Java Recap

J… What

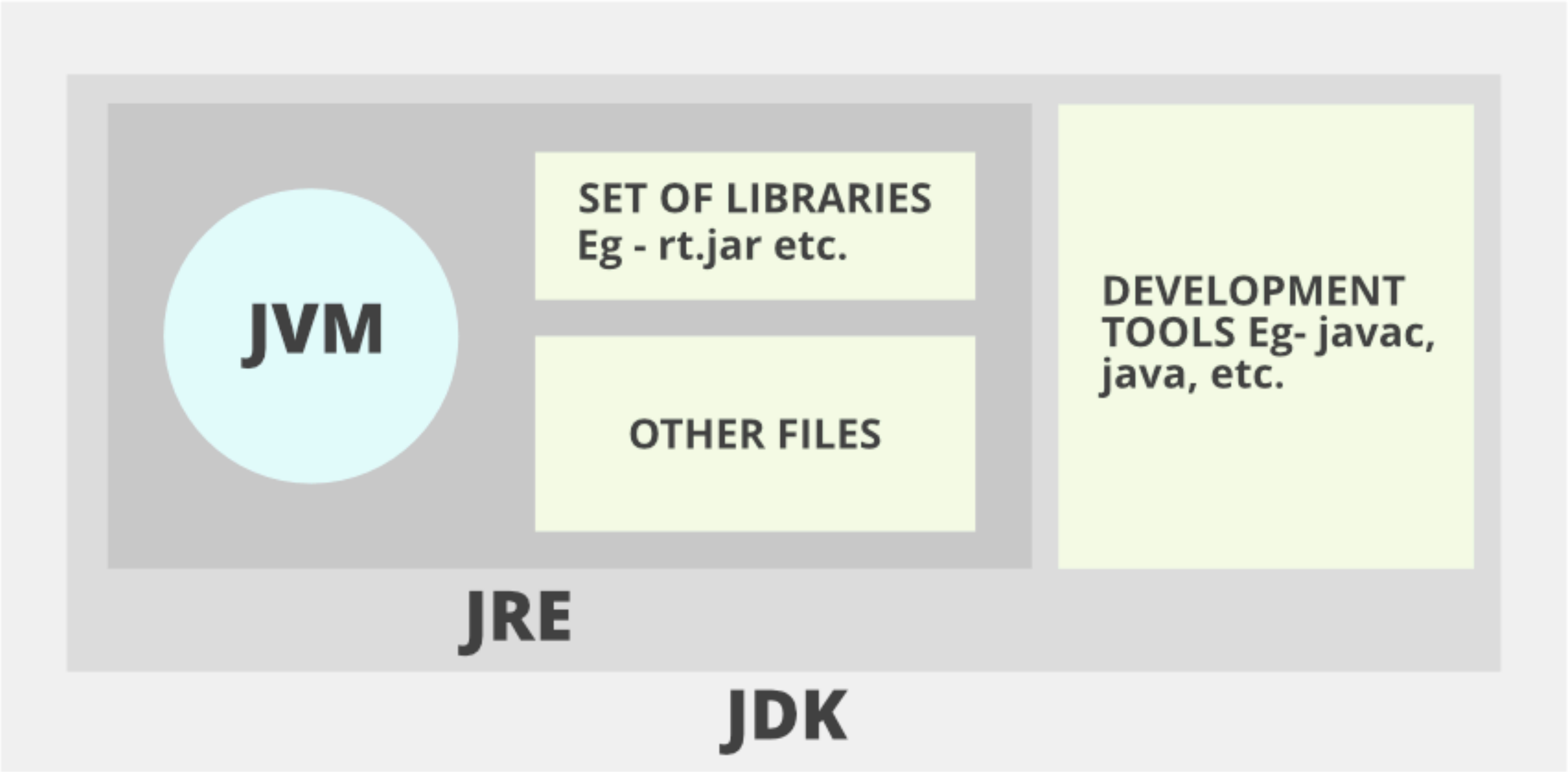
- A programming language & computing Plattform
  - Java vs JRE vs JDK
- Write Once Run Anywhere
- Free to use
- high-level language, class-based and object-oriented
- Statically typed
  - all variables must be declared stating its type and name

What Java is used for

- Server Applications
- Android Apps
- Desktop GUI Application
- Embedded Systems

JDK vs JRE vs JVM

- **JDK** (Java Development Kit) provides the environment to **develop and execute(run)** the Java program
  - Development Tools(to provide an environment to develop your java programs)
  - JRE (to execute your java program)



JVM

SET OF LIBRARIES
Eg - rt.jar etc.

OTHER FILES

DEVELOPMENT TOOLS Eg- javac, java, etc.

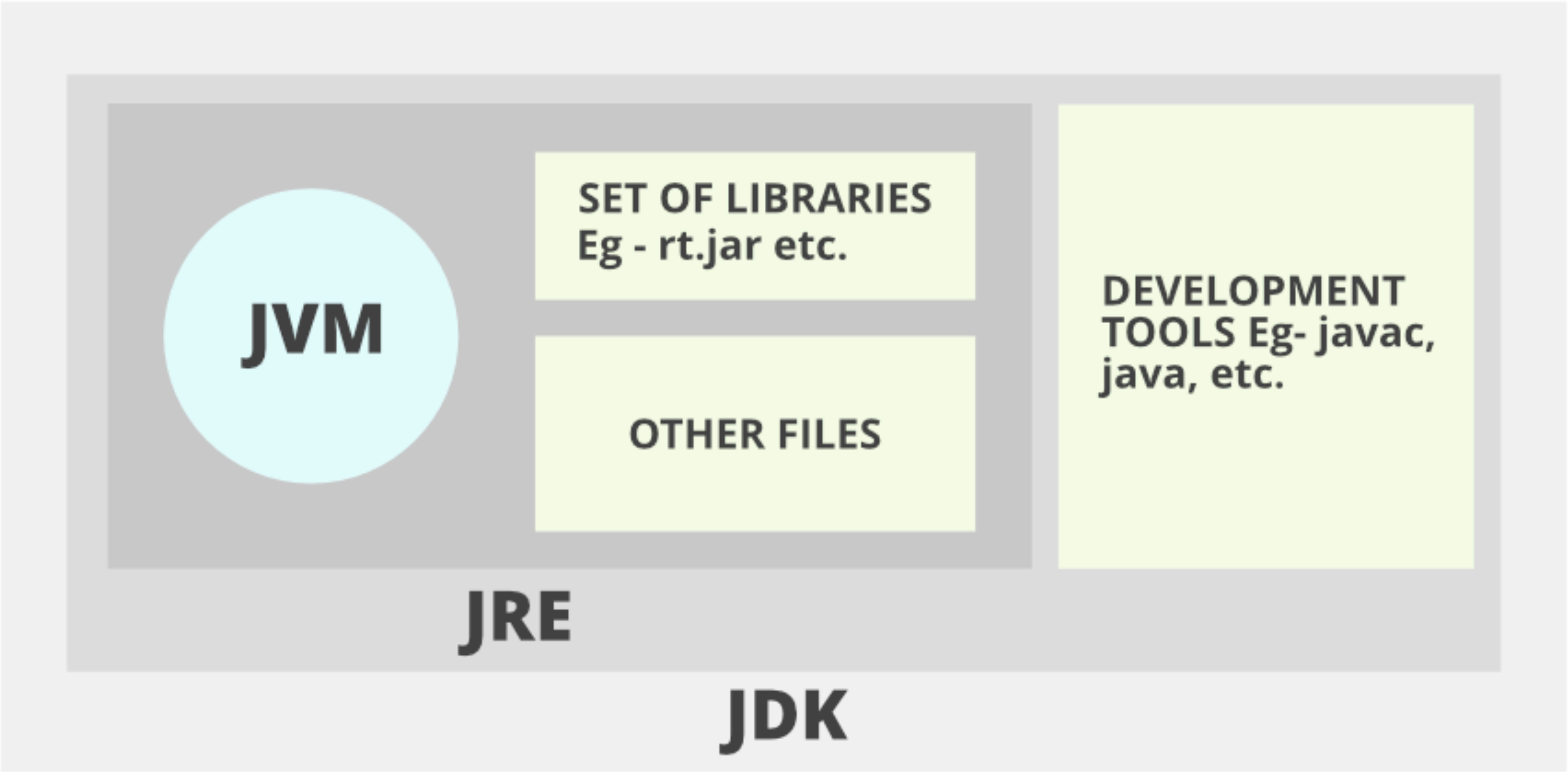
JRE

JDK

JDK vs **JRE** vs JVM

- **JRE** (Java Runtime Environment) is an installation package that provides an environment to **only run(not develop)** java programs

- only used by those who only want to run Java programs / end-users of your applications (e.g. PCs, Servers)


JVM
SET OF LIBRARIES Eg - rt.jar etc.
OTHER FILES
DEVELOPMENT TOOLS Eg- javac, java, etc.
JRE
JDK

JDK vs JRE vs **JVM**
(Java Virtual Machine)

- important part of JDK and JRE
- Whatever Java program you run using JRE or JDK goes into JVM
- responsible for executing the java program line by line
- This bytecode gets interpreted on different machines
    - converts Java bytecode into machines language

- Java byte code —> intermediate language
    - High-level languages that compile to java byte code: Java, Kotlin, Scala, Groovy, …

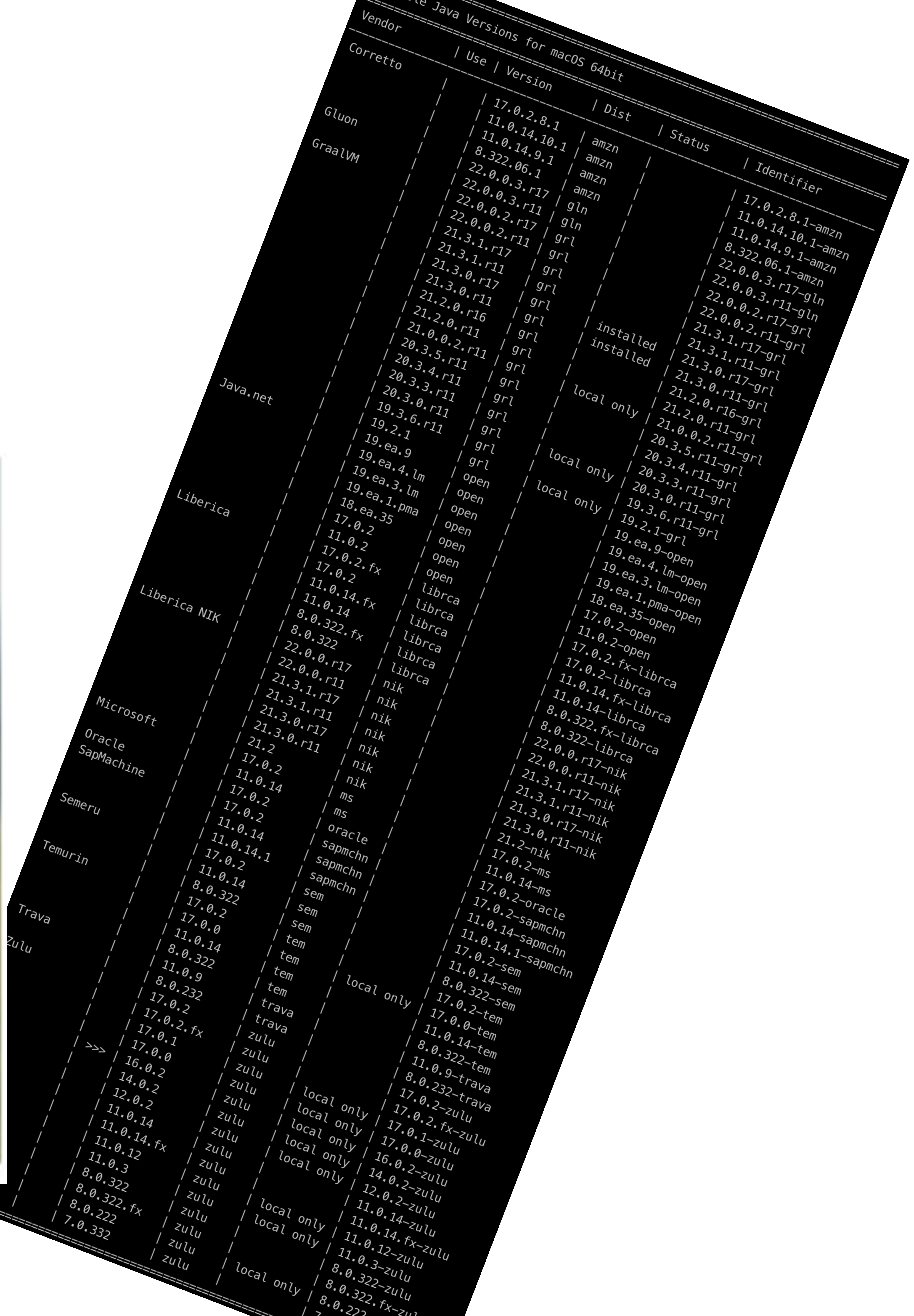
JVM

Garbage Collection

- JVM creates a heap area on startup
  - known as runtime data area where all the objects (instances of class) are stored
  - Since this area is limited, it is required to manage this area efficiently by removing the objects that are no longer in use
  - process of removing unused objects from heap memory is known as **Garbage collection** (part of memory management in Java)

Different JDKs

(Java Development Kit)

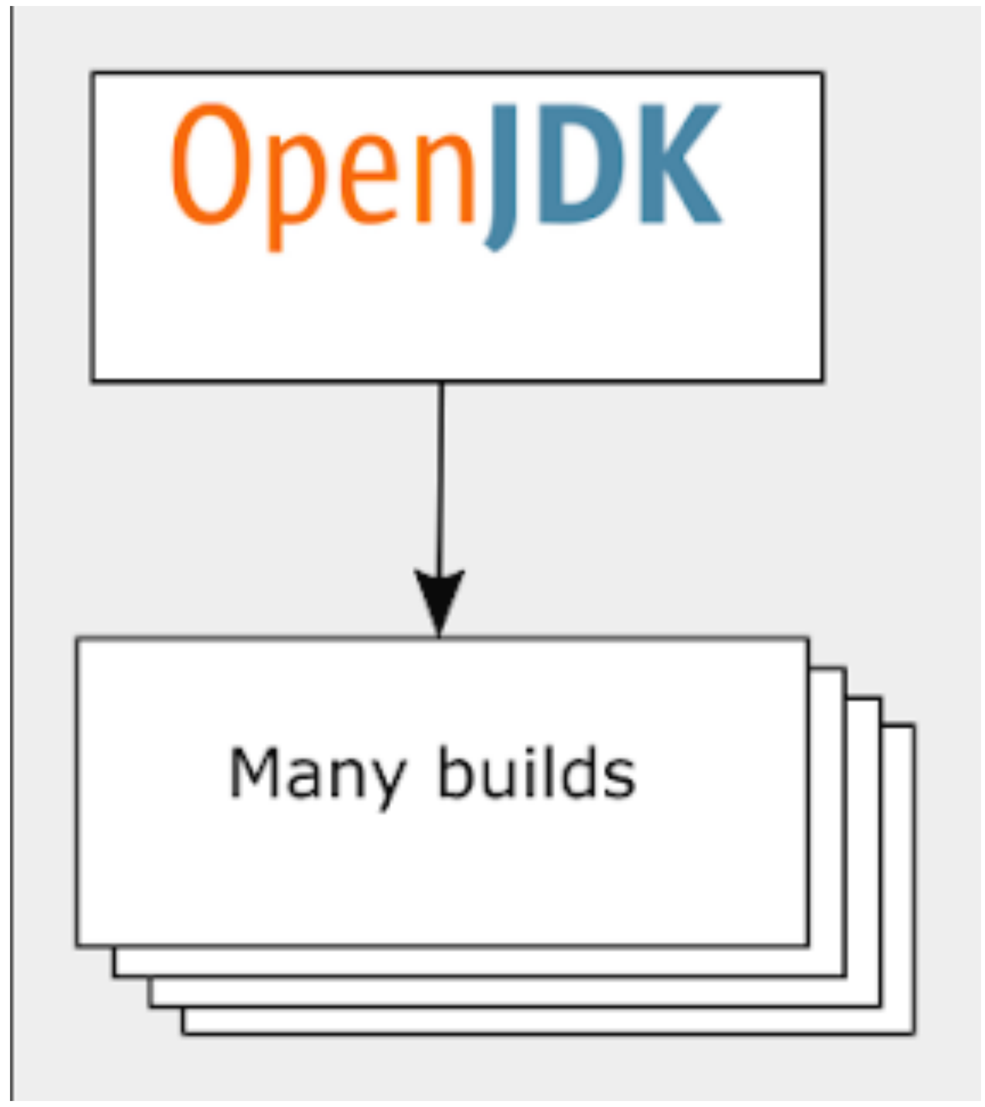


```
================================================================================
le Java Versions for macOS 64bit
================================================================================
 Vendor        | Use | Version      | Dist    | Status     | Identifier
--------------------------------------------------------------------------------
 Corretto      |     | 17.0.2.8.1   | amzn    |            | 17.0.2.8.1-amzn
               |     | 11.0.14.10.1 | amzn    |            | 11.0.14.10.1-amzn
               |     | 11.0.14.9.1  | amzn    |            | 11.0.14.9.1-amzn
               |     | 8.322.06.1   | amzn    |            | 8.322.06.1-amzn
 Gluon         |     | 22.0.0.3.r17 | gln     |            | 22.0.0.3.r17-gln
               |     | 22.0.0.3.r11 | gln     |            | 22.0.0.3.r11-gln
 GraalVM       |     | 22.0.0.2.r17 | grl     |            | 22.0.0.2.r17-grl
               |     | 22.0.0.2.r11 | grl     |            | 22.0.0.2.r11-grl
               |     | 21.3.1.r17   | grl     |            | 21.3.1.r17-grl
               |     | 21.3.1.r11   | grl     |            | 21.3.1.r11-grl
               |     | 21.3.0.r17   | grl     | installed  | 21.3.0.r17-grl
               |     | 21.3.0.r11   | grl     | installed  | 21.3.0.r11-grl
               |     | 21.2.0.r16   | grl     |            | 21.2.0.r16-grl
               |     | 21.2.0.r11   | grl     |            | 21.2.0.r11-grl
               |     | 21.0.0.2.r11 | grl     | local only | 21.0.0.2.r11-grl
               |     | 20.3.5.r11   | grl     |            | 20.3.5.r11-grl
               |     | 20.3.4.r11   | grl     |            | 20.3.4.r11-grl
               |     | 20.3.3.r11   | grl     |            | 20.3.3.r11-grl
               |     | 20.3.0.r11   | grl     | local only | 20.3.0.r11-grl
               |     | 19.3.6.r11   | grl     |            | 19.3.6.r11-grl
               |     | 19.2.1       | grl     | local only | 19.2.1-grl
 Java.net      |     | 19.ea.9      | open    |            | 19.ea.9-open
               |     | 19.ea.4.lm   | open    |            | 19.ea.4.lm-open
               |     | 19.ea.3.lm   | open    |            | 19.ea.3.lm-open
               |     | 19.ea.1.pma  | open    |            | 19.ea.1.pma-open
               |     | 18.ea.35     | open    |            | 18.ea.35-open
               |     | 17.0.2       | open    |            | 17.0.2-open
               |     | 11.0.2       | open    |            | 11.0.2-open
 Liberica      |     | 17.0.2.fx    | librca  |            | 17.0.2.fx-librca
               |     | 17.0.2       | librca  |            | 17.0.2-librca
               |     | 11.0.14.fx   | librca  |            | 11.0.14.fx-librca
               |     | 11.0.14      | librca  |            | 11.0.14-librca
               |     | 8.0.322.fx   | librca  |            | 8.0.322.fx-librca
               |     | 8.0.322      | librca  |            | 8.0.322-librca
 Liberica NIK  |     | 22.0.0.r17   | nik     |            | 22.0.0.r17-nik
               |     | 22.0.0.r11   | nik     |            | 22.0.0.r11-nik
               |     | 21.3.1.r17   | nik     |            | 21.3.1.r17-nik
               |     | 21.3.1.r11   | nik     |            | 21.3.1.r11-nik
               |     | 21.3.0.r17   | nik     |            | 21.3.0.r17-nik
               |     | 21.3.0.r11   | nik     |            | 21.3.0.r11-nik
               |     | 21.2         | nik     |            | 21.2-nik
 Microsoft     |     | 17.0.2       | ms      |            | 17.0.2-ms
               |     | 11.0.14      | ms      |            | 11.0.14-ms
 Oracle        |     | 17.0.2       | oracle  |            | 17.0.2-oracle
 SapMachine    |     | 17.0.2       | sapmchn |            | 17.0.2-sapmchn
               |     | 11.0.14      | sapmchn |            | 11.0.14-sapmchn
               |     | 11.0.14.1    | sapmchn |            | 11.0.14.1-sapmchn
 Semeru        |     | 17.0.2       | sem     |            | 17.0.2-sem
               |     | 11.0.14      | sem     |            | 11.0.14-sem
               |     | 8.0.322      | sem     |            | 8.0.322-sem
 Temurin       |     | 17.0.2       | tem     |            | 17.0.2-tem
               |     | 17.0.0       | tem     | local only | 17.0.0-tem
               |     | 11.0.14      | tem     |            | 11.0.14-tem
               |     | 8.0.322      | tem     |            | 8.0.322-tem
 Trava         |     | 11.0.9       | trava   |            | 11.0.9-trava
               |     | 8.0.232      | trava   |            | 8.0.232-trava
 Zulu          |     | 17.0.2       | zulu    |            | 17.0.2-zulu
               |     | 17.0.2.fx    | zulu    |            | 17.0.2.fx-zulu
               |     | 17.0.1       | zulu    | local only | 17.0.1-zulu
               | >>> | 17.0.0       | zulu    | local only | 17.0.0-zulu
               |     | 16.0.2       | zulu    | local only | 16.0.2-zulu
               |     | 14.0.2       | zulu    | local only | 14.0.2-zulu
               |     | 12.0.2       | zulu    | local only | 12.0.2-zulu
               |     | 11.0.14      | zulu    |            | 11.0.14-zulu
               |     | 11.0.14.fx   | zulu    |            | 11.0.14.fx-zulu
               |     | 11.0.12      | zulu    | local only | 11.0.12-zulu
               |     | 11.0.3       | zulu    | local only | 11.0.3-zulu
               |     | 8.0.322      | zulu    |            | 8.0.322-zulu
               |     | 8.0.322.fx   | zulu    |            | 8.0.322.fx-zul
               |     | 8.0.222      | zulu    | local only | 8.0.222
               |     | 7.0.332      | zulu    |            | 7
================================================================================
```

Different JDKs

(Java Development Kit)

- The truth: https://github.com/openjdk/jdk/
- Provided by different vendors
  - Oracle JDK —> branded with paid commercial support
  - Azul Zulu —> branded with paid commercial support
  - AdoptOpenJDK / Adoptium Temurin—> free and unbranded



