HSBC Technology Graduate Training

Programming Fundamentals: Java

Day 3 (Morning)

Wednesday 28 October 2020 | 9am

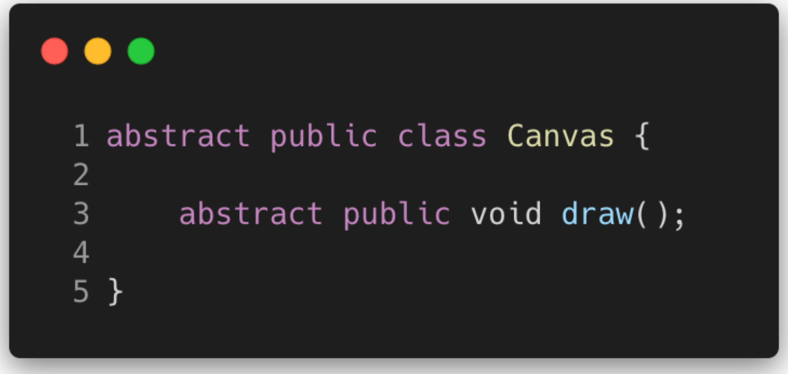
Contents

- Naming conventions
- Exceptions
- User Defined Exceptions
- Exception Handler Ordering
- Converting String to Number
- Methods vs. Procedures vs. Functions
- Arrays

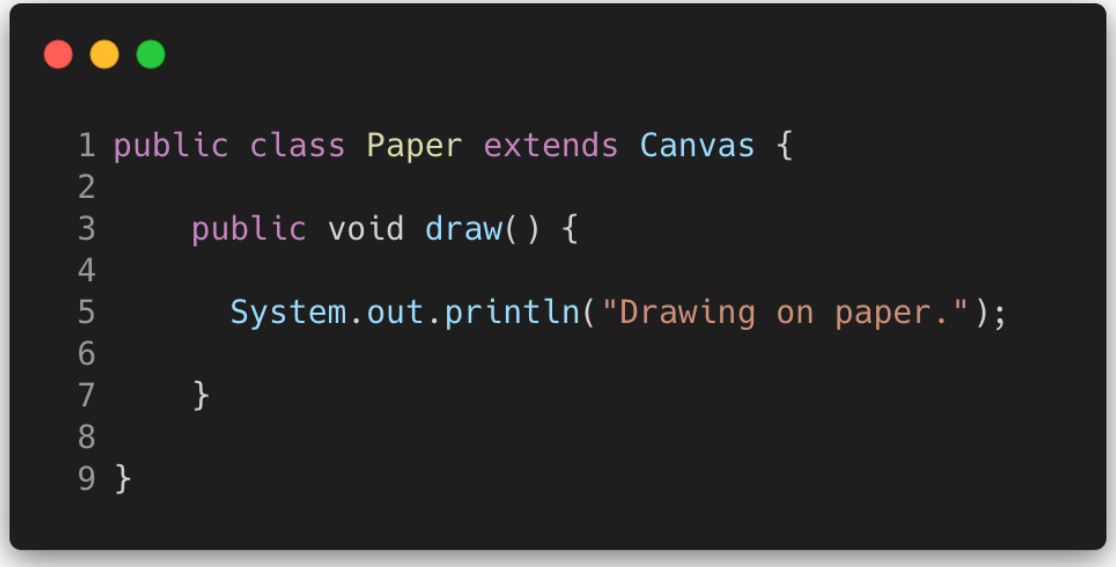
Naming conventions

NAMING CONVENTIONS: CLASSES

- Should be a noun (e.g. Canvas, Paper, City, London).
- First letter of class name should be upper-case.
- Classes are nouns because they represent the type of objects.



```java
abstract public class Canvas {

    abstract public void draw();

}
```

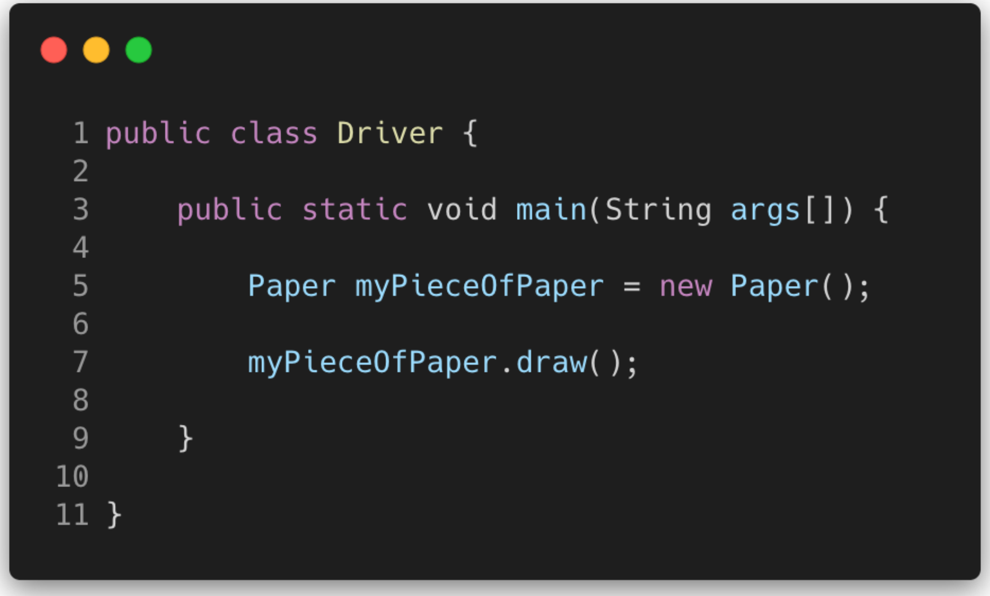


```java
public class Paper extends Canvas {

    public void draw() {

      System.out.println("Drawing on paper.");

    }

}
```

- In the example above, Paper inherits Canvas. In other words, Paper IS-A Canvas.
- When we create an object from class Paper, we can say this object is of type Paper.

NAMING CONVENTIONS: METHODS

- Should be a verb (e.g. Draw, Run, Stop).
- Camel case. For example (doSomething, drawLine, drawCircle).
- Methods are verbs because they take an action upon an object.



```
 1 public class Driver {
 2
 3     public static void main(String args[]) {
 4
 5         Paper myPieceOfPaper = new Paper();
 6
 7         myPieceOfPaper.draw();
 8
 9     }
10
11 }
```

- Line 5: Create a new object of type Paper called myPieceOfPaper.
- Line 7: Perform the draw() method on myPieceOfPaper.

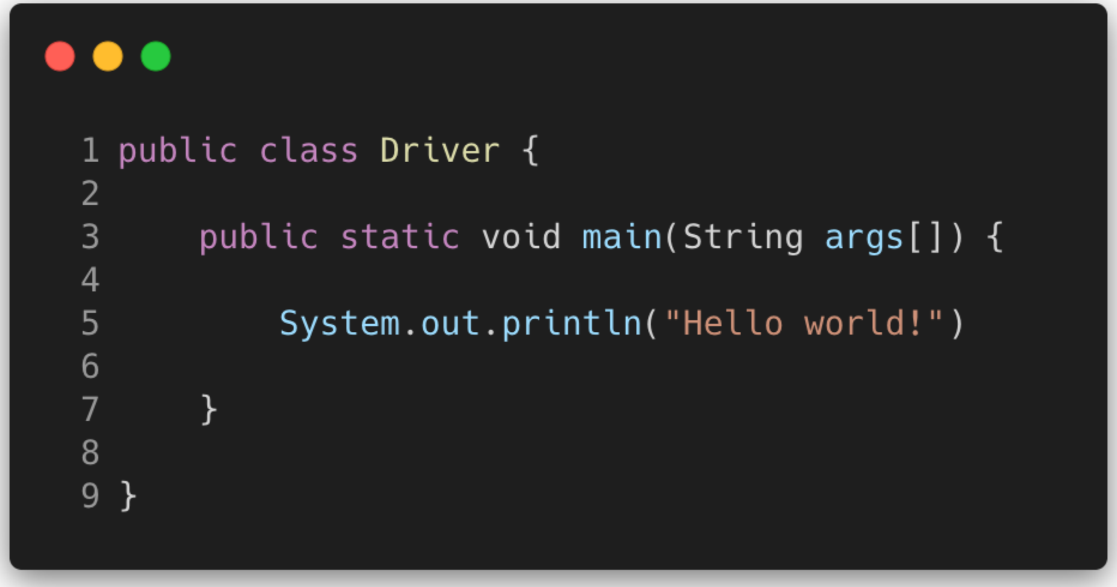
Exceptions

EXCEPTION

- **In Java, there are two types of fail scenarios:**
  - **Errors.**
  - **Exceptions.**

EXCEPTION: COMPILE TIME ERRORS

- Let’s look at errors first. What is a compile time error?
- It is an error that occurs before you run your code (at compile time).
- Often caused by syntax errors.

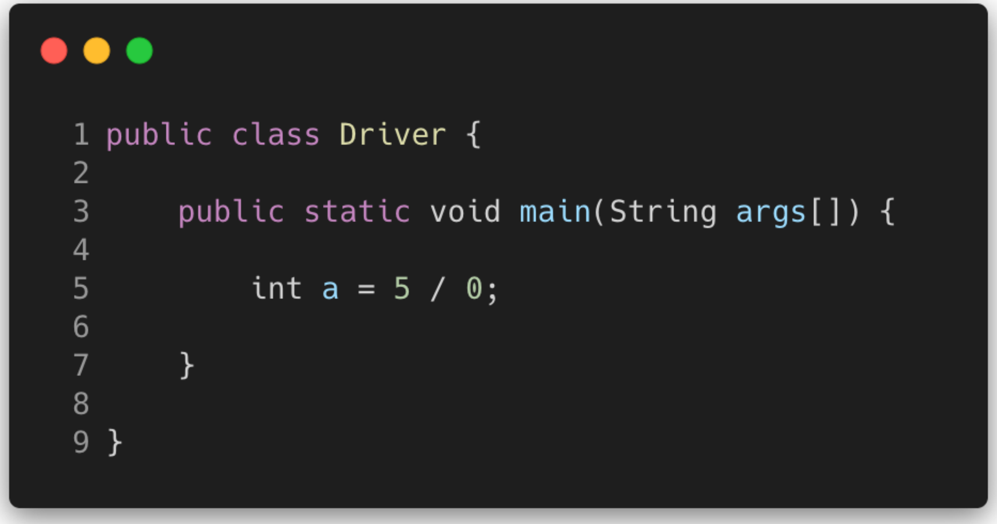


```java
1 public class Driver {
2
3     public static void main(String args[]) {
4
5         System.out.println("Hello world!")
6
7     }
8
9 }
```

- This code won’t compile and will throw an error at compile time.
- Line 5: Missing semi-colon at the end of the line.

