Java Programming

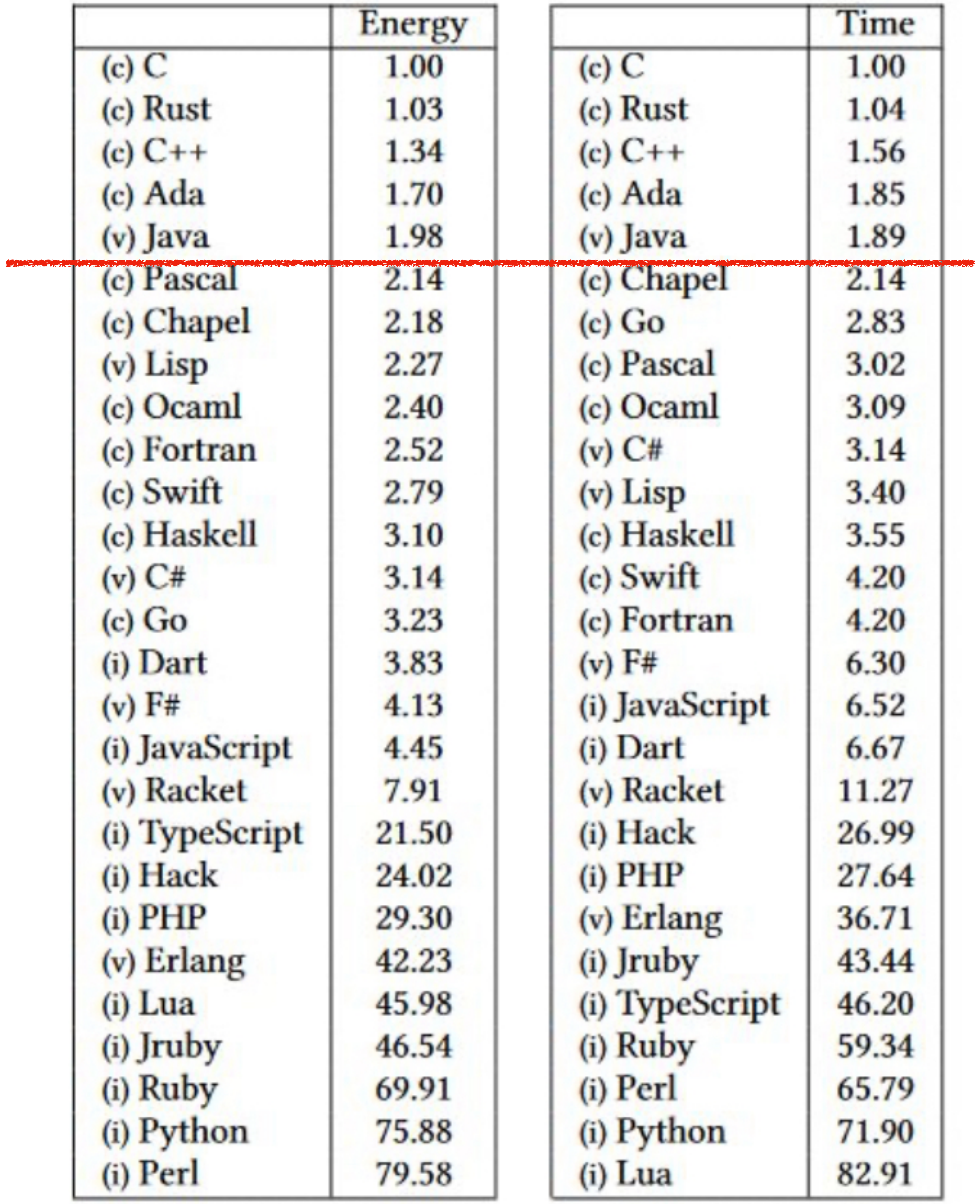
Fundamentals of Programming in Java

Ivo Vondrak, 2024

Why Java?

Because of this …

The table shows data from the paper by Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, João Saraiva, Ranking programming languages by energy efficiency, Science of Computer Programming, 2021



	Energy
(c) C	1.00
(c) Rust	1.03
(c) C++	1.34
(c) Ada	1.70
(v) Java	1.98
(c) Pascal	2.14
(c) Chapel	2.18
(v) Lisp	2.27
(c) Ocaml	2.40
(c) Fortran	2.52
(c) Swift	2.79
(c) Haskell	3.10
(v) C#	3.14
(c) Go	3.23
(i) Dart	3.83
(v) F#	4.13
(i) JavaScript	4.45
(v) Racket	7.91
(i) TypeScript	21.50
(i) Hack	24.02
(i) PHP	29.30
(v) Erlang	42.23
(i) Lua	45.98
(i) Jruby	46.54
(i) Ruby	69.91
(i) Python	75.88
(i) Perl	79.58

	Time
(c) C	1.00
(c) Rust	1.04
(c) C++	1.56
(c) Ada	1.85
(v) Java	1.89
(c) Chapel	2.14
(c) Go	2.83
(c) Pascal	3.02
(c) Ocaml	3.09
(v) C#	3.14
(v) Lisp	3.40
(c) Haskell	3.55
(c) Swift	4.20
(c) Fortran	4.20
(v) F#	6.30
(i) JavaScript	6.52
(i) Dart	6.67
(v) Racket	11.27
(i) Hack	26.99
(i) PHP	27.64
(v) Erlang	36.71
(i) Jruby	43.44
(i) TypeScript	46.20
(i) Ruby	59.34
(i) Perl	65.79
(i) Python	71.90
(i) Lua	82.91

References

- SCHILDT, Herbert, 2022. Java: A Beginner’s Guide, Ninth Edition. 9 edition. New York: McGraw-Hill Education. ISBN 978-1260463552.

- Oracle.The Java™ Tutorials. accessed September 29, 2022,http://docs.oracle.com/javase/tutorial/index.html.

- Oracle.The Java™ SE API. accessed September 29, 2022, http://docs.oracle.com/javase/17/docs/api/index.html

- BLOCH, Joshua, 2018. Effective Java. 3 edition. Boston: Addison-Wesley Professional. ISBN 978-0-13-468599-1.

- SCHILDT, Herbert, 2021. Java: The Complete Reference, Twelfth Edition. 12 edition. New York: McGraw-Hill Education. ISBN 978-1260463415.

- BARNES. Objects First with Java: A Practical Introduction Using BlueJ, Global Edition. 6th edition. Boston: Pearson, 2016. ISBN 978-1-292-15904-1.

1st Part: Basics

- Java as a Technology
- Java program structure
- Variables, types, operators
- Objects and classes
- Structure of classes, enumerations
- Java control structures
- Arrays
- Methods with a variable number of arguments

Java as a Technology

Features of Java

- Architecture Neutral and Portable
- Object Oriented
- Robust, Dynamic and Secure
- Multithreaded
- Distributed
- Simple language – core is API

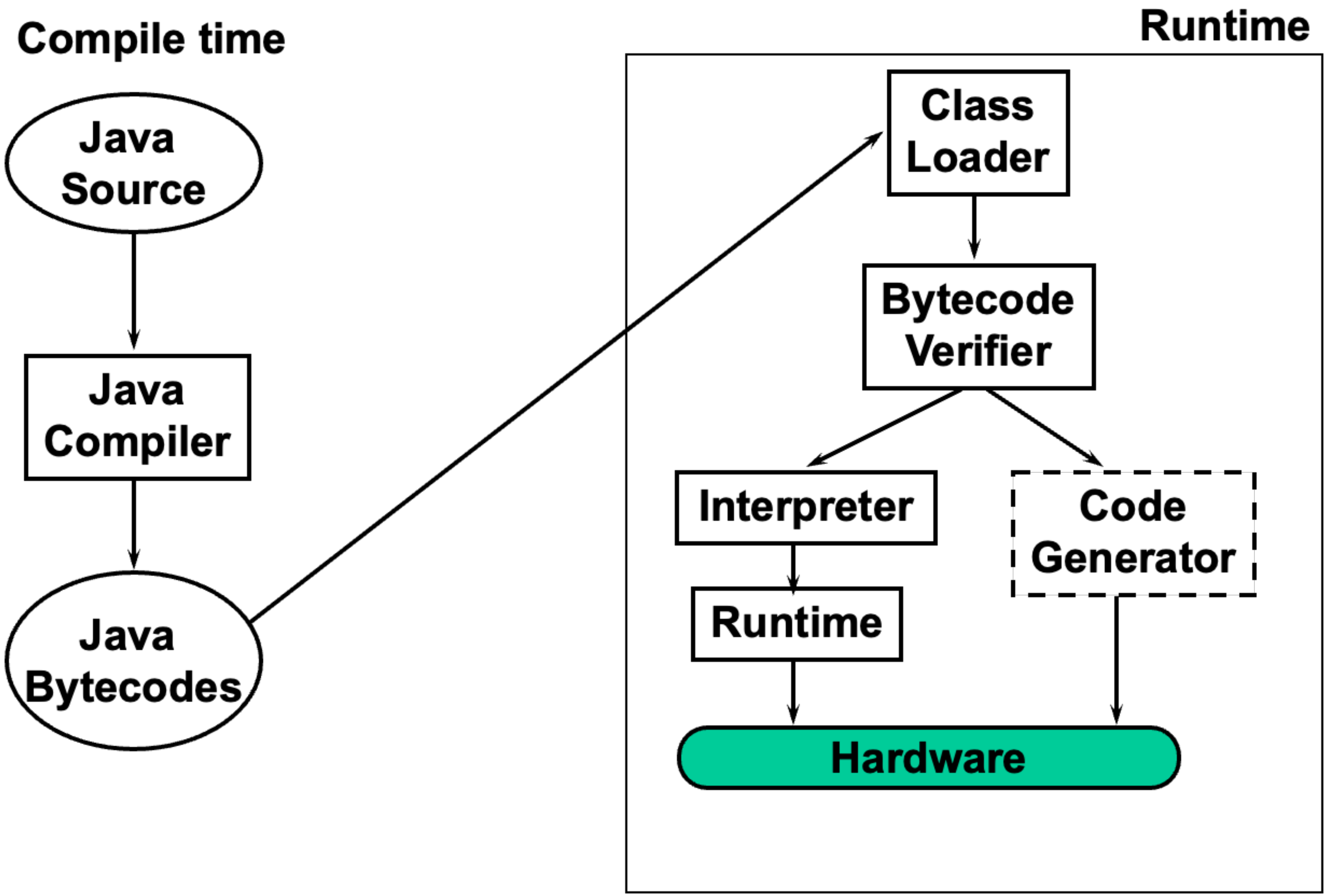
Architecture Neural

„Write once, run anywhere“ Sun Microsystems

- Java source code is “compiled” into **high-level, machine independent, Java Bytecode** (.class files) format.
    - packages java.io.*, java.util.*, java.awt.*
- **Java Virtual Machine** is an imaginary machine that is implemented by emulating it in software on a real machine.
    - JVM specification provides concrete definitions for implementation of instruction set, register set, class file format, stack ..

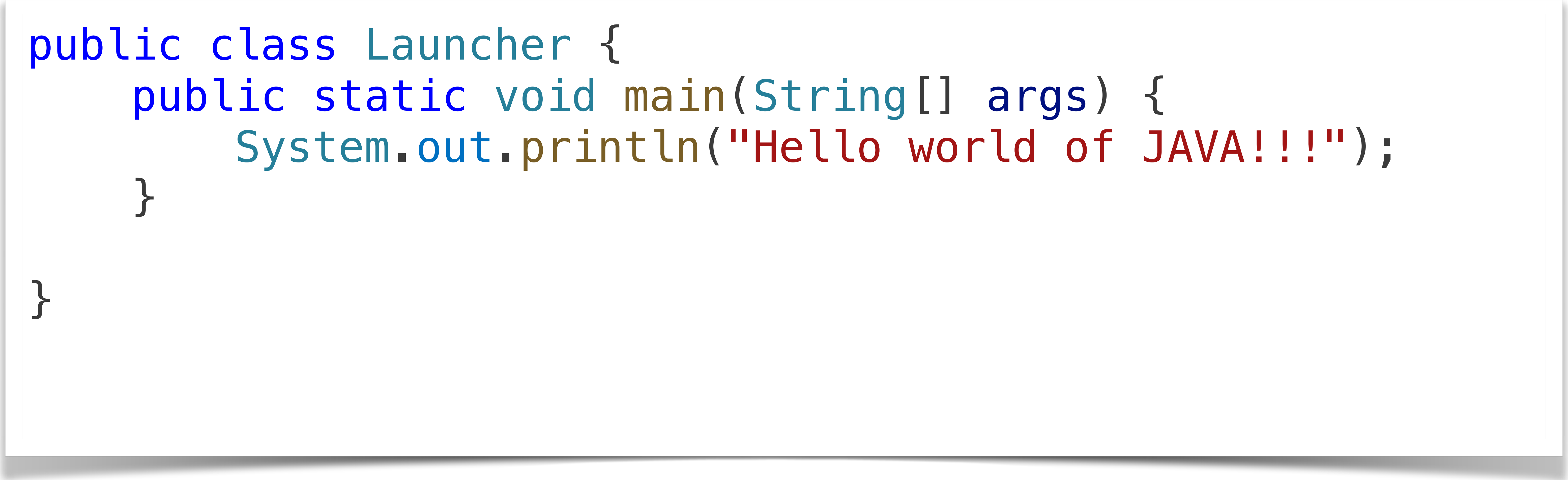
Compile Time and Runtime

Java program runs in a JVM

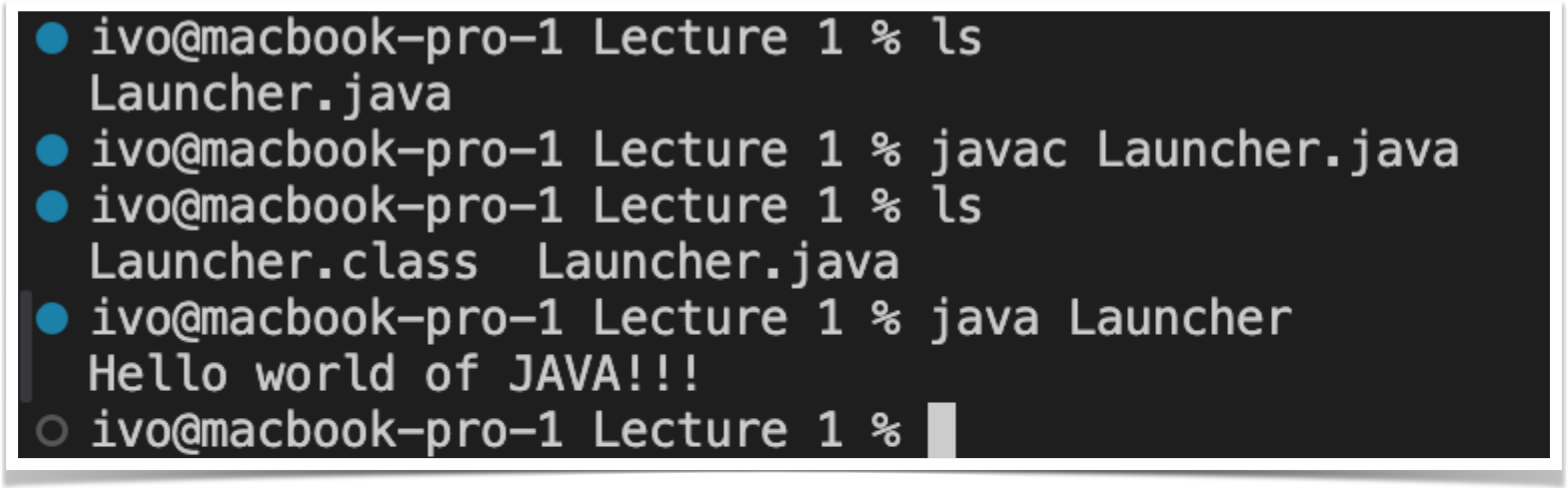


Java Program Structure

main as an entry point to an application



```
public class Launcher {
    public static void main(String[] args) {
        System.out.println("Hello world of JAVA!!!");
    }

}
```



```
● ivo@macbook-pro-1 Lecture 1 % ls
  Launcher.java
● ivo@macbook-pro-1 Lecture 1 % javac Launcher.java
● ivo@macbook-pro-1 Lecture 1 % ls
  Launcher.class  Launcher.java
● ivo@macbook-pro-1 Lecture 1 % java Launcher
  Hello world of JAVA!!!
○ ivo@macbook-pro-1 Lecture 1 % 
```

1. Source file Launcher.java compiled

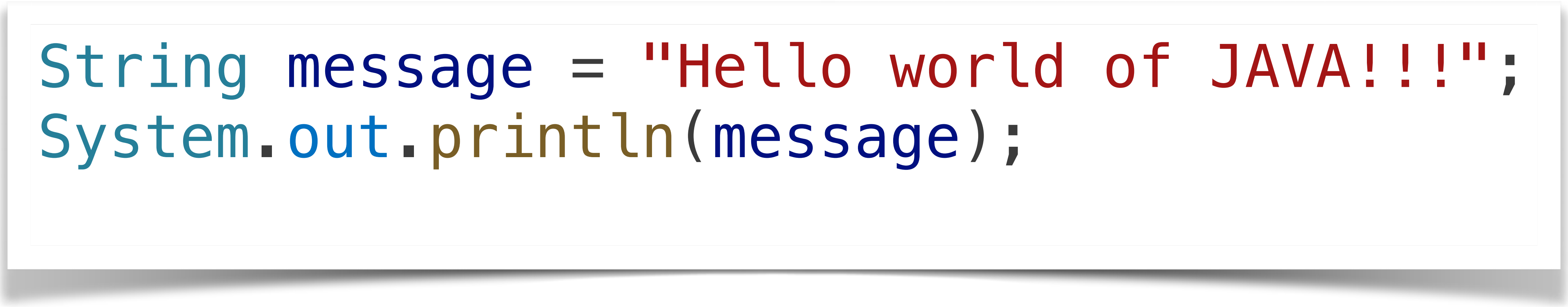
2. Launcher.class executed in JVM

Basic Java Constructs

Comments and Statements

- **Comments**
  // comment on one line
  /* comment on one or more lines */
  /** documenting comment, comment that should be included in any automatically generated documentation (the HTML files generated by the javadoc command **/

- **Statements** form the smallest executable unit in a program



```java
String message = "Hello world of JAVA!!!";
System.out.println(message);
```

Identifiers

Identifiers name variables, functions, classes, and objects - anything that programmers need to identify and use.

- ident
- nameOfSomething
- _name
- User_name1
- $alsoValid

Data Types

Data types refer to the classification of data that tells the compiler or interpreter how the programmer intends to use the data.

- **Primitive types** – only values
- **Object types** – reference to the instance of class:
    - types from Java (more than 18000) – e.g. String
    - defined by user – e.g. Rectangle

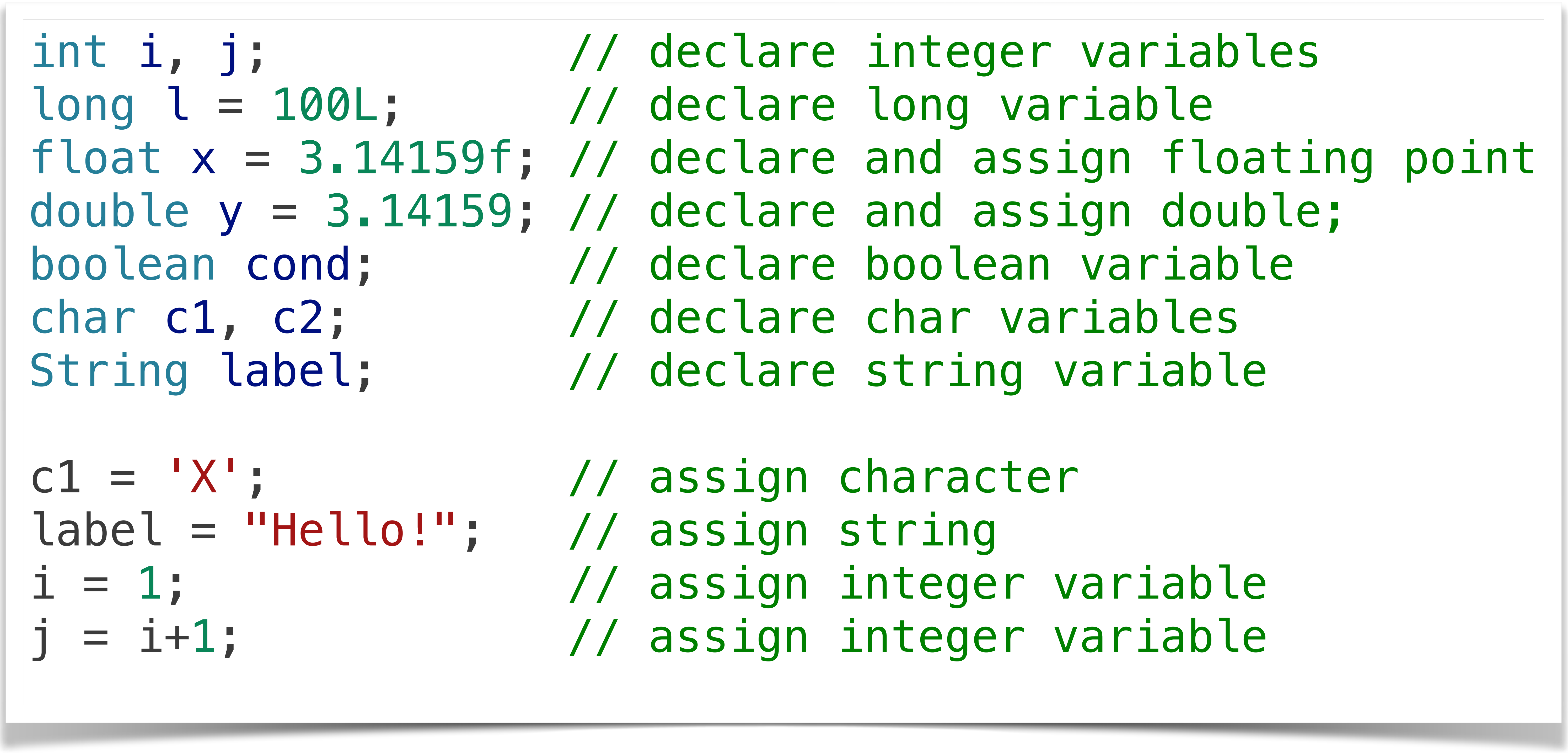
Primitive Data Types

Java uses five basic element types: boolean, character, integer, floating point, and string.

Type	Contains	Default	Size
boolean	true or false	false	1 bits
char	unicode character	\u0000	16 bits
byte	signed integer	0	8 bits
short	signed integer	0	16 bits
int	signed integer	0	32 bits
long	signed integer	0	64 bits
float	floating point	0.0	32 bits
double	floating point	0.0	64 bits
String	string of chars	null	?? bits

Declarations and Assignment

Different data types determine the kind of operations that can be performed on the data, how much space it occupies in memory, and how the bits representing the data are interpreted.



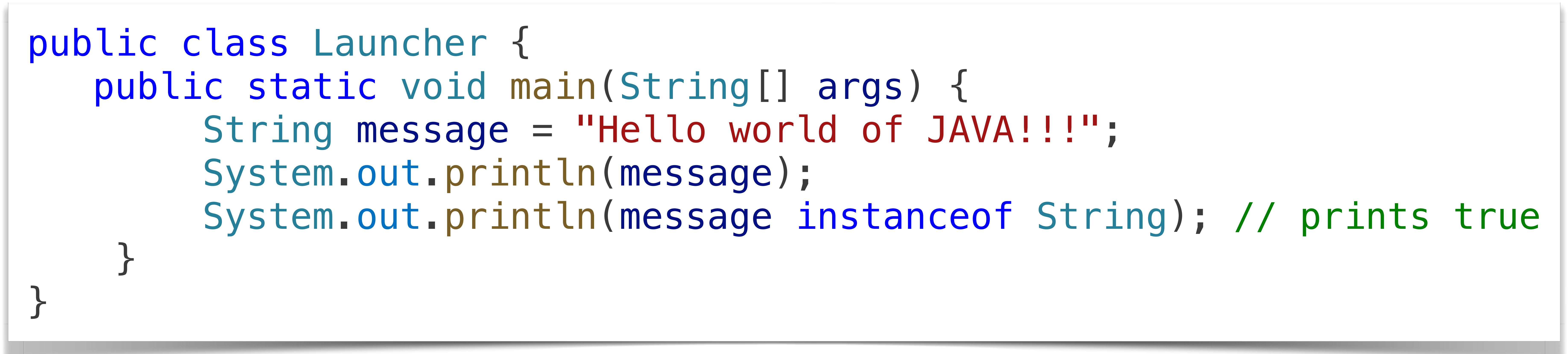
```
int i, j;           // declare integer variables
long l = 100L;      // declare long variable
float x = 3.14159f; // declare and assign floating point
double y = 3.14159; // declare and assign double;
boolean cond;       // declare boolean variable
char c1, c2;        // declare char variables
String label;       // declare string variable

c1 = 'X';           // assign character
label = "Hello!";   // assign string
i = 1;              // assign integer variable
j = i+1;            // assign integer variable
```

Operators

unary, binary, assignment, relational, logical, ternary, bitwise, cast

- Java support **almost all of the standard C operators:**
  =   >   <   !   ~   ?:
  ==   <=   >=   !=   && || ++   --
  +   -   *   /   & | ^   %   <<   >>   >>>
  +=   -=   *=   /=   &= |=   ^=   %=   <<=   >>=   >>>=

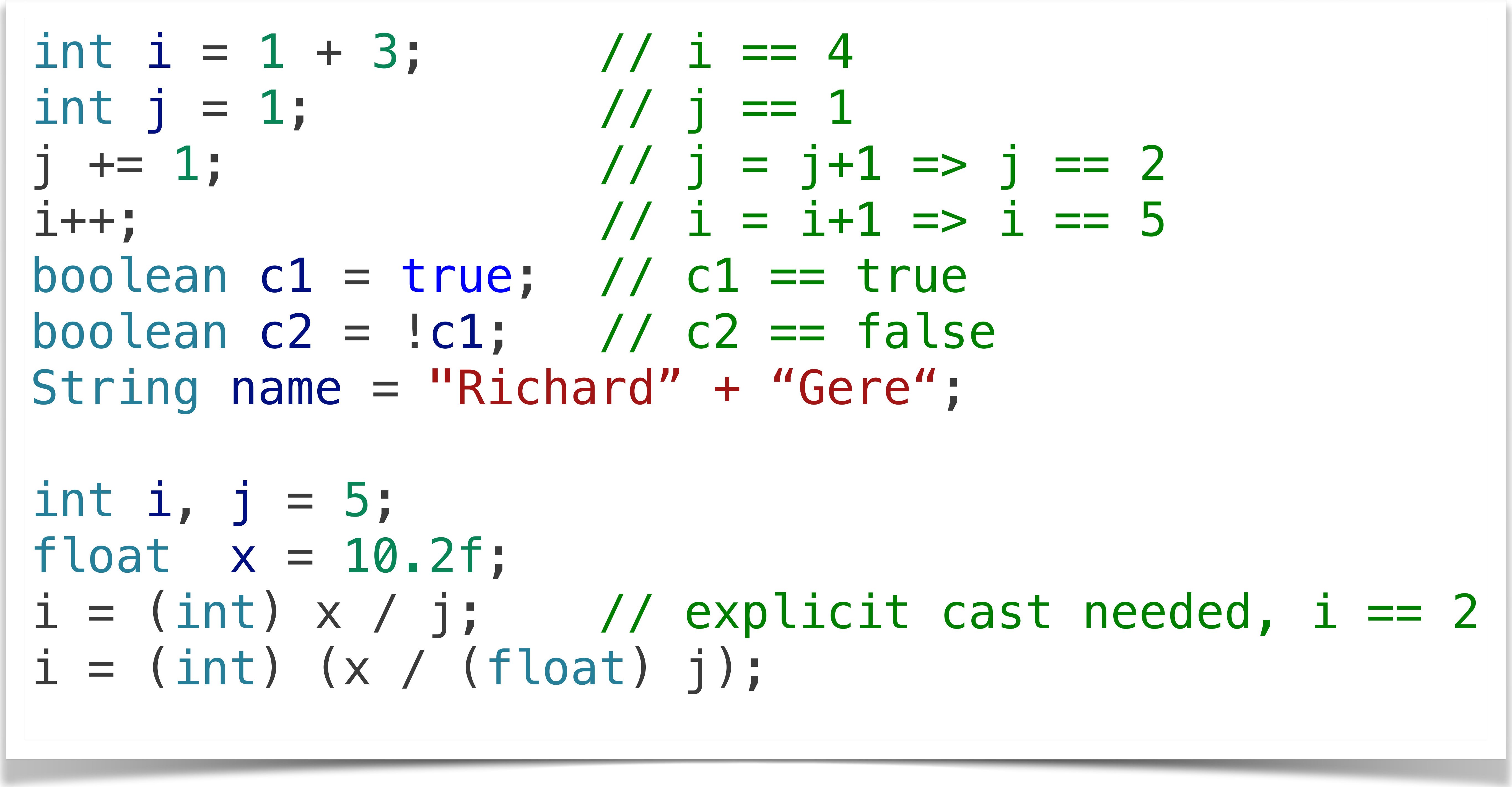
- Operator **instanceof** returns true if the object on the left-hand side is an instance of type specified on its right side.



```
public class Launcher {
   public static void main(String[] args) {
        String message = "Hello world of JAVA!!!";
        System.out.println(message);
        System.out.println(message instanceof String); // prints true
    }
}
```

Using Operators

Just some examples ...



```
int i = 1 + 3;      // i == 4
int j = 1;          // j == 1
j += 1;             // j = j+1 => j == 2
i++;                // i = i+1 => i == 5
boolean c1 = true;  // c1 == true
boolean c2 = !c1;   // c2 == false
String name = "Richard” + “Gere“;

int i, j = 5;
float  x = 10.2f;
i = (int) x / j;    // explicit cast needed, i == 2
i = (int) (x / (float) j);
```

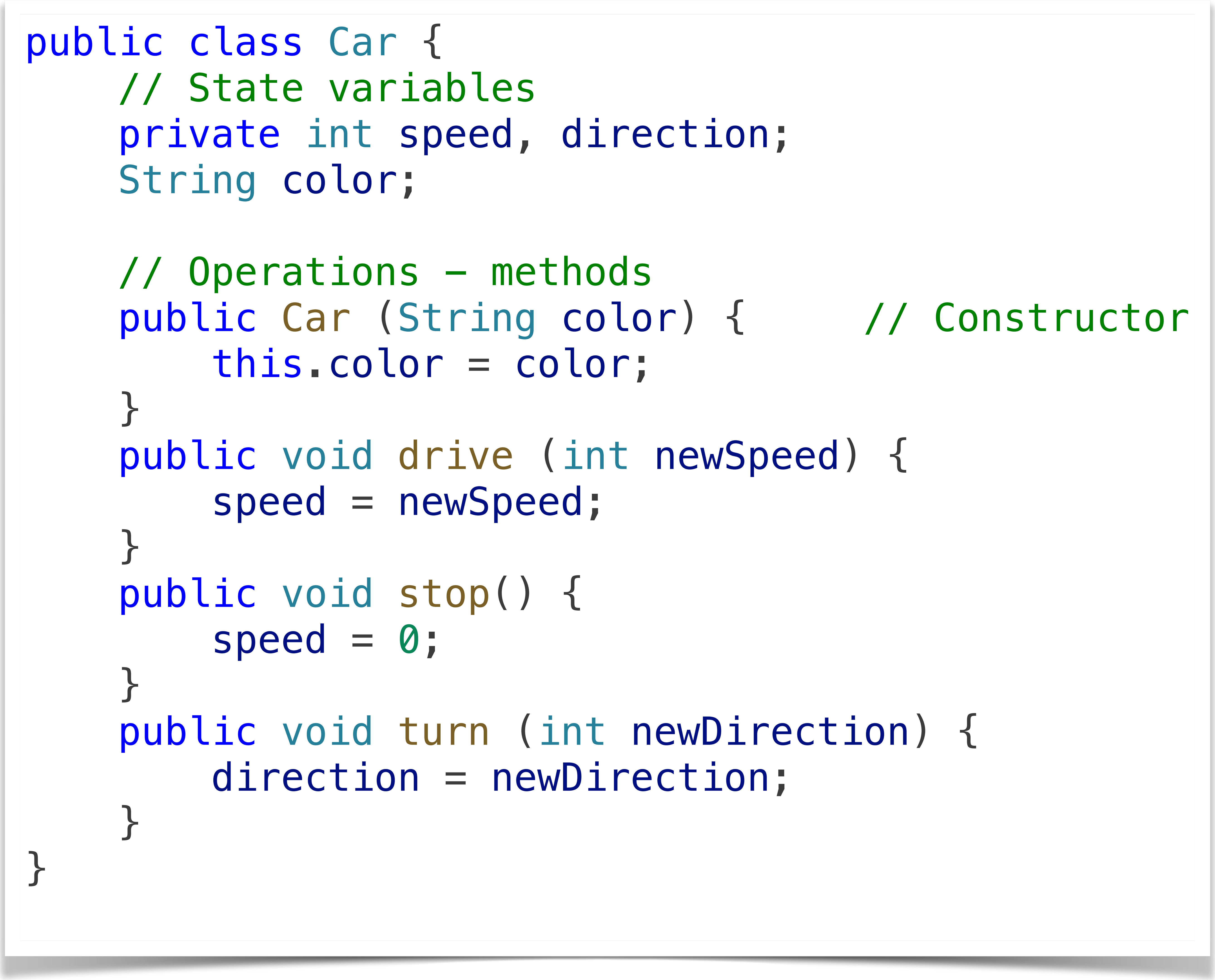
Object Type

Objects and classes

- **Object** is an distinguishable entity that has:
    - **Identity**: an uniqueness which distinguishes it from all other objects
    - **Behavior**: services it provides to another objects
    - **State**: value of attributes held by an object
- **Class** is an abstraction of objects with similar implementation
    - Class is definition of set of similar objects
    - Every object is an instance of one class

Class Definition

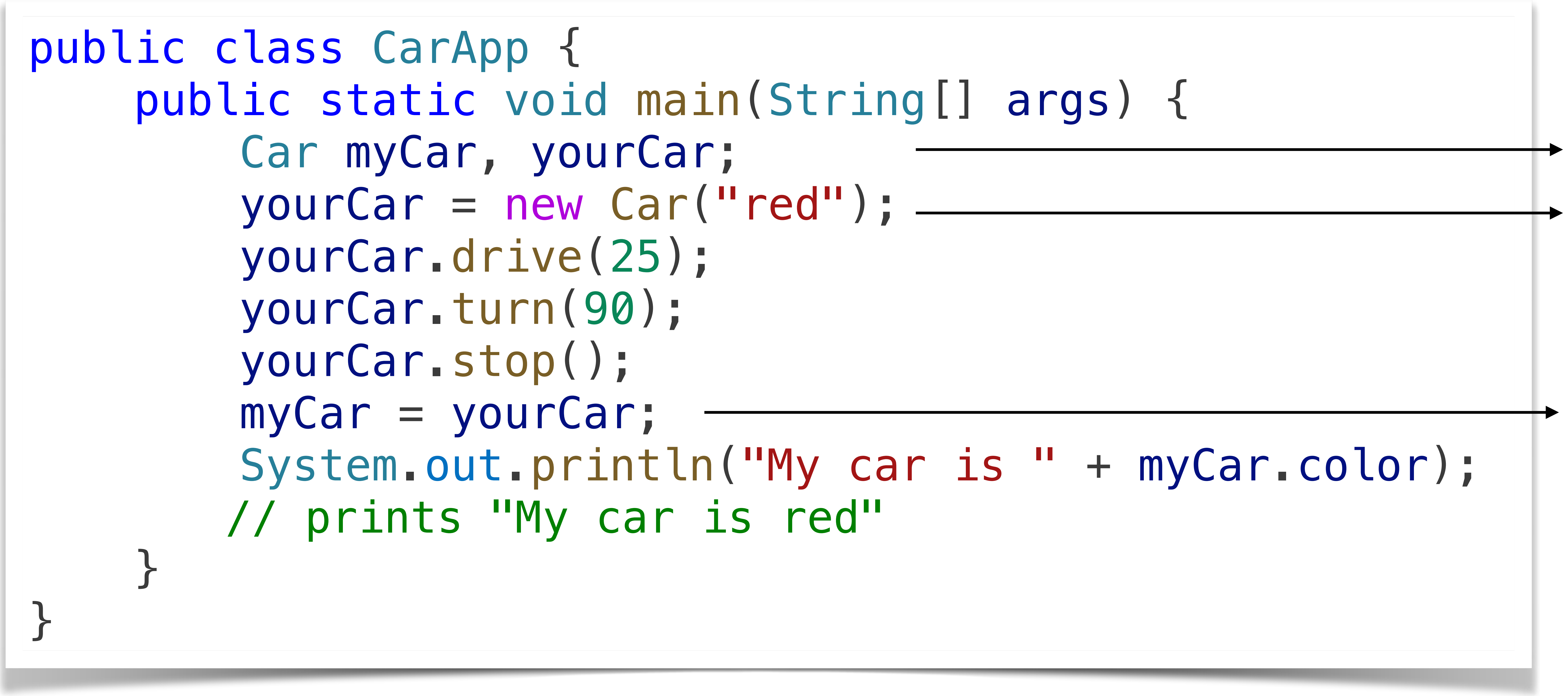
Class definition of a car



```
public class Car {
    // State variables
    private int speed, direction;
    String color;

    // Operations - methods
    public Car (String color) {     // Constructor
        this.color = color;
    }
    public void drive (int newSpeed) {
        speed = newSpeed;
    }
    public void stop() {
        speed = 0;
    }
    public void turn (int newDirection) {
        direction = newDirection;
    }
}
```

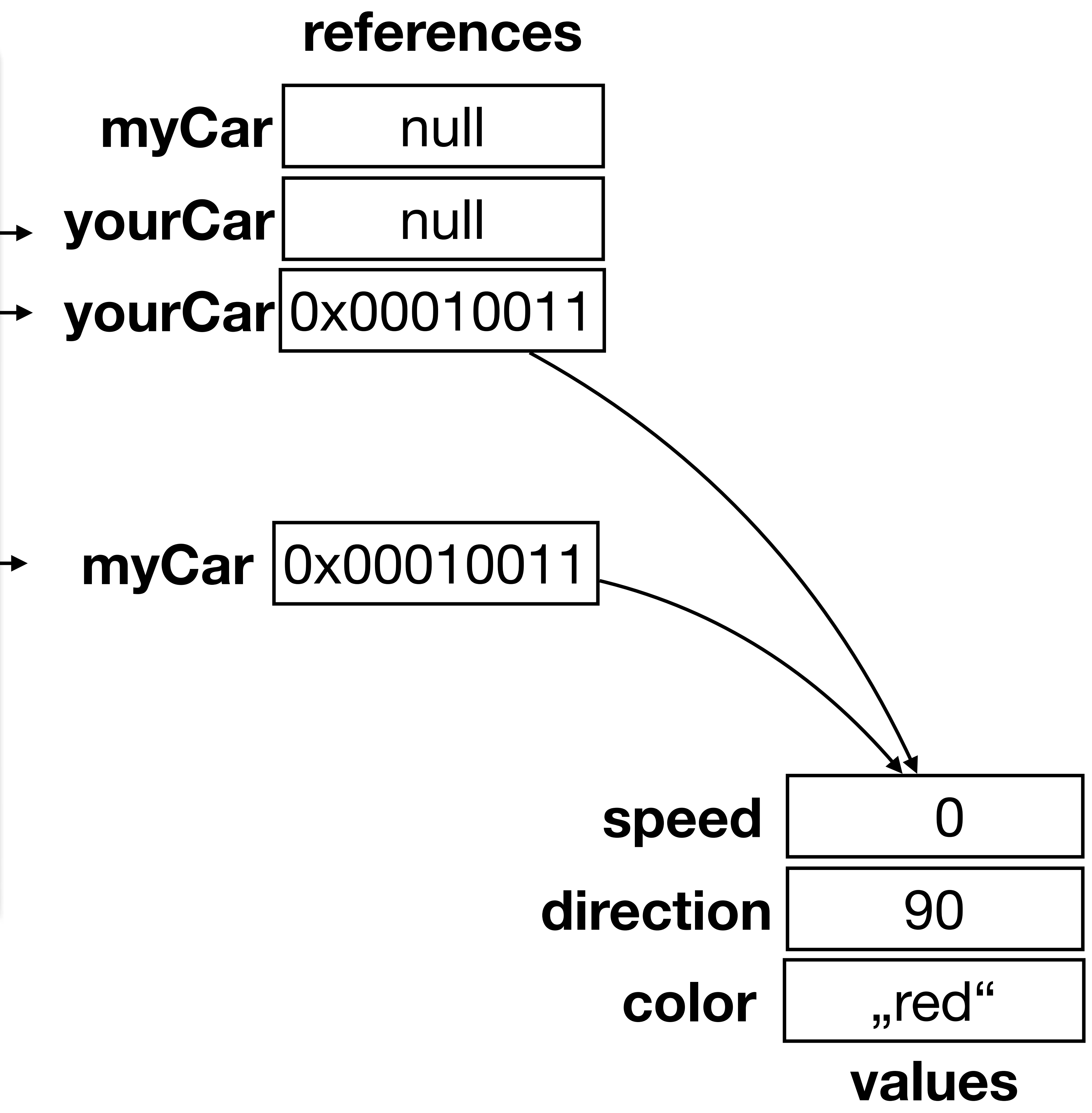
Creating and Using an Object

Object is an instance of a class



```
public class CarApp {
    public static void main(String[] args) {
        Car myCar, yourCar;
        yourCar = new Car("red");
        yourCar.drive(25);
        yourCar.turn(90);
        yourCar.stop();
        myCar = yourCar;
        System.out.println("My car is " + myCar.color);
       // prints "My car is red"
    }
}
```

In Memory



references

myCar null

yourCar null

yourCar 0x00010011

myCar 0x00010011

speed 0

direction 90

color „red“

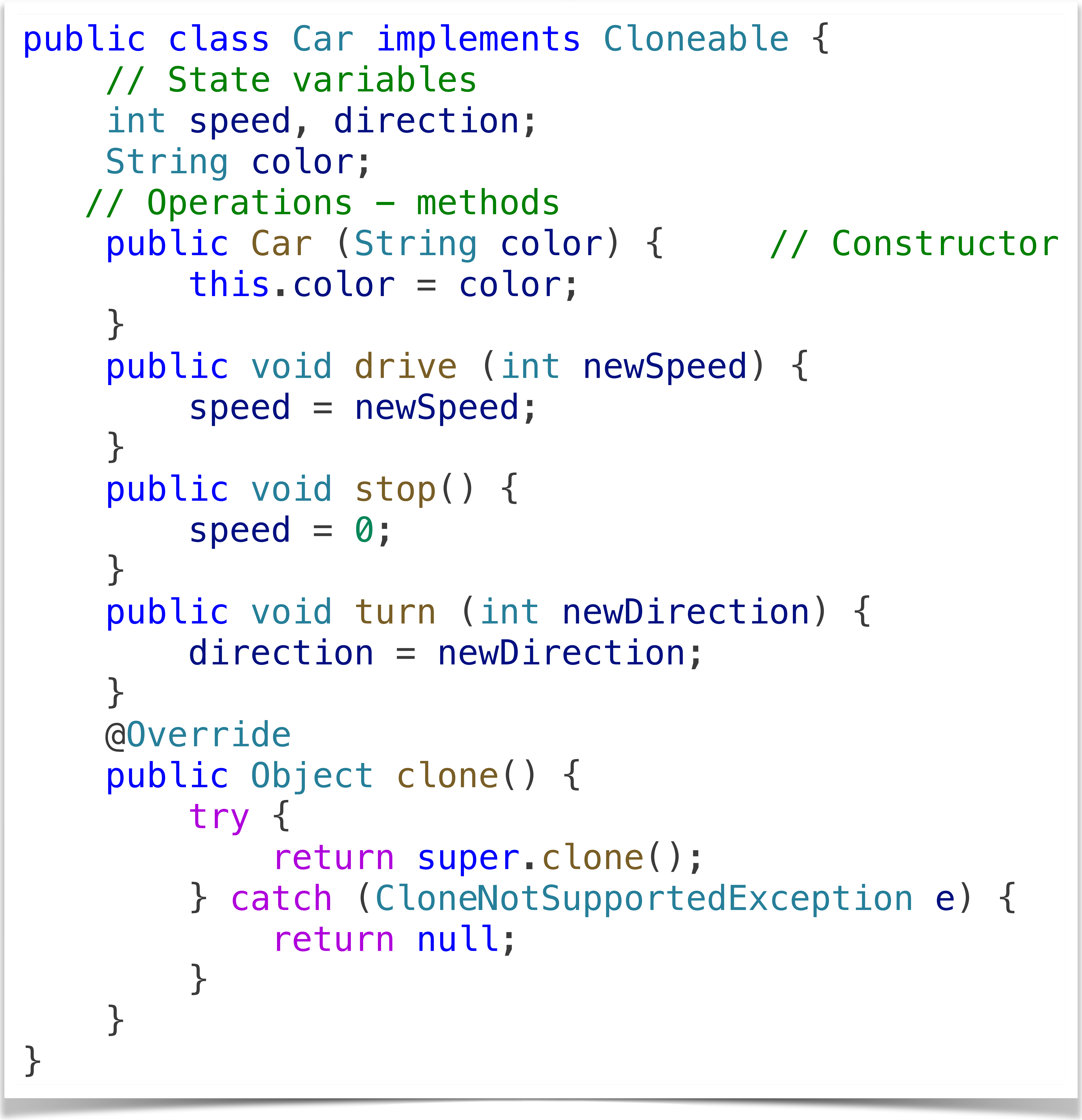
values

Copying Object

Object must be Cloneable

All these new things are
going to be explained later.

{



```java
public class Car implements Cloneable {
    // State variables
    int speed, direction;
    String color;
   // Operations – methods
    public Car (String color) {     // Constructor
        this.color = color;
    }
    public void drive (int newSpeed) {
        speed = newSpeed;
    }
    public void stop() {
        speed = 0;
    }
    public void turn (int newDirection) {
        direction = newDirection;
    }
    @Override
    public Object clone() {
        try {
            return super.clone();
        } catch (CloneNotSupportedException e) {
            return null;
        }
    }
}
```

Copied Objects Usage

To copy the object method clone() must be used.

In Memory

```
public class CarApp {
    public static void main(String[] args) {
        Car myCar, yourCar;
        yourCar = new Car("red");
        yourCar.drive(25);
        yourCar.turn(90);
        yourCar.stop();
        myCar = (Car) yourCar.clone();
        myCar.drive(100);
        System.out.println("My car speed is "+myCar.speed);
       // prints "My car speed is 100"
        System.out.println("Your car speed is  "+yourCar.speed);
       // prints „Your car speed is 0"
    }
}
```

references

myCar null

yourCar null

yourCar 0x00010011

myCar 0x00011011

speed	100
direction	90
color	„red“
	values

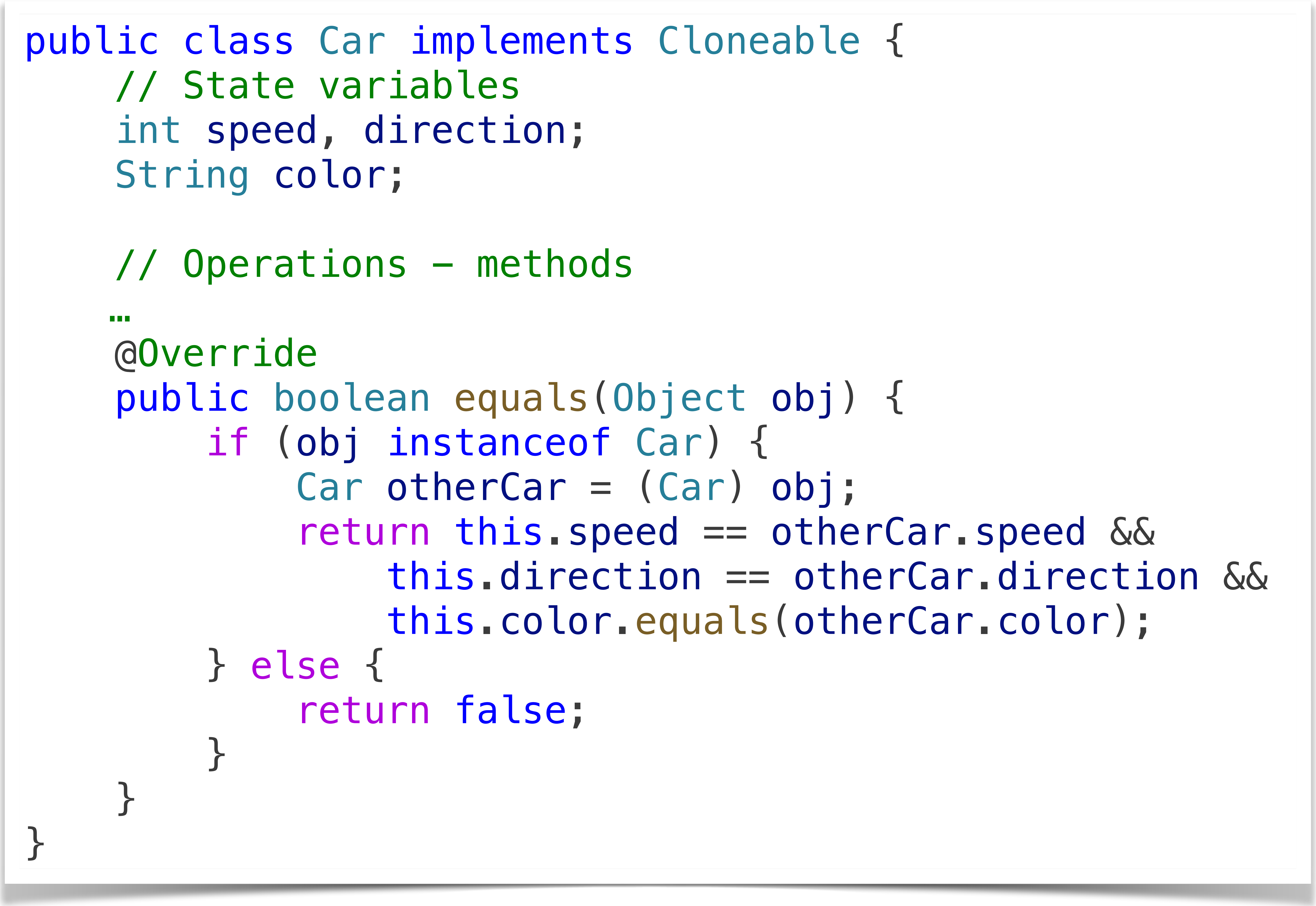
speed	0
direction	90
color	„red“
	values

Checking Objects for Equality

What objects are equal …

- Operator **==** tests whether two variables refer to the same object (identity), not whether two object contain the same values.