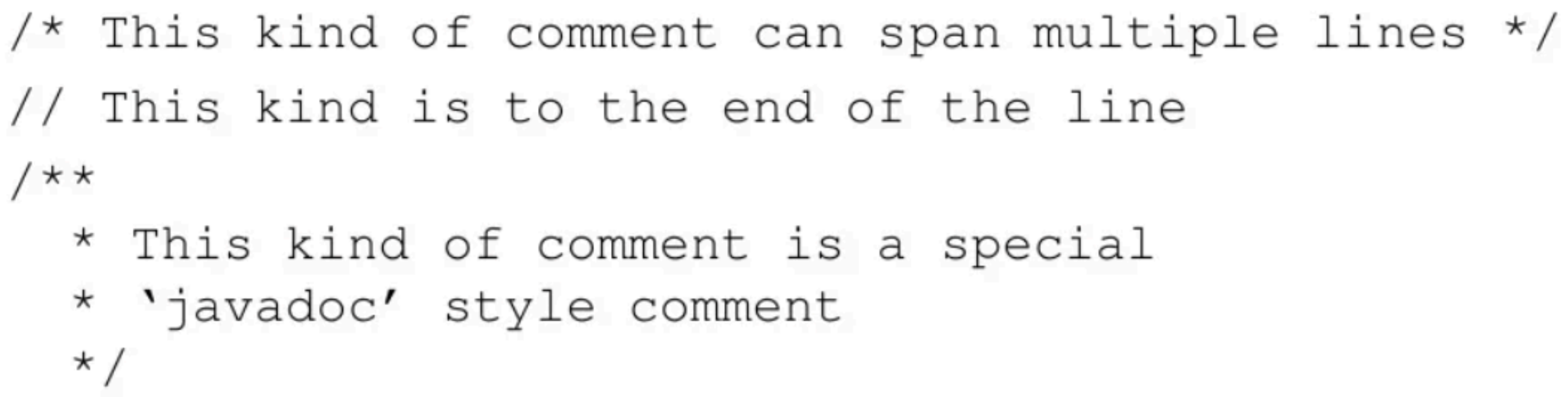
OOP
(Object Oriented Programming)

- a programming paradigm —> concept of „objects"
  - Object
    - data
      - Fields (often known as attributes or properties)
    - code
      - in the form of procedures (often known as methods)

Usage

Syntax

Comments

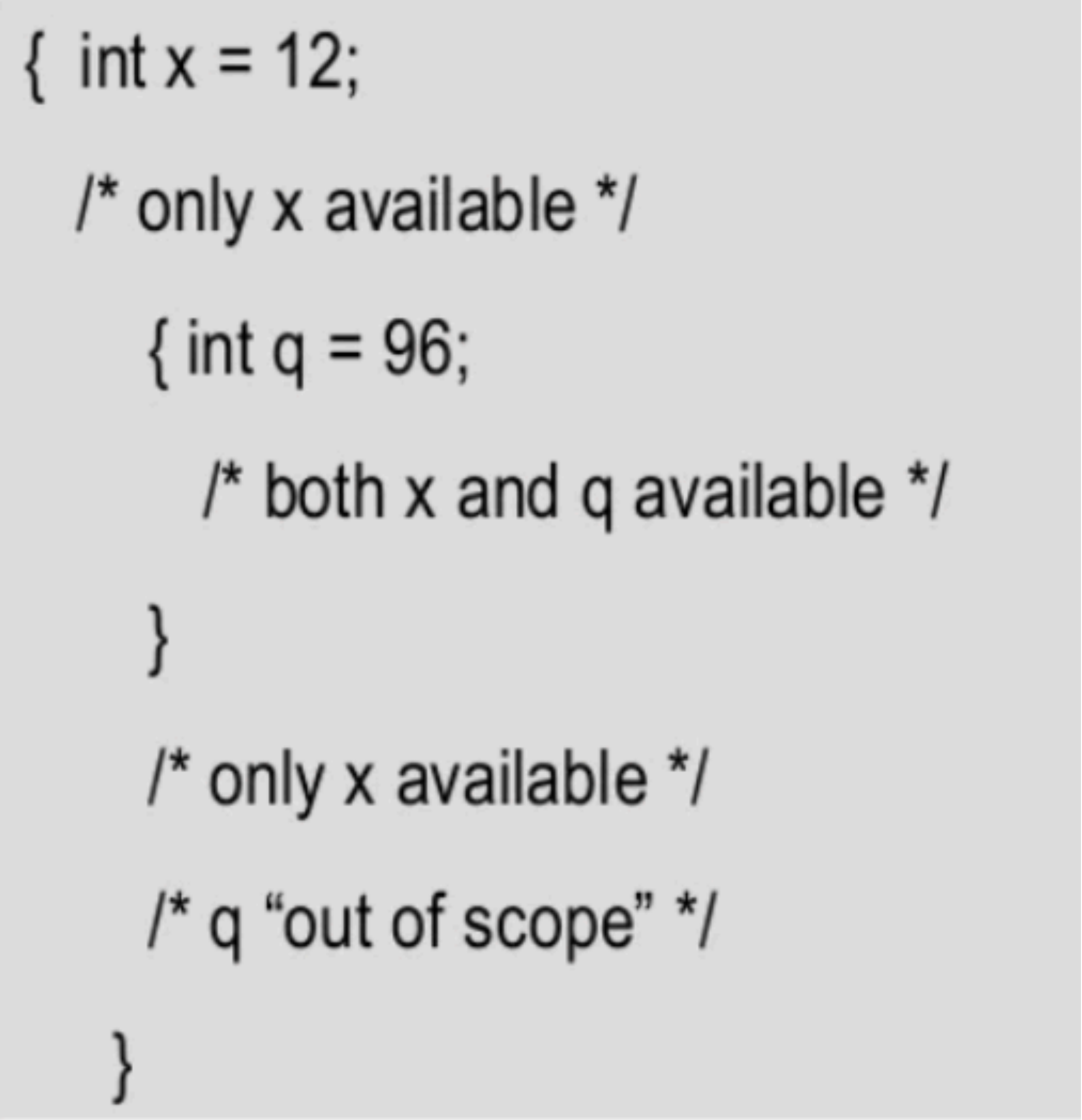


```
/* This kind of comment can span multiple lines */
// This kind is to the end of the line
/**
  * This kind of comment is a special
  * ‘javadoc’ style comment
  */
```

Syntax

Scoping

- A scope is determined by the placement of curly braces {}
- A variable defined within a scope is available only to the end of that scope



```
{  int x = 12;
  /* only x available */
     { int q = 96;
       /* both x and q available */
     }
     /* only x available */
     /* q “out of scope” */
   }
```

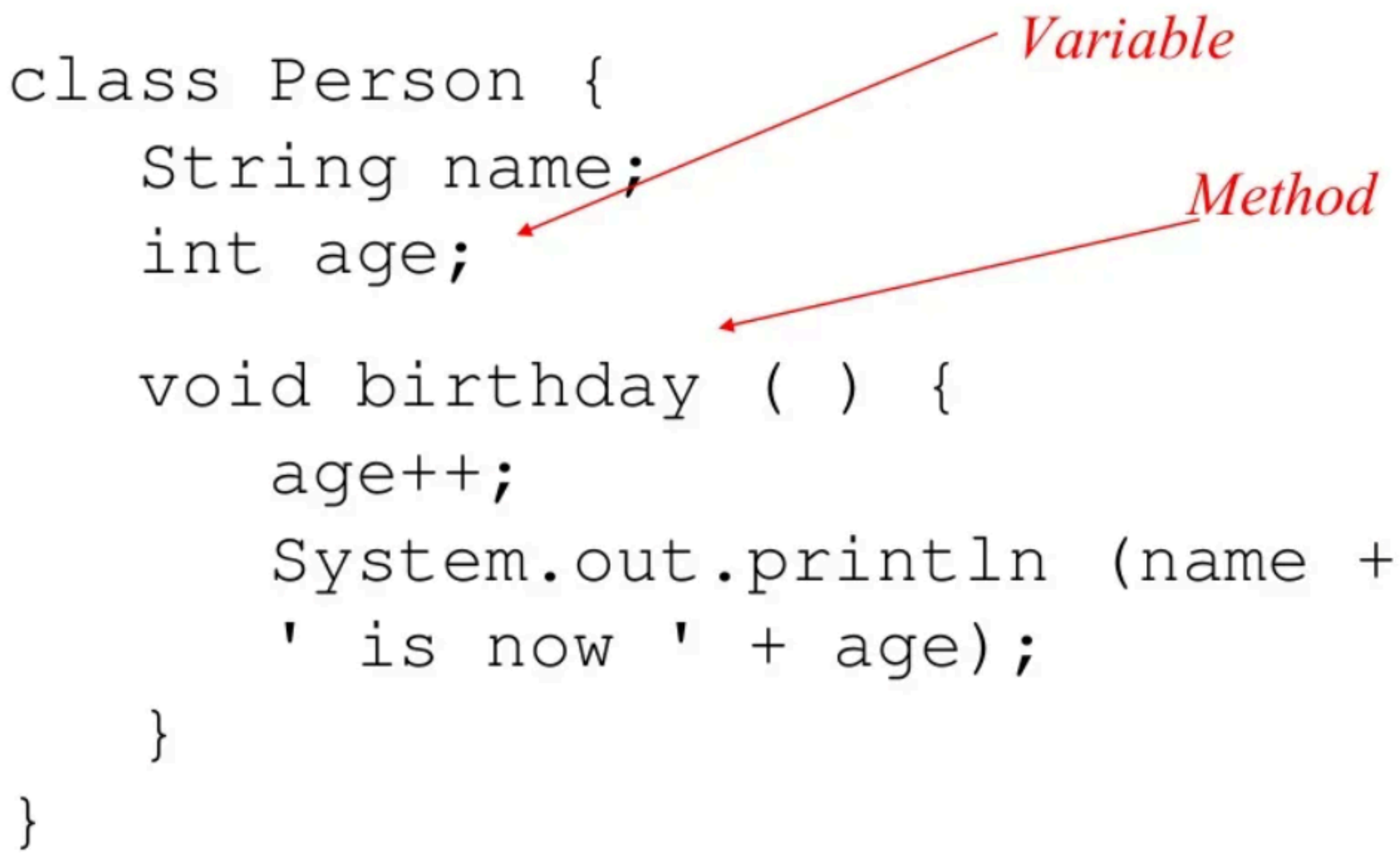
Syntax

Classes

- The class is the fundamental concept in JAVA (and other 00PLs)
- A class describes some data object(s), and the operations (or methods) that can be applied to those objects
- Every object and method in Java belongs to a class
- Classes have data (fields) and code (methods) and classes (member classes or inner classes)
- Static methods and fields belong to the class itself
- Others belong to instances

Syntax

A Class



```
class Person {
   String name;
   int age;

   void birthday ( ) {
      age++;
      System.out.println (name +
      ' is now ' + age);
   }
}
```

Variable
Method

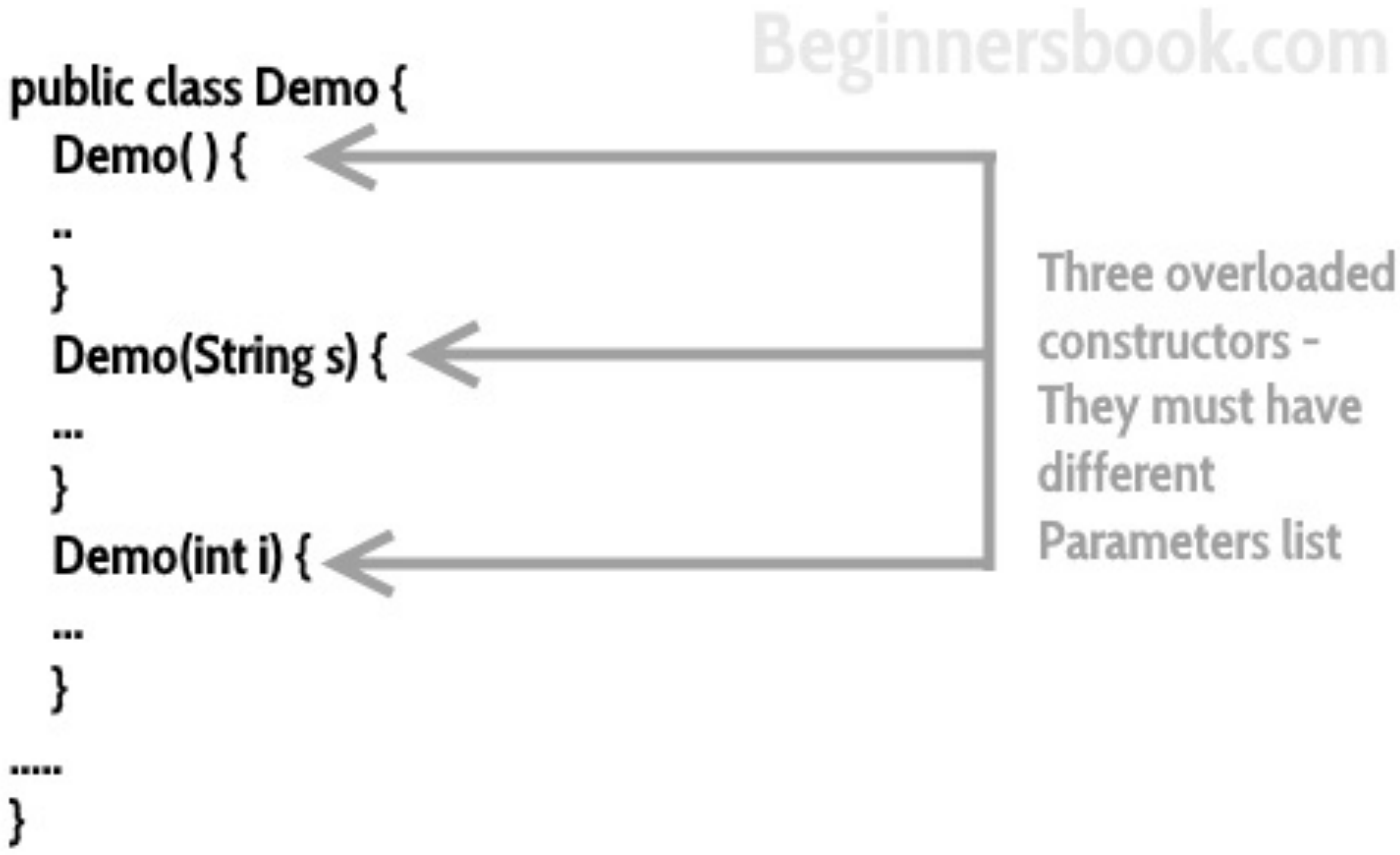
Syntax

Constructors

- Classes should define one or more methods to create or construct instances of the class
- Their name is the same as the class name
  - note deviation from convention that methods begin with lower case
- Constructors are differentiated by the number and types of their arguments
  - An example of overloading
- If you don't define a constructor, a default one will be created.
- Constructors automatically invoke the zero argument constructor of their superclass when they begin (note thatthis yields a recursive process!)

Syntax

Constructors



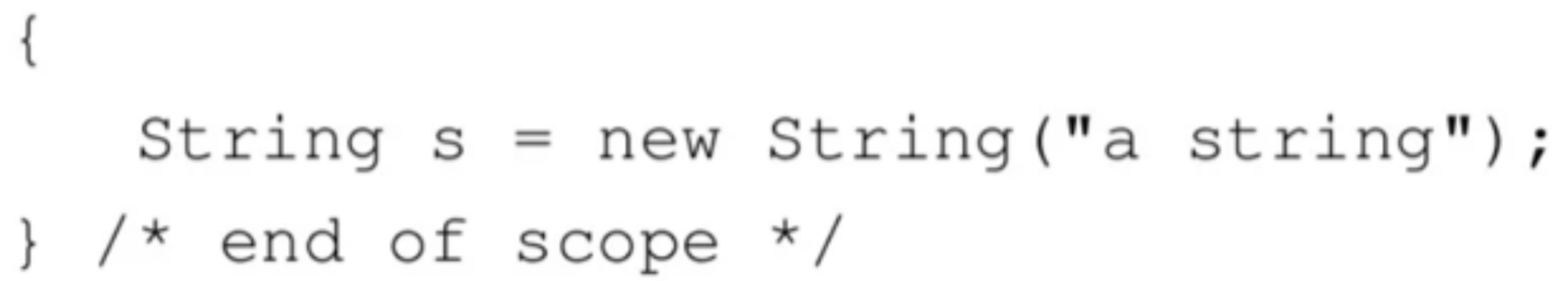
```
public class Demo {
   Demo( ) {
   ..
   }
   Demo(String s) {
   ...
   }
   Demo(int i) {
   ...
   }
.....
}
```

Three overloaded constructors - They must have different Parameters list

Syntax

Scope Of Objects

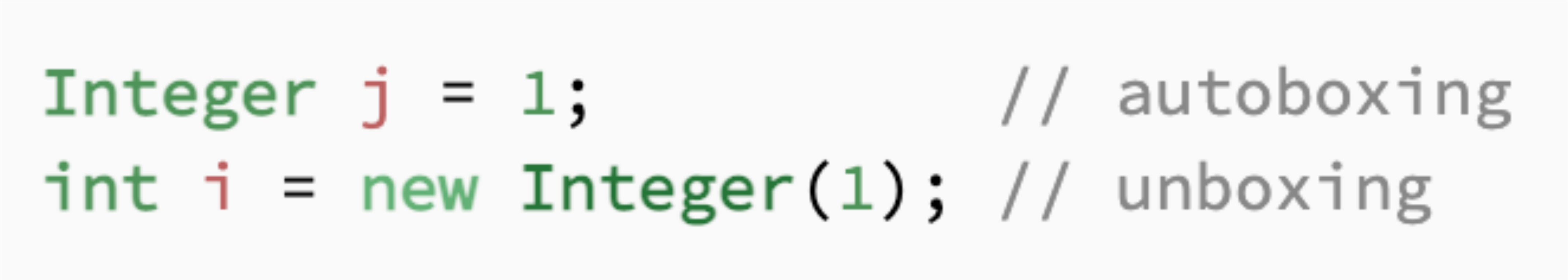
- objects are instances of classes, which also determine their types
- Java objects don't have the same lifetimes as primitives
- When you create a Java object using **new**, it hangs around past the end of the scope.
- Here, the scope of name ***s*** is delimited by the {}s but the String object hangs around until GC'd



```
{
  String s = new String("a string");
} /* end of scope */
```

Syntax
Types

- Primitive types
- Reference types / Objects
- Every primitive type corresponds to a reference type
- Under the hood, Java performs a conversion between the primitive and reference types:

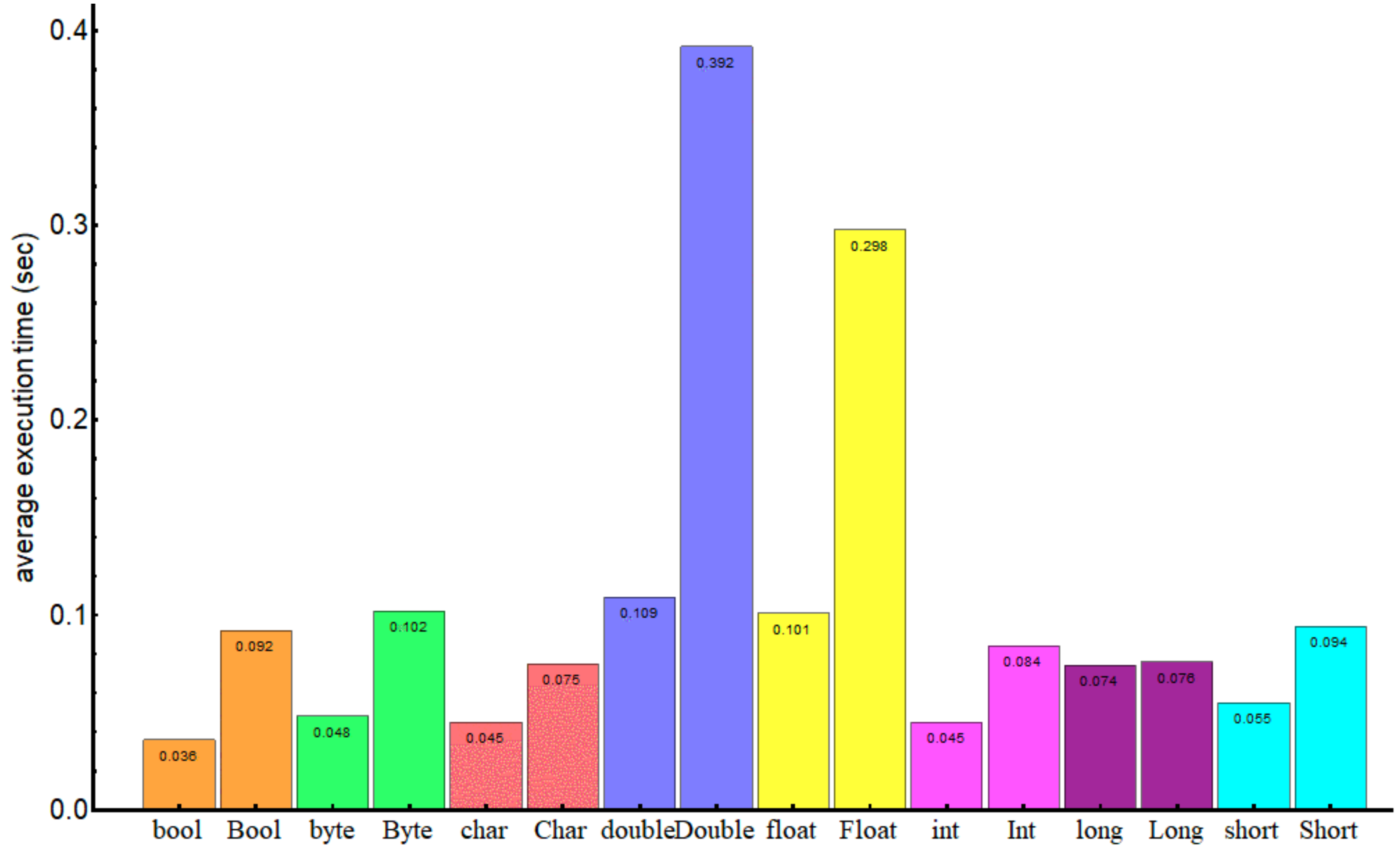


```
Integer j = 1;             // autoboxing
int i = new Integer(1); // unboxing
```

Syntax

Primitive types vs Reference Types

- boolean – Boolean
- byte – Byte
- short, char – Short, Character
- int, float – Integer, Float
- long, double – Long, Double



- Memory footprint
- Nullability
- Default values
  - primitive types are *0* (in the corresponding representation)
  - Reference types are ‚null‘