EXCEPTIONS

- **What is an exception?**
- **An error that occurs at run-time (when your code is being run).**
- **Dividing by 0 is a common error.**



```
public class Driver {

    public static void main(String args[]) {

        int a = 5 / 0;

    }

}
```

- **This code will throw an <u>exception</u> when it is run.**

EXCEPTIONS

- **What happens when an exception occurs?**

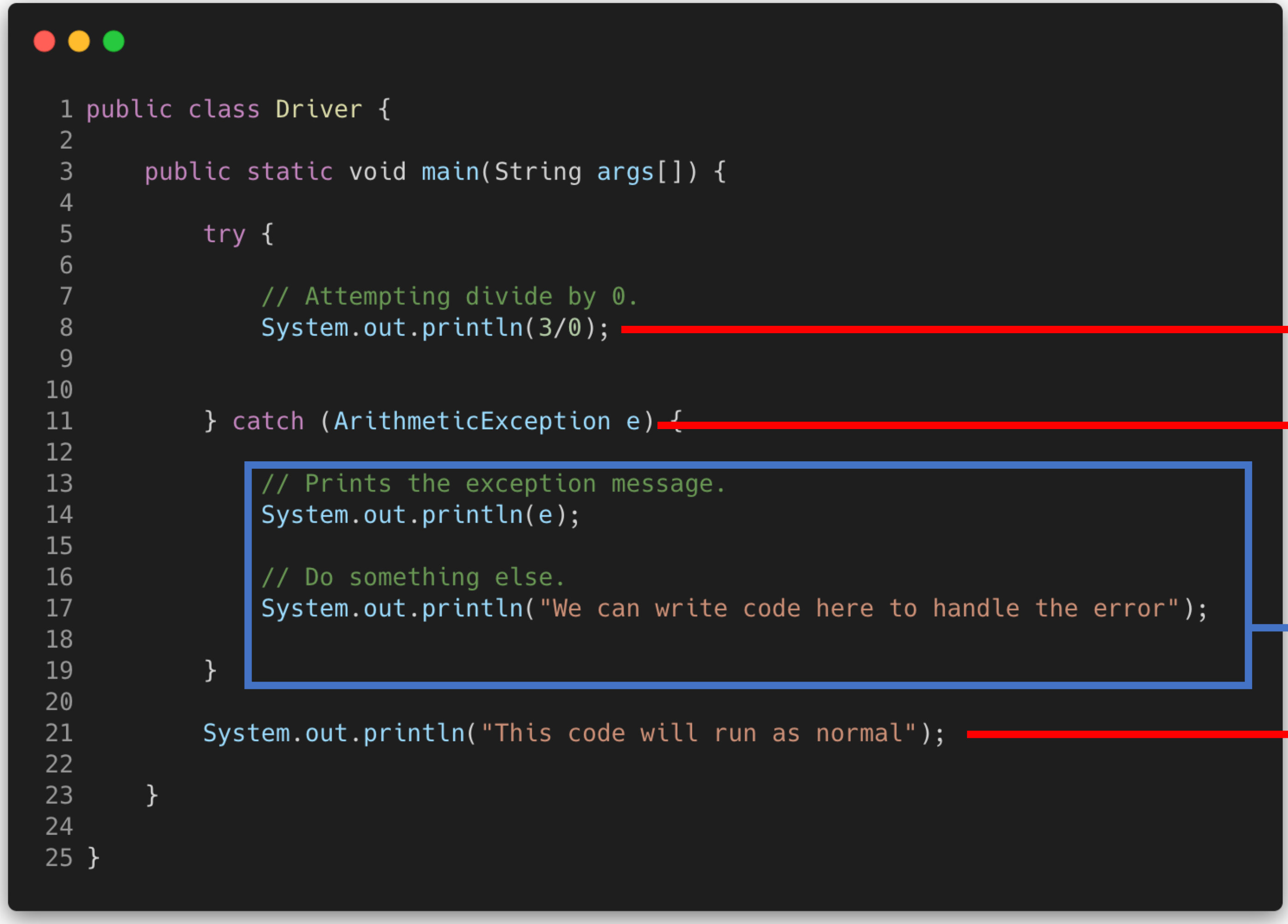
1. **Java will create an object of that exception class (e.g. FileNotFoundException, ArithmeticException). This class inherits the Exception class. In the previous example, Java will create an ArithmeticException.**

2. **Java will start looking for a user-defined exception handler. An exception handler is a block of code that will be executed when a specific exception is raised by Java.**

3. **If Java doesn’t find a user-defined exception handler. It will go to the default exception handler.**

4. **The default exception handler displays the exception message and terminates the program.**

EXCEPTIONS: EXAMPLE (WITH HANDLER)



```
public class Driver {

    public static void main(String args[]) {

        try {

            // Attempting divide by 0.
            System.out.println(3/0);


        } catch (ArithmeticException e) {

            // Prints the exception message.
            System.out.println(e);

            // Do something else.
            System.out.println("We can write code here to handle the error");

        }

        System.out.println("This code will run as normal");

    }

}
```

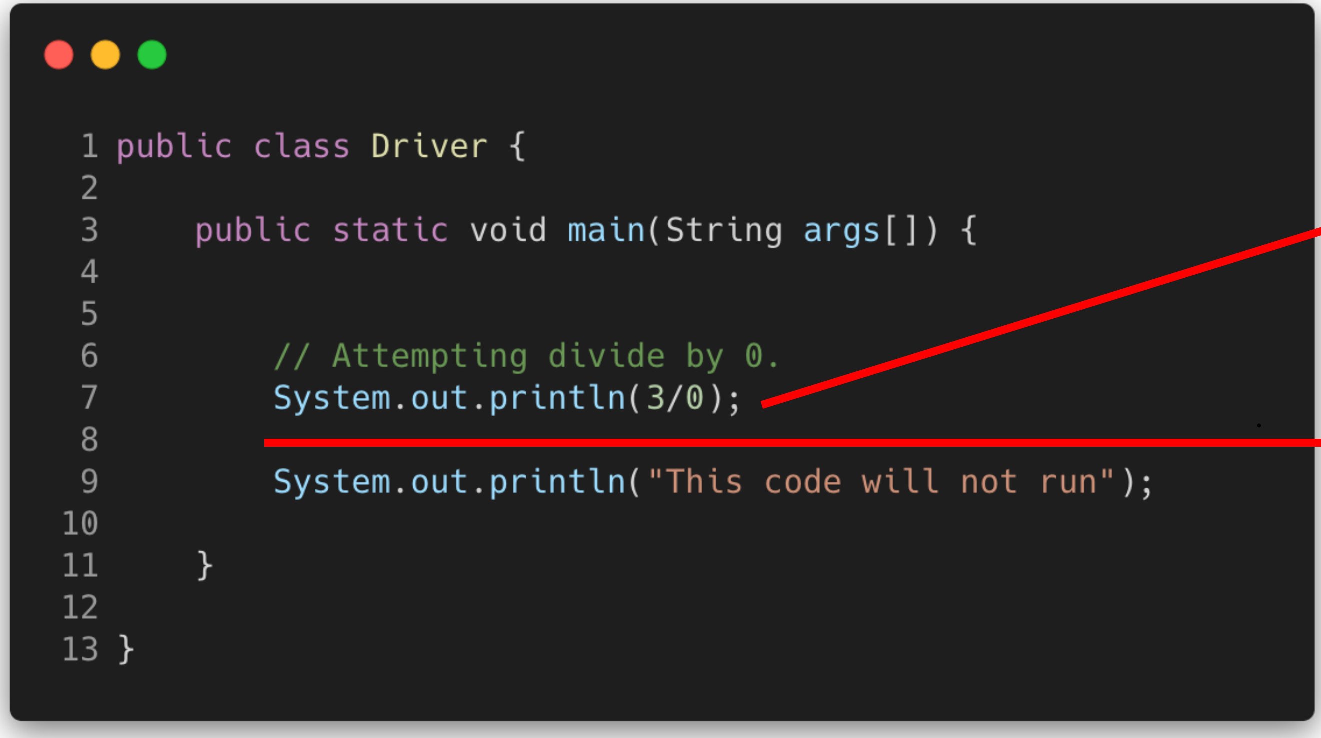
3/0 will cause Java to raise an exception. An ArithmeticException to be precise.

Java matches the exception thrown to a user-defined exception handler. In this case, an ArithmeticException handler is found.

The code in the ArithmeticException handler is run.

The program does not terminate. It continues running.

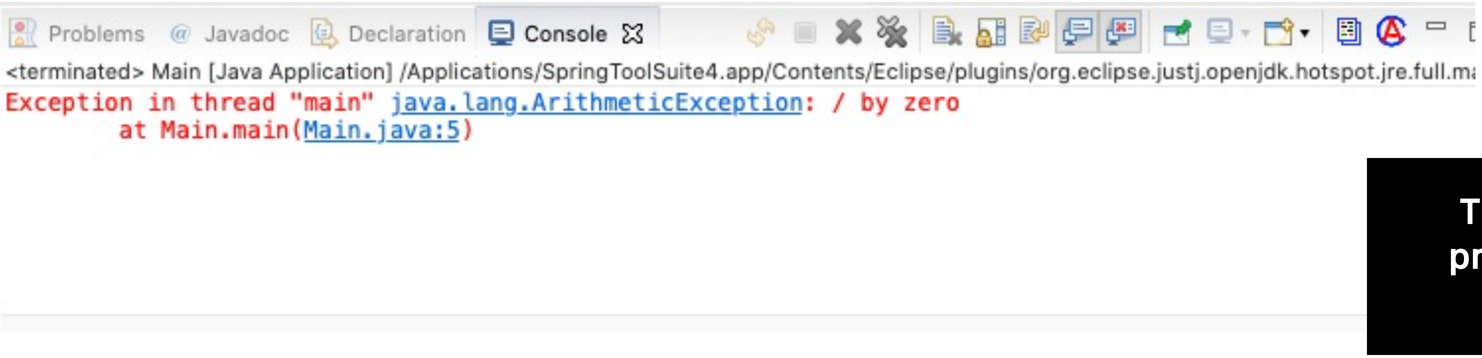
EXCEPTIONS: EXAMPLE (WITHOUT HANDLER)



```
public class Driver {

  public static void main(String args[]) {


    // Attempting divide by 0.
    System.out.println(3/0);

    System.out.println("This code will not run");

  }

}
```

3/0 will cause Java to raise an exception. An ArithmeticException to be precise. Java will look for a handler with ArithmeticException. None exists.

Program terminates here. Default exception handler is run.



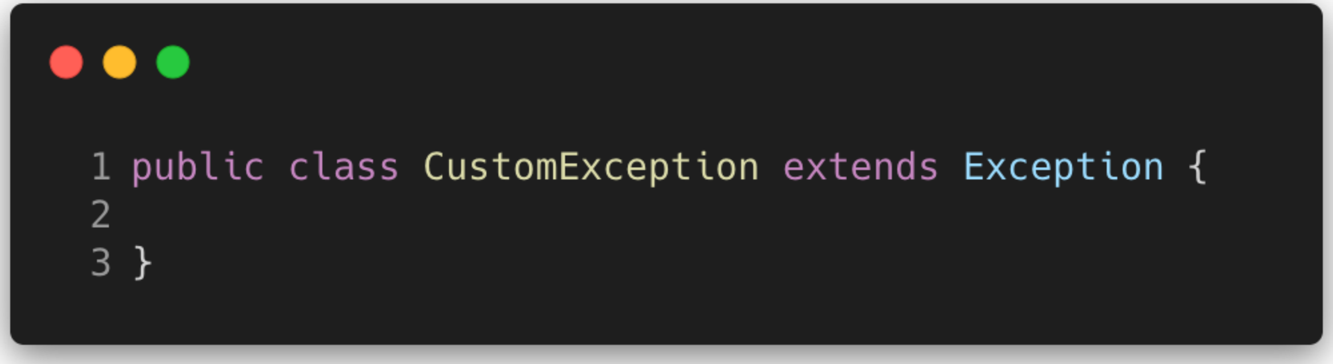
```
<terminated> Main [Java Application] /Applications/SpringToolSuite4.app/Contents/Eclipse/plugins/org.eclipse.justj.openjdk.hotspot.jre.full.ma
Exception in thread "main" java.lang.ArithmeticException: / by zero
	at Main.main(Main.java:5)
```

The default exception handler terminates the program and displays the error message in the console.