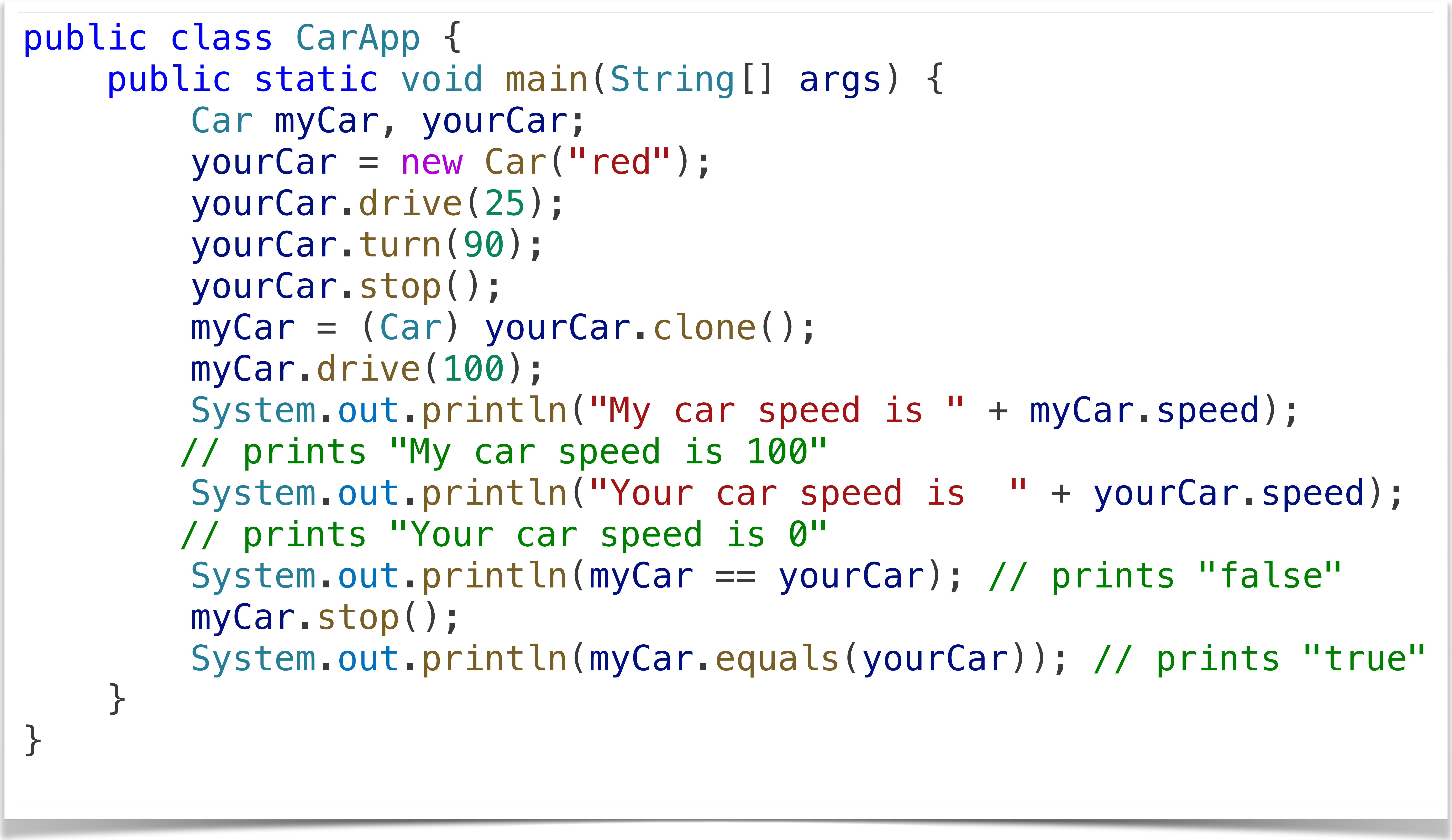
- In Java, number of classes define an method **equals()** **that compares containment (state) of objects.**



```
public class Car implements Cloneable {
    // State variables
    int speed, direction;
    String color;

    // Operations - methods
    …
    @Override
    public boolean equals(Object obj) {
        if (obj instanceof Car) {
            Car otherCar = (Car) obj;
            return this.speed == otherCar.speed &&
                this.direction == otherCar.direction &&
                this.color.equals(otherCar.color);
        } else {
            return false;
        }
    }
}
```

Equality Checking

Different cars with the equal state …

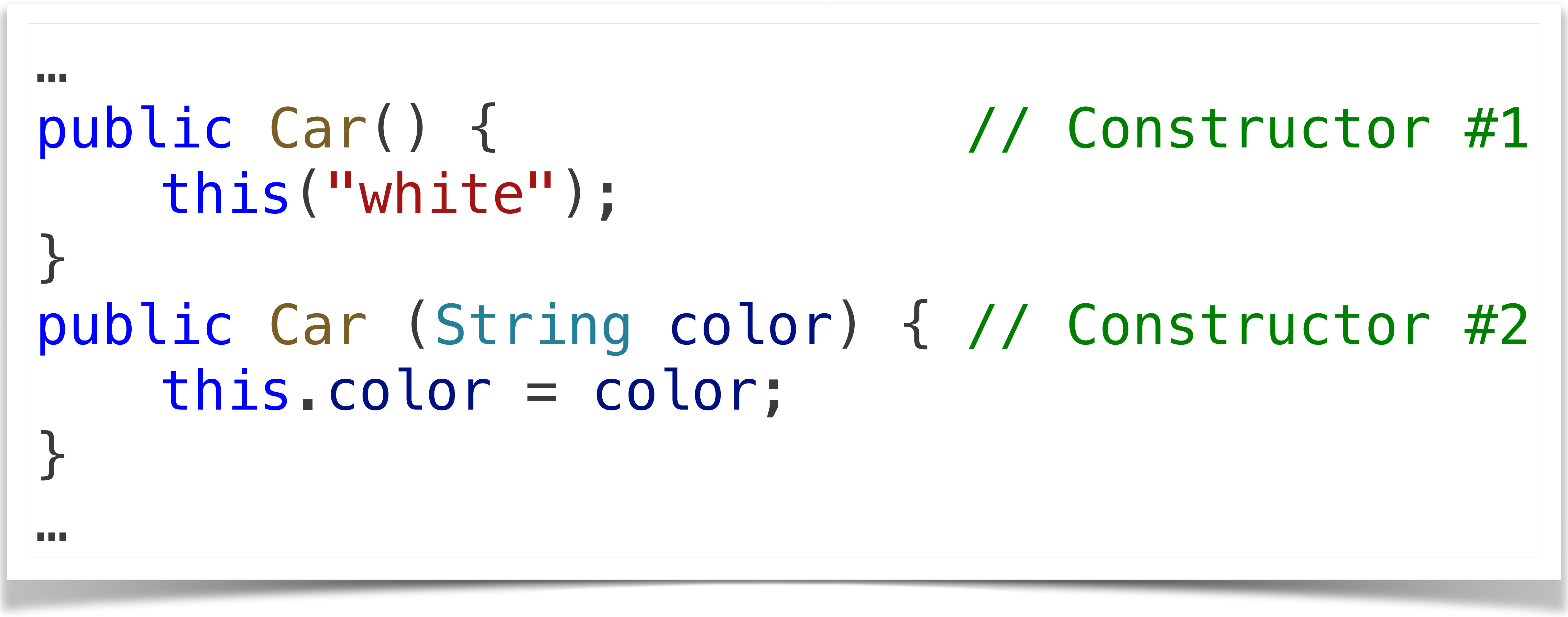


```
public class CarApp {
  public static void main(String[] args) {
    Car myCar, yourCar;
    yourCar = new Car("red");
    yourCar.drive(25);
    yourCar.turn(90);
    yourCar.stop();
    myCar = (Car) yourCar.clone();
    myCar.drive(100);
    System.out.println("My car speed is " + myCar.speed);
    // prints "My car speed is 100"
    System.out.println("Your car speed is  " + yourCar.speed);
    // prints "Your car speed is 0"
    System.out.println(myCar == yourCar); // prints "false"
    myCar.stop();
    System.out.println(myCar.equals(yourCar)); // prints "true"
  }
}
```

Overloading Constructors

You can write more than one constructor in a class.

- Each overloaded constructor is named the same.
- But they differ in any of the following ways:
  - Number of parameters.
  - Types of parameters.
  - Ordering of parameters.



```
…
public Car() {                   // Constructor #1
    this("white");
}
public Car (String color) { // Constructor #2
    this.color = color;
}
…
```

Overloading Methods

Any method can be overloaded same way as constructors

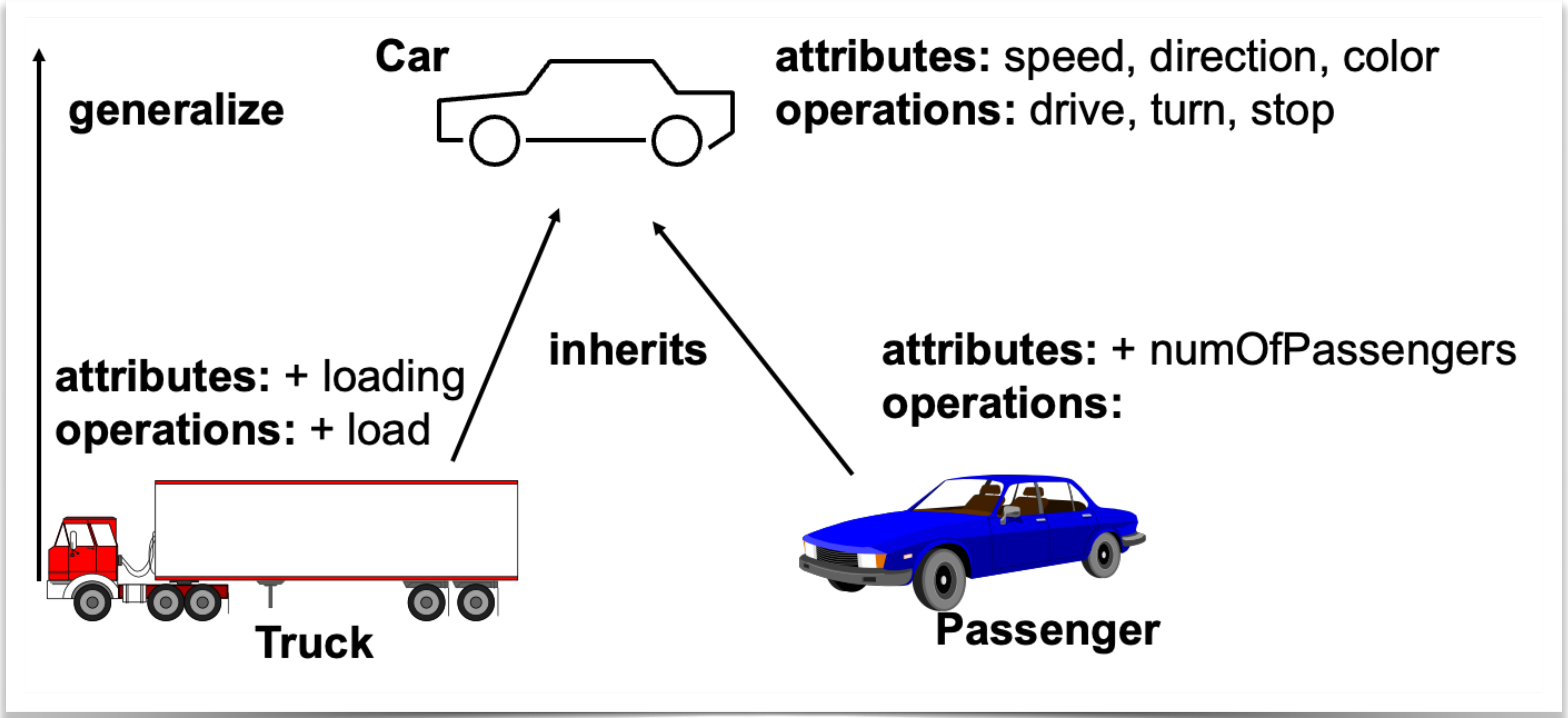
- All versions of an overloaded method are **named the same.**
- But differ in any of the following ways (in a **signature** of the method):
  - **Number of parameters**
  - **Types of parameters**
  - **Ordering of parameters**
- The method signature does not include ...
  - **Name of parameters**
  - **Method return type**

Generalization and Inheritance

Taxonomy is a method of categorizing and organizing entities into groups based on shared characteristics

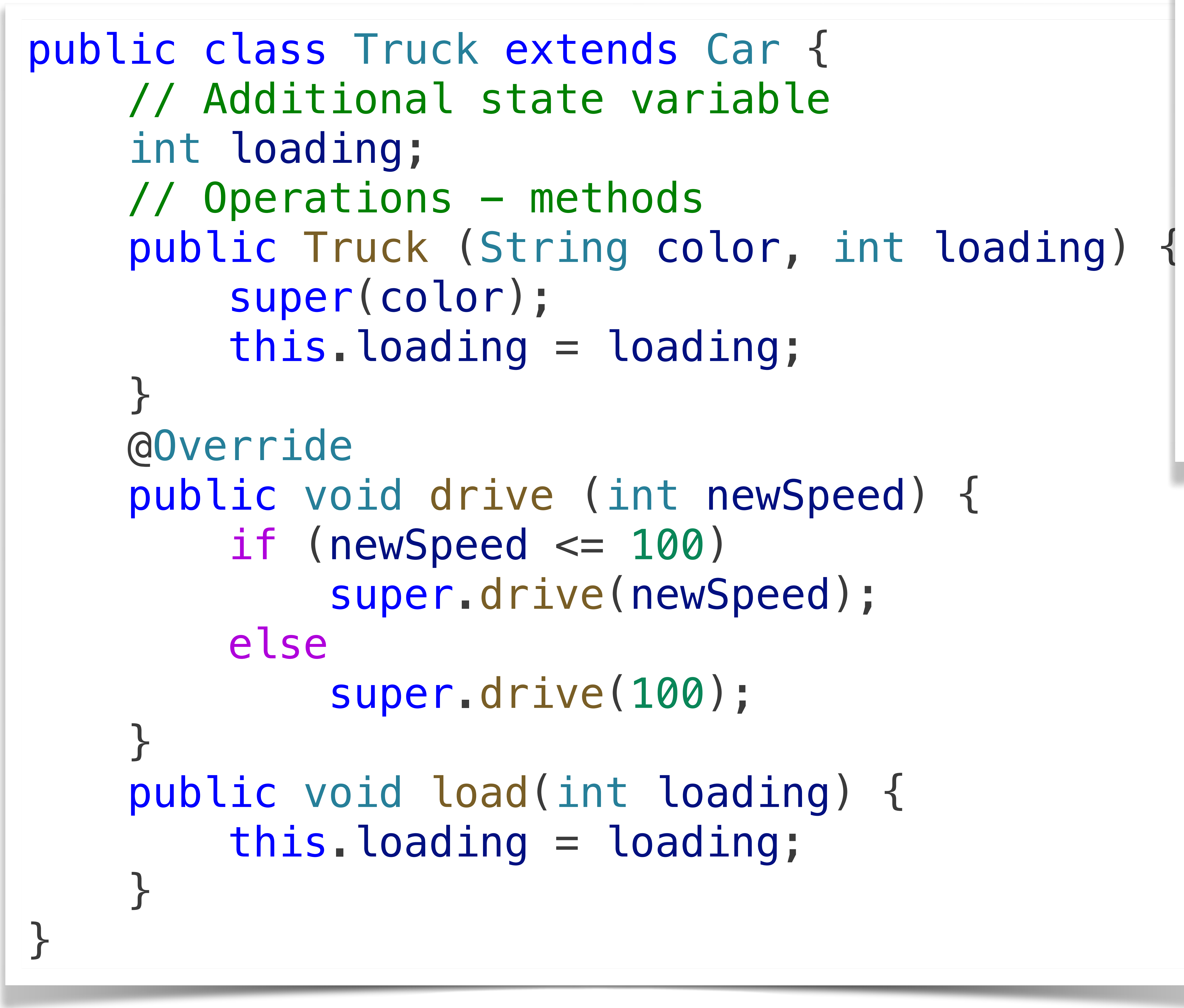
- **Generalization** is the relationship between a class and one or more refined versions of it.

- **Inheritance** refers to the mechanism of sharing attributes and operations.

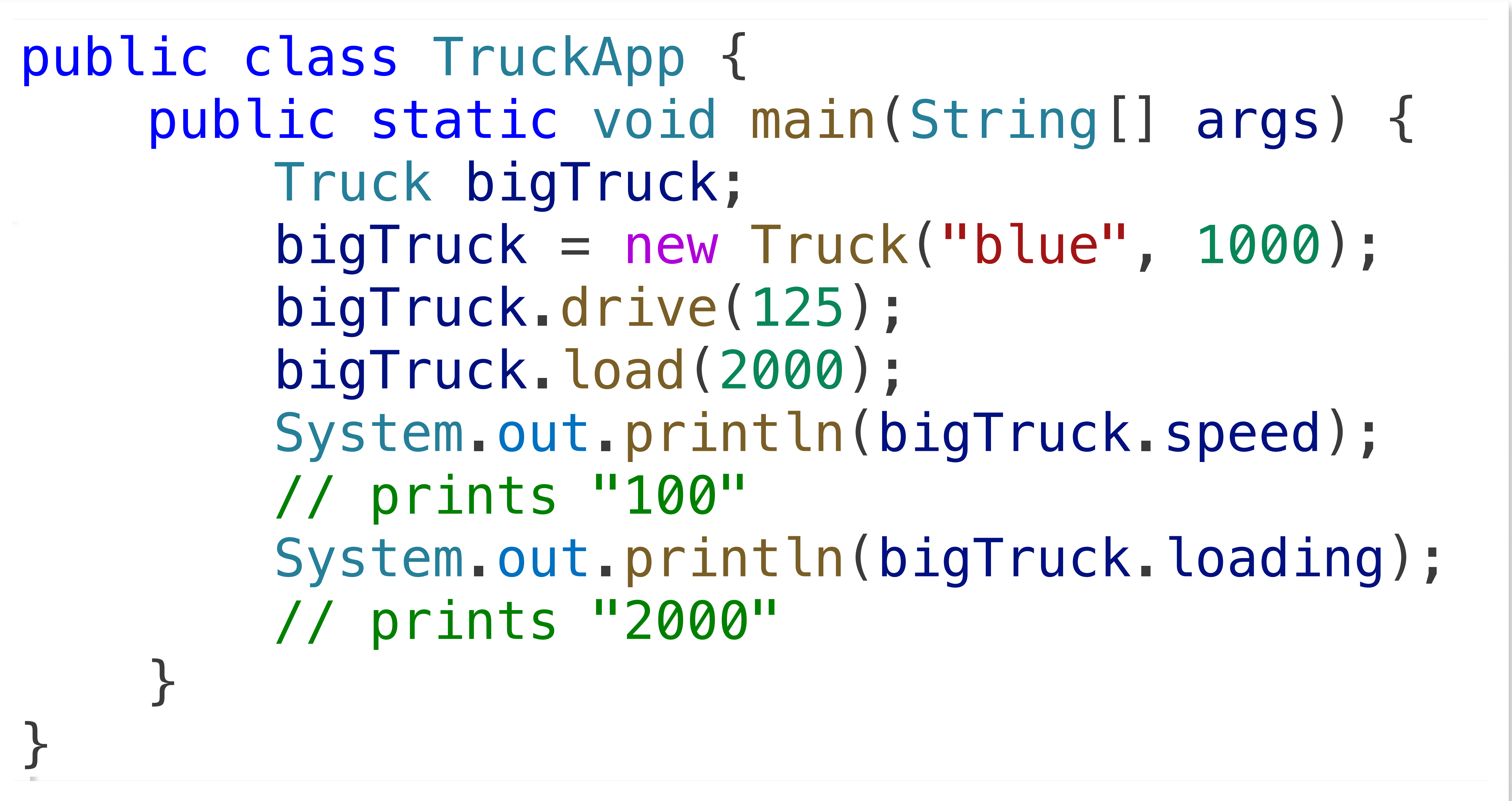

generalize
Car
attributes: speed, direction, color
operations: drive, turn, stop
inherits
attributes: + loading
operations: + load
Truck
attributes: + numOfPassengers
operations:
Passenger

Subclassing

Subclass **extends** class



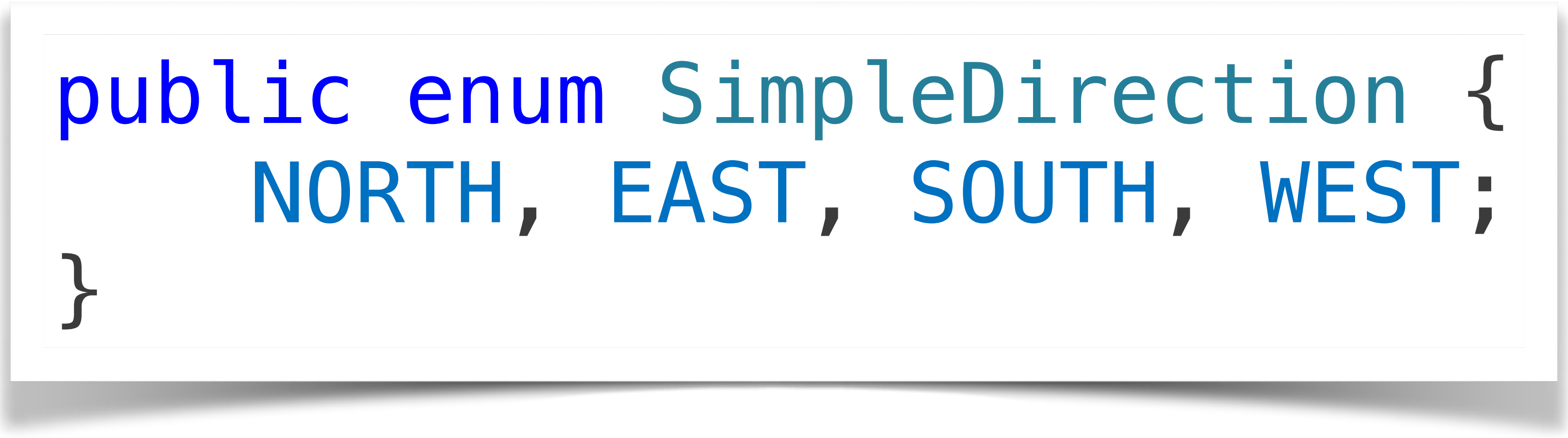
```
public class Truck extends Car {
    // Additional state variable
    int loading;
    // Operations - methods
    public Truck (String color, int loading) {
        super(color);
        this.loading = loading;
    }
    @Override
    public void drive (int newSpeed) {
        if (newSpeed <= 100)
            super.drive(newSpeed);
        else
            super.drive(100);
    }
    public void load(int loading) {
        this.loading = loading;
    }
}
```



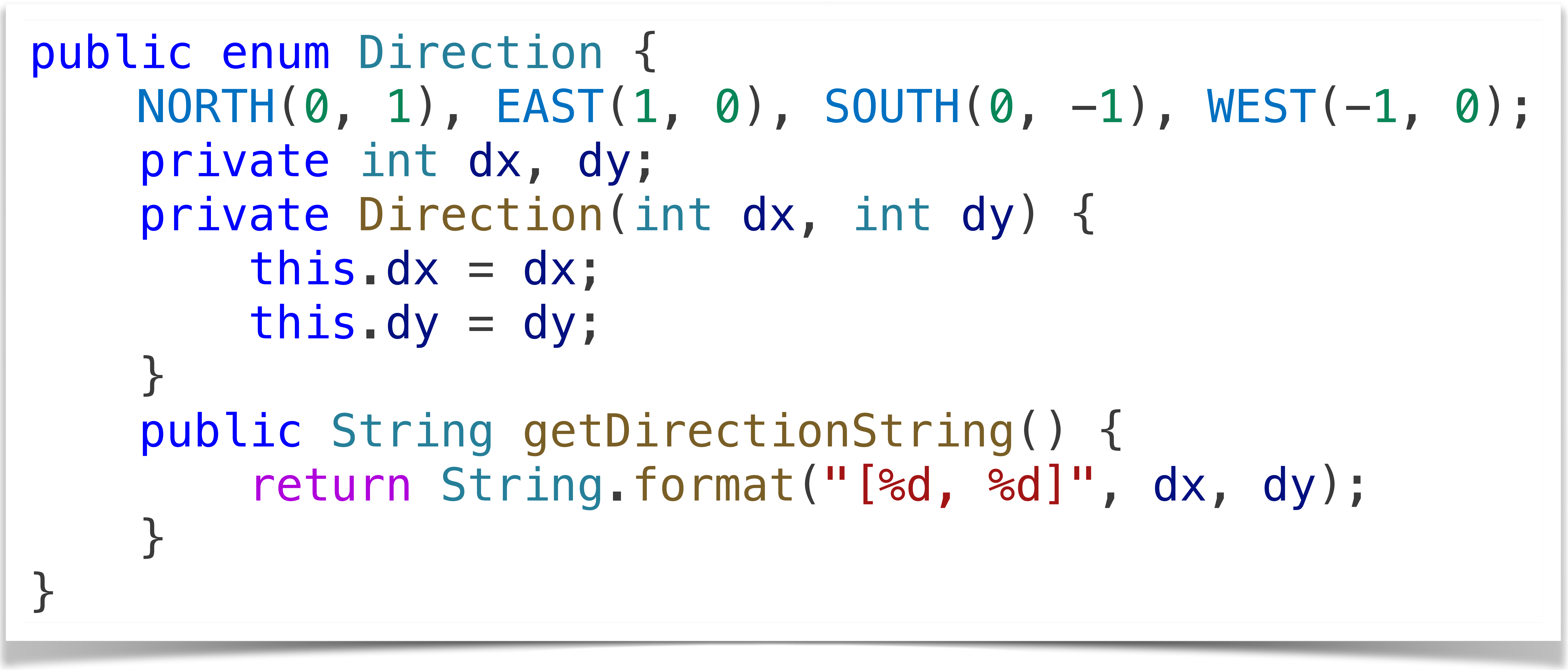
```
public class TruckApp {
    public static void main(String[] args) {
        Truck bigTruck;
        bigTruck = new Truck("blue", 1000);
        bigTruck.drive(125);
        bigTruck.load(2000);
        System.out.println(bigTruck.speed);
        // prints "100"
        System.out.println(bigTruck.loading);
        // prints "2000"
    }
}
```

Enumeration

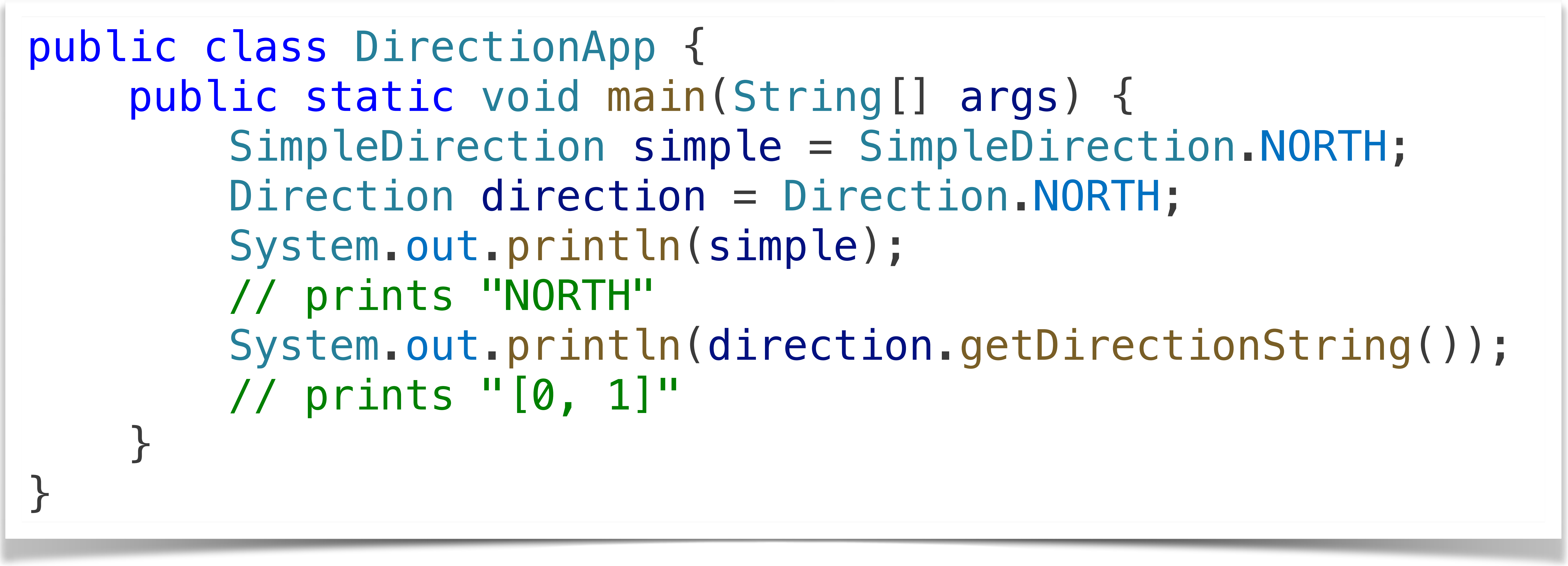
Special object type



```java
public enum SimpleDirection {
    NORTH, EAST, SOUTH, WEST;
}
```



```java
public enum Direction {
    NORTH(0, 1), EAST(1, 0), SOUTH(0, -1), WEST(-1, 0);
    private int dx, dy;
    private Direction(int dx, int dy) {
        this.dx = dx;
        this.dy = dy;
    }
    public String getDirectionString() {
        return String.format("[%d, %d]", dx, dy);
    }
}
```



```java
public class DirectionApp {
    public static void main(String[] args) {
        SimpleDirection simple = SimpleDirection.NORTH;
        Direction direction = Direction.NORTH;
        System.out.println(simple);
        // prints "NORTH"
        System.out.println(direction.getDirectionString());
        // prints "[0, 1]"
    }
}
```

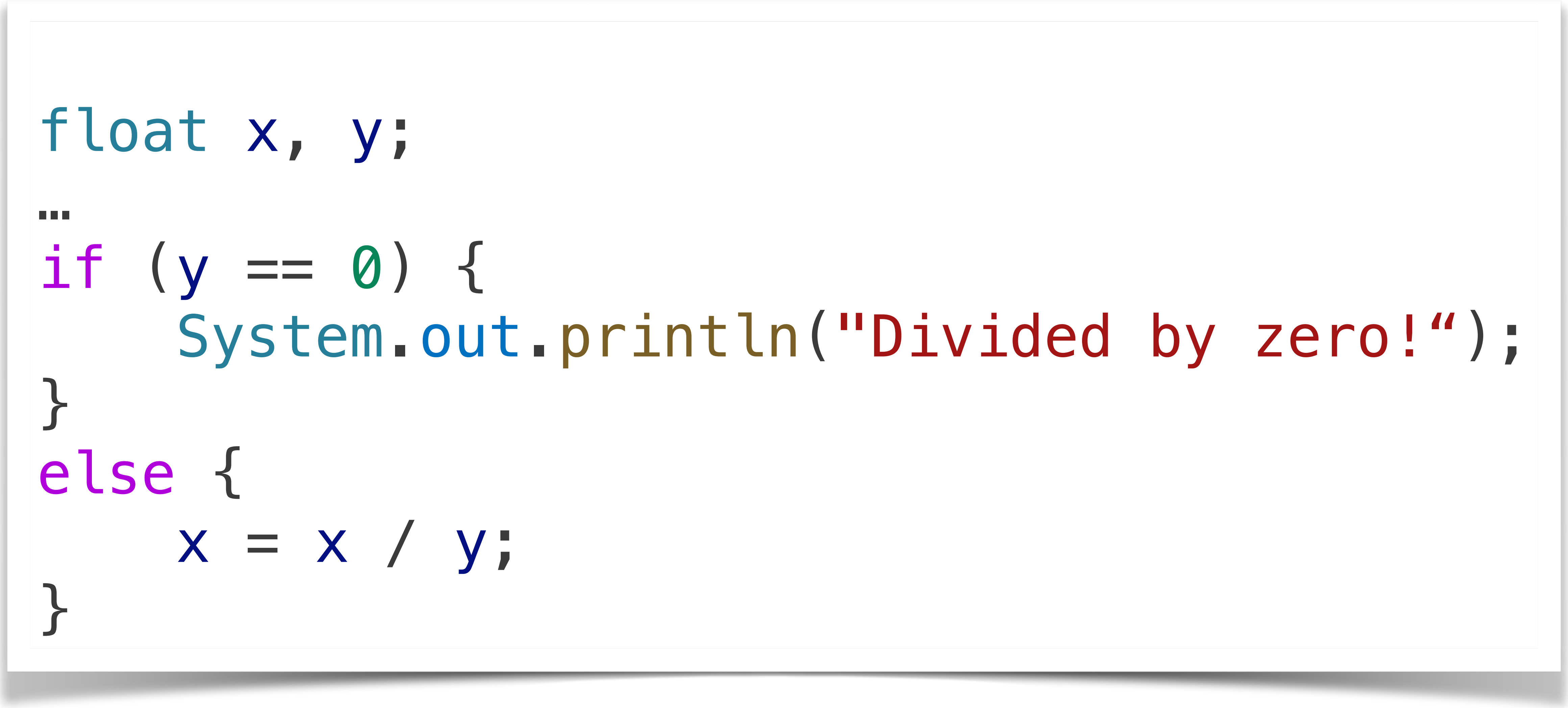
Java Control Structures

Legacy from C/C++

- Very similar
  - block of code; **if**, **if/else**; ternary operator; **switch**, loops (**for**, **while**, **do-while**); **break**, **continue**
- Conditional expression has to be **boolean** type (**implicit conversion from int is not allowed**)
- Types used in switch - primitive: **byte, char, short, int**; object: **String**, enumeration (**enum**)
- **for** exists in a form of a **for-each** construction.

Branching Statement *if-else*

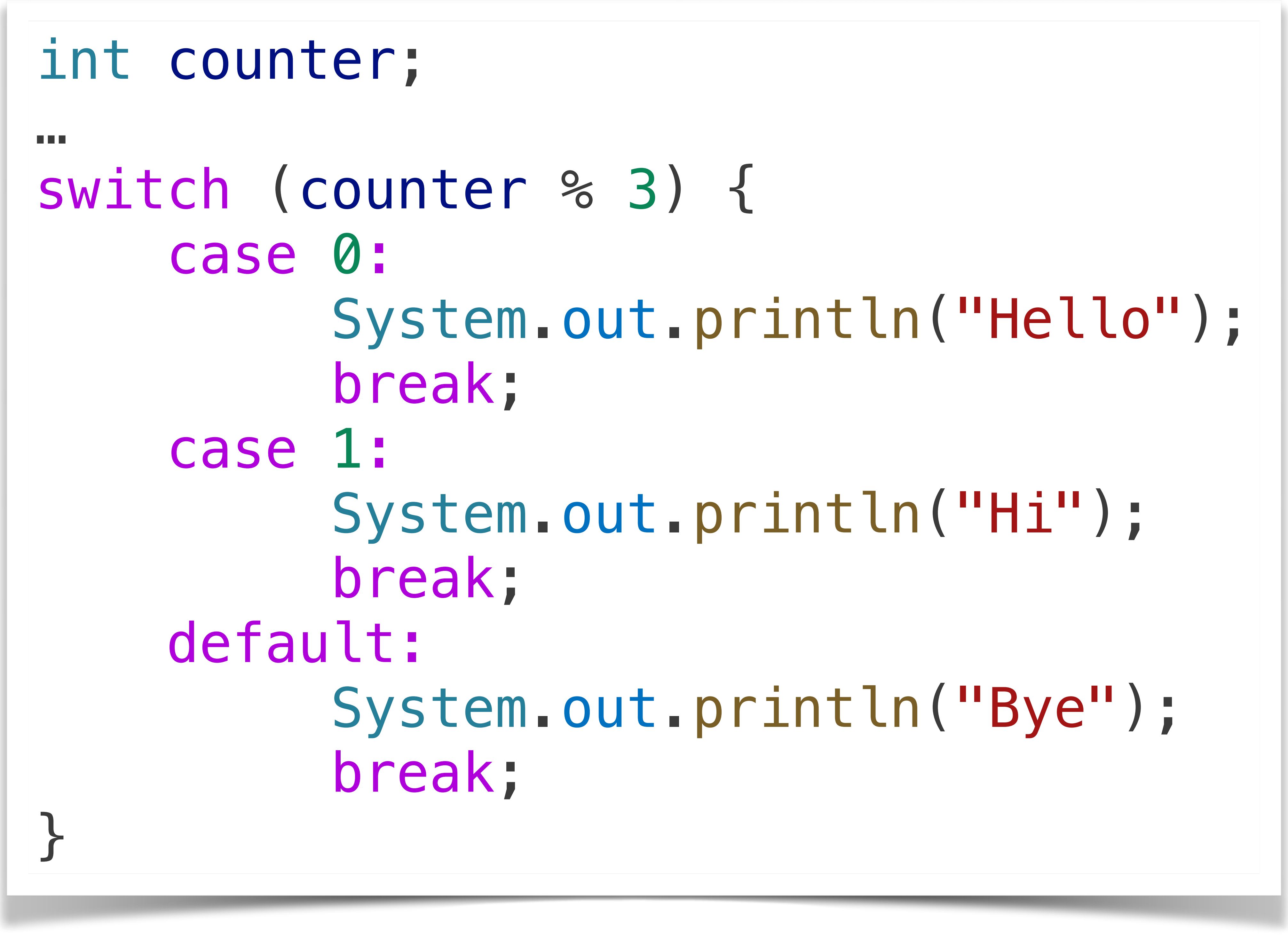
```
if (boolean) {
  statements;
}
else {
  statements;
}
```



```
float x, y;
…
if (y == 0) {
    System.out.println("Divided by zero!“);
}
else {
    x = x / y;
}
```

Branching Statement *switch*

```
switch (expr) {
  case expr1:
        statements;
        break;
  case expr2:
        statements;
        break;
  default:
        statements;
}
```



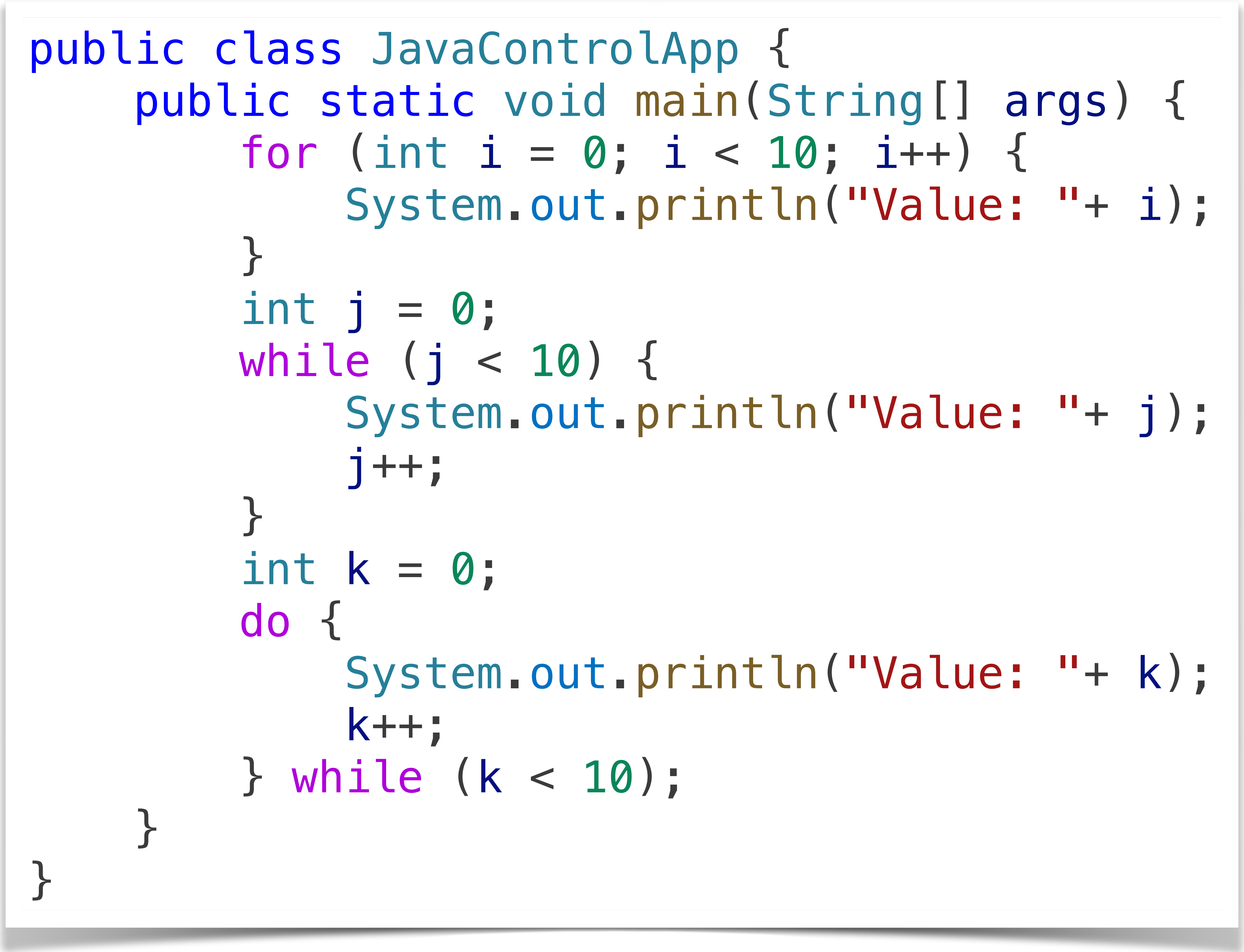
```
int counter;
…
switch (counter % 3) {
    case 0:
        System.out.println("Hello");
        break;
    case 1:
        System.out.println("Hi");
        break;
    default:
        System.out.println("Bye");
        break;
}
```

Loop Statements *for*, *while*, and *do*

for (init_expr; test_expr; increment_expr) {
statements;
}

while (boolean) {
statements;
}

do {
statements;
} **while** (boolean);



```java
public class JavaControlApp {
    public static void main(String[] args) {
        for (int i = 0; i < 10; i++) {
            System.out.println("Value: "+ i);
        }
        int j = 0;
        while (j < 10) {
            System.out.println("Value: "+ j);
            j++;
        }
        int k = 0;
        do {
            System.out.println("Value: "+ k);
            k++;
        } while (k < 10);
    }
}
```

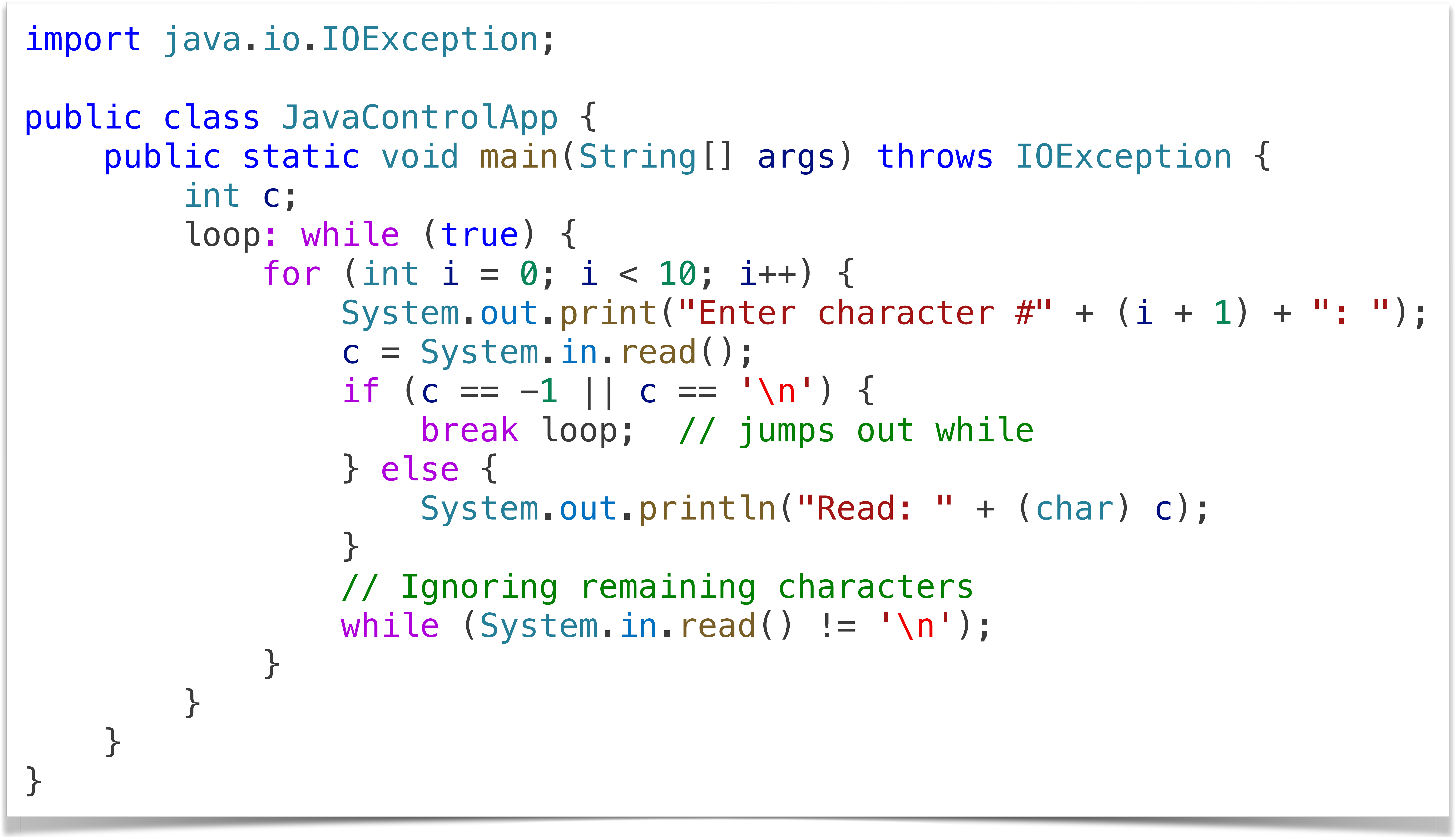
General Flow Control

label: *statement;* // statement must be a loop statement

break *[label]*

continue *[label]*

return *expr;*



```
import java.io.IOException;

public class JavaControlApp {
    public static void main(String[] args) throws IOException {
        int c;
        loop: while (true) {
            for (int i = 0; i < 10; i++) {
                System.out.print("Enter character #" + (i + 1) + ": ");
                c = System.in.read();
                if (c == -1 || c == '\n') {
                    break loop;  // jumps out while
                } else {
                    System.out.println("Read: " + (char) c);
                }
                // Ignoring remaining characters
                while (System.in.read() != '\n');
            }
        }
    }
}
```

Exceptions and Exception Handling

```
try {
    critical_statements;
}
catch (ExceptionType e) {
    // Handle exception object e
}
finally {
    always_statements;
}
```

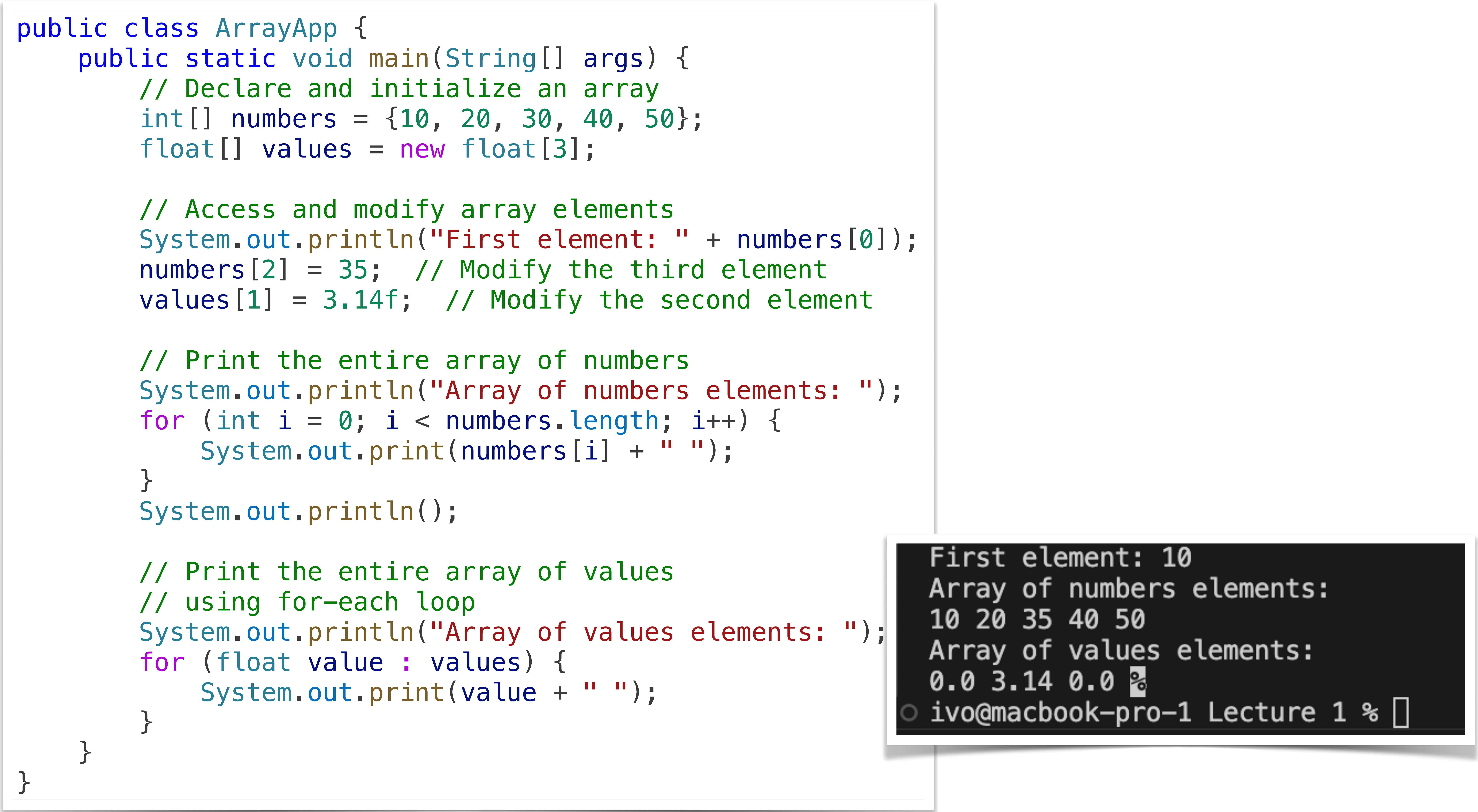
- **Declaring Exceptions:** *void* method(arg...) *throws* ExceptionType {...}

- **Defining and Generating Exceptions:** *throw new* MyException(“text to show”)

Arrays

An array is a data structure that allows you to store multiple values of the same data type in a single variable.

- **Fixed Size:** Once an array is created, its size cannot be changed. The size is defined when the array is instantiated.
- **Zero-Based Indexing:** The elements of an array are accessed using an index that starts from 0 for the first element and goes up to length-1 for the last element.
- **Homogeneous Elements:** All elements in an array must be of the same data type.