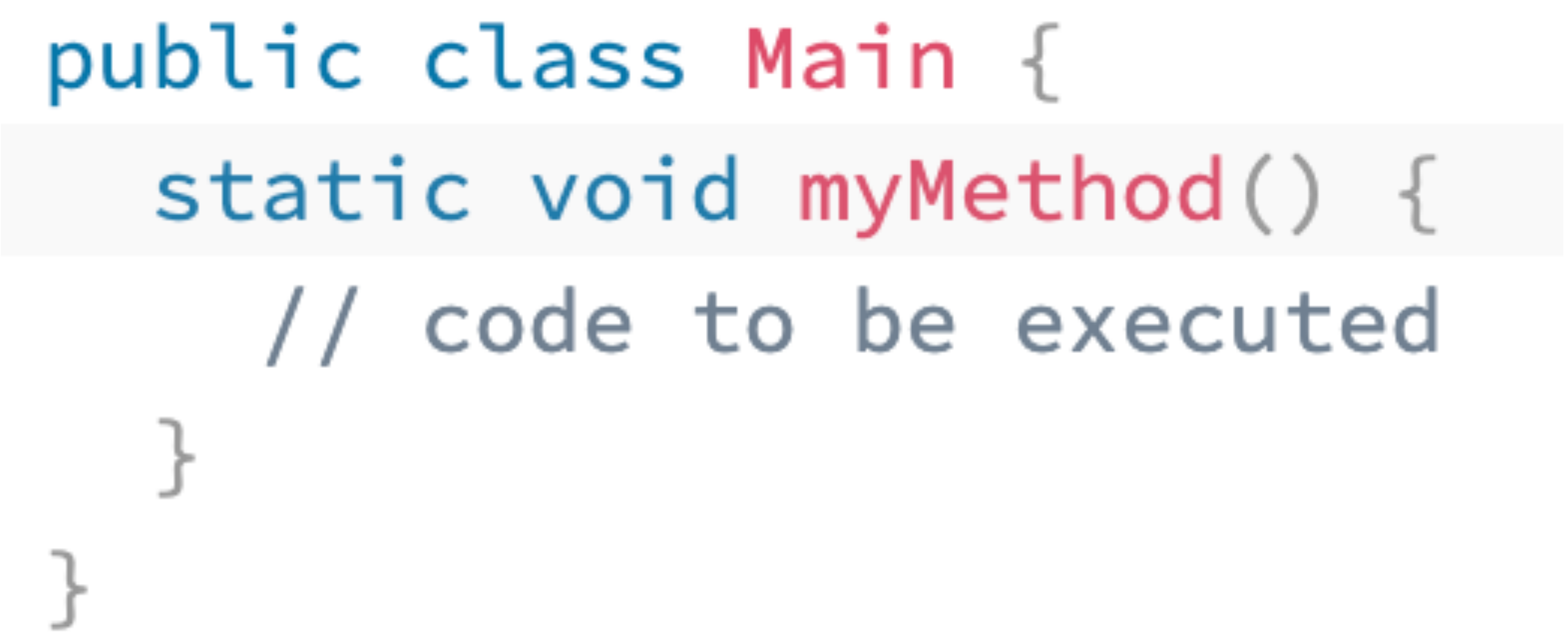
Syntax

Methods

- A method must be declared within a class.
- is defined with the name of the method, followed by parentheses **()**
- Java provides some pre-defined methods, such as `System.out.println()`
  - you can also create your own methods to perform certain actions

Syntax

Methods



```java
public class Main {
  static void myMethod() {
    // code to be executed
  }
}
```

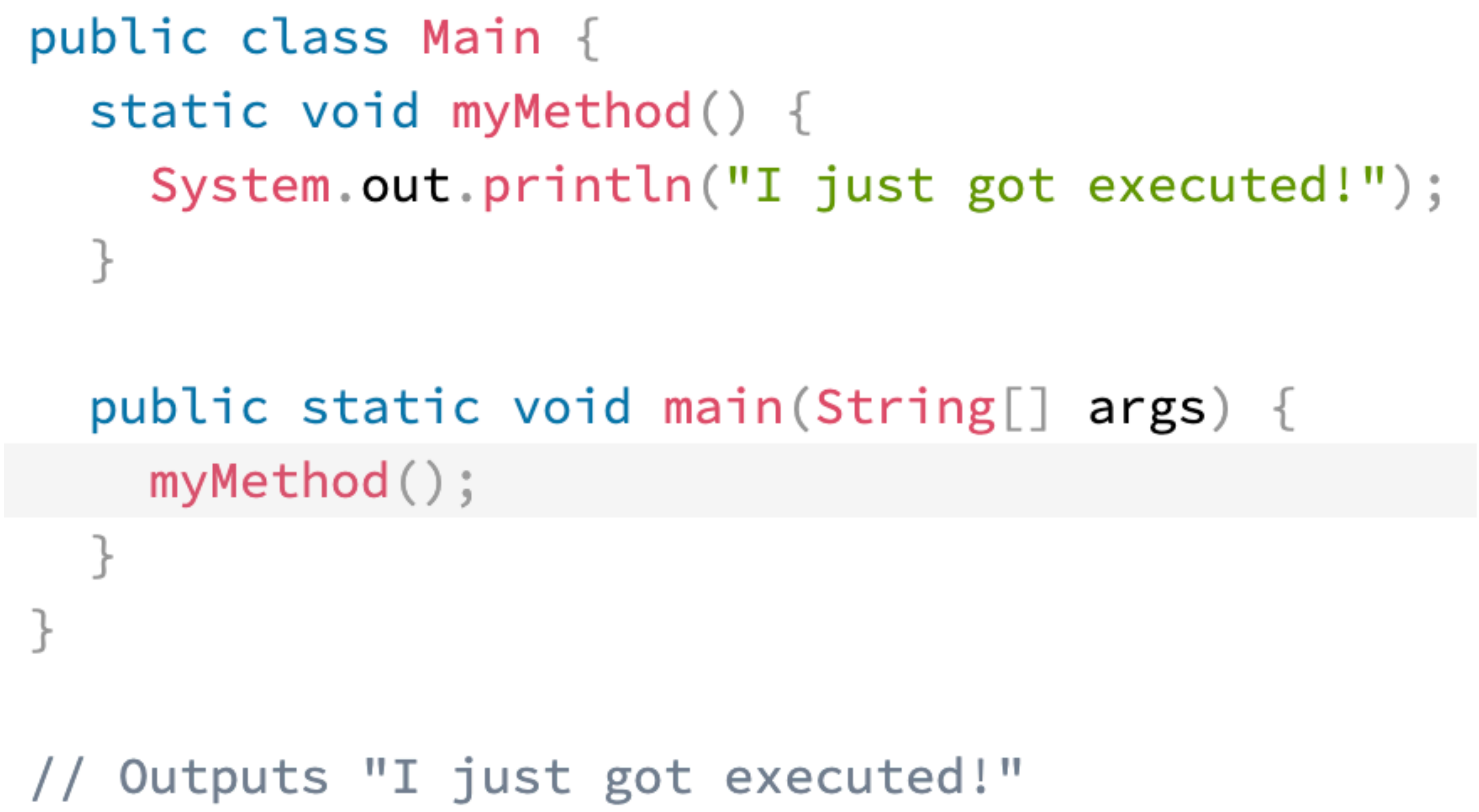
- `myMethod()` is the name of the method

- `static` means that the method belongs to the Main class and not an object of the Main class. You will learn more about objects and how to access methods through objects later

- `void` means that this method does not have a return value. You will learn more about return values later in this chapter

Syntax

Calling a Method

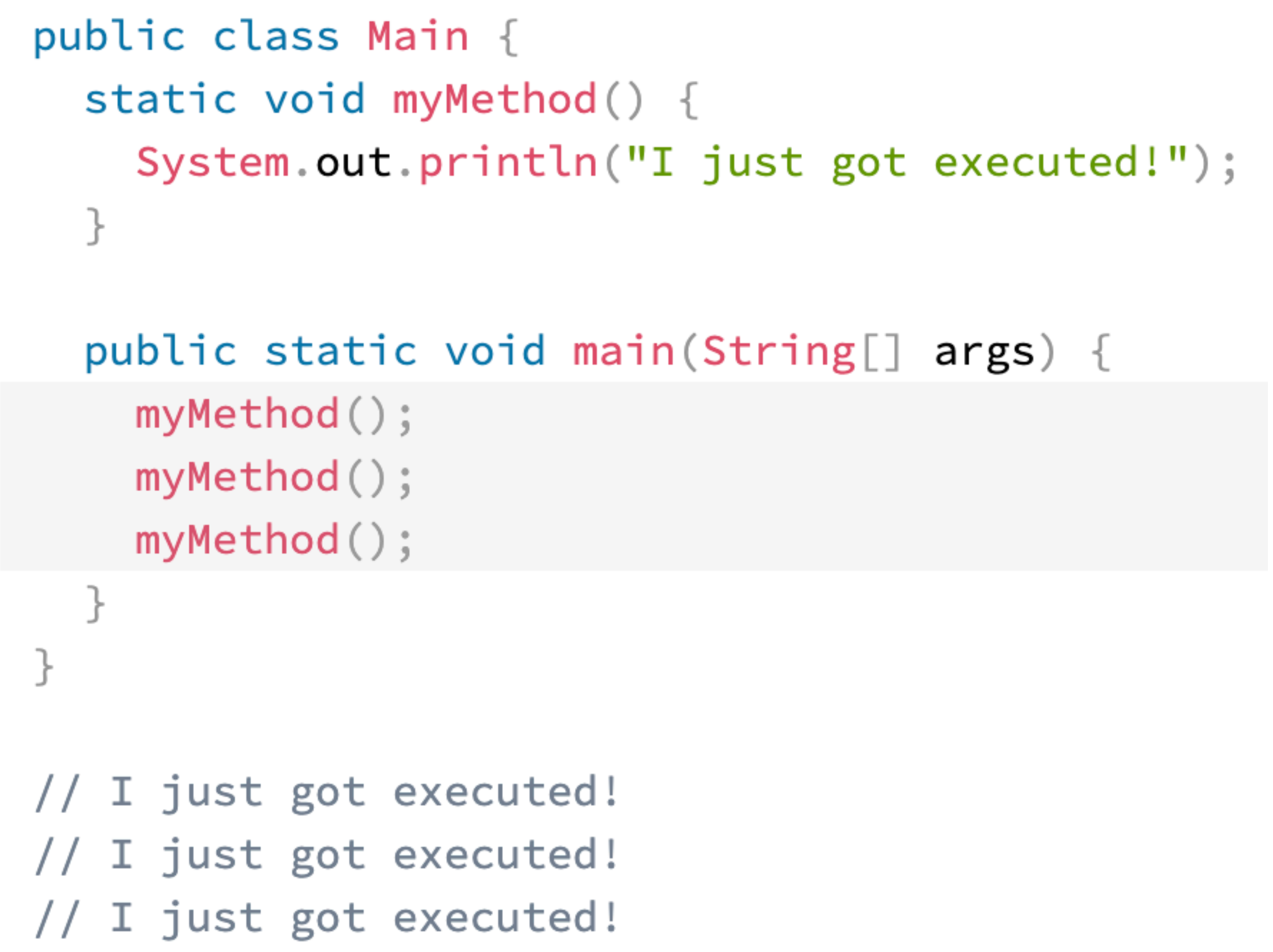


```
public class Main {
  static void myMethod() {
    System.out.println("I just got executed!");
  }

  public static void main(String[] args) {
    myMethod();
  }
}

// Outputs "I just got executed!"
```

- To call a method in Java, write the method's name followed by two parentheses **()** and a semicolon**;**

Syntax

Calling a Method



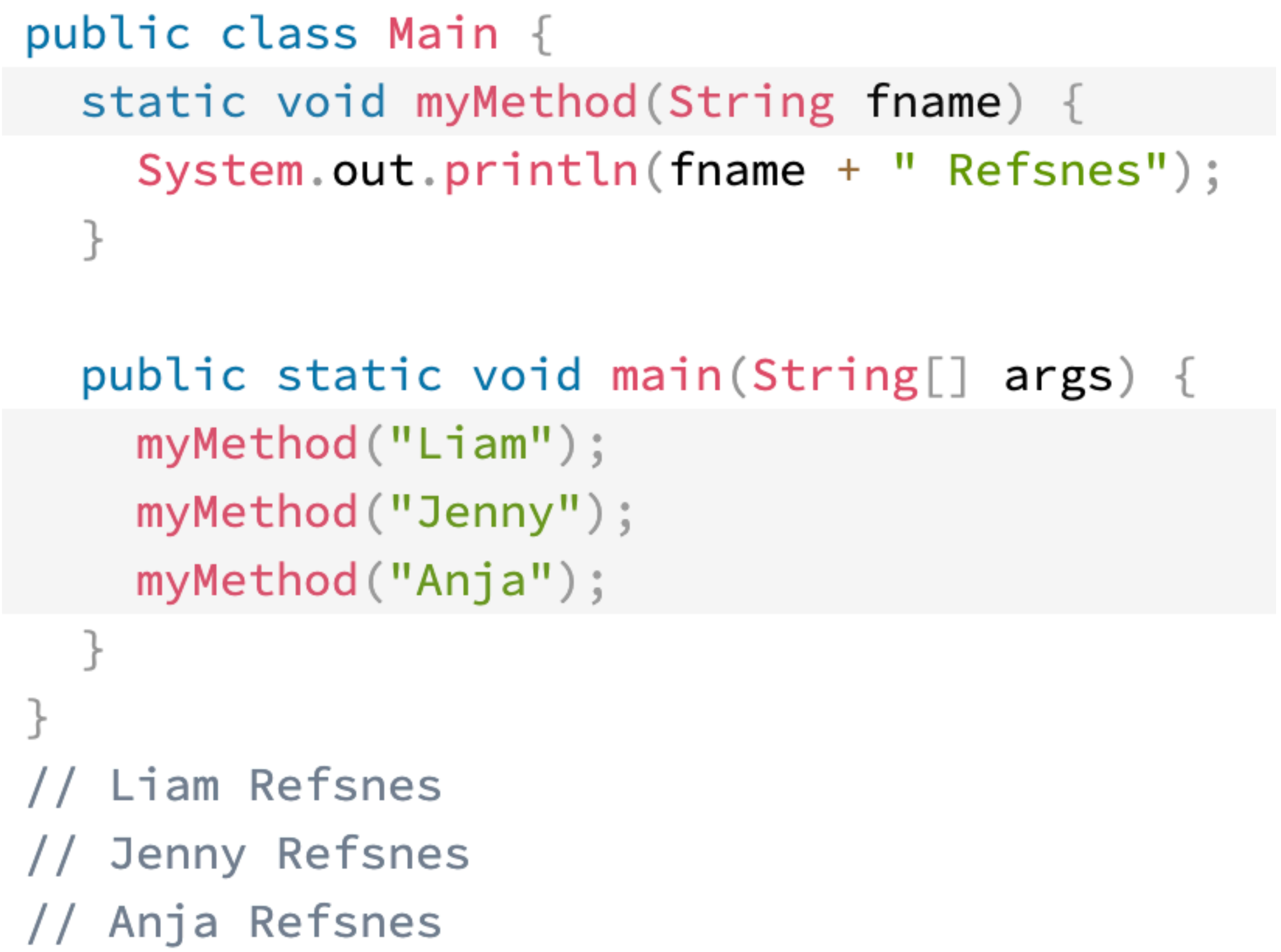
```
public class Main {
  static void myMethod() {
    System.out.println("I just got executed!");
  }

  public static void main(String[] args) {
    myMethod();
    myMethod();
    myMethod();
  }
}

// I just got executed!
// I just got executed!
// I just got executed!
```

- A method can also be called multiple times

Syntax

Parameters and Arguments

- Information can be passed to methods as parameter. Parameters act as variables inside the method.
- Parameters are specified after the method name, inside the parentheses.
- You can add as many parameters as you want, just separate them with a comma.

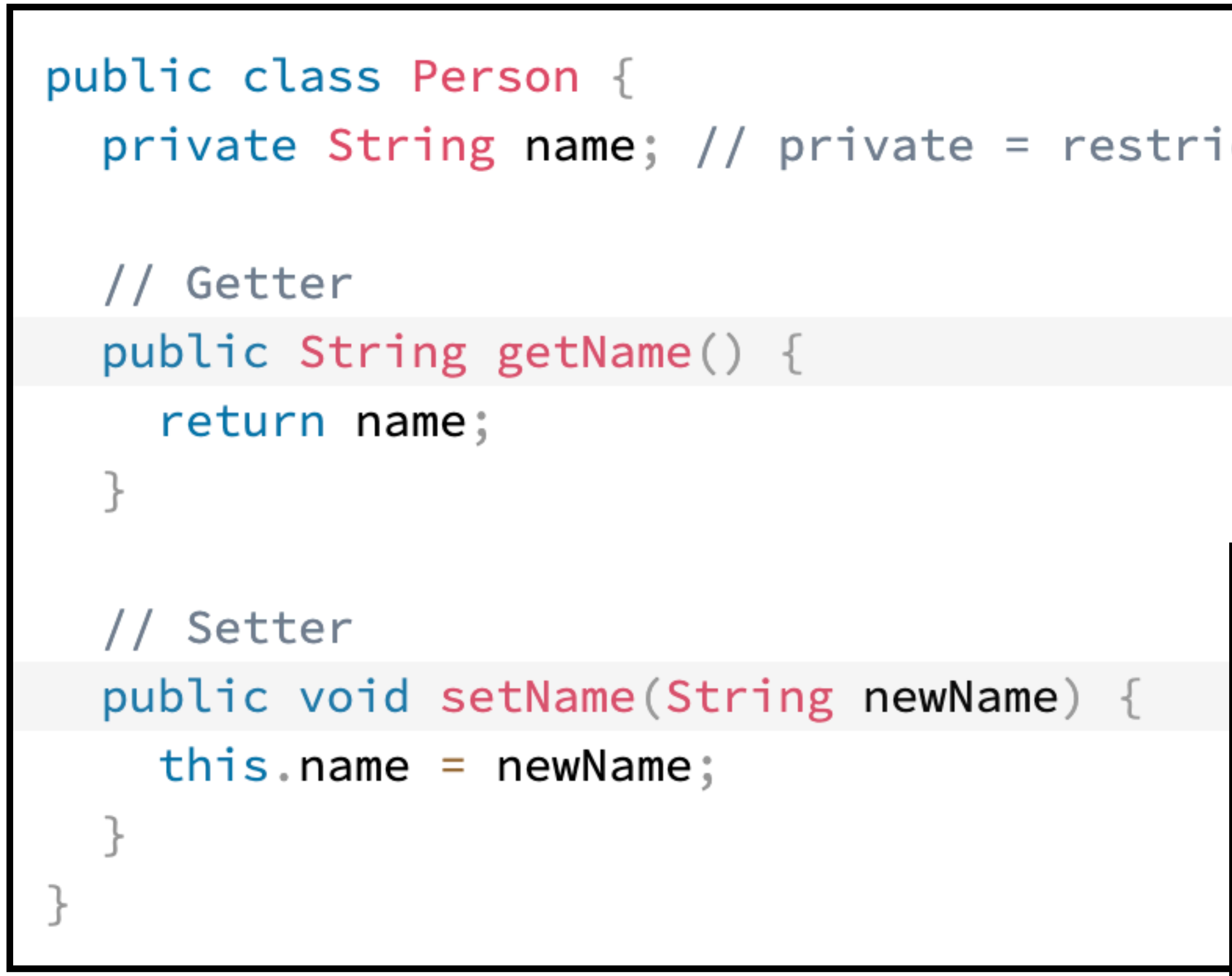


```
public class Main {
  static void myMethod(String fname) {
    System.out.println(fname + " Refsnes");
  }

  public static void main(String[] args) {
    myMethod("Liam");
    myMethod("Jenny");
    myMethod("Anja");
  }
}
// Liam Refsnes
// Jenny Refsnes
// Anja Refsnes
```

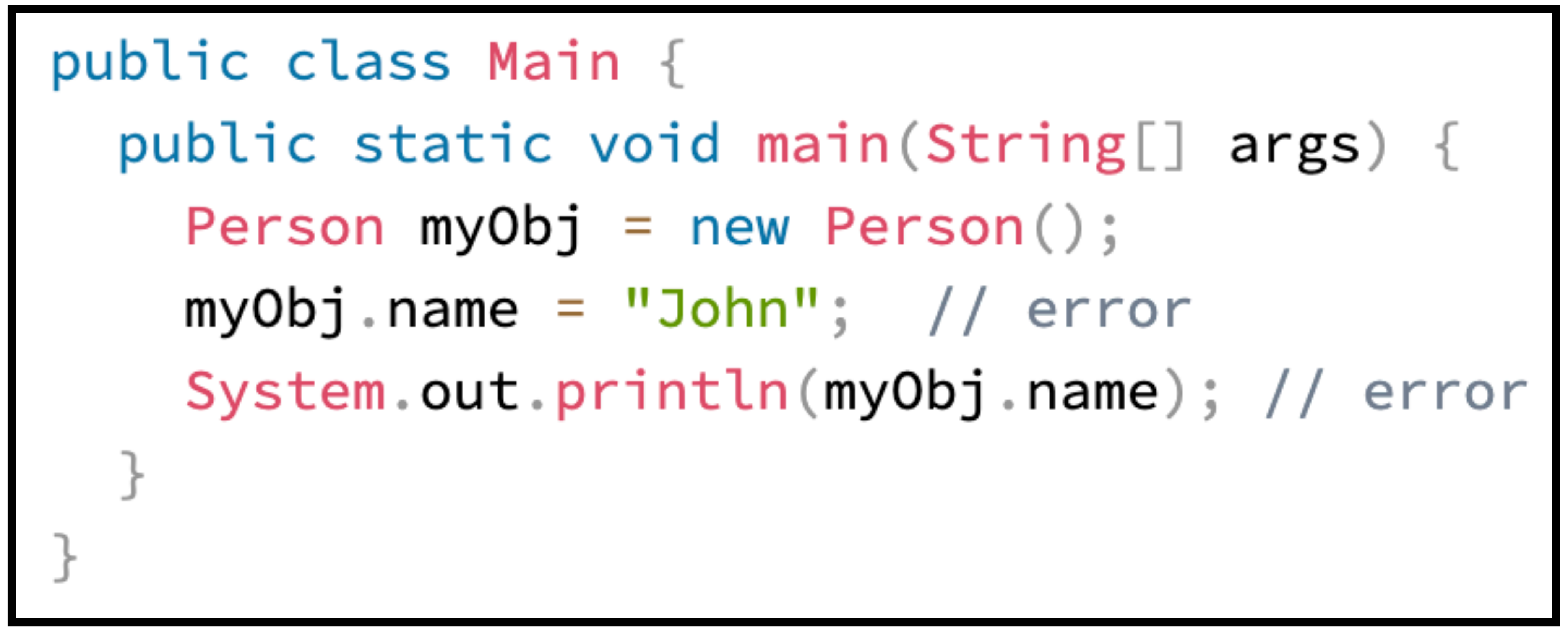
Getters and setters

- A getter is a method that extracts information from an instance.
  - One benefit: you can include additional computation in a getter.
- A setter is a method that inserts information into an instance (also known as mutators).
  - A setter method can check the validity of the new value (e.g., between 1 and 7) or trigger a side effect (e.g., update a display)
- Getters and setters can be used even without underlying matching variables
- Considered good 00 practice
- Essential to javabeans
- Convention: for variable fooBar of type String, define
  - getFooBar() { … }
  - setFooBar(String x) { … }