User-Defined Exceptions

USER-DEFINED EXCEPTIONS

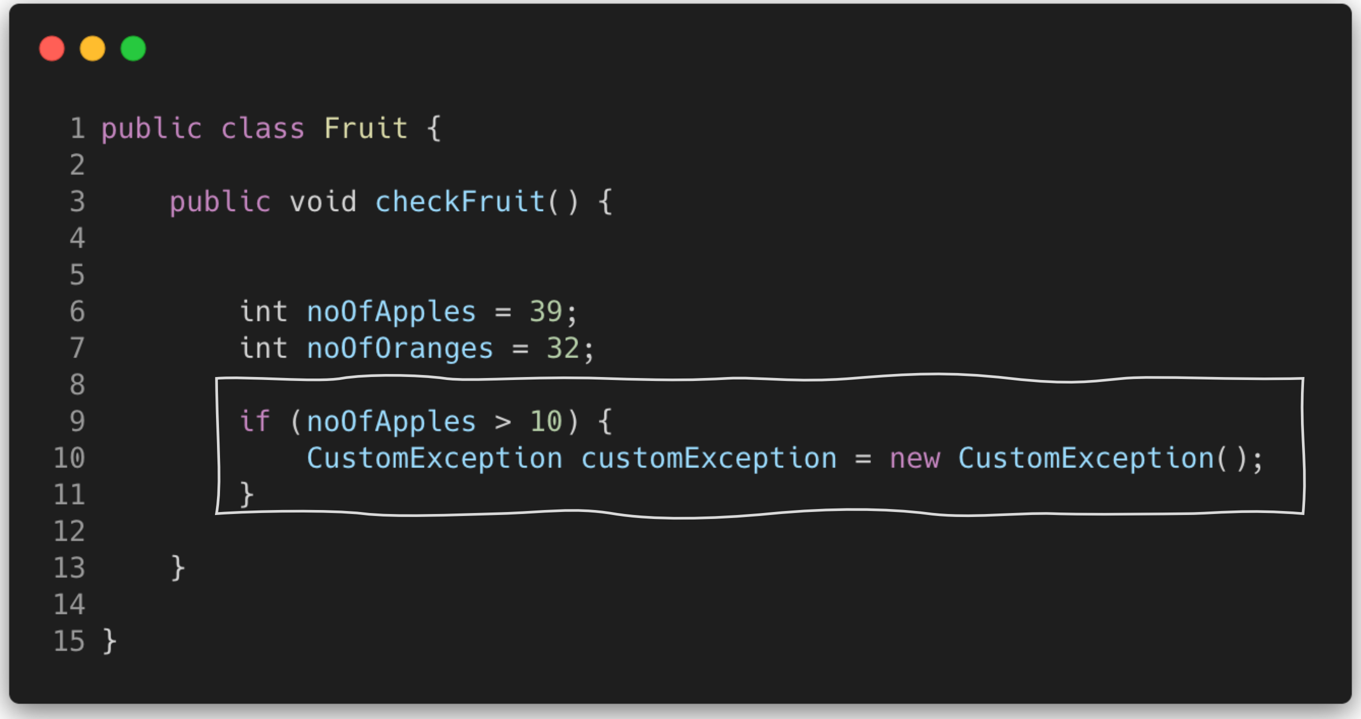
- In addition to the exception classes provided by Java such as ArithmeticException, we can define our own exception class.
- The first step to creating our own exception is to create a new class that inherits the Exception class.
- The example below shows the process of creating a custom exception class named CustomException.



```java
public class CustomException extends Exception {

}
```

USER-DEFINED EXCEPTIONS

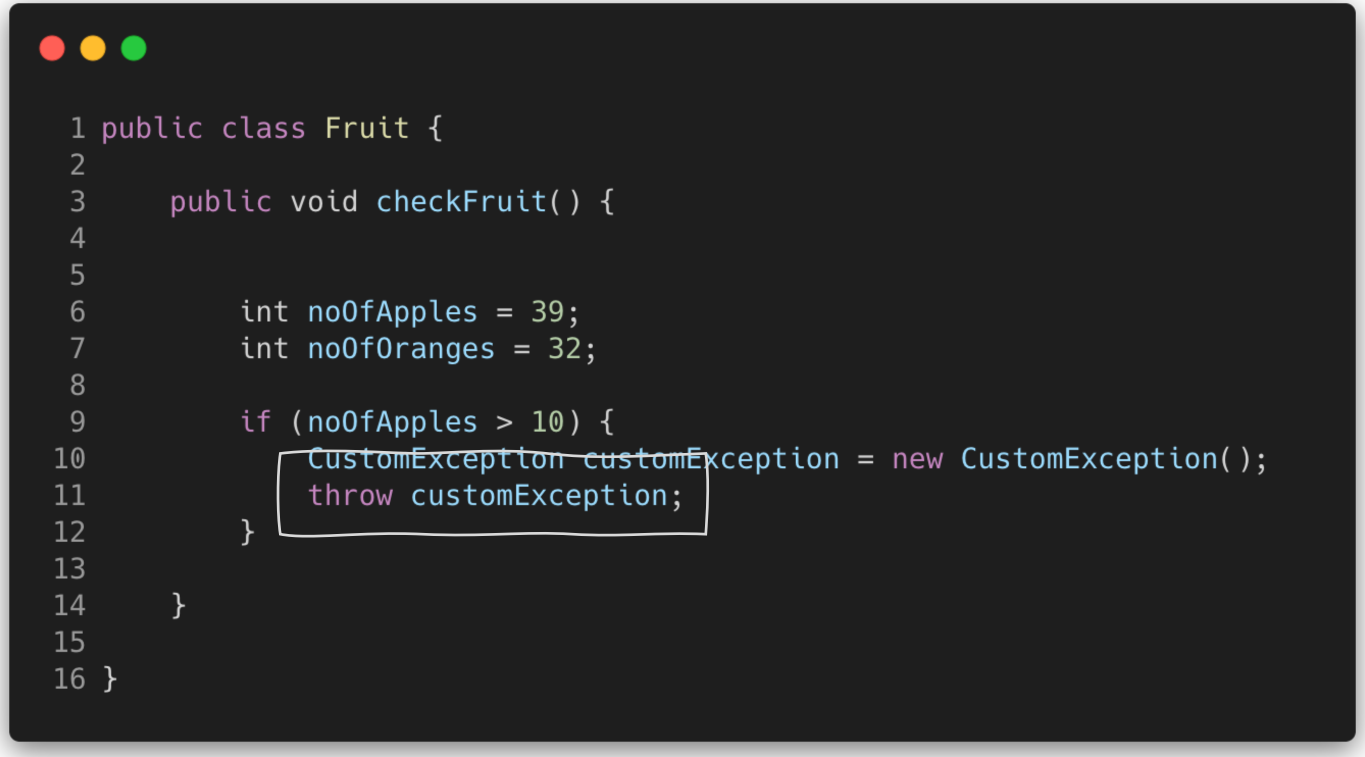
- Once we have defined our own exception, we need to create a new object.
- In our case, we need to create an object from the class CustomException.
- We create the object when we wish to raise the exception.
- In the example below, we wish to throw a CustomException when the value noOfApples is greater than 10.



```java
public class Fruit {

    public void checkFruit() {


        int noOfApples = 39;
        int noOfOranges = 32;

        if (noOfApples > 10) {
            CustomException customException = new CustomException();
        }

    }

}
```

USER-DEFINED EXCEPTIONS

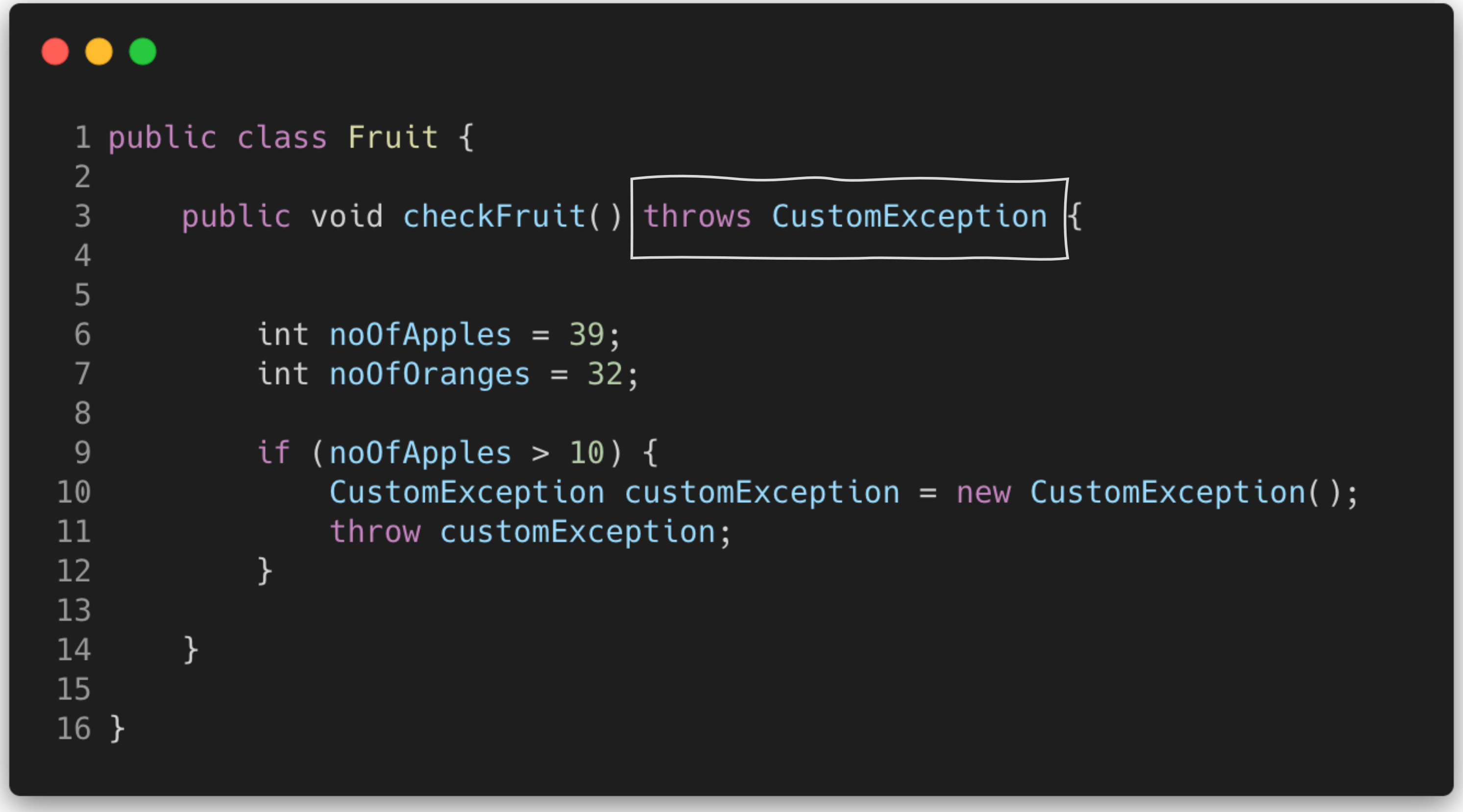
- **After our CustomException has been raised, we need to look for a user-defined exception handler that matches CustomException.**
- **We can use the keyword throw so that Java can catch the exception if the relevant handler has been defined.**



```
public class Fruit {

    public void checkFruit() {


        int noOfApples = 39;
        int noOfOranges = 32;

        if (noOfApples > 10) {
            CustomException customException = new CustomException();
            throw customException;
        }

    }

}
```

USER-DEFINED EXCEPTIONS

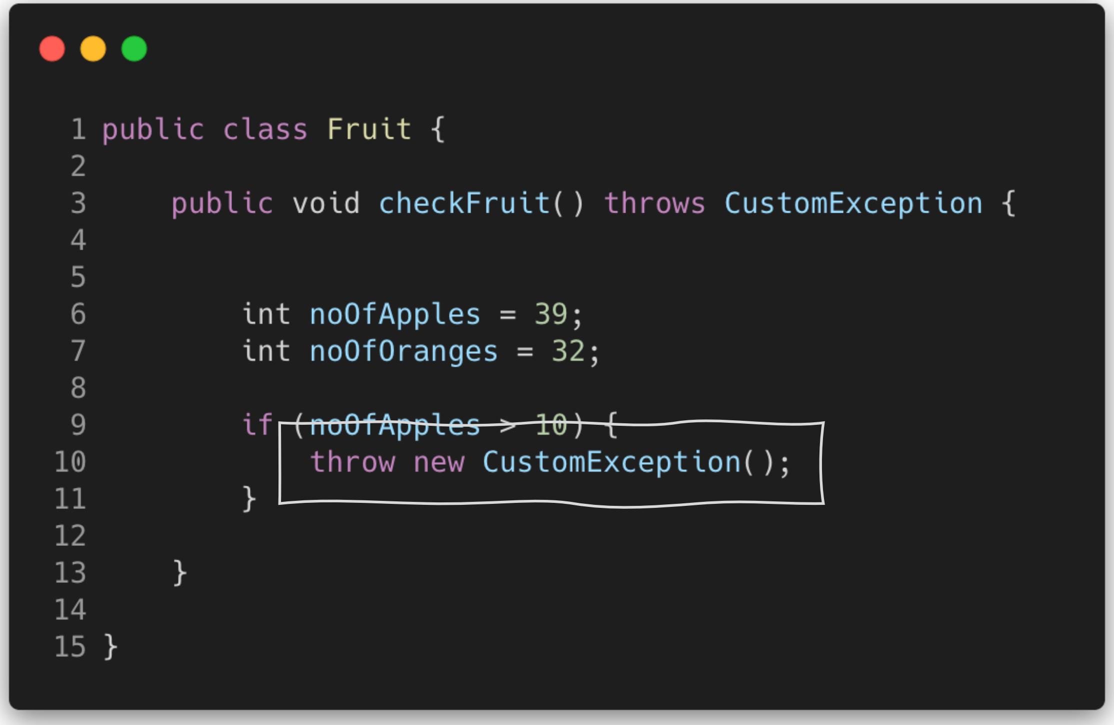
- This code is still not valid. We need to declare that the method checkFruit() may throw a CustomException.



```java
public class Fruit {

    public void checkFruit() throws CustomException {


        int noOfApples = 39;
        int noOfOranges = 32;

        if (noOfApples > 10) {
            CustomException customException = new CustomException();
            throw customException;
        }

    }

}
```

