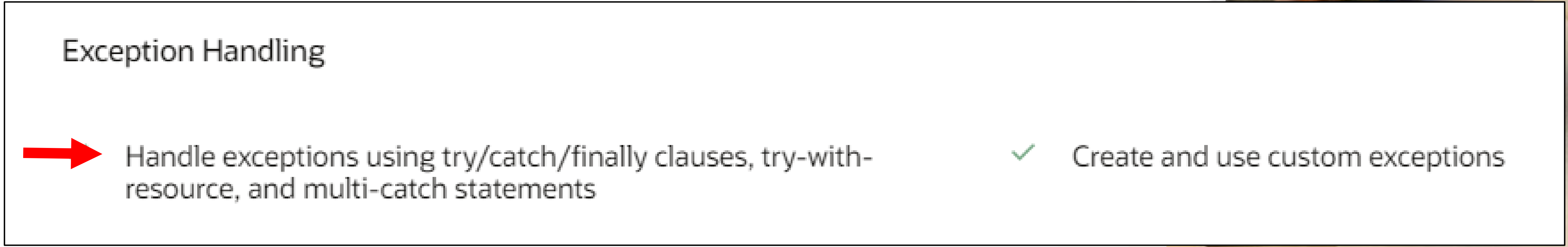
Exception Handling

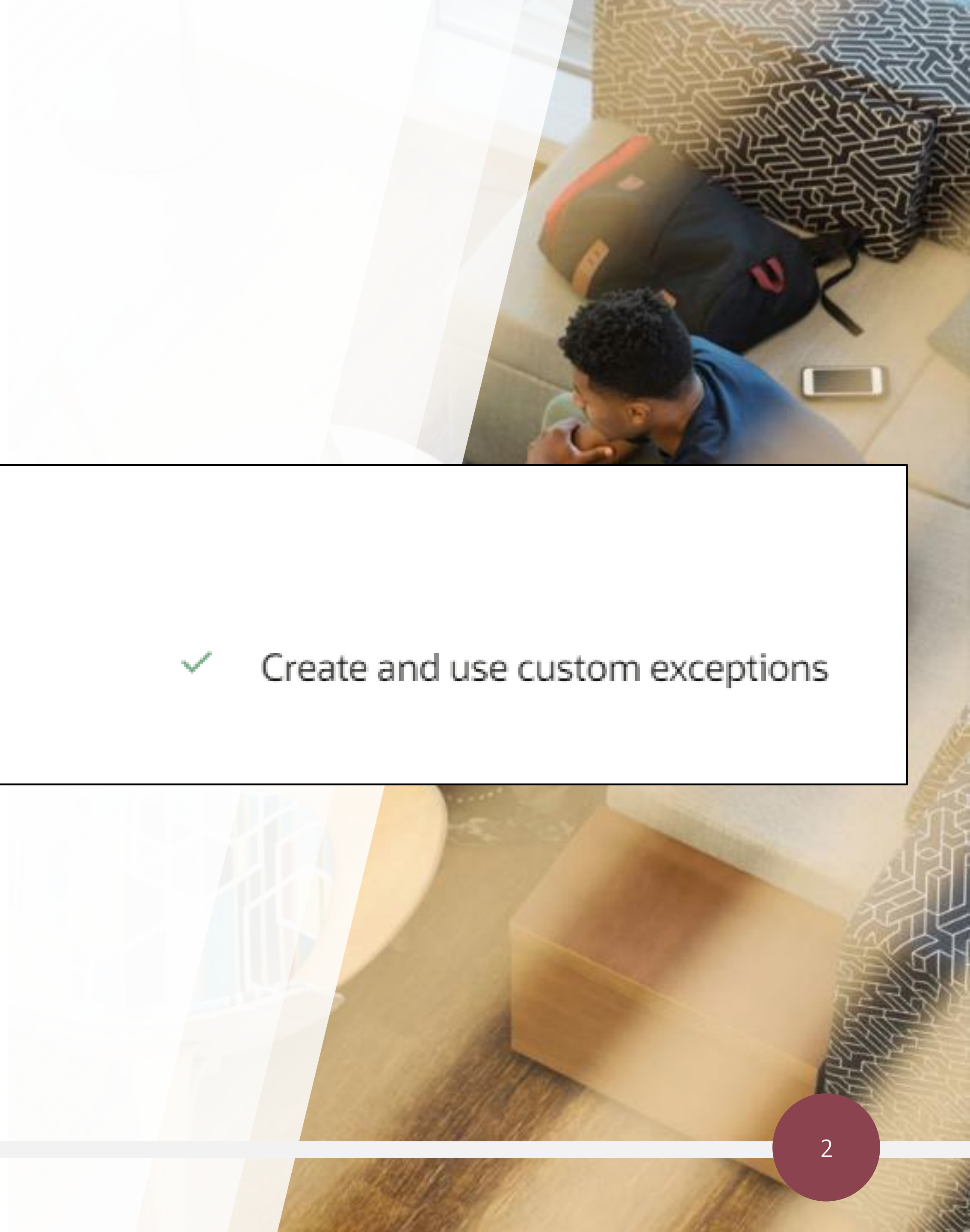
Exception Handling



Exception Handling

Handle exceptions using try/catch/finally clauses, try-with-resource, and multi-catch statements

✓ Create and use custom exceptions



Exceptions

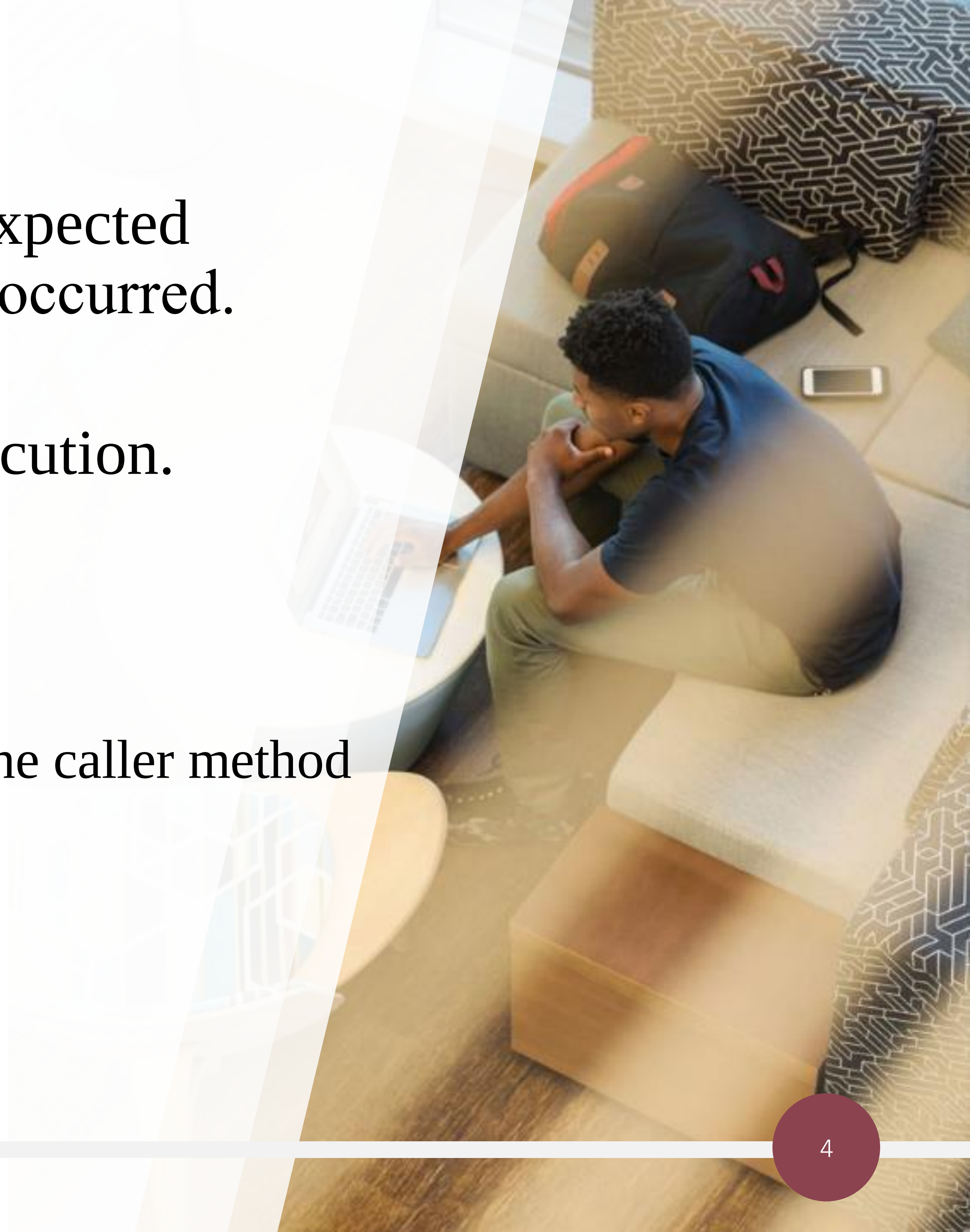
- Java’s exception-handling features give us the following benefits:
  - ➢ an elegant mechanism that produces efficient and organised error-handling code
  - ➢ errors are detected easily without the need to write special code to test return values
  - ➢ exception-*handling* code is cleanly separated from exception-*generating* code
  - ➢ the same exception-handling code can be used to deal with a range of possible exceptions

Exceptions

- An exception indicates that something unexpected happened. An “exceptional” condition has occurred.

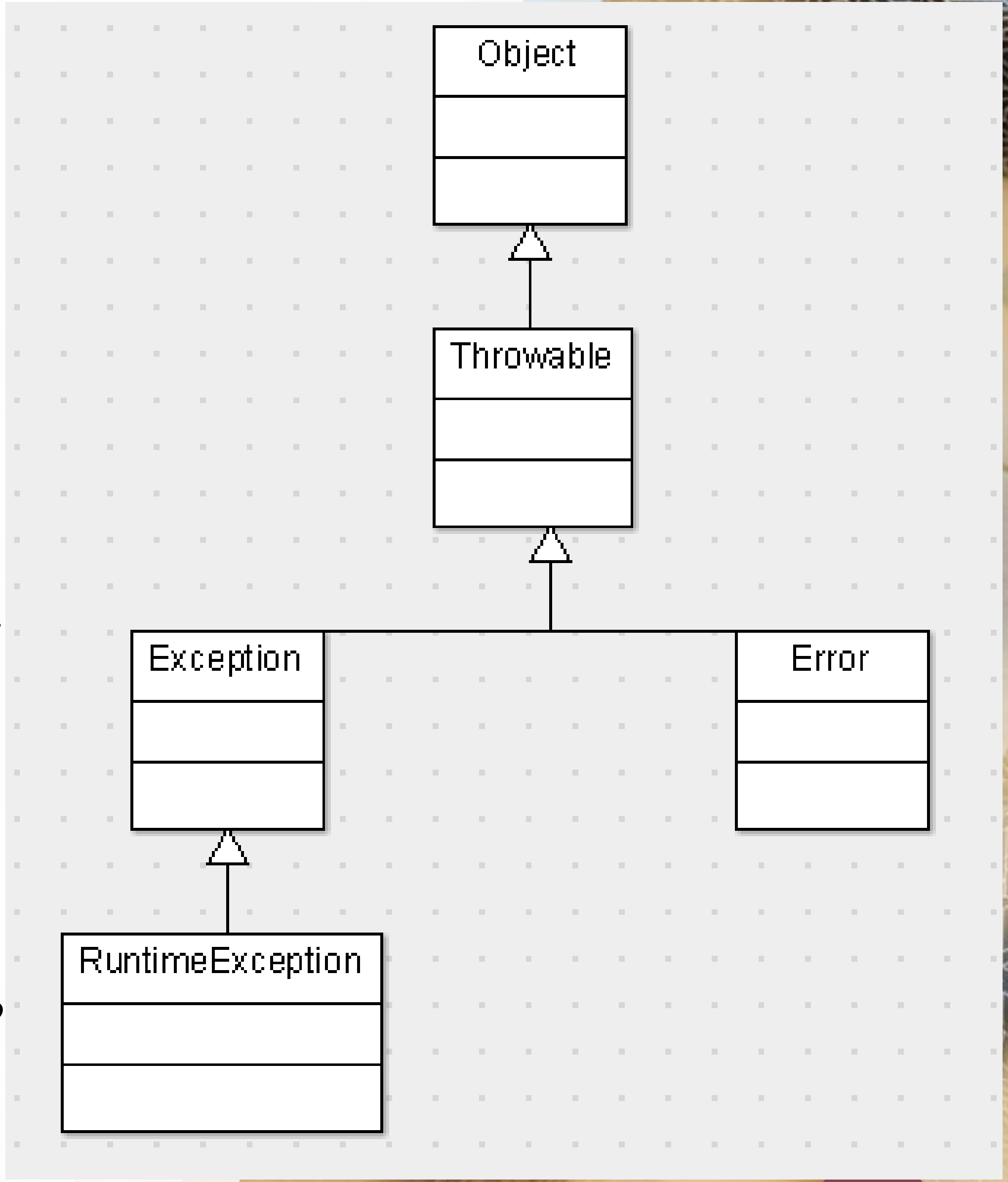
- An exception alters the normal flow of execution.

