Foundations of Programming 2: Exceptions

FoP 2 Teaching Team, Faculty of Computer Science, Universitas Indonesia
Correspondence: Fariz Darari (fariz@cs.ui.ac.id)

[Handle mistakes, in an elegant way]

Why?

Despite all the efforts,
our code might still throw problems (= exceptions).

Are we going to let our program crash?

Why?

Despite all the efforts,
our code might still throw problems (= exceptions).

Are we going to let our program crash?

Of course not!

What's the matter?

```
System.out.println(5/0);
```

What's the matter?

```
System.out.println(5/0);
```

```
Exception in thread "main"
java.lang.ArithmeticException: / by zero
```

What's the matter?

```
System.out.println("Ashhhiaaaaapp!".charAt(100));
```

What's the matter?

```
System.out.println("Ashhhiaaaaapp!".charAt(100));
```

Exception in thread "main"
negeri.+62.AlayException

What's the matter?

```
System.out.println("Ashhhiaaaaapp!".charAt(100));
```

```
Exception in thread "main"
java.lang.StringIndexOutOfBoundsException:
String index out of range: 100
```

Quiz time: Make two exceptions!

Quiz time: Make two exceptions!

```
List<String> l = null; l.isEmpty();
```

Quiz time: Make two exceptions!

```
List<String> l = null; l.isEmpty();
```

```
Exception in thread "main" java.lang.NullPointerException
```

Quiz time: Make two exceptions!

```
List<String> l = null; l.isEmpty();
```

```
Exception in thread "main" java.lang.NullPointerException
```

```
int[] arr = new int[-5];
```

Quiz time: Make two exceptions!

```
List<String> l = null; l.isEmpty();
```

```
Exception in thread "main" java.lang.NullPointerException
```

```
int[] arr = new int[-5];
```

```
Exception in thread "main"
java.lang.NegativeArraySizeException: -5
```

Quiz time: Can you create a code throwing `java.lang.StackOverflowError`?

Quiz time: Can you create a code throwing `java.lang.StackOverflowError`?

Calling `f()` when `f()` is defined as below:

```
public static void f() {
    f();
}
```

About exceptions

- The ideal time to catch an error is at compile time.
- However, not all errors can be detected at compile time.
- Improved error recovery is one of the most powerful ways to increase the robustness of your code.
- *To create a robust system, each component must be robust*.
- By providing an error-reporting model using exceptions, Java allows components to reliably communicate problems to client code.

Flow of code execution

- Normal main sequential code execution, the program doing what it meant to accomplish.
- Exception handling code execution, the main program flow was interrupted by an error or some other condition that prevent the continuation of the normal main sequential code execution.

Division code: Can you break this code?

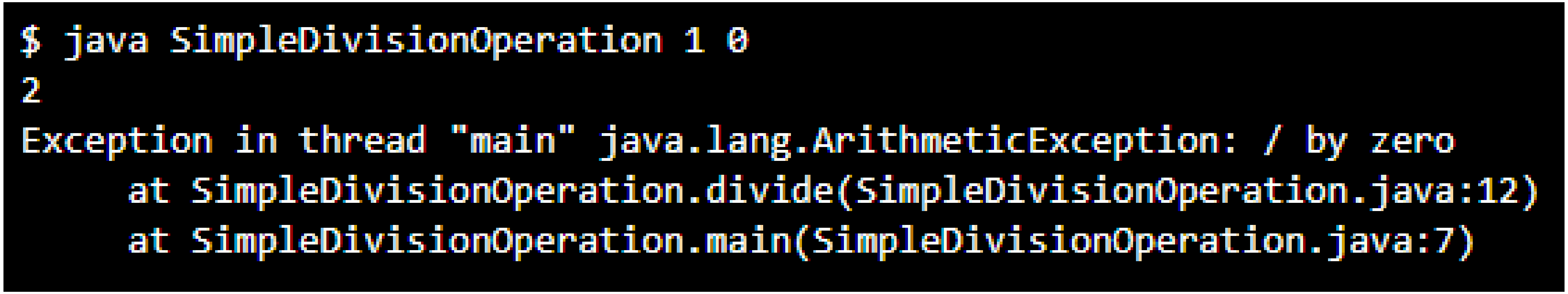
```
public class SimpleDivisionOperation {
  public static void main(String[] args) {
    System.out.println(divide(4, 2));
    if (args.length > 1) {
      int arg0 = Integer.parseInt(args[0]);
      int arg1 = Integer.parseInt(args[1]);
      System.out.println(divide(arg0, arg1));
    }
  }

  public static int divide(int a, int b) {
    return a / b;
  }
}
```

Division by zero

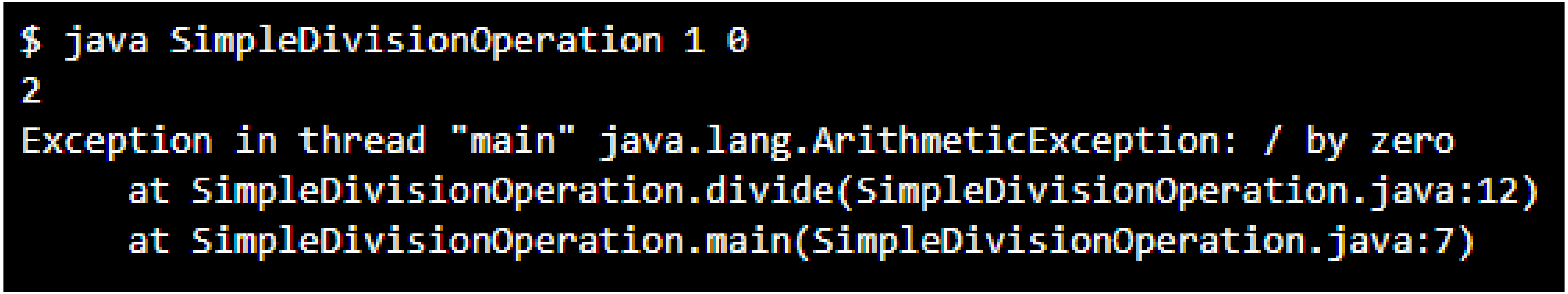
- The program defines a method divide that does a simple division.
- It can safely be assumed that when the divide(4, 2) statement is called, it would return the number 2.
- However, consider the next statement, where the program relies upon the provided command line arguments to generate a division operation.