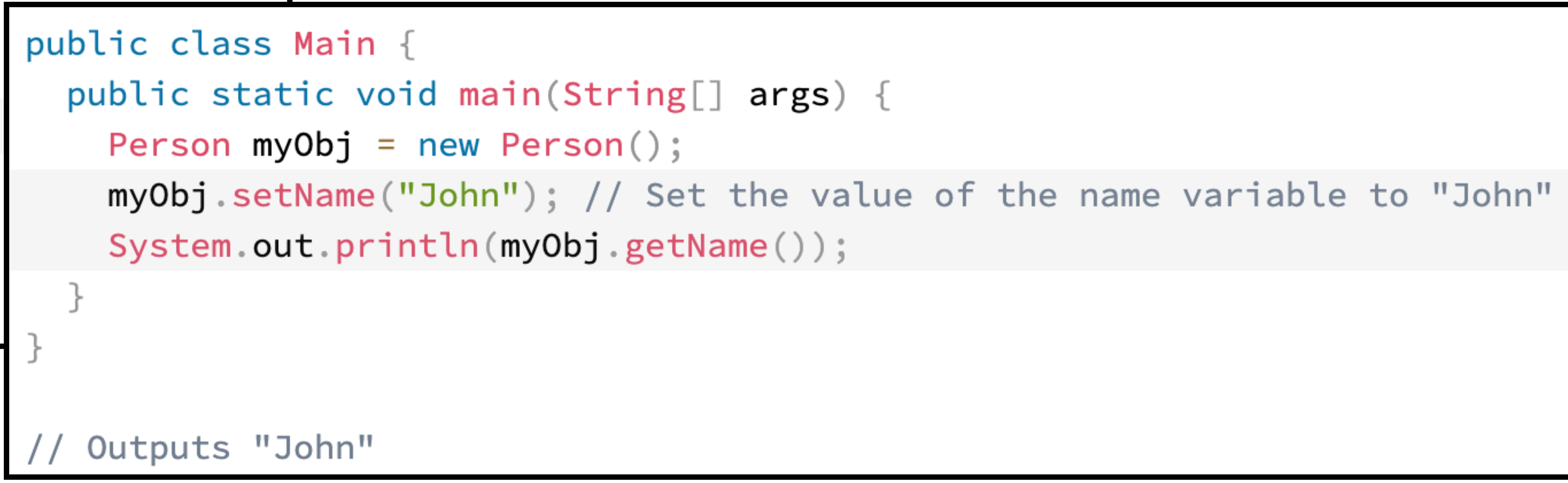
Getters and setters



```java
public class Person {
  private String name; // private = restricted access

  // Getter
  public String getName() {
    return name;
  }

  // Setter
  public void setName(String newName) {
    this.name = newName;
  }
}
```



```java
public class Main {
  public static void main(String[] args) {
    Person myObj = new Person();
    myObj.name = "John";  // error
    System.out.println(myObj.name); // error
  }
}
```



```java
public class Main {
  public static void main(String[] args) {
    Person myObj = new Person();
    myObj.setName("John"); // Set the value of the name variable to "John"
    System.out.println(myObj.getName());
  }
}

// Outputs "John"
```

Abstraction

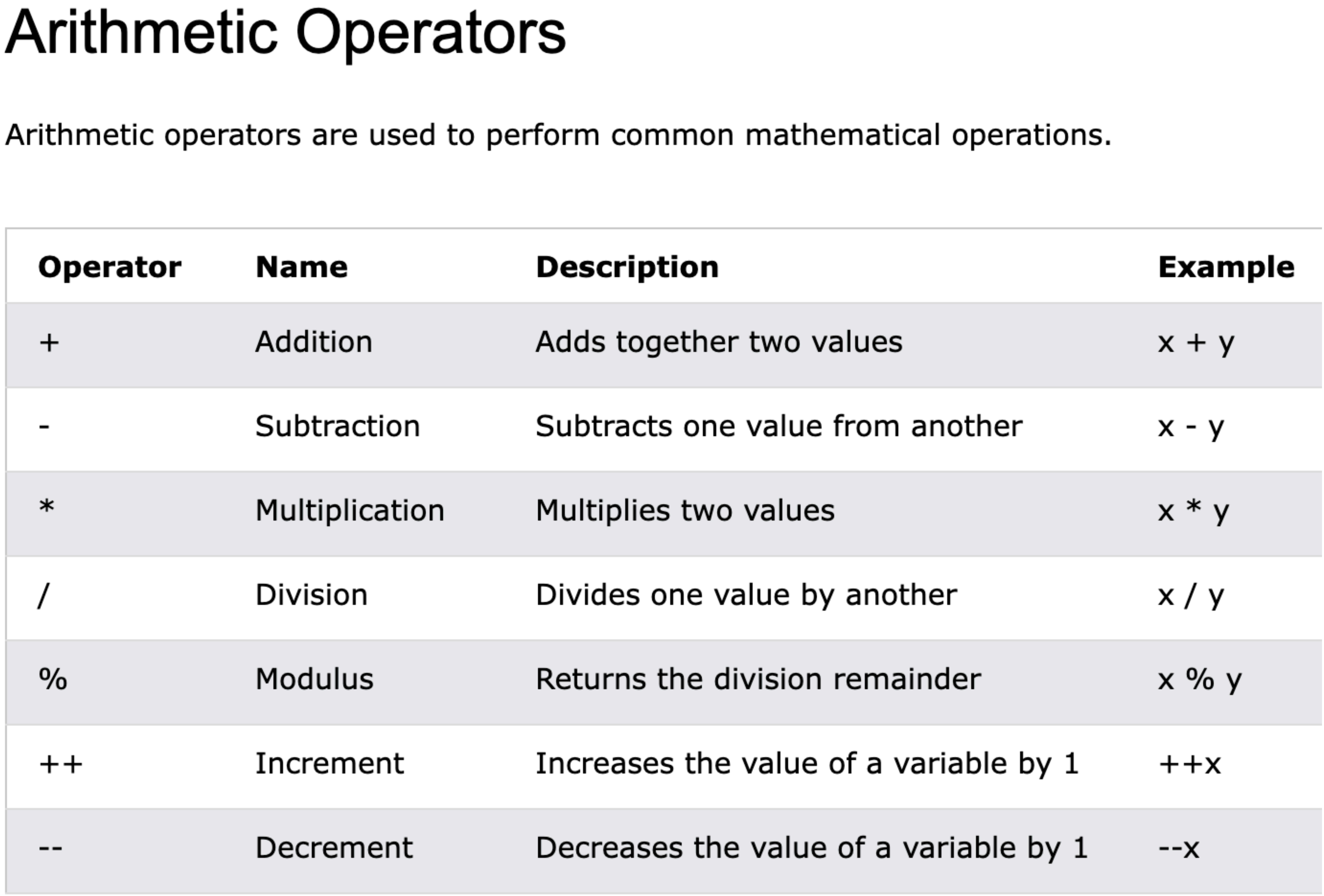
Extending a class

- Class hierarchies reflect subclass-superclass relations among classes
- One arranges classes in hierarchies:
    - A class inherits instance variables and instance methods from all of its superclasses.
    - You can specify only ONE superclass for any class.
- When a subclass-superclass chain contains multiple instance methods with the same signature (name, arity, and argument types), the one closest to the target instance in the subclass-superclass chain is the one executed.
    - All others are shadowed/overridden.
- Something like multiple inheritance can be done via interfaces (more on this later)

Operators

- Operators are used to perform operations on variables and values.
- Java divides the operators into the following groups:
  - Arithmetic operators
  - Assignment operators
  - Comparison operators
  - Logical operators

Operators

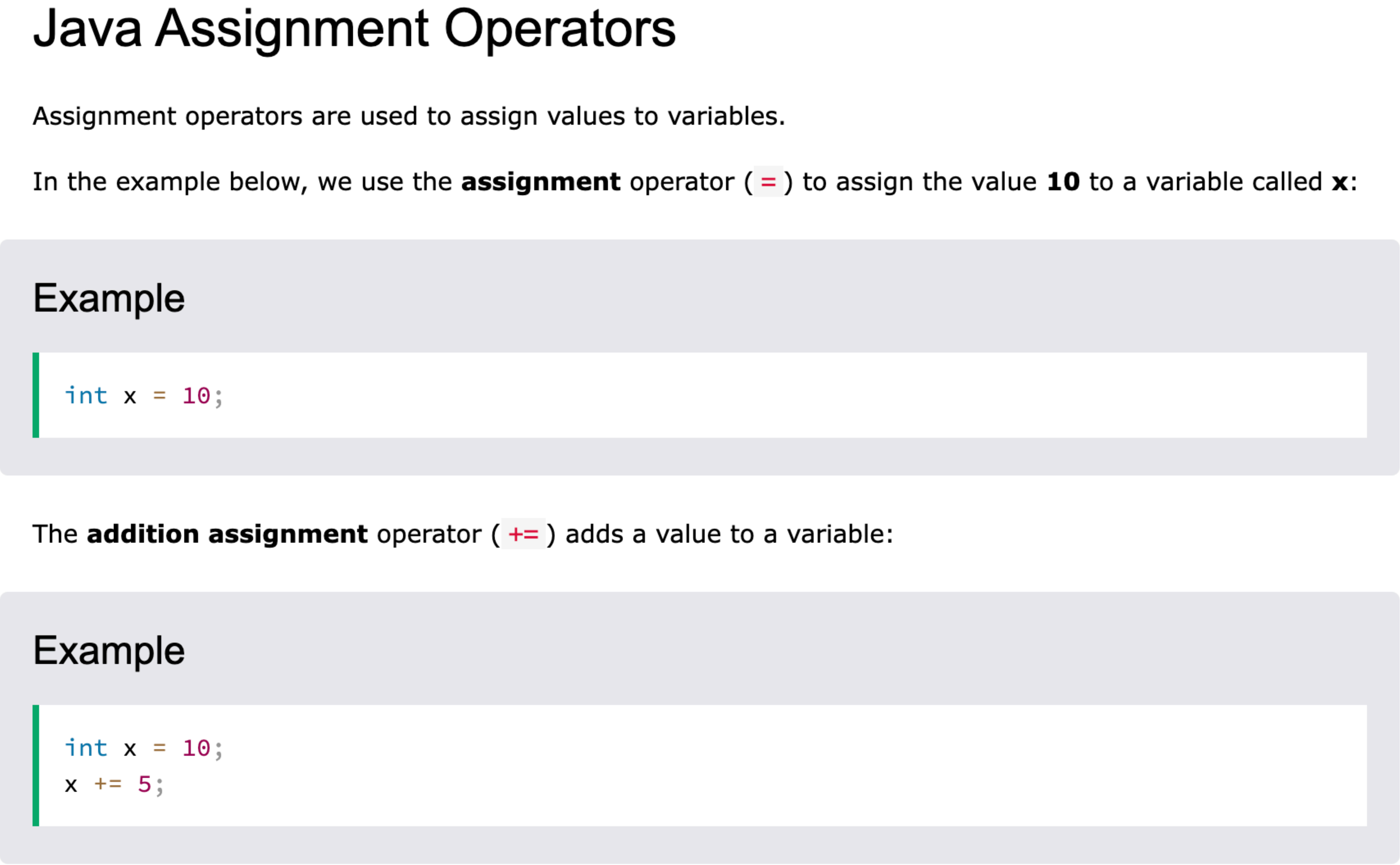


Arithmetic Operators

Arithmetic operators are used to perform common mathematical operations.

Operator	Name	Description	Example
+	Addition	Adds together two values	x + y
-	Subtraction	Subtracts one value from another	x - y
*	Multiplication	Multiplies two values	x * y
/	Division	Divides one value by another	x / y
%	Modulus	Returns the division remainder	x % y
++	Increment	Increases the value of a variable by 1	++x
--	Decrement	Decreases the value of a variable by 1	--x

Operators



Java Assignment Operators

Assignment operators are used to assign values to variables.

In the example below, we use the **assignment** operator (`=`) to assign the value **10** to a variable called **x**:

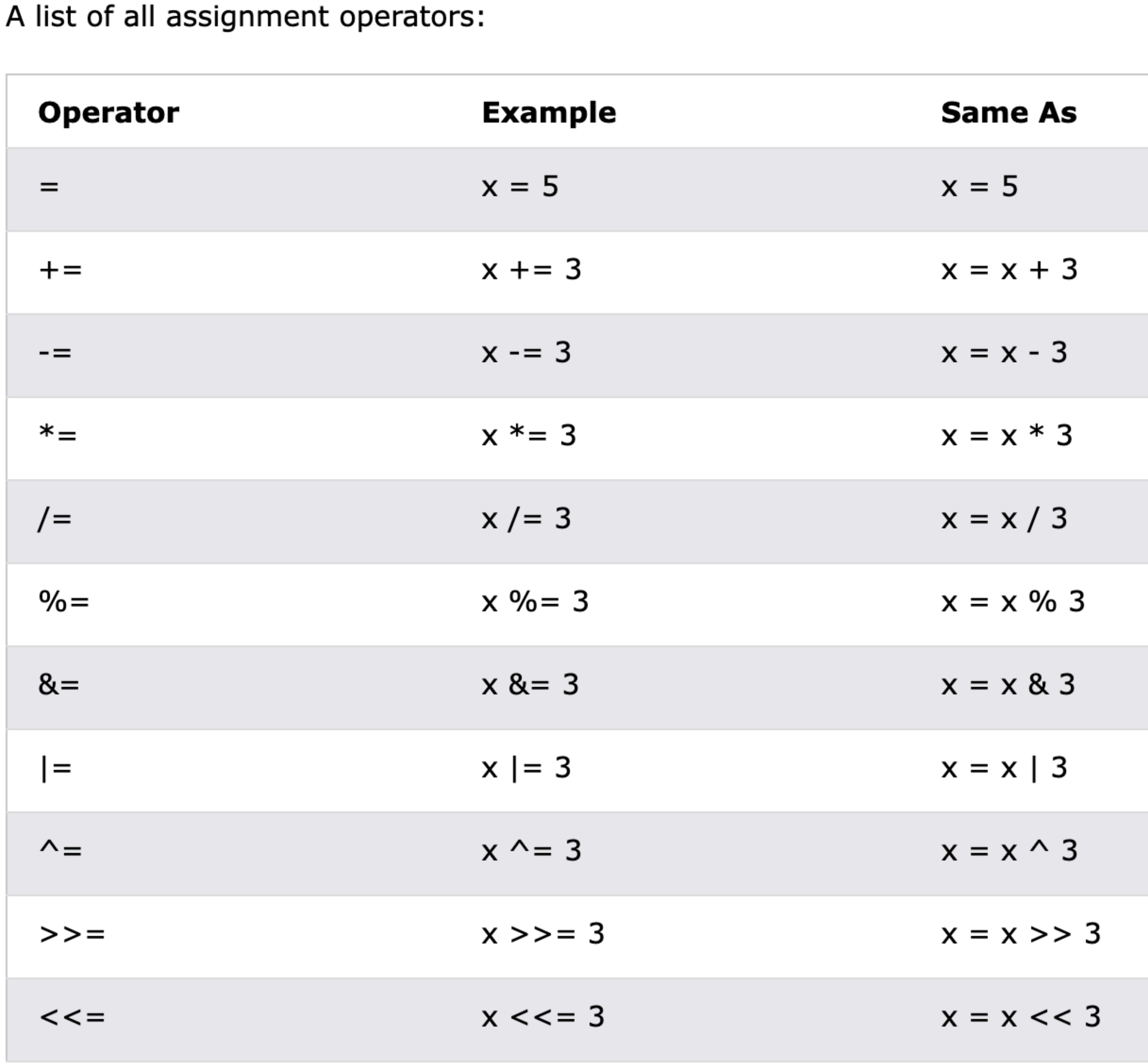
Example

```
int x = 10;
```

The **addition assignment** operator (`+=`) adds a value to a variable:

Example

```
int x = 10;
x += 5;
```

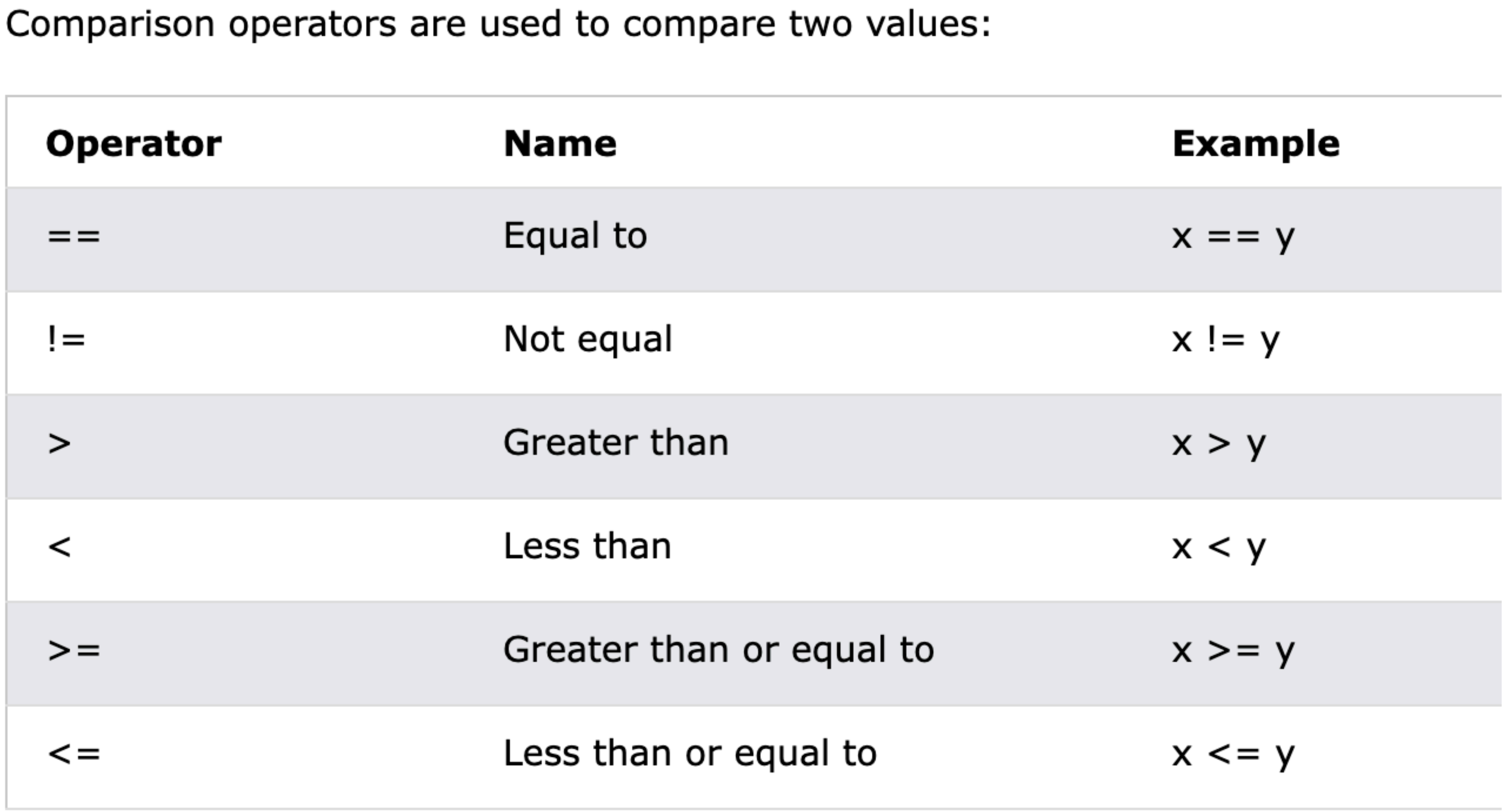


A list of all assignment operators:

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

Operators

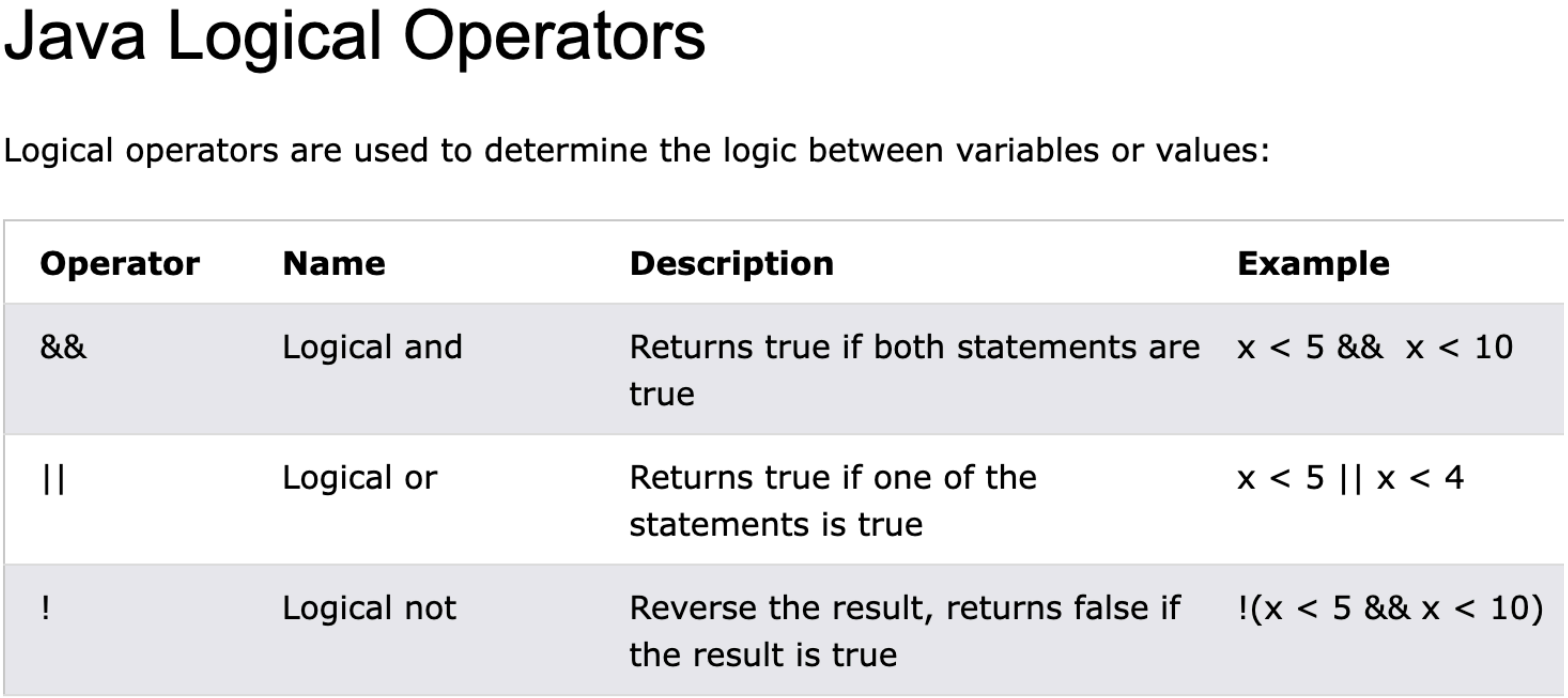
Java Comparison Operators



Comparison operators are used to compare two values:

Operator	Name	Example
==	Equal to	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

Operators



Java Logical Operators

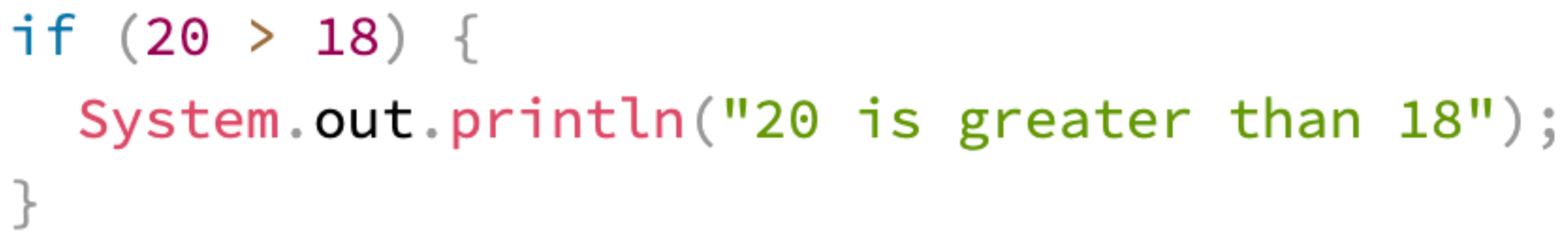
Logical operators are used to determine the logic between variables or values:

Operator	Name	Description	Example
&&	Logical and	Returns true if both statements are true	x < 5 && x < 10
||	Logical or	Returns true if one of the statements is true	x < 5 || x < 4
!	Logical not	Reverse the result, returns false if the result is true	!(x < 5 && x < 10)

Control Statements

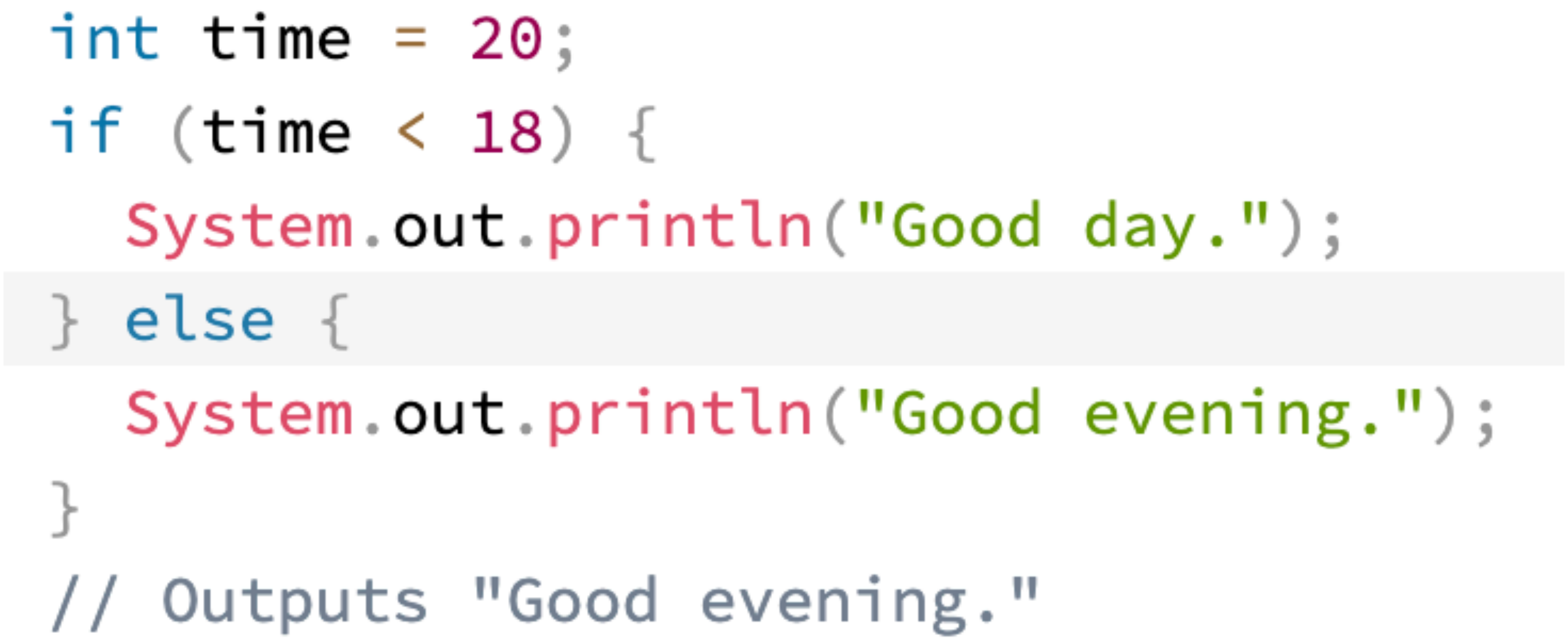
If-condition

- Use the `if` statement to specify a block of Java code to be executed if a condition is `true`.



```
if (20 > 18) {
  System.out.println("20 is greater than 18");
}
```

- Use the `else` statement to specify a block of code to be executed if the condition is `false`.