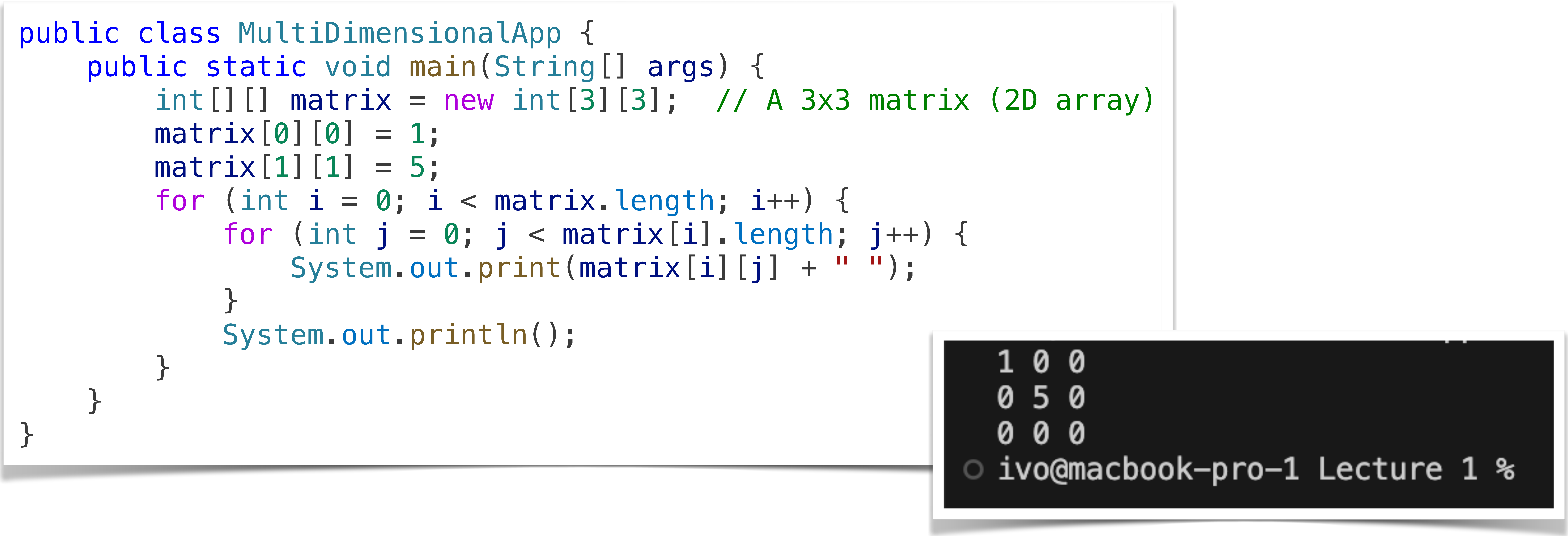


```java
public class ArrayApp {
    public static void main(String[] args) {
        // Declare and initialize an array
        int[] numbers = {10, 20, 30, 40, 50};
        float[] values = new float[3];

        // Access and modify array elements
        System.out.println("First element: " + numbers[0]);
        numbers[2] = 35;  // Modify the third element
        values[1] = 3.14f;  // Modify the second element

        // Print the entire array of numbers
        System.out.println("Array of numbers elements: ");
        for (int i = 0; i < numbers.length; i++) {
            System.out.print(numbers[i] + " ");
        }
        System.out.println();

        // Print the entire array of values
        // using for-each loop
        System.out.println("Array of values elements: ");
        for (float value : values) {
            System.out.print(value + " ");
        }
    }
}
```

```
 First element: 10
 Array of numbers elements:
 10 20 35 40 50
 Array of values elements:
 0.0 3.14 0.0 %
○ ivo@macbook-pro-1 Lecture 1 % ▯
```

Multidimensional Arrays

Java also supports multidimensional arrays, such as 2D arrays (arrays of arrays):



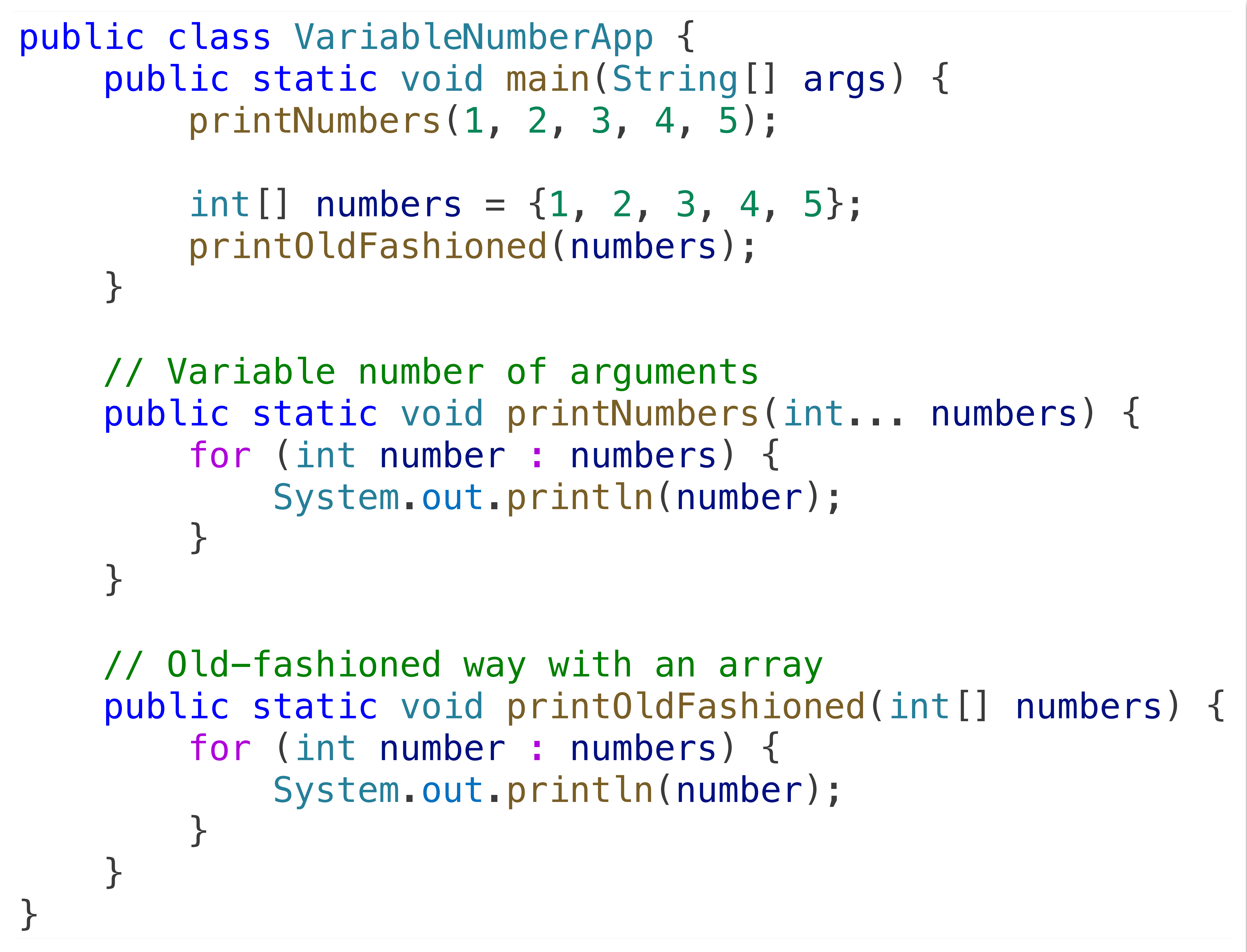
```java
public class MultiDimensionalApp {
    public static void main(String[] args) {
        int[][] matrix = new int[3][3];  // A 3x3 matrix (2D array)
        matrix[0][0] = 1;
        matrix[1][1] = 5;
        for (int i = 0; i < matrix.length; i++) {
            for (int j = 0; j < matrix[i].length; j++) {
                System.out.print(matrix[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```

```
  1 0 0
  0 5 0
  0 0 0
○ ivo@macbook-pro-1 Lecture 1 %
```

Method with a variable number of arguments

An alternative declaration of a method parameter of array type

- Declared with “…”
- Construction of array is not necessary in the case of a method calling
- Method is called with a variation number of parameters separated by “,”
- Parameters are accessed in the method as they were in an ordinary array



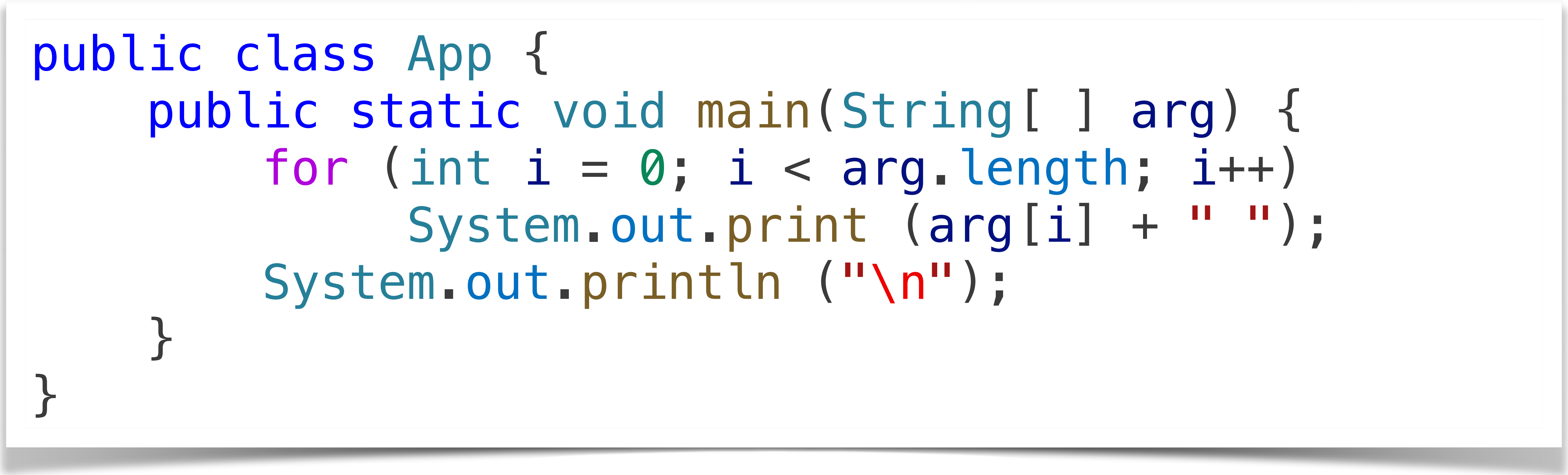
```java
public class VariableNumberApp {
    public static void main(String[] args) {
        printNumbers(1, 2, 3, 4, 5);

        int[] numbers = {1, 2, 3, 4, 5};
        printOldFashioned(numbers);
    }

    // Variable number of arguments
    public static void printNumbers(int... numbers) {
        for (int number : numbers) {
            System.out.println(number);
        }
    }

    // Old-fashioned way with an array
    public static void printOldFashioned(int[] numbers) {
        for (int number : numbers) {
            System.out.println(number);
        }
    }
}
```

2nd Part: More than Basics

- Program structure, packages and classes
- Advanced Object-Oriented Approach
- Classes and Interfaces
- Object construction and destruction

Program Structure

A program in Java consists of one or more class definitions, each of which has been compiled into its own .class file of Java Virtual Machine object code. In case of Java application one of these classes must define a method main().



```
public class App {
    public static void main(String[ ] arg) {
        for (int i = 0; i < arg.length; i++)
             System.out.print (arg[i] + " ");
        System.out.println ("\n");
    }
}
```

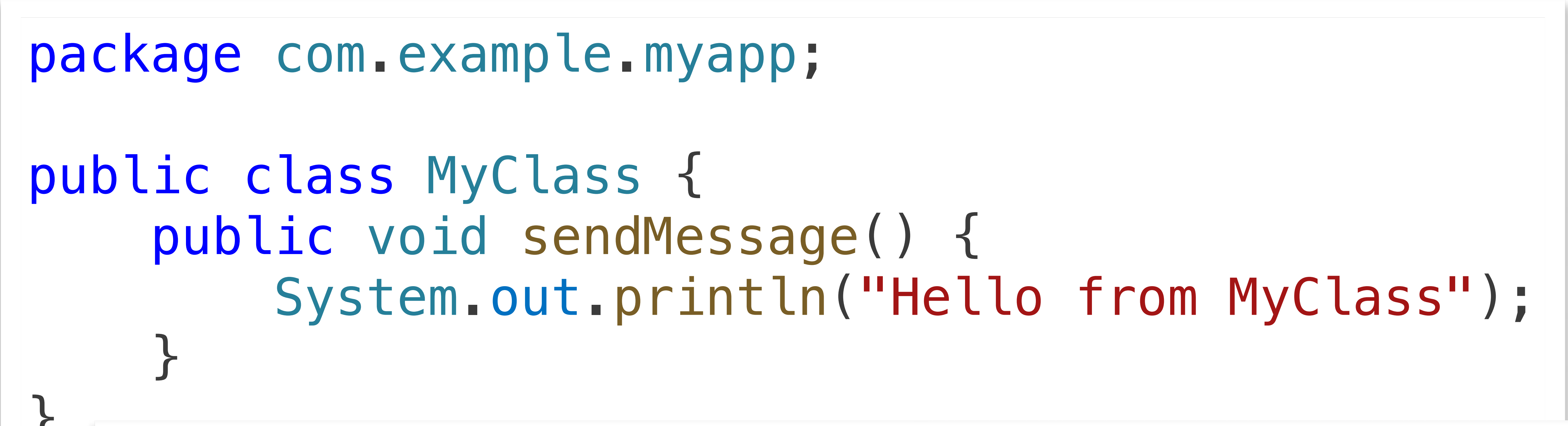
Packages and Classes

A package is a namespace that organizes a set of related classes

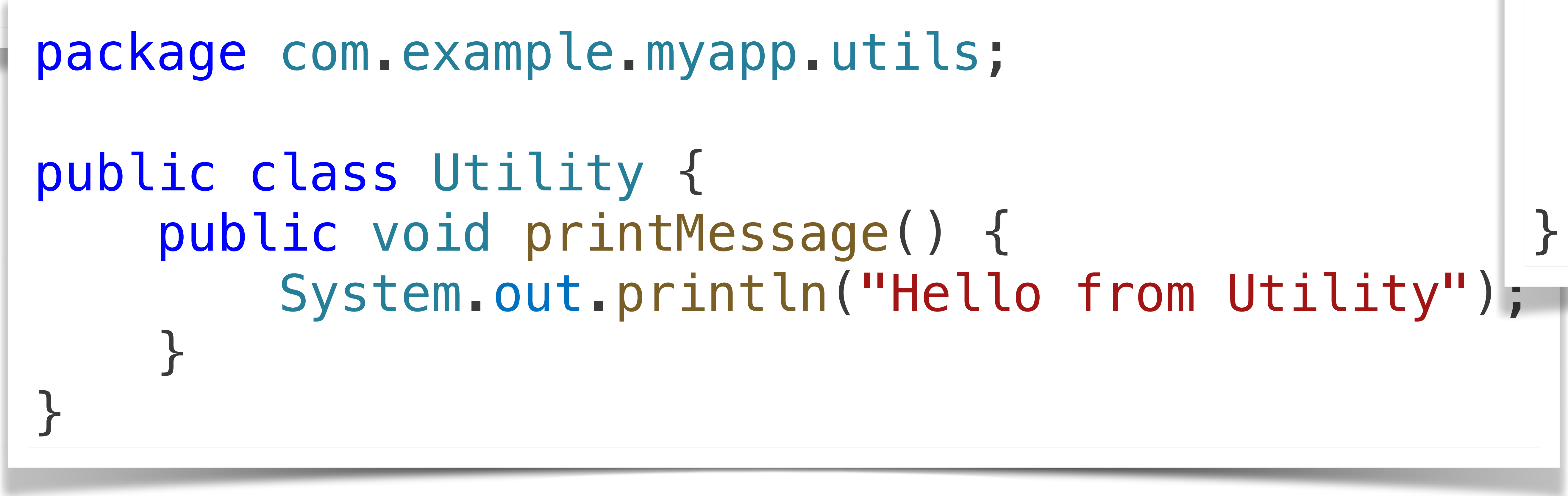
- Every compiled class is stored in a separate file (.class). This class must be stored in a directory that has the same components as the package name =>
    - com.example.myapp.MyClass and com\example\myapp\MyClass.class
    - com.example.myapp.utils.Utility and com\example\myapp\utils\Utility.class
- Source code file (.java) consists of one or more class definitions. Only one class may be declared public and the source file must have the same name

Defining and Importing Packages

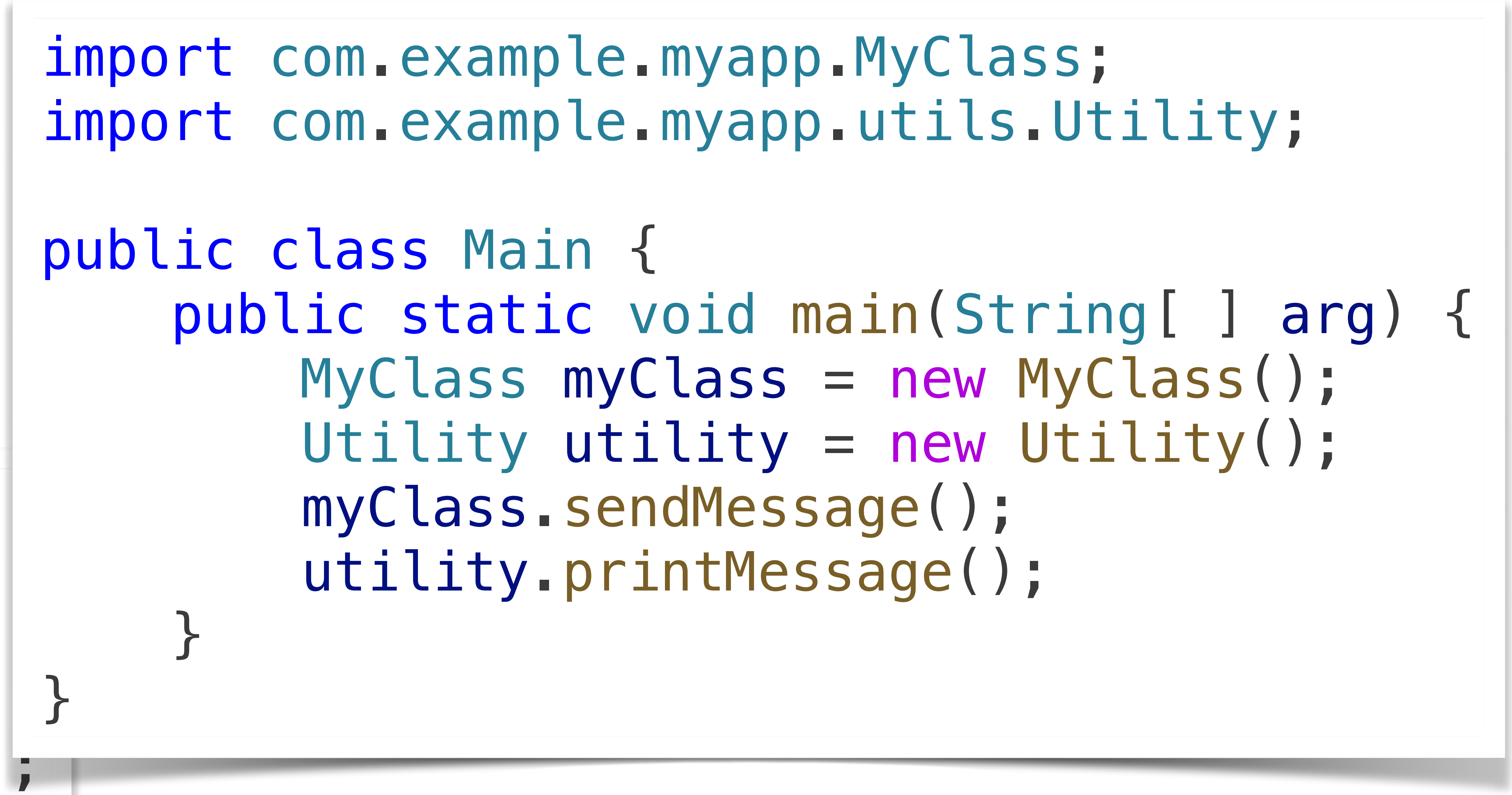
To declare a package, you use the package keyword at the beginning of your Java source file, followed by the package name. To use a class from another package, you need to import it using the import statement. This allows you to refer to classes by their short names rather than their fully qualified names.



```java
package com.example.myapp;

public class MyClass {
    public void sendMessage() {
        System.out.println("Hello from MyClass");
    }
}
```



```java
package com.example.myapp.utils;

public class Utility {
    public void printMessage() {
        System.out.println("Hello from Utility");
    }
}
```



```java
import com.example.myapp.MyClass;
import com.example.myapp.utils.Utility;

public class Main {
    public static void main(String[ ] arg) {
        MyClass myClass = new MyClass();
        Utility utility = new Utility();
        myClass.sendMessage();
        utility.printMessage();
    }
}
```

You can also import all classes from a package using a wildcard (*)

Default Package and Access Modifiers

If you don't specify a package at the beginning of your Java file, the class is placed in the "default" package. The default package has no explicit name, and classes in the default package cannot be imported by classes in other packages.

- **public:** The class or member is accessible from any other class.
- **protected:** The member is accessible within its own package and by subclasses.
- **default (package-private):** The class or member is accessible only within its own package (no modifier is specified).
- **private:** The member is accessible only within its own class.

Packages in Java are a powerful way to organize your code into a structured hierarchy, manage naming conflicts, and control access. They are essential for building modular and maintainable applications.

Packages in the Java Class Library

The classes of the Java class library are organized into packages

- **java.lang** provides classes that are fundamental to the design of the Java language. It is automatically imported into all Java programs.
- **java.awt** (Abstract Window Toolkit) provides classes to build GUI components
- **java.net** provides for networking applications
- **java.time** provides classes for dates, time, instants, and durations
- Any many many others …

Libraries and JARs

Compiled classes could be packed into one jar archive (zip format) and reused

- JVM looks up classes relative to the directories specifies by the CLASSPATH environment variable or by parameter -classpath (-cp) passed as the argument for java statement.

- CLASSPATH= .;c:\java;c:\projects\mylib\classes.jar
  java Main

  or

  java –cp .;c:\java;c:\projects\mylib\classes.jar Main

Object-Oriented Approach

Java's object-oriented nature makes it a powerful and flexible language for developing complex software systems. It encourages the use of objects to model real-world entities, promotes code reuse through inheritance, and enhances maintainability and scalability through encapsulation, polymorphism, and abstraction.

- Object, Type, and Class
- Subtypes and Subclasses
- Creating and Destroying Objects
- Class Variables and Methods
- Data Hiding and Encapsulation
- Abstract Classes

Object, Type, and Class

Interfaces and classes

- **Object** is an distinguishable entity that has:
  Identity: an uniqueness which distinguishes it from all other objects; Behavior: services it provides to another objects; State: value of attributes held by an object

- **Type: visible interface and behavior**

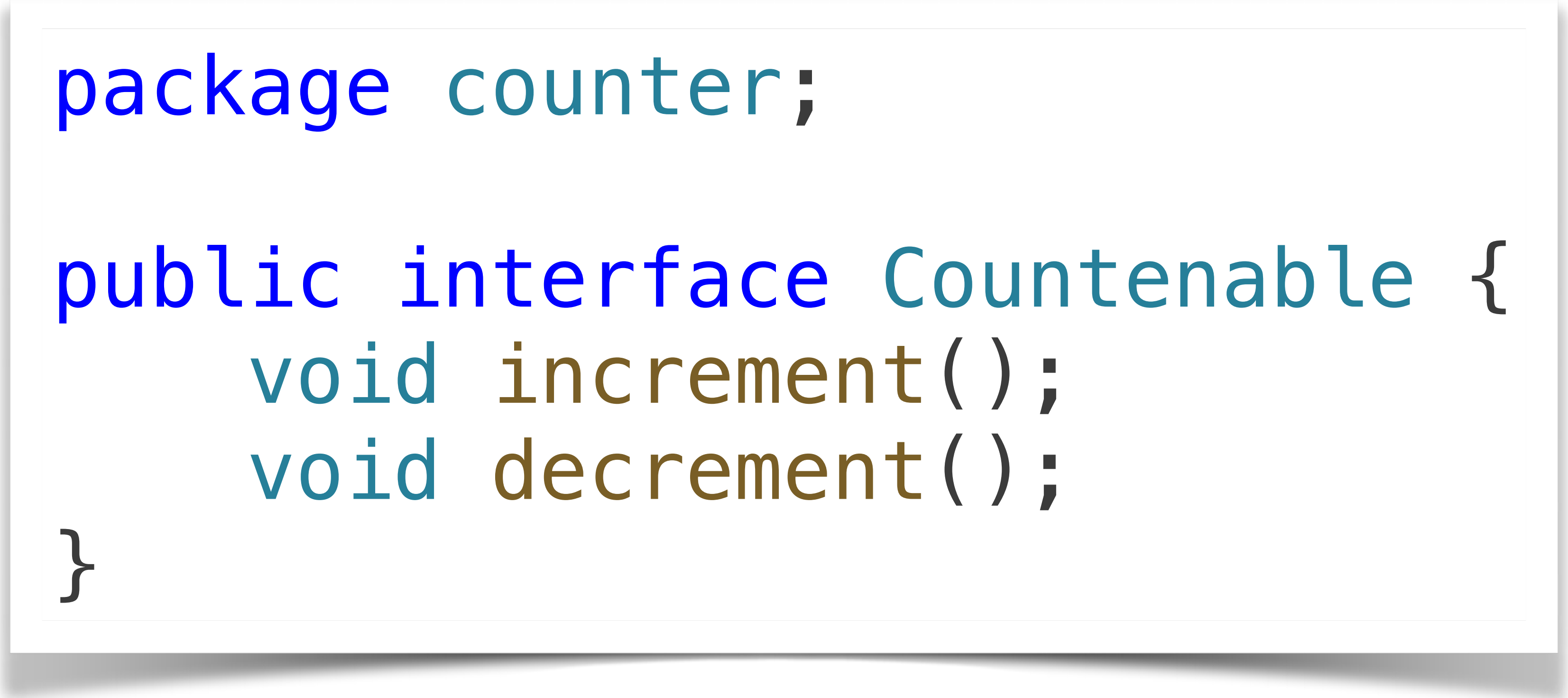
  - **Usually the object is a member of multiple types**

  - **Two objects with different implementation may be the same type**

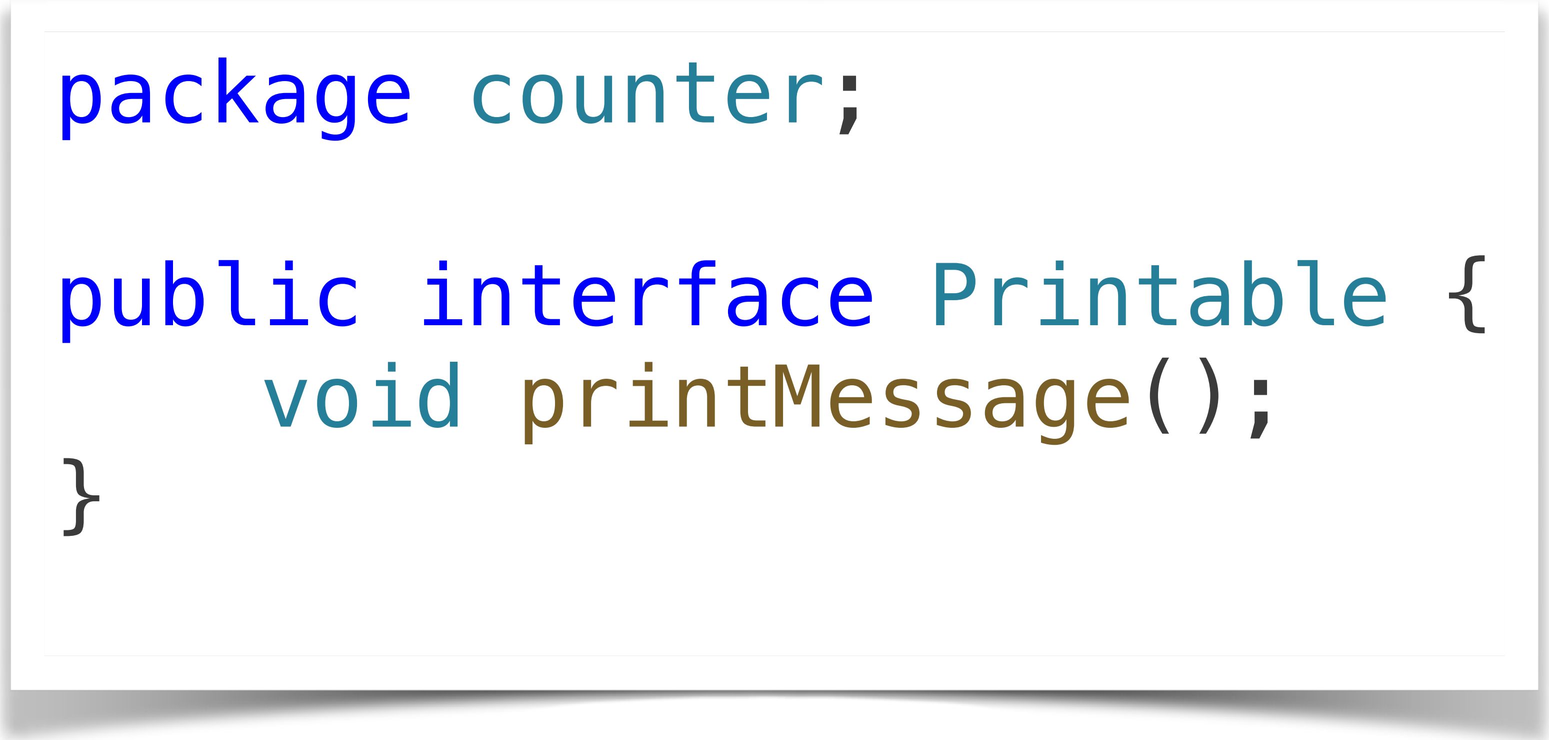
- **Class** is an abstraction of objects with similar implementation:
  Class is definition of set of similar objects; Every object is an instance of the one class

Interface and Class Declaration

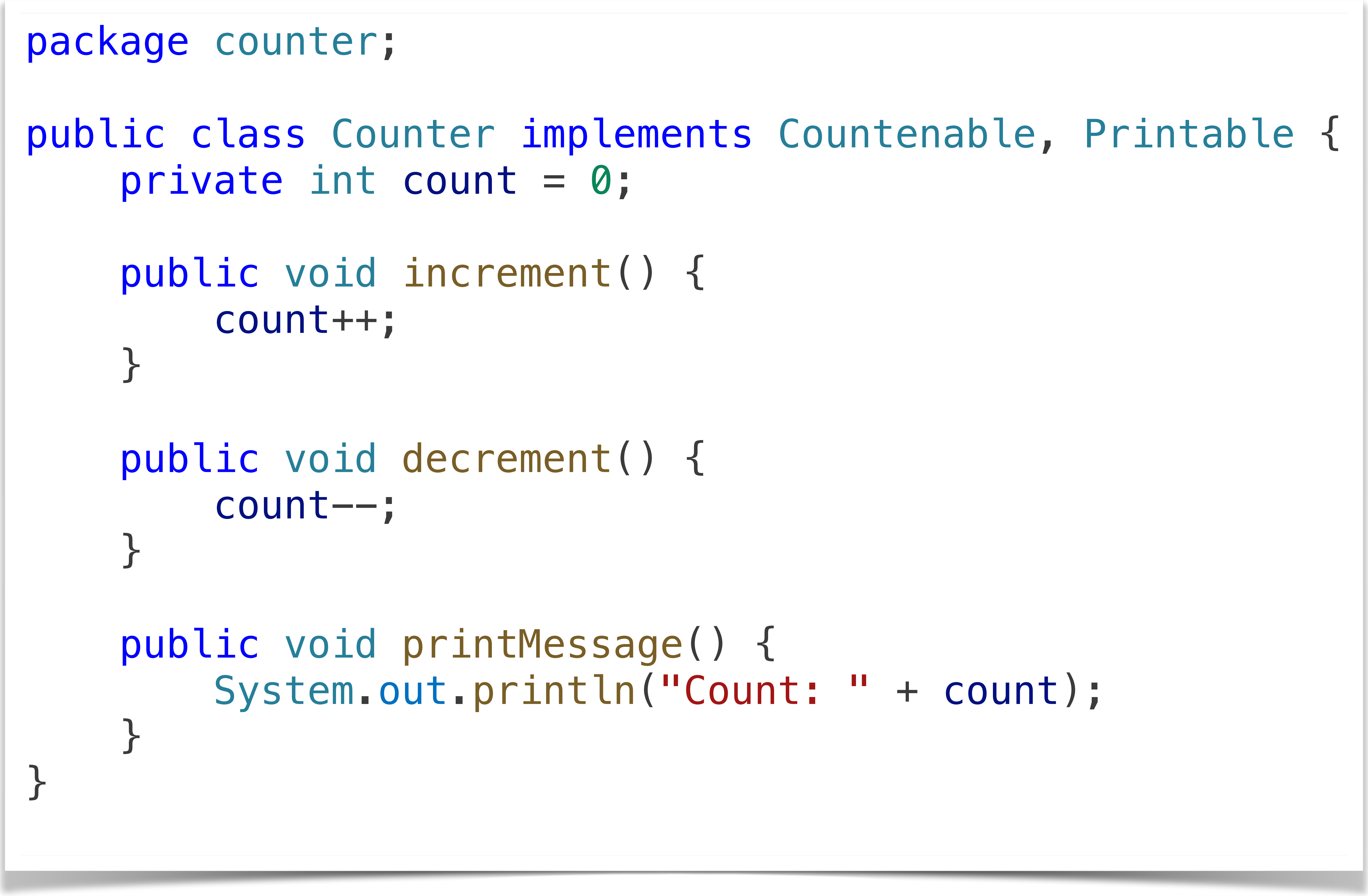
Countenable and Printable Types



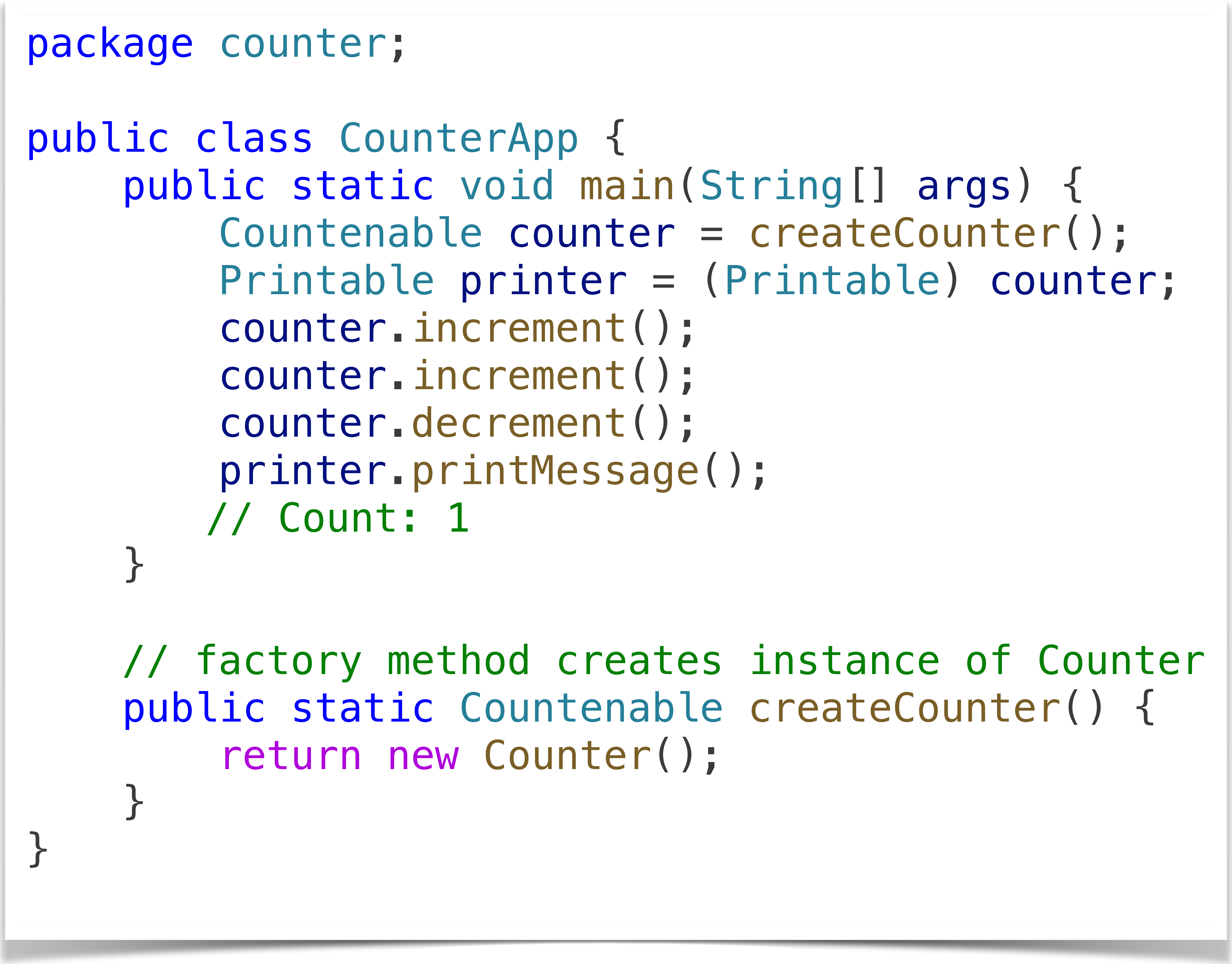
```java
package counter;

public interface Countenable {
    void increment();
    void decrement();
}
```



```java
package counter;

public interface Printable {
    void printMessage();
}
```



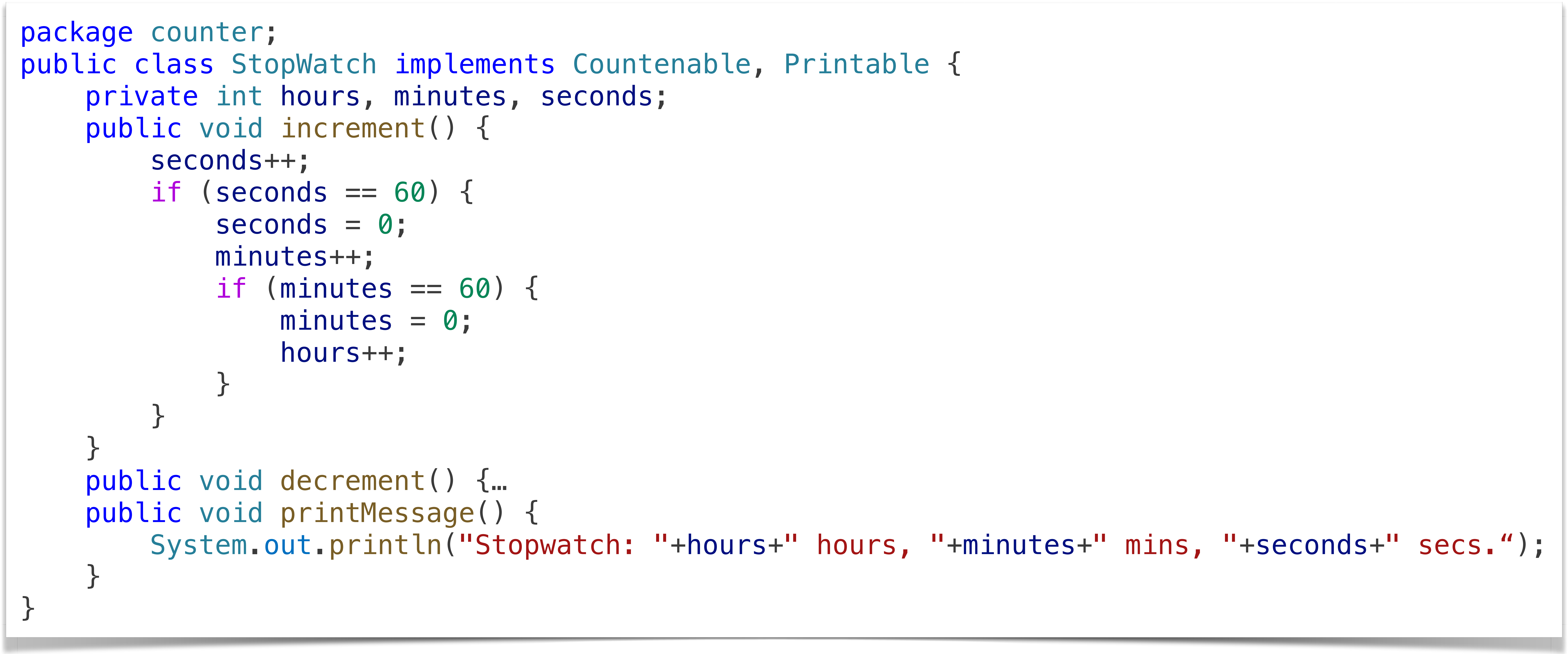
```java
package counter;

public class Counter implements Countenable, Printable {
    private int count = 0;

    public void increment() {
        count++;
    }

    public void decrement() {
        count--;
    }

    public void printMessage() {
        System.out.println("Count: " + count);
    }
}
```



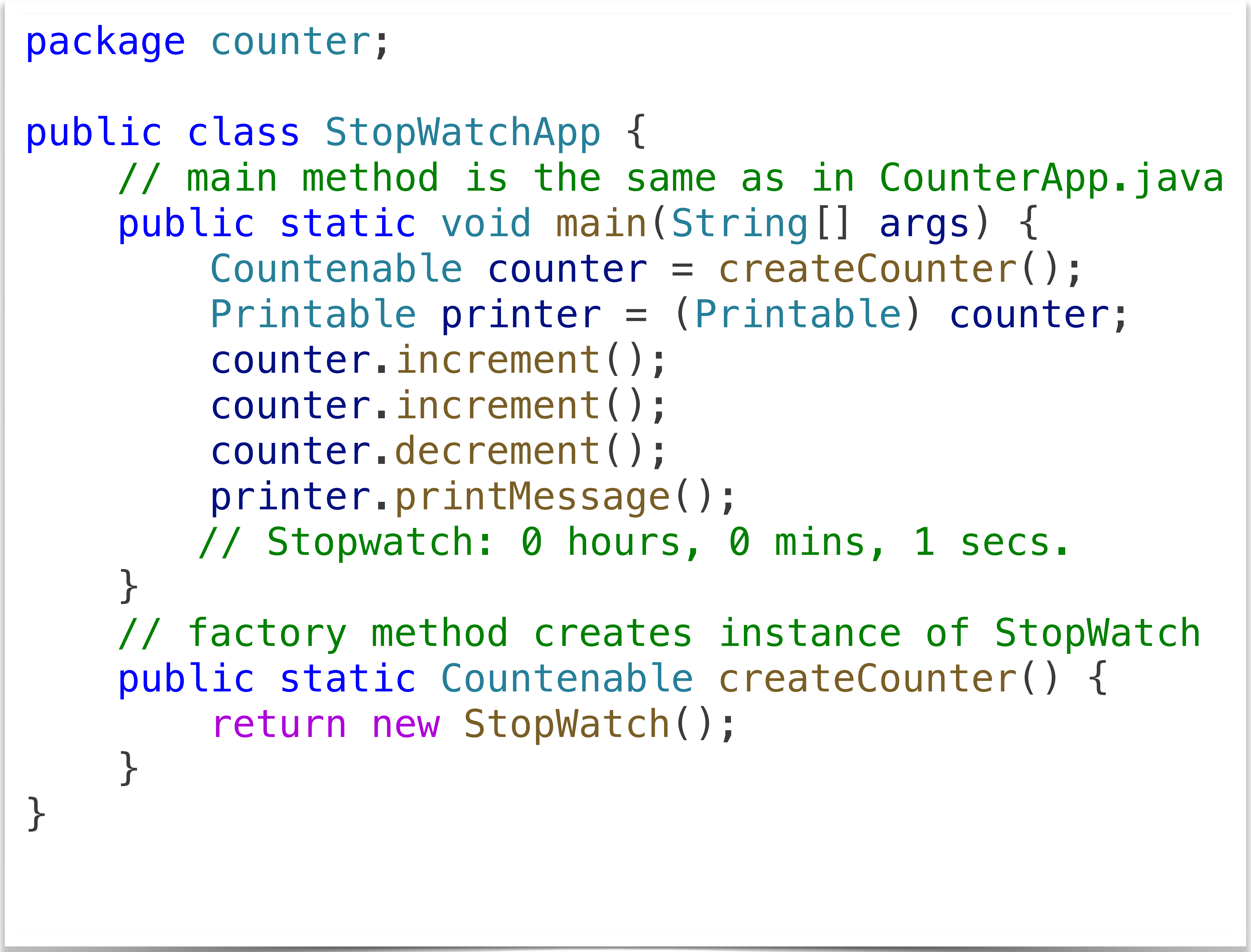
```
package counter;

public class CounterApp {
    public static void main(String[] args) {
        Countenable counter = createCounter();
        Printable printer = (Printable) counter;
        counter.increment();
        counter.increment();
        counter.decrement();
        printer.printMessage();
        // Count: 1
    }

    // factory method creates instance of Counter
    public static Countenable createCounter() {
        return new Counter();
    }
}
```

What is the Benefit?

Re-use of the code for completely different implementations of Counter!



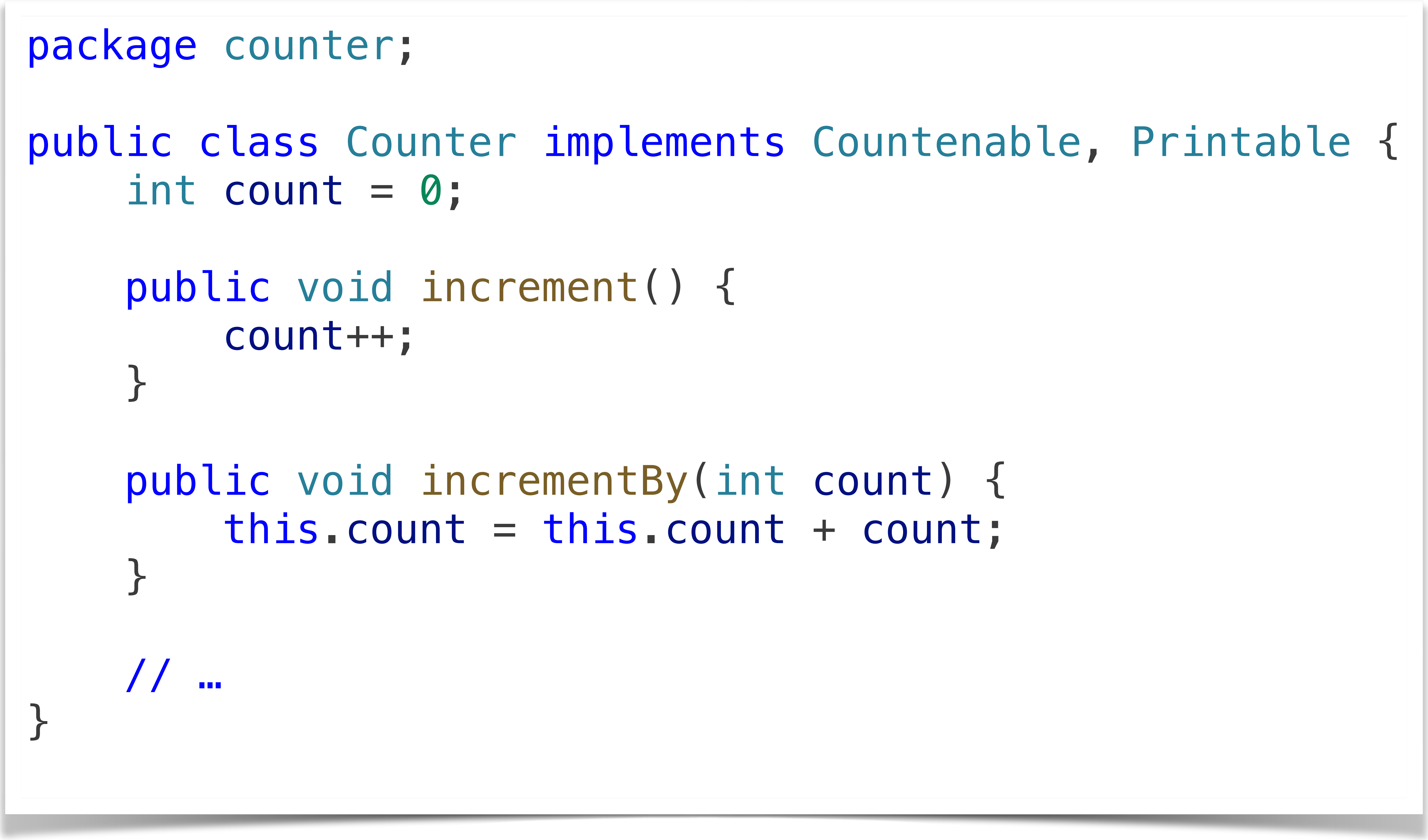
```
package counter;
public class StopWatch implements Countenable, Printable {
    private int hours, minutes, seconds;
    public void increment() {
        seconds++;
        if (seconds == 60) {
            seconds = 0;
            minutes++;
            if (minutes == 60) {
                minutes = 0;
                hours++;
            }
        }
    }
    public void decrement() {…
    public void printMessage() {
        System.out.println("Stopwatch: "+hours+" hours, "+minutes+" mins, "+seconds+" secs.“);
    }
}
```



```
package counter;

public class StopWatchApp {
    // main method is the same as in CounterApp.java
    public static void main(String[] args) {
        Countenable counter = createCounter();
        Printable printer = (Printable) counter;
        counter.increment();
        counter.increment();
        counter.decrement();
        printer.printMessage();
        // Stopwatch: 0 hours, 0 mins, 1 secs.
    }
    // factory method creates instance of StopWatch
    public static Countenable createCounter() {
        return new StopWatch();
    }
}
```

Referring to Object Itself

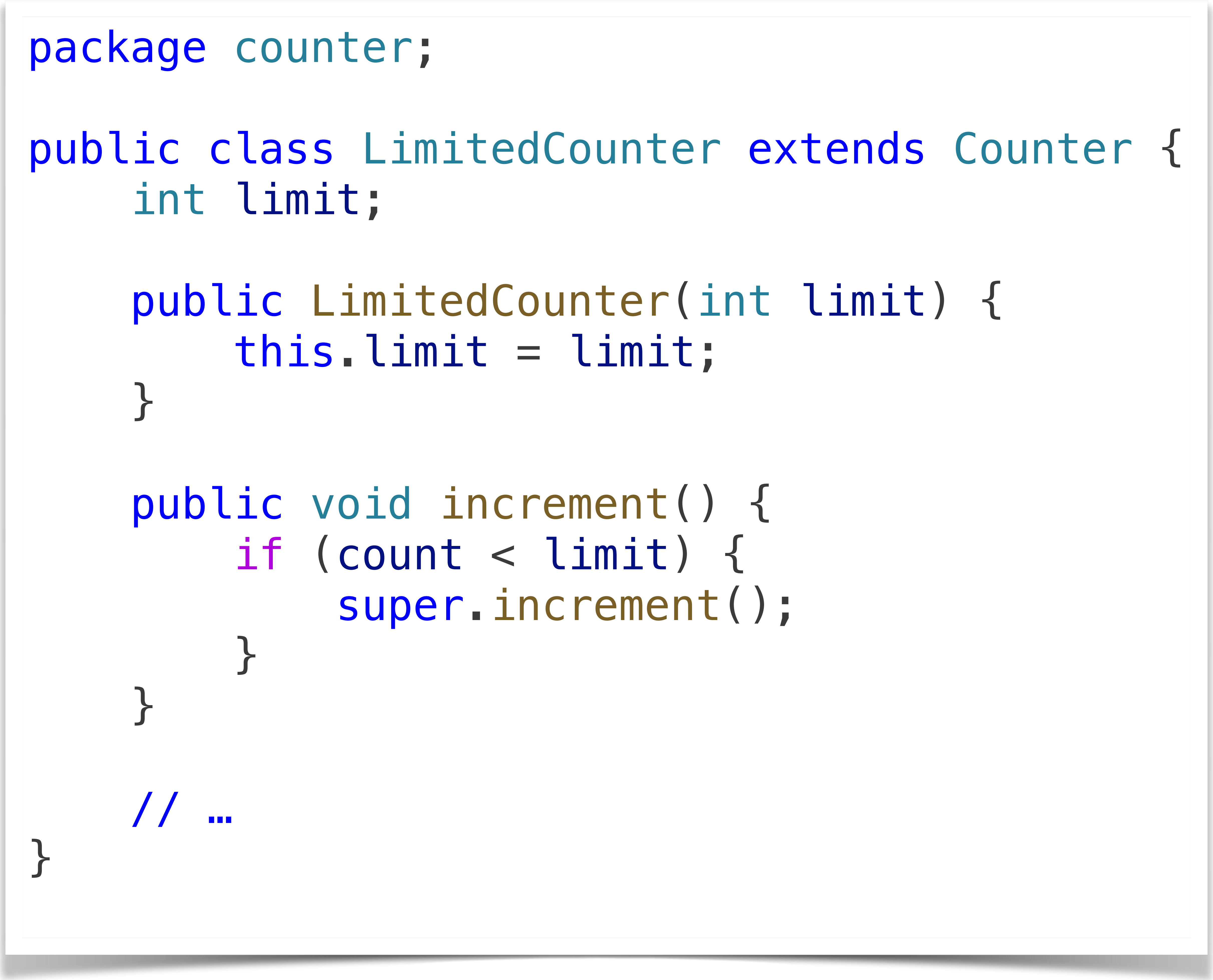
The keyword this can be used to refer to an object itself. If no object reference is specified implicitly this is used.