USER-DEFINED EXCEPTIONS

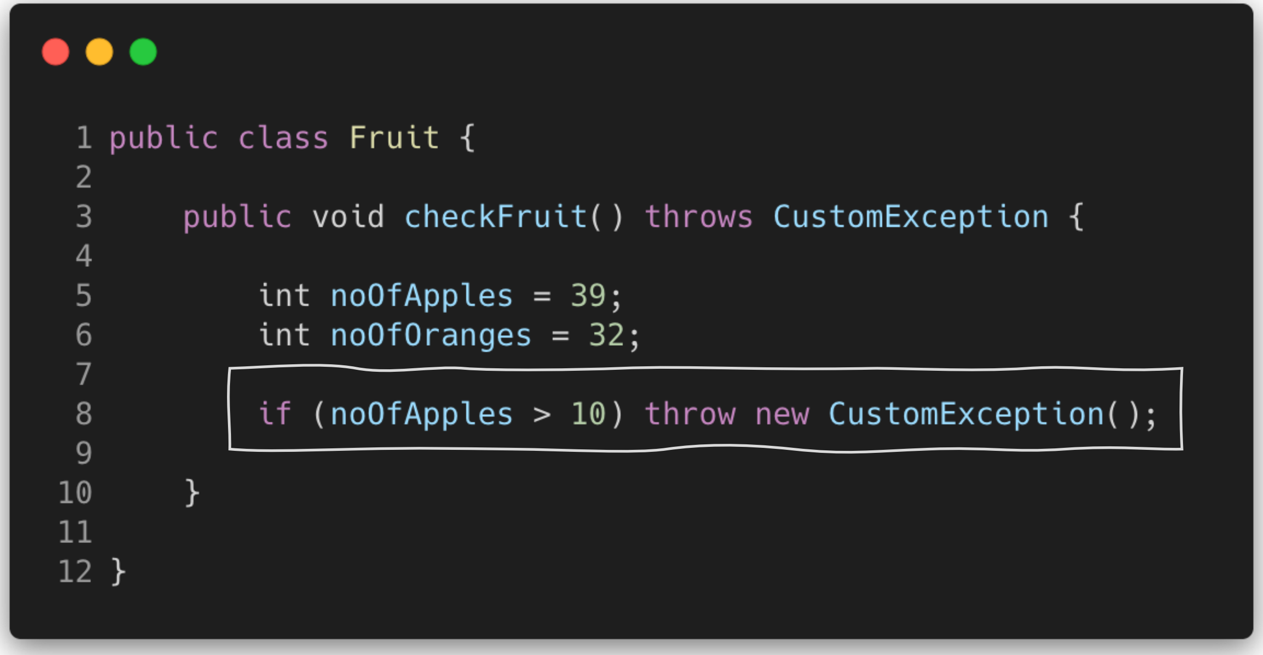
- **We can shorten this code by creating a new object of CustomException and throwing it on the same line.**



```
public class Fruit {

    public void checkFruit() throws CustomException {


        int noOfApples = 39;
        int noOfOranges = 32;

        if (noOfApples > 10) {
            throw new CustomException();
        }

    }

}
```

USER-DEFINED EXCEPTIONS (ASIDE)

- An aside: we can shorten a conditional statement if there is only one line in its satisfying block of code.
- We can now read Line 8 in an almost-English-like language now: "If the number of apples is greater than 10, throw a new CustomException".



```
 1 public class Fruit {
 2
 3     public void checkFruit() throws CustomException {
 4
 5         int noOfApples = 39;
 6         int noOfOranges = 32;
 7
 8         if (noOfApples > 10) throw new CustomException();
 9
10     }
11
12 }
```

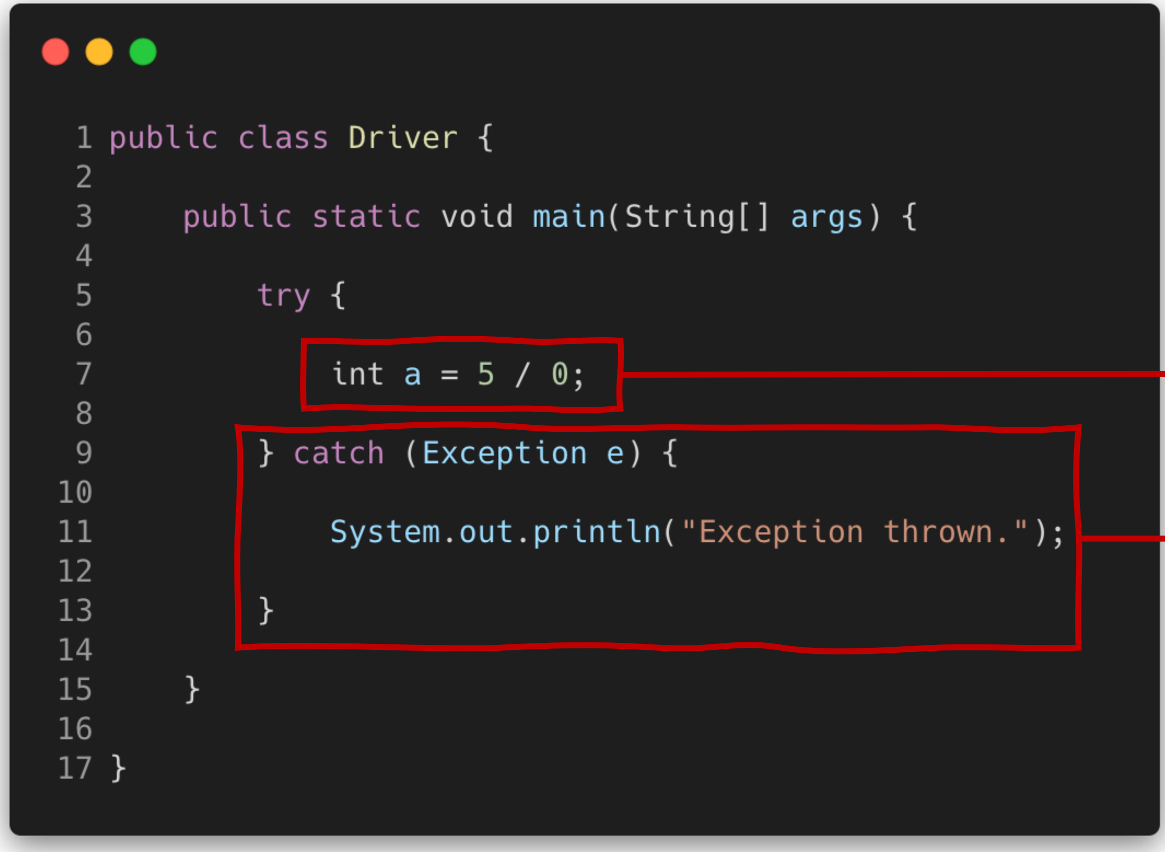
Exception Handler Ordering

EXCEPTION HANDLER ORDERING

- All exceptions inherit the class Exception.
- For example, ArithmeticException inherits Exception.
- ArrayOutOfBoundsException inherits Exception.
- Our CustomException inherits Exception.
- In other words, ArithmeticException, ArrayOutofBoundsException, CustomException etc. IS-A Exception.

EXCEPTION HANDLER ORDERING

- This example shows that ArithmeticException is caught by an Exception handler.
- This works because Exception is a parent of ArithmeticException.



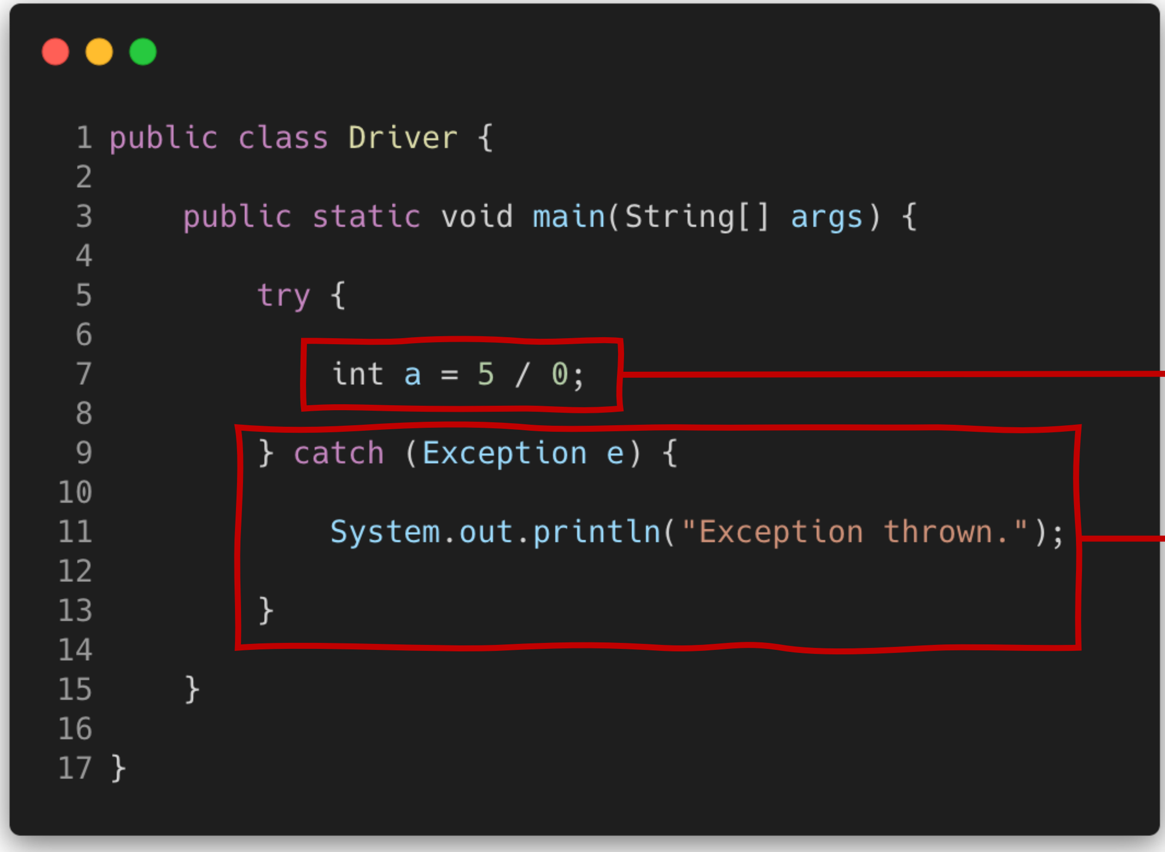
```java
public class Driver {

    public static void main(String[] args) {

        try {

            int a = 5 / 0;

        } catch (Exception e) {

            System.out.println("Exception thrown.");

        }

    }

}
```

This code will throw an ArithmeticException.

This block of code WILL run.

EXCEPTION HANDLER ORDERING

- It transpires that an Exception handler will catch all types of Exceptions.
- This is why although an ArithmeticException was thrown, the Exception handler still ran.