  What if the user provides the number zero (0) as the second argument?

Division by zero


```
$ java SimpleDivisionOperation 1 0
2
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at SimpleDivisionOperation.divide(SimpleDivisionOperation.java:12)
        at SimpleDivisionOperation.main(SimpleDivisionOperation.java:7)
```

Division by zero


```
$ java SimpleDivisionOperation 1 0
2
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at SimpleDivisionOperation.divide(SimpleDivisionOperation.java:12)
        at SimpleDivisionOperation.main(SimpleDivisionOperation.java:7)
```

The **stack trace** in the above example tells us more about the error, such as:

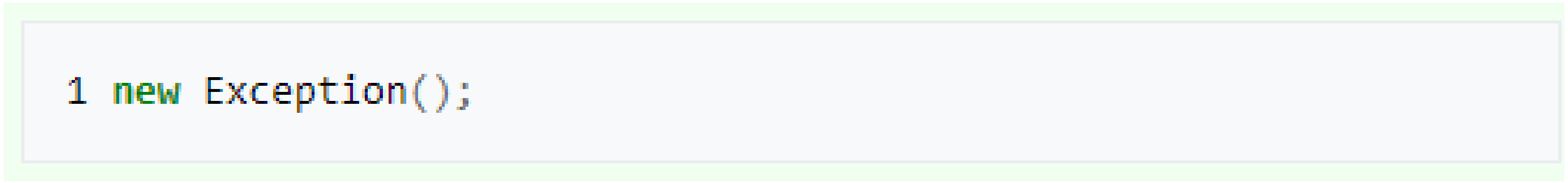
- the thread — **"main"** — where the exception occurred,
- the type of exception — **java.lang.ArithmeticException**,
- a readable display message — **/ by zero**, and
- the exact methods and the line numbers where the exception occurred.

Throwing an exception object explicitly

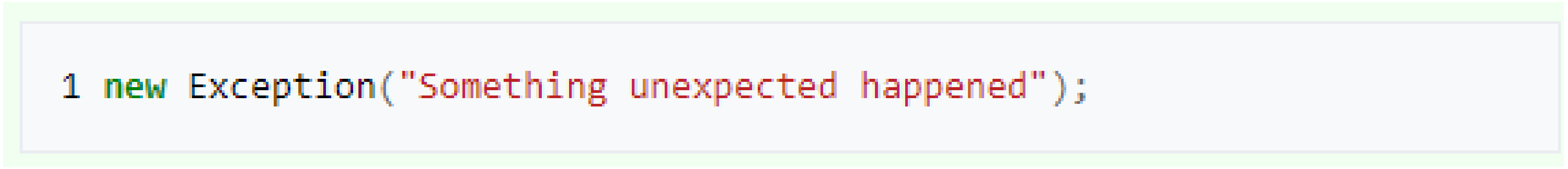
```
public class SimpleDivisionOperation {
 // ...

  public static int divide(int a, int b) {
    if (b == 0) {
      throw new ArithmeticException("OMG!");
    } else {
      return a / b;
    }
  }
}
```

Standard exceptions have two constructors

1. The default constructor; and,
2. A constructor taking a string argument so that you can place pertinent information in the exception.



```java
new Exception();
```