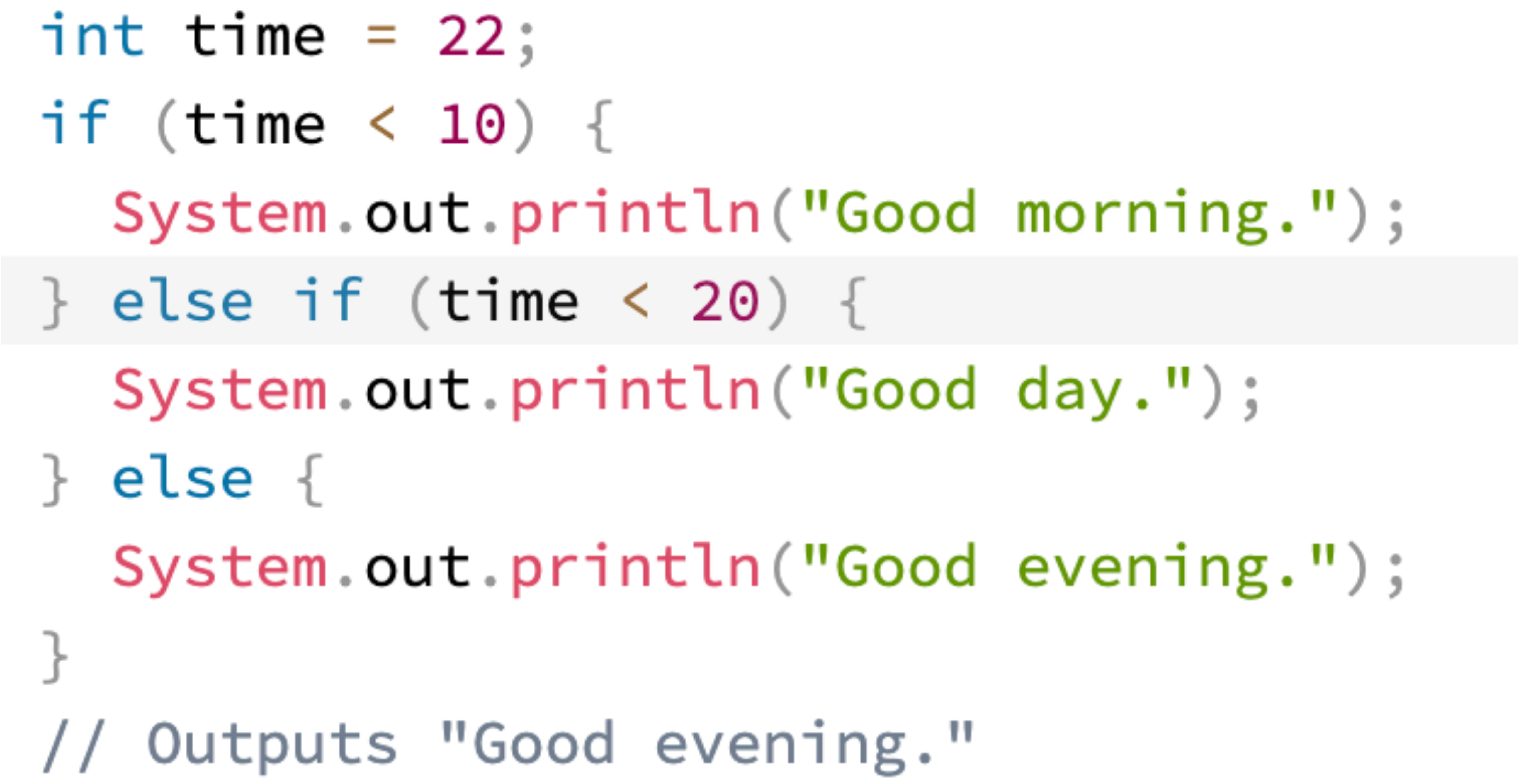


```
int time = 20;
if (time < 18) {
  System.out.println("Good day.");
} else {
  System.out.println("Good evening.");
}
// Outputs "Good evening."
```

Control Statements

If-condition

- Use the `else if` statement to specify a new condition if the first condition is `false`.

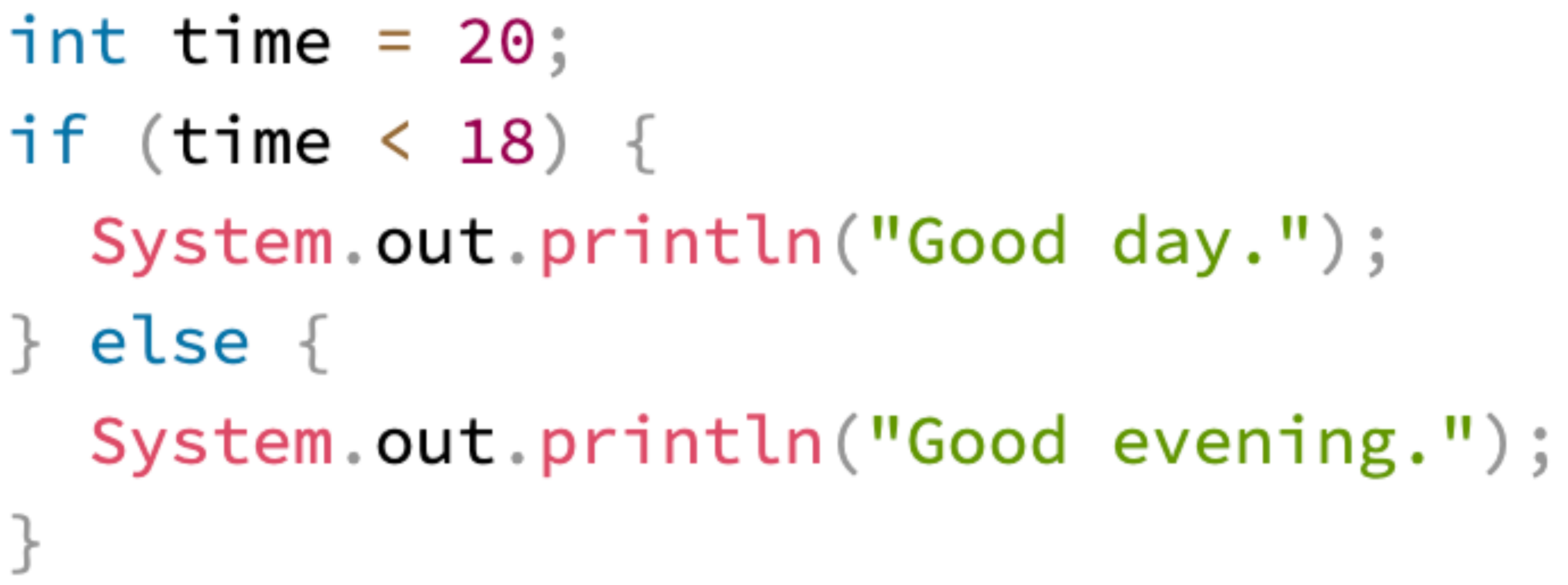


```java
int time = 22;
if (time < 10) {
  System.out.println("Good morning.");
} else if (time < 20) {
  System.out.println("Good day.");
} else {
  System.out.println("Good evening.");
}
// Outputs "Good evening."
```

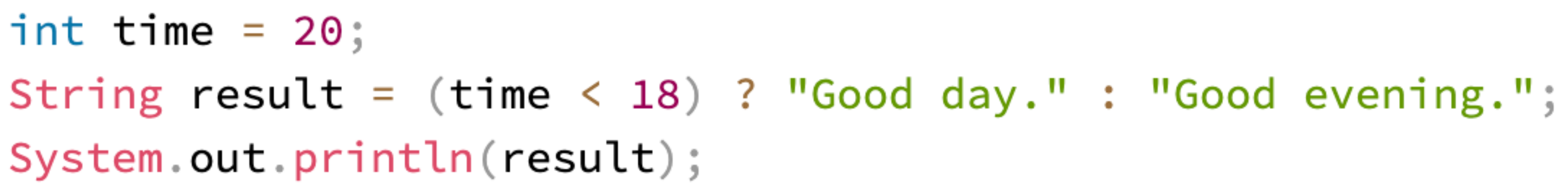
Control Statements

If-condition

- Short Hand If...Else (Ternary Operator)



```java
int time = 20;
if (time < 18) {
  System.out.println("Good day.");
} else {
  System.out.println("Good evening.");
}
```

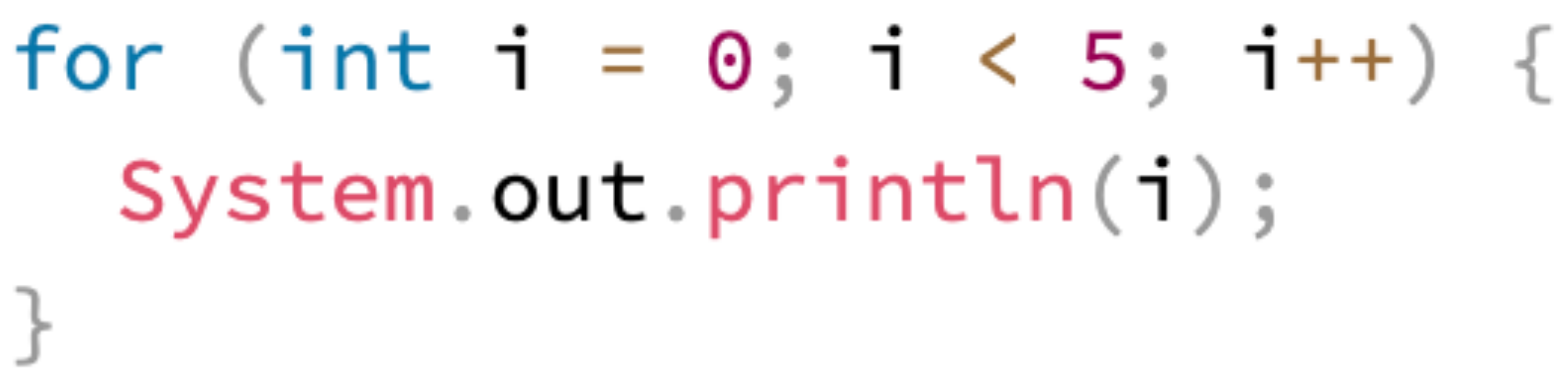


```java
int time = 20;
String result = (time < 18) ? "Good day." : "Good evening.";
System.out.println(result);
```

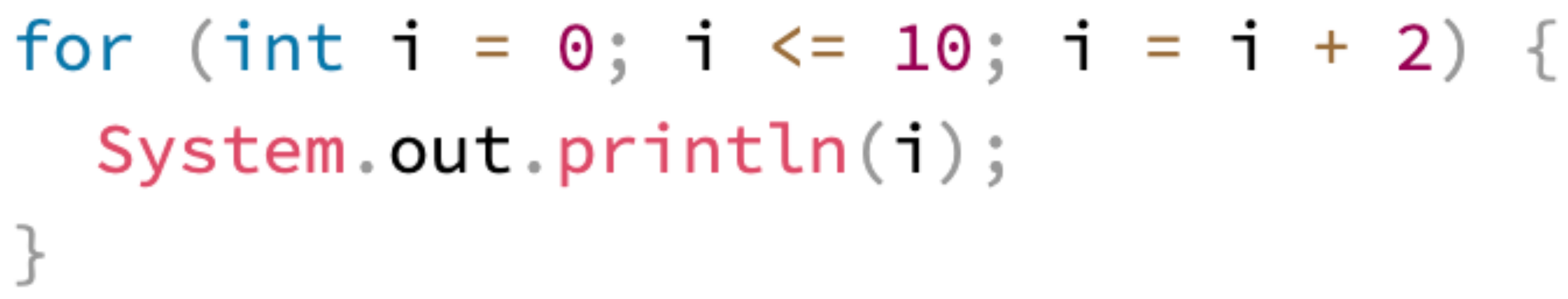
Loops

For

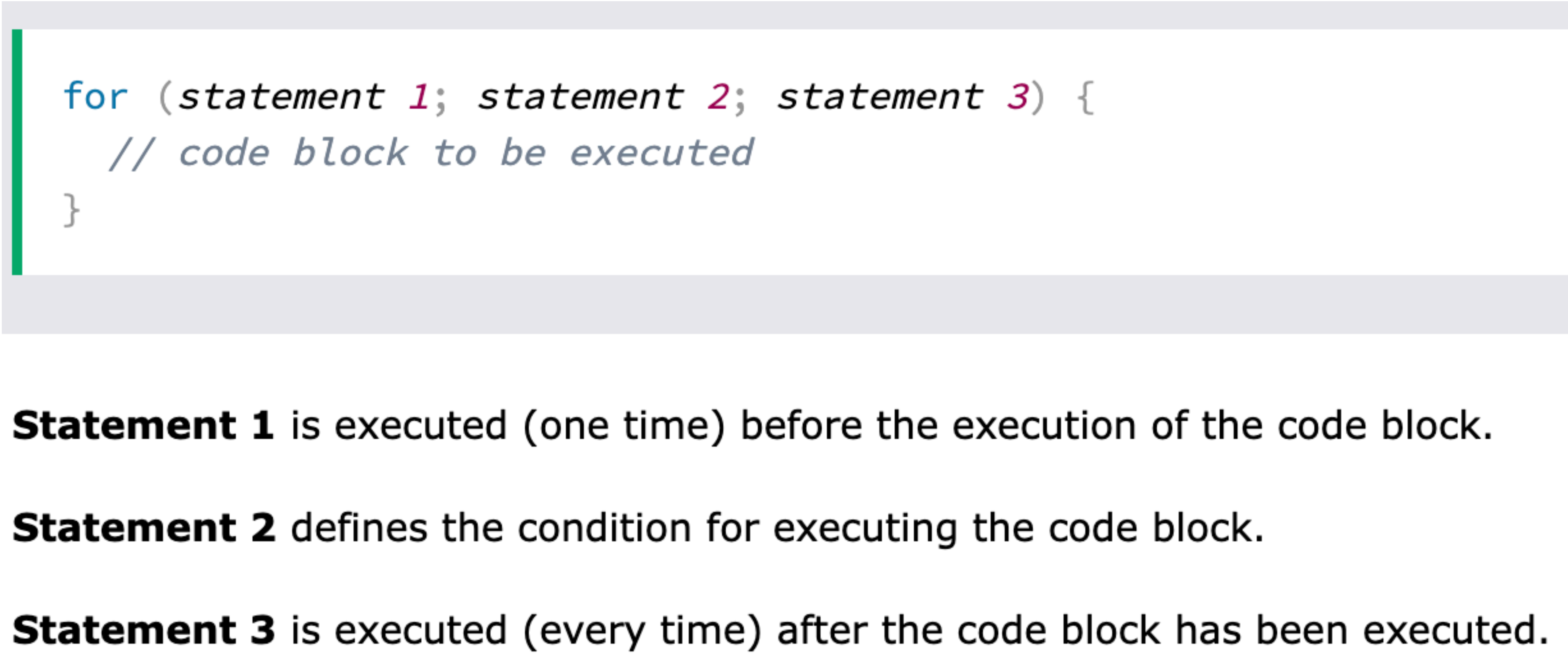
- When you know exactly how many times you want to loop through a block of code, use the `for` loop instead of a `while` loop:



```java
for (int i = 0; i < 5; i++) {
  System.out.println(i);
}
```



```java
for (int i = 0; i <= 10; i = i + 2) {
  System.out.println(i);
}
```



```java
for (statement 1; statement 2; statement 3) {
  // code block to be executed
}
```

Statement 1 is executed (one time) before the execution of the code block.

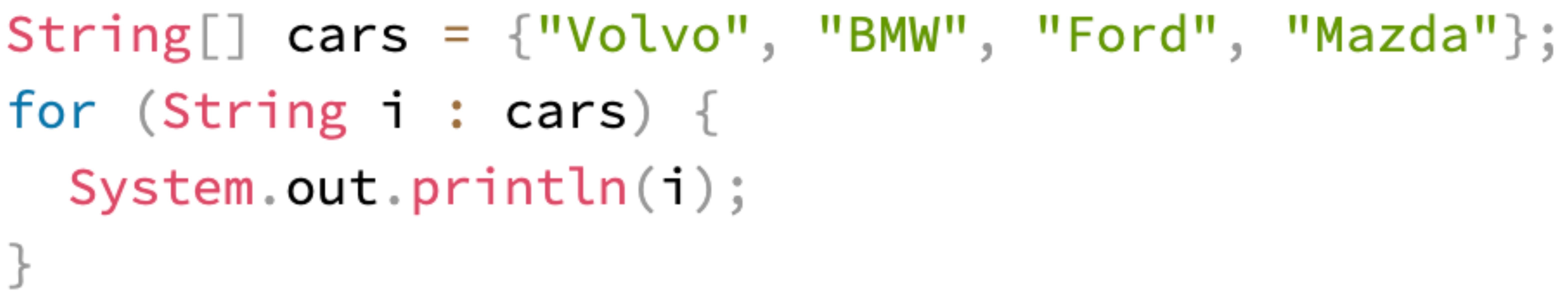
Statement 2 defines the condition for executing the code block.

Statement 3 is executed (every time) after the code block has been executed.