- There are two approaches:
    1. a method can handle the exception itself or
    2. the method can hand over responsibility to the caller method



Exceptions

- *Throwable* – all exceptions inherit from this class.
- *Error* – unexpected, serious conditions such as running out of memory. *Error* and all of it's subtypes are "unchecked" exceptions.
- *RuntimeException* – unexpected situations that are not necessarily fatal. *RuntimeException* and its subtypes are all "unchecked" exceptions.
- *Exception* – situations that can be anticipated (file IO error). *Exception* and its subtypes (except for *RuntimeException*) are all "checked" exceptions.

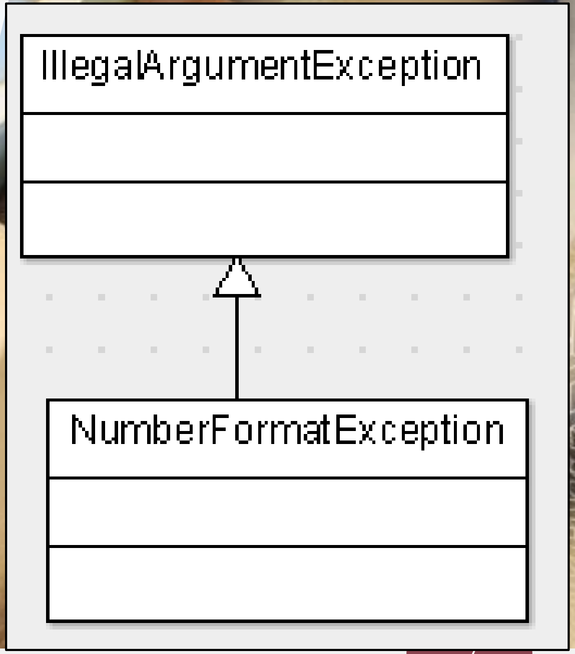


Checked/Unchecked Exceptions

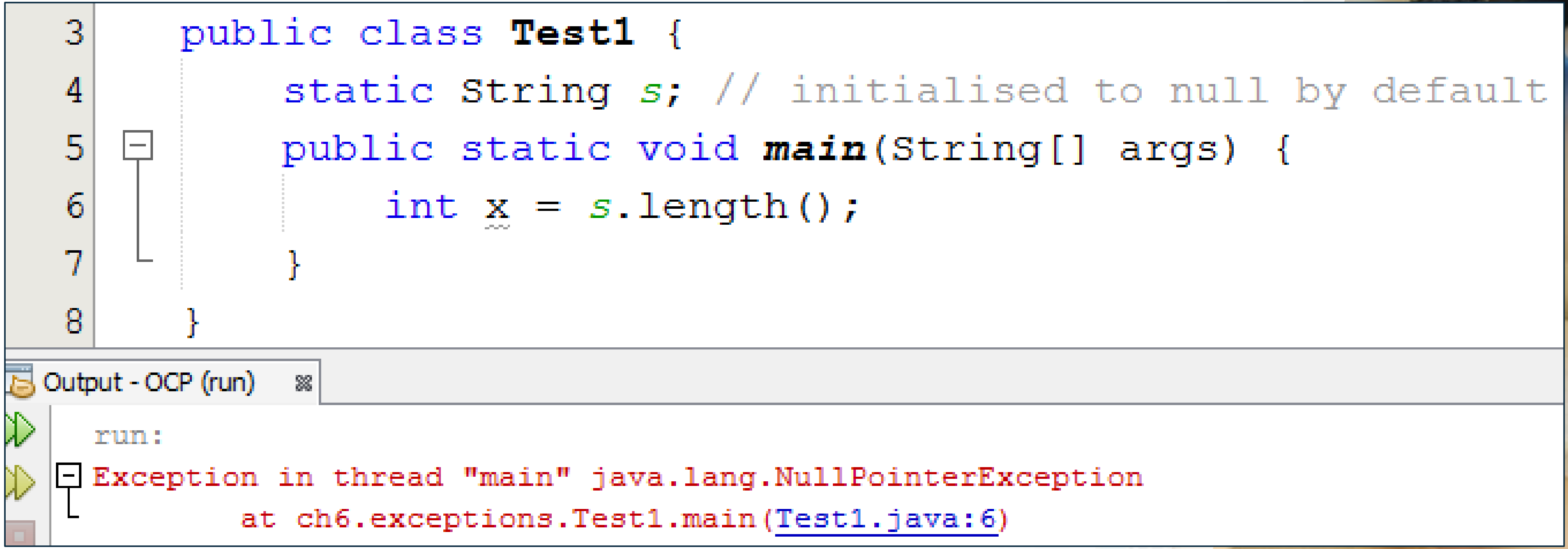
- "Catch or declare"
  - ➢ Because checked exceptions tend to be anticipated, Java makes sure some thought goes into them.
  - ➢ This rule states that all checked exceptions that could be thrown within a method are wrapped in a *try*-*catch* block or are thrown from the method (declared in the method signature).
  - ➢ *RuntimeException*'s do not have to obey this rule. This is why they are called unchecked exceptions.

RuntimeException Classes

- Common *RuntimeException*’s are:
  - ➢ *ArithmeticException* (divide by 0)
  - ➢ *ArrayIndexOutOfBoundsException* (accessing beyond the bounds of an array)
  - ➢ *ClassCastException* (have a reference point up the inheritance tree)
  - ➢ *IllegalArgumentException* (thrown by a method if it receives an invalid argument)
  - ➢ *NullPointerException* (trying to invoke a method when the reference is *null*)
  - ➢ *NumberFormatException* (trying to format “two” to 2; should be “2”.



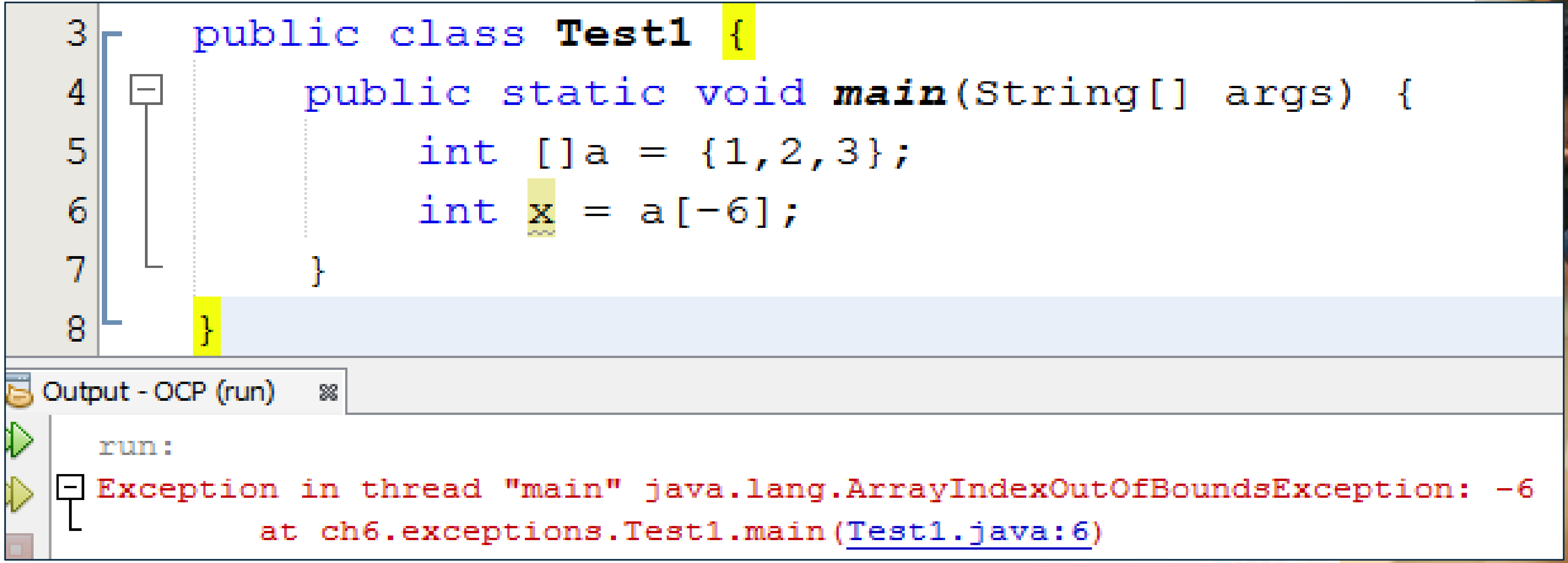
RuntimeExceptions in Code



```java
public class Test1 {
    static String s; // initialised to null by default
    public static void main(String[] args) {
        int x = s.length();
    }
}
```

```
run:
Exception in thread "main" java.lang.NullPointerException
        at ch6.exceptions.Test1.main(Test1.java:6)
```

RuntimeExceptions in Code

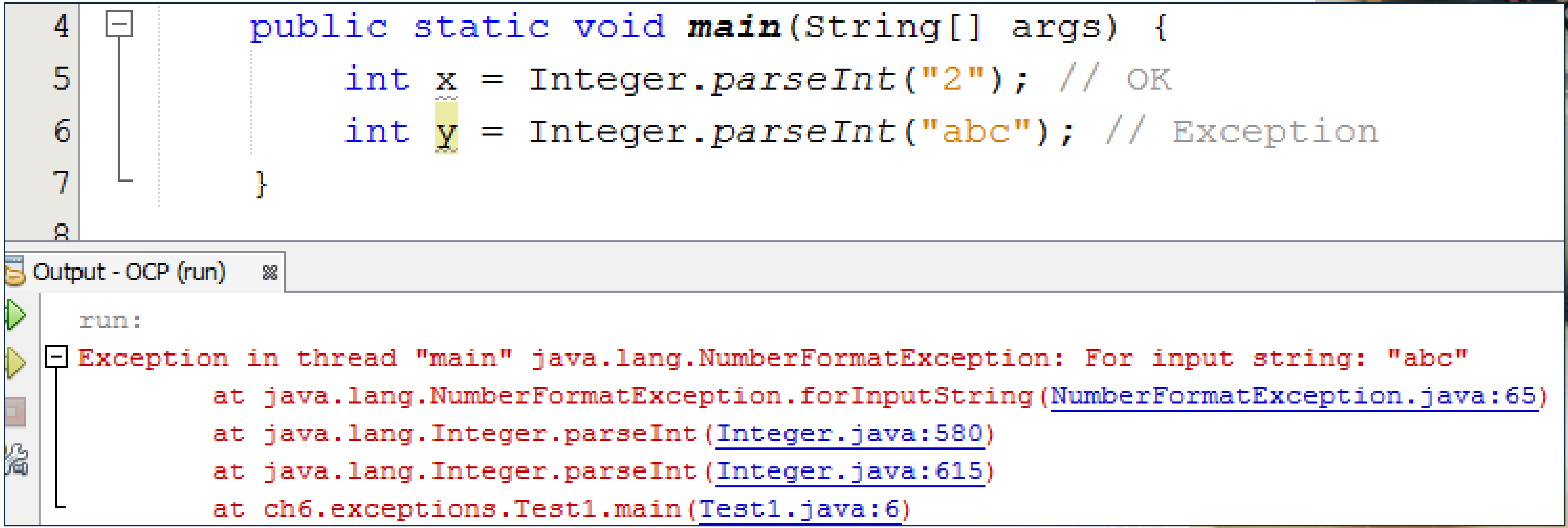


```
public class Test1 {
    public static void main(String[] args) {
        int []a = {1,2,3};
        int x = a[-6];
    }
}
```