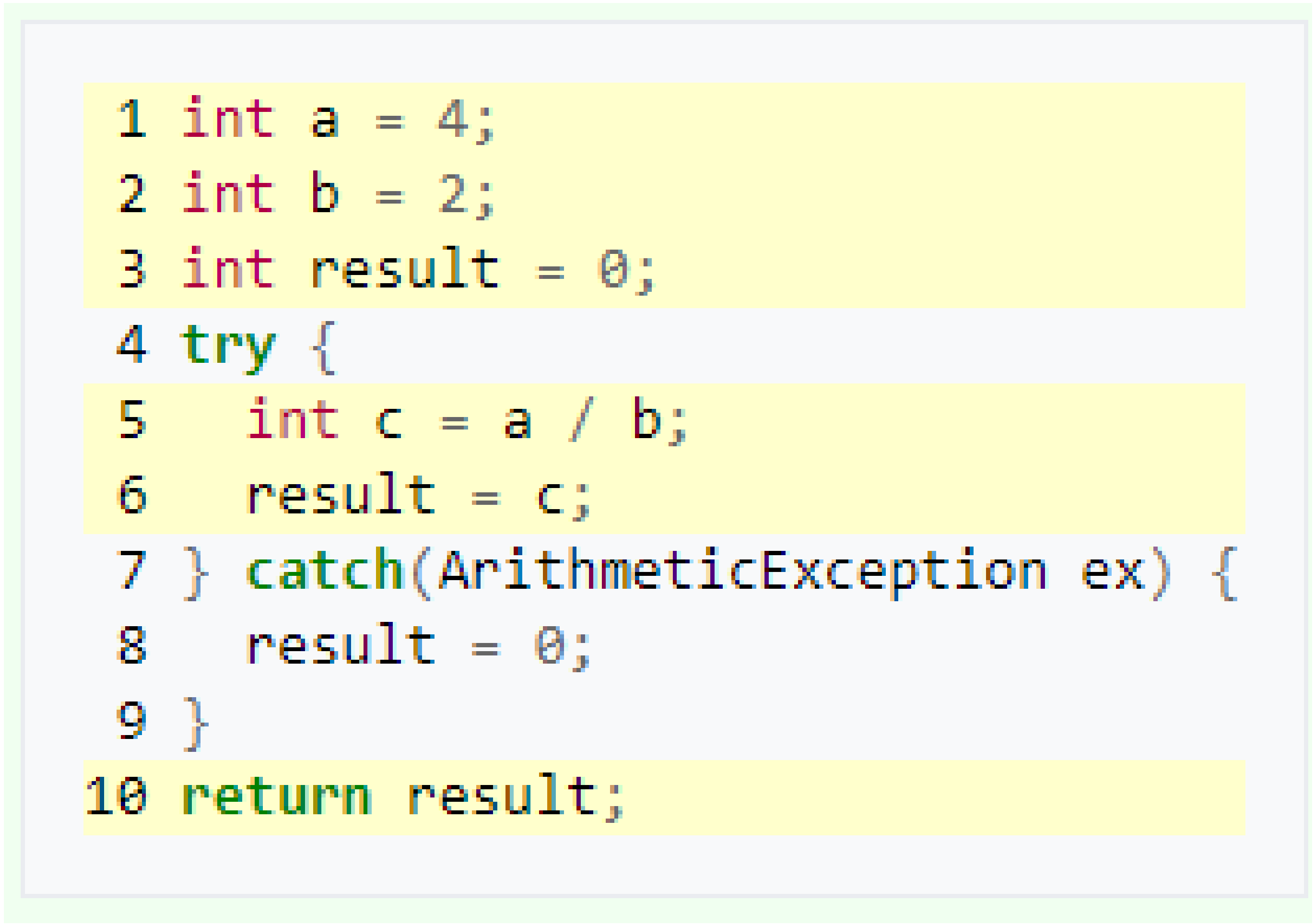
```java
new Exception("Something unexpected happened");
```

try/catch statement

By default, when an exception is thrown,
the current method is interrupted, the calling method
is interrupted too and so on till the main method.

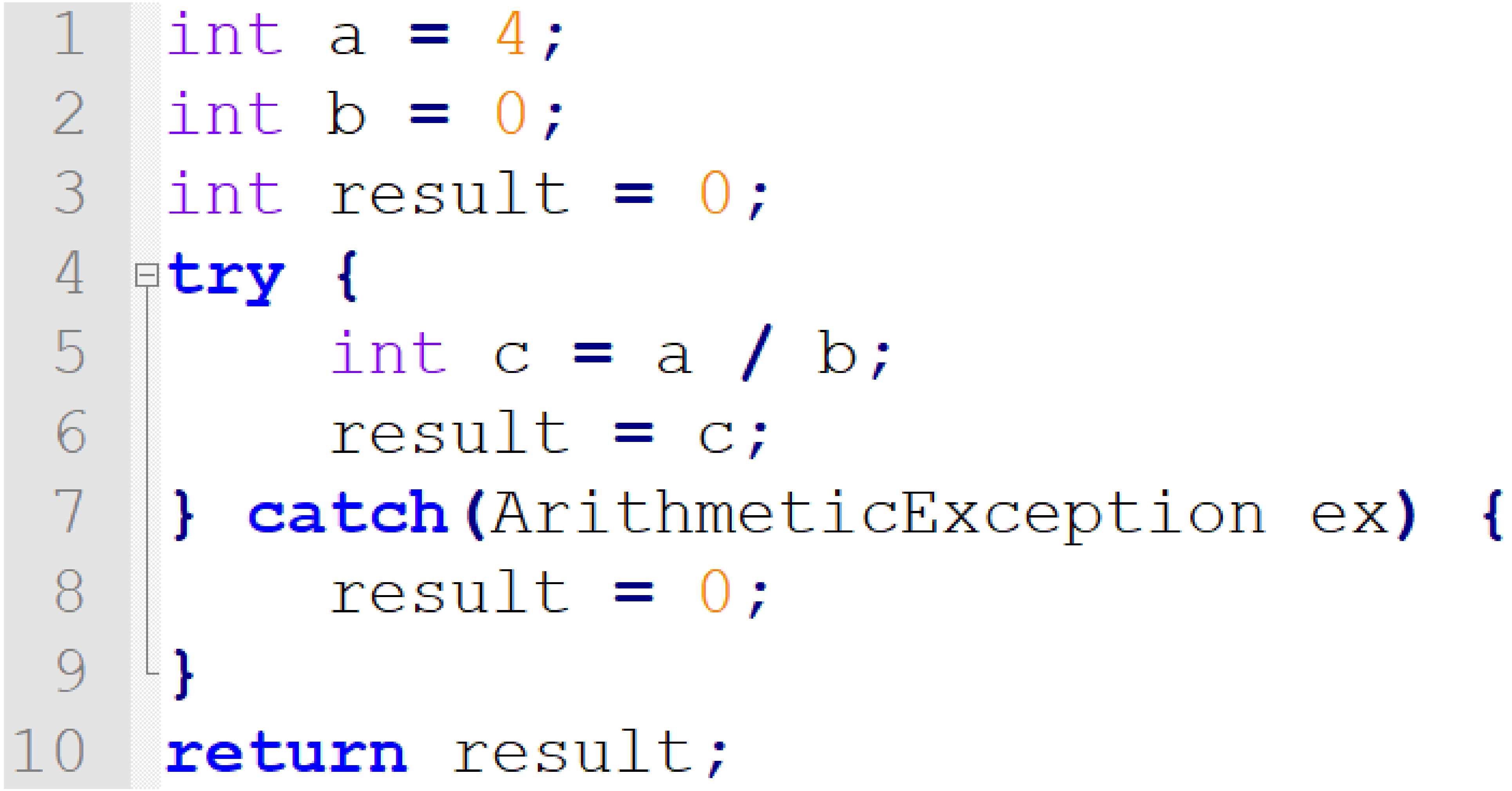
A thrown exception can also be caught using
a try/catch statement.