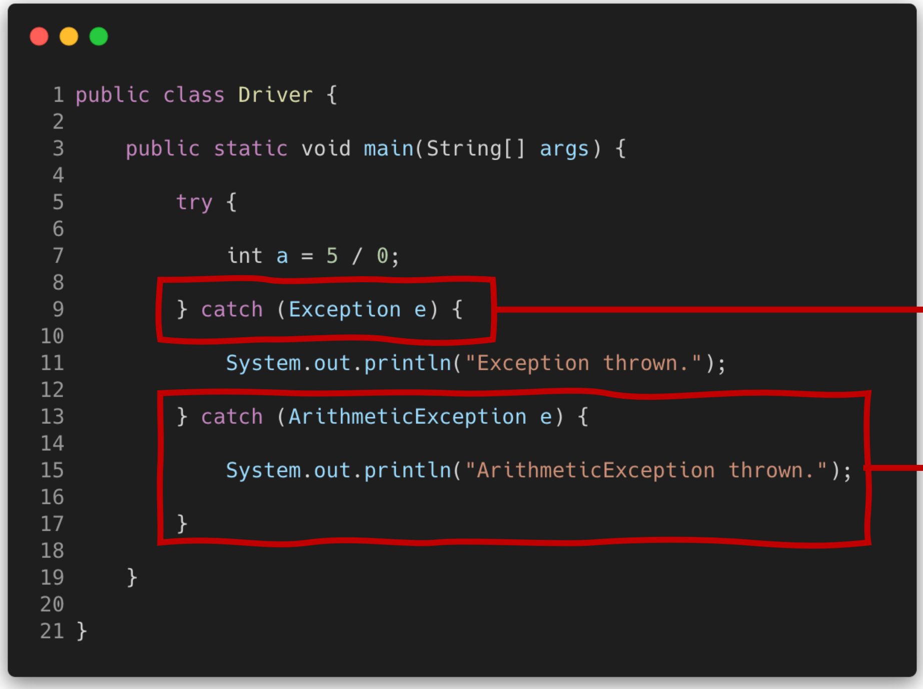
```
public class Driver {

    public static void main(String[] args) {

        try {

            int a = 5 / 0;

        } catch (Exception e) {

            System.out.println("Exception thrown.");

        }

    }

}
```

This code will throw an ArithmeticException.

This block of code WILL run.

EXCEPTION HANDLER ORDERING

- We can have more than one exception handler. The order matters though.
- In this example, we have placed the ArithmeticException handler after the Exception handler.
- Once a handler is run, it will skip all subsequent handlers.
- This code will not compile because the ArithmeticException handler is unreachable. It cannot be reached because all exceptions will be caught by the Exception handler.



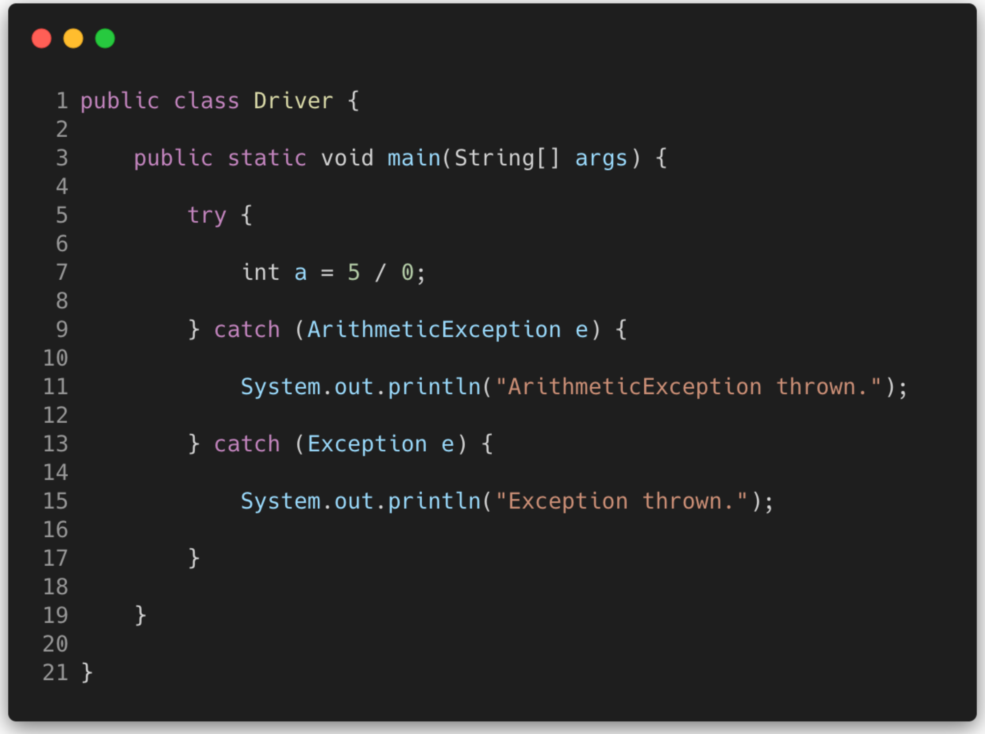
```java
public class Driver {

    public static void main(String[] args) {

        try {

            int a = 5 / 0;

        } catch (Exception e) {

            System.out.println("Exception thrown.");

        } catch (ArithmeticException e) {

            System.out.println("ArithmeticException thrown.");

        }

    }

}
```

This handler catches all errors, so this handler will be run.

This handler is unreachable.

EXCEPTION HANDLER ORDERING

- Instead it will make sense to have the Exception handler as the last one.
- This means that if an exception is thrown, for instance ArrayOutOfBoundsException, and we have no implemented a handler for it, the generic Exception handler will be thrown instead and we avoid crashing the program.

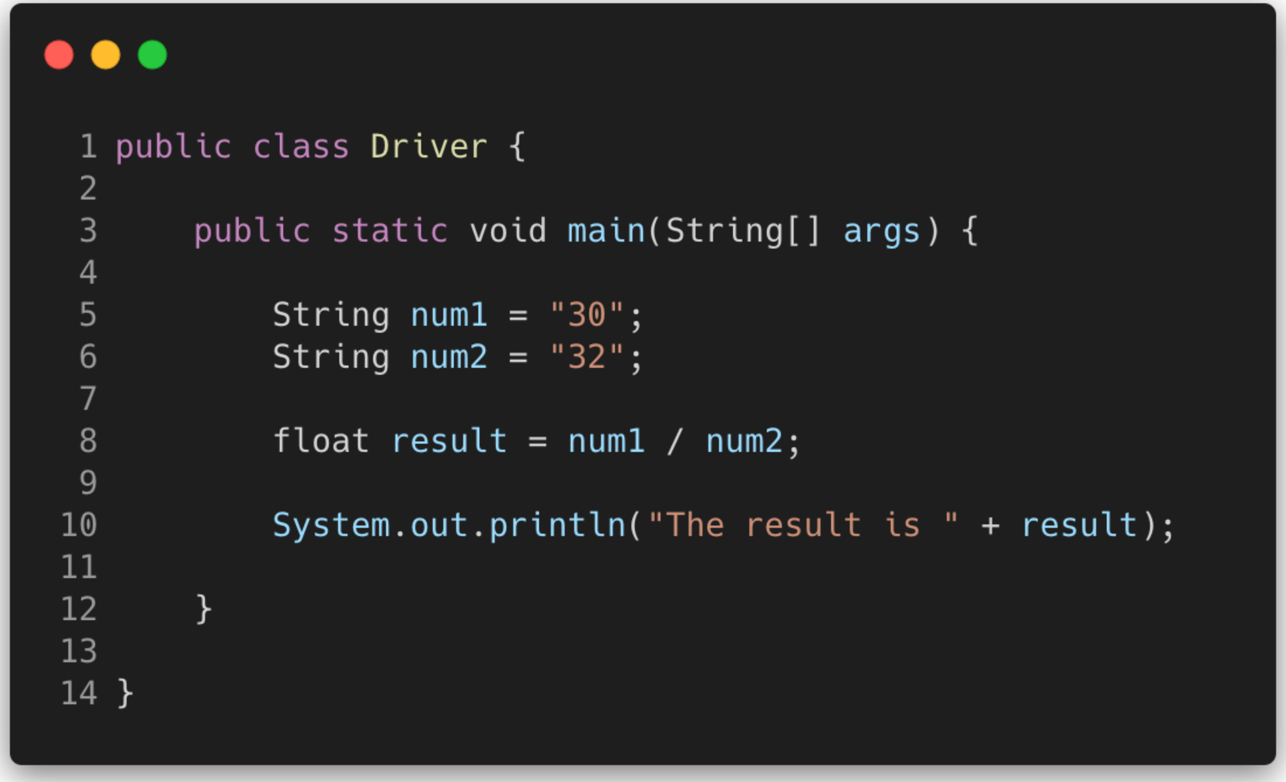


```java
public class Driver {

    public static void main(String[] args) {

        try {

            int a = 5 / 0;

        } catch (ArithmeticException e) {

            System.out.println("ArithmeticException thrown.");

        } catch (Exception e) {

            System.out.println("Exception thrown.");

        }

    }

}
```

Converting a String to Number

CONVERTING STRINGS TO NUMBERS

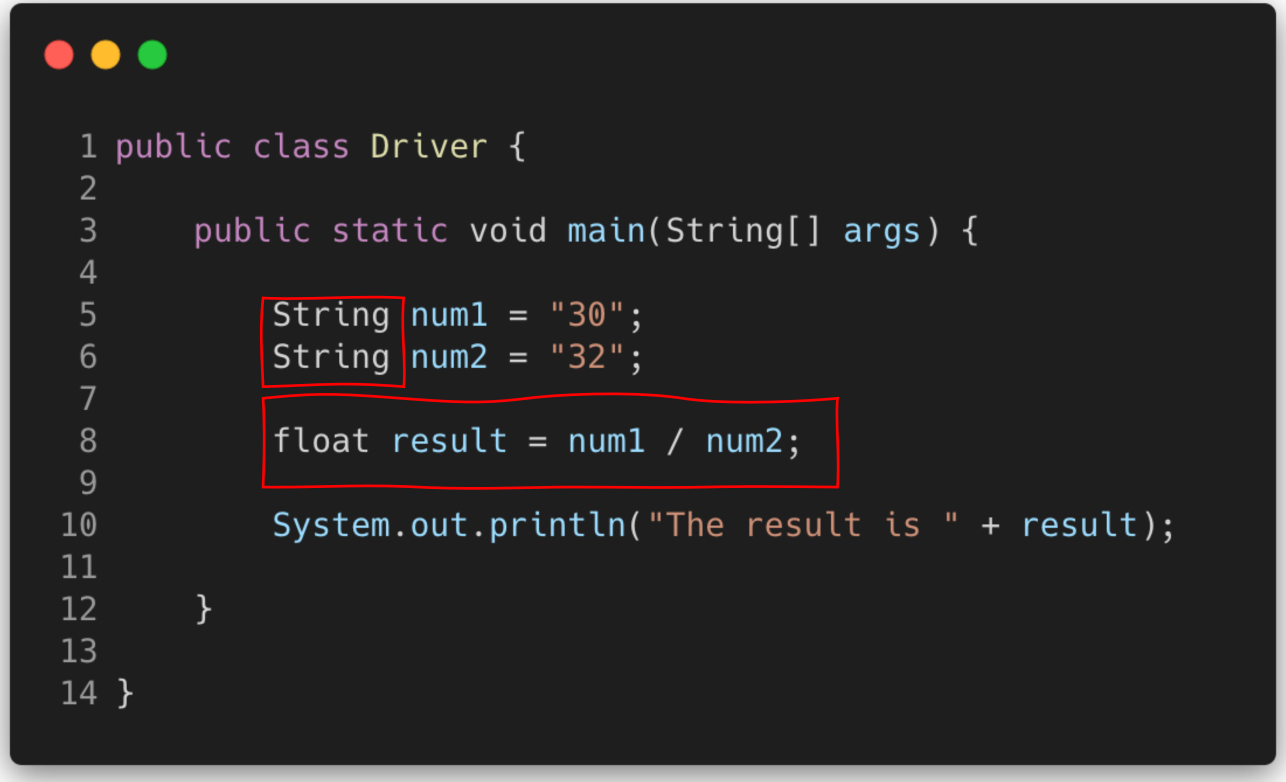
- **Is this code valid?**



```java
public class Driver {

    public static void main(String[] args) {

        String num1 = "30";
        String num2 = "32";

        float result = num1 / num2;

        System.out.println("The result is " + result);

    }

}
```