Loops

For-Each

- There is also a "**for-each**" loop, which is used exclusively to loop through elements in an **array**:

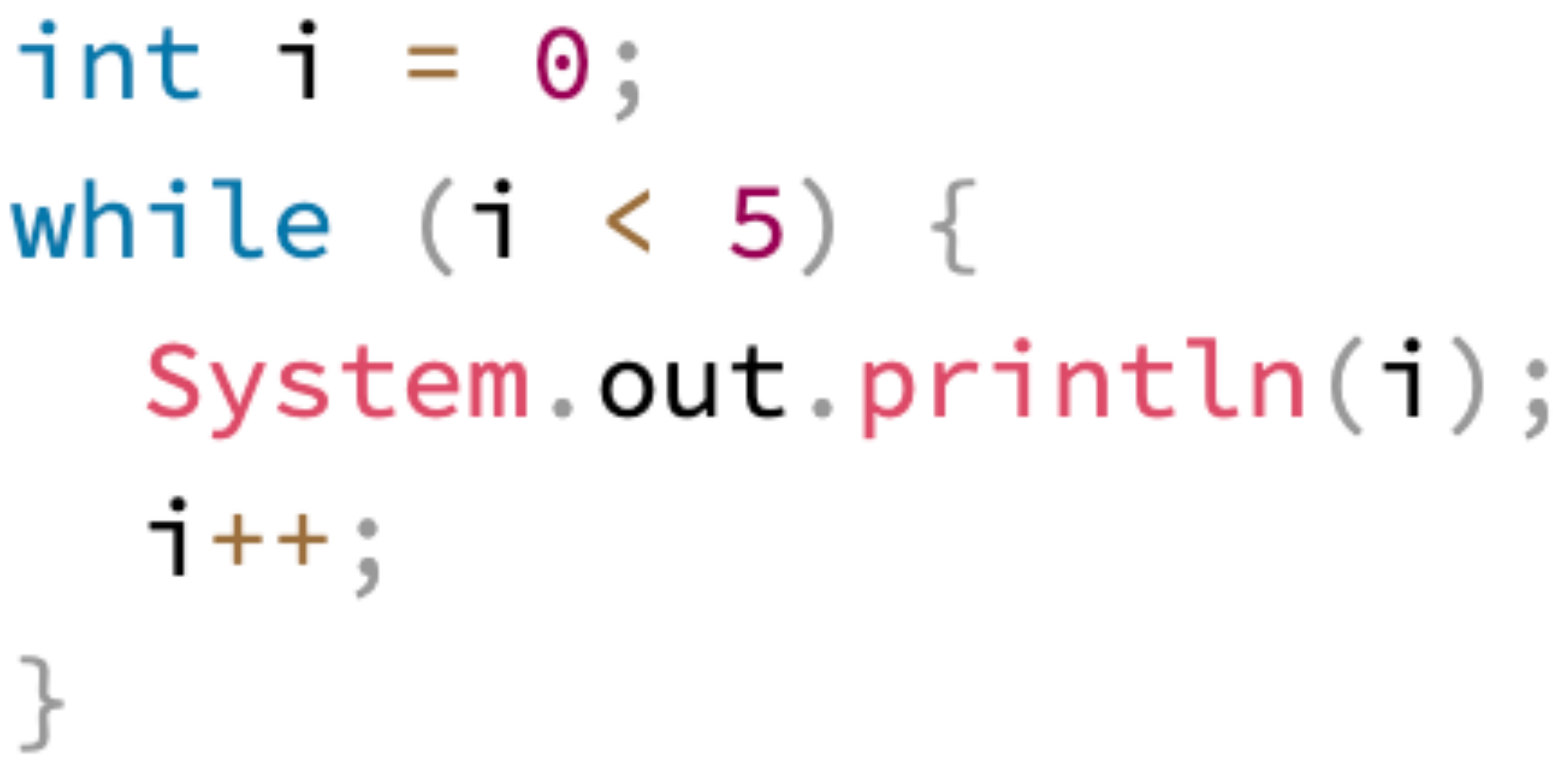


```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
for (String i : cars) {
  System.out.println(i);
}
```

Loops

While

- Loops can execute a block of code as long as a specified condition is reached.
- Loops are handy because they save time, reduce errors, and they make code more readable.
- The `while` loop loops through a block of code as long as a specified condition is `true`:

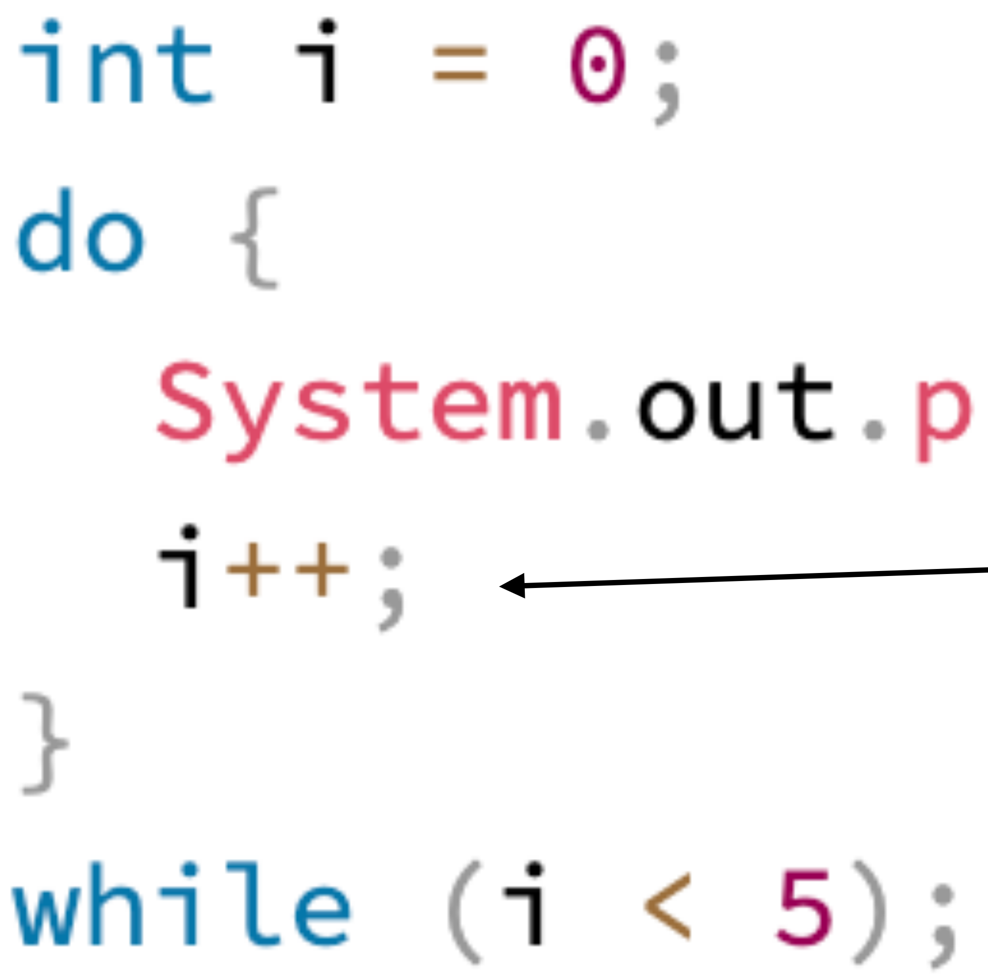


```
int i = 0;
while (i < 5) {
  System.out.println(i);
  i++;
}
```

Loops

Do While

- The `do/while` loop is a variant of the `while` loop. This loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.



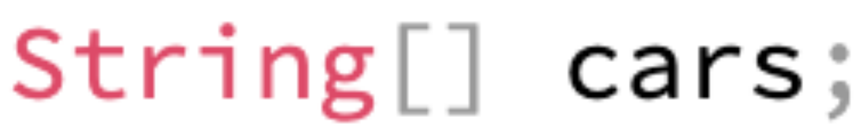
```
int i = 0;
do {
  System.out.println(i);
  i++;
}
while (i < 5);
```

Do not forget to increase the variable used in the condition, otherwise the loop will never end!

Arrays

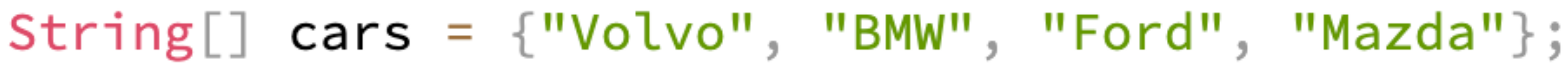
Creation

- Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.
- To declare an array, define the variable type with **square brackets**:



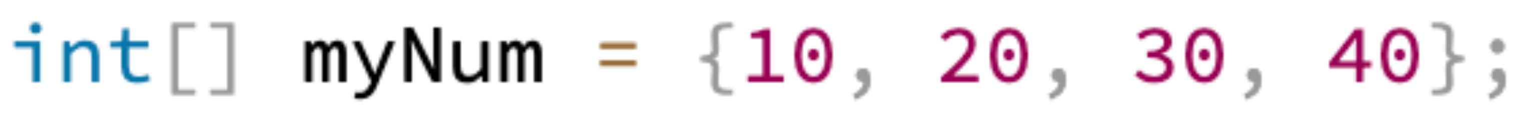
```
String[] cars;
```

- use an array literal to add values:



```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
```

- To create an array of integers, you could write:

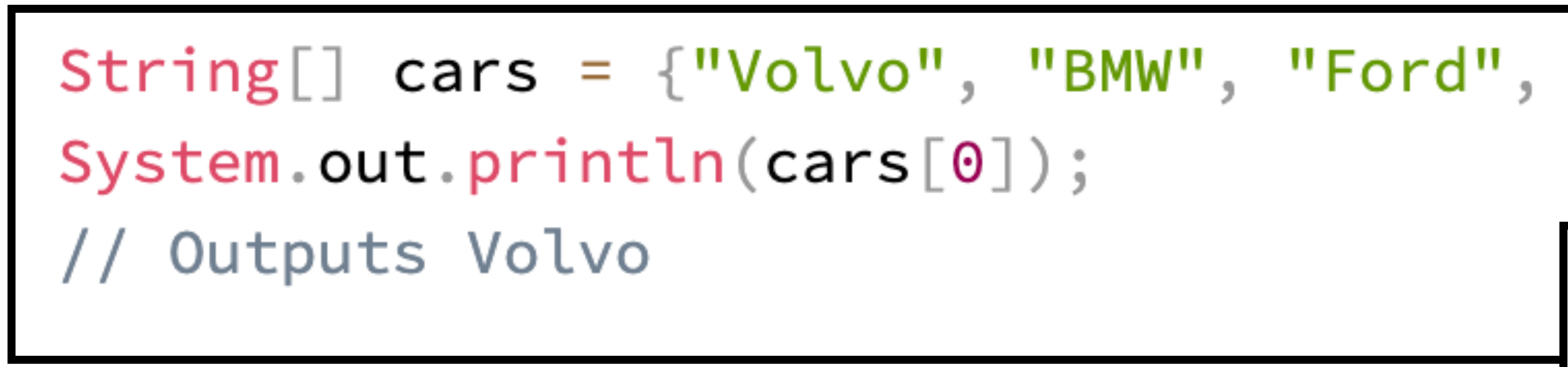


```
int[] myNum = {10, 20, 30, 40};
```

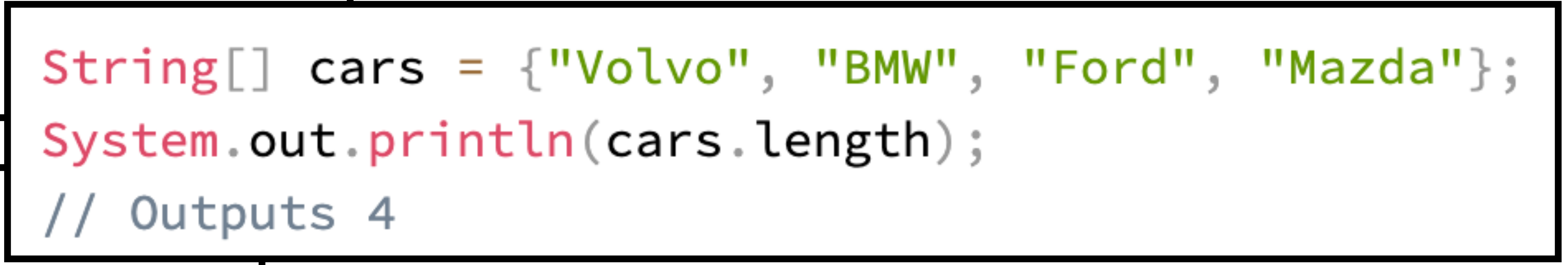
Arrays

Accessing

- You access an array element by referring to the index number.
- This statement accesses the value of the first element in cars:

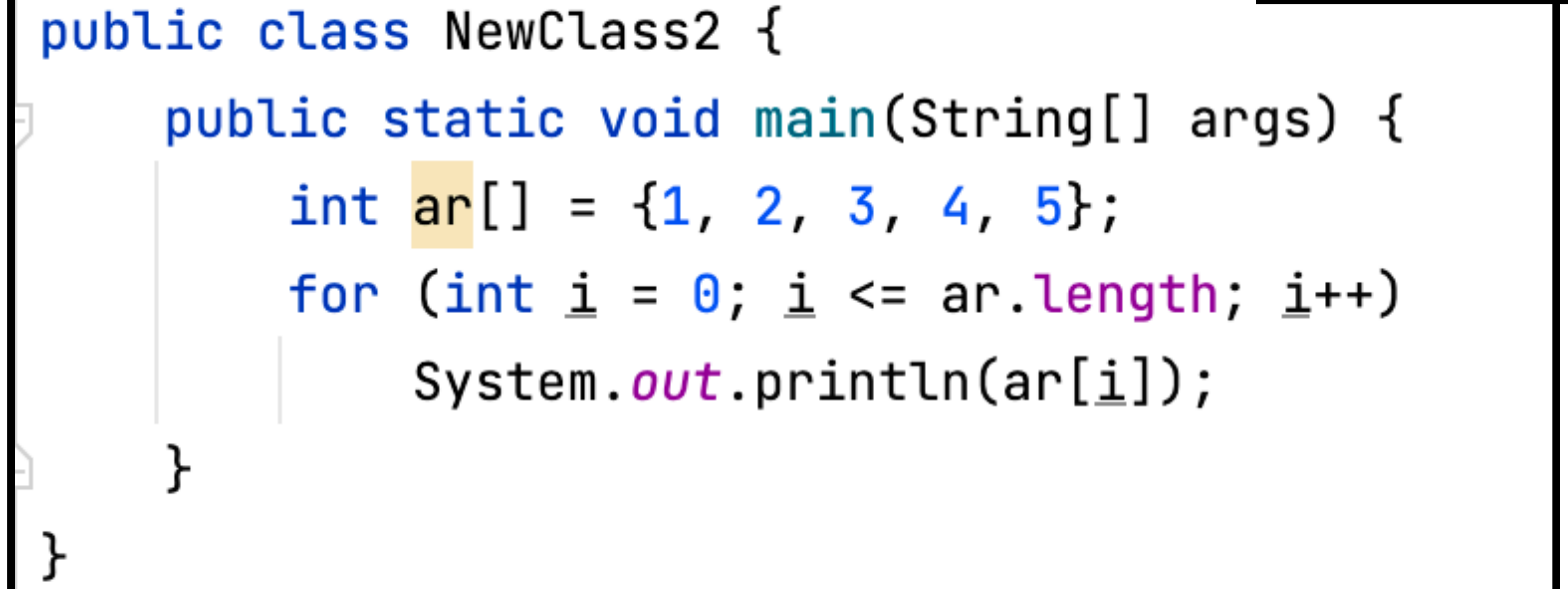


```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
System.out.println(cars[0]);
// Outputs Volvo
```



```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
System.out.println(cars.length);
// Outputs 4
```

- Be careful:

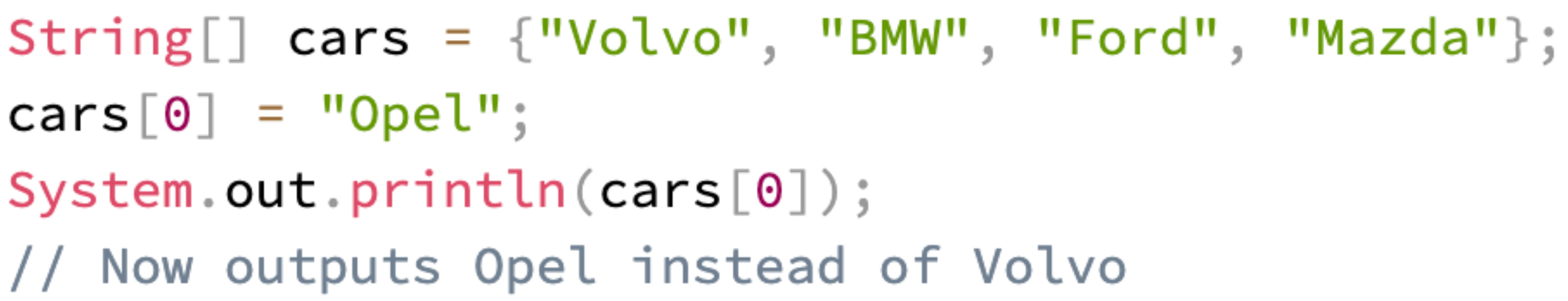


```
// A Common cause index out of bound
public class NewClass2 {
    public static void main(String[] args) {
        int ar[] = {1, 2, 3, 4, 5};
        for (int i = 0; i <= ar.length; i++)
            System.out.println(ar[i]);
    }
}
```

Arrays

Changing

- To change the value of a specific element, refer to the index number:

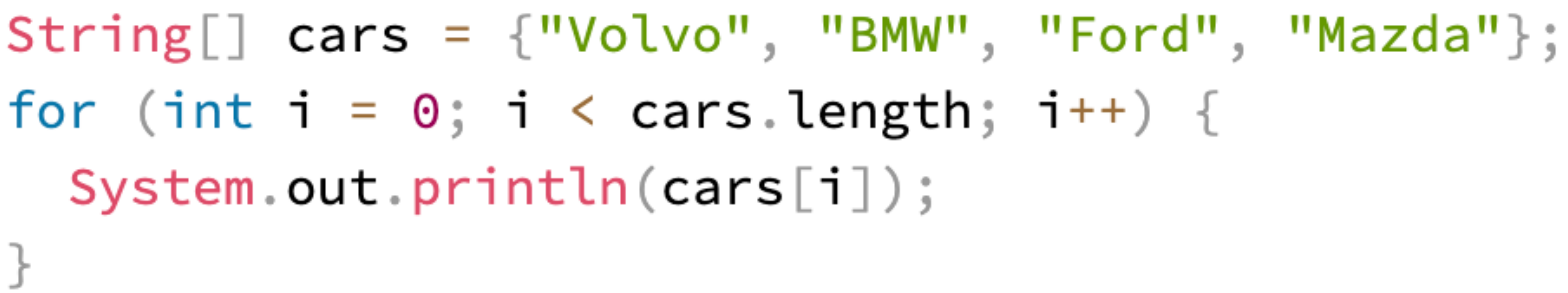


```java
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
cars[0] = "Opel";
System.out.println(cars[0]);
// Now outputs Opel instead of Volvo
```

Arrays

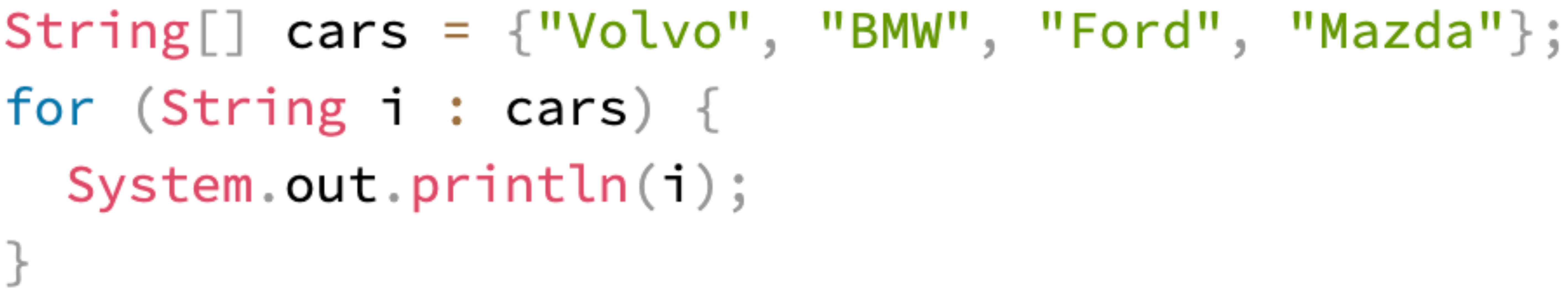
Loop Through

- You can loop through the array elements with the `for` loop, and use the `length` property to specify how many times the loop should run.



```java
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
for (int i = 0; i < cars.length; i++) {
  System.out.println(cars[i]);
}
```

- Using for-each loop ==>



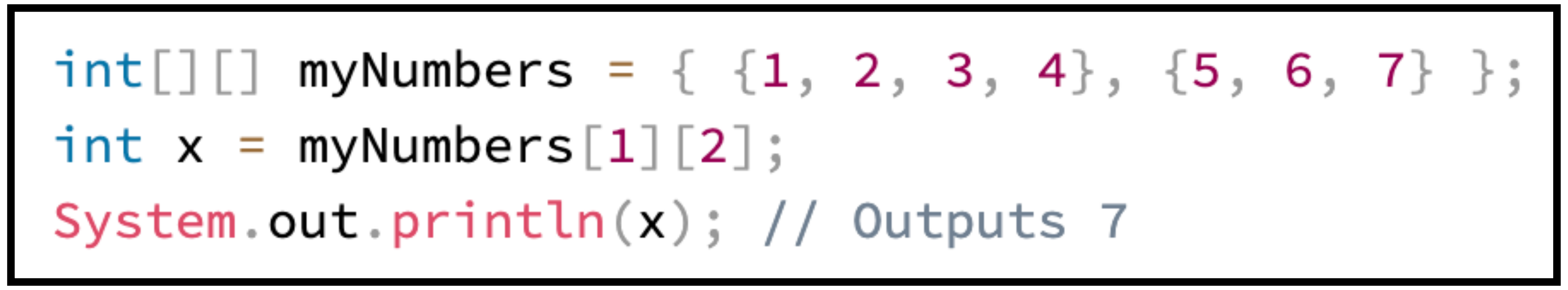
```java
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
for (String i : cars) {
  System.out.println(i);
}
```

Arrays

Multidimensional

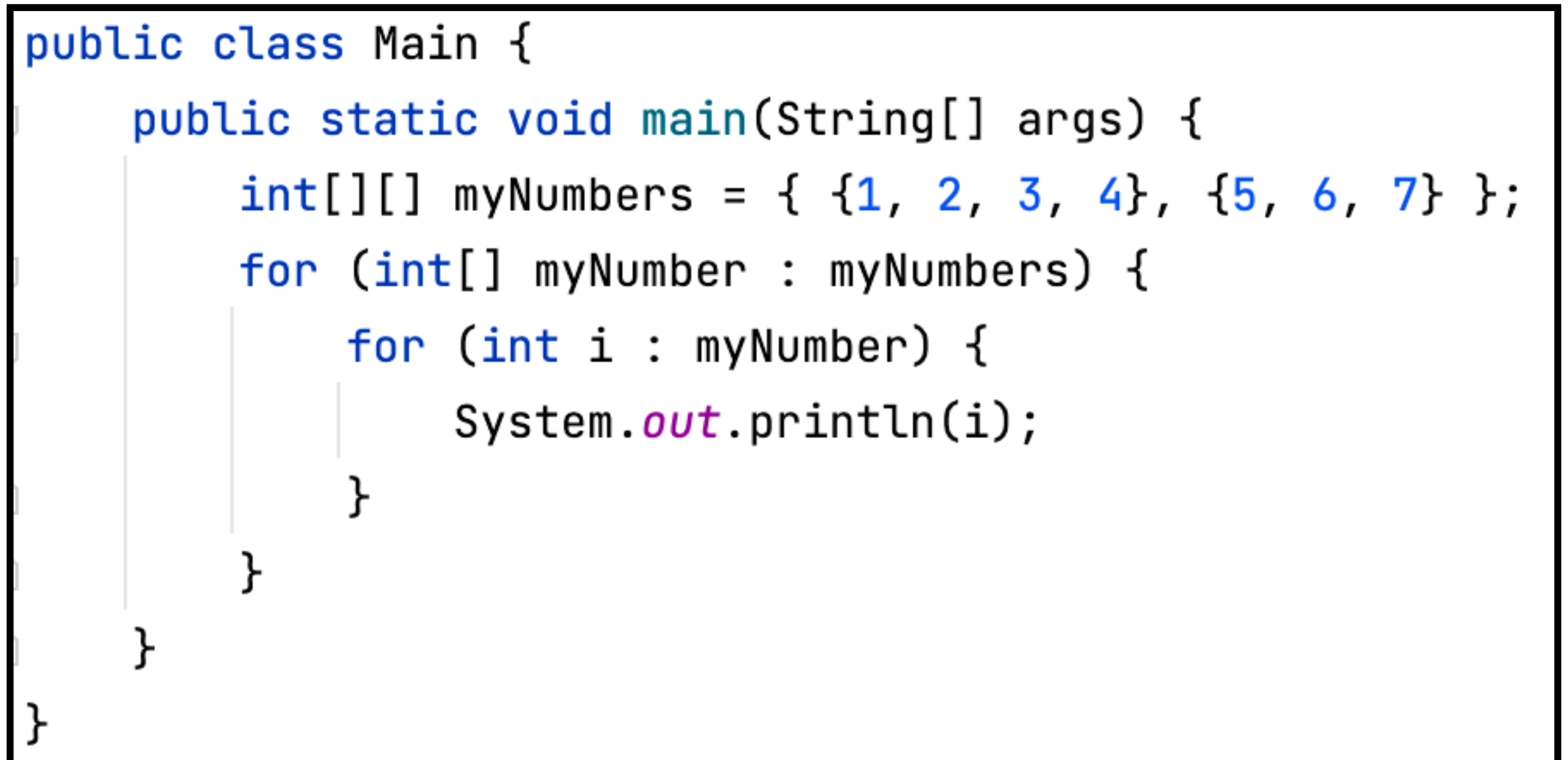
- A multidimensional array is an array of arrays.