```
package counter;

public class Counter implements Countenable, Printable {
    int count = 0;

    public void increment() {
        count++;
    }

    public void incrementBy(int count) {
        this.count = this.count + count;
    }

    // …
}
```

Referring to the Parent Class

The keyword super allows to reference methods that were overriden.



```java
package counter;

public class LimitedCounter extends Counter {
    int limit;

    public LimitedCounter(int limit) {
        this.limit = limit;
    }

    public void increment() {
        if (count < limit) {
            super.increment();
        }
    }

    // …
}
```

Constructors

Initialization of the new object

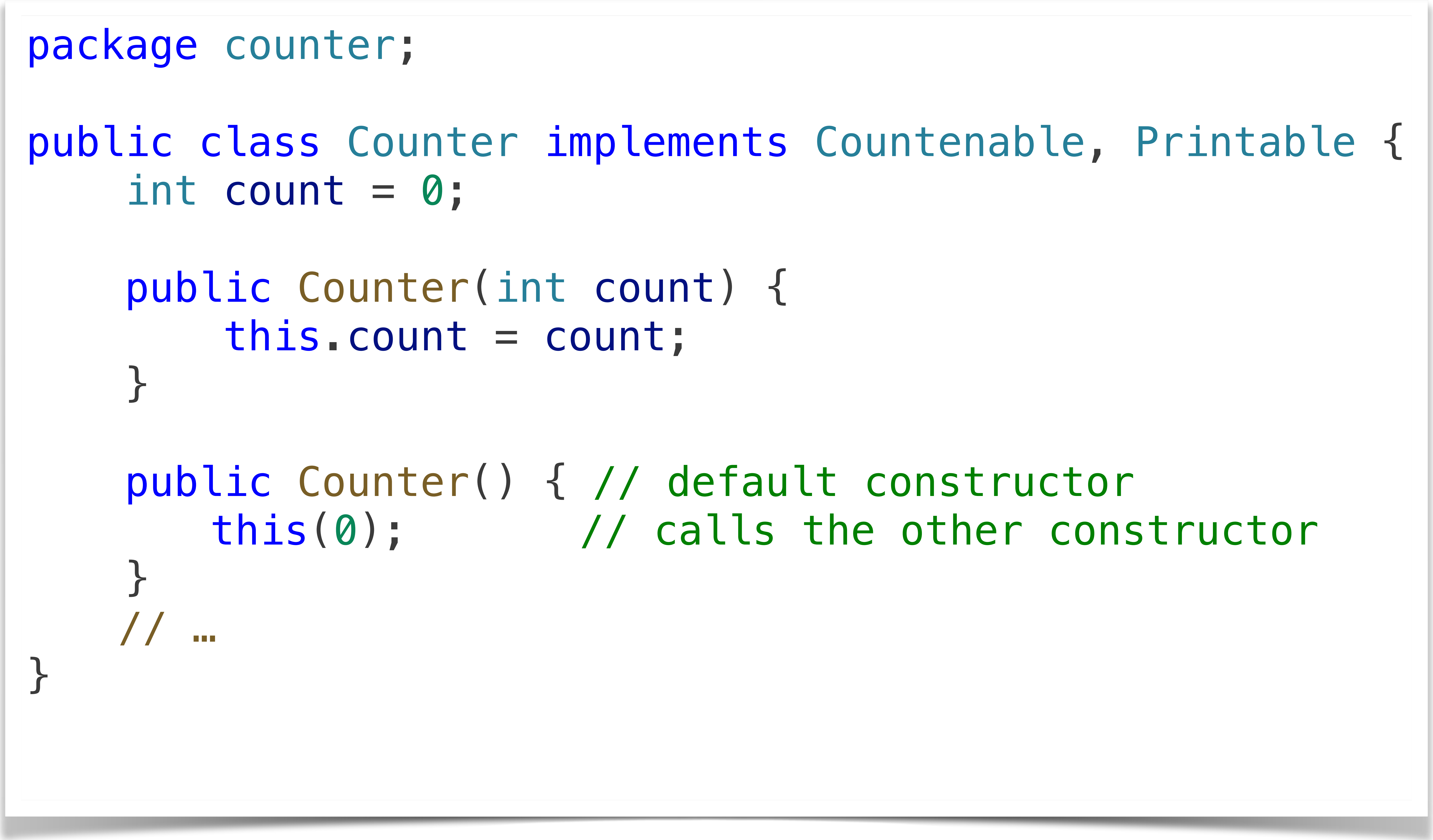
- Every class has at least one constructor method responsible for initialization of the new object. If no constructor is defined Java creates default one with no arguments.

- The constructor name is always the same as the class name.

- The return object is implicitly an instance of the class. No return type is specified, nor is the void keyword used.

Multiple Constructors

Many ways how the new object is initialized



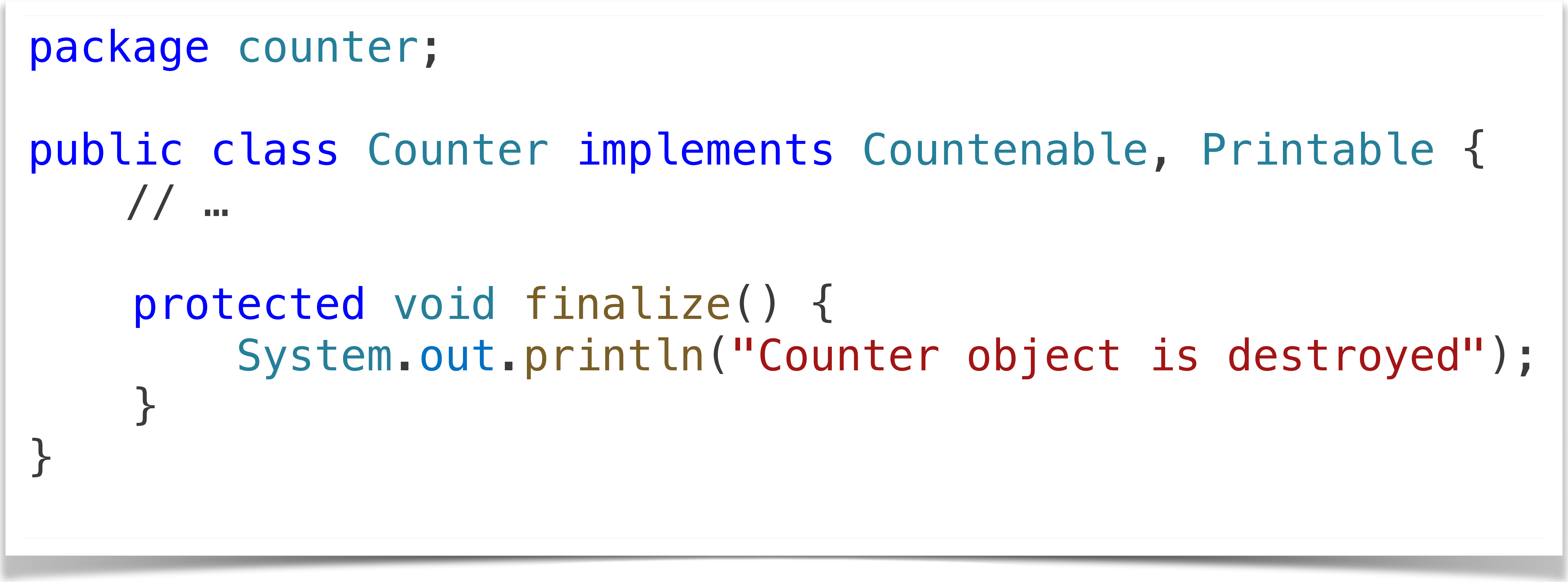
```java
package counter;

public class Counter implements Countenable, Printable {
    int count = 0;

    public Counter(int count) {
        this.count = count;
    }

    public Counter() { // default constructor
        this(0);       // calls the other constructor
    }
    // …
}
```

Object Destruction

Garbage Collection destroys objects that are no longer needed.

- Garbage Collection runs as low priority thread when nothing else is going on or when the interpreter has run out of memory.

- Java finalize method performs finalization for an object.

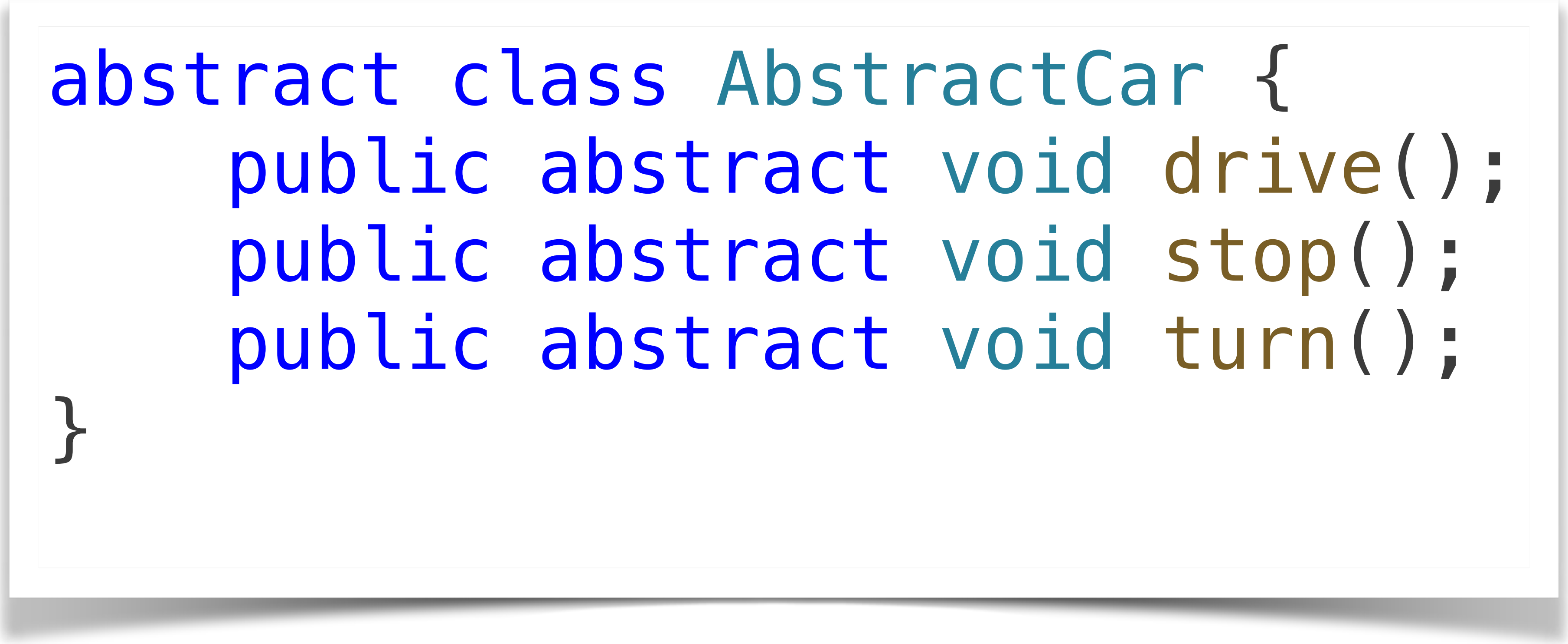


```
package counter;

public class Counter implements Countenable, Printable {
    // …

    protected void finalize() {
        System.out.println("Counter object is destroyed");
    }
}
```

Abstract Class

An abstract class in Java is a class that cannot be instantiated on its own and is meant to be subclassed.

- An abstract method has no body; it has a signature definition followed by a semicolon. Any class with an abstract method is automatically abstract.
- An abstract class cannot be instantiated.
- A subclass of an abstract class can be instantiated if it overrides each of the abstract methods and provides an implementation.



```java
abstract class AbstractCar {
    public abstract void drive();
    public abstract void stop();
    public abstract void turn();
}
```

3rd Part: Advanced

- Nested classes
- Lambda expression
- Generics
- Wrapper classes

Nested Classes

A nested class is a class defined within another class. Nested classes can be used for various purposes, and they offer several advantages, such as encapsulation, organization, and improved code readability.

- **Static Nested Class:** This is essentially a static class that is defined within another class.

- **Inner Class (Non-static Nested Class)**: An inner class is a non-static nested class, and it can access the instance variables and methods of the outer class.

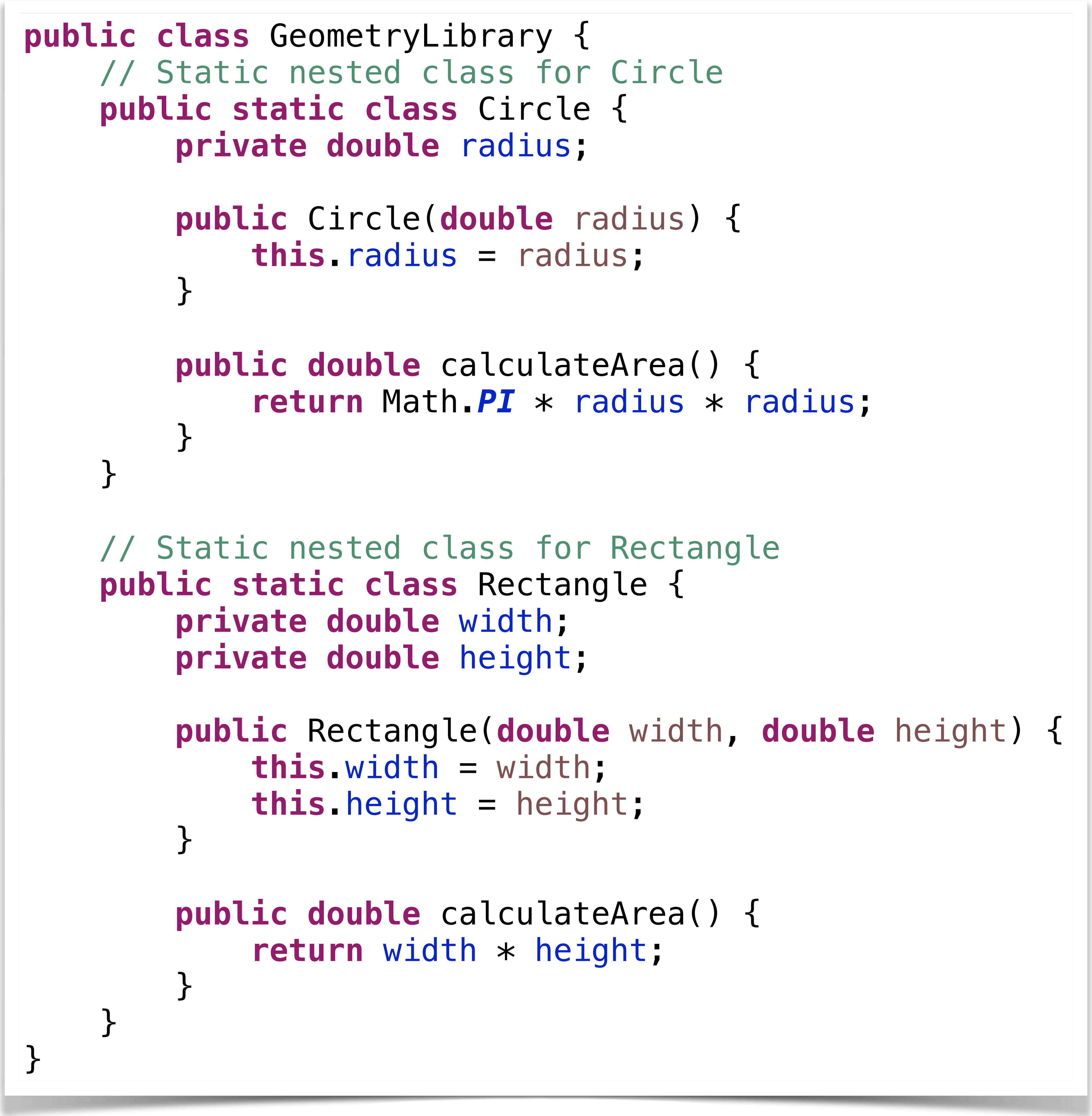
- **Local Class:** Local classes are defined within methods, constructors, or blocks.

- **Anonymous Inner Class:** Anonymous inner classes are a special type of inner class that don't have a name.

Static Nested Class

This is essentially a static class that is defined within another class. It is associated with the outer class but does not have access to the instance variables of the outer class. You can create an instance of a static nested class without creating an instance of the outer class.

```java
public class OuterClass {
  static class StaticNestedClass {
    // ...
  }
}

// Creating an instance of the static nested class
OuterClass.StaticNestedClass nestedObj = new OuterClass.StaticNestedClass();
```



```java
public class GeometryLibrary {
    // Static nested class for Circle
    public static class Circle {
        private double radius;

        public Circle(double radius) {
            this.radius = radius;
        }

        public double calculateArea() {
            return Math.PI * radius * radius;
        }
    }

    // Static nested class for Rectangle
    public static class Rectangle {
        private double width;
        private double height;

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

        public double calculateArea() {
            return width * height;
        }
    }
}
```

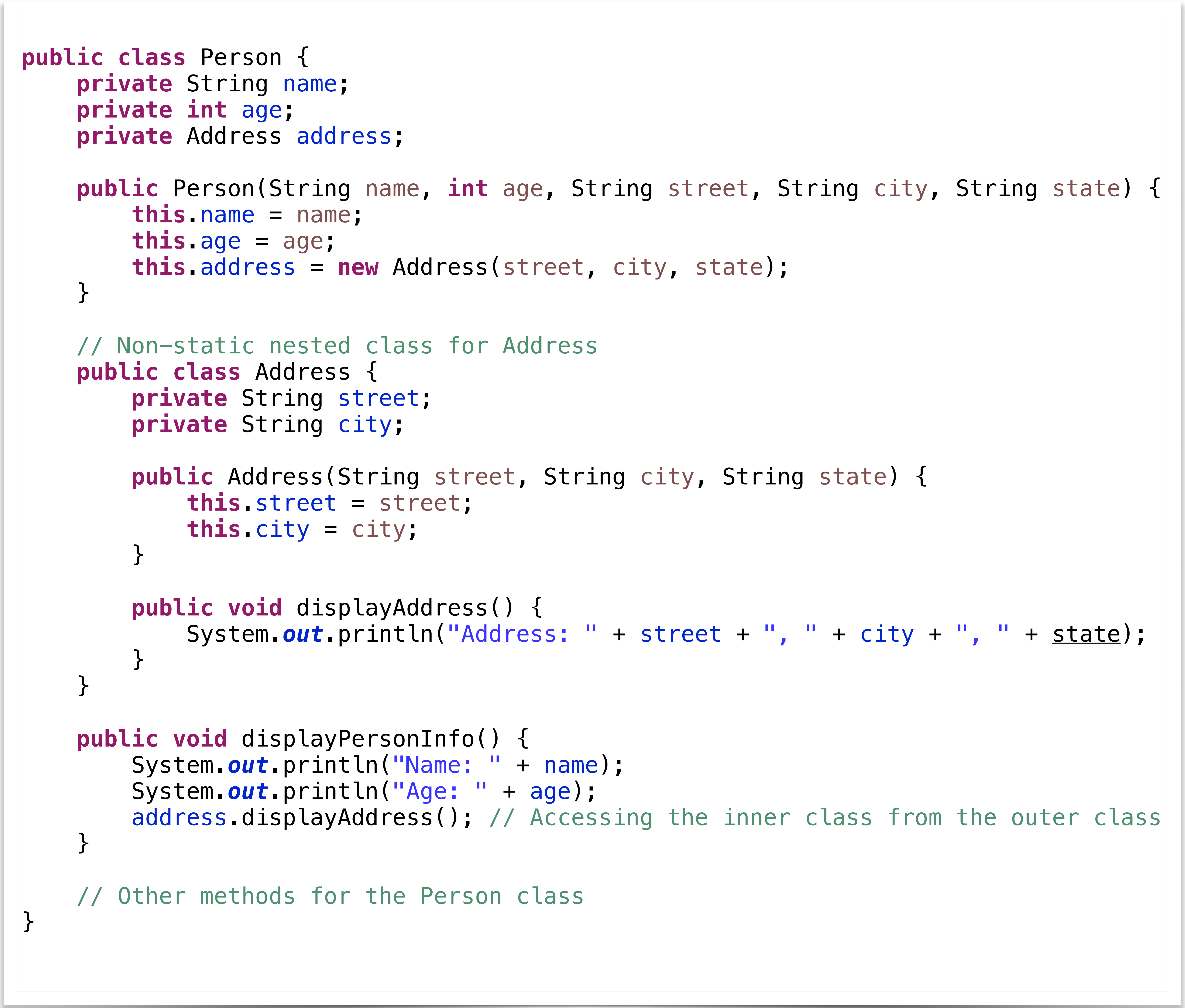
```java
public class GeometryLibrary {
    // Static nested class for Circle
    public static class Circle {
        private double radius;

        public Circle(double radius) {
            this.radius = radius;
        }

        public double calculateArea() {
            return Math.PI * radius * radius;
        }
    }

    // Static nested class for Rectangle
    public static class Rectangle {
        private double width;
        private double height;

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

        public double calculateArea() {
            return width * height;
        }
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        GeometryLibrary.Circle circle = new GeometryLibrary.Circle(5.0);
        double circleArea = circle.calculateArea();
        System.out.println("Circle Area: " + circleArea);

        GeometryLibrary.Rectangle rectangle = new GeometryLibrary.Rectangle(4.0, 6.0);
        double rectangleArea = rectangle.calculateArea();
        System.out.println("Rectangle Area: " + rectangleArea);
    }
}
```

Inner Class

An inner class is a non-static nested class, and it can access the instance variables and methods of the outer class. To create an instance of an inner class, you typically need an instance of the outer class.

```java
class OuterClass {
    class InnerClass {
        // ...
    }
}

OuterClass outerObj = new OuterClass();
OuterClass.InnerClass innerObj = outerObj.new InnerClass();
```



```java
public class Person {
    private String name;
    private int age;
    private Address address;

    public Person(String name, int age, String street, String city, String state) {
        this.name = name;
        this.age = age;
        this.address = new Address(street, city, state);
    }

    // Non-static nested class for Address
    public class Address {
        private String street;
        private String city;

        public Address(String street, String city, String state) {
            this.street = street;
            this.city = city;
        }

        public void displayAddress() {
            System.out.println("Address: " + street + ", " + city + ", " + state);
        }
    }

    public void displayPersonInfo() {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
        address.displayAddress(); // Accessing the inner class from the outer class
    }

    // Other methods for the Person class
}
```

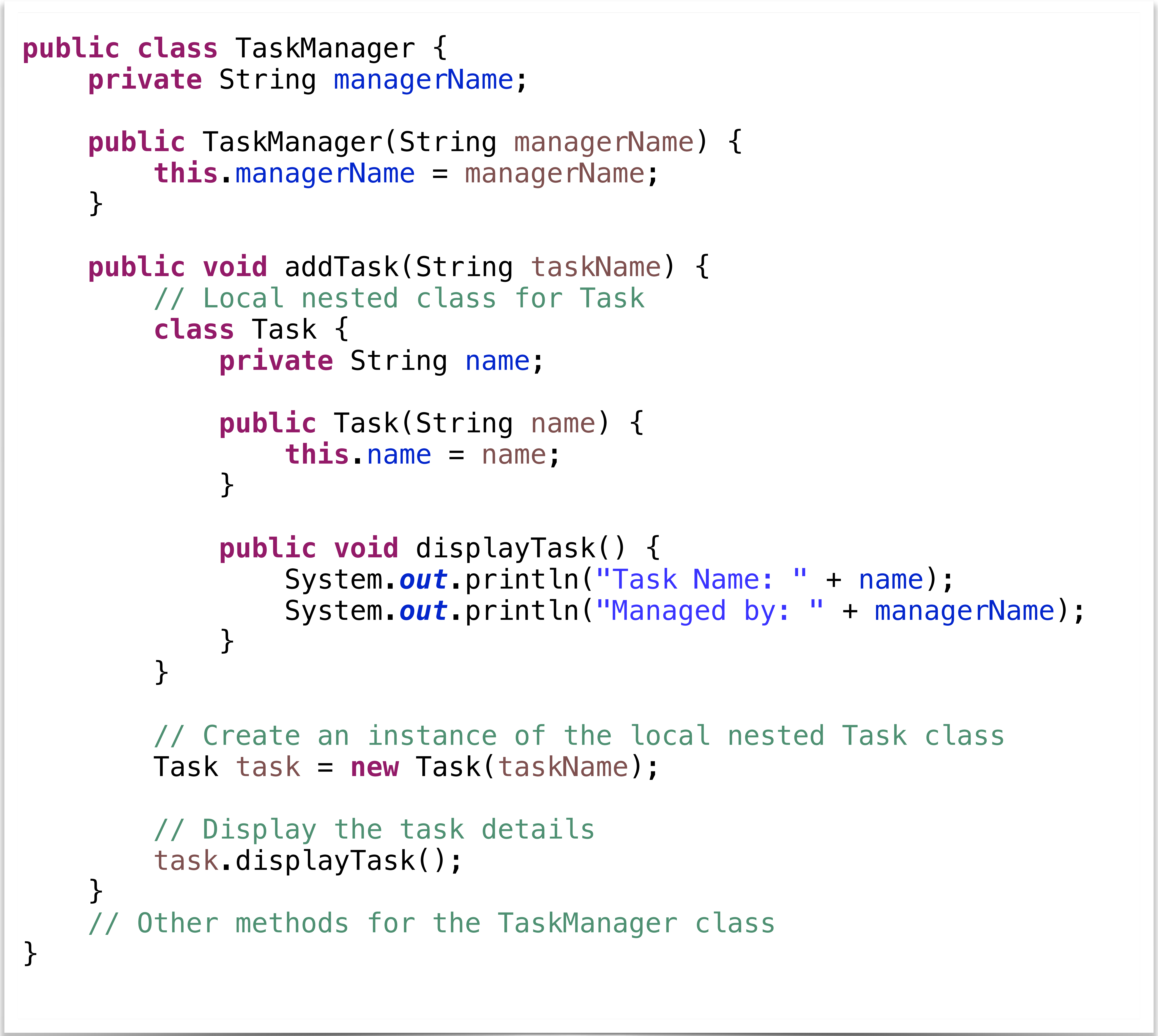
```java
public class Main {
    public static void main(String[] args) {
        Person person = new Person("John Doe", 30, "123 Main St", "Anytown", "CA");
        person.displayPersonInfo();
    }
}
```

```java
public class Person {
    private String name;
    private int age;
    private Address address;

    public Person(String name, int age, String street, String city) {
        this.name = name;
        this.age = age;
        this.address = new Address(street, city);
    }

    // Non-static nested class for Address
    public class Address {
        private String street;
        private String city;

        public Address(String street, String city) {
            this.street = street;
            this.city = city;
        }

        public void displayAddress() {
            System.out.println("Address: " + street + ", " + city + ");
        }
    }

    public void displayPersonInfo() {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
        address.displayAddress(); // Accessing the inner class from the outer class
    }

    // Other methods for the Person class
}
```

Local Class

Local classes are defined within methods, constructors, or blocks. They can only be accessed within that particular scope. Local classes are often used when you need a class for a specific, limited purpose within a method.

```java
public class OuterClass {
    void someMethod() {
        class LocalClass {
            // ...
        }
        LocalClass localObj = new LocalClass();
    }
}
```



```java
public class TaskManager {
    private String managerName;

    public TaskManager(String managerName) {
        this.managerName = managerName;
    }

    public void addTask(String taskName) {
        // Local nested class for Task
        class Task {
            private String name;

            public Task(String name) {
                this.name = name;
            }

            public void displayTask() {
                System.out.println("Task Name: " + name);
                System.out.println("Managed by: " + managerName);
            }
        }

        // Create an instance of the local nested Task class
        Task task = new Task(taskName);

        // Display the task details
        task.displayTask();
    }
    // Other methods for the TaskManager class
}
```

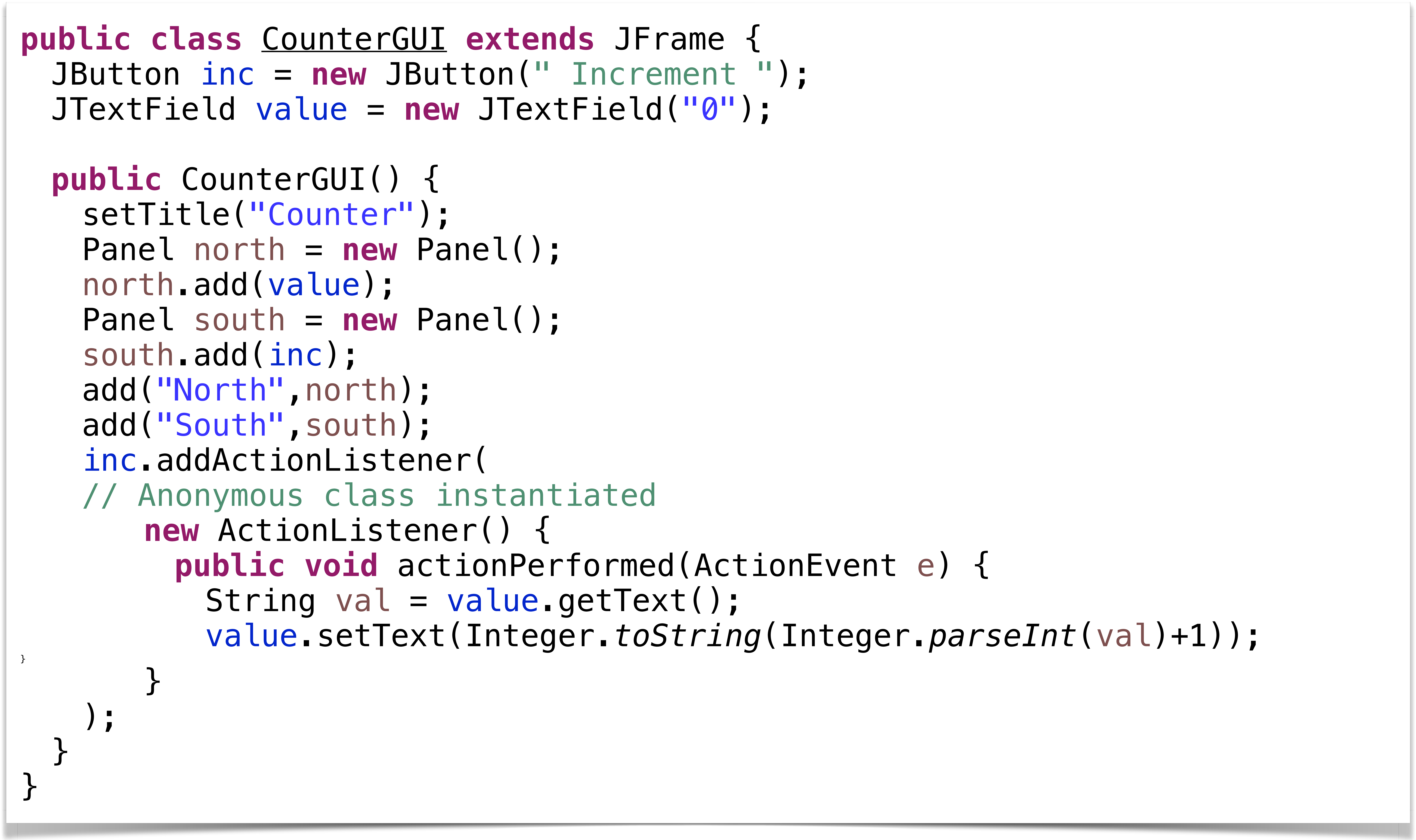
```java
public class TaskManager {
    private String managerName;

    public TaskManager(String managerName)
        this.managerName = managerName;
    }

    public void addTask(String taskName) {
        // Local nested class for Task
        class Task {
            private String name;

            public Task(String name) {
                this.name = name;
            }

            public void displayTask() {
                System.out.println("Task Name: " + name);
                System.out.println("Managed by: " + managerName);
            }
        }

        // Create an instance of the local nested Task class
        Task task = new Task(taskName);

        // Display the task details
        task.displayTask();
    }
    // Other methods for the TaskManager class
}
```

```java
public class Main {
    public static void main(String[] args) {
        TaskManager taskManager = new TaskManager("John");
        taskManager.addTask("Complete project report");
        taskManager.addTask("Schedule team meeting");
    }
}
```

Anonymous Class

Anonymous inner classes are a special type of inner class that don't have a name. They are often used when you need to provide an implementation for an interface or extend a class for a one-time, small use case.

```
interface MyInterface {
    void myMethod();
}

public class OuterClass {
    void doSomething() {
        MyInterface anonymousObj = new MyInterface() {
            @Override
            public void myMethod() {
                // Implementation of the interface method
            }
        };
    }
}
```



```
public class CounterGUI extends JFrame {
  JButton inc = new JButton(" Increment ");
  JTextField value = new JTextField("0");

  public CounterGUI() {
    setTitle("Counter");
    Panel north = new Panel();
    north.add(value);
    Panel south = new Panel();
    south.add(inc);
    add("North",north);
    add("South",south);
    inc.addActionListener(
    // Anonymous class instantiated
      new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          String val = value.getText();
          value.setText(Integer.toString(Integer.parseInt(val)+1));
}
      }
    );
  }
}
```

Lambda Expressions

A lambda expression, also known as a lambda function, is a feature that allows you to write concise, inline implementations of single-method interfaces (functional interfaces).

- Basic syntax: **(parameters) -> expression**

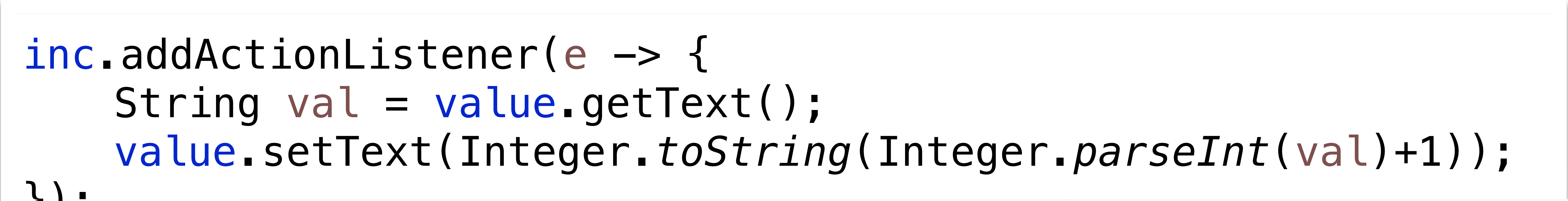
- **Parameters:** These are the input parameters that the lambda expression takes. If a lambda takes no parameters, you can simply use empty parentheses (). For a lambda with a single parameter, you can omit the parentheses around the parameter.

- **Expression:** This is the code block or statement(s) that represents the implementation of the functional interface's single abstract method. The result of the expression is the return value of the lambda function.

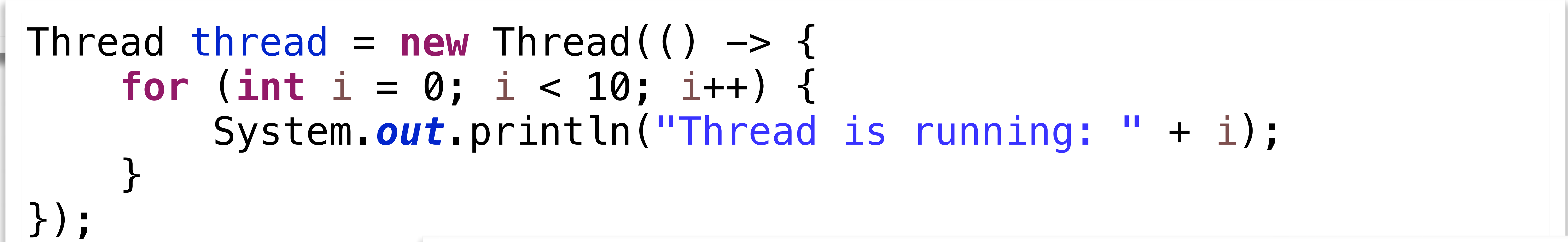
- **(int a, int b) -> a + b**

How to Use Lambda Functions

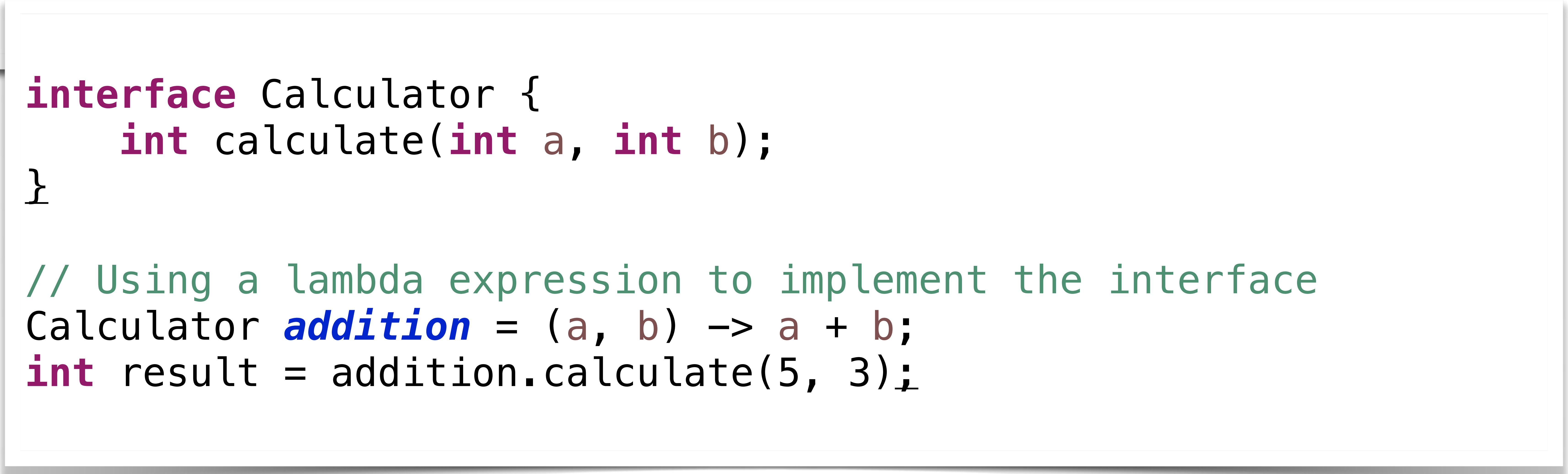
Lambda expressions are often used with functional interfaces, which are interfaces with a single abstract method.



```java
inc.addActionListener(e -> {
    String val = value.getText();
    value.setText(Integer.toString(Integer.parseInt(val)+1));
});
```



```java
Thread thread = new Thread(() -> {
    for (int i = 0; i < 10; i++) {
        System.out.println("Thread is running: " + i);
    }
});
thread.start();
```



```java
interface Calculator {
    int calculate(int a, int b);
}

// Using a lambda expression to implement the interface
Calculator addition = (a, b) -> a + b;
int result = addition.calculate(5, 3);
```

Generics

Generics in Java are a powerful feature that allow you to write code that operates on objects of different types in a type-safe and reusable manner. Generics provide a way to create classes, interfaces, and methods that work with specific types specified at compile time.

- **Type Safety:** Generics help catch type-related errors at compile time rather than runtime. This means you can write more reliable and bug-free code.

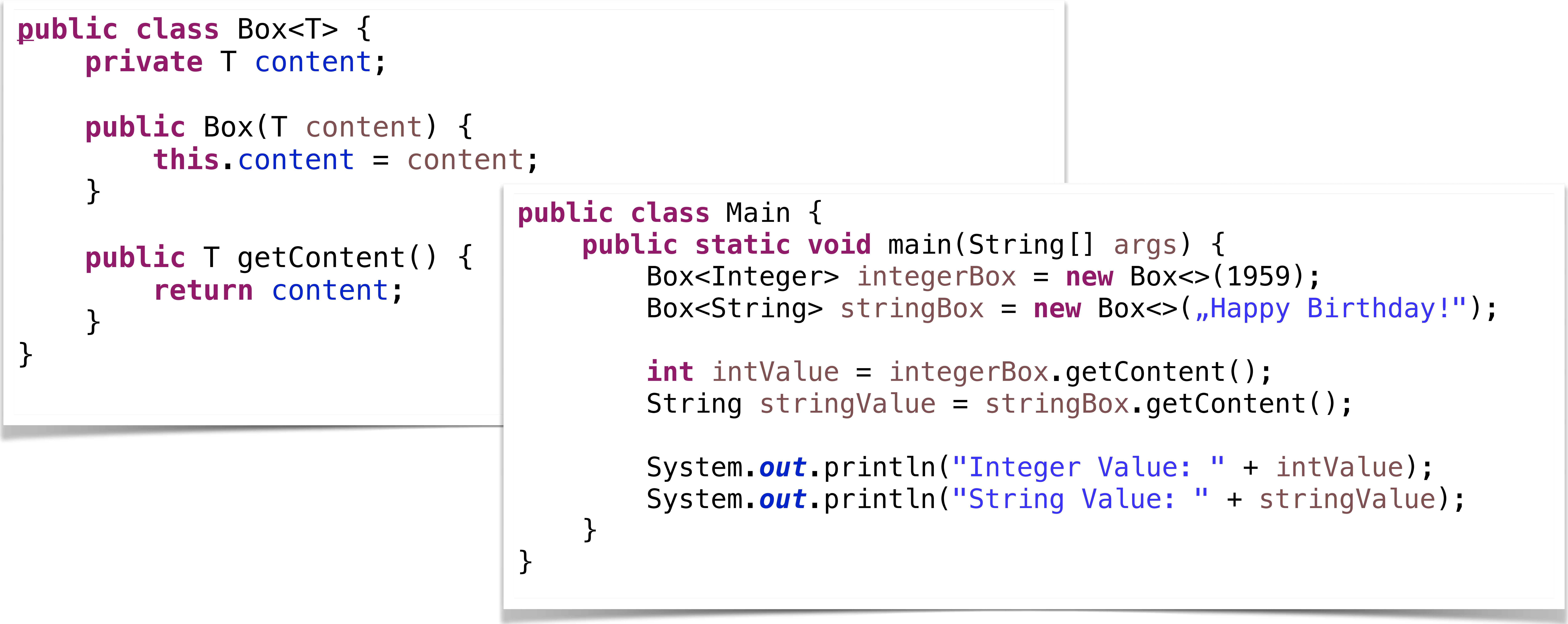
- **Code Reusability:** With generics, you can create classes, methods, and interfaces that work with a variety of data types, reducing the need for duplicate code.

- **Improved Readability:** Generics make your code more self-documenting because you can express the intended type of data explicitly.

- **Compile-Time Checks:** The Java compiler checks the correctness of your generic code at compile time, ensuring that the specified types are consistent.

Generic Classes

You can create generic classes by specifying one or more type parameters in angle brackets <T>. These type parameters represent the type(s) that the class will work with.

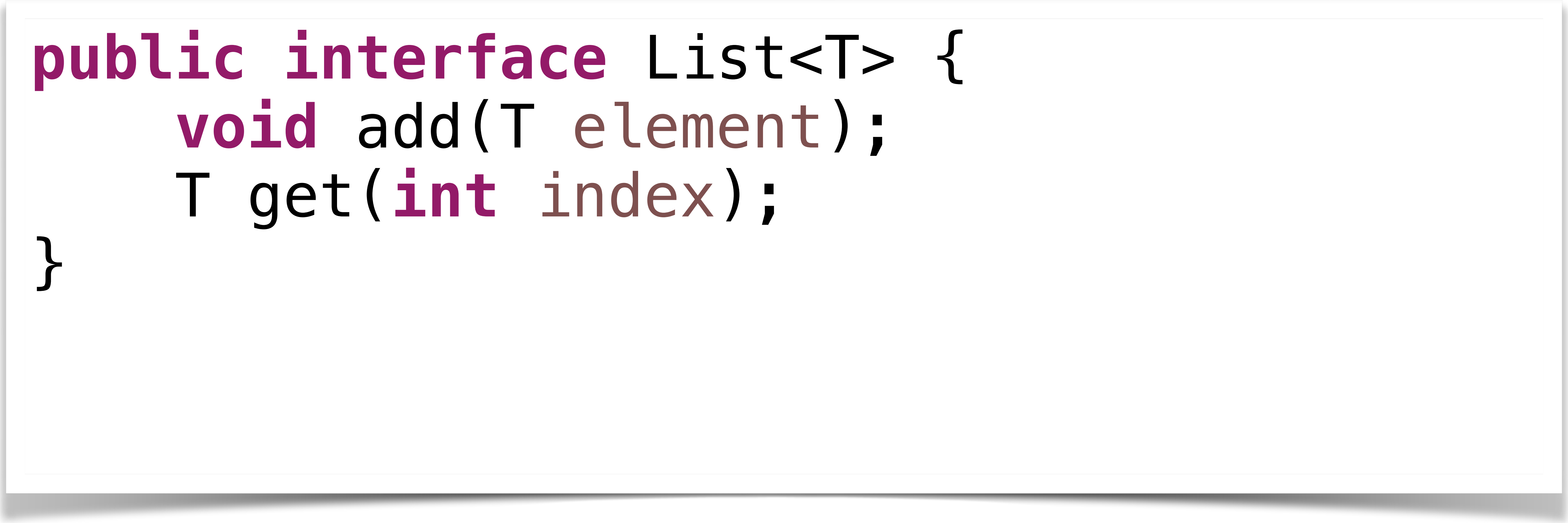


```java
public class Box<T> {
    private T content;

    public Box(T content) {
        this.content = content;
    }

    public T getContent() {
        return content;
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<>(1959);
        Box<String> stringBox = new Box<>(„Happy Birthday!");

        int intValue = integerBox.getContent();
        String stringValue = stringBox.getContent();

        System.out.println("Integer Value: " + intValue);
        System.out.println("String Value: " + stringValue);
    }
}
```

Generic Interfaces

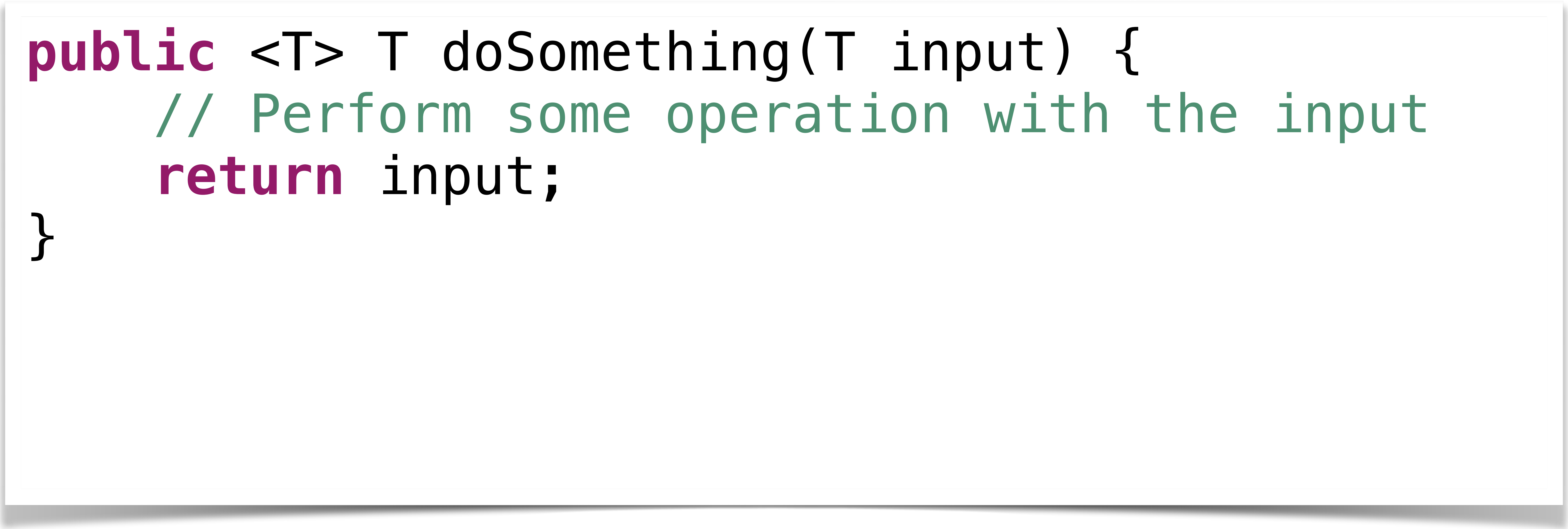
Interfaces can also be generic. They define a protocol for implementing classes with specific types.



```
public interface List<T> {
    void add(T element);
    T get(int index);
}
```

Generic Methods

You can create generic methods within non-generic classes, allowing you to use generics in a more specified way.