CONVERTING STRINGS TO NUMBERS

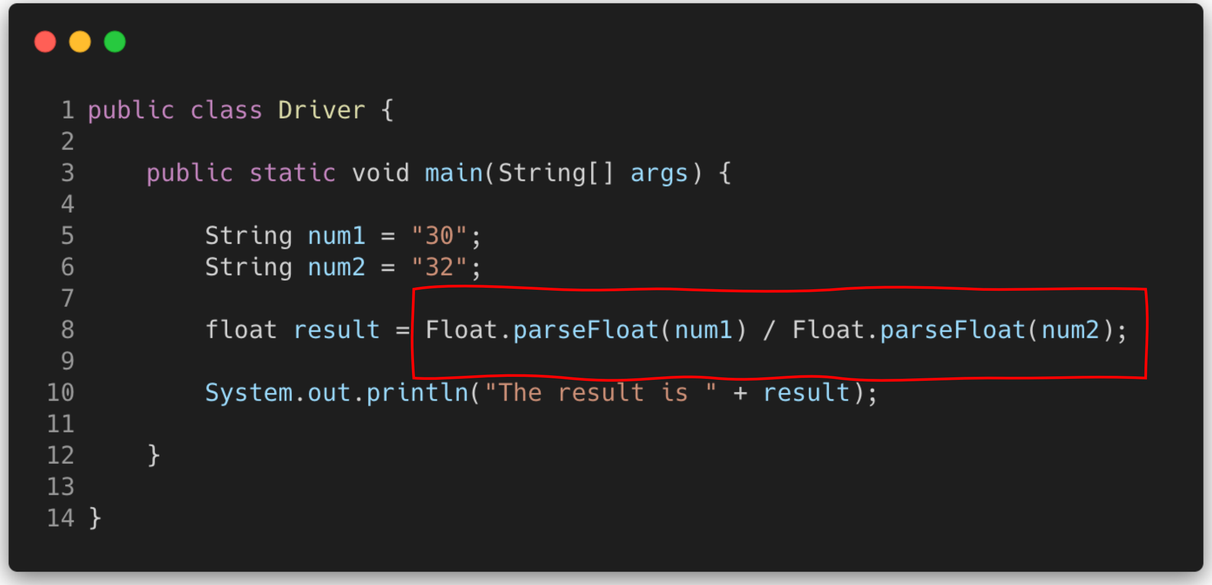
- **No, we can’t divide two strings and assign it to an int variable.**



```java
public class Driver {

    public static void main(String[] args) {

        String num1 = "30";
        String num2 = "32";

        float result = num1 / num2;

        System.out.println("The result is " + result);

    }

}
```

CONVERTING STRINGS TO NUMBERS

- We need to convert num1 and num2 from strings to a numeric type, one that we can do mathematical calculations on.
- Because result is a float, we want to convert num1 and num2 to a float.
- Java provides a large number of built-in classes to do common operations such as these.
- To convert a String to a float, we can use `Float.parseFloat(x)` where x is a String.



```java
public class Driver {

    public static void main(String[] args) {

        String num1 = "30";
        String num2 = "32";

        float result = Float.parseFloat(num1) / Float.parseFloat(num2);

        System.out.println("The result is " + result);

    }

}
```

CONVERTING STRINGS TO NUMBERS

- **Why is it that we can call `Float.parseFloat(num1)` without first creating an object from class Float?**
- **Because parseFloat is a static method and pertains to the class Float, not an object of Float.**

CONVERTING STRINGS TO NUMBERS

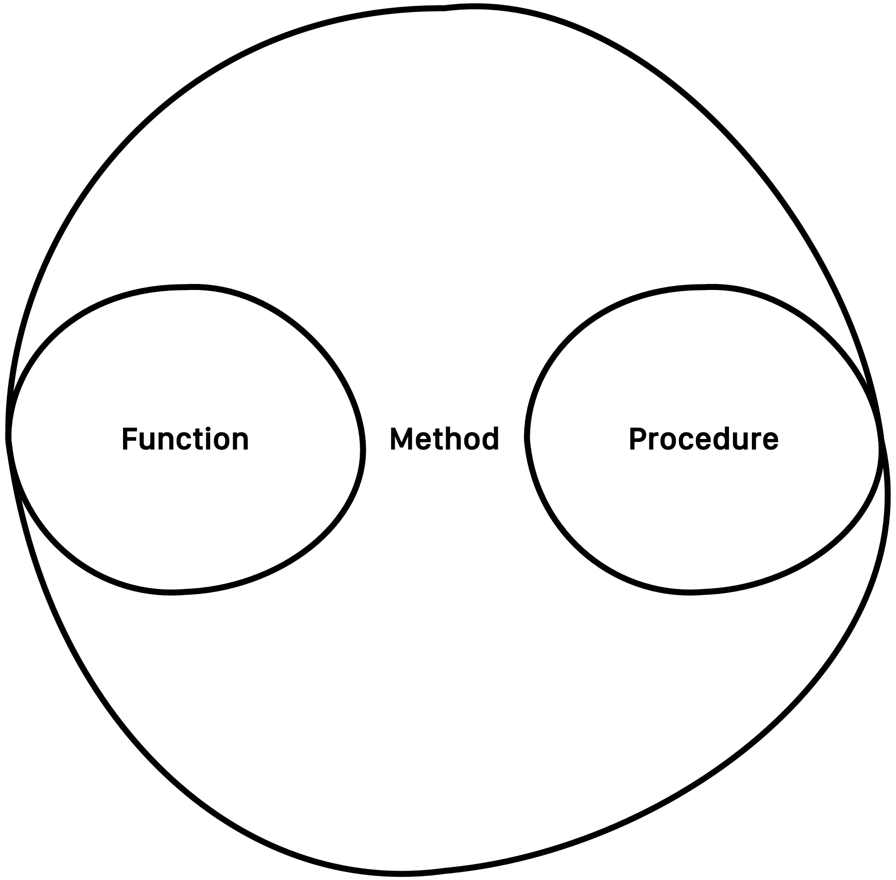
- **There are other built-in methods that allow you to convert from various types.**

Float.parseFloat(x)	Convert string x to float.
Integer.parseInt(x)	Convert string x to int.
Double.parseDouble(x)	Convert string x to double.
Long.parseLong(x)	Convert string x to long.
String.valueOf(x)	Convert numeric variable x to a string.

Methods vs. Functions vs. Procedures

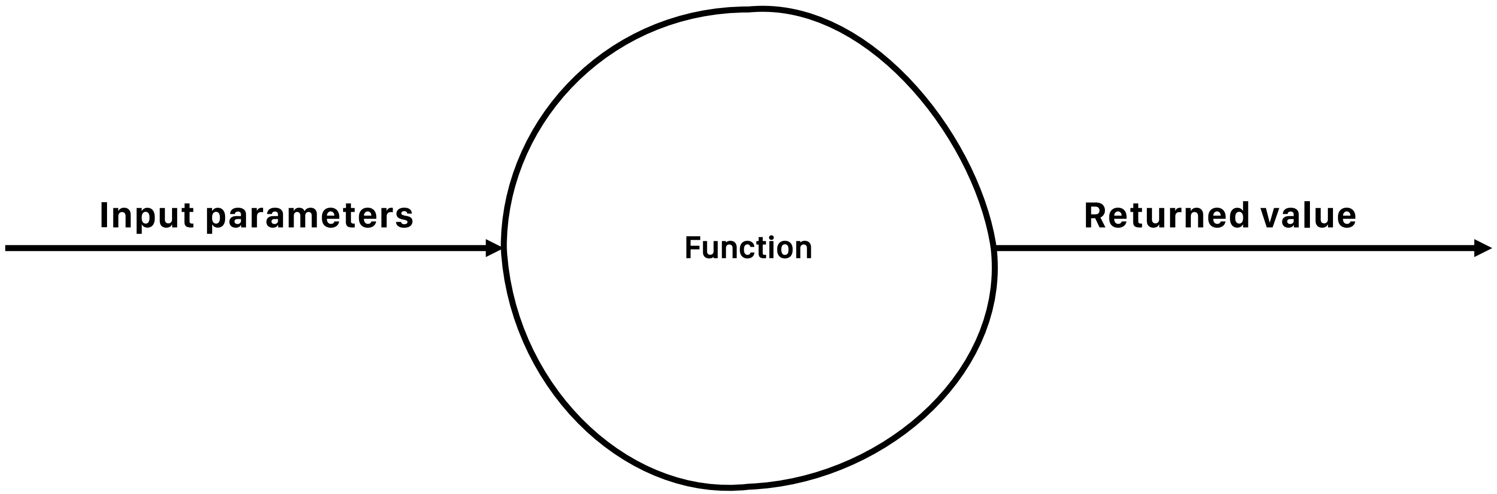
METHODS VS. FUNCTIONS VS. PROCEDURES

- **Functions and Procedures are both types of Methods.**



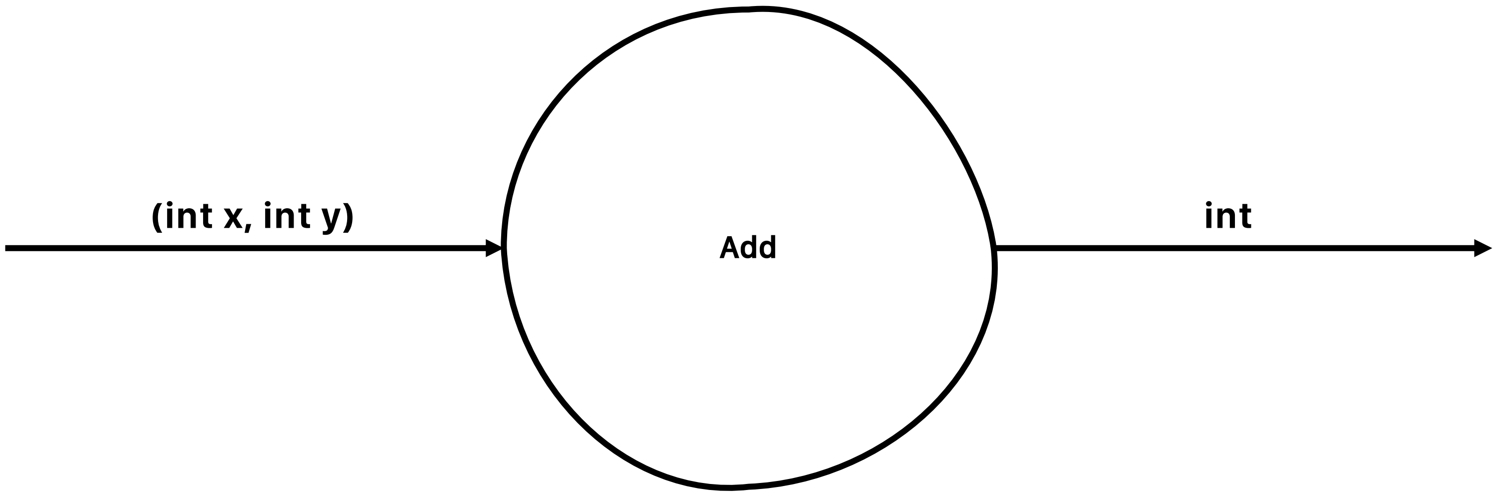
METHODS VS. FUNCTIONS VS. PROCEDURES

- **A Function returns a value.**



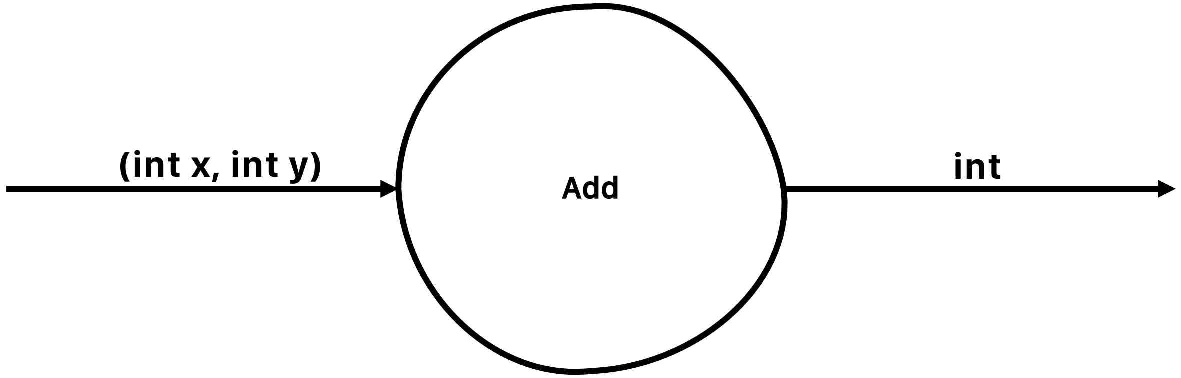
METHODS VS. FUNCTIONS VS. PROCEDURES

- For example, a function add will take two values as input parameters, and return a value.
- In the example below, we take two integers x and y, and specify that this function will return an int.