Output - OCP (run)

```
run:
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: -6
        at ch6.exceptions.Test1.main(Test1.java:6)
```

RuntimeExceptions in Code

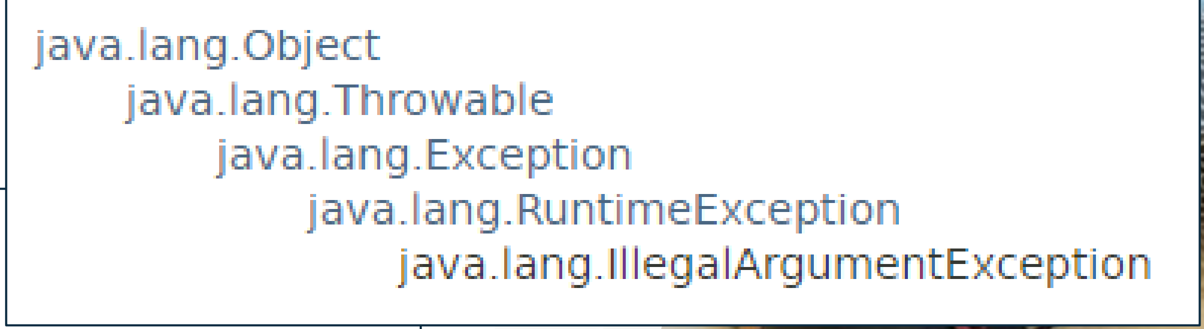


```
    public static void main(String[] args) {
        int x = Integer.parseInt("2"); // OK
        int y = Integer.parseInt("abc"); // Exception
    }
```

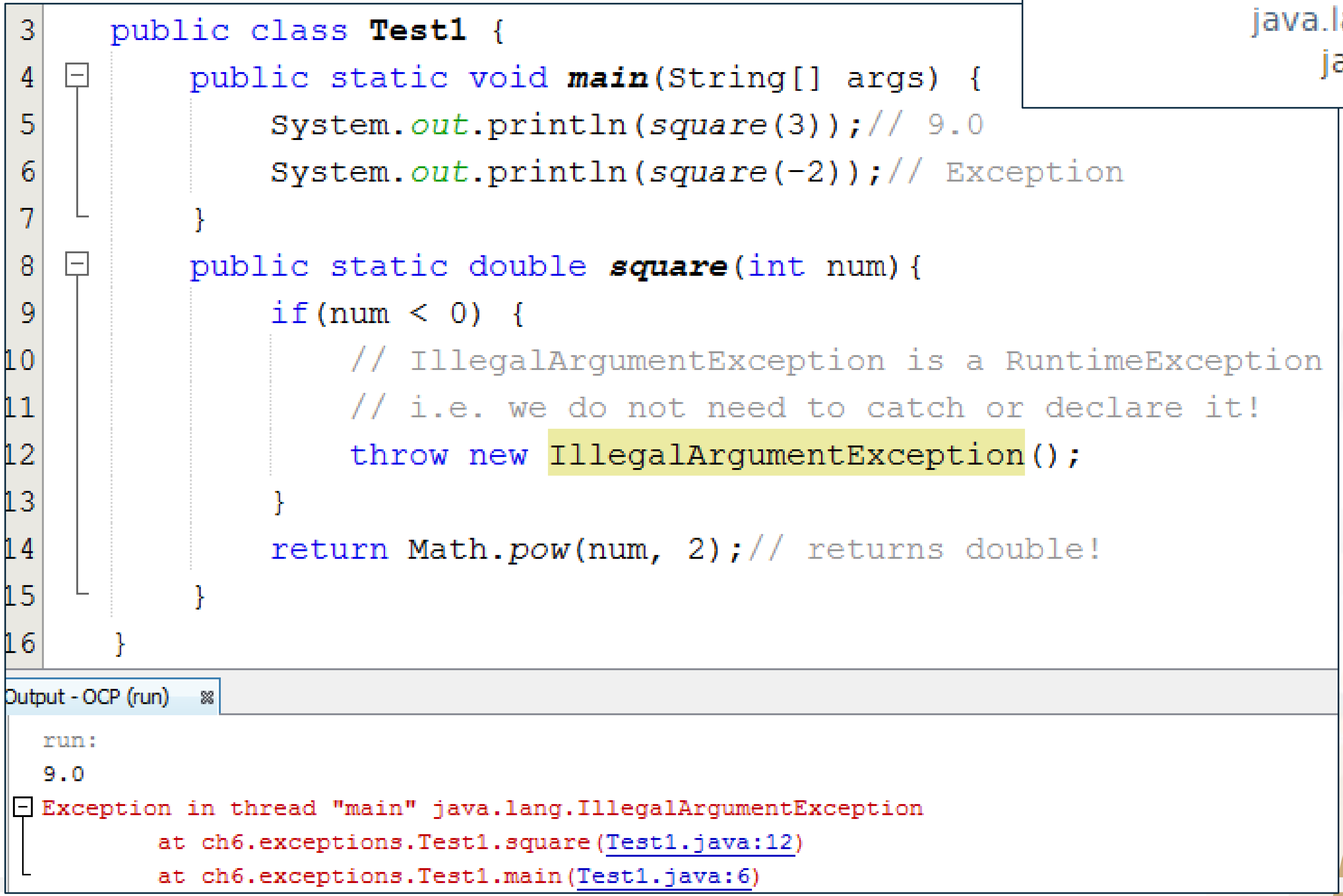
Output - OCP (run)

```
run:
Exception in thread "main" java.lang.NumberFormatException: For input string: "abc"
        at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
        at java.lang.Integer.parseInt(Integer.java:580)
        at java.lang.Integer.parseInt(Integer.java:615)
        at ch6.exceptions.Test1.main(Test1.java:6)
```

RuntimeExceptions in Code



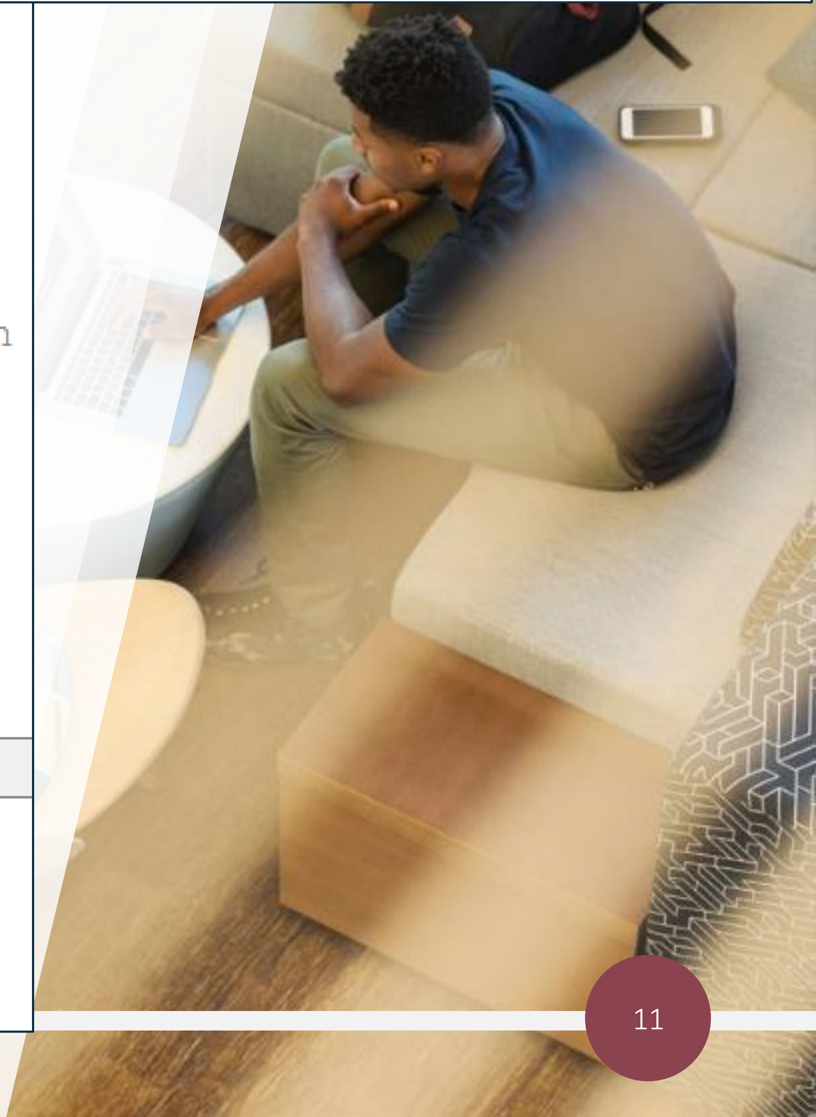
```
java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.lang.RuntimeException
                java.lang.IllegalArgumentException
```



```
public class Test1 {
    public static void main(String[] args) {
        System.out.println(square(3));// 9.0
        System.out.println(square(-2));// Exception
    }
    public static double square(int num){
        if(num < 0) {
            // IllegalArgumentException is a RuntimeException
            // i.e. we do not need to catch or declare it!
            throw new IllegalArgumentException();
        }
        return Math.pow(num, 2);// returns double!
    }
}
```

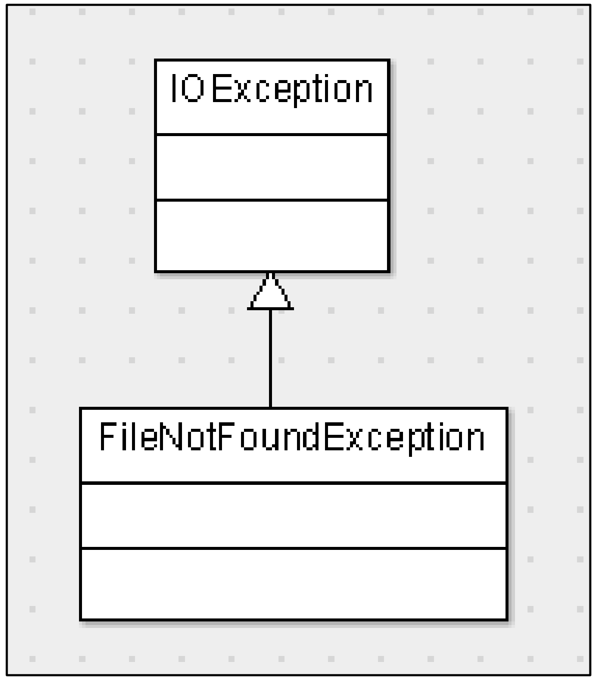
Output - OCP (run)