```java
public <T> T doSomething(T input) {
    // Perform some operation with the input
    return input;
}
```

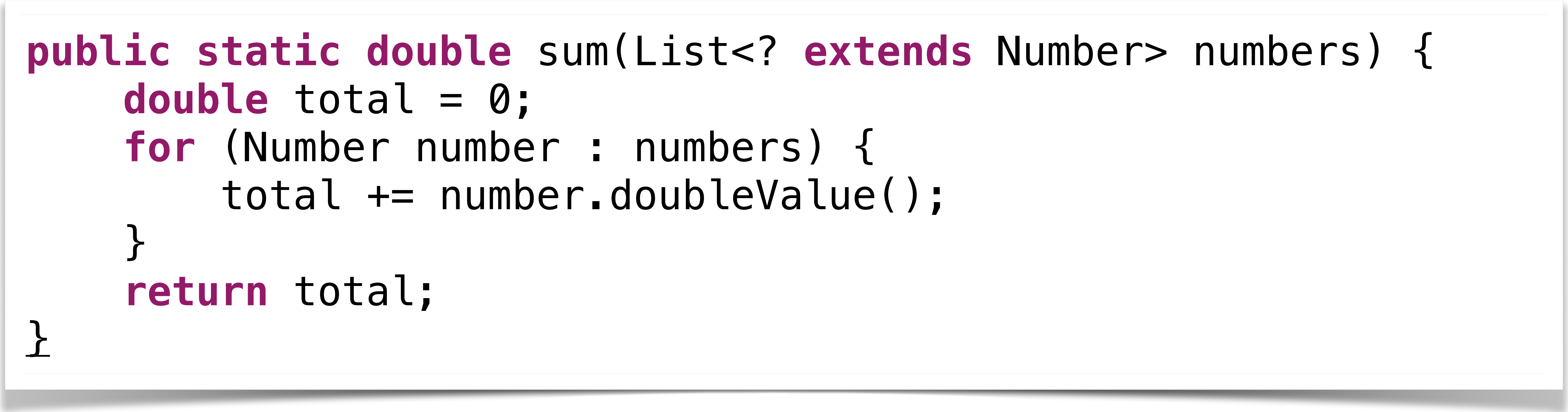
Wildcards

Wildcards are used to generalize generic types, making them more flexible and capable of working with unknown data types.

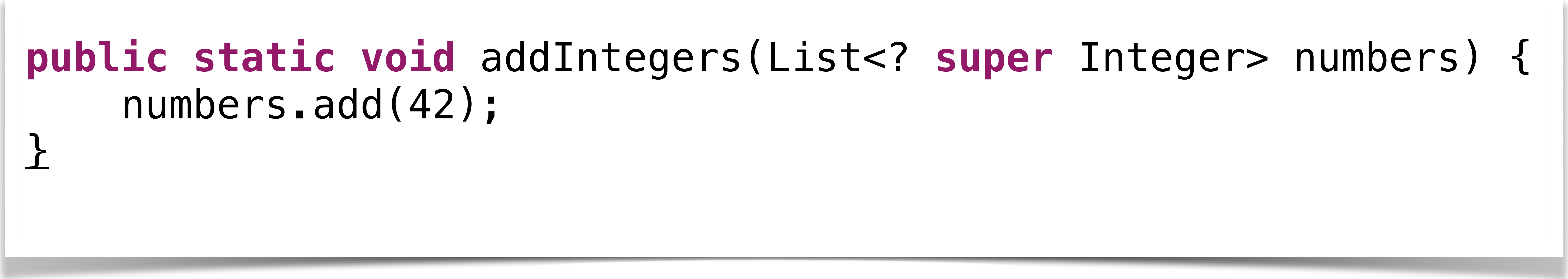
- **Upper Bounded Wildcard (*? extends T*):** This wildcard allows you to use the generic type ? for data types that are a subtype of type T or equal to type T. For example, *? extends Number* allows you to use the wildcard for any data type that is a subtype of or equal to the Number class, such as *Integer* or *Double*.

- **Lower Bounded Wildcard (*? super T*):** This wildcard allows you to use the generic type ? for data types that are supertypes of type T. This allows you to work with generic classes for data types higher in the class hierarchy than T. For example, *? super Integer* allows you to work with generic classes for data types that are supertypes of *Integer*, such as *Number*.

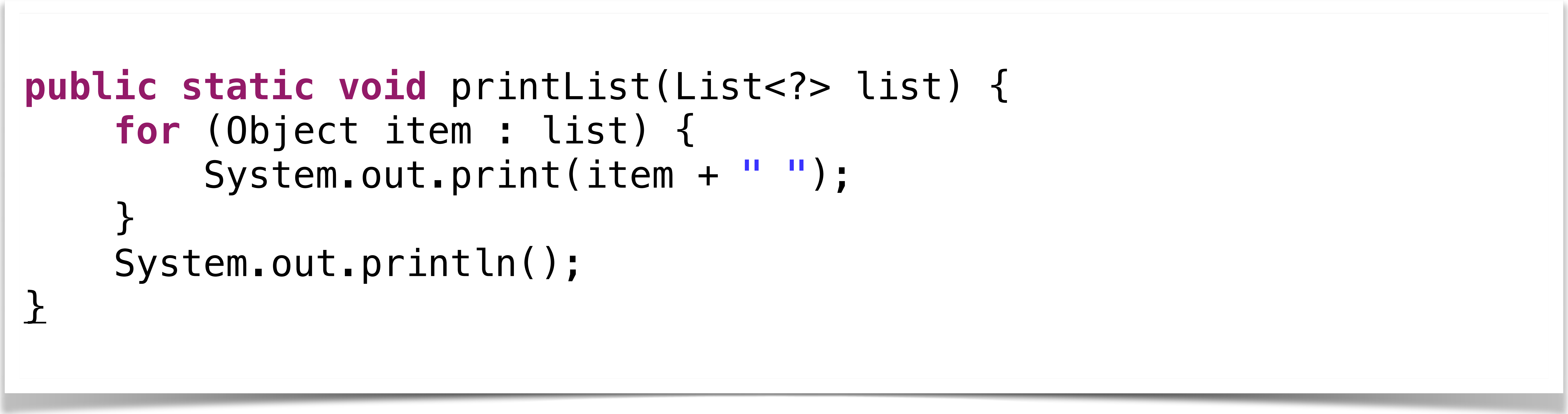
- **Unbounded Wildcard (*?*):** This wildcard allows you to work with any data type without specifying a specific type. It is suitable for situations where you do not need to know the specific data type but want to work with generic classes in a general way.



```
public static double sum(List<? extends Number> numbers) {
    double total = 0;
    for (Number number : numbers) {
        total += number.doubleValue();
    }
    return total;
}
```



```
public static void addIntegers(List<? super Integer> numbers) {
    numbers.add(42);
}
```



```
public static void printList(List<?> list) {
    for (Object item : list) {
        System.out.print(item + " ");
    }
    System.out.println();
}
```

Wrapper Classes

What about primitive types? 🤷‍♂️

- **Generic types are limited for working with object types.**
- For every primitive type exists corresponding object type.
- Java compiler converts between primitive type and its object equivalent – if it is necessary.
- **valueOf method** (e.g. *Integer.valueOf(73)*) converses from a primitive type or string to object wrapper
- **parseXXX method** (e.g. *parseFloat()*) parse String and returns specific value as primitive type
- **xxxValue method** (e.g. *floatValue()*) returns the value in a specific primitive type.

List of Wrapper Classes

How to create primitive types from String

Primitive	**Wrapper class**	**Conversion method from string**
boolean	Boolean	Boolean.parseBoolean(String s)
char	Character	…
byte	Byte	Byte.parseByte(String s) Byte.parseByte(String s, int radix)
short	Short	Short.parseShort(String s) Short.parseShort(String s, int radix)
int	Integer	Integer.parseInt(String s) Integer.parseInt(String s, int radix)
long	Long	Long.parseLong(String s) Long.parseLong(String s, int radix)
float	Float	Float.parseFloat(String s)
double	Double	Double.parseDouble(String s)

4th Part: Collection Framework

- Collections interfaces
- Collections implementation
- Collections utility class

Java Collection Framework

The Java Collections Framework (JCF) is a fundamental and comprehensive set of classes and interfaces in Java that provide various data structures and algorithms to work with collections of objects.

- **Interfaces:** These define the common methods and behaviors for different types of collections. The core collection interfaces include List, Set, Map, and Queue.
- **Classes:** These are concrete implementations of the collection interfaces. Common classes include ArrayList, LinkedList, HashSet, TreeSet, HashMap, and TreeMap, among others.
- **Algorithms:** The framework includes various utility methods for working with collections, such as sorting, searching, and shuffling.
- **Exceptions:** Specific exceptions are provided for situations like attempting to access an element that doesn't exist (NoSuchElementException) or adding duplicate elements to a Set (IllegalArgumentException).
- **Iteration:** Iteration is a common operation when working with collections. The framework provides iterators to traverse through collections.
- **Comparator:** The Comparator interface is used for custom sorting of objects in collections.

Collection Interface

The Collection interface is the root interface in the Java Collections Framework. It represents a basic collection of objects, and it defines methods common to all collection types, such as add, remove, and contains.

- **List Interface:** Extends the Collection interface. Lists are ordered collections that allow duplicate elements. Key implementations include ArrayList and LinkedList.

- **Set Interface:** Extends the Collection interface. Sets are collections that do not allow duplicate elements. Key implementations include HashSet, TreeSet, and LinkedHashSet.

- **Queue Interface:** Extends the Collection interface. Queues are specialized collections for managing elements in a first-in-first-out (FIFO) order. Key implementations include LinkedList and PriorityQueue.

List Interface

List is the ordered (with defined index for every element) collection that may contain duplicate elements.

- *add(int, E), set(int, E), addAll(int, Collection<E>), get(int):E, remove(int):E* **– add/remove elements to/from given position**
- *indexOf(Object):int, lastIndexOf(Object):int* **– find position of a given object**
- *listIterator():ListIterator* **– return iterator that allows forward/backward browsing**

Set Interface

Set is the collection of elements that does not contain duplicates.

- *add(E):boolean, addAll(Collection<E>), contains(Object):boolean* – **added constraints to inherited methods**

- **SortedSet:** extends the Set in a way that enables the ordering

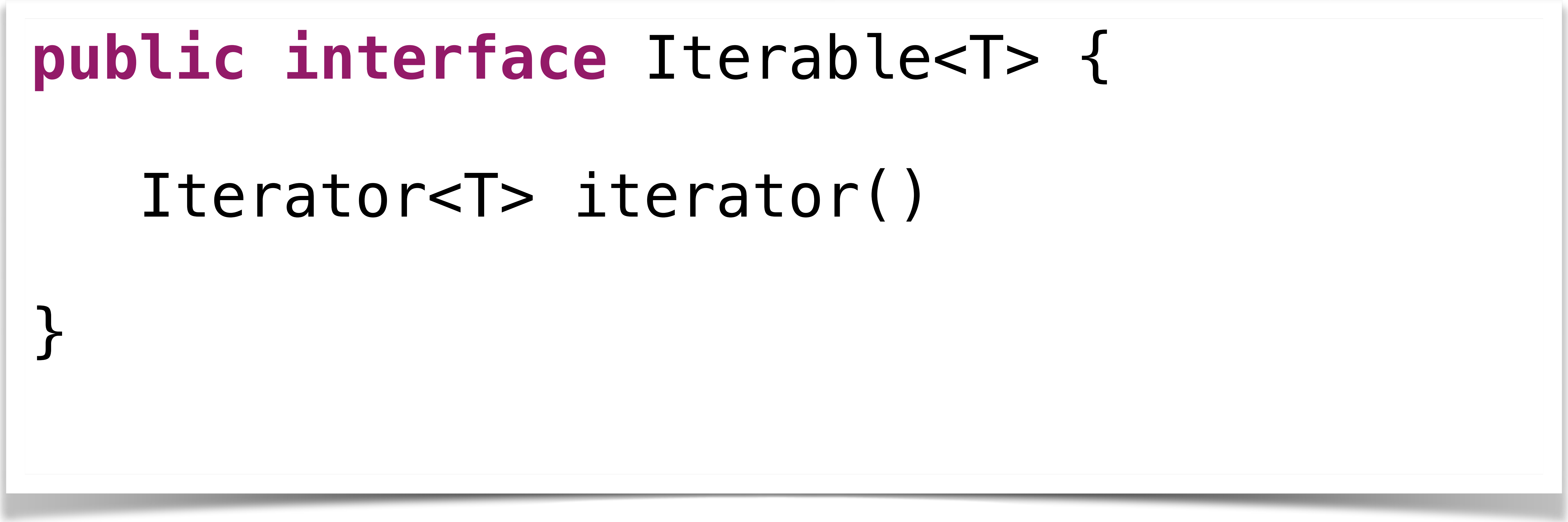
Queue Interface

Queue is a list of elements with a first in first out ordering.

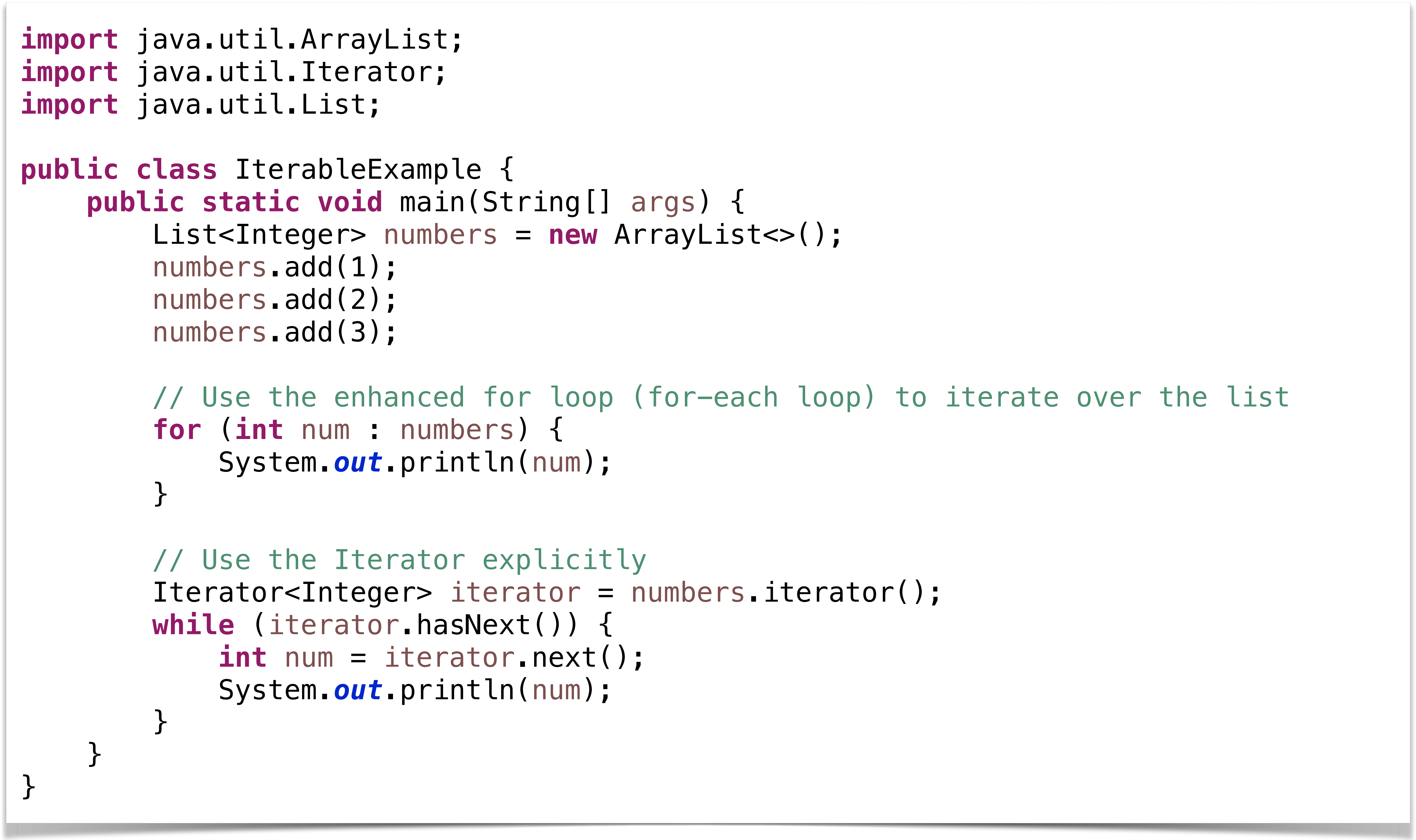
- *-add(E), offer(E)* – **enqueue**
- *-remove(): E, poll(): E* – **dequeue**
- *-element():E, peek():E* – **retrieves but not remove**
- **Deque:** extends the Queue by a protocol that is required by a stack (Last In First Out)

Iterable

Iterable is a base type for Collection that provides a comfort way for a loop construction



```
public interface Iterable<T> {

   Iterator<T> iterator()

}
```

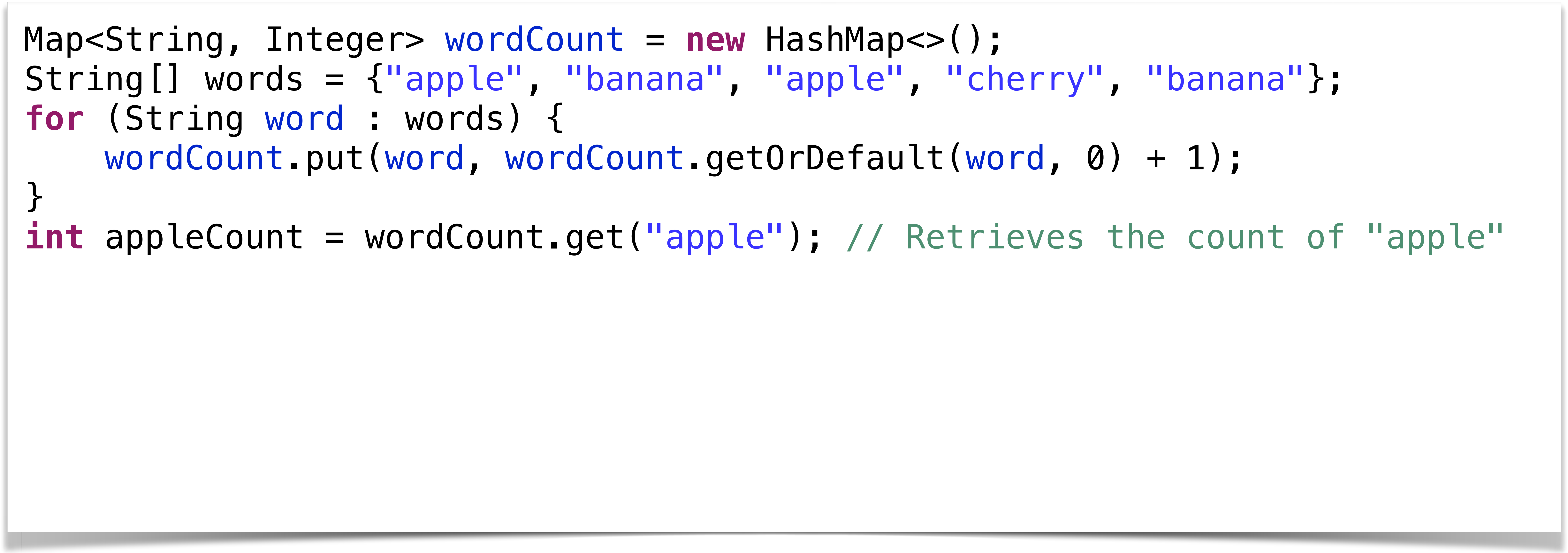


```java
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class IterableExample {
    public static void main(String[] args) {
        List<Integer> numbers = new ArrayList<>();
        numbers.add(1);
        numbers.add(2);
        numbers.add(3);

        // Use the enhanced for loop (for-each loop) to iterate over the list
        for (int num : numbers) {
            System.out.println(num);
        }

        // Use the Iterator explicitly
        Iterator<Integer> iterator = numbers.iterator();
        while (iterator.hasNext()) {
            int num = iterator.next();
            System.out.println(num);
        }
    }
}
```

Map Interface

The Map interface represents a collection of key-value pairs, where each key is associated with exactly one value.

- **SortedMap Interface:** Extends the Map interface. Sorted maps are maps that maintain their keys in sorted order. Key implementations include HashMap and TreeMap.



```java
Map<String, Integer> wordCount = new HashMap<>();
String[] words = {"apple", "banana", "apple", "cherry", "banana"};
for (String word : words) {
    wordCount.put(word, wordCount.getOrDefault(word, 0) + 1);
}
int appleCount = wordCount.get("apple"); // Retrieves the count of "apple"
```

Collections Implementation

- **Lists:**
  - **ArrayList:** Implements a dynamic array, which can dynamically grow and shrink as needed.
  - **LinkedList:** Implements a doubly-linked list, suitable for efficient element insertion and removal.
- **Sets:**
  - **HashSet:** Implements a set using a hash table, which provides fast access but does not guarantee order.
  - **LinkedHashSet:** Extends HashSet and maintains insertion order.
  - **TreeSet:** Implements a set using a red-black tree, which provides elements in sorted order.
- **Queues:**
  - **LinkedList:** Can be used as a queue with methods like offer, poll, and peek.
  - **PriorityQueue:** Implements a priority queue based on a heap data structure.
- **Maps:**
  - **HashMap:** Implements a map using a hash table for key-value pairs.
  - **LinkedHashMap:** Extends HashMap and maintains order of key-value pairs based on insertion order or access order.
  - **TreeMap:** Implements a map using a red-black tree for key-value pairs sorted by key.

Collections - Table View

Interface	Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Collection	HashSet	ArrayList	TreeSet	LinkedList	LinkedHashSet
-> List		ArrayList		LinkedList	
-> Set	HashSet		TreeSet		LinkedHashSet
-> Queue		ArrayDeque		LinkedList	
—> Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

Collections Utility Class

The goal is to perform various operations on collections (lists, sets, maps, etc.) and algorithms related to collections. It offers a collection of static methods to manipulate and work with collections in a more convenient and efficient manner.

- **Sorting Collections:** You can use methods like sort to sort lists in natural order or using a custom comparator.
- **Searching:** Methods like binarySearch are used to perform binary searches on sorted lists.
- **Shuffling:** The shuffle method randomizes the order of elements in a list.
- **Reversing:** You can reverse the order of elements in a list using reverse.
- **Filling Collections:** Methods like fill can be used to fill a list with a specified value.
- **Checking for Empty Collections:** The *empty* methods check if a collection is empty.

5th Part: I/O Streams

- Byte streams
- Character streams
- Streams for network operations
- Object streams - serialisation

Input and Output Streams

Input and output streams are used to read data from and write data to various sources such as files, network connections, or memory buffers

- **Byte Streams**: Used to handle raw binary data.
  - InputStream (for reading bytes)
  - OutputStream (for writing bytes)
- **Character Streams:** Used to handle character data (text) using encoding like UTF-8.
  - Reader (for reading characters)
  - Writer (for writing characters)

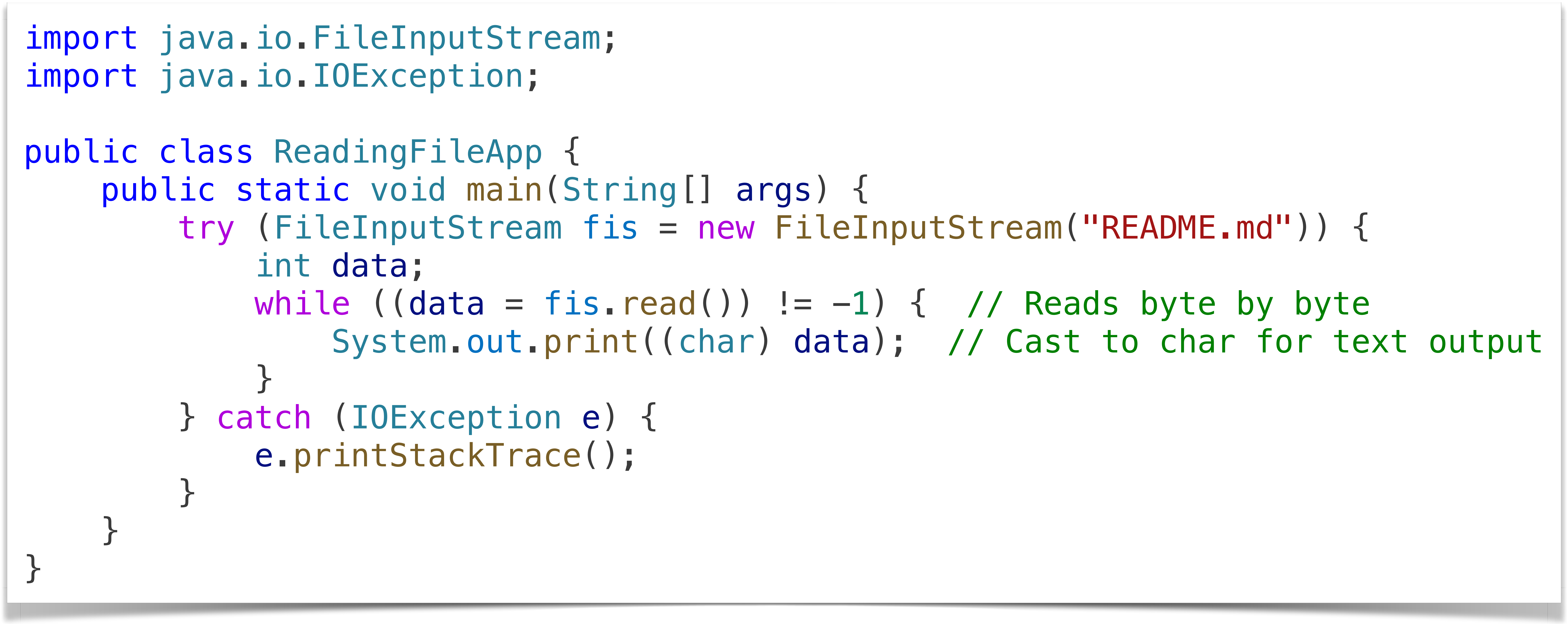
Byte Streams

InputStream and OutputStream

- **InputStream**: Used for reading byte data.
  - Commonly used subclasses:
    - FileInputStream: Reads from a file.
    - BufferedInputStream: Buffers the input for efficient reading.
    - ByteArrayInputStream: Reads from a byte array.
- **OutputStream**: Used for writing byte data.
  - Commonly used subclasses:
    - FileOutputStream: Writes to a file.
    - BufferedOutputStream: Buffers the output for efficient writing.
    - ByteArrayOutputStream: Writes to a byte array.

Reading from a File

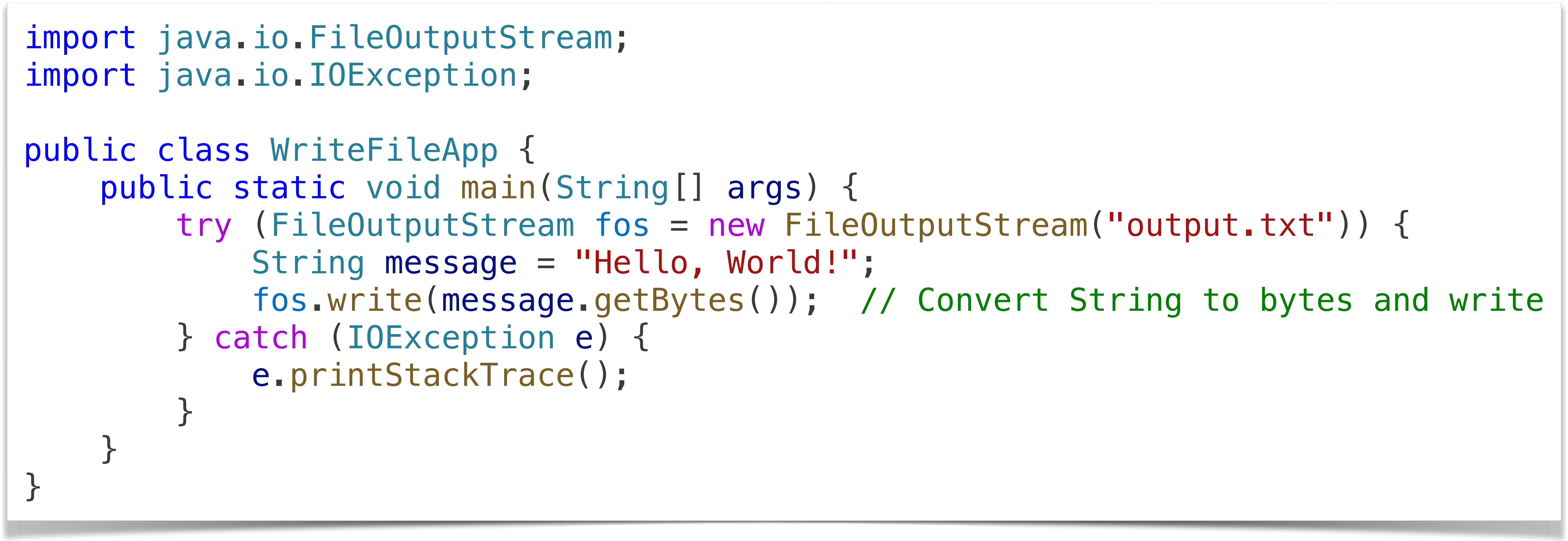
Reading from a file using FileInputStream



```java
import java.io.FileInputStream;
import java.io.IOException;

public class ReadingFileApp {
    public static void main(String[] args) {
        try (FileInputStream fis = new FileInputStream("README.md")) {
            int data;
            while ((data = fis.read()) != -1) {  // Reads byte by byte
                System.out.print((char) data);  // Cast to char for text output
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Writing to a File

Writing to a file using FileOutputStream



```java
import java.io.FileOutputStream;
import java.io.IOException;

public class WriteFileApp {
    public static void main(String[] args) {
        try (FileOutputStream fos = new FileOutputStream("output.txt")) {
            String message = "Hello, World!";
            fos.write(message.getBytes());  // Convert String to bytes and write
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

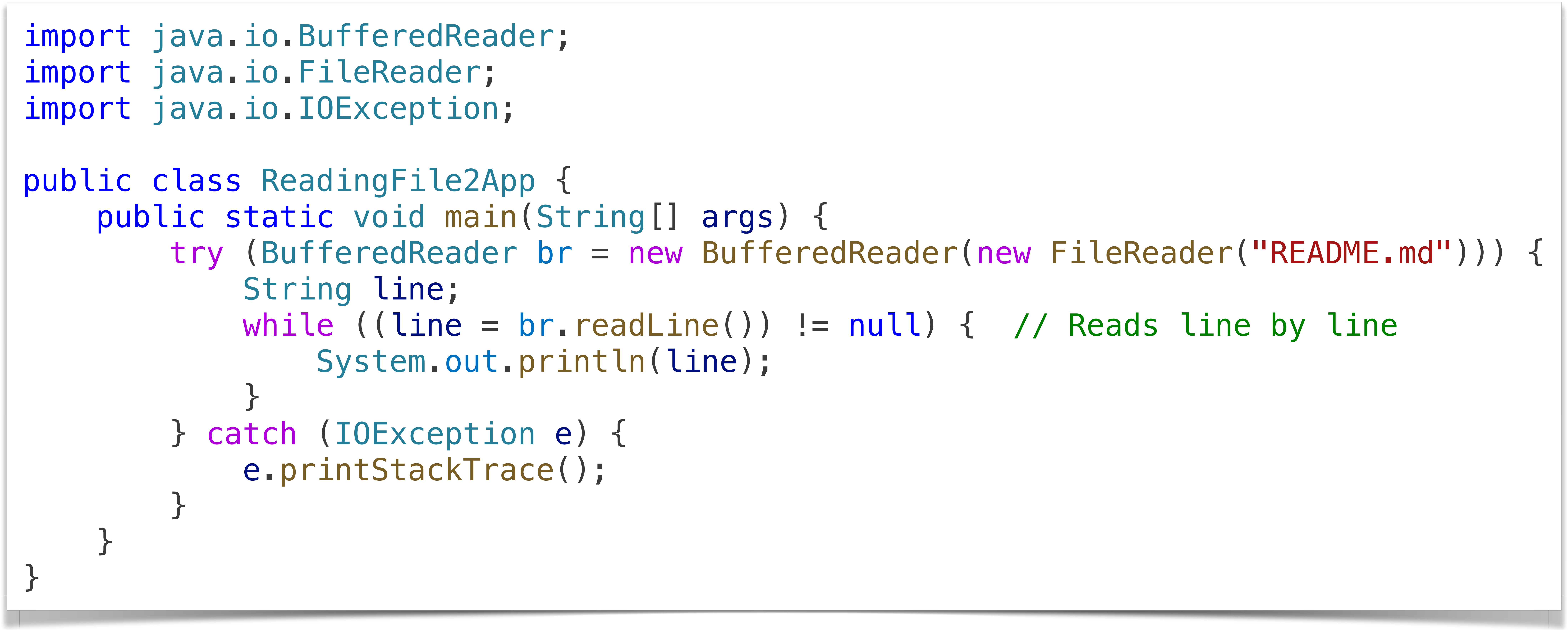
Character Streams

Reader and Writer

- **Reader:** Used for reading character data.
  - Commonly used subclasses:
    - FileReader: Reads characters from a file.
    - BufferedReader: Buffers the input for efficient reading.
- **Writer**: Used for writing character data.
  - Commonly used subclasses:
    - FileWriter: Writes characters to a file.
    - BufferedWriter: Buffers the output for efficient writing.

Reading from a File II

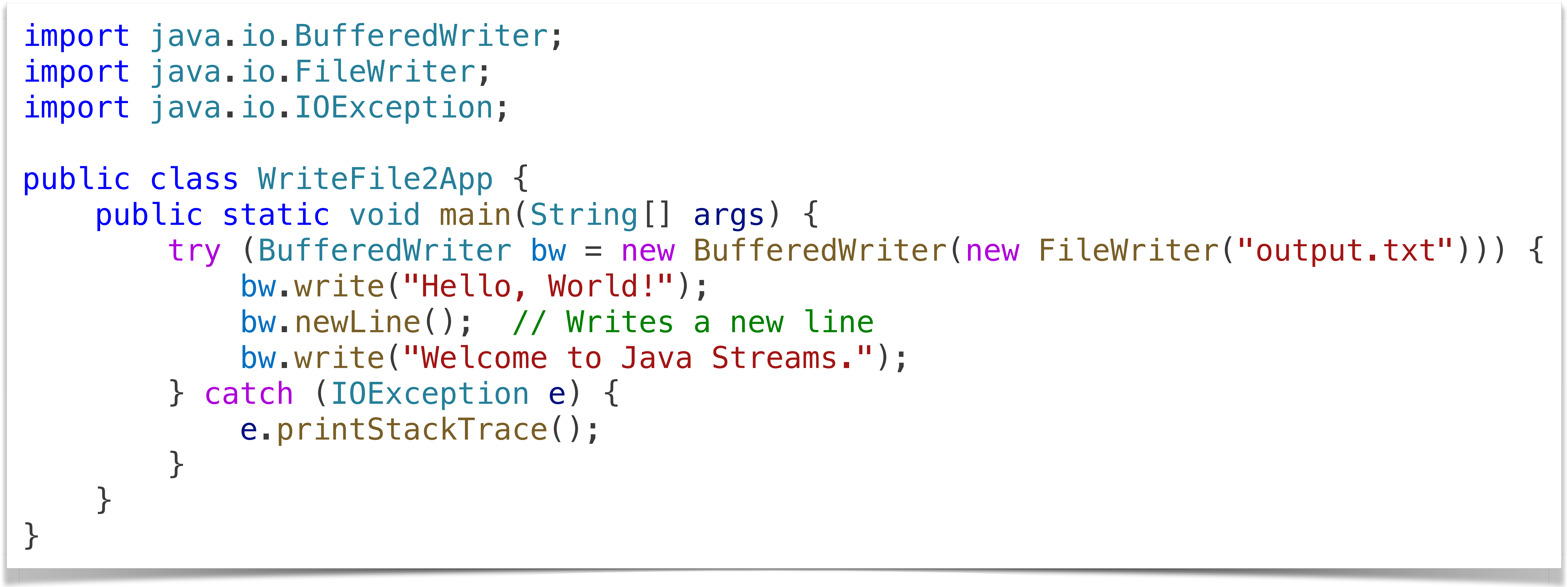
Reading from a file using BufferedReader



```java
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class ReadingFile2App {
    public static void main(String[] args) {
        try (BufferedReader br = new BufferedReader(new FileReader("README.md"))) {
            String line;
            while ((line = br.readLine()) != null) {  // Reads line by line
                System.out.println(line);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Writing to a File II

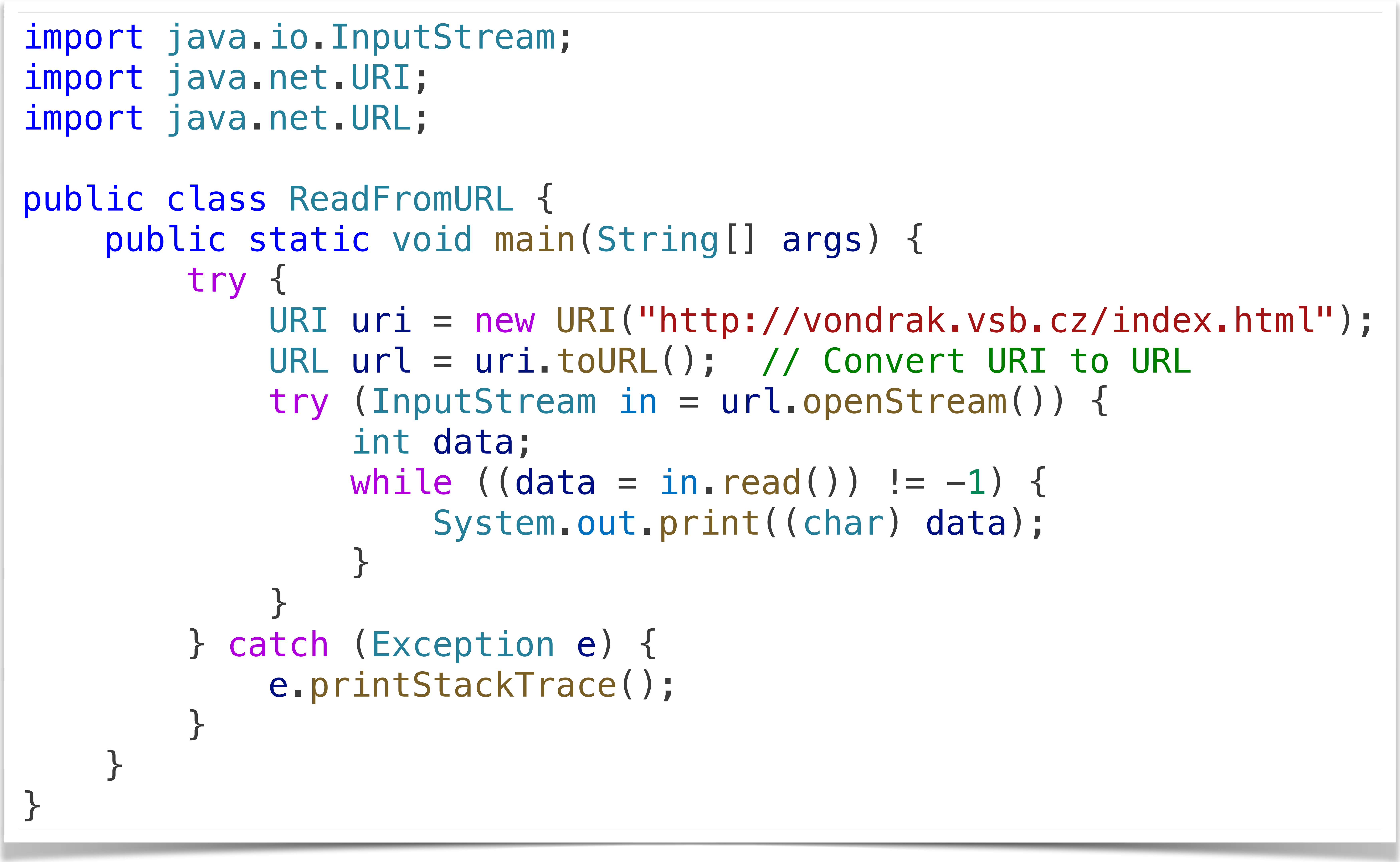
Writing to a file using BufferedWriter



```
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;

public class WriteFile2App {
    public static void main(String[] args) {
        try (BufferedWriter bw = new BufferedWriter(new FileWriter("output.txt"))) {
            bw.write("Hello, World!");
            bw.newLine();  // Writes a new line
            bw.write("Welcome to Java Streams.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Streams for Network Operations

You can also use InputStream and OutputStream with network sockets for reading and writing data over a network.



```java
import java.io.InputStream;
import java.net.URI;
import java.net.URL;

public class ReadFromURL {
    public static void main(String[] args) {
        try {
            URI uri = new URI("http://vondrak.vsb.cz/index.html");
            URL url = uri.toURL();  // Convert URI to URL
            try (InputStream in = url.openStream()) {
                int data;
                while ((data = in.read()) != -1) {
                    System.out.print((char) data);
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

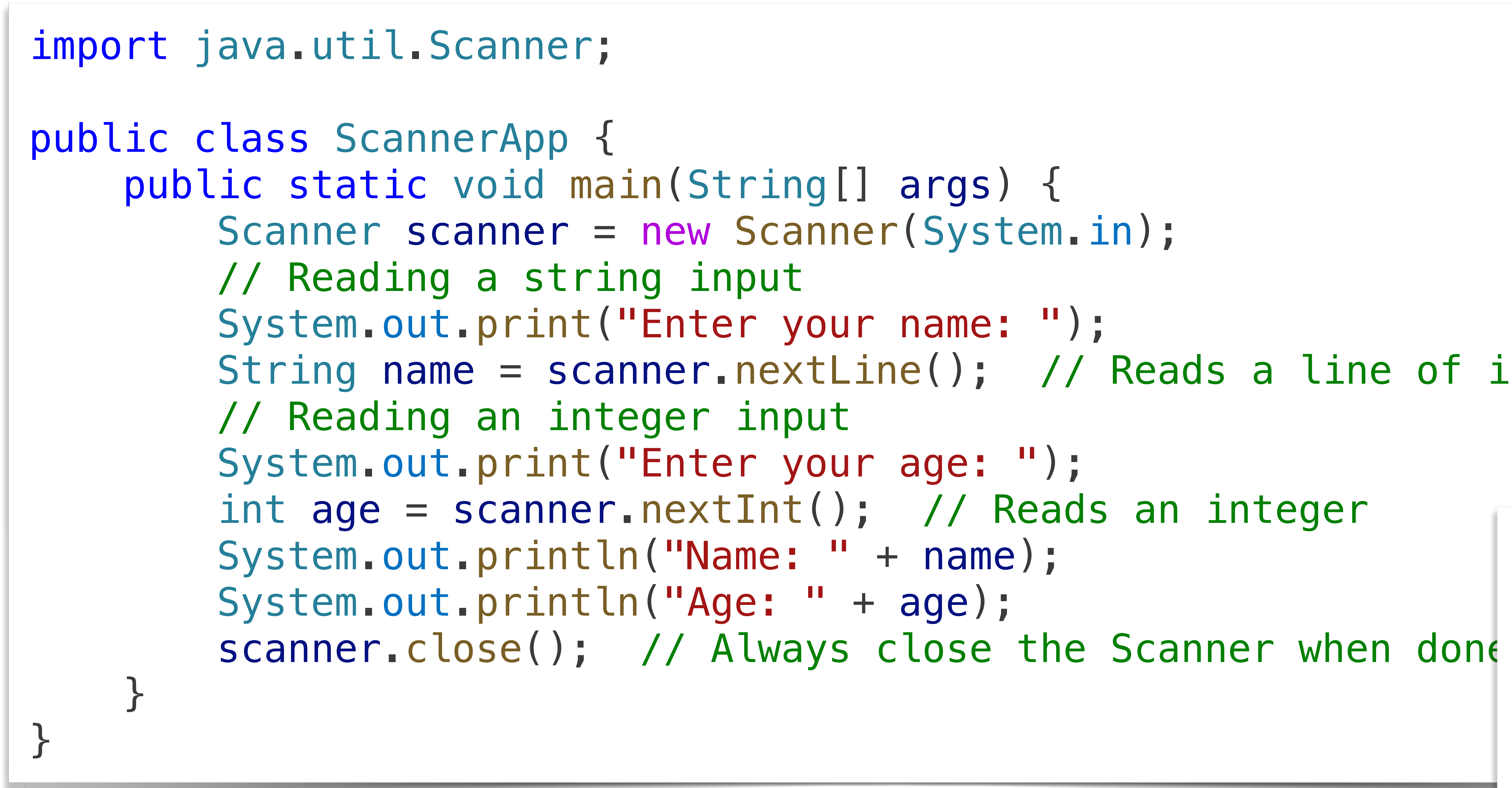
Scanning

The Scanner class in Java is used to parse and read user input from various sources, such as standard input (keyboard), files, or strings.

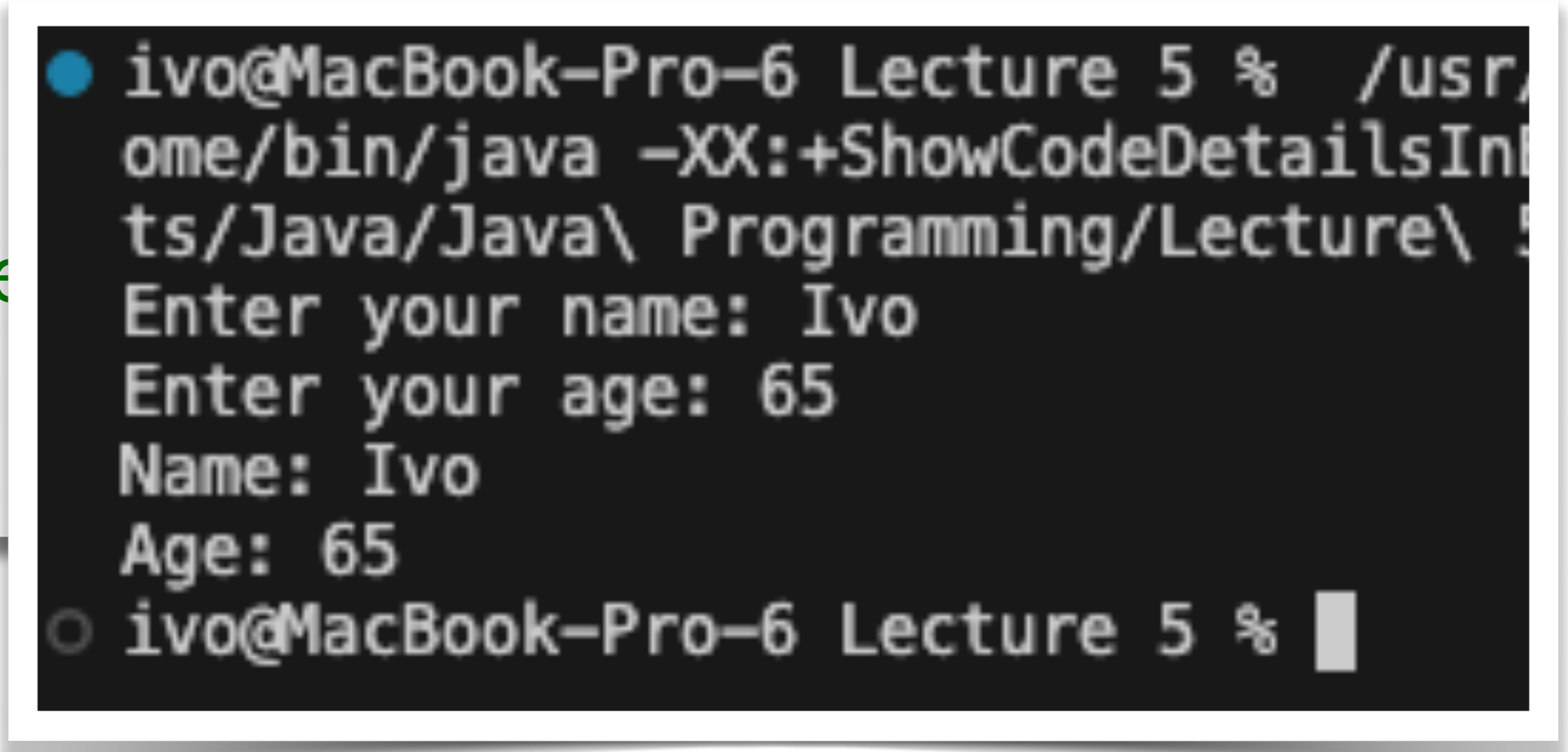
- **Key Methods of Scanner**:
  - nextLine(): Reads a full line of input as a String.
  - next(): Reads the next token (word) as a String.
  - nextInt(): Reads the next token as an int.
  - nextDouble(): Reads the next token as a double.
  - hasNext(): Checks if there’s another token available to read.
  - hasNextInt(): Checks if the next token is an integer.
  - close(): Closes the scanner to release the underlying resource (e.g., standard input).

Reading Input from the Keyboard

Scanner breaks the input into tokens based on delimiters (like spaces or newline characters), making it convenient to read and process different types of input.



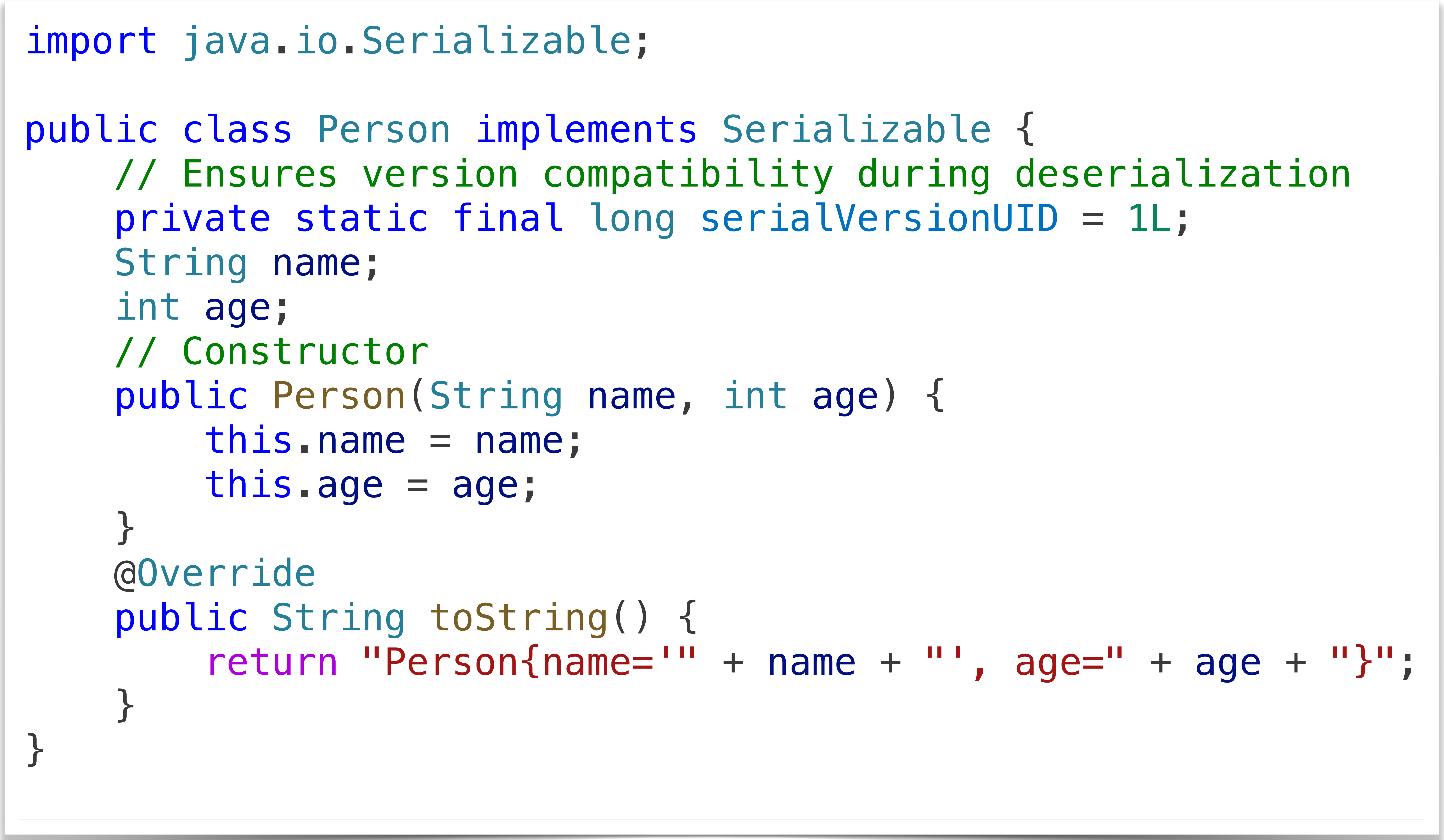
```java
import java.util.Scanner;

public class ScannerApp {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        // Reading a string input
        System.out.print("Enter your name: ");
        String name = scanner.nextLine();  // Reads a line of input
        // Reading an integer input
        System.out.print("Enter your age: ");
        int age = scanner.nextInt();  // Reads an integer
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
        scanner.close();  // Always close the Scanner when done
    }
}
```



```
● ivo@MacBook-Pro-6 Lecture 5 %  /usr
  ome/bin/java -XX:+ShowCodeDetailsIn
  ts/Java/Java\ Programming/Lecture\
  Enter your name: Ivo
  Enter your age: 65
  Name: Ivo
  Age: 65
○ ivo@MacBook-Pro-6 Lecture 5 % █
```

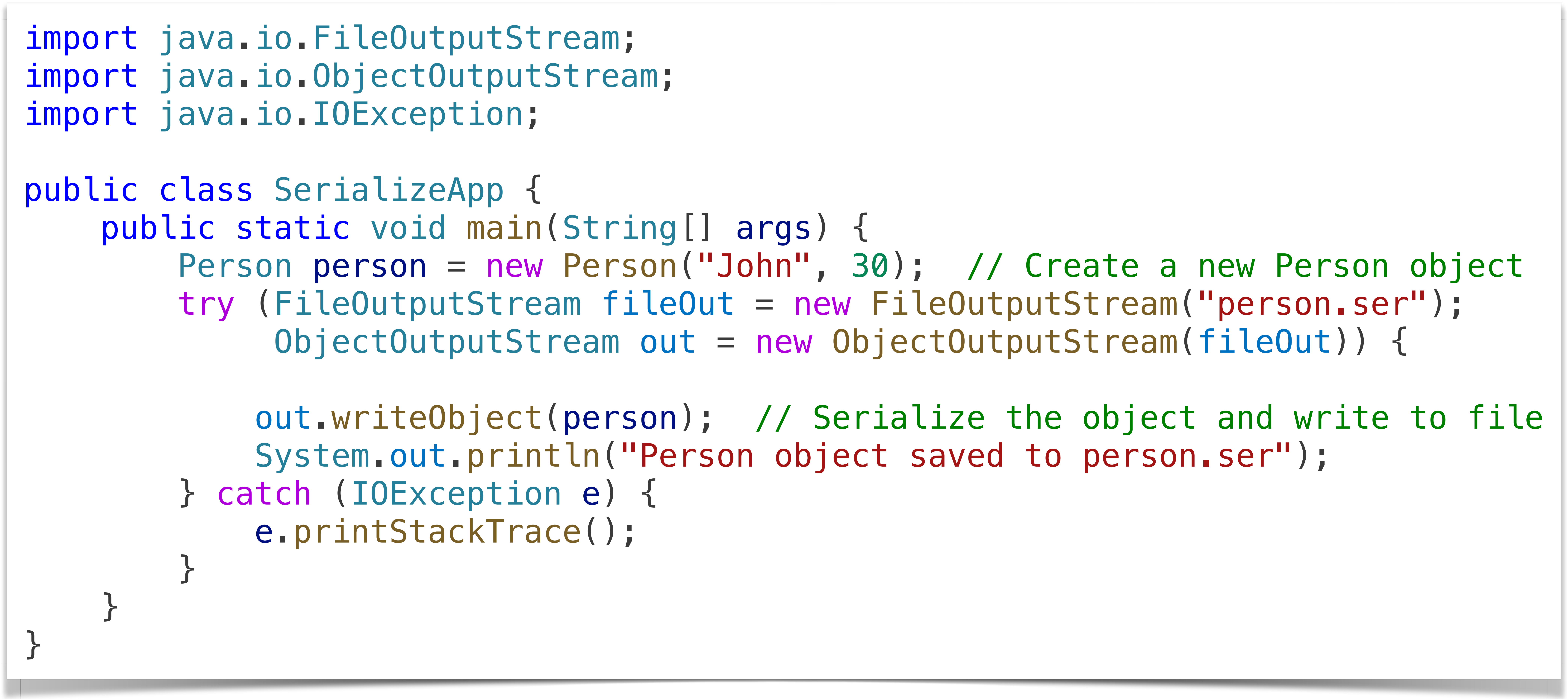
Object Streams - Serialization

In Java, you can save objects to a file using serialization. Serialization allows an object to be converted into a byte stream, which can then be saved to a file. Later, you can read the byte stream from the file and deserialize it to reconstruct the object.