try/catch statement: code



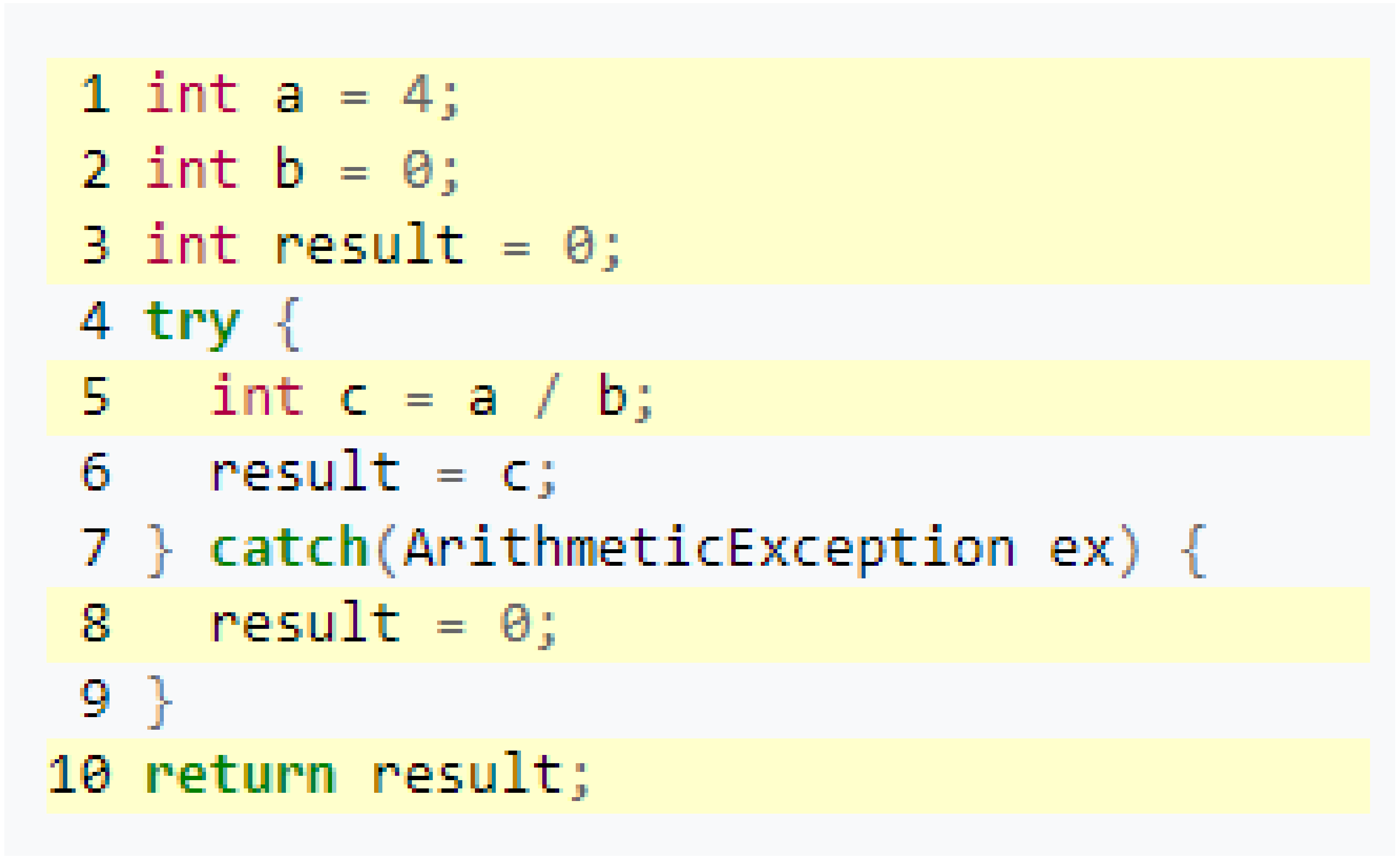
```java
int a = 4;
int b = 2;
int result = 0;
try {
  int c = a / b;
  result = c;
} catch(ArithmeticException ex) {
  result = 0;
}
return result;
```

Quiz time: Which code lines are executed when b = 0?