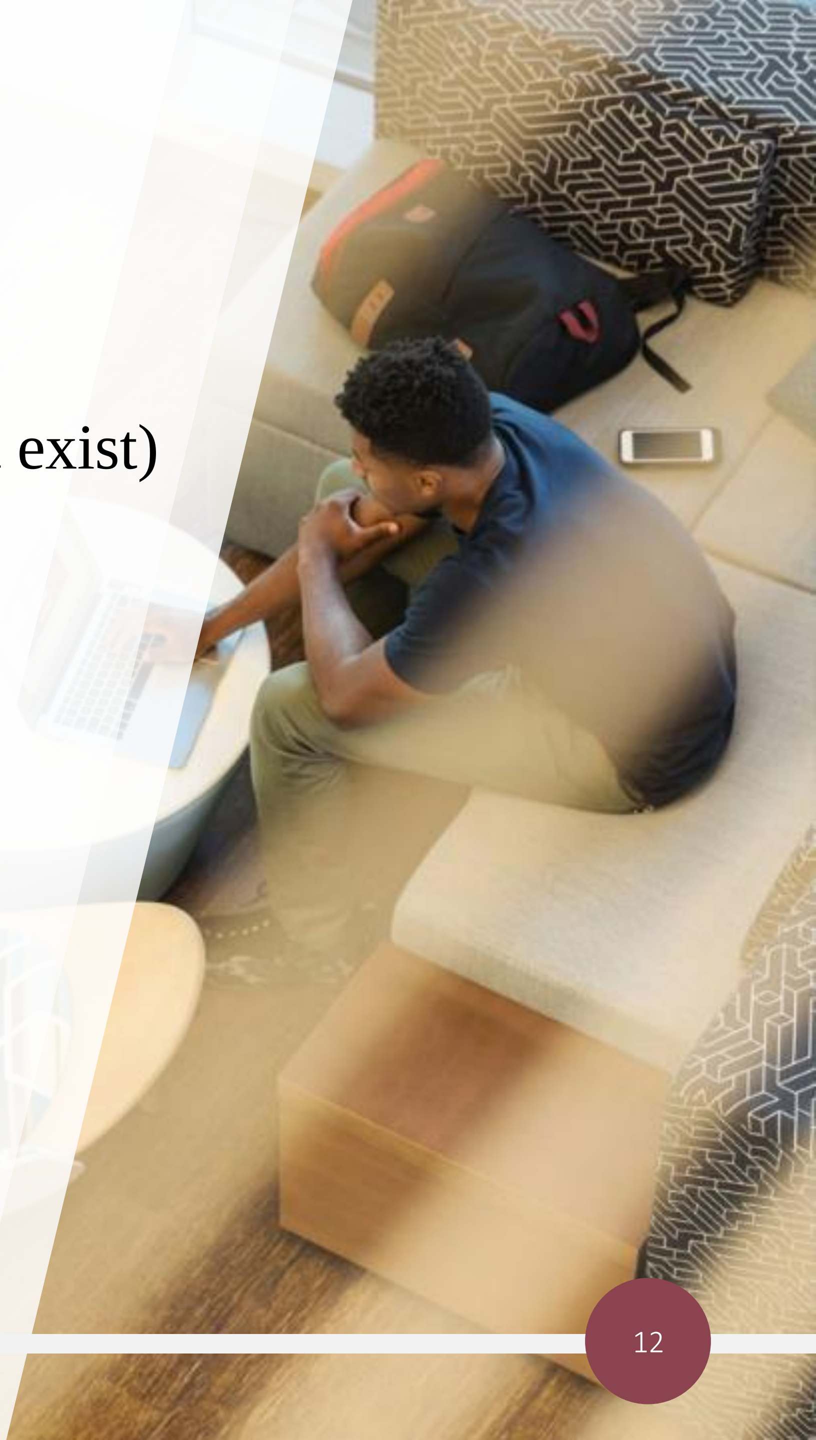
```
run:
9.0
Exception in thread "main" java.lang.IllegalArgumentException
        at ch6.exceptions.Test1.square(Test1.java:12)
        at ch6.exceptions.Test1.main(Test1.java:6)
```



Checked Exception Classes

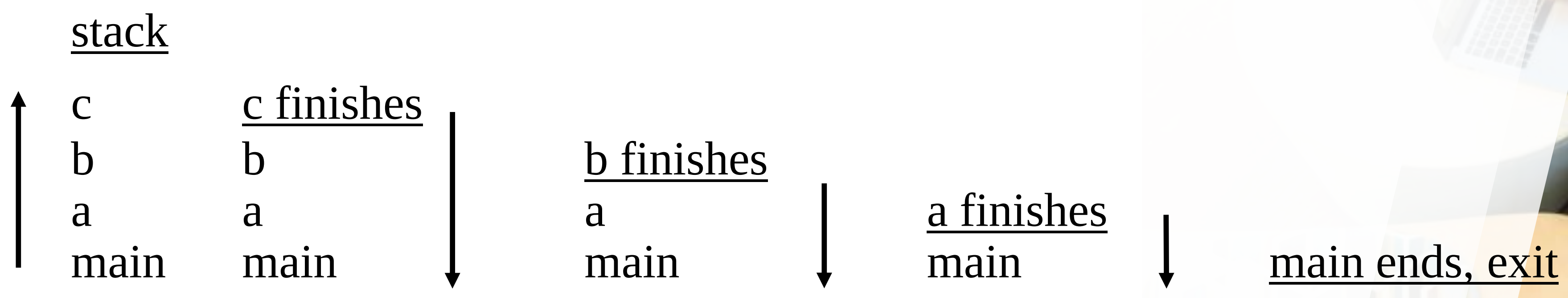
- Common checked *Exception* classes are:
  - ➢ *IOException* (reading/writing a file)
  - ➢ *FileNotFoundException* (trying to access a file that does not exist)





Call Stack

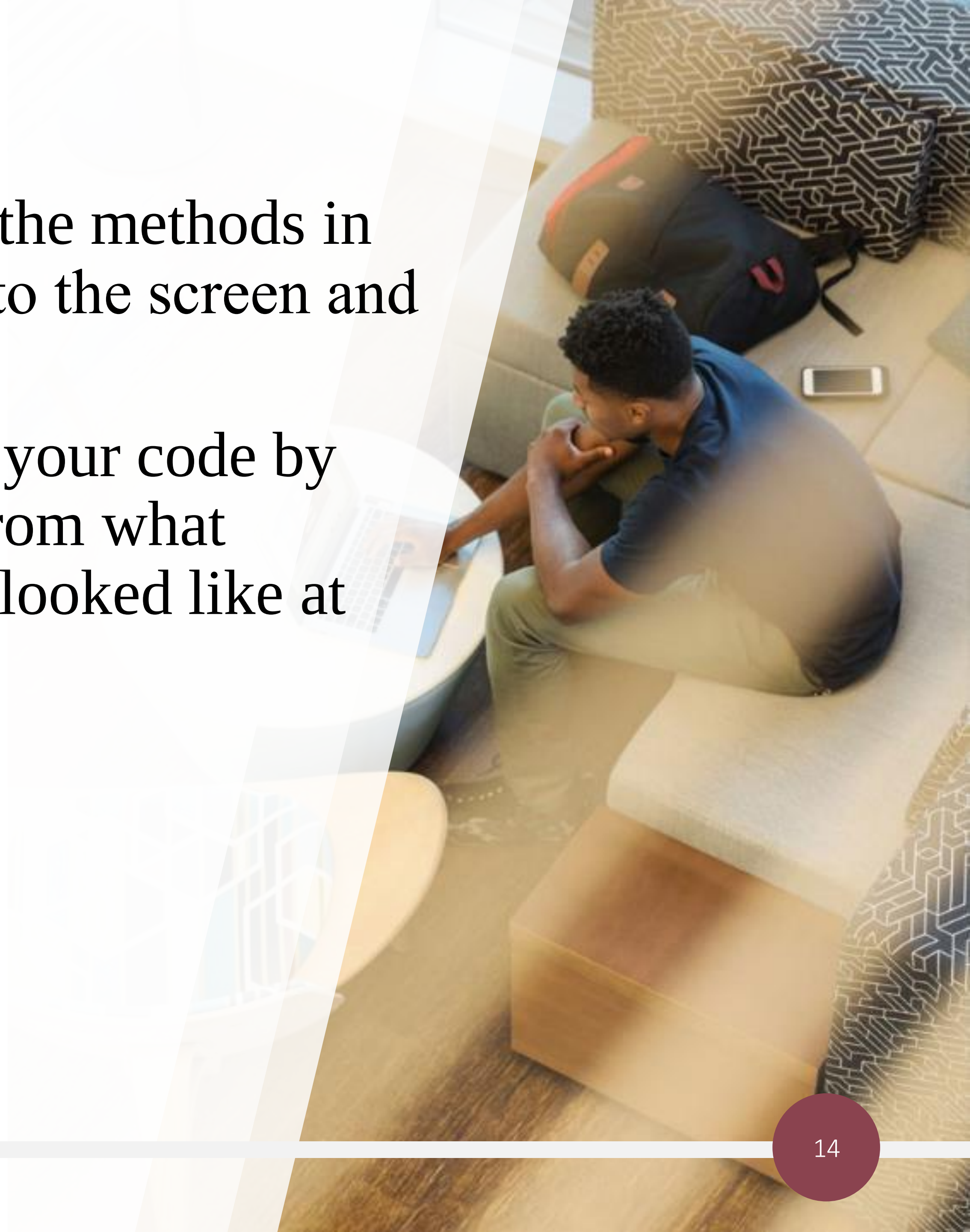
- The call stack is the chain of methods that your program executes to get to the current method.
- If a program starts in *main()* and *main()* calls method *a()*, which calls method *b()*, which in turn calls method *c()*, the call stack is as follows:



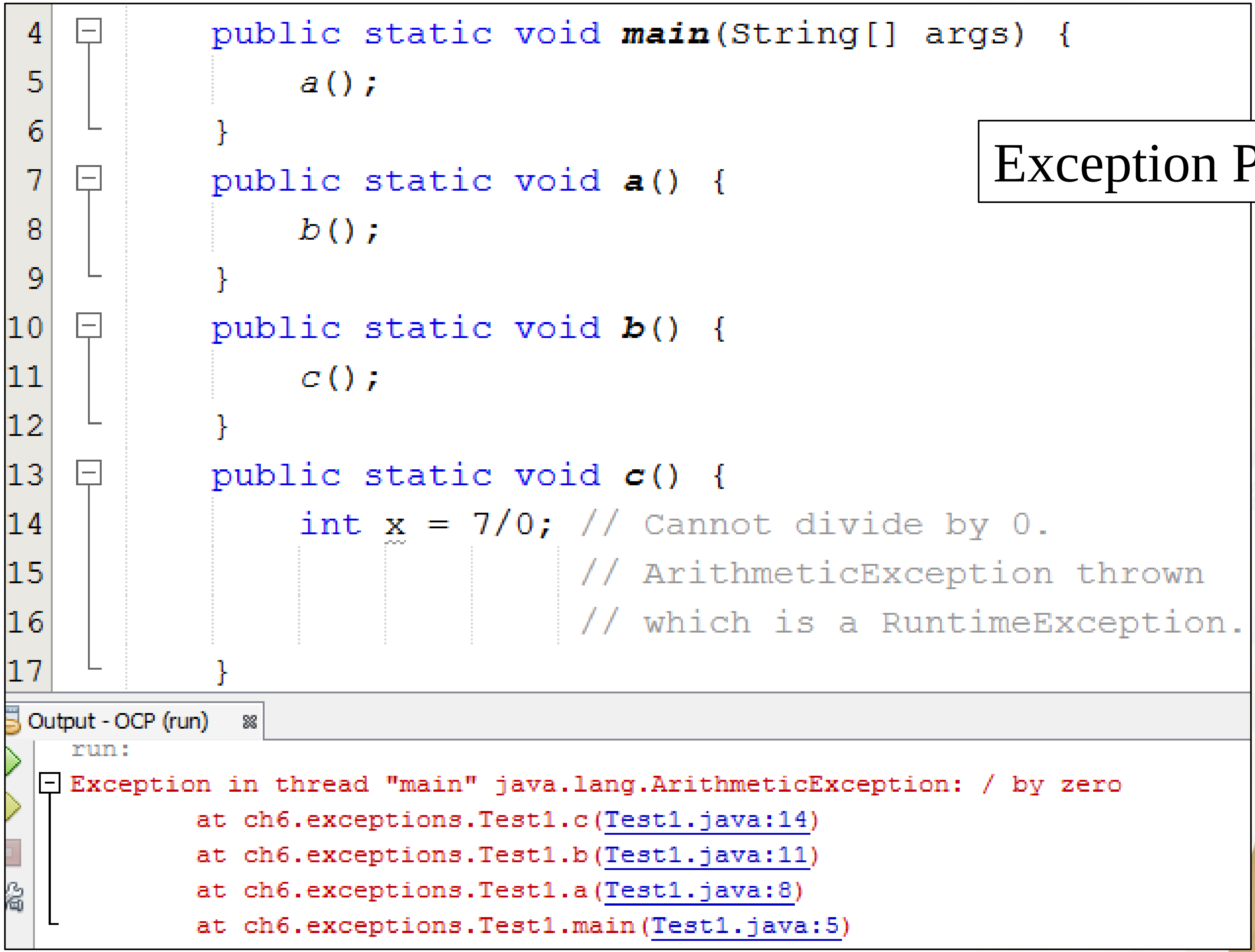
- The last method called is at the top of the stack and the first method is at the bottom.

Exception Propagation

- If an exception is never caught by any of the methods in the call stack, the call stack is “dumped” to the screen and your program stops running.
- The call stack displayed helps you debug your code by telling you what exception was thrown, from what method it was thrown and what the stack looked like at the time.



Exception Propagation



```java
    public static void main(String[] args) {
        a();
    }
    public static void a() {
        b();
    }
    public static void b() {
        c();
    }
    public static void c() {
        int x = 7/0; // Cannot divide by 0.
                     // ArithmeticException thrown
                     // which is a RuntimeException.
    }
```

Output - OCP (run)

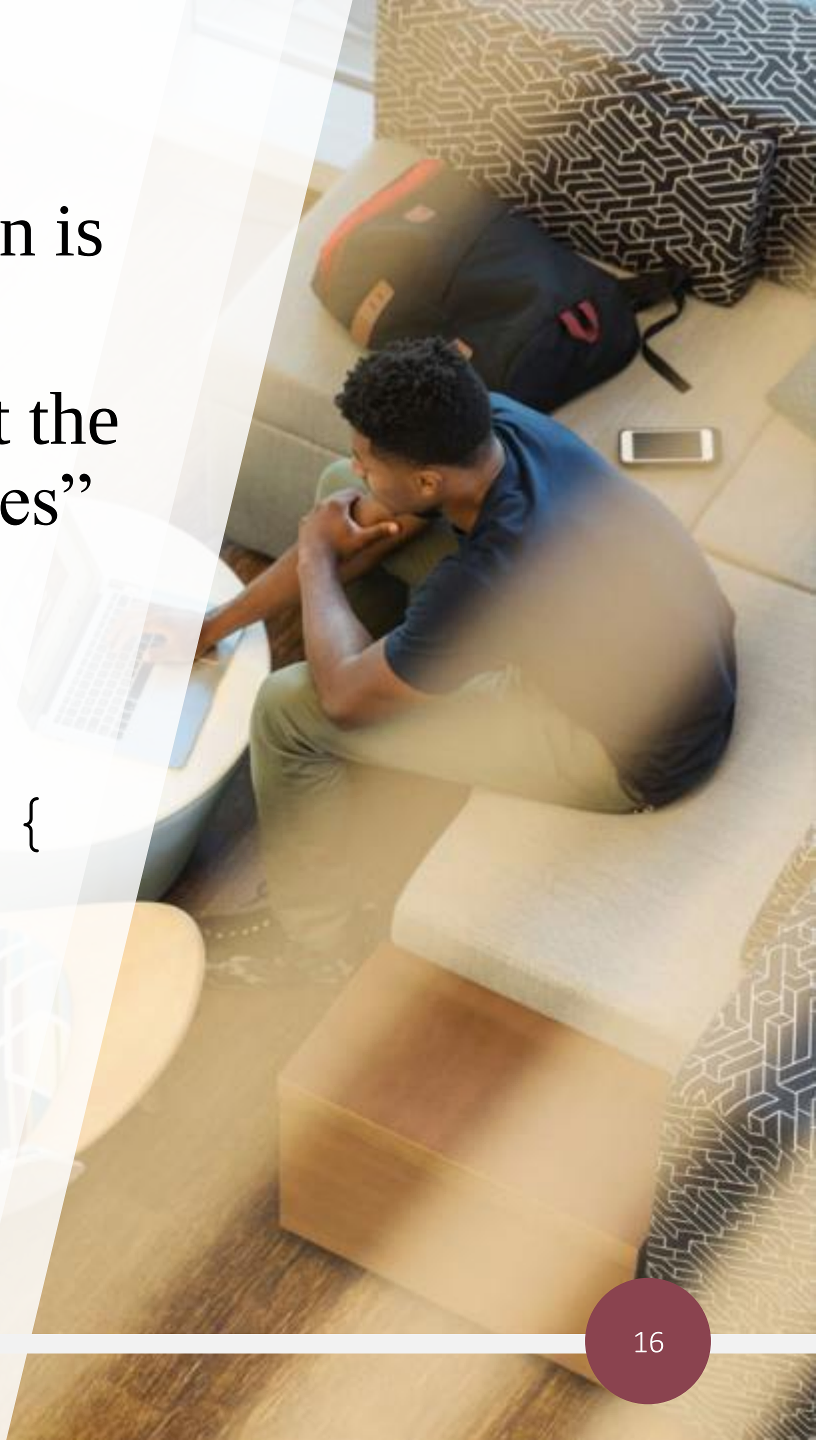
```
run:
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at ch6.exceptions.Test1.c(Test1.java:14)
        at ch6.exceptions.Test1.b(Test1.java:11)
        at ch6.exceptions.Test1.a(Test1.java:8)
        at ch6.exceptions.Test1.main(Test1.java:5)
```

try and *catch* blocks

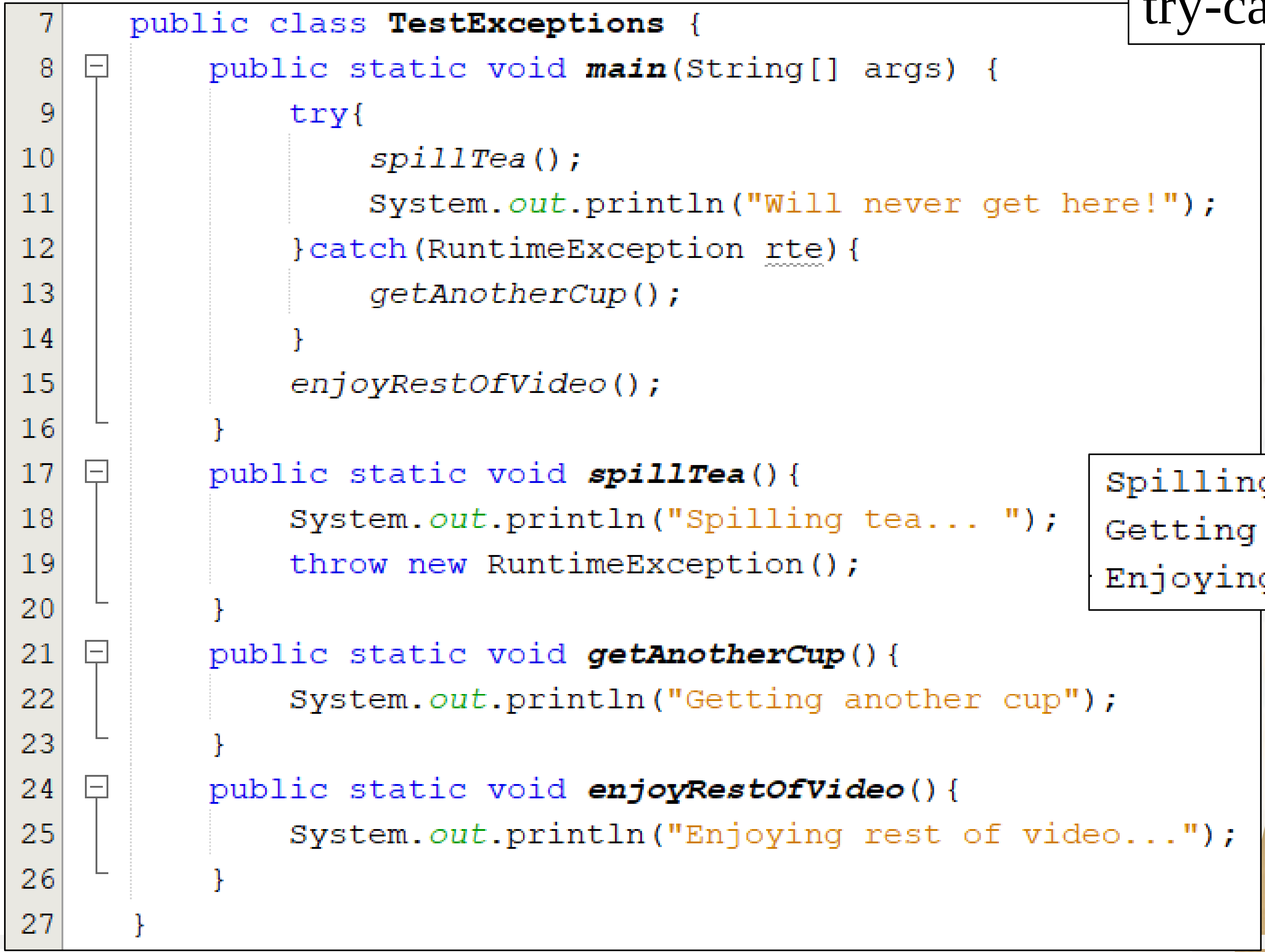
- When an exceptional event occurs in Java, an exception is said to be "thrown".
- The code that is responsible for doing something about the exception is called the "exception handler" and it "catches" the thrown exception.

```
try{
      // 'guarded region'
} catch (ExceptionType identifier){
      // handle this exception
}  // more catch blocks if needed
// some other unguarded, non-risky code…
```

- The {} around the *try* block are mandatory.
- A *catch/finally* is required.