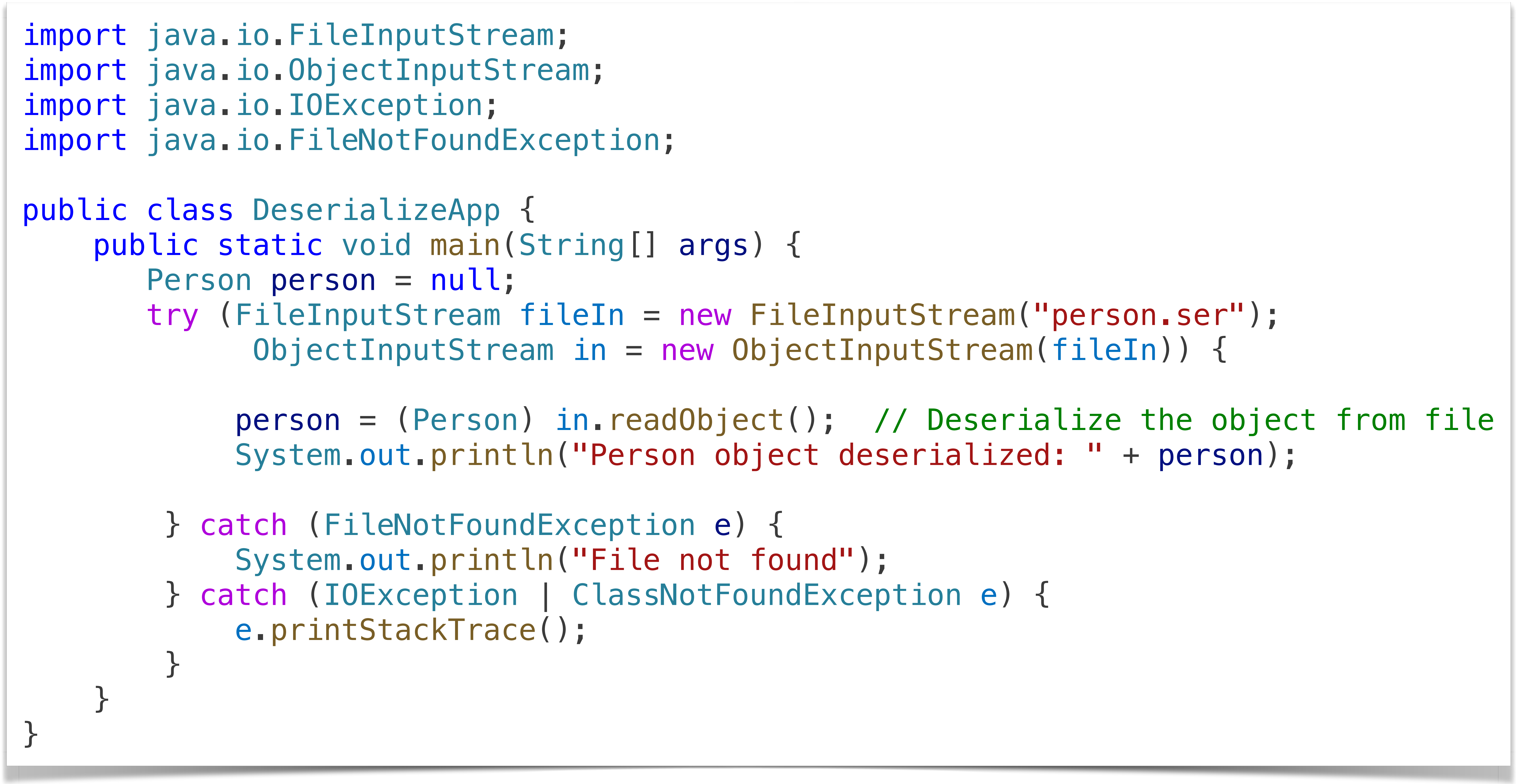
```java
import java.io.Serializable;

public class Person implements Serializable {
    // Ensures version compatibility during deserialization
    private static final long serialVersionUID = 1L;
    String name;
    int age;
    // Constructor
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    @Override
    public String toString() {
        return "Person{name='" + name + "', age=" + age + "}";
    }
}
```

Save (Serialize) the Object



```java
import java.io.FileOutputStream;
import java.io.ObjectOutputStream;
import java.io.IOException;

public class SerializeApp {
    public static void main(String[] args) {
        Person person = new Person("John", 30);  // Create a new Person object
        try (FileOutputStream fileOut = new FileOutputStream("person.ser");
             ObjectOutputStream out = new ObjectOutputStream(fileOut)) {

            out.writeObject(person);  // Serialize the object and write to file
            System.out.println("Person object saved to person.ser");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Read (Deserialize) the Object



```
import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.io.IOException;
import java.io.FileNotFoundException;

public class DeserializeApp {
    public static void main(String[] args) {
       Person person = null;
       try (FileInputStream fileIn = new FileInputStream("person.ser");
             ObjectInputStream in = new ObjectInputStream(fileIn)) {

            person = (Person) in.readObject();  // Deserialize the object from file
            System.out.println("Person object deserialized: " + person);

        } catch (FileNotFoundException e) {
            System.out.println("File not found");
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

Final Notes

Object streams can store complex structure of objects connected by references – it can handle also loops

- The class must implement the Serializable interface, which is a marker interface (it does not have any methods).

- The serialVersionUID is used to ensure that a serialized object can be deserialized correctly even if the class has changed slightly (version control).

- Use try-with-resources to automatically close resources such as file streams.

- Attributes marked as a transient are not saved during serialization.

  - *transient private String personID;*

6th Part: Multithreading

- Threads
- Synchronization
- Producer-Consumer problem
- Locks and conditions

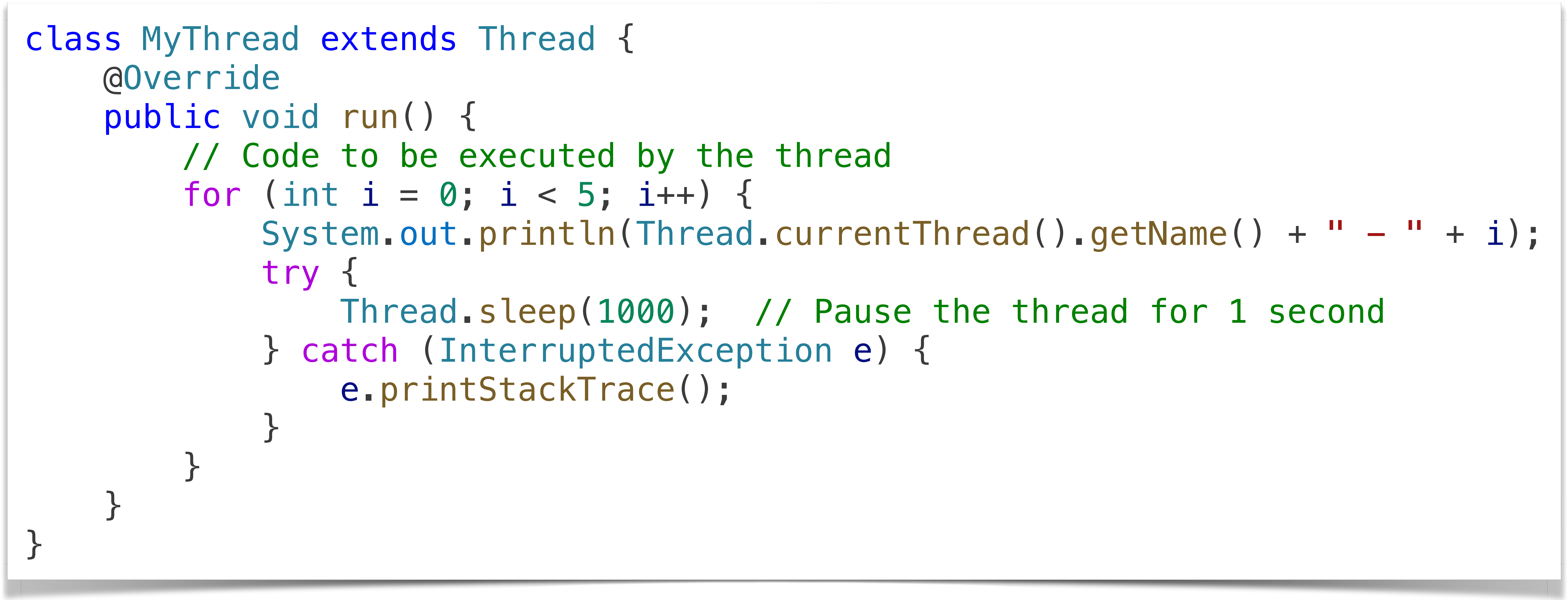
Threads

Multithreading in Java allows you to run multiple threads (lightweight subprocesses) concurrently, enabling better performance and resource utilization

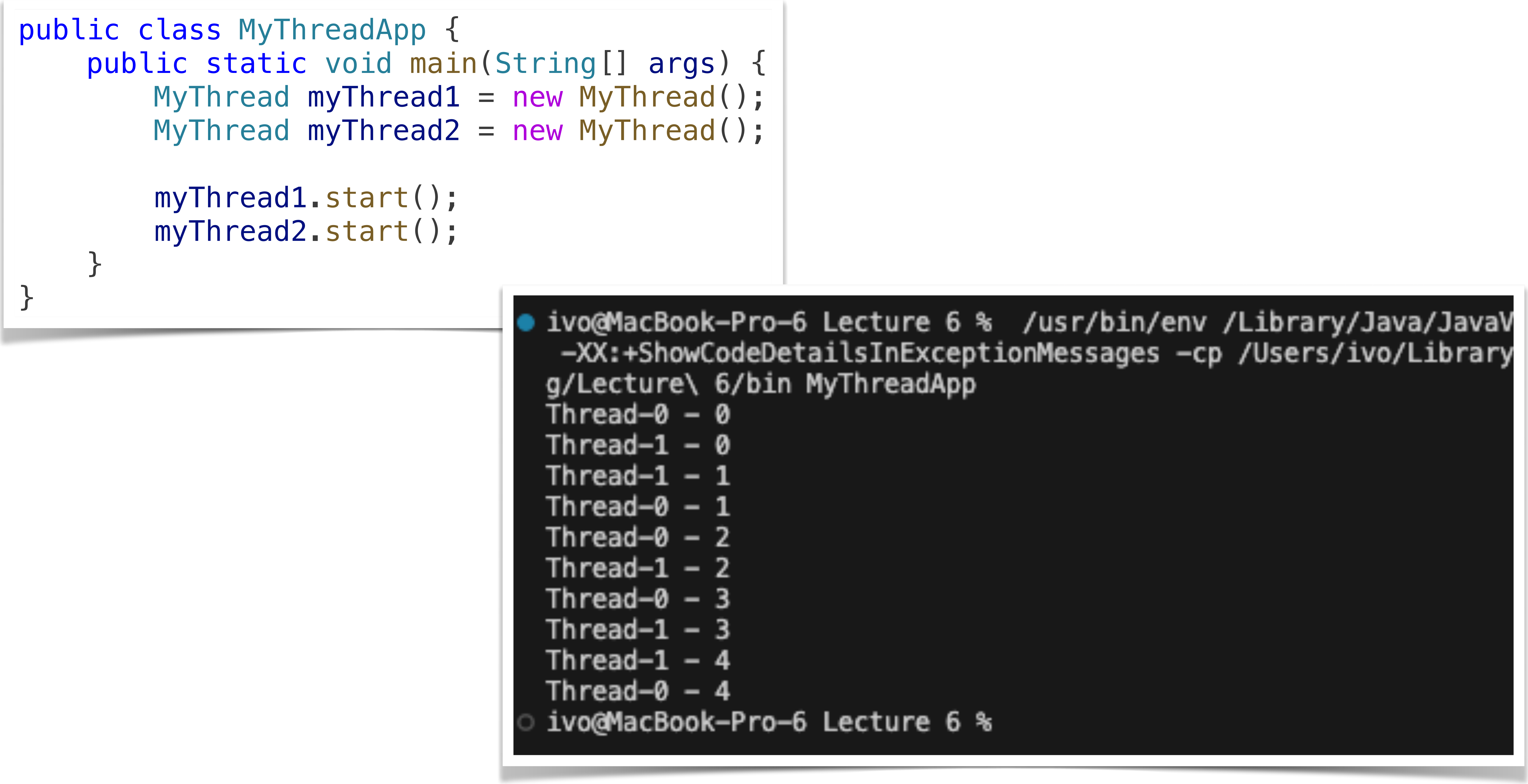
- A thread in Java is an independent path of execution
- Thread class is used to create and manage threads.
- There are two main ways to create a thread in Java:
  - By Extending the Thread Class
  - By Implementing the Runnable Interface

Extending the Thread Class

When you extend the Thread class, you need to override its run() method, which contains the code that the thread will execute.



```
class MyThread extends Thread {
    @Override
    public void run() {
        // Code to be executed by the thread
        for (int i = 0; i < 5; i++) {
            System.out.println(Thread.currentThread().getName() + " - " + i);
            try {
                Thread.sleep(1000);  // Pause the thread for 1 second
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

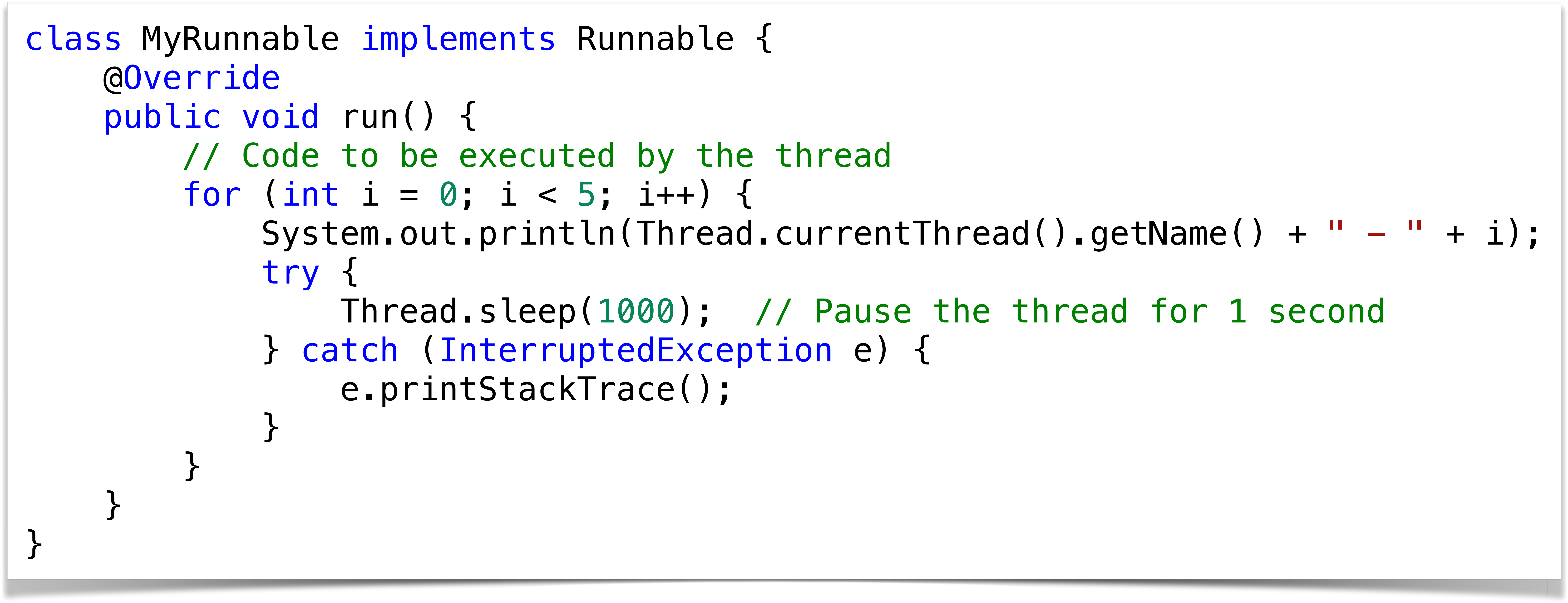


```java
public class MyThreadApp {
    public static void main(String[] args) {
        MyThread myThread1 = new MyThread();
        MyThread myThread2 = new MyThread();

        myThread1.start();
        myThread2.start();
    }
}
```

```
● ivo@MacBook-Pro-6 Lecture 6 %  /usr/bin/env /Library/Java/JavaV
  -XX:+ShowCodeDetailsInExceptionMessages -cp /Users/ivo/Library
 g/Lecture\ 6/bin MyThreadApp
 Thread-0 - 0
 Thread-1 - 0
 Thread-1 - 1
 Thread-0 - 1
 Thread-0 - 2
 Thread-1 - 2
 Thread-0 - 3
 Thread-1 - 3
 Thread-1 - 4
 Thread-0 - 4
○ ivo@MacBook-Pro-6 Lecture 6 %
```

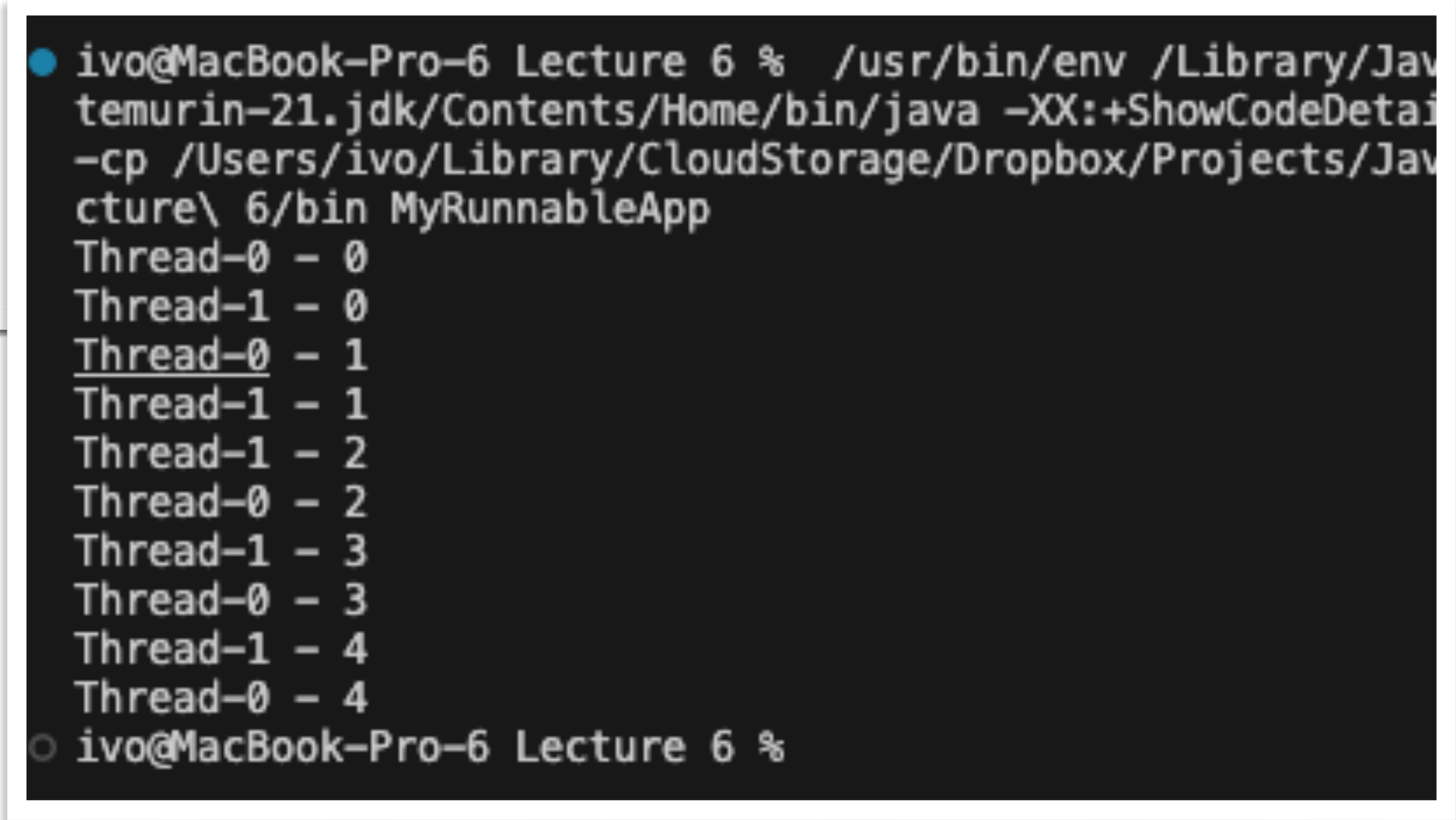
Implementing the Runnable Interface

Another approach is to implement the Runnable interface. This is considered more flexible because it allows your class to extend another class while still supporting multithreading.



```java
class MyRunnable implements Runnable {
    @Override
    public void run() {
        // Code to be executed by the thread
        for (int i = 0; i < 5; i++) {
            System.out.println(Thread.currentThread().getName() + " - " + i);
            try {
                Thread.sleep(1000);  // Pause the thread for 1 second
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```java
public class MyRunnableApp {
    public static void main(String[] args) {
        // Creating Runnable objects
        MyRunnable runnable1 = new MyRunnable();
        MyRunnable runnable2 = new MyRunnable();

        // Creating threads and passing the Runnable objects
        Thread thread1 = new Thread(runnable1);
        Thread thread2 = new Thread(runnable2);

        // Starting the threads
        thread1.start();
        thread2.start();
    }
}
```



```
● ivo@MacBook-Pro-6 Lecture 6 %  /usr/bin/env /Library/Jav
  temurin-21.jdk/Contents/Home/bin/java -XX:+ShowCodeDetai
  -cp /Users/ivo/Library/CloudStorage/Dropbox/Projects/Jav
  cture\ 6/bin MyRunnableApp
  Thread-0 - 0
  Thread-1 - 0
  Thread-0 - 1
  Thread-1 - 1
  Thread-1 - 2
  Thread-0 - 2
  Thread-1 - 3
  Thread-0 - 3
  Thread-1 - 4
  Thread-0 - 4
○ ivo@MacBook-Pro-6 Lecture 6 %
```

Some Important Notes

Key Methods in Thread Class

- start(): Begins the execution of the thread.
- run(): Contains the code to be executed when the thread is running.
- sleep(long millis): Puts the current thread to sleep for the specified milliseconds.
- join(): Allows one thread to wait for the completion of another.
- setPriority(): Sets the priority of a thread.
- getName(): Retrieves the name of the thread.

Synchronization

Since Java is a multithreaded system, care must be taken to prevent multiple threads from modifying objects simultaneously. Section of code that must not be executed simultaneously are known as “critical section”.

- Statement synchronized: *synchronized (expression) statement*
  - expression must resolve to an object or array
  - statement is the code of critical section.
  - The synchronized statement attempts to acquire an exclusive lock for the object or array and it does not execute the critical section code until it can obtain this lock.
- Method modifier synchronized indicates that entire method is critical section code. For a synchronized instance method, Java obtains an exclusive lock on the class instance. For a synchronized class method, Java obtains an exclusive lock on the class.

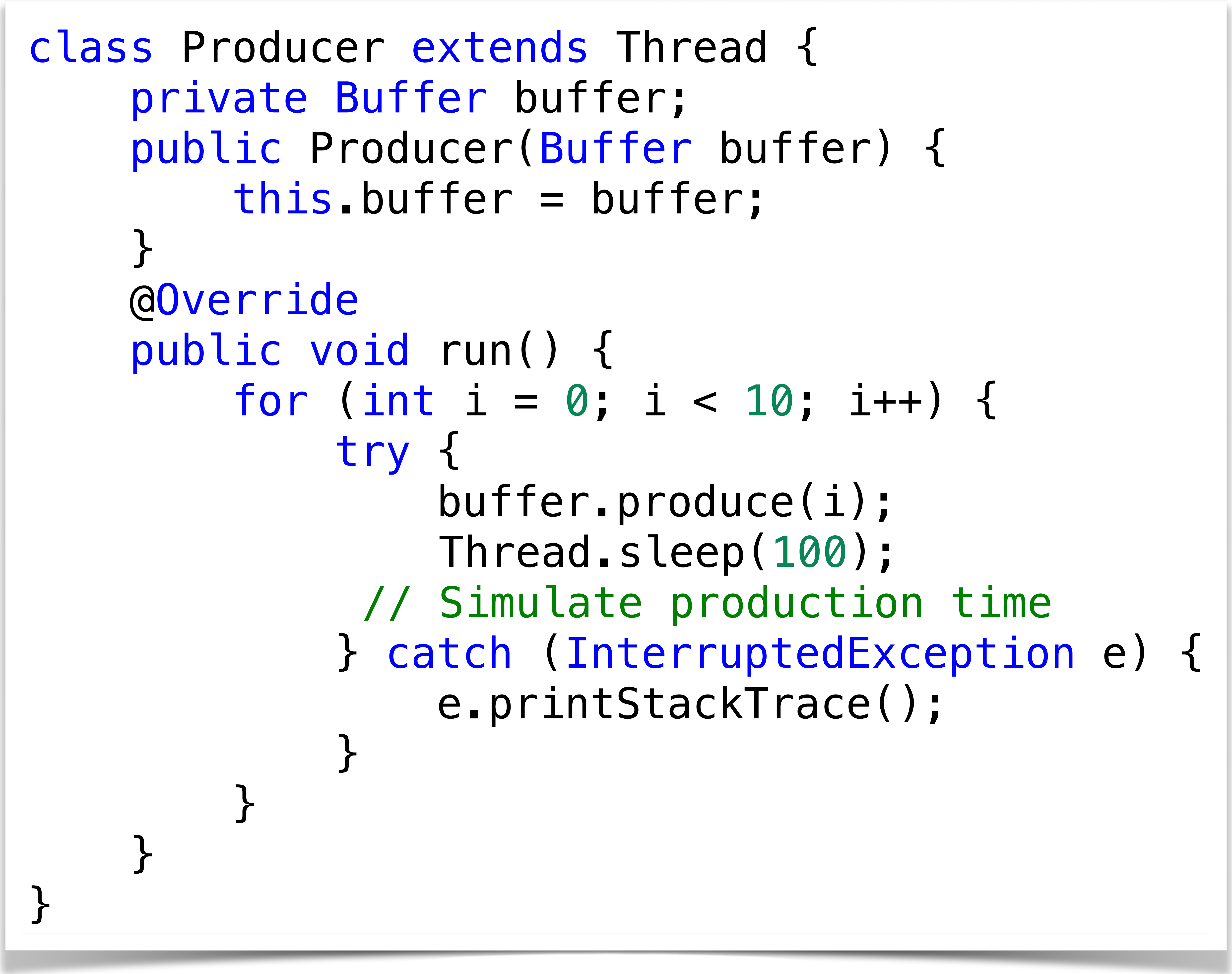
Multiple Threads Communication

In Java, threads often need to communicate and synchronize their actions to ensure correct program behavior. This is especially important when multiple threads operate on shared resources. Java provides several mechanisms for thread communication

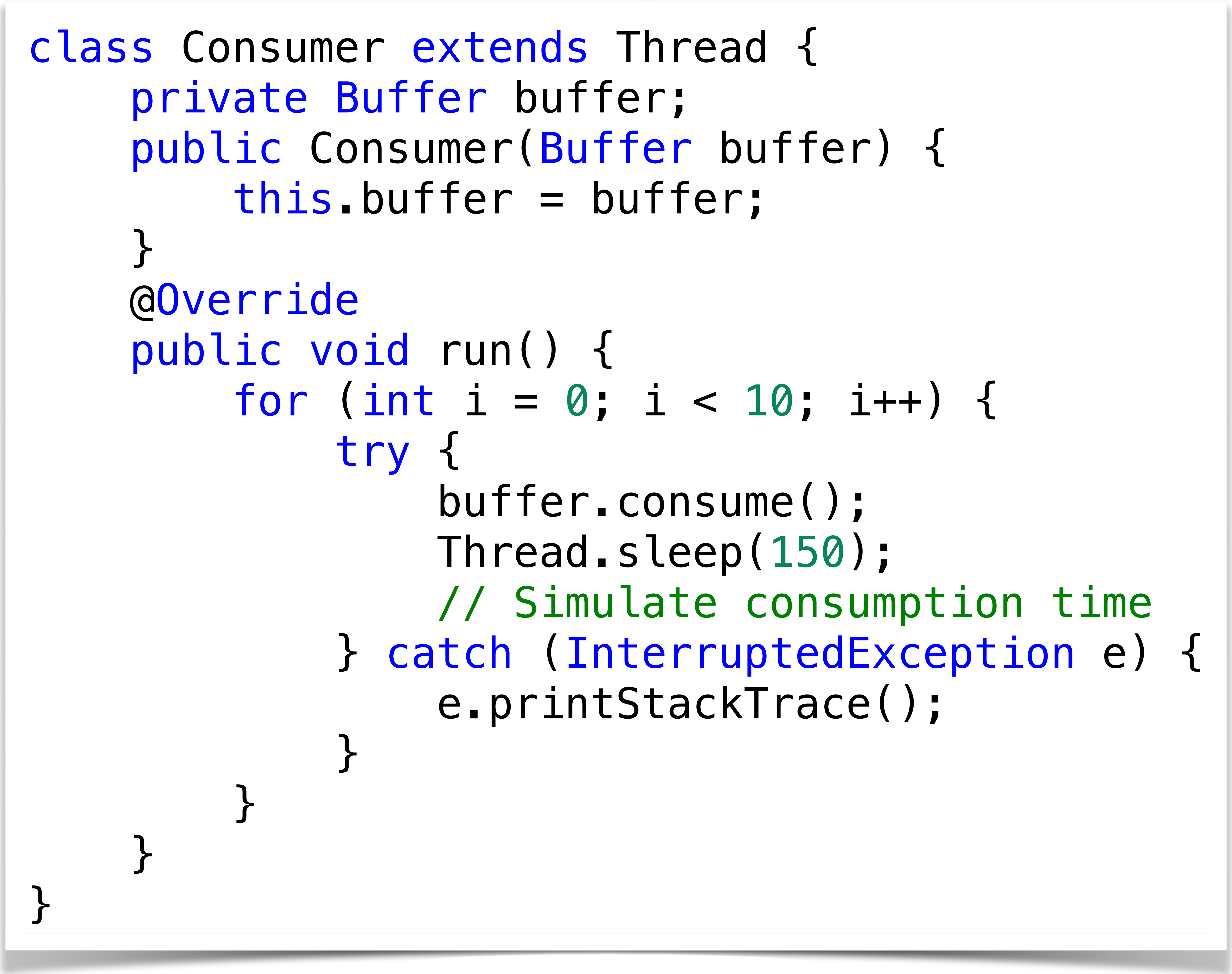
- wait(), notify(), and notifyAll(): These methods allow threads to communicate by pausing and resuming their execution based on certain conditions.
- Locks and Conditions (from java.util.concurrent): For more complex thread synchronization, you can use explicit locks and condition objects.
- join() method: This allows one thread to wait for another to finish execution.

Producer-Consumer Problem

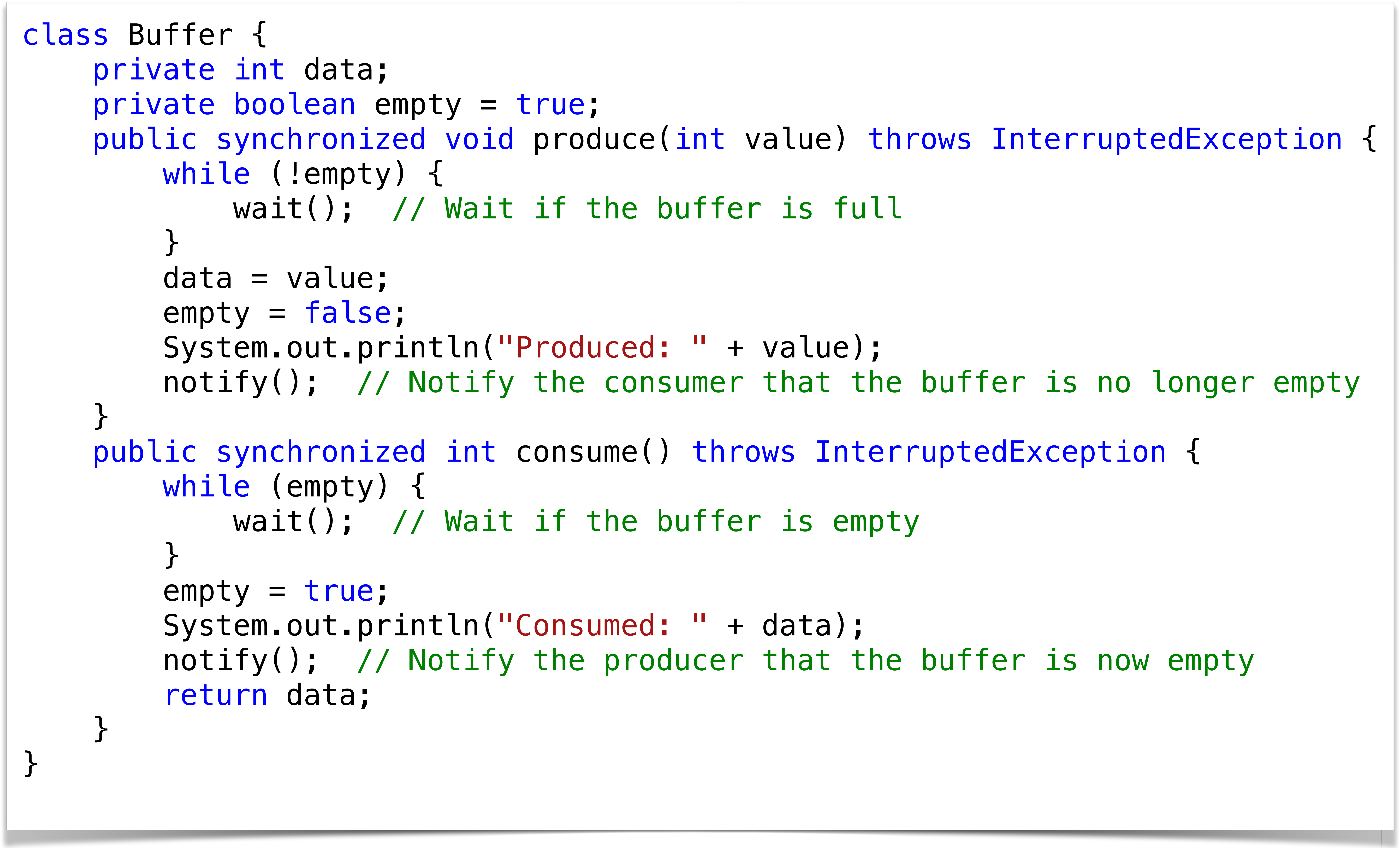
In the producer-consumer problem, one thread (the producer) produces data and another thread (the consumer) consumes it. The producer must wait if the “buffer” is full, and the consumer must wait if the “buffer” is empty



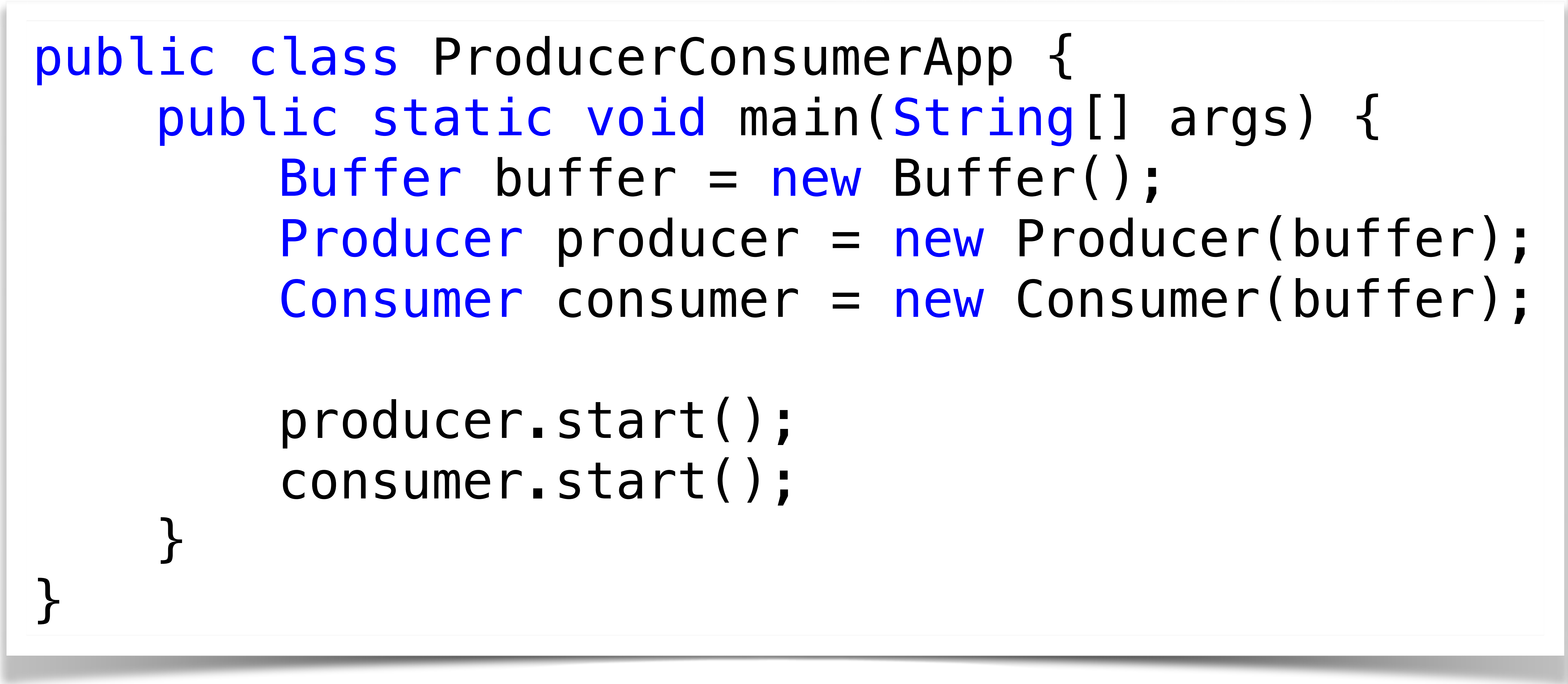
```java
class Producer extends Thread {
    private Buffer buffer;
    public Producer(Buffer buffer) {
        this.buffer = buffer;
    }
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            try {
                buffer.produce(i);
                Thread.sleep(100);
             // Simulate production time
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```



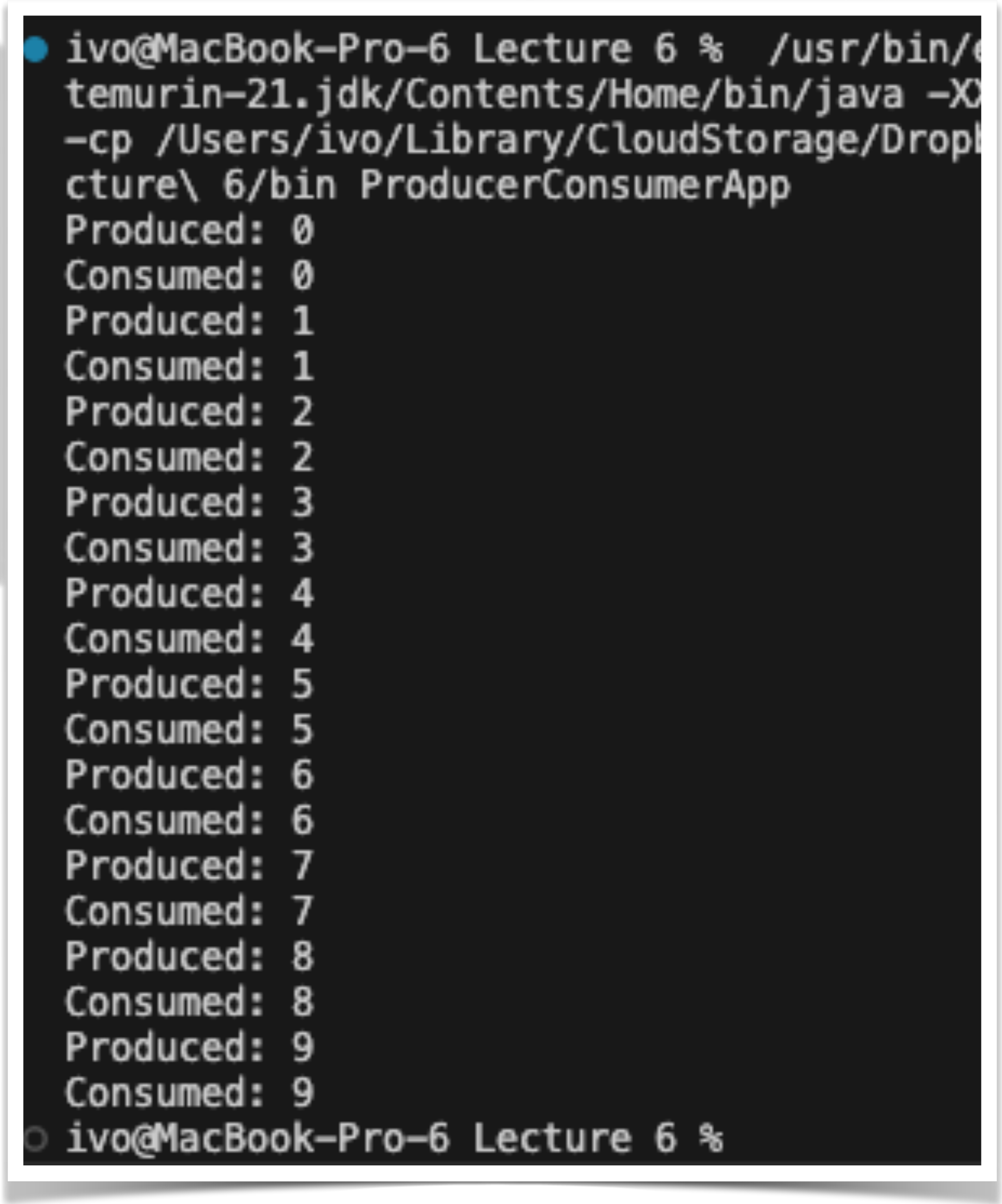
```java
class Consumer extends Thread {
    private Buffer buffer;
    public Consumer(Buffer buffer) {
        this.buffer = buffer;
    }
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            try {
                buffer.consume();
                Thread.sleep(150);
                // Simulate consumption time
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```