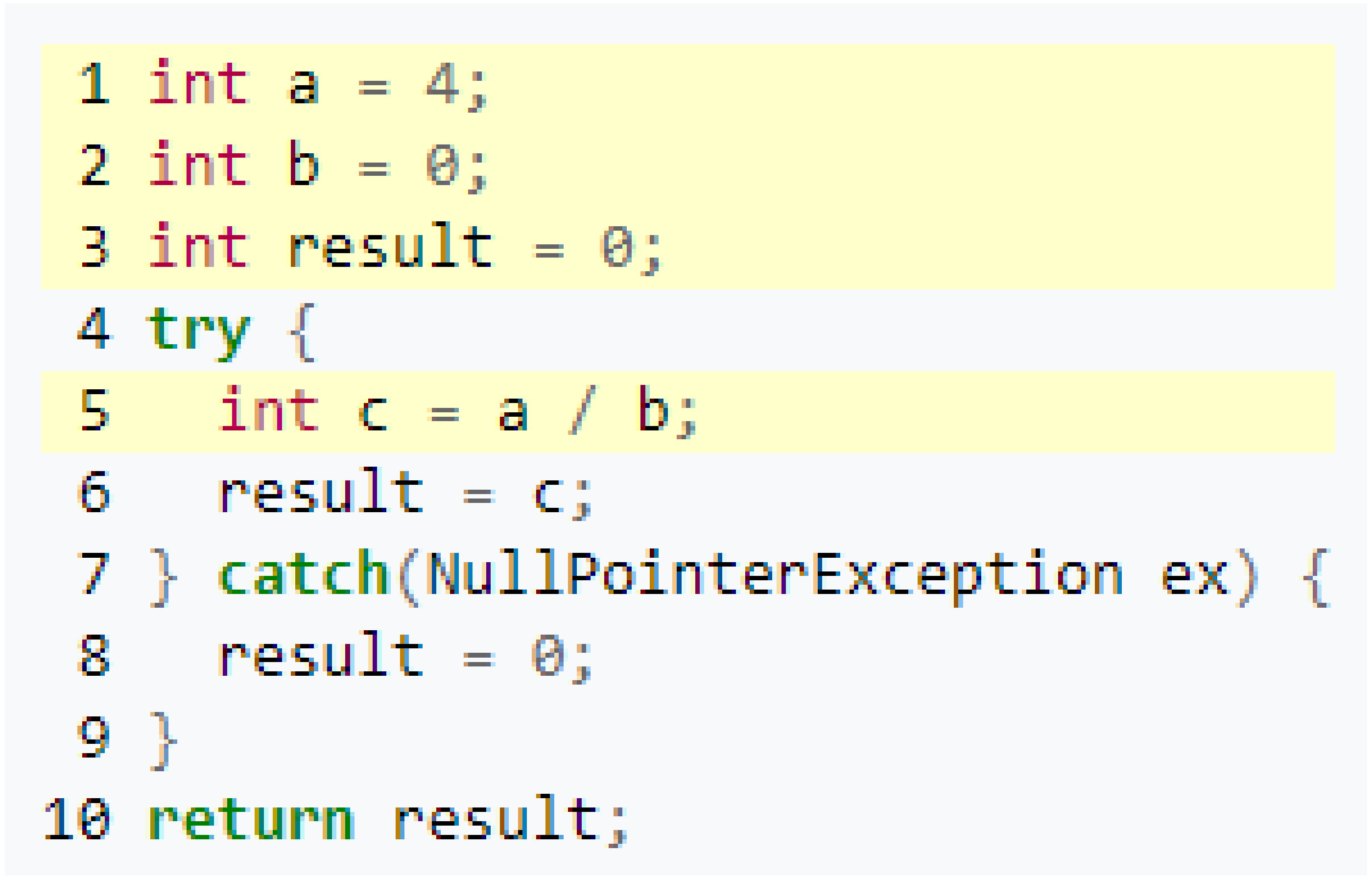
```
1  int a = 4;
2  int b = 0;
3  int result = 0;
4  try {
5      int c = a / b;
6      result = c;
7  } catch(ArithmeticException ex) {
8      result = 0;
9  }
10 return result;
```

Solution



```java
int a = 4;
int b = 0;
int result = 0;
try {
  int c = a / b;
  result = c;
} catch(ArithmeticException ex) {
  result = 0;
}
return result;
```

What's going on here?



```
int a = 4;
int b = 0;
int result = 0;
try {
  int c = a / b;
  result = c;
} catch(NullPointerException ex) {
  result = 0;
}
return result;
```

catch blocks

- A try/catch statement can contain several catch blocks, to handle different exceptions in different ways.
- An exception object may match several catch block but only the first catch block that matches the object will be executed.
- A catch-block will catch a thrown exception if and only if:
  - the thrown exception object is the same as the exception object specified by the catch-block.
  - the thrown exception object is the subtype of the exception object specified by the catch-block.

Quiz time: Fix this using exception handling

```
// .. inside main method

System.out.println(Integer.parseInt("tahu apa yang paling
bikin sakit? tahu kamu sudah ada yang punya T_T"));
System.out.println("Cetak aku pls");
```

Solution

```
// .. inside main method

try {
     System.out.println(Integer.parseInt("tahu apa yang 
paling bikin sakit? tahu kamu sudah ada yang punya T_T"));
} catch(java.lang.NumberFormatException ex) {
     System.out.println("Value tidak benar");
}
System.out.println("Cetak aku pls");
```

Quiz time: Can you fix this?

```
public class Exercise {
     public static void main(String[] args) {

          try {
               System.out.print(2/0);
          }
          catch(Exception e) { }
          catch(ArithmeticException e) { }

     }
}
```

finally block

- A finally block can be added after the catch blocks.
- A finally block is always executed, even when no exception is thrown, an exception is thrown and caught, or an exception is thrown and not caught.
- It's a place to put code that should always be executed after an unsafe operation like a file close or a database disconnection.
- You can define a try block without catch block, however, in this case, it must be followed by a finally block.