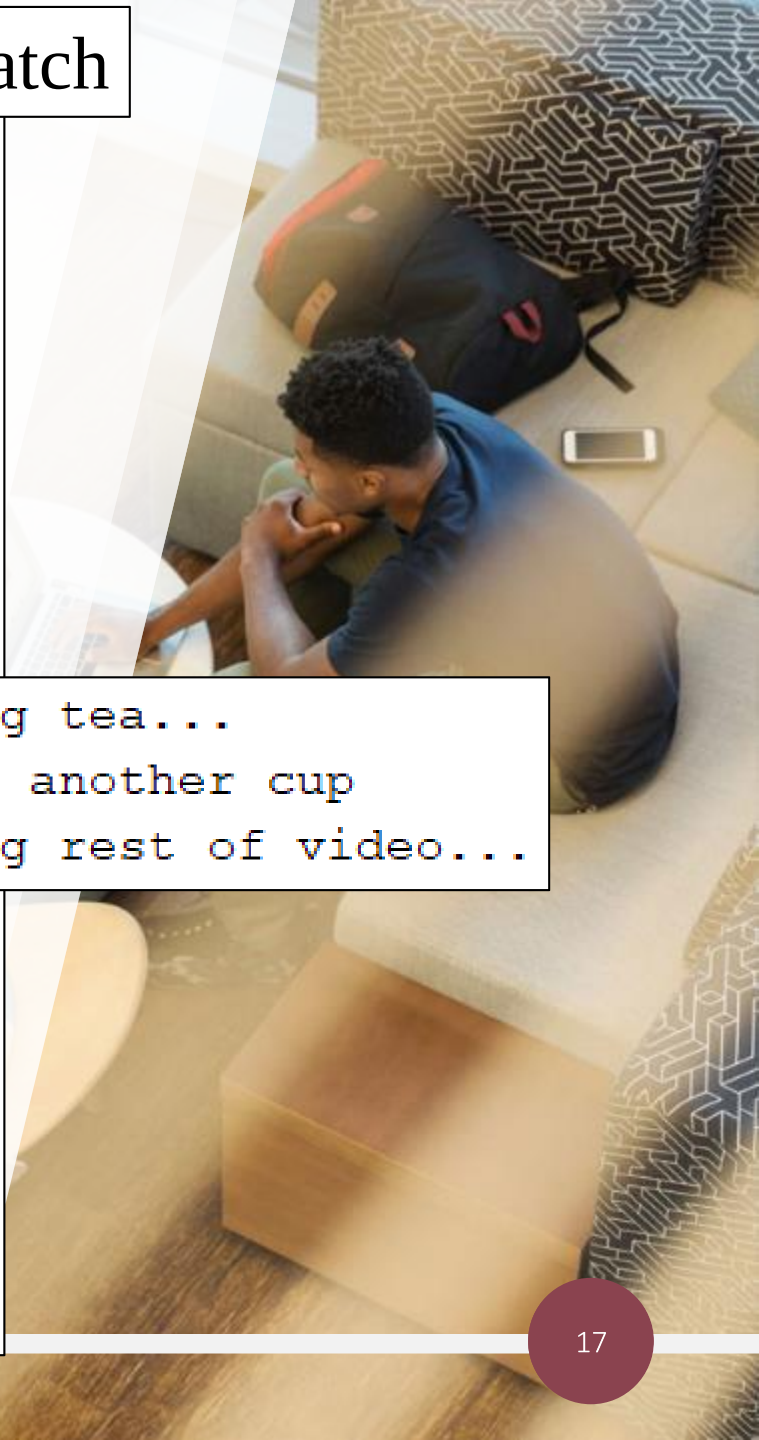


try-catch

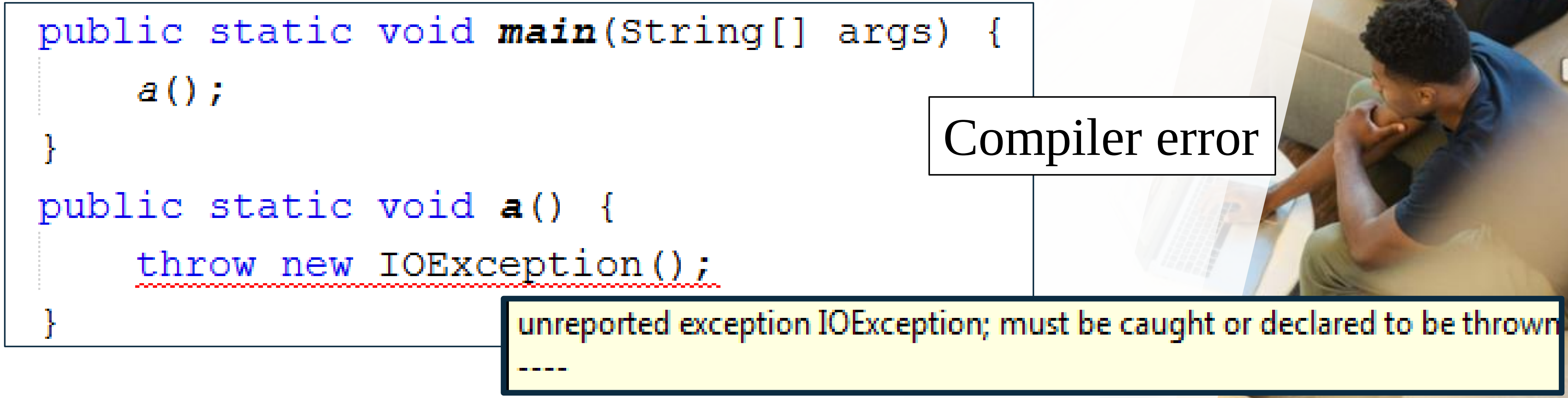


```java
public class TestExceptions {
    public static void main(String[] args) {
        try{
            spillTea();
            System.out.println("Will never get here!");
        }catch(RuntimeException rte){
            getAnotherCup();
        }
        enjoyRestOfVideo();
    }
    public static void spillTea(){
        System.out.println("Spilling tea... ");
        throw new RuntimeException();
    }
    public static void getAnotherCup(){
        System.out.println("Getting another cup");
    }
    public static void enjoyRestOfVideo(){
        System.out.println("Enjoying rest of video...");
    }
}
```

```
Spilling tea...
Getting another cup
Enjoying rest of video...
```



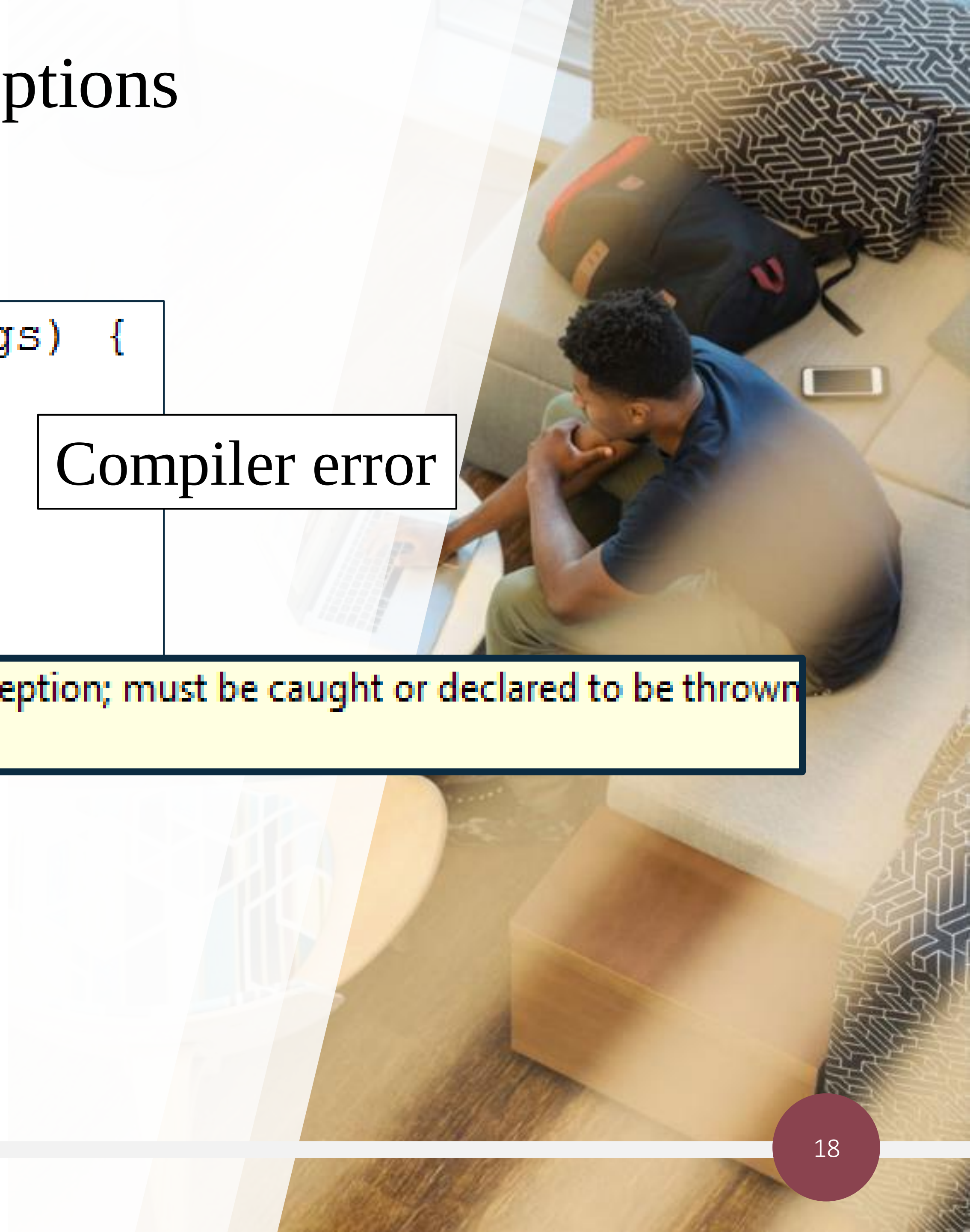
Catch or Declare Checked Exceptions



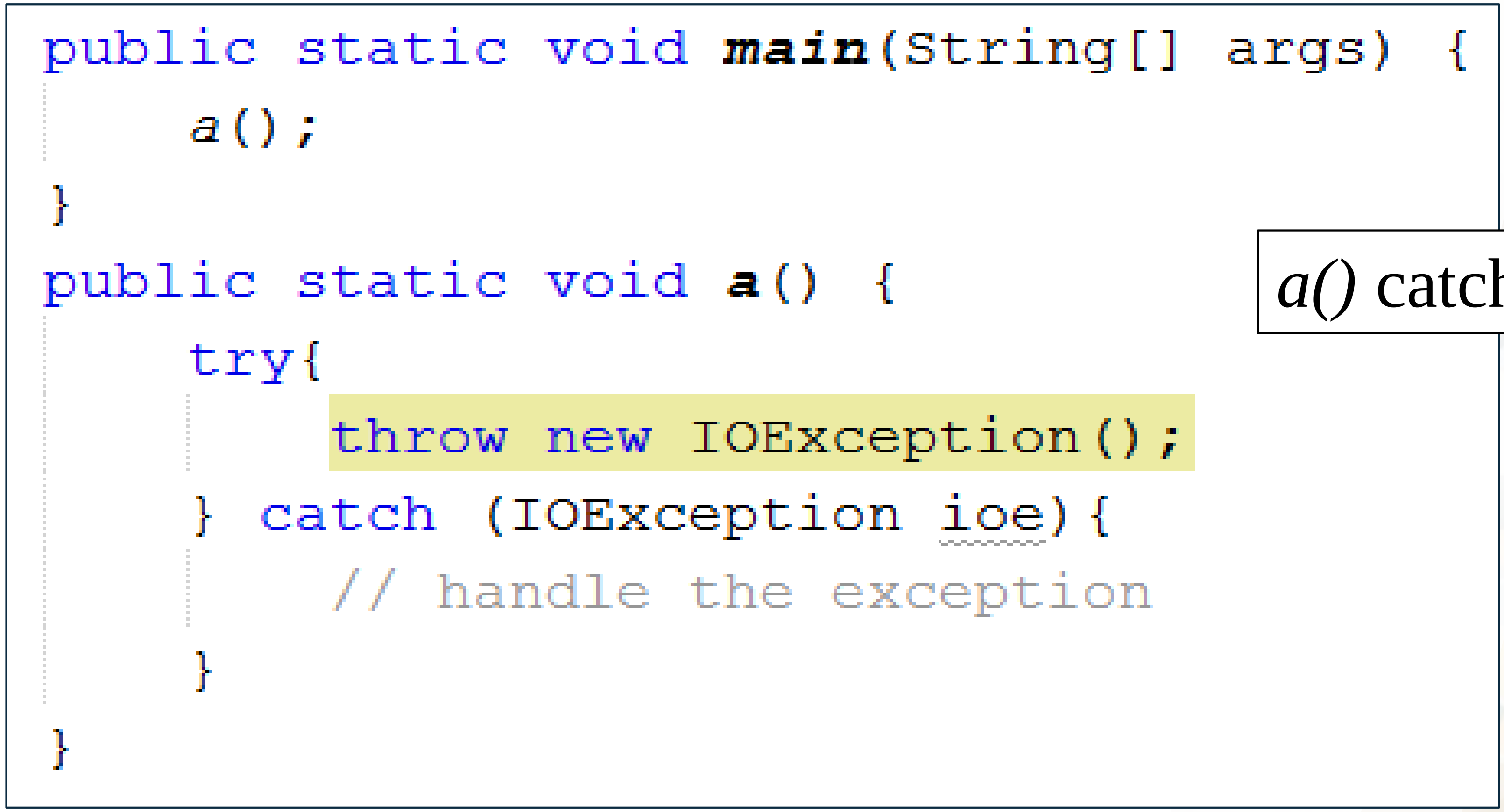
```
public static void main(String[] args) {
    a();
}
public static void a() {
    throw new IOException();
}
```