```java
class Buffer {
    private int data;
    private boolean empty = true;
    public synchronized void produce(int value) throws InterruptedException {
        while (!empty) {
            wait();  // Wait if the buffer is full
        }
        data = value;
        empty = false;
        System.out.println("Produced: " + value);
        notify();  // Notify the consumer that the buffer is no longer empty
    }
    public synchronized int consume() throws InterruptedException {
        while (empty) {
            wait();  // Wait if the buffer is empty
        }
        empty = true;
        System.out.println("Consumed: " + data);
        notify();  // Notify the producer that the buffer is now empty
        return data;
    }
}
```



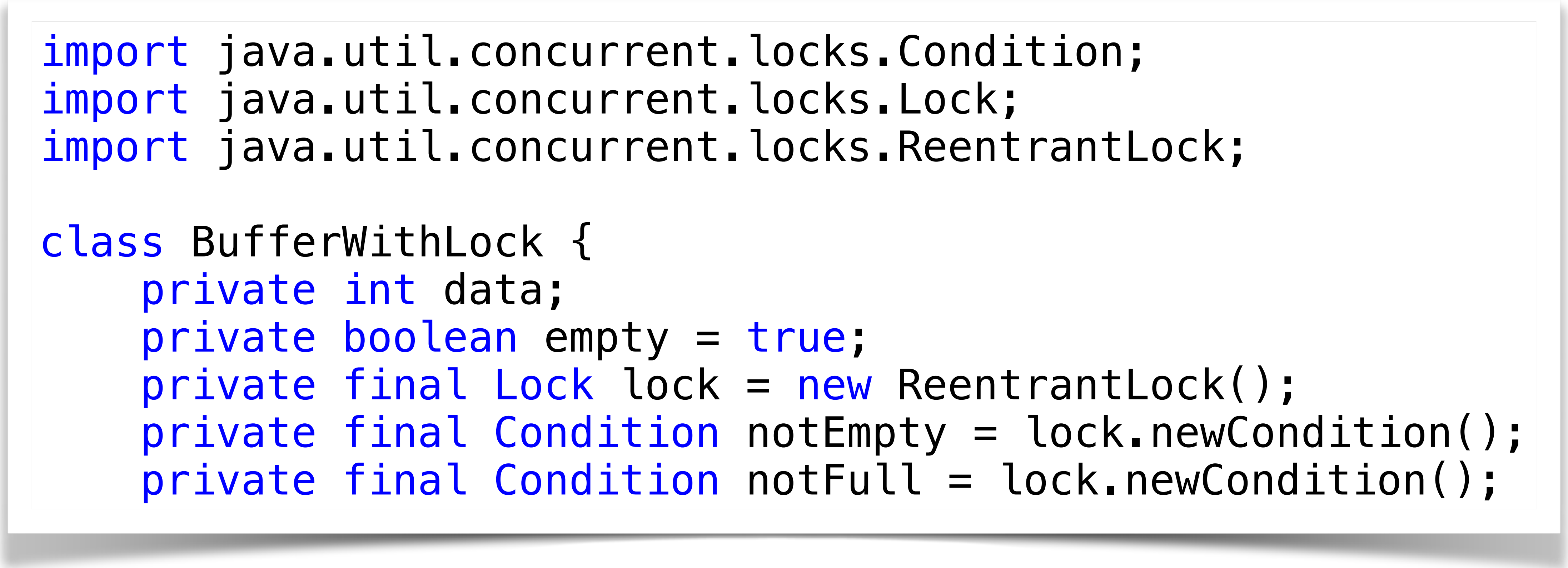
```
public class ProducerConsumerApp {
    public static void main(String[] args) {
        Buffer buffer = new Buffer();
        Producer producer = new Producer(buffer);
        Consumer consumer = new Consumer(buffer);

        producer.start();
        consumer.start();
    }
}
```



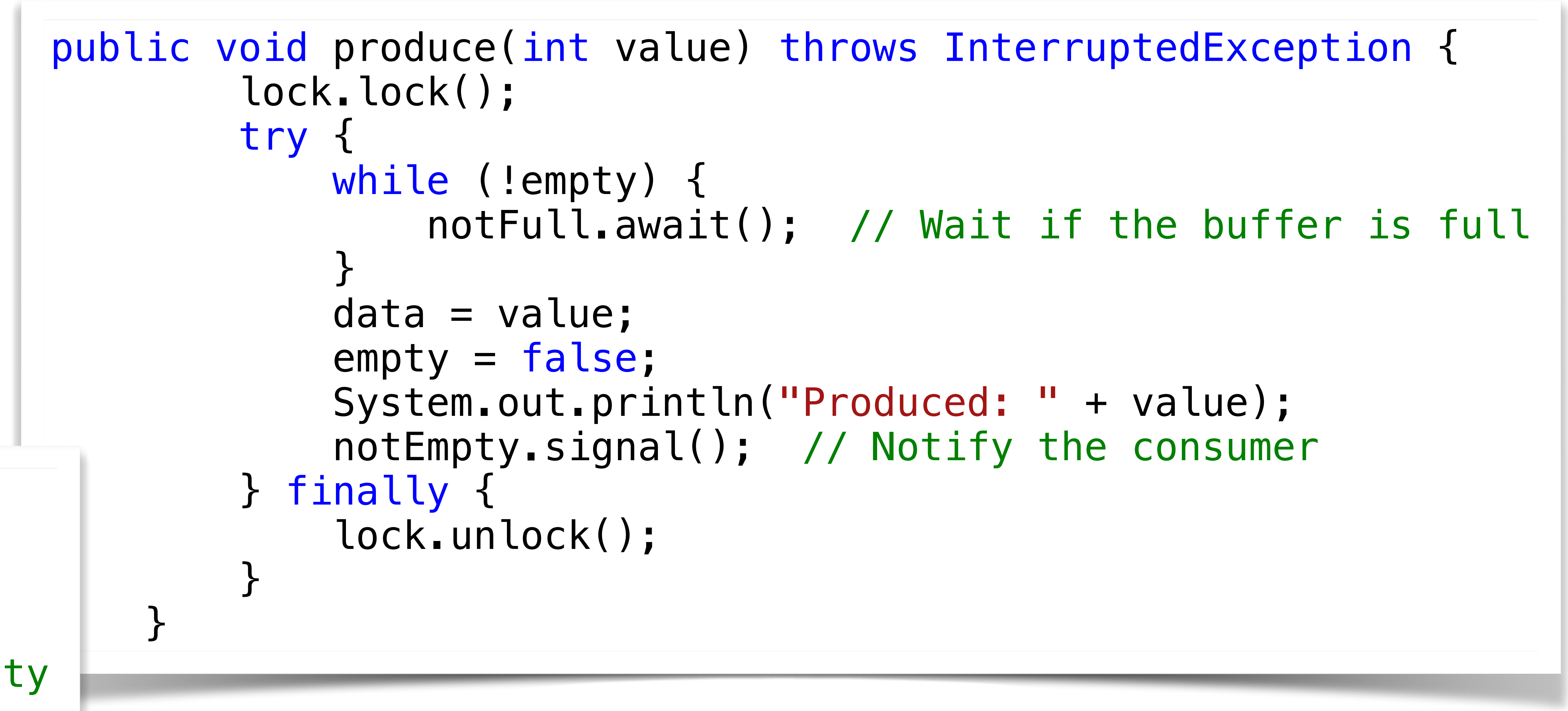
```
ivo@MacBook-Pro-6 Lecture 6 %  /usr/bin/
temurin-21.jdk/Contents/Home/bin/java -X
-cp /Users/ivo/Library/CloudStorage/Drop
cture\ 6/bin ProducerConsumerApp
Produced: 0
Consumed: 0
Produced: 1
Consumed: 1
Produced: 2
Consumed: 2
Produced: 3
Consumed: 3
Produced: 4
Consumed: 4
Produced: 5
Consumed: 5
Produced: 6
Consumed: 6
Produced: 7
Consumed: 7
Produced: 8
Consumed: 8
Produced: 9
Consumed: 9
ivo@MacBook-Pro-6 Lecture 6 %
```

Using Locks and Conditions

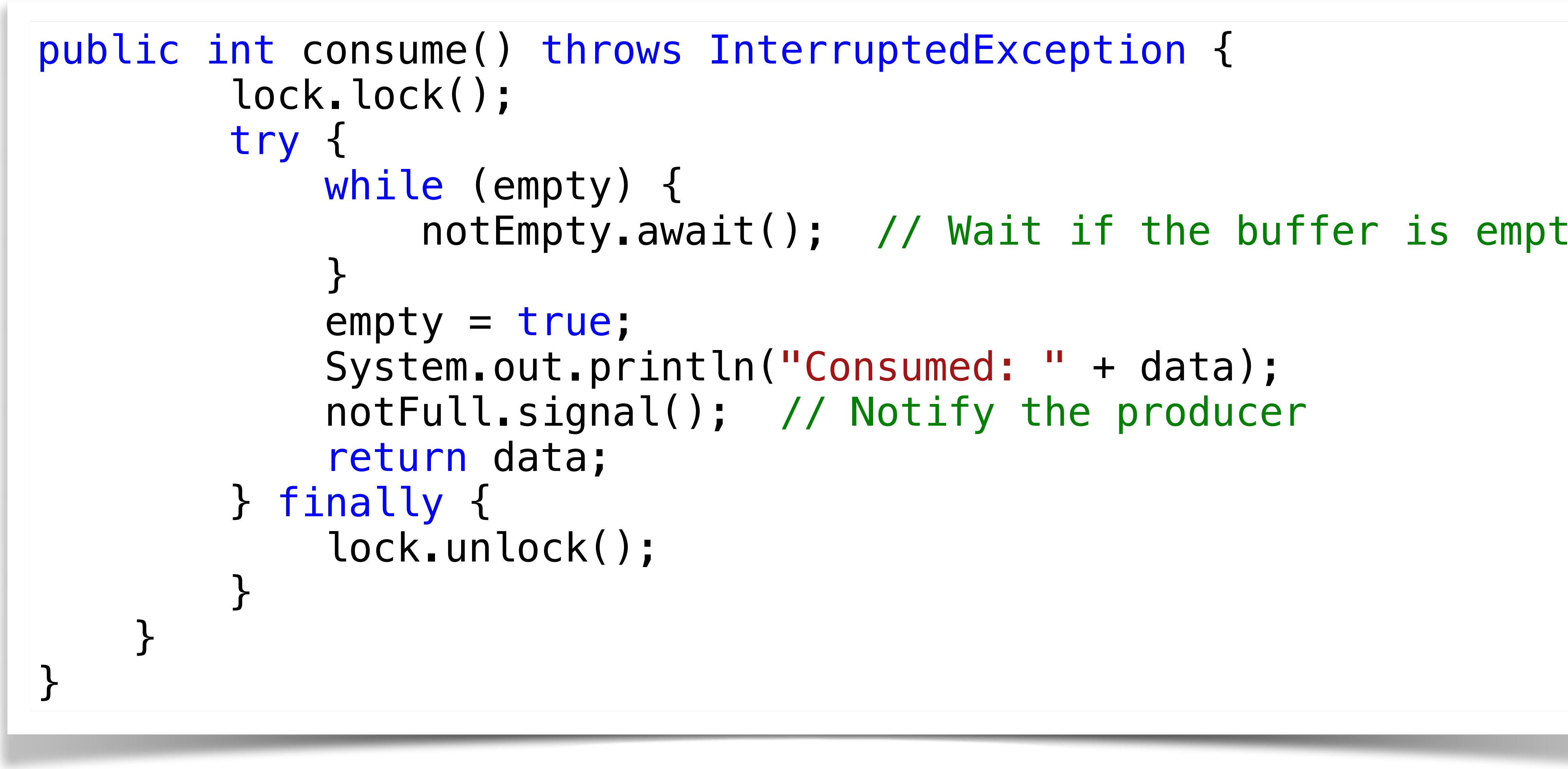
The Lock and Condition interfaces in provide more advanced control over thread communication and synchronization.



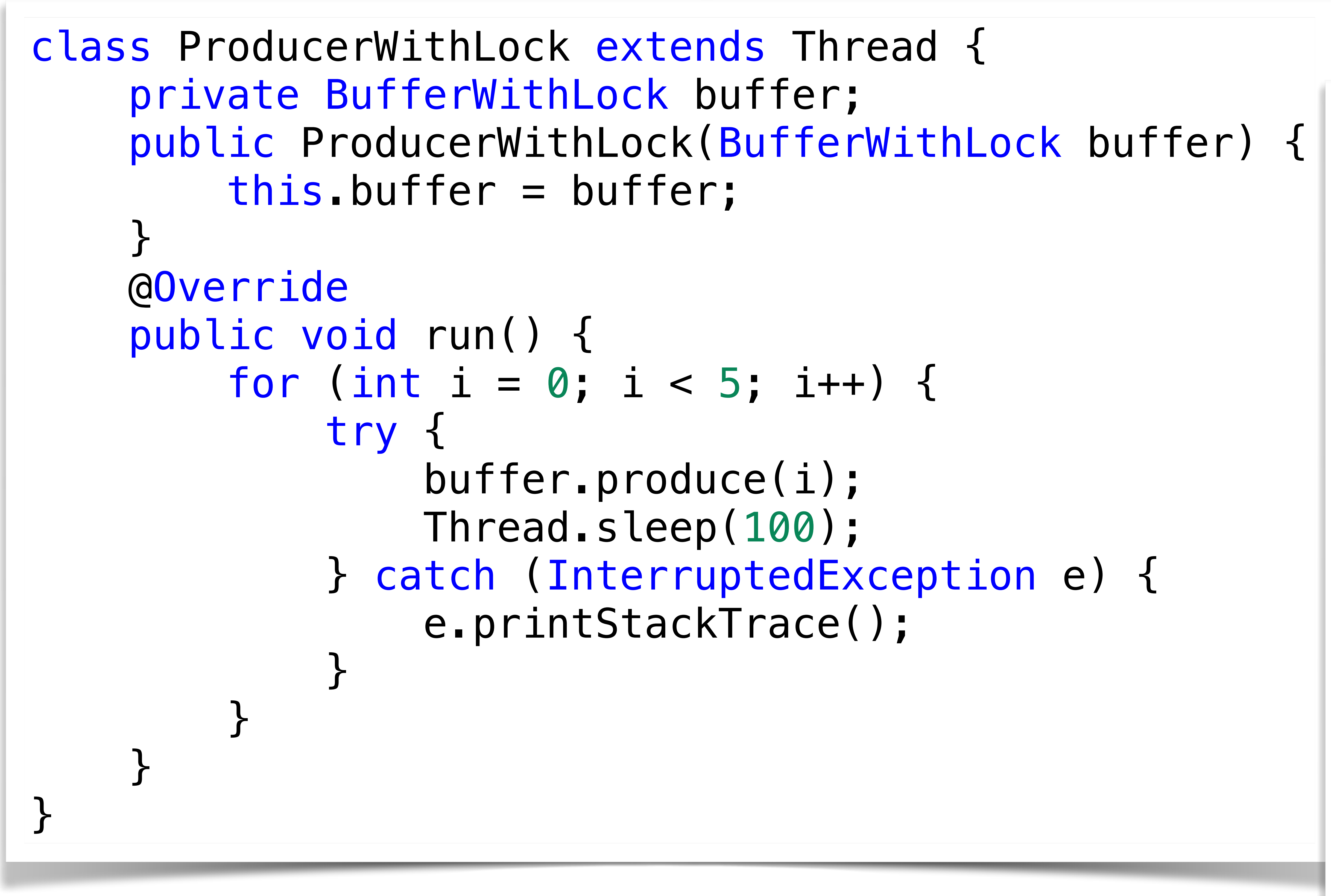
```java
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

class BufferWithLock {
    private int data;
    private boolean empty = true;
    private final Lock lock = new ReentrantLock();
    private final Condition notEmpty = lock.newCondition();
    private final Condition notFull = lock.newCondition();
```



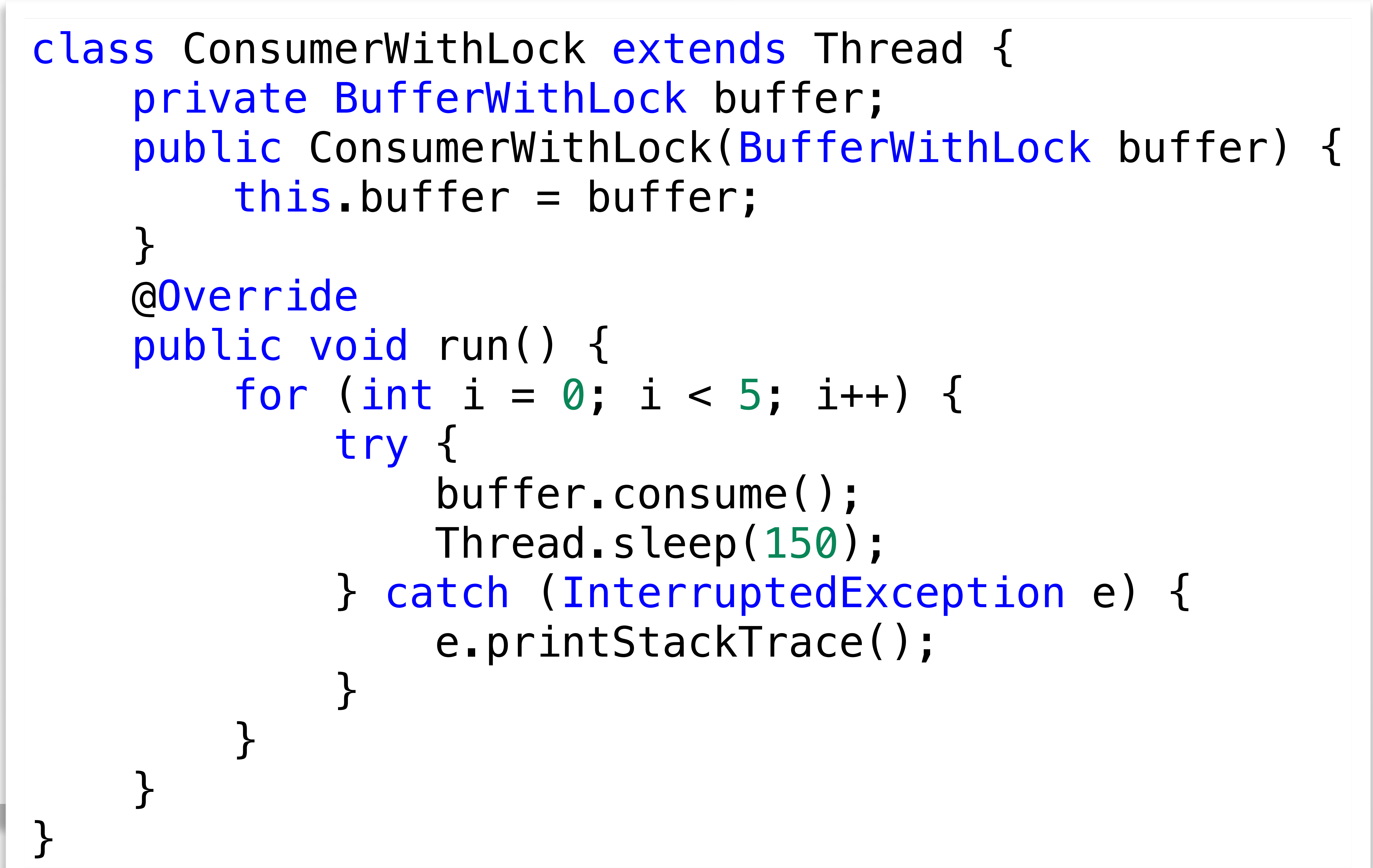
```java
public void produce(int value) throws InterruptedException {
        lock.lock();
        try {
            while (!empty) {
                notFull.await();  // Wait if the buffer is full
            }
            data = value;
            empty = false;
            System.out.println("Produced: " + value);
            notEmpty.signal();  // Notify the consumer
        } finally {
            lock.unlock();
        }
    }
```



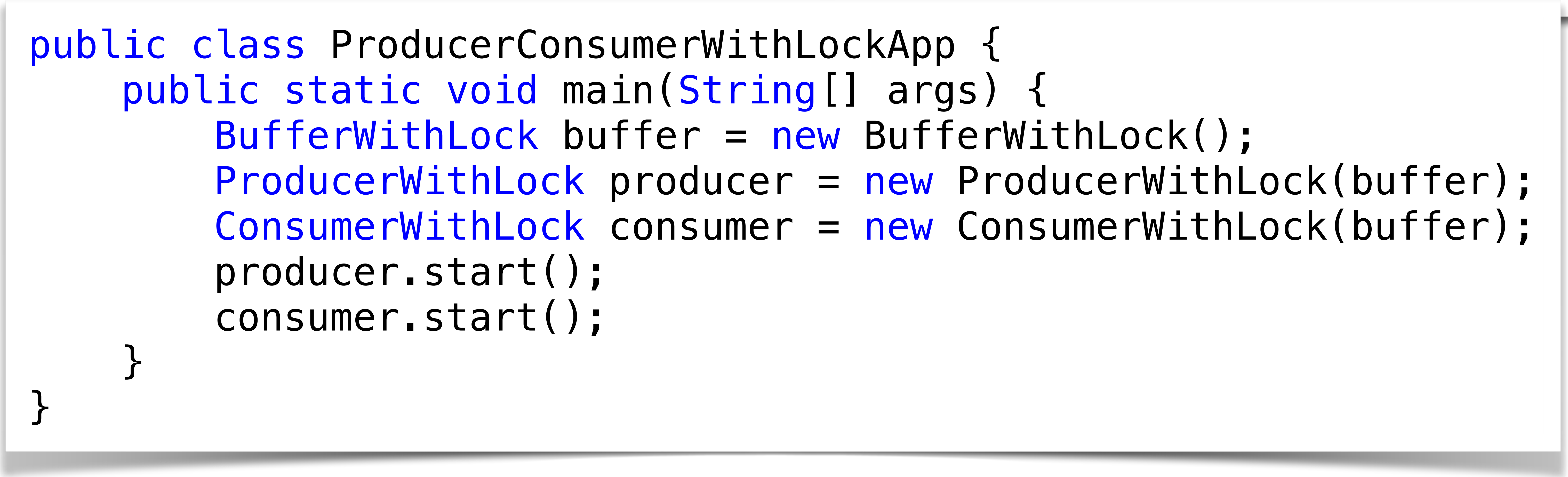
```java
public int consume() throws InterruptedException {
        lock.lock();
        try {
            while (empty) {
                notEmpty.await();  // Wait if the buffer is empty
            }
            empty = true;
            System.out.println("Consumed: " + data);
            notFull.signal();  // Notify the producer
            return data;
        } finally {
            lock.unlock();
        }
    }
}
```



```java
class ProducerWithLock extends Thread {
    private BufferWithLock buffer;
    public ProducerWithLock(BufferWithLock buffer) {
        this.buffer = buffer;
    }
    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            try {
                buffer.produce(i);
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```



```java
class ConsumerWithLock extends Thread {
    private BufferWithLock buffer;
    public ConsumerWithLock(BufferWithLock buffer) {
        this.buffer = buffer;
    }
    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            try {
                buffer.consume();
                Thread.sleep(150);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

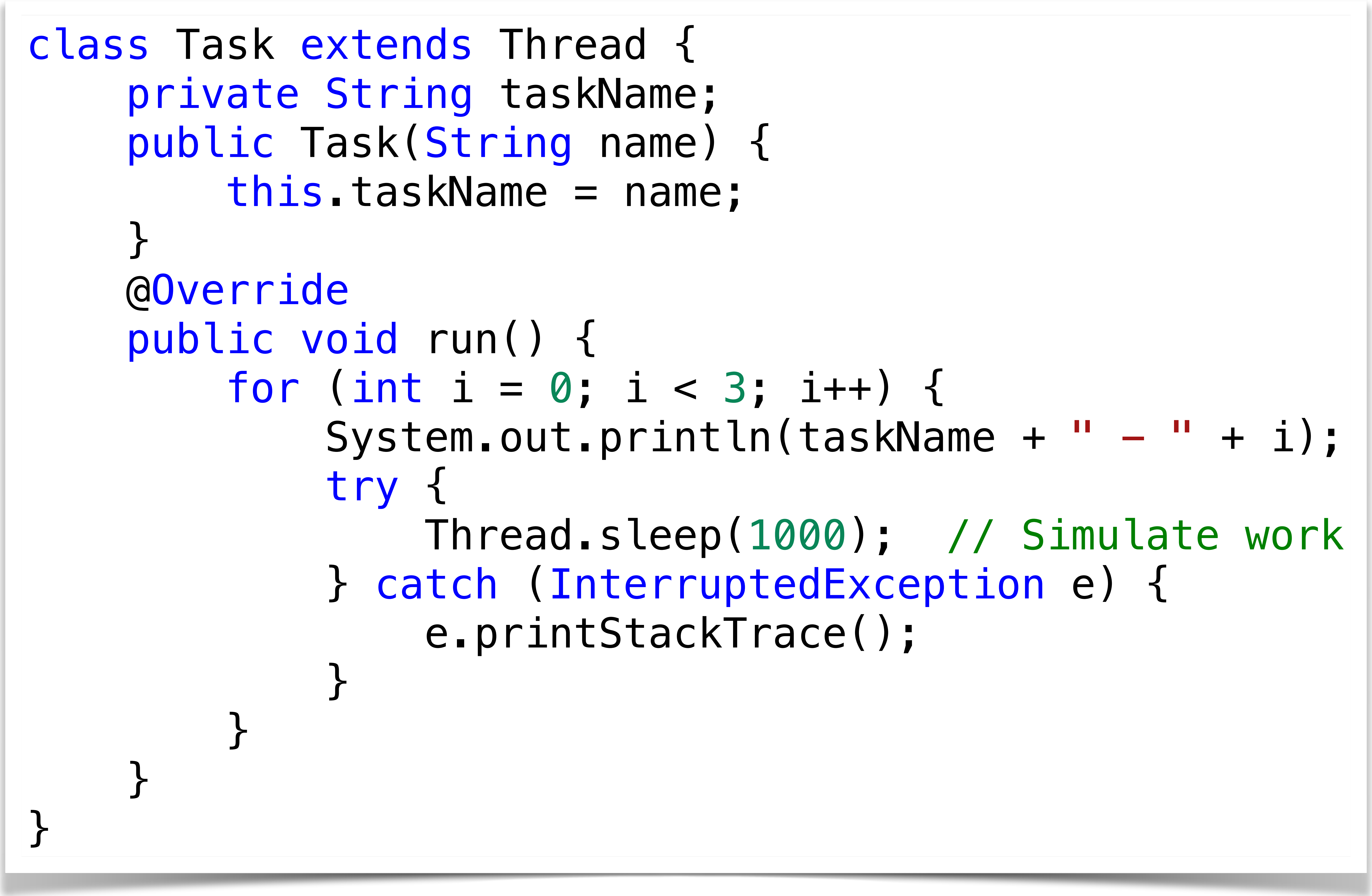


```java
public class ProducerConsumerWithLockApp {
    public static void main(String[] args) {
        BufferWithLock buffer = new BufferWithLock();
        ProducerWithLock producer = new ProducerWithLock(buffer);
        ConsumerWithLock consumer = new ConsumerWithLock(buffer);
        producer.start();
        consumer.start();
    }
}
```

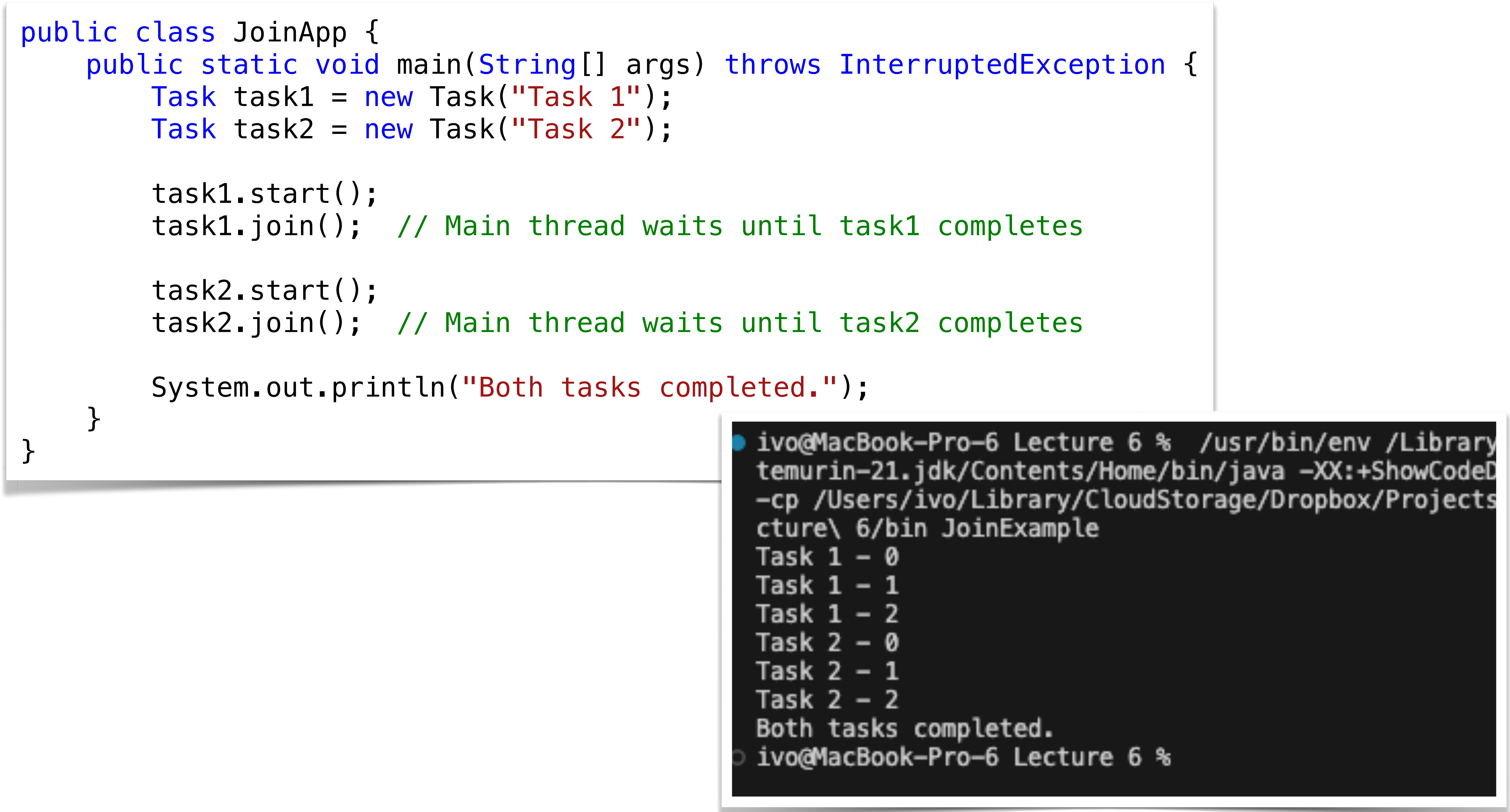
- A Lock is used instead of synchronized blocks.
- Conditions (notEmpty and notFull) are used to signal between the producer and consumer.

Using join() for Thread Communication

The join() method allows one thread to wait for the completion of another thread



```
class Task extends Thread {
    private String taskName;
    public Task(String name) {
        this.taskName = name;
    }
    @Override
    public void run() {
        for (int i = 0; i < 3; i++) {
            System.out.println(taskName + " - " + i);
            try {
                Thread.sleep(1000);  // Simulate work
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```



```java
public class JoinApp {
    public static void main(String[] args) throws InterruptedException {
        Task task1 = new Task("Task 1");
        Task task2 = new Task("Task 2");

        task1.start();
        task1.join();  // Main thread waits until task1 completes

        task2.start();
        task2.join();  // Main thread waits until task2 completes

        System.out.println("Both tasks completed.");
    }
}
```

```
ivo@MacBook-Pro-6 Lecture 6 %  /usr/bin/env /Library
temurin-21.jdk/Contents/Home/bin/java -XX:+ShowCodeD
-cp /Users/ivo/Library/CloudStorage/Dropbox/Projects
cture\ 6/bin JoinExample
Task 1 - 0
Task 1 - 1
Task 1 - 2
Task 2 - 0
Task 2 - 1
Task 2 - 2
Both tasks completed.
ivo@MacBook-Pro-6 Lecture 6 %
```

7th Part: GUI Framework

- Swing
- JavaFX
- Event handling in GUI

Graphical User Interface

In Java, Graphical User Interface (GUI) development is mainly handled using two primary frameworks:

- **Swing**:
  - A lightweight GUI toolkit that is part of the Java Standard Library.
- **JavaFX**:
  - A more modern and feature-rich toolkit that is designed to replace Swing. JavaFX provides advanced capabilities for building rich desktop applications.

Swing

Swing is a part of the Java Foundation Classes (JFC) and is widely used for creating desktop applications. It builds upon the Abstract Window Toolkit (AWT) but provides more powerful and flexible components.

- **Key Components:**
  - JFrame: A window with a title bar and borders.
  - JPanel: A container for organizing components.
  - JButton: A button component for user interactions.
  - JLabel: A component for displaying text.
  - JTextField: A text input field.

```java
import javax.swing.*;

public class SimpleSwingApp {
    public static void main(String[] args) {
        // Create a new JFrame (the window)
        JFrame frame = new JFrame("Simple Swing App");
        frame.setSize(400, 300);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        // Create a button
        JButton button = new JButton("Click Me!");
        // Add a button click event listener
        button.addActionListener(e -> JOptionPane.showMessageDialog(frame, "Button Clicked!"));
        // Add button to the frame
        frame.getContentPane().add(button);
        // Make the window visible
        frame.setVisible(true);
    }
}
```

Button Clicked!

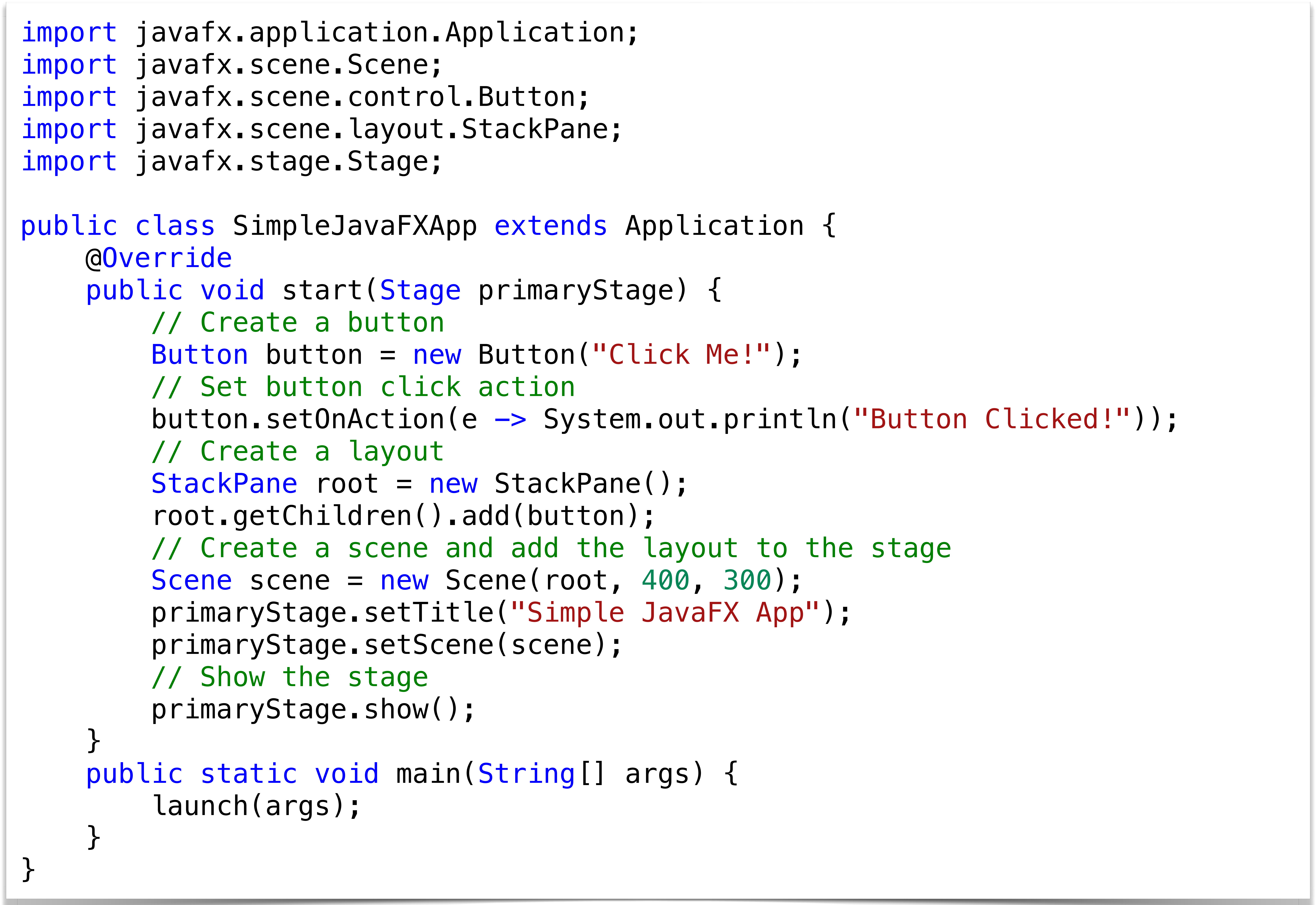
- **Explanation:**
  - JFrame: Represents the main window of the application.
  - JButton: A button component is added to the frame, and an event listener is used to respond to button clicks.
  - ActionListener: Handles the button's click event.

- **Layout Management:**
  - Swing provides several layout managers such as FlowLayout, BorderLayout, and GridLayout to arrange components within containers.

JavaFX

JavaFX is a newer framework introduced to replace Swing and provides a more modern approach to building rich UIs, with support for advanced features such as media integration, CSS styling, and hardware-accelerated graphics.

- **Key Features of JavaFX**:
  - Declarative UI using FXML.
  - Built-in support for animations and graphics.
  - Modern UI components.
  - CSS for styling the UI.
  - Support for media playback and 3D graphics.



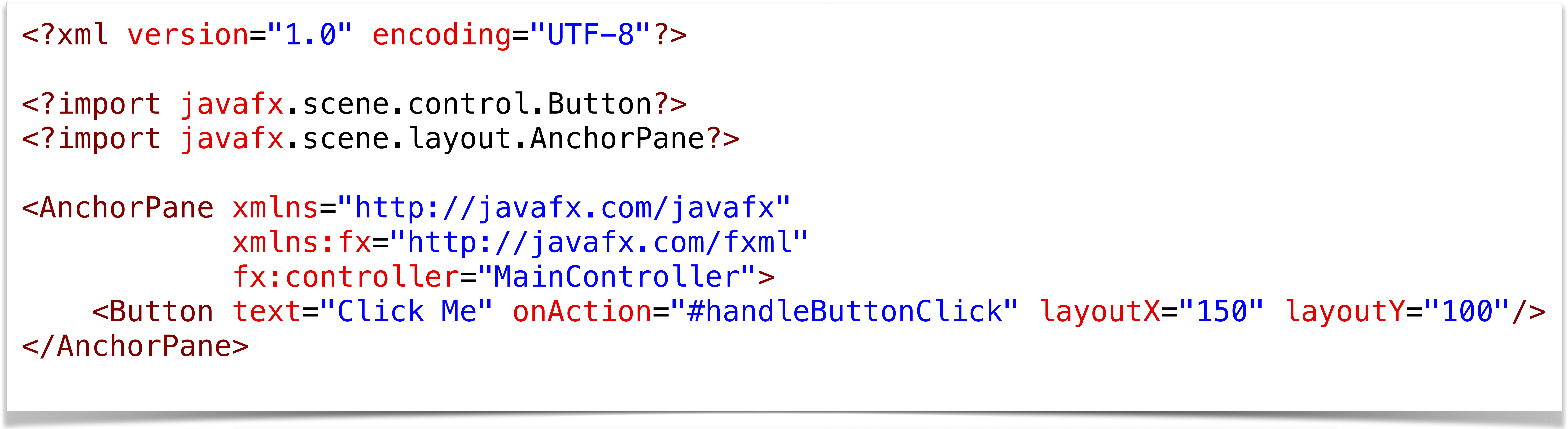
```java
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class SimpleJavaFXApp extends Application {
    @Override
    public void start(Stage primaryStage) {
        // Create a button
        Button button = new Button("Click Me!");
        // Set button click action
        button.setOnAction(e -> System.out.println("Button Clicked!"));
        // Create a layout
        StackPane root = new StackPane();
        root.getChildren().add(button);
        // Create a scene and add the layout to the stage
        Scene scene = new Scene(root, 400, 300);
        primaryStage.setTitle("Simple JavaFX App");
        primaryStage.setScene(scene);
        // Show the stage
        primaryStage.show();
    }
    public static void main(String[] args) {
        launch(args);
    }
}
```

- **Explanation**:
  - Application: The main class that extends Application to build JavaFX applications.
  - Stage: Represents the main window (similar to JFrame in Swing)
  - Scene: Holds all UI elements in a scene graph.
  - Button: A button component with an event handler for clicks.

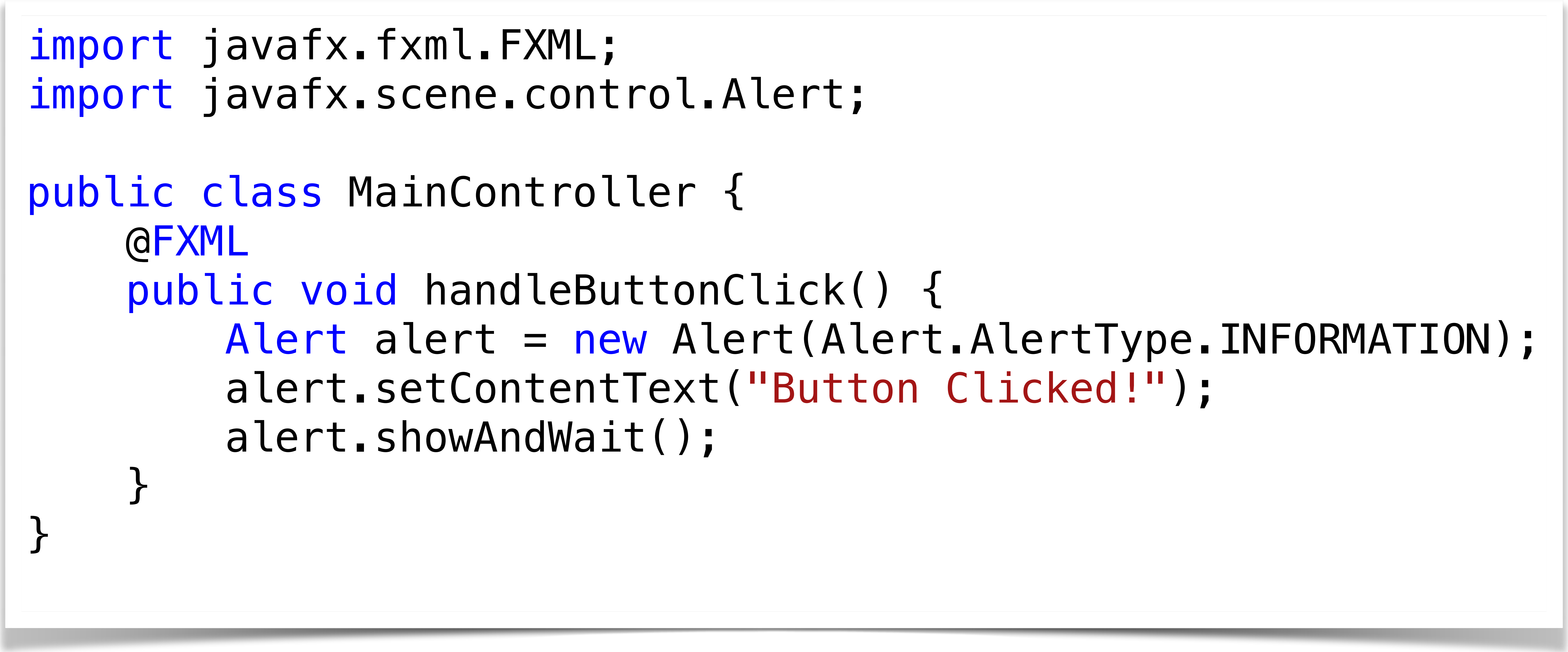
Using FXML in JavaFX

FXML is an XML-based language used to describe JavaFX GUIs. It allows for a clean separation between the UI layout and the logic.

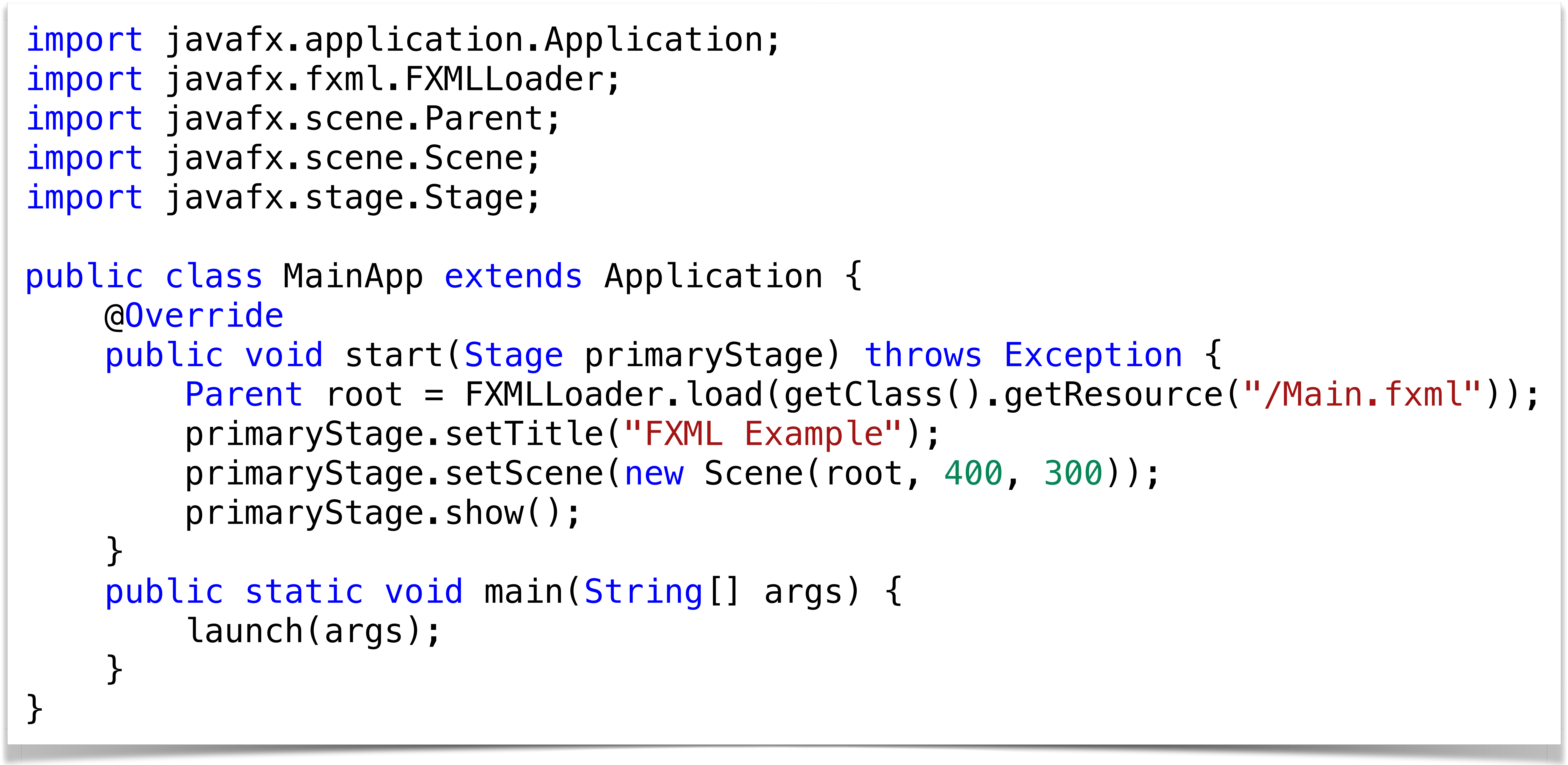
- FXML file: Contains the structure and layout of the UI.
- Controller: The Java class that handles user interactions and events.



```xml
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.Button?>
<?import javafx.scene.layout.AnchorPane?>

<AnchorPane xmlns="http://javafx.com/javafx"
            xmlns:fx="http://javafx.com/fxml"
            fx:controller="MainController">
    <Button text="Click Me" onAction="#handleButtonClick" layoutX="150" layoutY="100"/>
</AnchorPane>
```



```java
import javafx.fxml.FXML;
import javafx.scene.control.Alert;

public class MainController {
    @FXML
    public void handleButtonClick() {
        Alert alert = new Alert(Alert.AlertType.INFORMATION);
        alert.setContentText("Button Clicked!");
        alert.showAndWait();
    }
}
```



```java
import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class MainApp extends Application {
    @Override
    public void start(Stage primaryStage) throws Exception {
        Parent root = FXMLLoader.load(getClass().getResource("/Main.fxml"));
        primaryStage.setTitle("FXML Example");
        primaryStage.setScene(new Scene(root, 400, 300));
        primaryStage.show();
    }
    public static void main(String[] args) {
        launch(args);
    }
}
```

Comparison of Swing and JavaFX

Feature	Swing	JavaFX
Introduced	1997	2008
Declarative UI	No	Yes, via FXML
Styling	Limited (basic Look and Feel)	CSS-based styling
Graphics	Basic 2D graphics	Rich 2D and 3D graphics
Animation	Difficult to implement manually	Built-in support for animations
Media	No native support	Built-in media playback
Performance	Slower (especially with heavy UIs)	Faster (hardware-accelerated)
Future Support	Largely deprecated	Actively maintained and updated

Event Handling in GUI

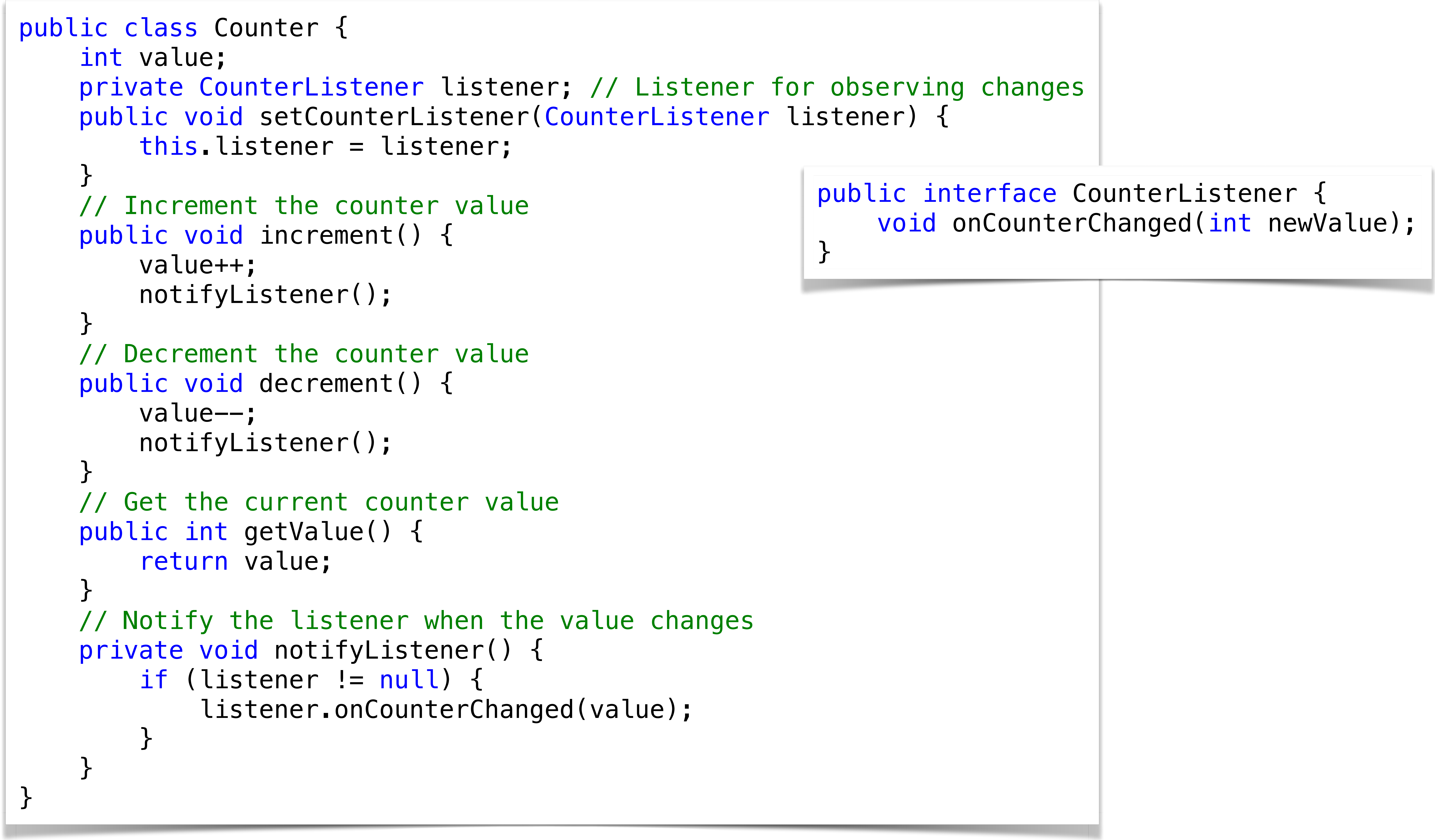
Both Swing and JavaFX have robust event-handling mechanisms

- In Swing, event listeners such as ActionListener, MouseListener, etc., are attached to UI components to handle user actions.
- In JavaFX, event handlers (such as setOnAction(), setOnKeyPressed()) or methods in the controller class are used to define how the application reacts to user interactions.

Counter Controlled by GUI in Swing

Model class (Counter) holds the logic for the counter, and the GUI (CounterApp) observes and updates its display based on the current state of the Counter.

- The Counter class will have:
  - An integer field to store the count value.
  - Methods to increment, decrement, and get the current value.
  - A listener mechanism to notify the GUI when the counter value changes.
- Modify the GUI (CounterApp):
  - The CounterApp will observe the Counter by implementing the CounterListener interface and updating the GUI when the counter value changes.



```java
public class Counter {
    int value;
    private CounterListener listener; // Listener for observing changes
    public void setCounterListener(CounterListener listener) {
        this.listener = listener;
    }
    // Increment the counter value
    public void increment() {
        value++;
        notifyListener();
    }
    // Decrement the counter value
    public void decrement() {
        value--;
        notifyListener();
    }
    // Get the current counter value
    public int getValue() {
        return value;
    }
    // Notify the listener when the value changes
    private void notifyListener() {
        if (listener != null) {
            listener.onCounterChanged(value);
        }
    }
}
```

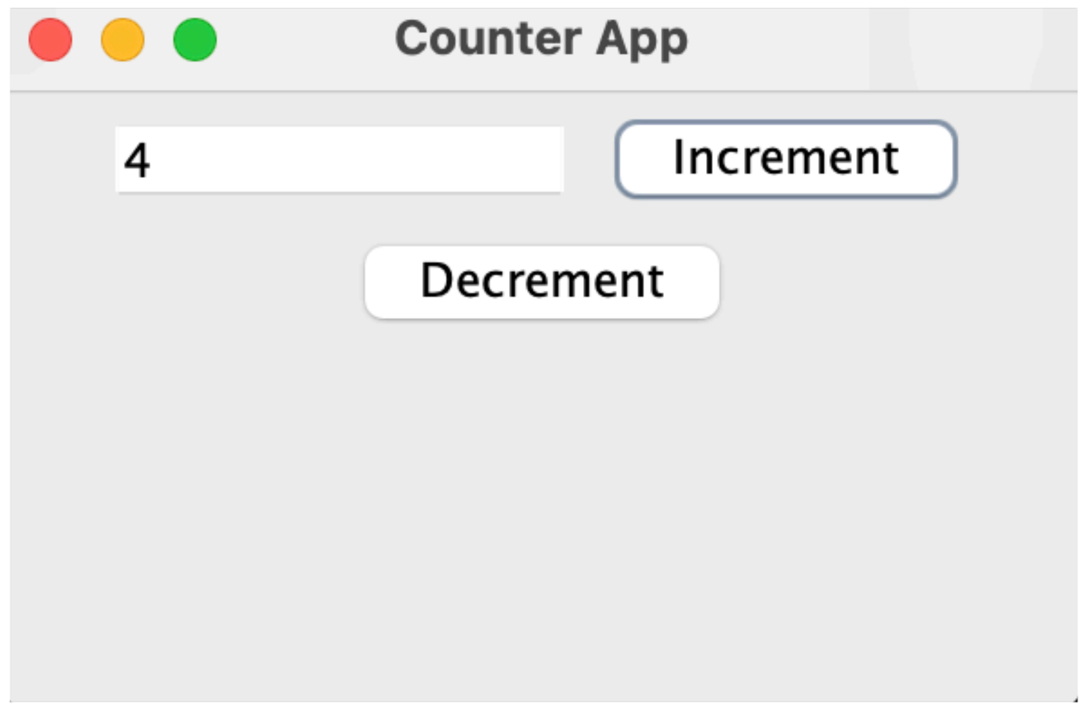
```java
public interface CounterListener {
    void onCounterChanged(int newValue);
}
```

```java
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
```

```java
public class CounterApp implements CounterListener {
            JFrame frame;
            JTextField stateField;
            JButton incrementButton;
            JButton decrementButton;
            Counter counter;
    public CounterApp() {
        counter = new Counter();
        counter.setCounterListener(this);
        frame = new JFrame("Counter App");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(300, 200);
        frame.setLayout(new FlowLayout());
        stateField = new JTextField(10);
        stateField.setEditable(false);
        stateField.setText(String.valueOf(counter.getValue()));
        incrementButton = new JButton("Increment");
        incrementButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                counter.increment(); // Call the increment method on the Counter
            }
        });
        decrementButton = new JButton("Decrement");
        decrementButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                counter.decrement(); // Call the decrement method on the Counter
            }
        });
        frame.add(stateField);
        frame.add(incrementButton);
        frame.add(decrementButton);
        frame.setVisible(true);
    }
```

```java
    // Implementation of the CounterListener interface
     @Override
     public void onCounterChanged(int newValue) {
         stateField.setText(String.valueOf(counter.getValue()));
         // Update the GUI when the counter changes
     }
```

```java
    public static void main(String[] args) {
        // Run the app on the Event Dispatch Thread (EDT)
        SwingUtilities.invokeLater(() -> new CounterApp());
    }
}
```



Summary

Swing or JavaFX

- Swing is older but still widely used for creating desktop applications. It’s simple but less modern compared to JavaFX.
- JavaFX is more powerful and versatile, providing support for modern UI features like CSS styling, animations, and media integration. It is the recommended framework for new Java GUI development.
- You can use JavaFX with FXML for cleaner, declarative UI code, separating logic from presentation.