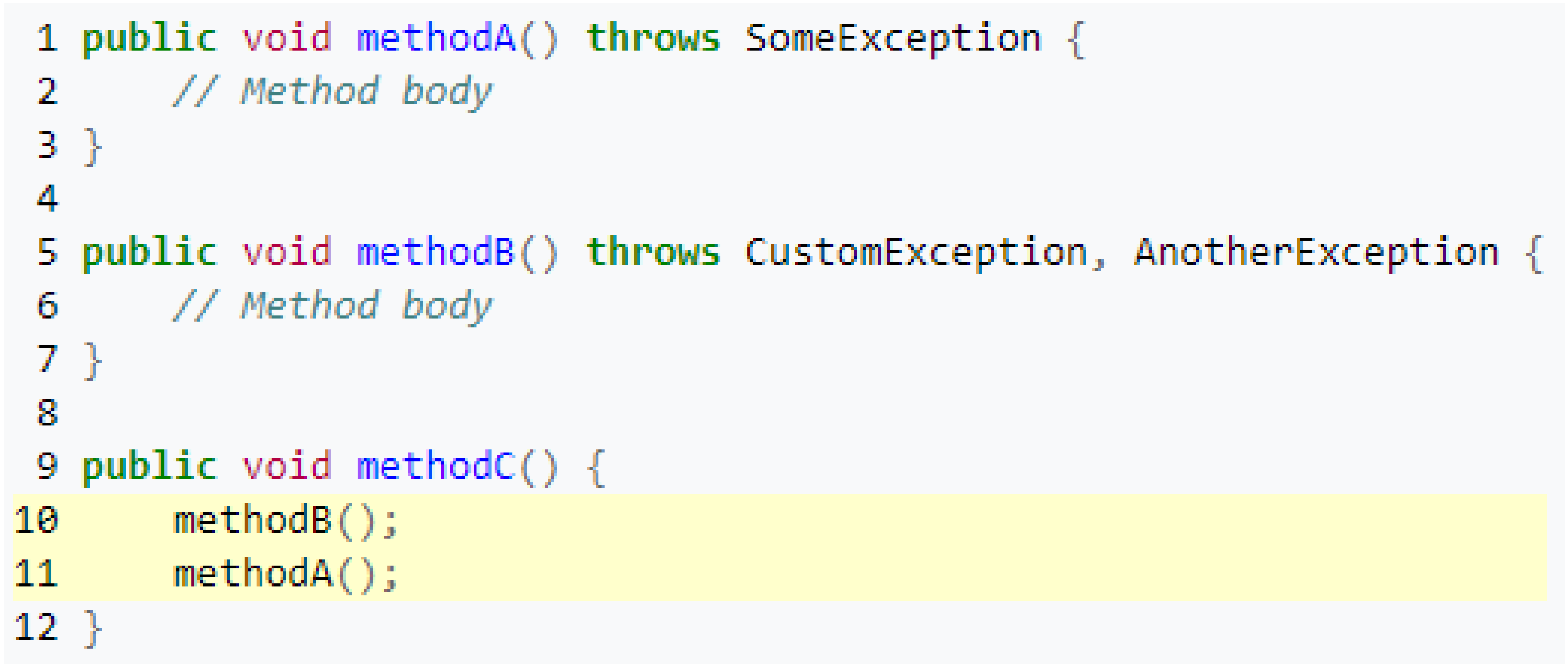
Quiz time: What's the output?

```
public class Exercise {
     public static void main(String[] args) {
          try {
               System.out.print(2/0);
          }
          catch(ArithmeticException e) {
               System.out.print("A");
          }
          finally {
               System.out.print("B");
          }
     }
}
```

Quiz time: What's the output?

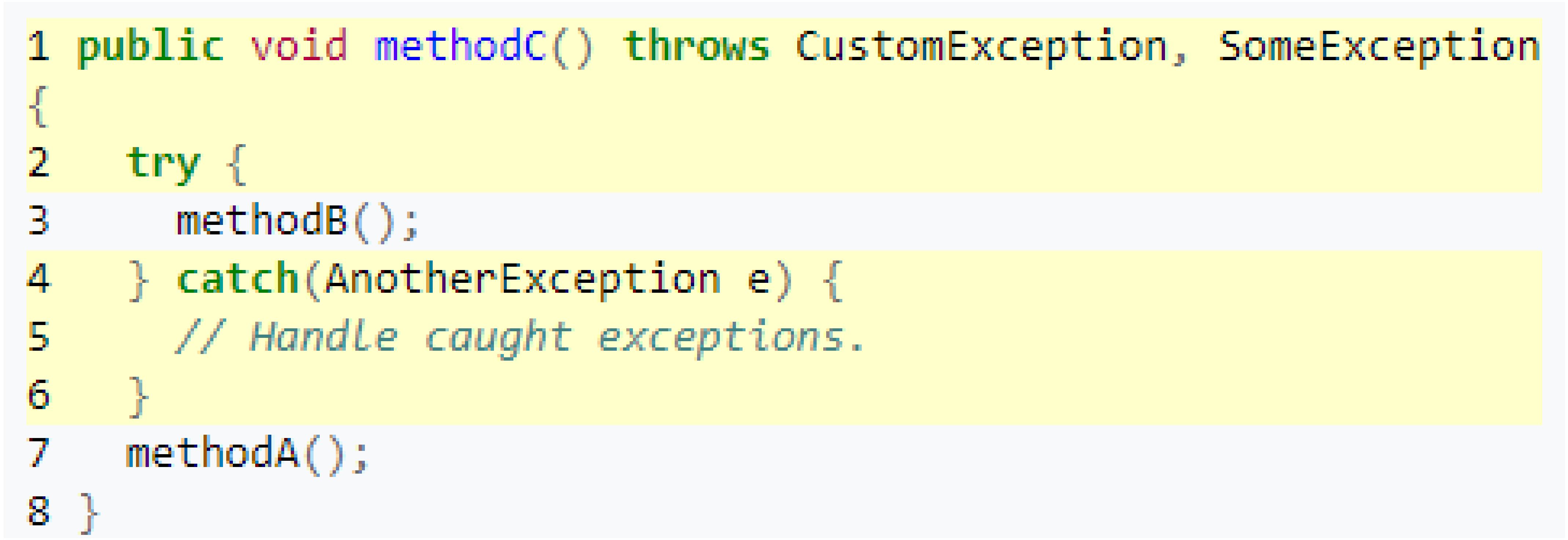
```
public class Exercise {
      public static void main(String[] args) {
            try {
                  System.out.print(2/1);
            }
            catch(ArithmeticException e) {
                  System.out.print("A");
            }
            finally {
                  System.out.print("B");
            }
      }
}
```