unreported exception IOException; must be caught or declared to be thrown
----

Compiler error



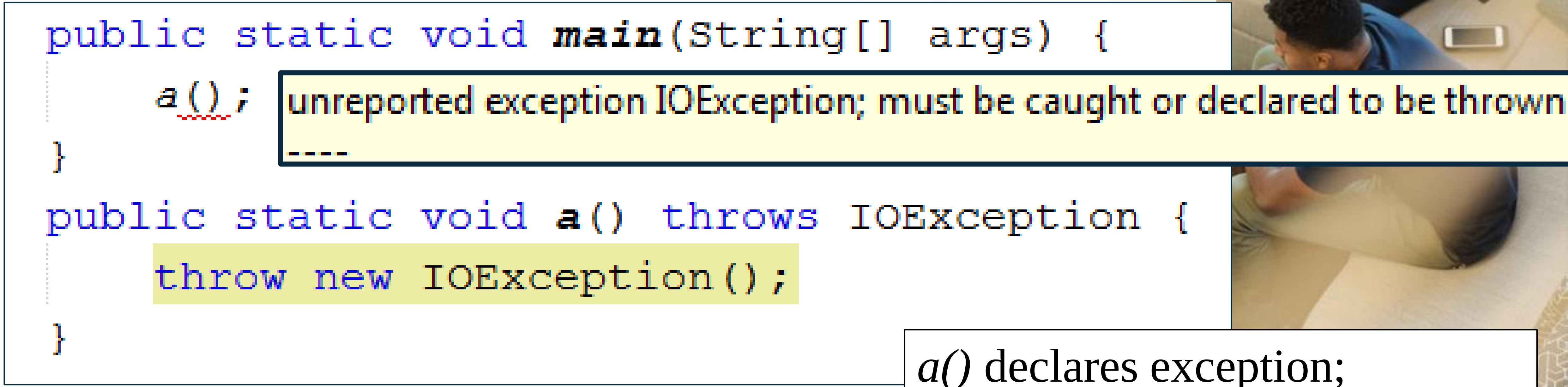
Catch or Declare Checked Exceptions



```java
public static void main(String[] args) {
    a();
}
public static void a() {
    try{
        throw new IOException();
    } catch (IOException ioe){
        // handle the exception
    }
}
```

a() catches exception

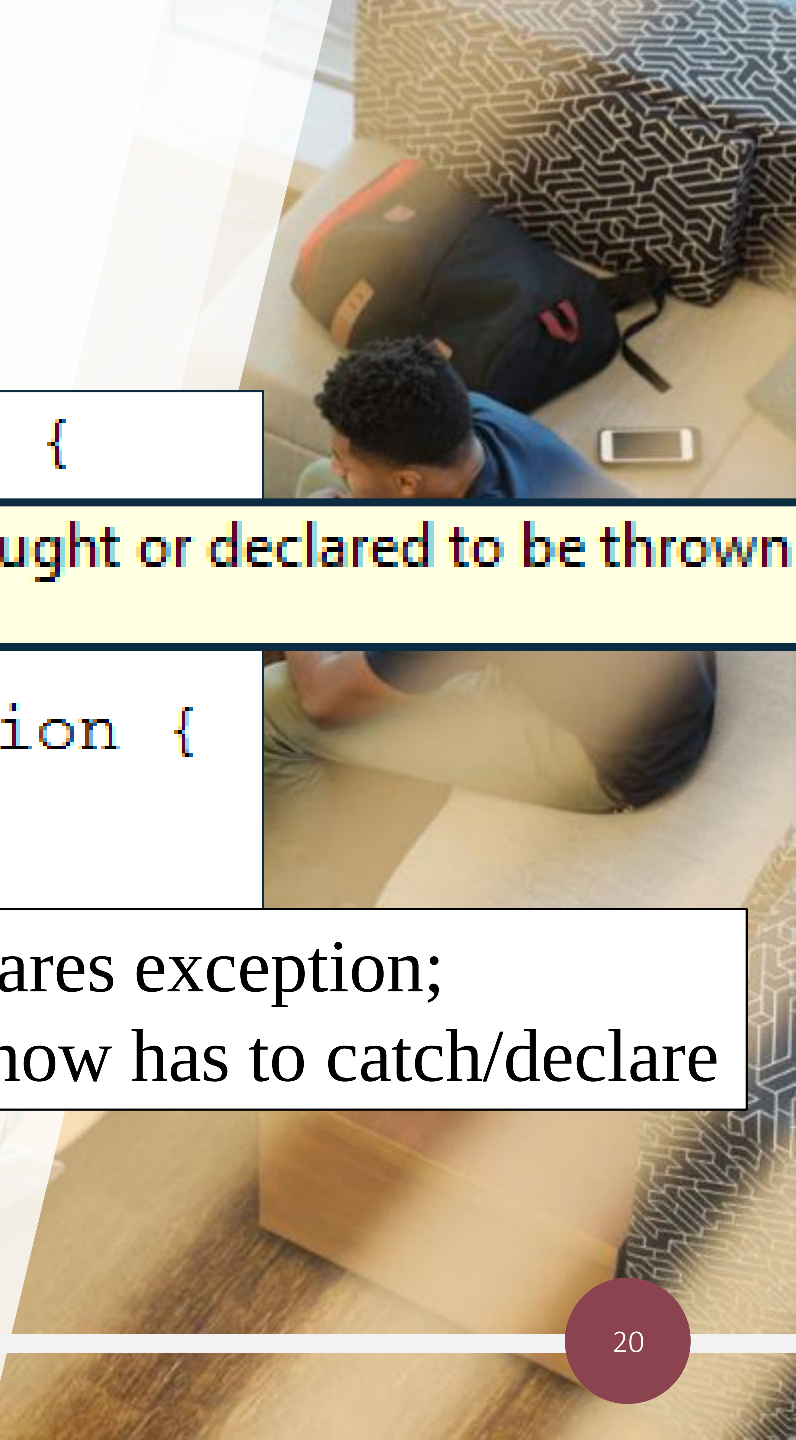
Catch or **Declare** Checked Exceptions



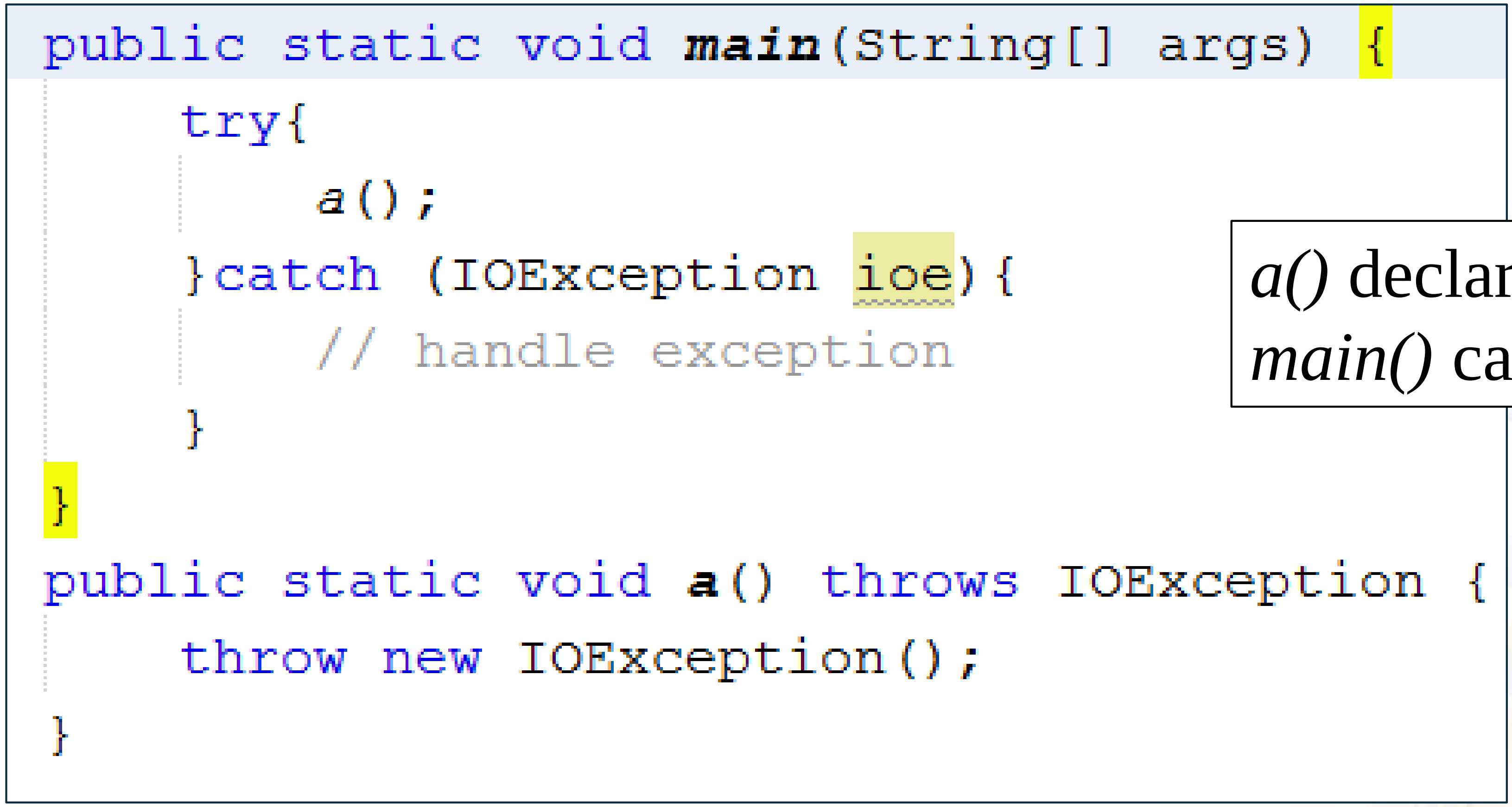
```
public static void main(String[] args) {
    a();
}
public static void a() throws IOException {
    throw new IOException();
}
```

```
unreported exception IOException; must be caught or declared to be thrown
----
```