- **How It Works:**
  - The Counter class contains all the logic to manage the counter state (increment, decrement, and notify listeners of changes).
  - The CounterApp class handles the user interface. It listens for changes in the counter's value by implementing the CounterListener interface.
  - When a button is clicked (increment or decrement), the Counter is updated, and the CounterListener notifies the GUI to update the display.
- **Benefits of This Design:**
  - Separation of Concerns: The business logic (counter) is separated from the presentation logic (GUI), making the application easier to maintain and test.
  - Observer Pattern: The Counter notifies the GUI when the value changes, adhering to the observer design pattern.

8th Part: Java API for DBMS

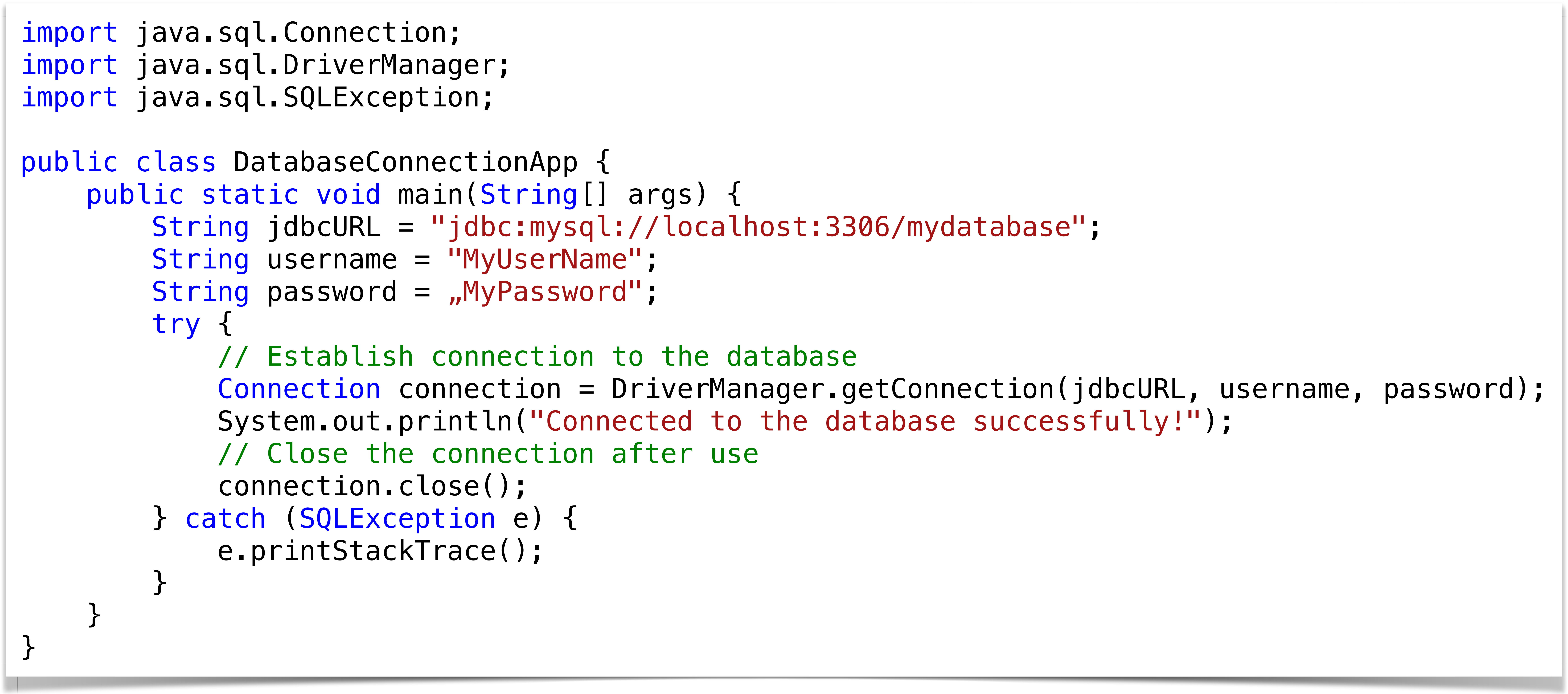
- Java database management system
- Performing SQL Queries

Database Management System (DBMS)

To use a DBMS in Java, you typically rely on JDBC (Java Database Connectivity) API

- Each DBMS has its own JDBC driver that enables communication between Java and the database.
  - MySQL: mysql-connector-java
  - PostgreSQL: postgresql
  - SQLite: sqlite-jdbc
- You can add these drivers to your project manually:
  - Download the JDBC .jar file and add it to your project's classpath.

Connecting to a MySQL Database

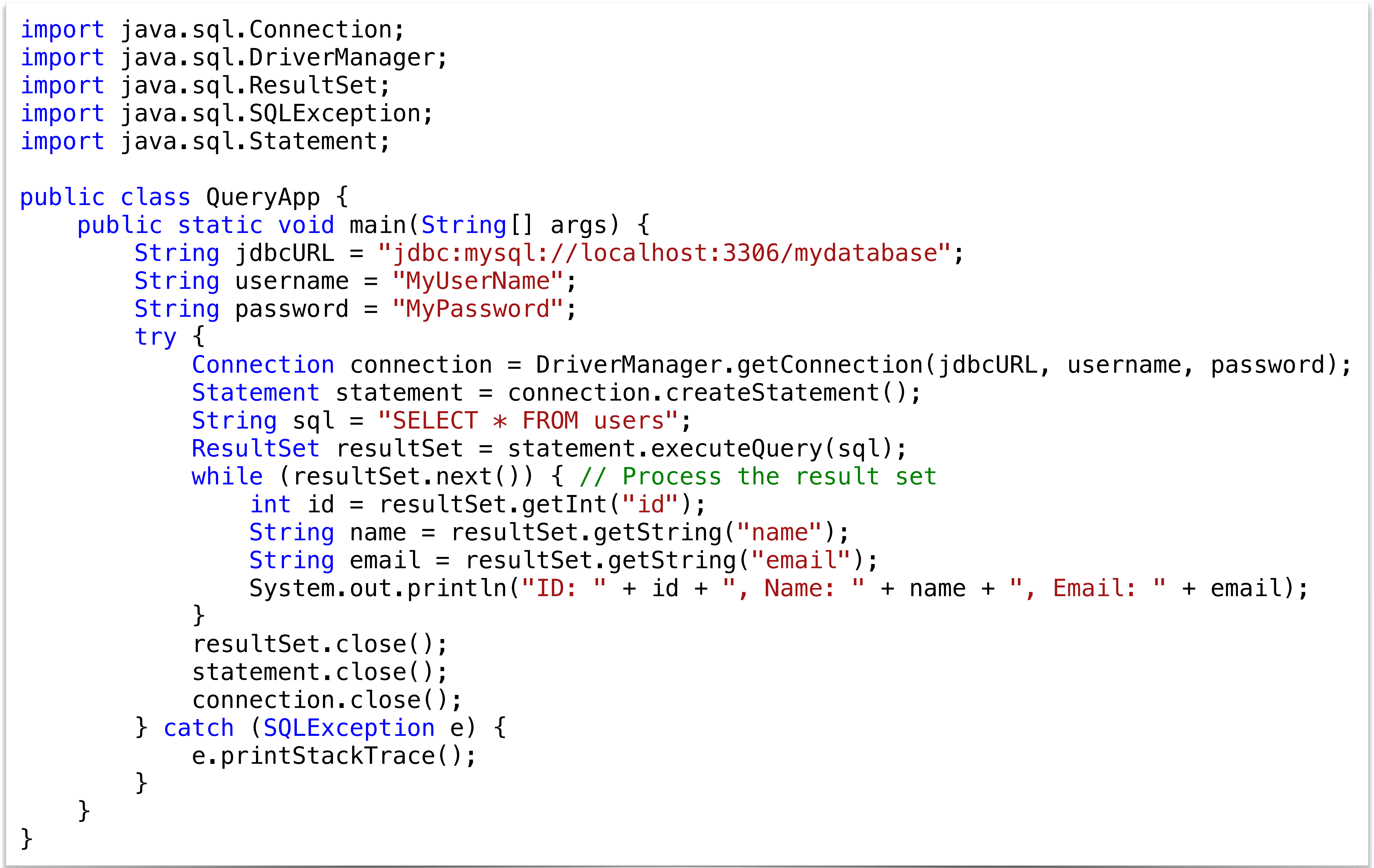


```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class DatabaseConnectionApp {
    public static void main(String[] args) {
        String jdbcURL = "jdbc:mysql://localhost:3306/mydatabase";
        String username = "MyUserName";
        String password = „MyPassword";
        try {
            // Establish connection to the database
            Connection connection = DriverManager.getConnection(jdbcURL, username, password);
            System.out.println("Connected to the database successfully!");
            // Close the connection after use
            connection.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Perform SQL Queries

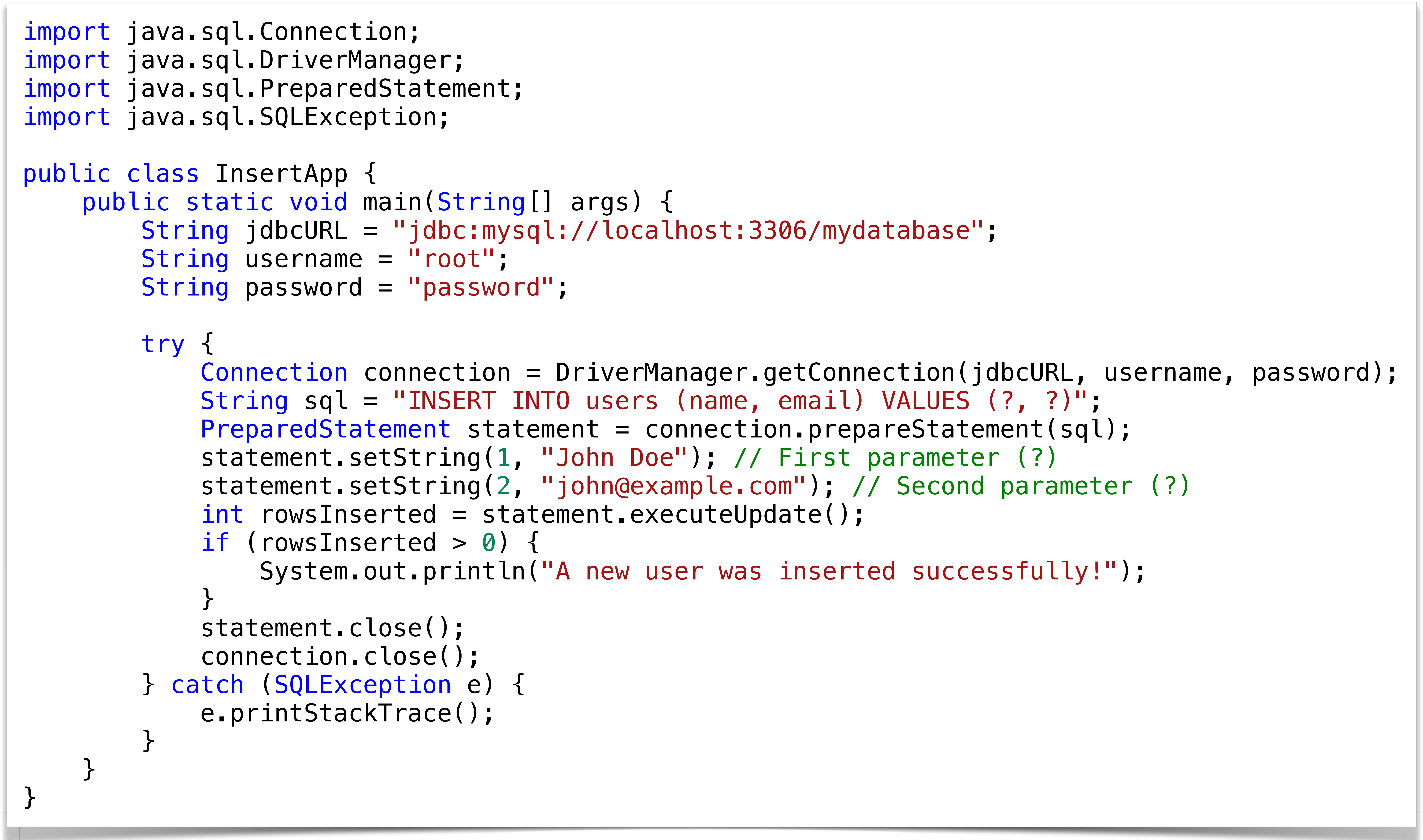
Once you have a connection, you can execute SQL statements such as SELECT, INSERT, UPDATE, and DELETE.

- Statement: Used for simple queries without parameters.
- PreparedStatement: Used for parameterized queries (helps prevent SQL injection).
- ResultSet: Represents the result set of a query.



```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class QueryApp {
    public static void main(String[] args) {
        String jdbcURL = "jdbc:mysql://localhost:3306/mydatabase";
        String username = "MyUserName";
        String password = "MyPassword";
        try {
            Connection connection = DriverManager.getConnection(jdbcURL, username, password);
            Statement statement = connection.createStatement();
            String sql = "SELECT * FROM users";
            ResultSet resultSet = statement.executeQuery(sql);
            while (resultSet.next()) { // Process the result set
                int id = resultSet.getInt("id");
                String name = resultSet.getString("name");
                String email = resultSet.getString("email");
                System.out.println("ID: " + id + ", Name: " + name + ", Email: " + email);
            }
            resultSet.close();
            statement.close();
            connection.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Executing an INSERT Query



```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public class InsertApp {
    public static void main(String[] args) {
        String jdbcURL = "jdbc:mysql://localhost:3306/mydatabase";
        String username = "root";
        String password = "password";

        try {
            Connection connection = DriverManager.getConnection(jdbcURL, username, password);
            String sql = "INSERT INTO users (name, email) VALUES (?, ?)";
            PreparedStatement statement = connection.prepareStatement(sql);
            statement.setString(1, "John Doe"); // First parameter (?)
            statement.setString(2, "john@example.com"); // Second parameter (?)
            int rowsInserted = statement.executeUpdate();
            if (rowsInserted > 0) {
                System.out.println("A new user was inserted successfully!");
            }
            statement.close();
            connection.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

- **Summary of Steps**:
  - Add the JDBC driver to your project (via Maven/Gradle or manually).
  - Establish a connection to the database using DriverManager.
  - Execute SQL queries using Statement or PreparedStatement.
  - Use ResultSet to handle query results.
  - Handle exceptions and close resources.

9th Part: Networking

- Socket-based communication
- HTTP Requests and Responses
- Remote method invocation

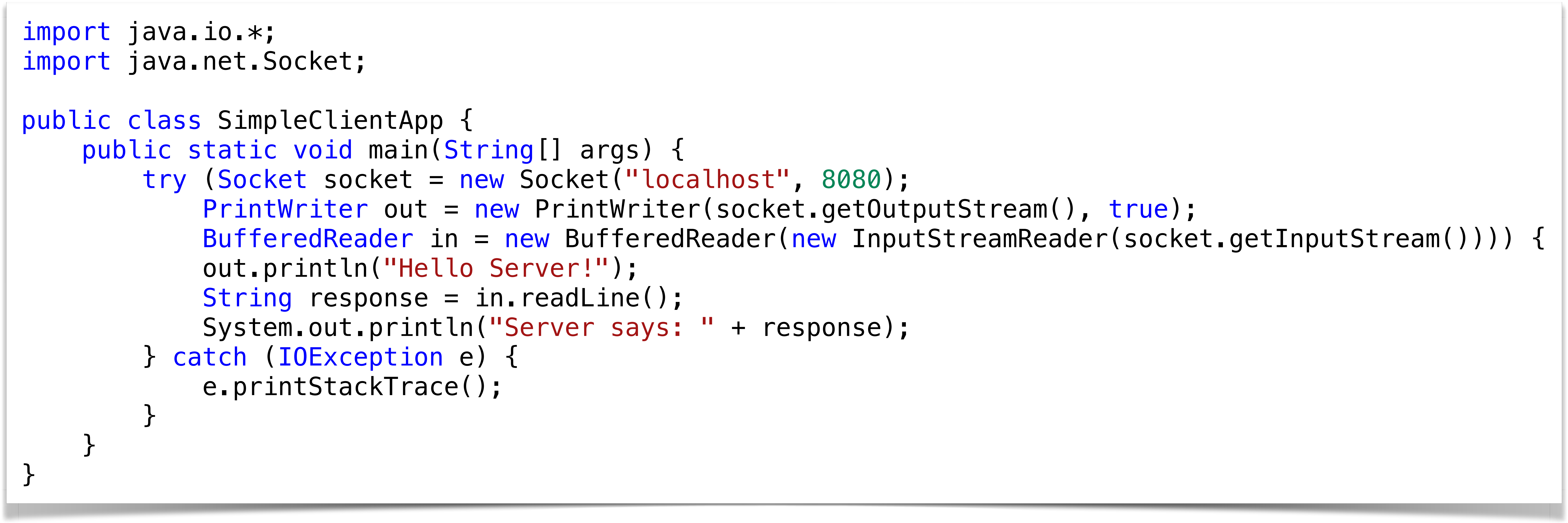
Networking

Java provides a powerful set of APIs for networking, enabling developers to create networked applications that communicate over TCP/IP, HTTP, or other protocols.

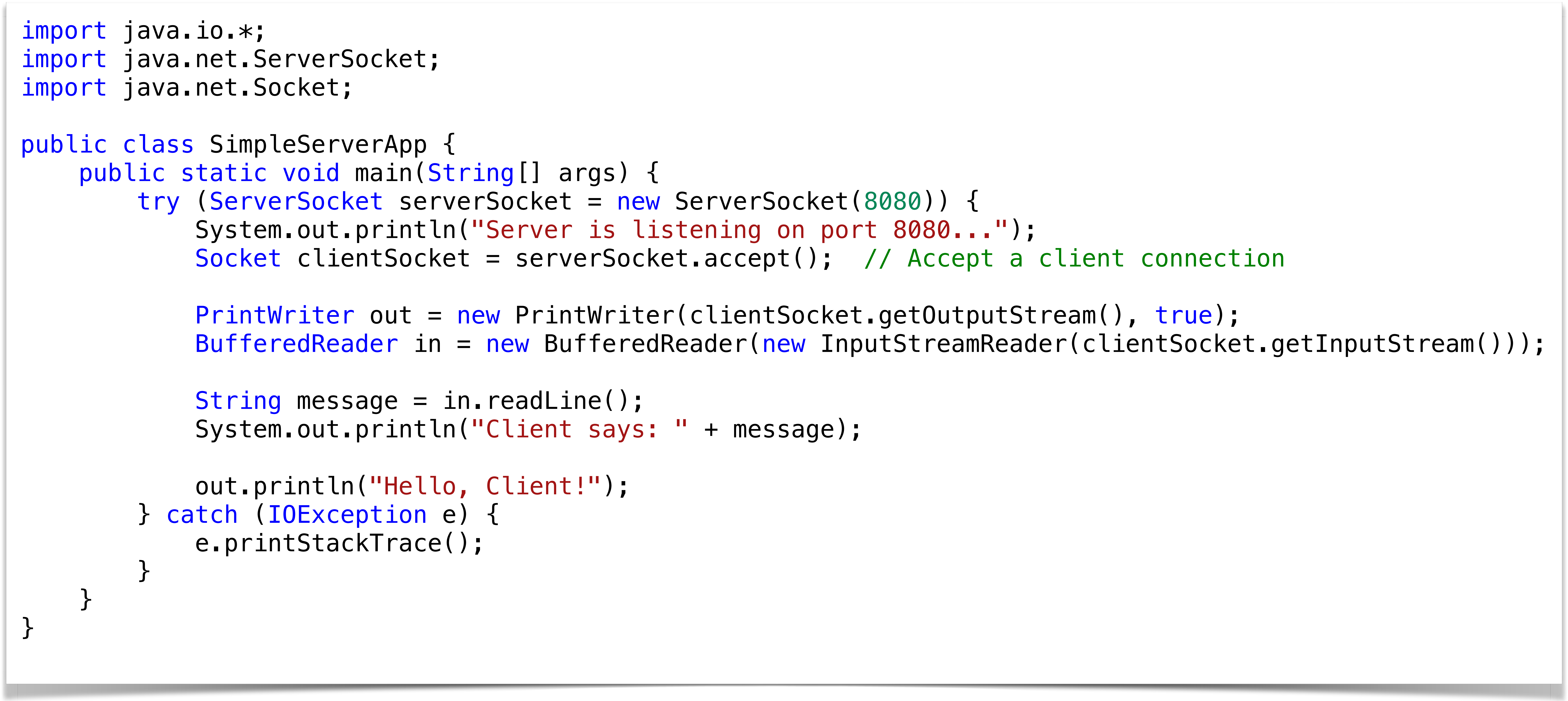
- Summary of Java Networking Classes:
  - Socket / ServerSocket: TCP communication.
  - DatagramSocket: UDP communication.
  - URL / HttpURLConnection: Web/HTTP access.
  - InetAddress: Handling IP addresses.
  - SocketChannel / ServerSocketChannel: Non-blocking I/O for networking.

Socket-based Communication Between Programs

A socket is one end of a two-way communication link between two programs running over the network.

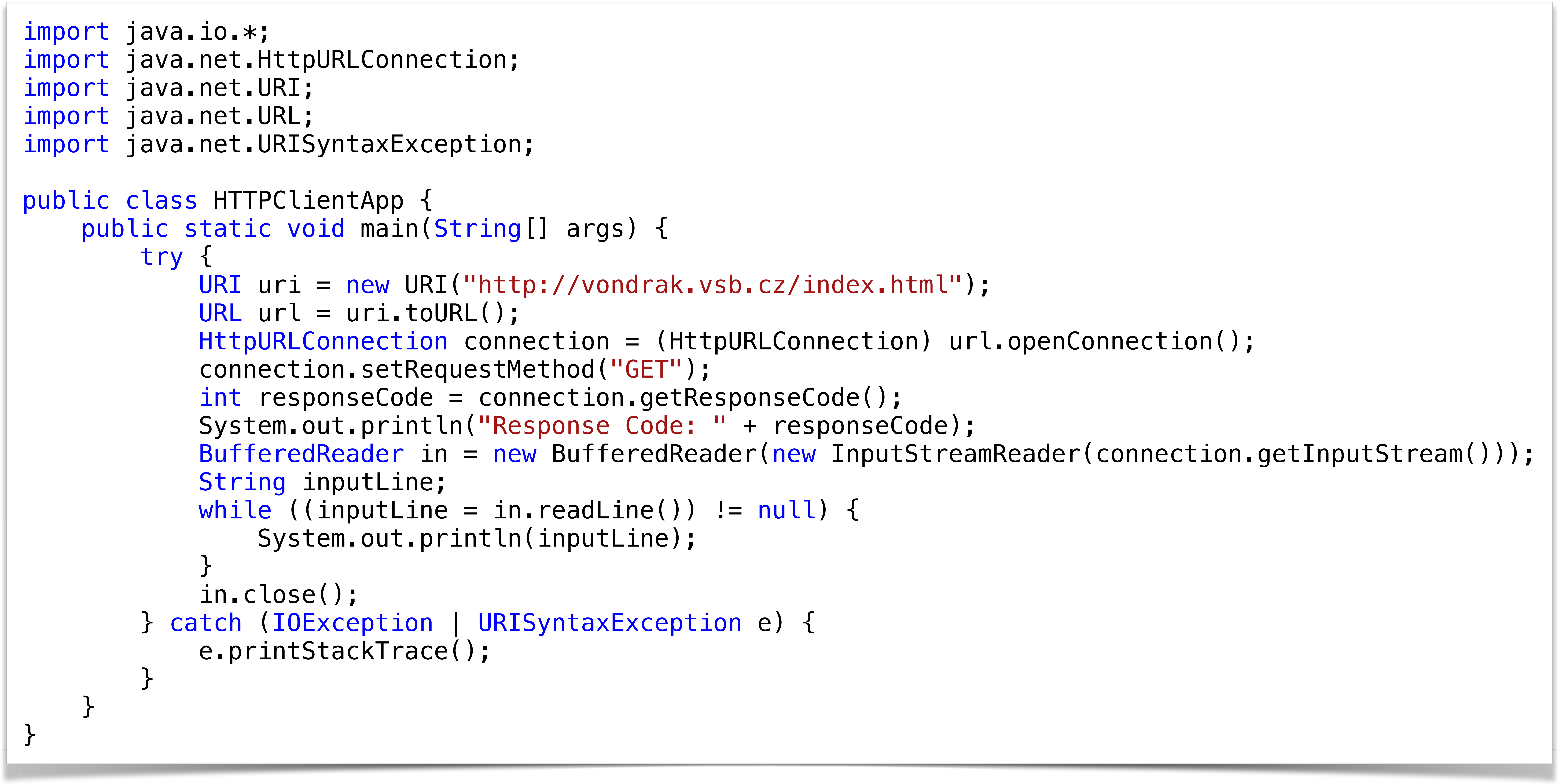


```
import java.io.*;
import java.net.Socket;

public class SimpleClientApp {
    public static void main(String[] args) {
        try (Socket socket = new Socket("localhost", 8080);
            PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
            BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()))) {
            out.println("Hello Server!");
            String response = in.readLine();
            System.out.println("Server says: " + response);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```



```java
import java.io.*;
import java.net.ServerSocket;
import java.net.Socket;

public class SimpleServerApp {
    public static void main(String[] args) {
        try (ServerSocket serverSocket = new ServerSocket(8080)) {
            System.out.println("Server is listening on port 8080...");
            Socket clientSocket = serverSocket.accept();  // Accept a client connection

            PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);
            BufferedReader in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));

            String message = in.readLine();
            System.out.println("Client says: " + message);

            out.println("Hello, Client!");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Control over HTTP Requests and Responses



```java
import java.io.*;
import java.net.HttpURLConnection;
import java.net.URI;
import java.net.URL;
import java.net.URISyntaxException;

public class HTTPClientApp {
    public static void main(String[] args) {
        try {
            URI uri = new URI("http://vondrak.vsb.cz/index.html");
            URL url = uri.toURL();
            HttpURLConnection connection = (HttpURLConnection) url.openConnection();
            connection.setRequestMethod("GET");
            int responseCode = connection.getResponseCode();
            System.out.println("Response Code: " + responseCode);
            BufferedReader in = new BufferedReader(new InputStreamReader(connection.getInputStream()));
            String inputLine;
            while ((inputLine = in.readLine()) != null) {
                System.out.println(inputLine);
            }
            in.close();
        } catch (IOException | URISyntaxException e) {
            e.printStackTrace();
        }
    }
}
```

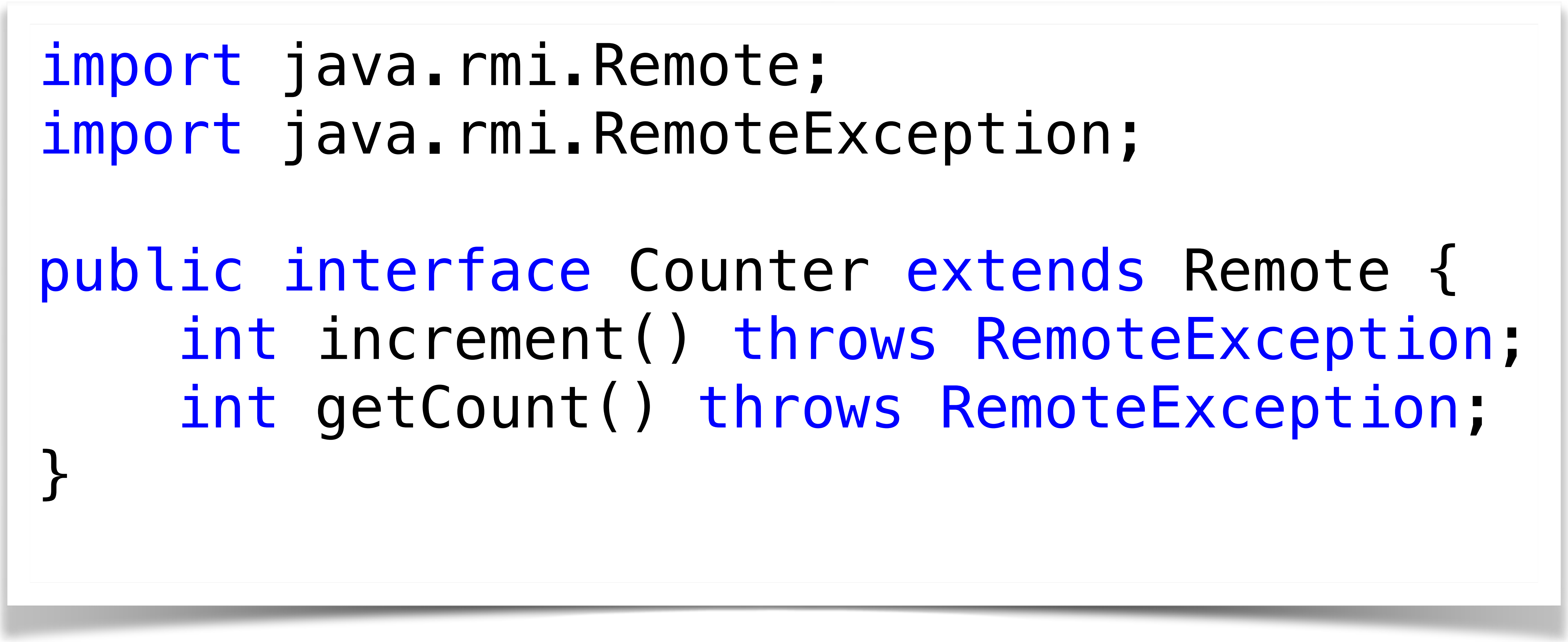
Remote Method Invocation

Java RMI (Remote Method Invocation) allows objects residing on different JVMs (even on different machines) to communicate with each other as if they were local.

- How Java RMI works:
  - Remote Interface: Defines the methods that can be called remotely.
  - Remote Object: Implements the remote interface and extends UnicastRemoteObject.
  - RMI Registry: Registers the remote objects so clients can look them up.
  - Client: Looks up the remote object in the registry and invokes methods on it.

Define the Remote Interface

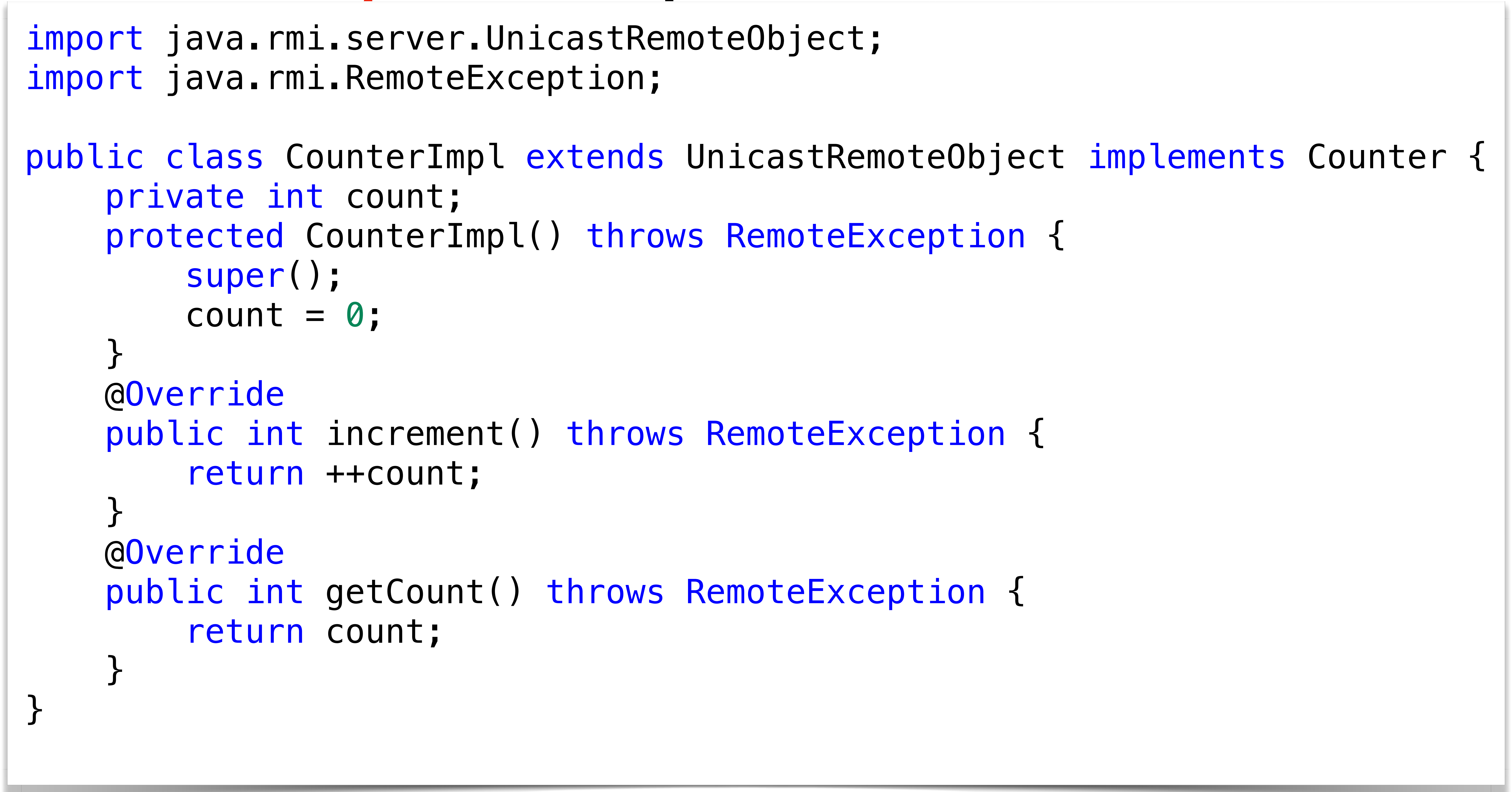
This interface defines the methods that can be called remotely. It must extend java.rmi.Remote and all methods must throw RemoteException.



```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Counter extends Remote {
    int increment() throws RemoteException;
    int getCount() throws RemoteException;
}
```

Implement the Remote Object

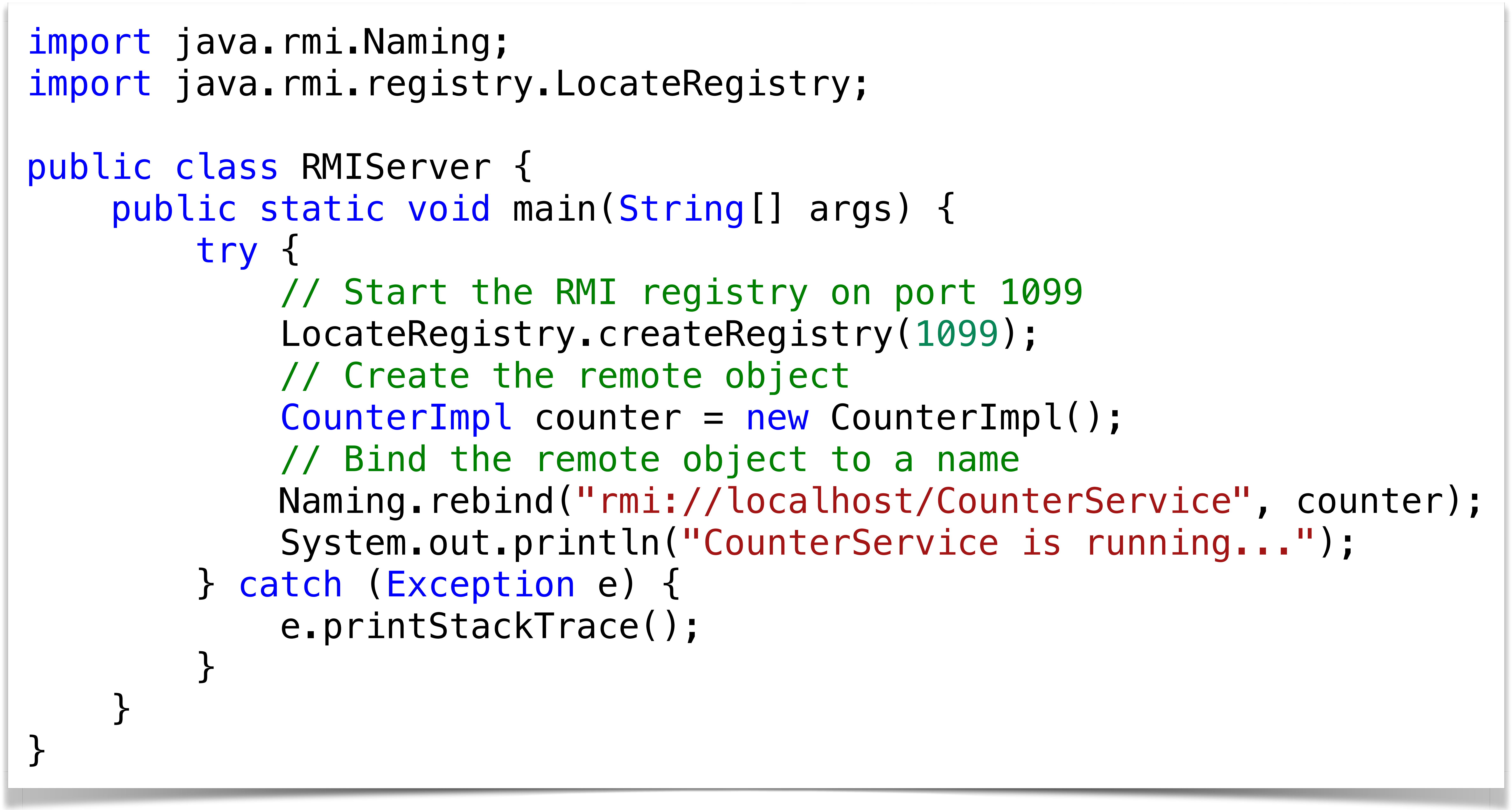
The class that implements the remote interface must extend UnicastRemoteObject and implement the methods.



```java
import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;

public class CounterImpl extends UnicastRemoteObject implements Counter {
    private int count;
    protected CounterImpl() throws RemoteException {
        super();
        count = 0;
    }
    @Override
    public int increment() throws RemoteException {
        return ++count;
    }
    @Override
    public int getCount() throws RemoteException {
        return count;
    }
}
```

Create the Server (RMI Registry Binding)

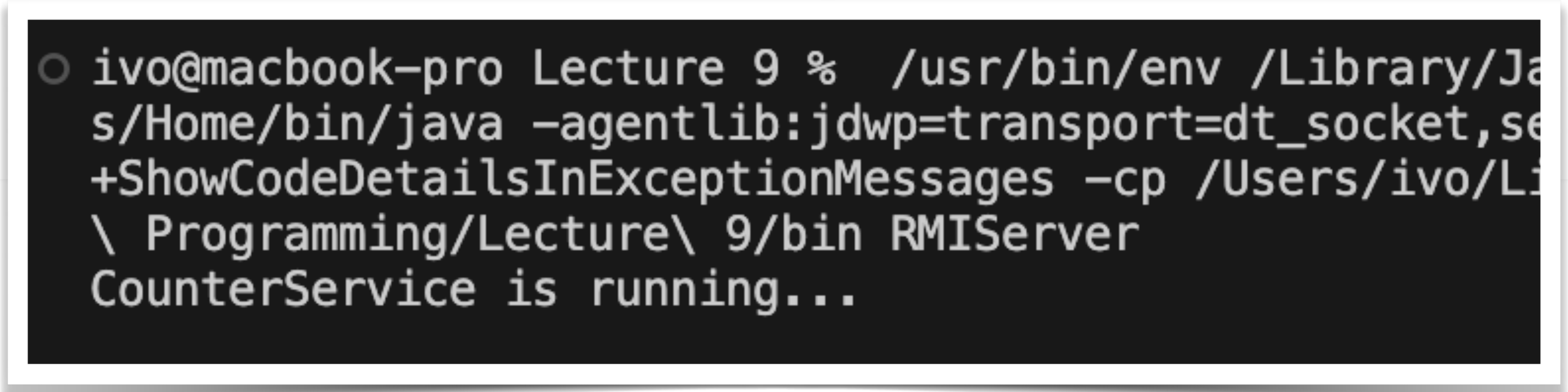
On the server side, you need to create the RMI registry and bind the remote object to it.



```
import java.rmi.Naming;
import java.rmi.registry.LocateRegistry;

public class RMIServer {
    public static void main(String[] args) {
        try {
            // Start the RMI registry on port 1099
            LocateRegistry.createRegistry(1099);
            // Create the remote object
            CounterImpl counter = new CounterImpl();
            // Bind the remote object to a name
            Naming.rebind("rmi://localhost/CounterService", counter);
            System.out.println("CounterService is running...");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

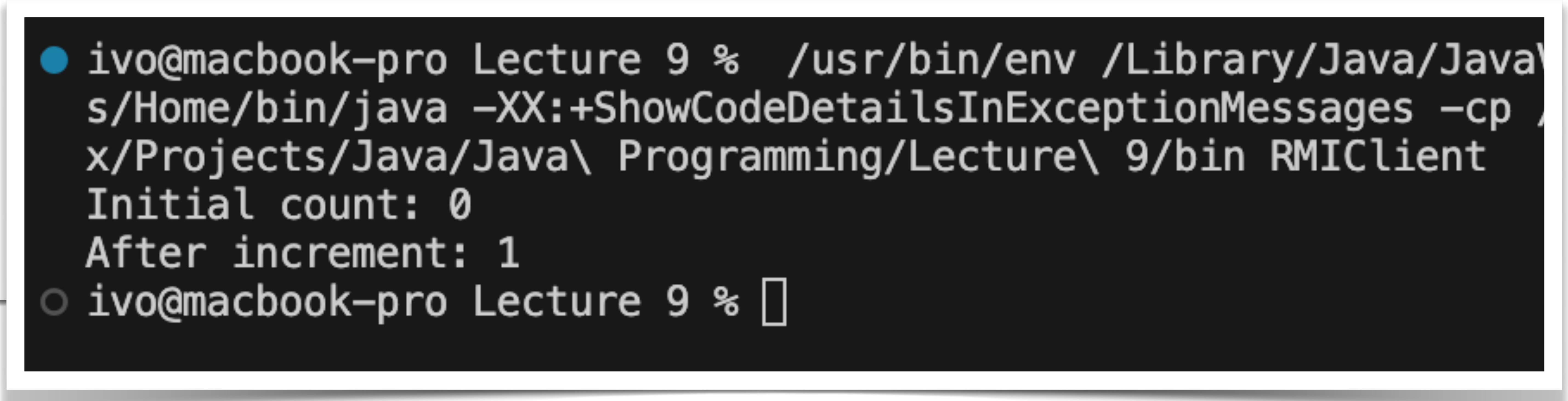
Create the Client

On the client side, you look up the remote object and invoke methods on it.



```
○ ivo@macbook-pro Lecture 9 %  /usr/bin/env /Library/Ja
  s/Home/bin/java -agentlib:jdwp=transport=dt_socket,se
  +ShowCodeDetailsInExceptionMessages -cp /Users/ivo/Li
  \ Programming/Lecture\ 9/bin RMIServer 
  CounterService is running...
```

```java
import java.rmi.Naming;

public class RMIClient {
    public static void main(String[] args) {
        try {
            // Look up the remote object from the RMI registry
            Counter counter = (Counter) Naming.lookup("rmi://localhost/CounterService");
            // Call methods on the remote object
            System.out.println("Initial count: " + counter.getCount());
            counter.increment();
            System.out.println("After increment: " + counter.getCount());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```



```
● ivo@macbook-pro Lecture 9 %  /usr/bin/env /Library/Java/Java\
  s/Home/bin/java -XX:+ShowCodeDetailsInExceptionMessages -cp /
  x/Projects/Java/Java\ Programming/Lecture\ 9/bin RMIClient 
  Initial count: 0
  After increment: 1
○ ivo@macbook-pro Lecture 9 % ▯
```

RMI Summary

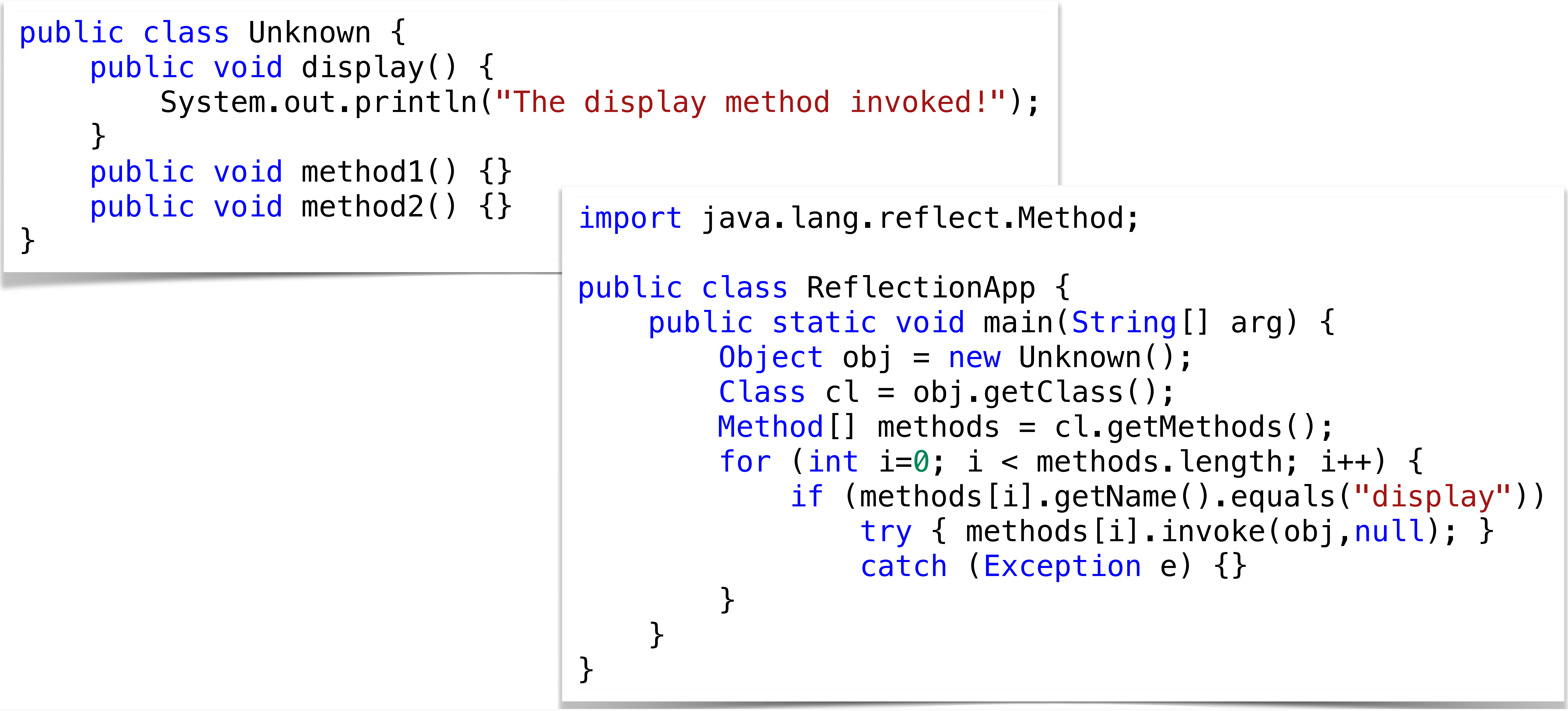
- **Common Uses of RMI**:
  - Distributed systems where multiple JVMs need to communicate.
  - Client-server applications (e.g., a chat system or distributed task execution).
  - Remote object management and interaction in enterprise systems.
- **Limitations of RMI**:
  - Java RMI works only with Java applications.
  - It’s a relatively old technology and may not be ideal for modern web-based distributed applications, where alternatives like gRPC or RESTful APIs are more popular.

10th Part: Final Notes

Java Reflection

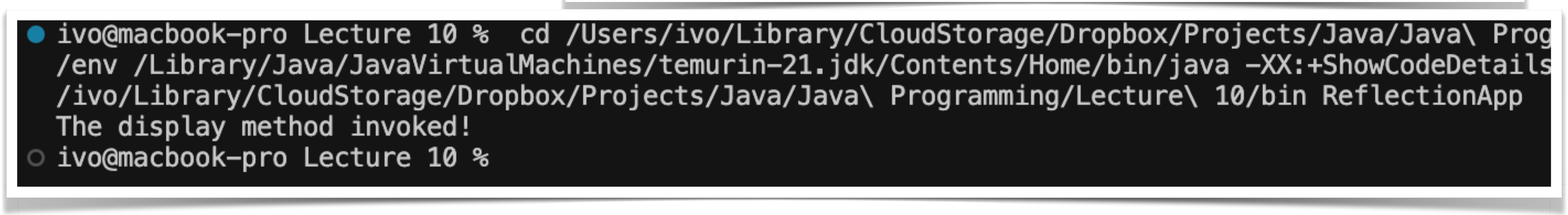
Java reflection tools enable introspection about the classes and objects in the current JVM

- A Field object represents a reflected field (a class variable or an instance variable).
- A Method object represents a reflected method (an abstract method, an instance method, or a class method).
- A Constructor object represents a reflected constructor



```java
public class Unknown {
    public void display() {
        System.out.println("The display method invoked!");
    }
    public void method1() {}
    public void method2() {}
}
```

```java
import java.lang.reflect.Method;

public class ReflectionApp {
    public static void main(String[] arg) {
        Object obj = new Unknown();
        Class cl = obj.getClass();
        Method[] methods = cl.getMethods();
        for (int i=0; i < methods.length; i++) {
            if (methods[i].getName().equals("display"))
                try { methods[i].invoke(obj,null); }
                catch (Exception e) {}
        }
    }
}
```



```
● ivo@macbook-pro Lecture 10 %  cd /Users/ivo/Library/CloudStorage/Dropbox/Projects/Java/Java\ Prog
  /env /Library/Java/JavaVirtualMachines/temurin-21.jdk/Contents/Home/bin/java -XX:+ShowCodeDetails
  /ivo/Library/CloudStorage/Dropbox/Projects/Java/Java\ Programming/Lecture\ 10/bin ReflectionApp
  The display method invoked!
○ ivo@macbook-pro Lecture 10 %
```

Annotations

Annotations have a number of uses, among them:

- Information for the compiler: Annotations can be used by the compiler to detect errors or suppress warnings
- Compiler-time and deployment-time processing: Software tools can process annotation information to generate code, XML files, and so forth
- Runtime processing: Some annotations are available to be examined at runtime (reflection)

Annotations Used by the Compiler

There are three annotation types that are predefined by the language specification itself

- @Deprecated: indicates that the marked element is deprecated and should no longer be used
- @Override: informs the compiler that the element is meant to override an element declared in a superclass
- @SuppressWarnings: tells the compiler to suppress specific warnings that it would otherwise generate