- To create a two-dimensional array, add each array within its own set of **curly braces**:



```
int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} };
int x = myNumbers[1][2];
System.out.println(x); // Outputs 7
```

- We can also use a `for loop` inside another `for loop` to get the elements of a two-dimensional array (we still have to point to the two indexes):

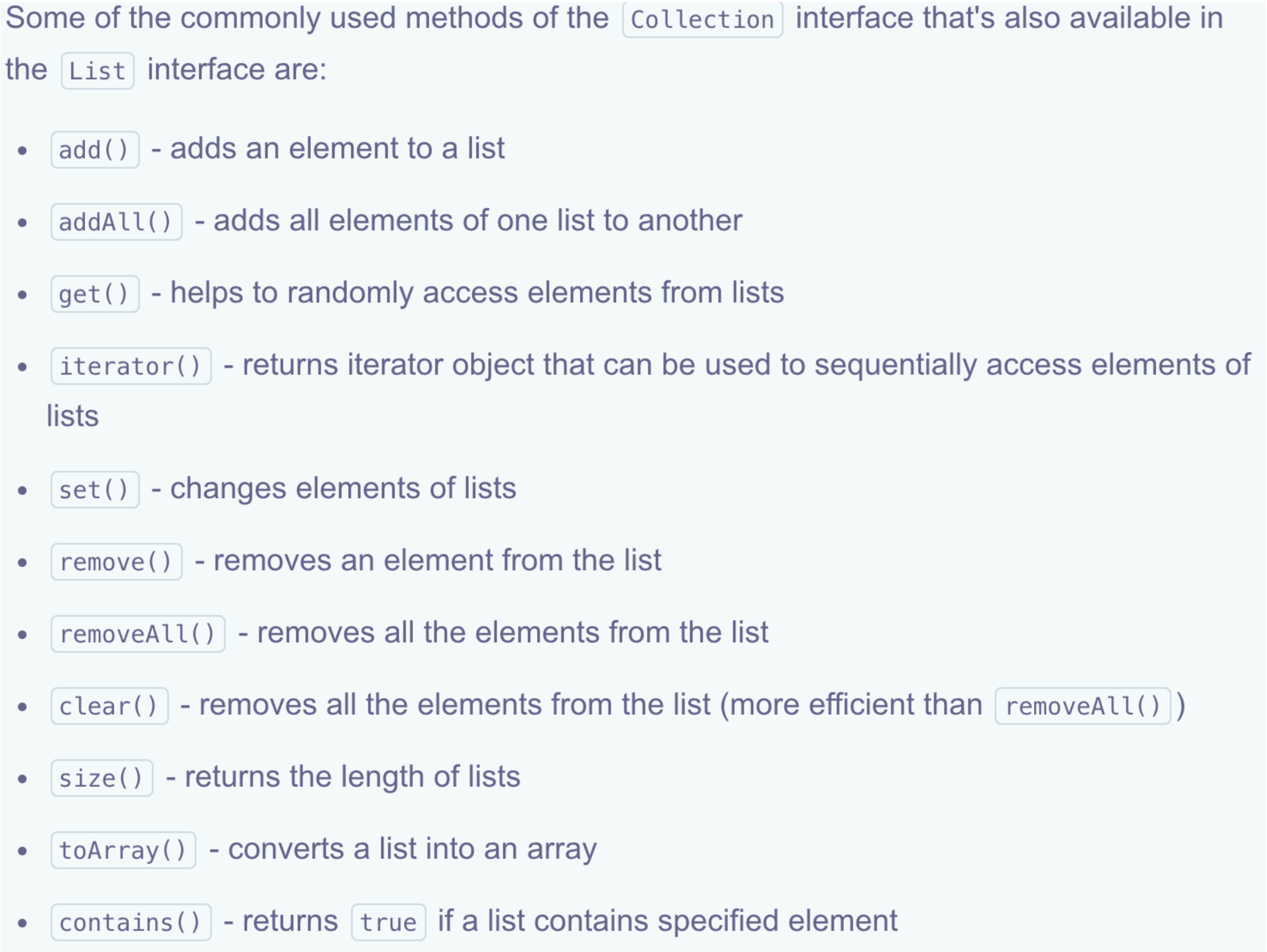


```
public class Main {
    public static void main(String[] args) {
        int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} };
        for (int[] myNumber : myNumbers) {
            for (int i : myNumber) {
                System.out.println(i);
            }
        }
    }
}
```

Collections

- The *Java Collections* API provides a set of classes and interfaces that makes it easier to work with collections of objects, e.g. lists, maps, sets etc.

- The Java Collection interface represents the operations possible on a generic collection, like on a List, Set, Map, etc.



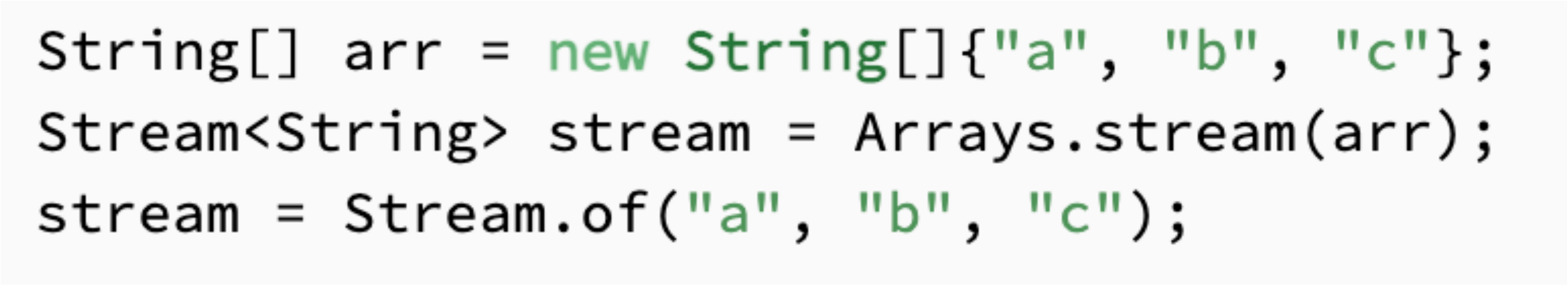
Some of the commonly used methods of the `Collection` interface that's also available in the `List` interface are:

- `add()` - adds an element to a list
- `addAll()` - adds all elements of one list to another
- `get()` - helps to randomly access elements from lists
- `iterator()` - returns iterator object that can be used to sequentially access elements of lists
- `set()` - changes elements of lists
- `remove()` - removes an element from the list
- `removeAll()` - removes all the elements from the list
- `clear()` - removes all the elements from the list (more efficient than `removeAll()`)
- `size()` - returns the length of lists
- `toArray()` - converts a list into an array
- `contains()` - returns `true` if a list contains specified element

Streams

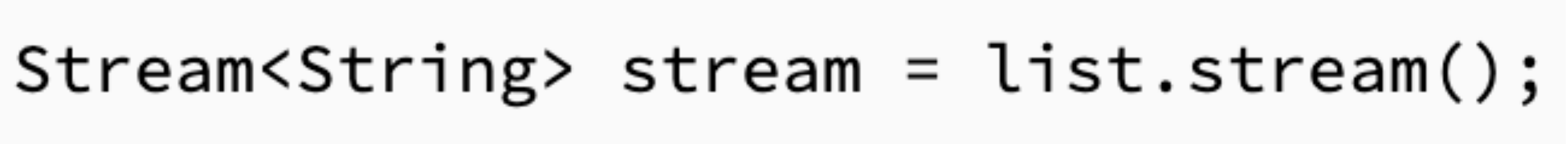
Creation

- Streams can be created from different element sources e.g. collection or array with the help of *stream()* and *of()* methods:



```
String[] arr = new String[]{"a", "b", "c"};
Stream<String> stream = Arrays.stream(arr);
stream = Stream.of("a", "b", "c");
```

- A *stream()* default method is added to the *Collection* interface and allows creating a *Stream<T>* using any collection as an element source*:*

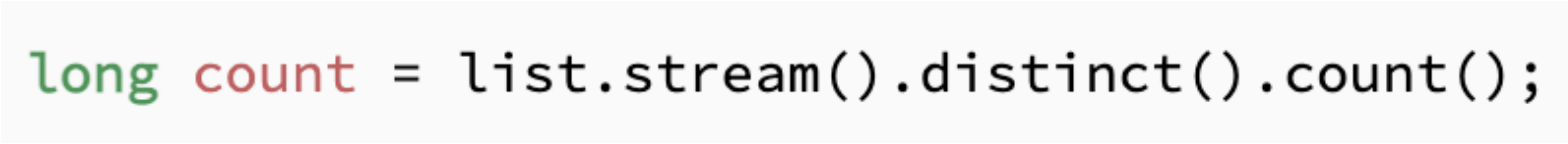


```
Stream<String> stream = list.stream();
```

Streams

Operations

- There are many useful operations that can be performed on a stream.
  - Divided into **intermediate operations** (return *Stream<T>*) that allow chaining
  - And **terminal operations** (return a result of definite type)



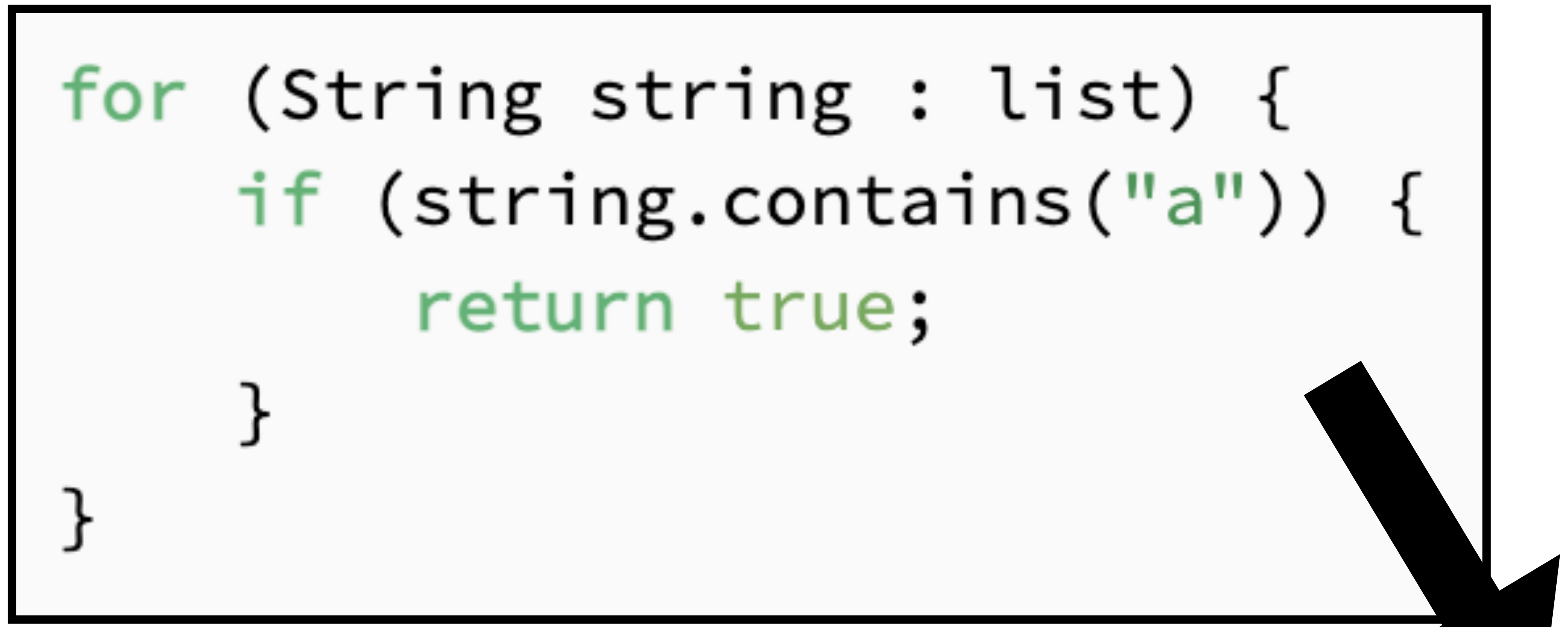
```
long count = list.stream().distinct().count();
```

  - So, the *distinct()* method represents an intermediate operation, which creates a new stream of unique elements of the previous stream. And the *count()* method is a terminal operation*,* which returns stream's size.

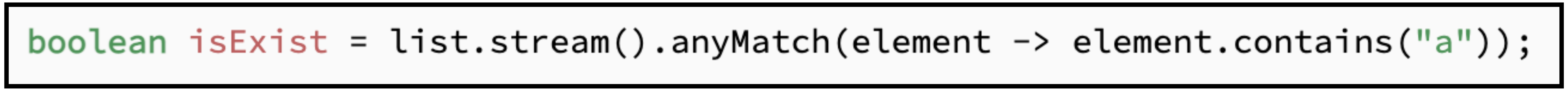
Streams

Iterating

- Stream API helps to substitute *for*, *for-each*, and *while* loops
- allows concentrating on operation's logic, but not on the iteration over the sequence of elements



```
for (String string : list) {
    if (string.contains("a")) {
        return true;
    }
}
```

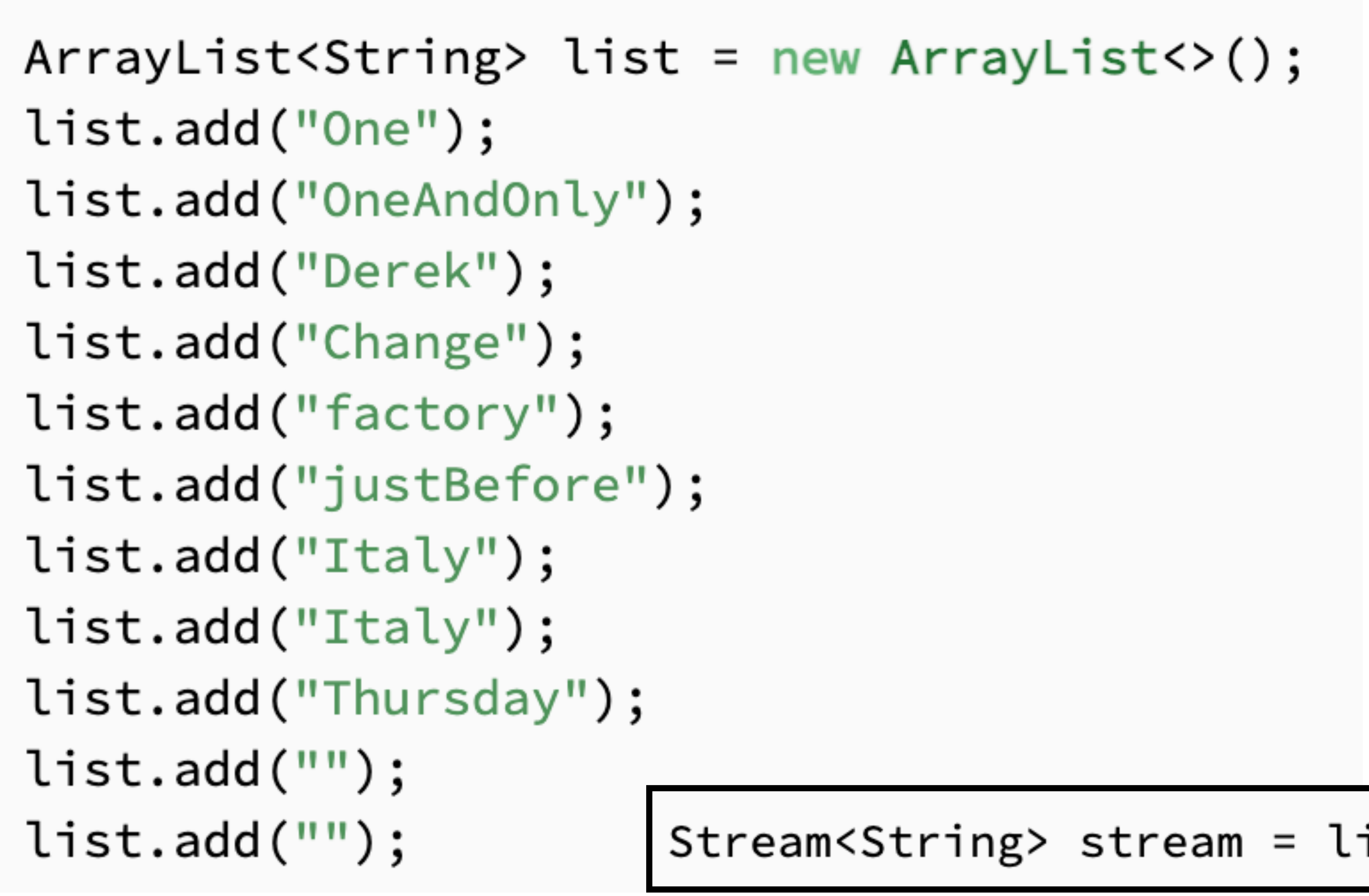


```
boolean isExist = list.stream().anyMatch(element -> element.contains("a"));
```

Streams

Filtering

- The *filter()* method allows us to pick a stream of elements that satisfy a predicate



```
ArrayList<String> list = new ArrayList<>();
list.add("One");
list.add("OneAndOnly");
list.add("Derek");
list.add("Change");
list.add("factory");
list.add("justBefore");
list.add("Italy");
list.add("Italy");
list.add("Thursday");
list.add("");
list.add("");
```

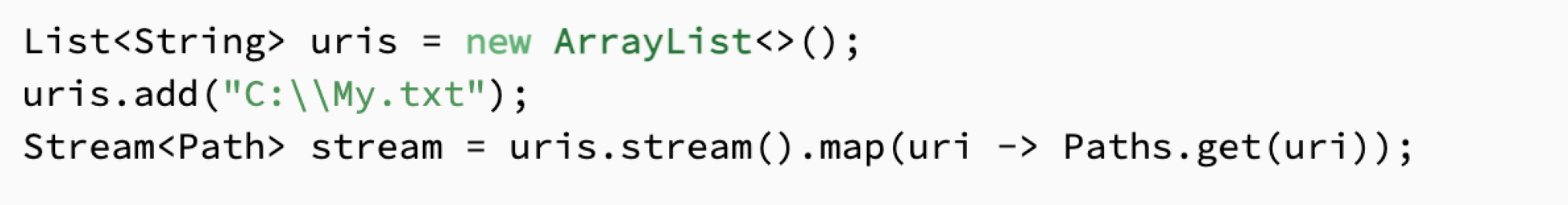
A creates a *Stream<String>* of the *List<String>*, finds all elements of this stream which contain *char "d"*, and creates a new stream containing only the filtered elements:

```
Stream<String> stream = list.stream().filter(element -> element.contains("d"));
```

Streams

Mapping

- To convert elements of a *Stream* by applying a special function to them and to collect these new elements into a *Stream*, we can use the *map()* method:



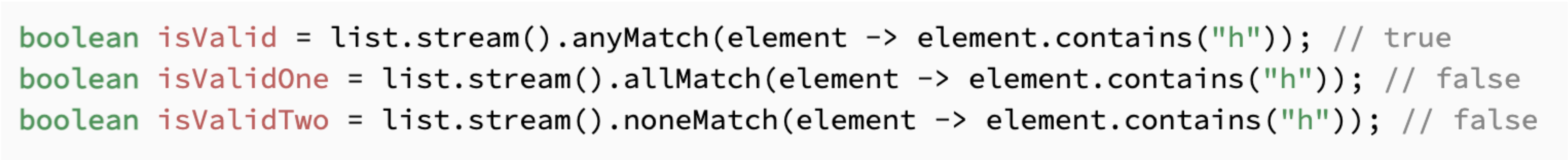
```
List<String> uris = new ArrayList<>();
uris.add("C:\\My.txt");
Stream<Path> stream = uris.stream().map(uri -> Paths.get(uri));
```

- So, the code above converts *Stream<String>* to the *Stream<Path>* by applying a specific lambda expression to every element of the initial *Stream*.

Streams

Matching

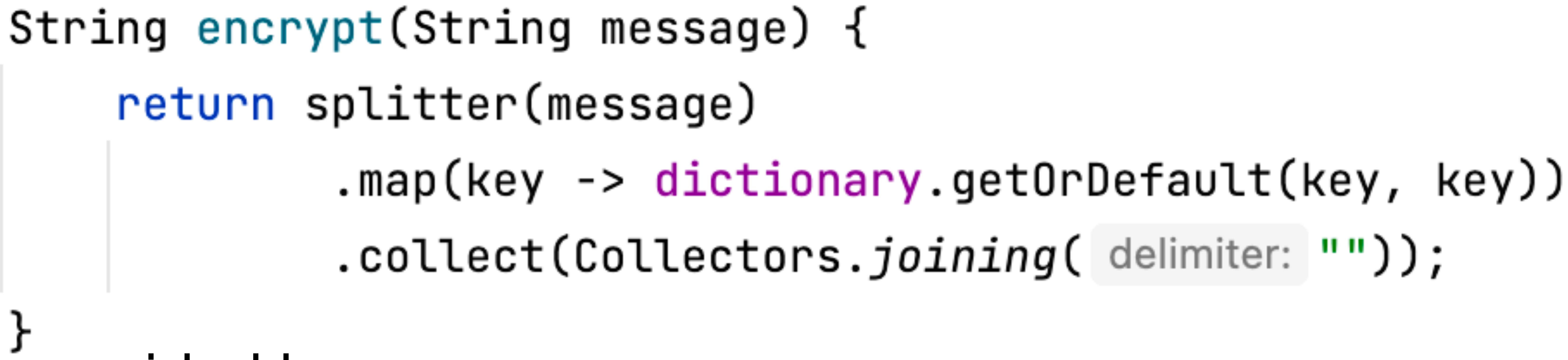
- Stream API gives a handy set of instruments to validate elements of a sequence according to some predicate.
  - *anyMatch(), allMatch(), noneMatch().*
  - Their names are self-explanatory. Those are terminal operations that return a *boolean*



```
boolean isValid = list.stream().anyMatch(element -> element.contains("h")); // true
boolean isValidOne = list.stream().allMatch(element -> element.contains("h")); // false
boolean isValidTwo = list.stream().noneMatch(element -> element.contains("h")); // false
```

Streams

Collecting



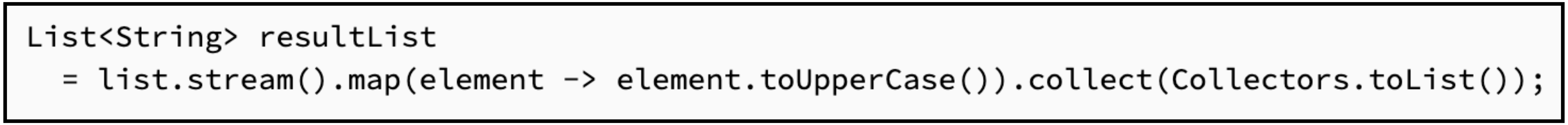
```
String encrypt(String message) {
    return splitter(message)
            .map(key -> dictionary.getOrDefault(key, key))
            .collect(Collectors.joining( delimiter: ""));
}
```

- The reduction can also be provided by the *collect()* method of type *Stream*

- handy in case of converting a stream to a *Collection* or a *Map* or a single string

- There is a utility class *Collectors* which provide a solution for almost all typical collecting operations.