Example of handling exceptions: *oops*



```java
public void methodA() throws SomeException {
    // Method body
}

public void methodB() throws CustomException, AnotherException {
    // Method body
}

public void methodC() {
    methodB();
    methodA();
}
```

Example of handling exceptions



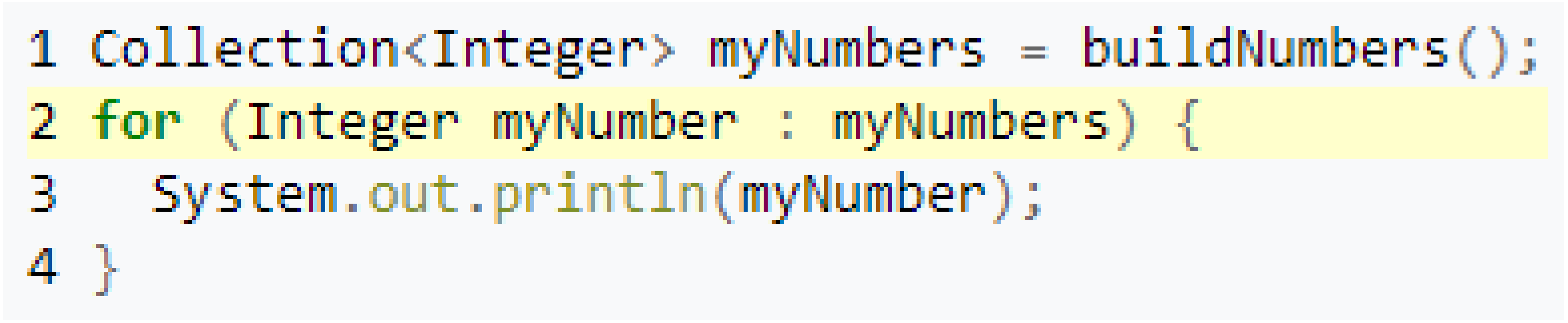
```java
public void methodC() throws CustomException, SomeException {
  try {
    methodB();
  } catch(AnotherException e) {
    // Handle caught exceptions.
  }
  methodA();
}
```

NullPointerException

In Java, a special null value can be assigned to an object reference. NullPointerException is thrown when an application attempts to use an object reference that has the null value. These include:

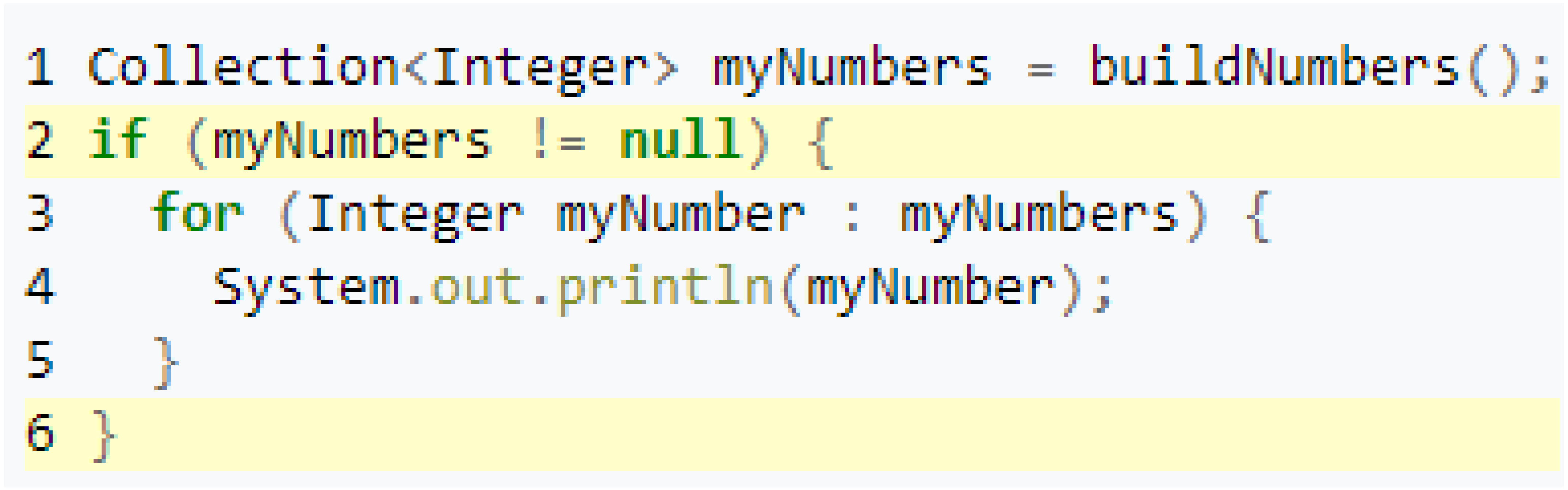
- Calling an instance method on the object referred by a null reference.
- Accessing or modifying an instance field of the object referred by a null reference.
- If the reference type is an array type, taking the length of a null reference.
- If the reference type is an array type, accessing or modifying the slots of a null reference.