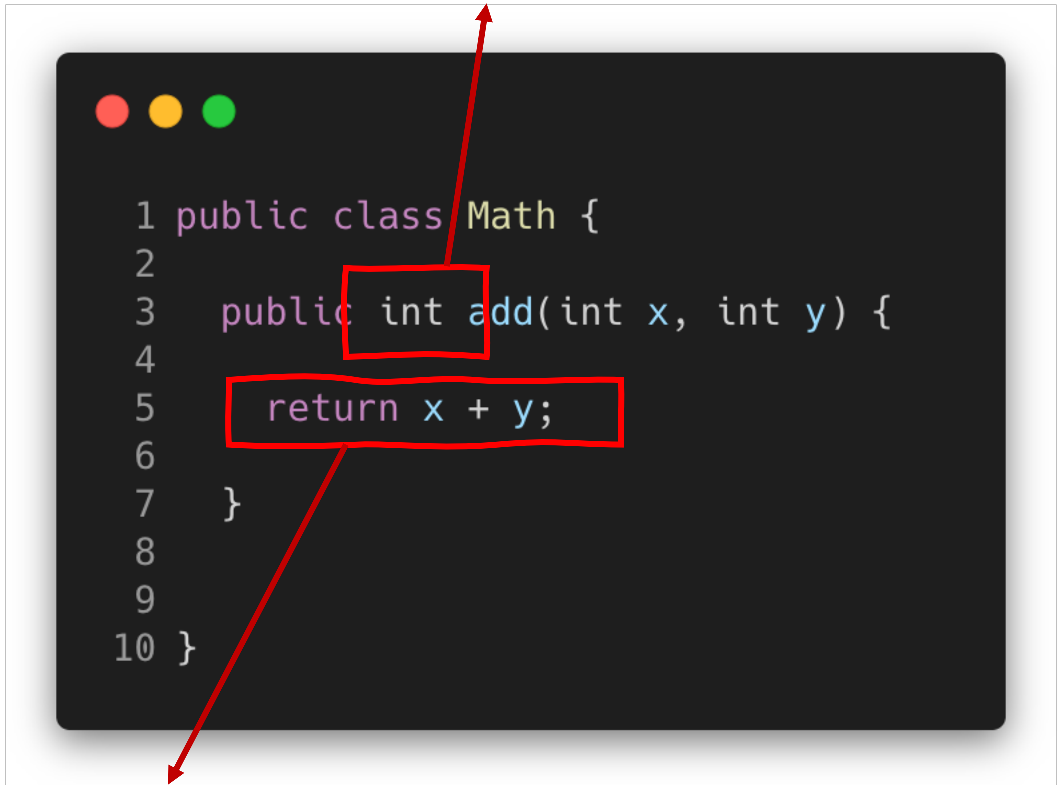


METHODS VS. FUNCTIONS VS. PROCEDURES

- When we declare a function, we must declare a return type.
- That is, we must declare what type of value will be output from the function.
- For example, in the add example:



We declare the return type here.

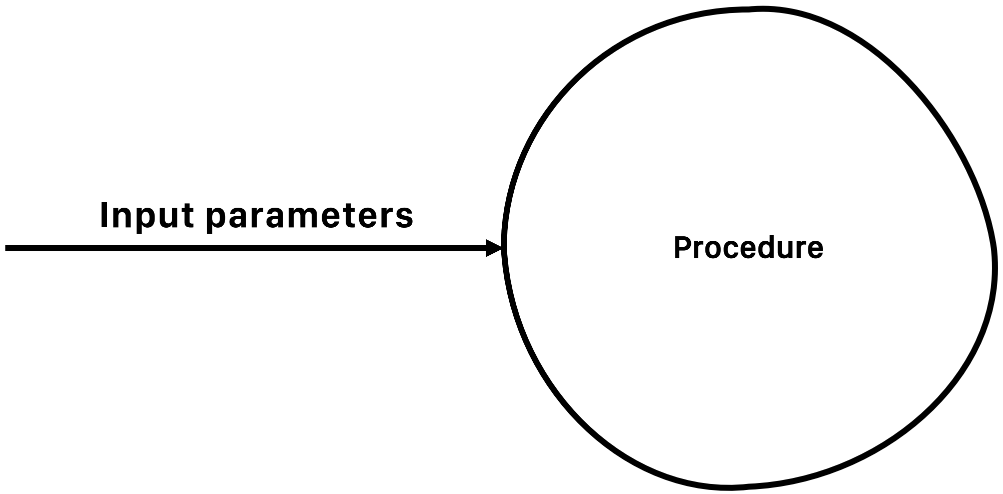


```
public class Math {

  public int add(int x, int y) {

    return x + y;

  }



}
```

The return keyword provides the output of the function.

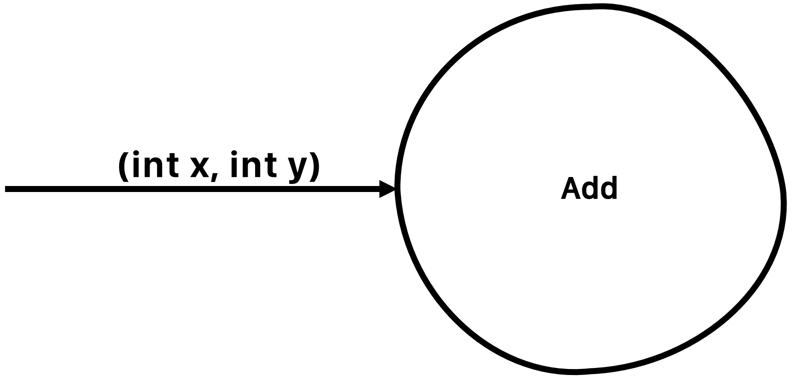
METHODS VS. FUNCTIONS VS. PROCEDURES

- **A Procedure does not return a value.**

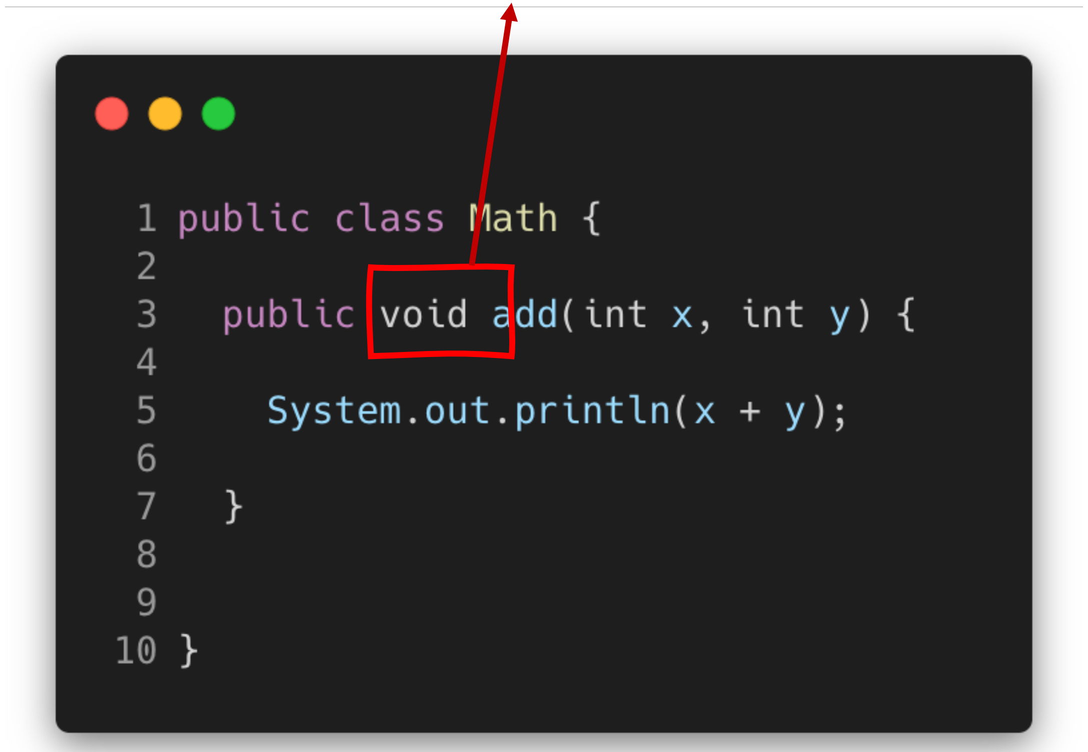


METHODS VS. FUNCTIONS VS. PROCEDURES

- **We can have an add procedure that simply printed the value instead of returning a value.**
- **In this case, no value is returned so we declare the return type as void.**



This method does not return a value.

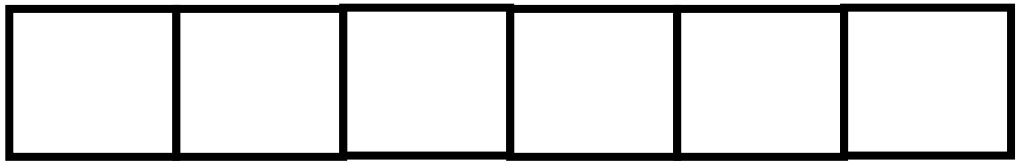


```java
public class Math {

  public void add(int x, int y) {

    System.out.println(x + y);

  }


}
```

Arrays

ARRAYS

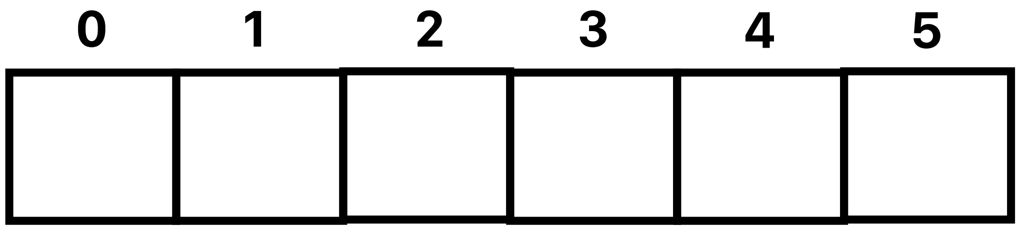
- We’ve looked at how to store values in variables.
- A variable can store a single reference to a value or object.
- However, what if we wanted to store more than one value or object in a group?
- We can do this using arrays.
- Arrays can be thought of as a group of variables.
- Arrays can be visualized like this:



- This array can hold 6 items.

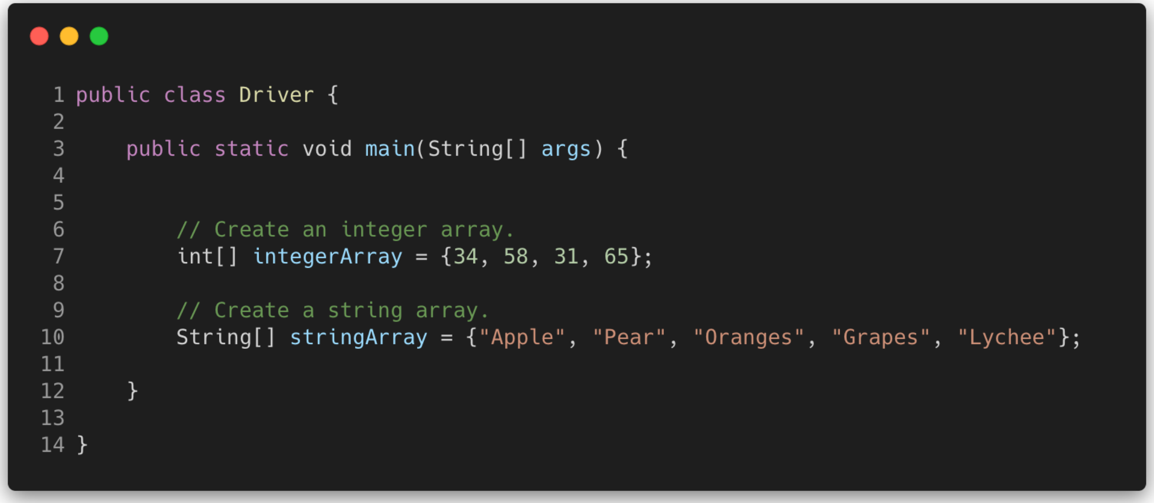
ARRAYS

- Each cell in an array has an index number.
- The index is a number assigned to a cell of an array.
- Indexes start at 0 and increment by 1 for each cell.
- The diagram below shows the indexes of the cells of an array.