  - For some, not trivial tasks, a custom *Collector* can be created.



```
List<String> resultList
  = list.stream().map(element -> element.toUpperCase()).collect(Collectors.toList());
```

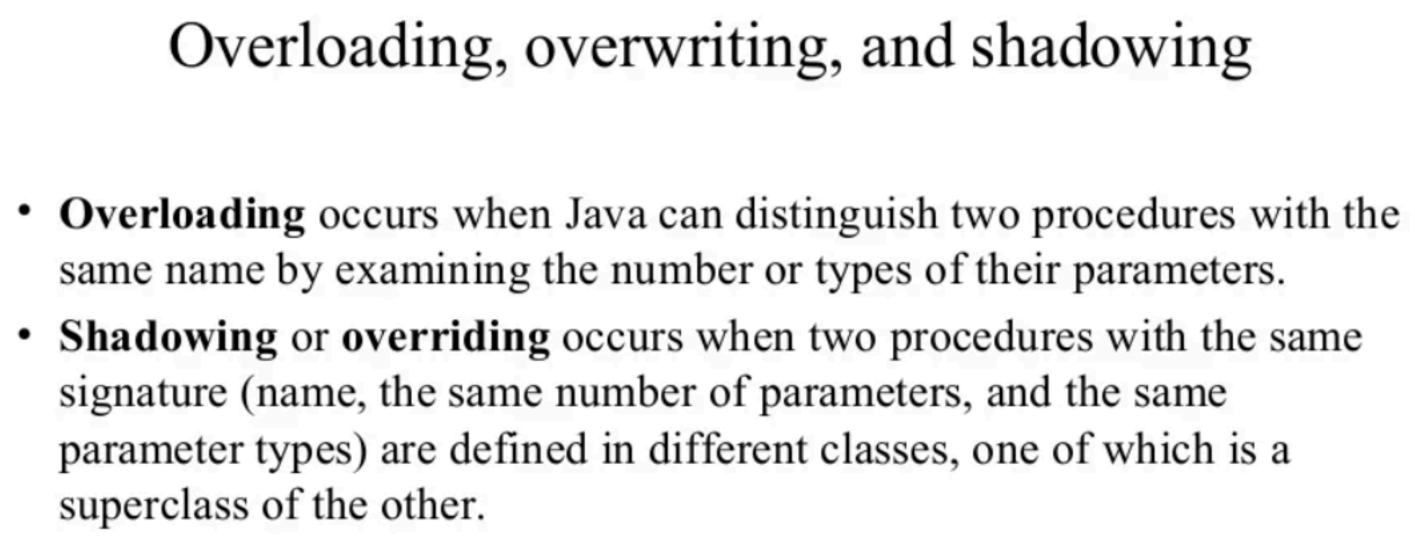
Data hiding and encapsulation

- Data-hiding or encapsulation is an important part of the 00 paradigm.
- Classes should carefully control access to their data and methods in order to
    - Hide the irrelevant implementation-level details so they can be easily changed
    - Protect the class against accidental or malicious damage.
    - Keep the externally visible class simple and easy to document
- Java has a simple access control mechanism to help with encapsulation
    - Modifiers: public, protected, private, and package (default)

Polymorphism

- ability of an object to take many forms
  - allows us to perform the same action in many different ways

- Method Overloading vs Method overriding/shadowing

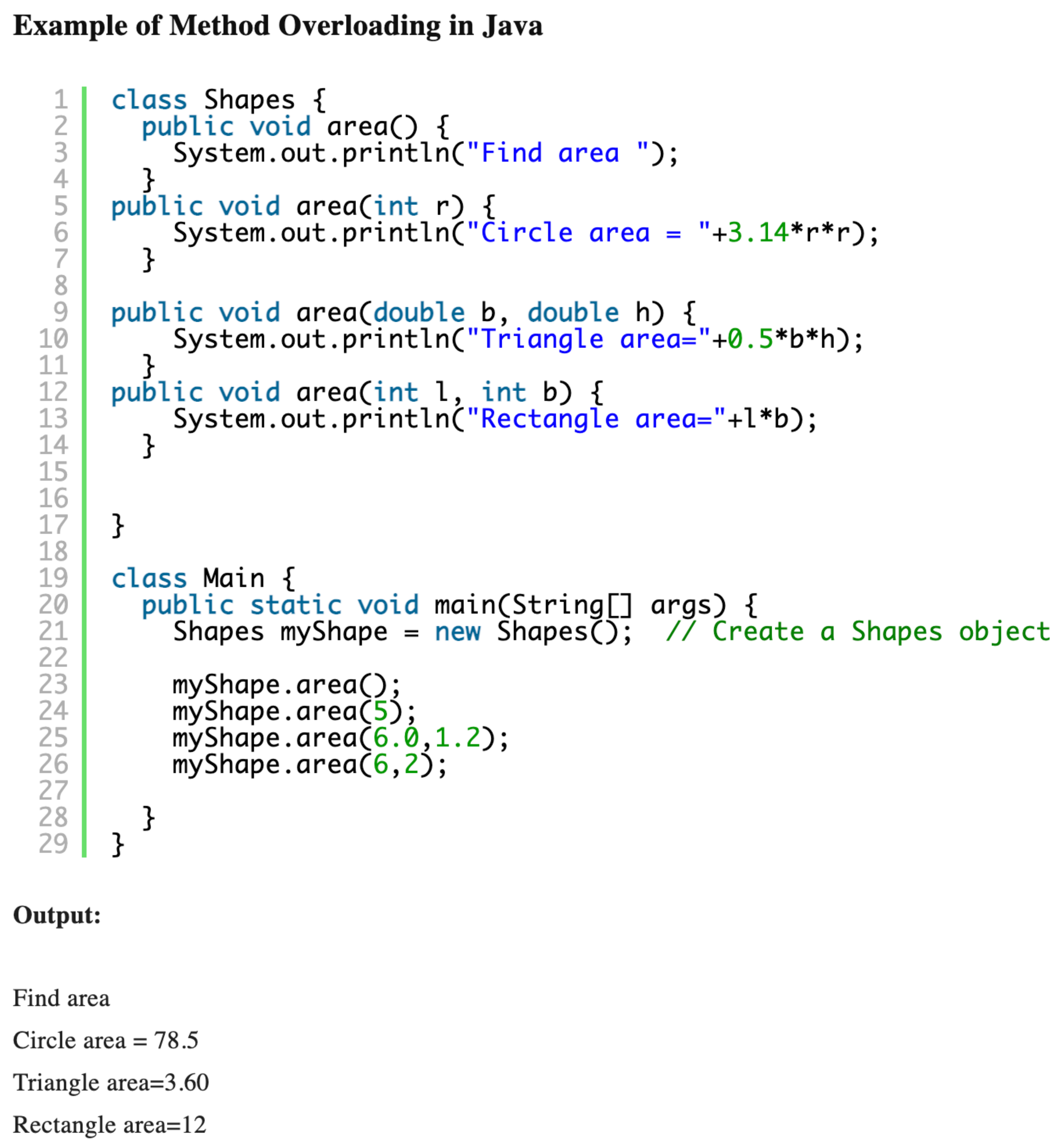


Overloading, overwriting, and shadowing

- **Overloading** occurs when Java can distinguish two procedures with the same name by examining the number or types of their parameters.
- **Shadowing** or **overriding** occurs when two procedures with the same signature (name, the same number of parameters, and the same parameter types) are defined in different classes, one of which is a superclass of the other.

Method Overloading

- occurs when there is more than one method of the same name in the class



Example of Method Overloading in Java

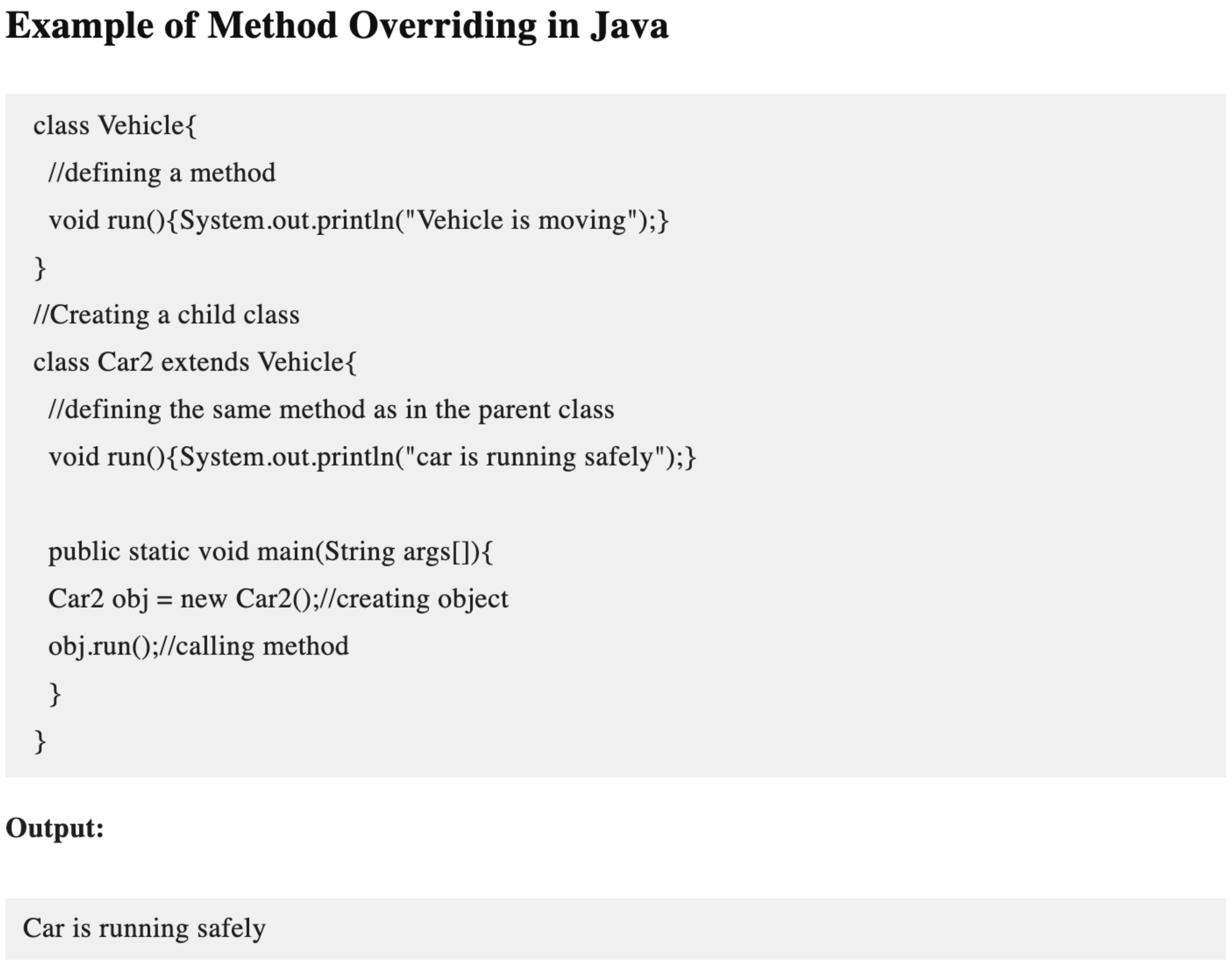
```java
class Shapes {
  public void area() {
    System.out.println("Find area ");
  }
public void area(int r) {
    System.out.println("Circle area = "+3.14*r*r);
  }

public void area(double b, double h) {
    System.out.println("Triangle area="+0.5*b*h);
  }
public void area(int l, int b) {
    System.out.println("Rectangle area="+l*b);
  }


}

class Main {
  public static void main(String[] args) {
    Shapes myShape = new Shapes();  // Create a Shapes object

    myShape.area();
    myShape.area(5);
    myShape.area(6.0,1.2);
    myShape.area(6,2);

  }
}
```

Output:

```
Find area
Circle area = 78.5
Triangle area=3.60
Rectangle area=12
```

Method Overriding

- occurs when a subclass or a child class has the same method as declared in the parent class



Example of Method Overriding in Java

```
class Vehicle{
  //defining a method
  void run(){System.out.println("Vehicle is moving");}
}
//Creating a child class
class Car2 extends Vehicle{
  //defining the same method as in the parent class
  void run(){System.out.println("car is running safely");}

  public static void main(String args[]){
  Car2 obj = new Car2();//creating object
  obj.run();//calling method
  }
}
```

Output:

```
Car is running safely
```

Exceptions

- can occur for many different reasons
  - A user has entered an invalid data.
  - A file that needs to be opened cannot be found.
  - A network connection has been lost in the middle of communications or the JVM has run out of memory.
- caused by user error, others by programmer error, and others by physical resources

Categories of Exceptions

- **Checked** Exceptions
  - checked (notified) by the compiler at compilation-time
  - also called as compile time exceptions
- **Unchecked** Exceptions
  - occurs at the time of execution
  - also called as **Runtime Exceptions**
    - programming bugs, such as logic errors or improper use of an API
    - ignored at the time of compilation

Exceptions

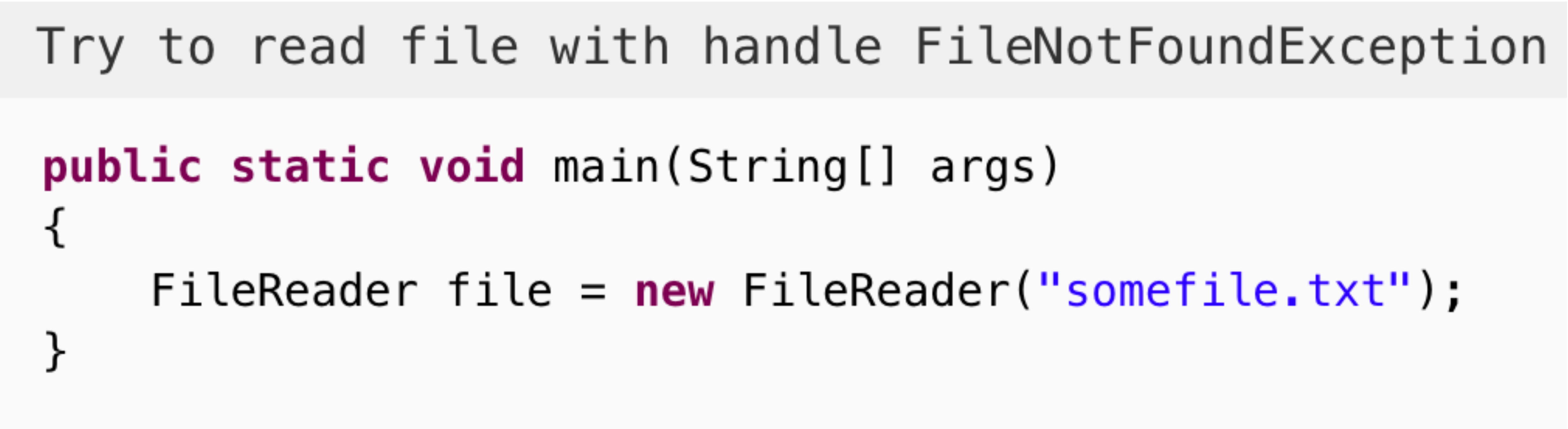
- An exception (or exceptional event) is a problem that arises during the execution of a program.
- When an **Exception** occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.
- An exception can occur for many different reasons. Following are some scenarios where an exception occurs.
  - A user has entered an invalid data.
  - A file that needs to be opened cannot be found.
  - A network connection has been lost in the middle of communications
  - JVM has run out of memory, …
- **In Java, all errors and exceptions are of type with Throwable class.**

Exceptions

Checked

- Checked exceptions are checked by the Java compiler so they are called compile time exceptions.

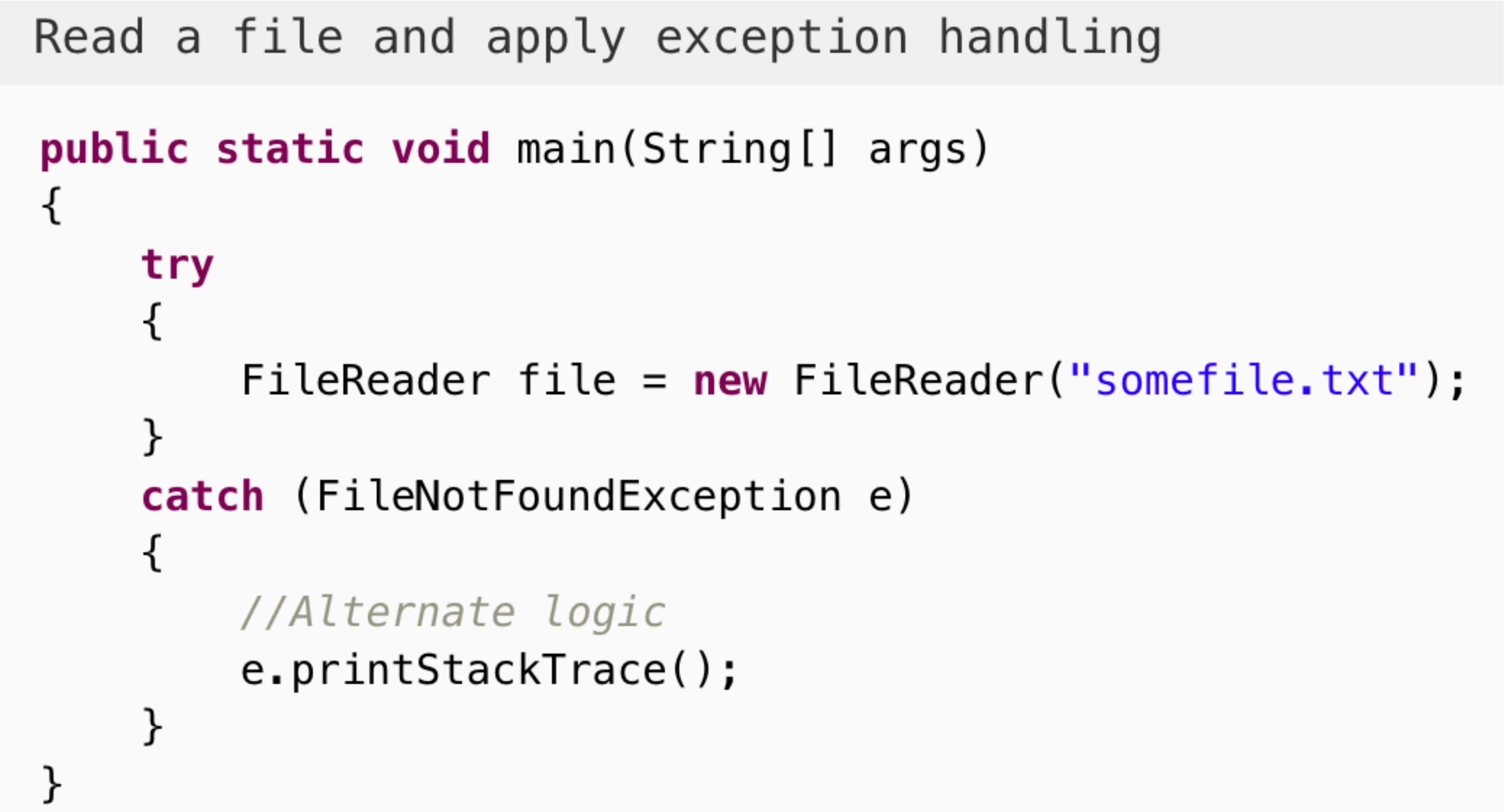
- E.g. FileNotFoundException is a checked exception in Java. Anytime, we want to read a file from the filesystem, Java forces us to handle an error situation where the file may not be present in the place.



Try to read file with handle FileNotFoundException

```java
public static void main(String[] args)
{
    FileReader file = new FileReader("somefile.txt");
}
```

will get compile-time error with the message – Unhandled exception type FileNotFoundException.



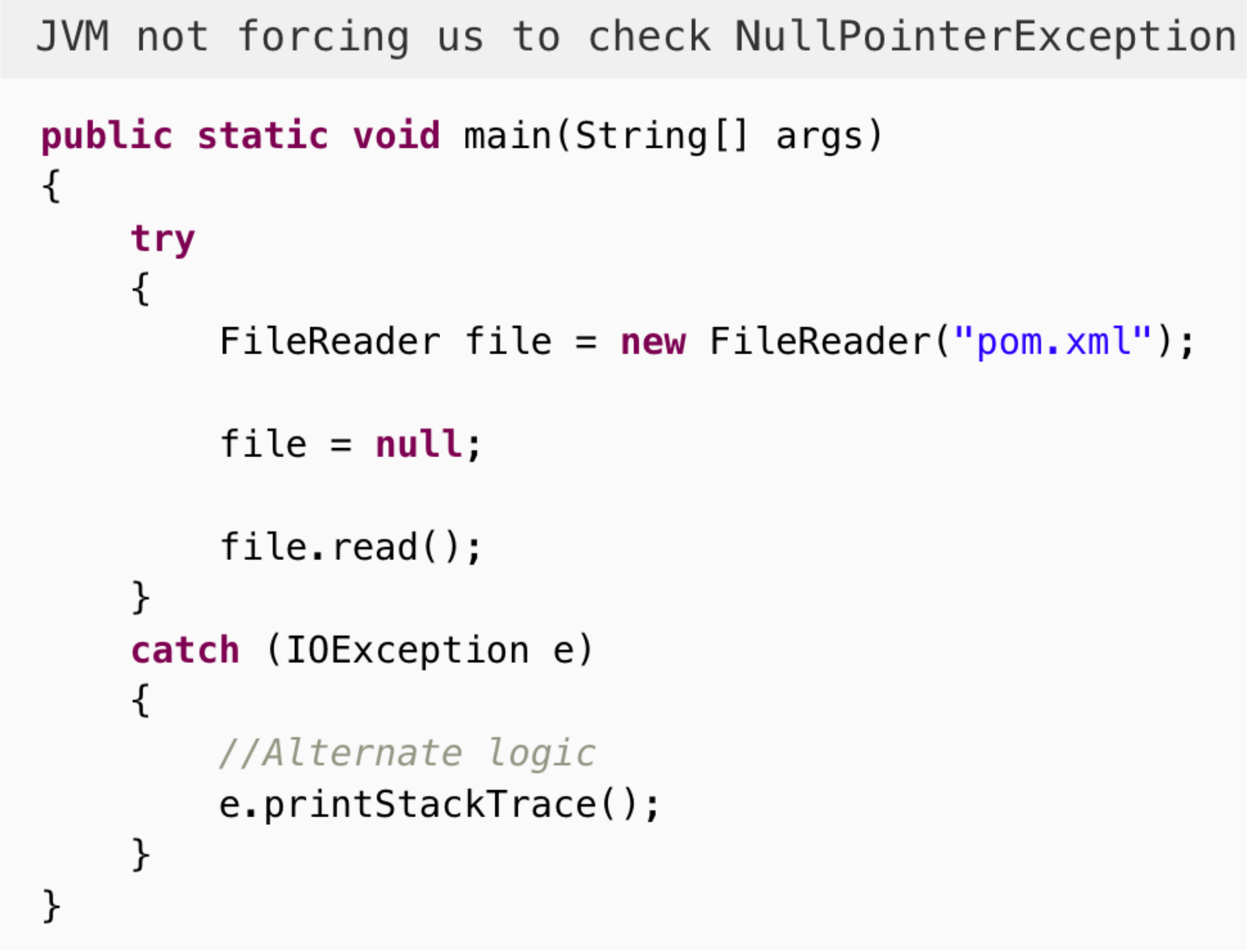
Read a file and apply exception handling

```java
public static void main(String[] args)
{
    try
    {
        FileReader file = new FileReader("somefile.txt");
    }
    catch (FileNotFoundException e)
    {
        //Alternate logic
        e.printStackTrace();
    }
}
```

Exceptions

Unchecked

- Unchecked exceptions will come into life and occur in the program, once any buggy code is executed.
- Unchecked Exceptions are subclasses of RuntimeException class.



JVM not forcing us to check NullPointerException

```java
public static void main(String[] args)
{
    try
    {
        FileReader file = new FileReader("pom.xml");

        file = null;

        file.read();
    }
    catch (IOException e)
    {
        //Alternate logic
        e.printStackTrace();
    }
}
```

The code in the given program does not give any compile-time error.
But when we run the example, it throws NullPointerException.
NullPointerException is an unchecked exception in Java.

Dependencies

- Why

Build Tooling

- How to create project

Testing

- JUnit
- Mocking