Quiz time: What can go wrong, and how to fix it?



```
Collection<Integer> myNumbers = buildNumbers();
for (Integer myNumber : myNumbers) {
  System.out.println(myNumber);
}
```

Quiz time: What can go wrong? Fixed!



```java
Collection<Integer> myNumbers = buildNumbers();
if (myNumbers != null) {
  for (Integer myNumber : myNumbers) {
    System.out.println(myNumber);
  }
}
```

Quiz time: What can go wrong? Fixed too!

```
Collection<Integer> myNumbers = buildNumbers();
try {
	for(Integer myNumber:myNumbers) {
		System.out.println(myNumber);
	}
} catch(NullPointerException e) {
	System.out.println("The collection is empty!");
}
```

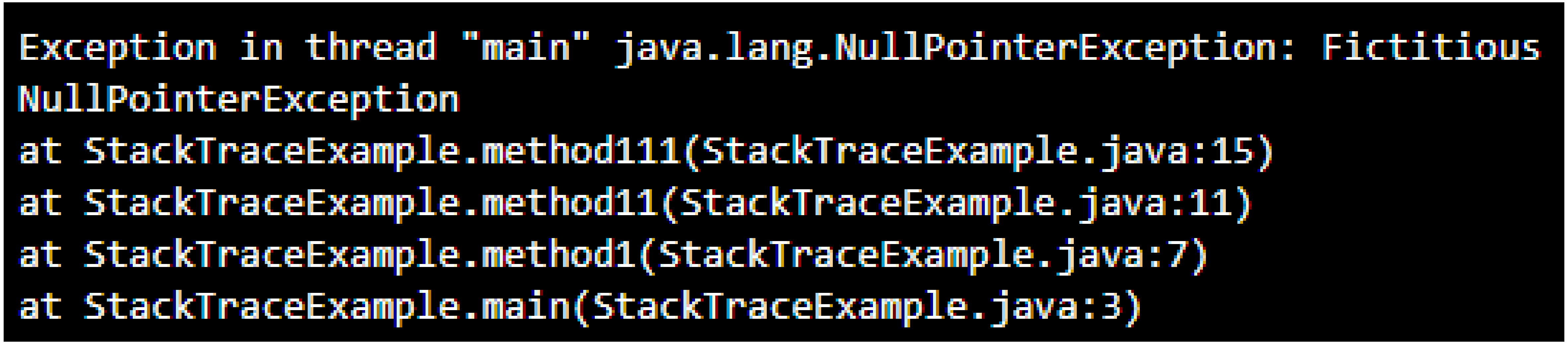
Stack trace

- A Stack Trace is a list of method calls from the point when the application was started to the current location of execution within the program.
- A Stack Trace is produced automatically by the Java Virtual Machine when an exception is thrown to indicate the location and progression of the program up to the point of the exception.
- The most recent method calls are at the top of the list.

Stack trace

```
public class StackTraceExample {
  public static void main(String[] args) {
    method1();
  }
  public static void method1() {
    method11();
  }
  public static void method11() {
    method111();
  }
  public static void method111() {
    throw new NullPointerException("Fictitious
NullPointerException");
  } }
```

Stack trace



```
Exception in thread "main" java.lang.NullPointerException: Fictitious
NullPointerException
at StackTraceExample.method111(StackTraceExample.java:15)
at StackTraceExample.method11(StackTraceExample.java:11)
at StackTraceExample.method1(StackTraceExample.java:7)
at StackTraceExample.main(StackTraceExample.java:3)
```

Checked vs. Unchecked Exceptions

A **checked exception** is a type of exception that must be either caught or declared in the method in which it is thrown.
For example, the java.io.IOException is a checked exception.
It extends **java.lang.Exception**.

Unchecked, uncaught, or runtime exceptions are exceptions that can be thrown without being caught or declared.
They extend **java.lang.RuntimeException**.

Checked Exception

```
public static void main(String[] args) {
            ioOperation(true);
        }

public static void ioOperation(boolean isResourceAvailable) throws
IOException {
  if (!isResourceAvailable) {
        throw new IOException();
  }
}
```

A **checked exception** is a type of exception that must be either caught or declared in the method in which it is thrown. The code above won't compile: **Unhandled exception type IOException**

Checked Exception

```
public static void main(String[] args) {
       try {
             ioOperation(true);
       } catch(IOException e) {
             System.out.println("IO Error");
       }

public static void ioOperation(boolean isResourceAvailable) throws
IOException {
  if (!isResourceAvailable) {
       throw new IOException();
  }
}
```

A **checked exception** must always be enclosed with try-catch (or add throws).

Unchecked Exception

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

public static void someOperation() {
      List<String> l = null;
      l.isEmpty();
}
```

An **unchecked exception** does not need to be caught or thrown.

Make your own exception

```
// OwnException.java
public class OwnException extends Exception {

      public OwnException(String errorMsg) {
            super(errorMsg);
      }

}
```

Make your own exception

```
// Exercise.java

public class Exercise {

        public static void main(String[] args) {

                try {
                        someMethod(5);
                } catch(OwnException e) {
                        System.out.println(e.getMessage());
                }

        }

        public static void someMethod(int i) throws OwnException {
                if(i > 0)
                        throw new OwnException("Ouch!");
        }

}
```

Quiztime: Make an exception class, and throw it whenever `age < 18` in the method `checkAge(int age)`.

Inspired by:

https://en.wikibooks.org/wiki/Java_Programming
Liang. Introduction to Java Programming. Tenth Edition. Pearson 2015.
Think Java book by Allen Downey and Chris Mayfield.
Eck. Introduction to Programming Using Java. 2014.