ARRAYS

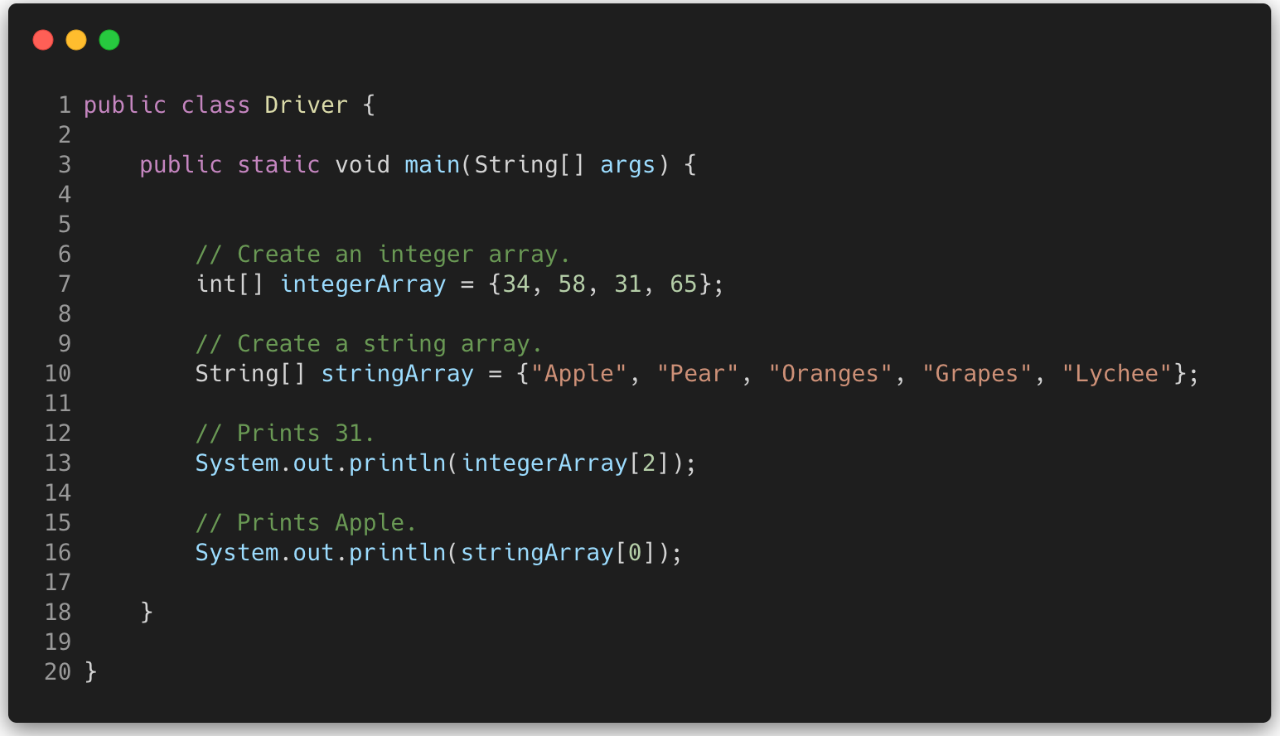
- We can use the [] notation to specify an array type.
- Every array requires a type that denotes the type of elements that will be stored within in.
- For instance, if we want to store integers in an array, the type of the array we need is int[].
- The example below shows how we can create arrays in Java.



```java
public class Driver {

    public static void main(String[] args) {


        // Create an integer array.
        int[] integerArray = {34, 58, 31, 65};

        // Create a string array.
        String[] stringArray = {"Apple", "Pear", "Oranges", "Grapes", "Lychee"};

    }

}
```

ARRAYS

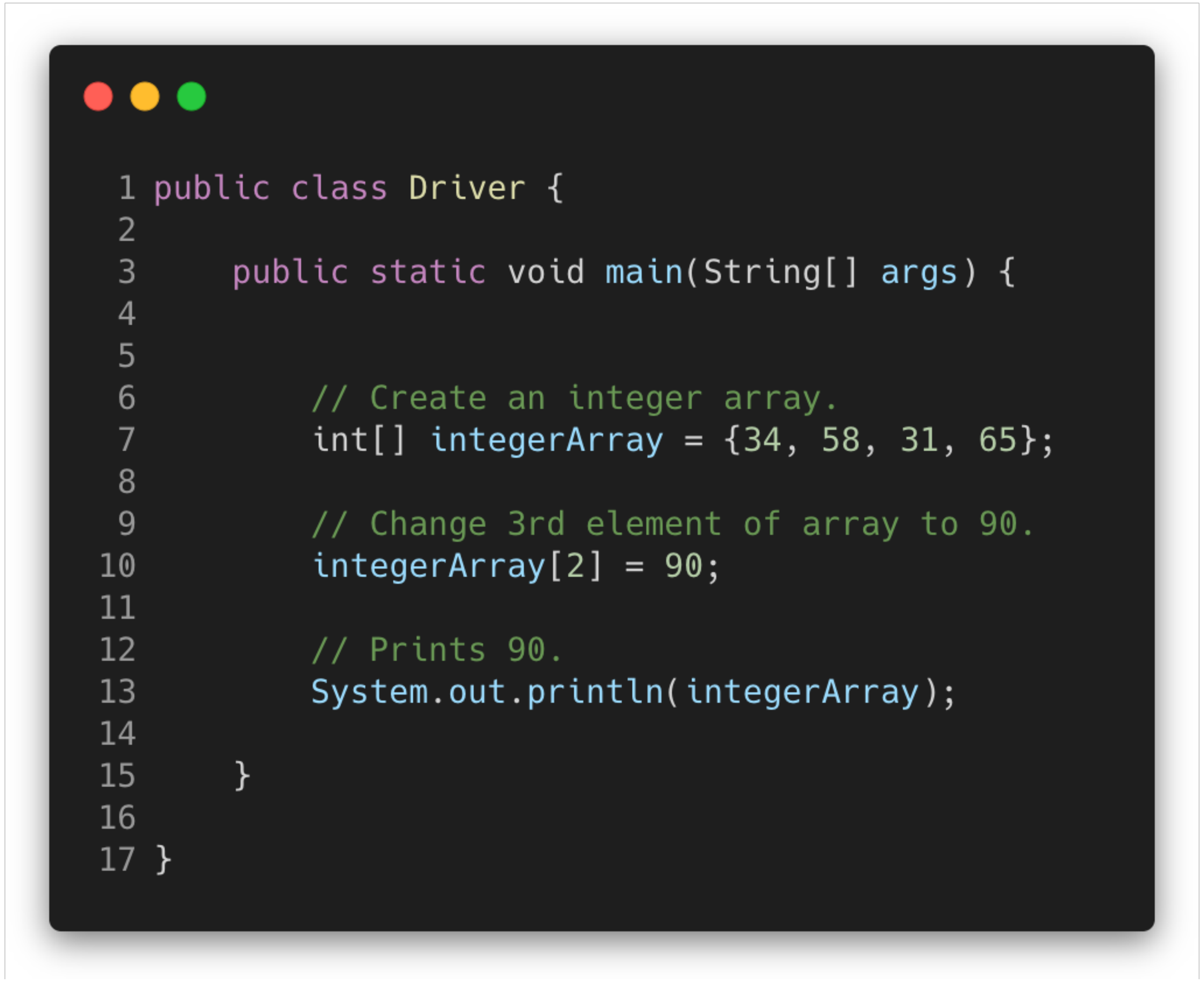
- We can access a value stored in an array by using the index number.
- For instance, if we wanted to access the 3rd element of the integerArray, we can use integerArray[2].
- Notice 2 in the square brackets. This refers to the 3rd element because the first index is 0.



```java
public class Driver {

    public static void main(String[] args) {


        // Create an integer array.
        int[] integerArray = {34, 58, 31, 65};

        // Create a string array.
        String[] stringArray = {"Apple", "Pear", "Oranges", "Grapes", "Lychee"};

        // Prints 31.
        System.out.println(integerArray[2]);

        // Prints Apple.
        System.out.println(stringArray[0]);

    }

}
```

ARRAYS

- **We can set individual cells of an array.**

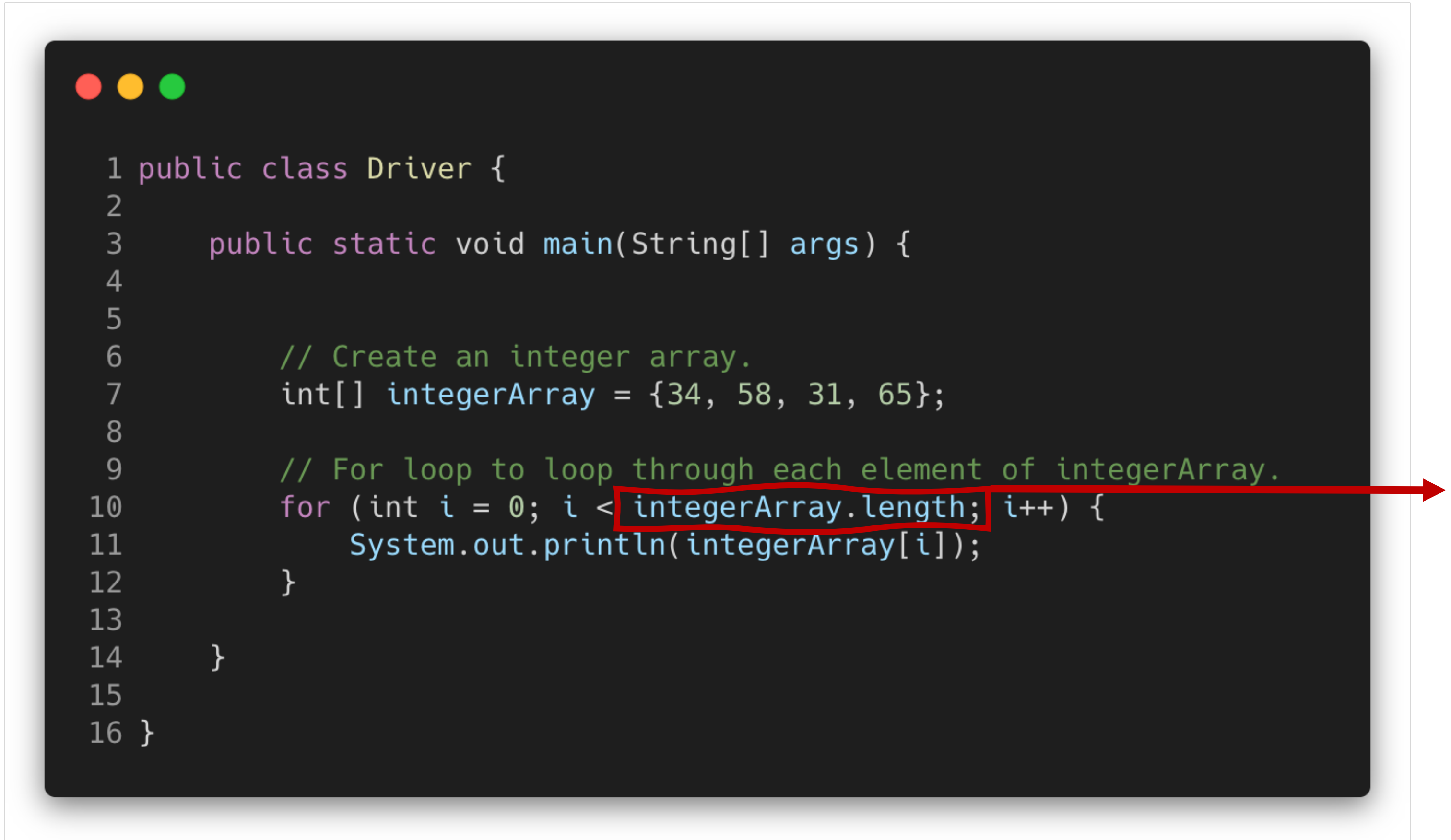


```java
public class Driver {

    public static void main(String[] args) {


        // Create an integer array.
        int[] integerArray = {34, 58, 31, 65};

        // Change 3rd element of array to 90.
        integerArray[2] = 90;

        // Prints 90.
        System.out.println(integerArray);

    }

}
```

Looping through an Array

LOOPING THROUGH AN ARRAY

- We can combine our knowledge of loops and arrays to loop through an array.
- For each cell of an array, we can take some take some action upon it.
- Let’s say we want to print all values of an array one by one.
- We can use a for loop to iterate through each value of an array.



```java
public class Driver {

    public static void main(String[] args) {


        // Create an integer array.
        int[] integerArray = {34, 58, 31, 65};

        // For loop to loop through each element of integerArray.
        for (int i = 0; i < integerArray.length; i++) {
            System.out.println(integerArray[i]);
        }

    }

}
```

Arrays have a data member length which specifies the number of elements within an array.