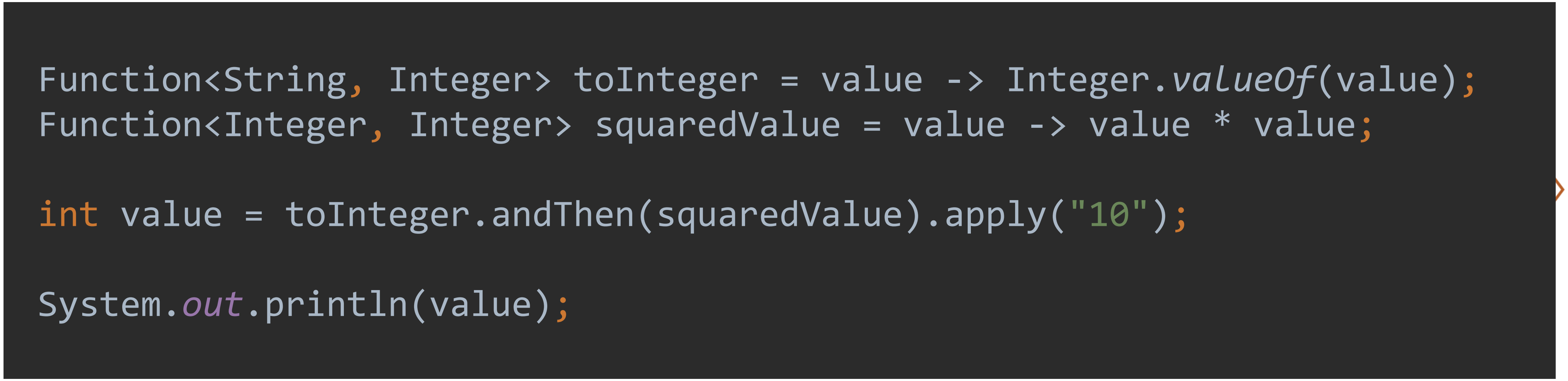
```java
Predicate<Integer> predicate = (number) -> number % 2 == 0;

Predicate<Integer> multilinePredicate = (number) -> {
    int module = number % 2;
    return module == 0;
};
```

BI-FUNCTION EXAMPLES



```java
BiFunction<Integer, Integer, Integer> function = (a, b) -> a + b;

BiFunction<Integer, Integer, Integer> multilineFunction = (a, b) -> {
    return a + b;
};
```

FUNCTION CHAINS



```java
Function<String, Integer> toInteger = value -> Integer.valueOf(value);
Function<Integer, Integer> squaredValue = value -> value * value;

int value = toInteger.andThen(squaredValue).apply("10");

System.out.println(value);
```

REFERENCES

REFERENCES

- https://www.javatpoint.com/java-lambda-expressions
- https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html
- https://hevodata.com/learn/java-lambda-expressions/
- https://www.baeldung.com/java-8-lambda-expressions-tips

QUESTIONS?

THANK YOU!