a() declares exception;
main() now has to catch/declare

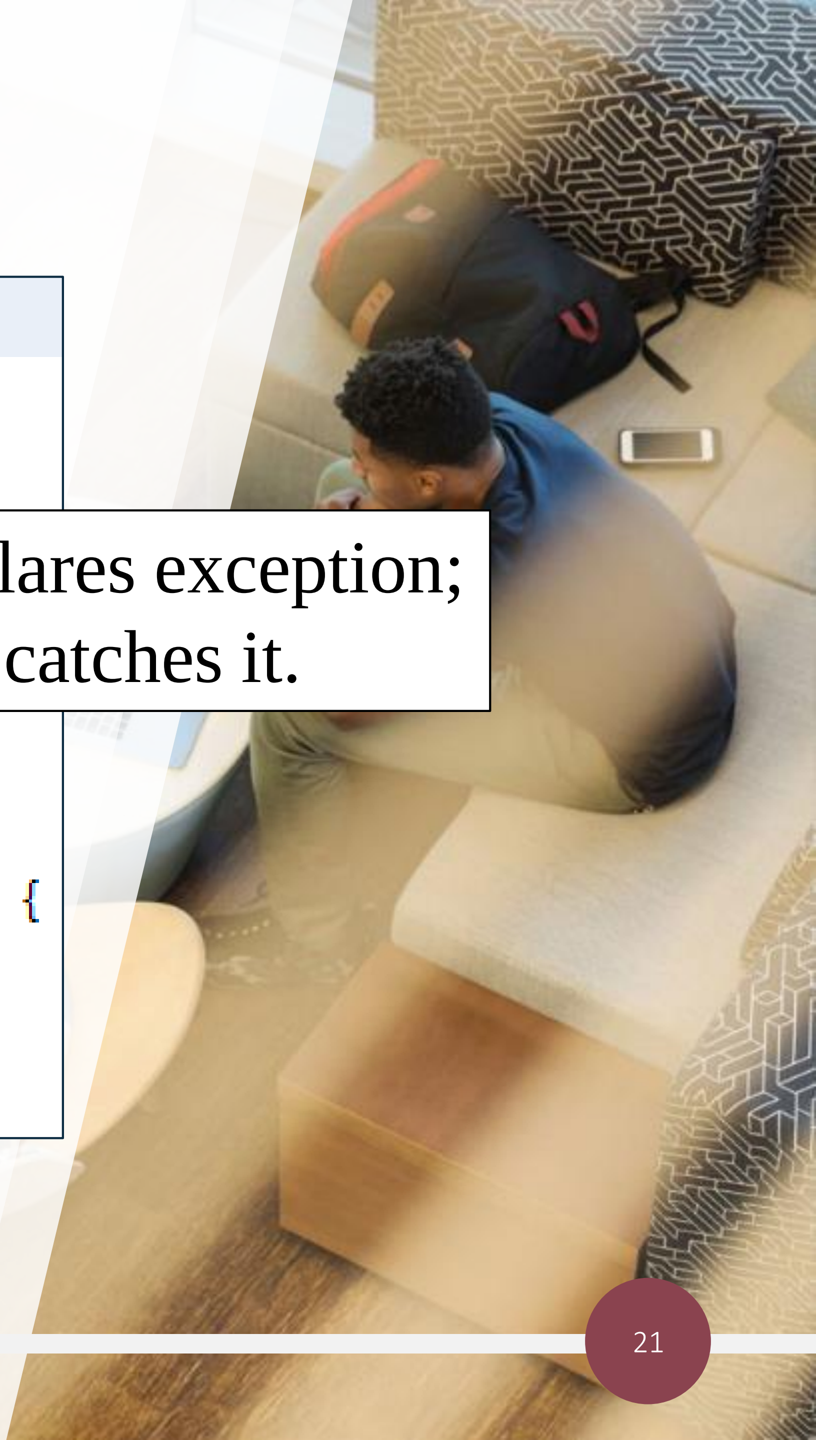


Catch or Declare Checked Exceptions

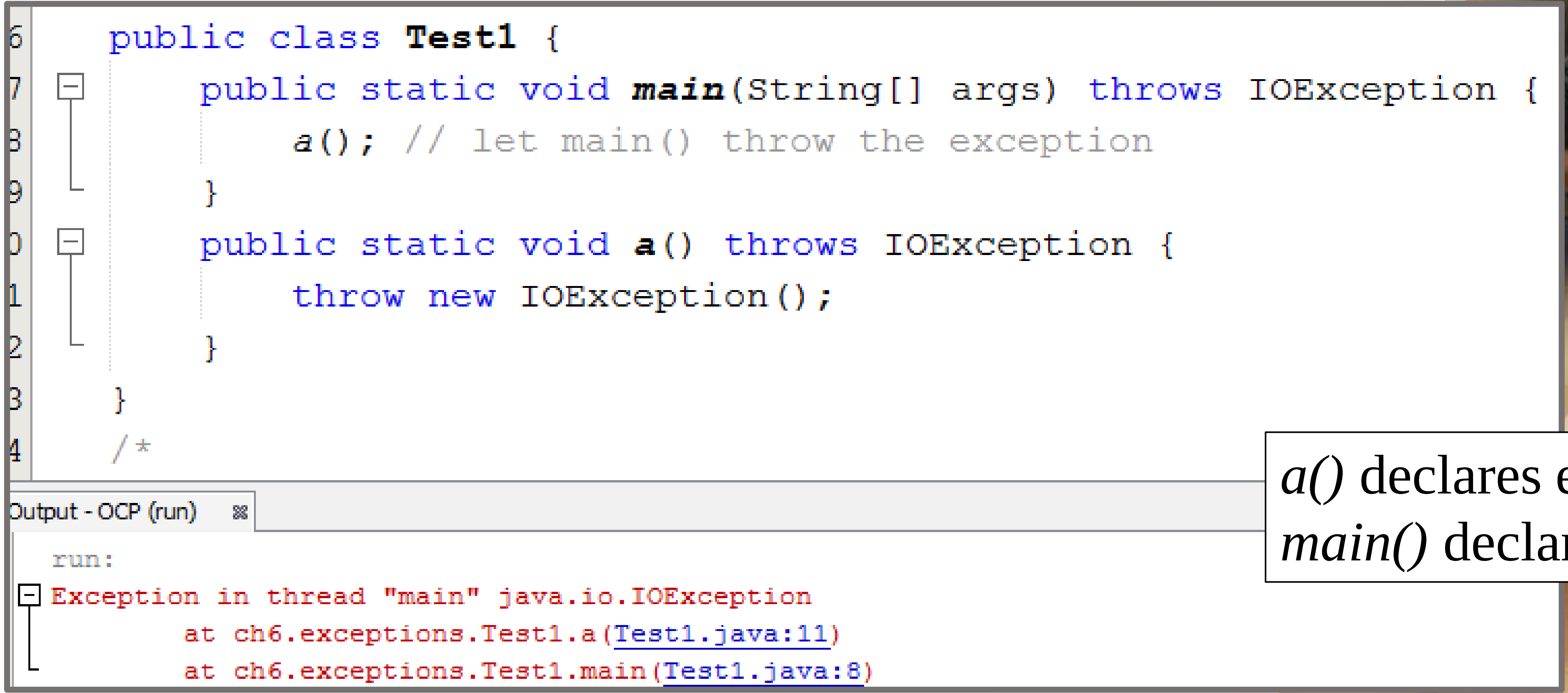


```
public static void main(String[] args) {
    try{
        a();
    }catch (IOException ioe){
        // handle exception
    }
}
public static void a() throws IOException {
    throw new IOException();
}
```

a() declares exception;
main() catches it.



Catch or Declare Checked Exceptions



```java
public class Test1 {
    public static void main(String[] args) throws IOException {
        a(); // let main() throw the exception
    }
    public static void a() throws IOException {
        throw new IOException();
    }
}
/*
```

Output - OCP (run)

```
run:
Exception in thread "main" java.io.IOException
        at ch6.exceptions.Test1.a(Test1.java:11)
        at ch6.exceptions.Test1.main(Test1.java:8)
```

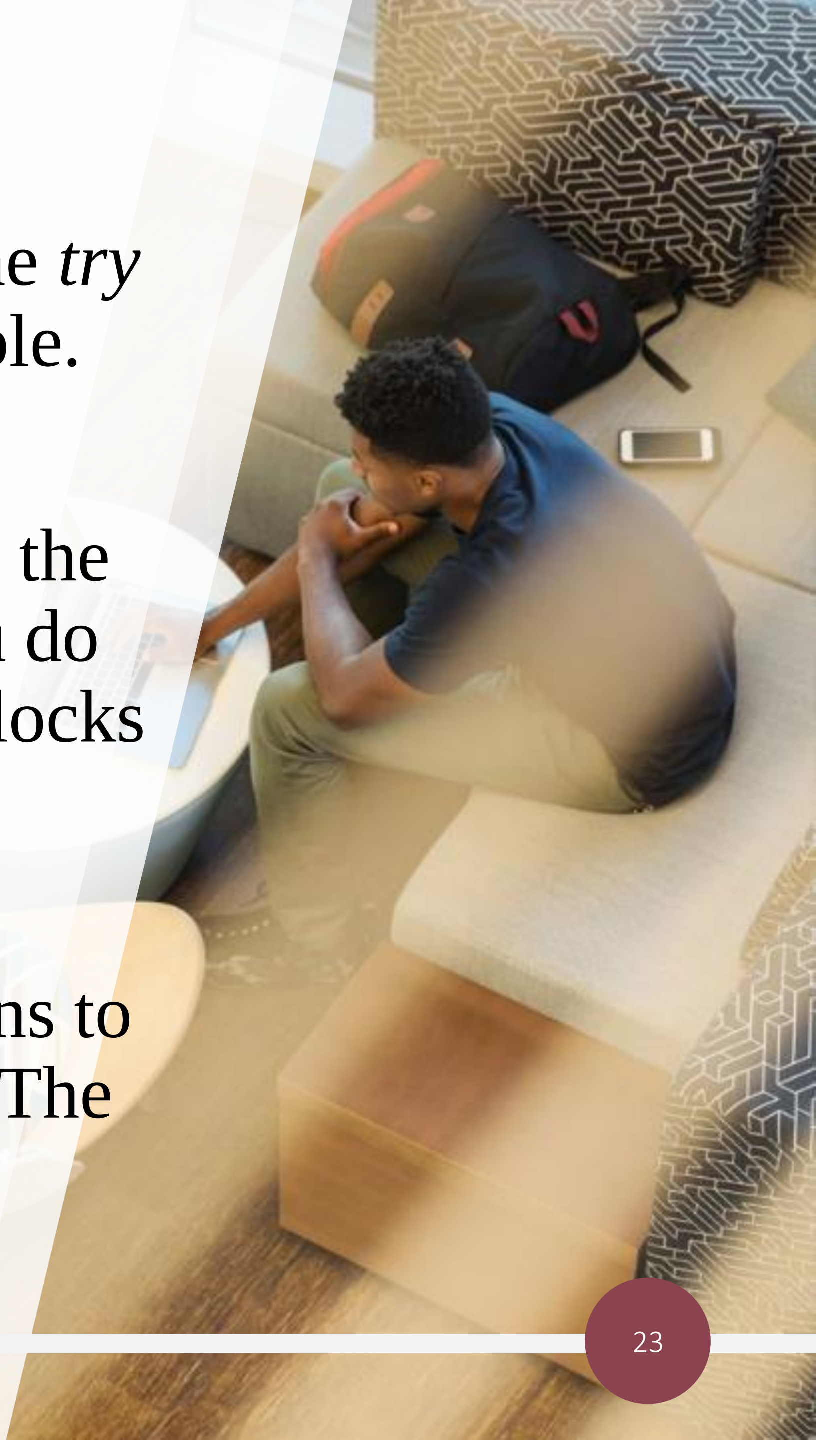
a() declares exception; *main()* declares it also.

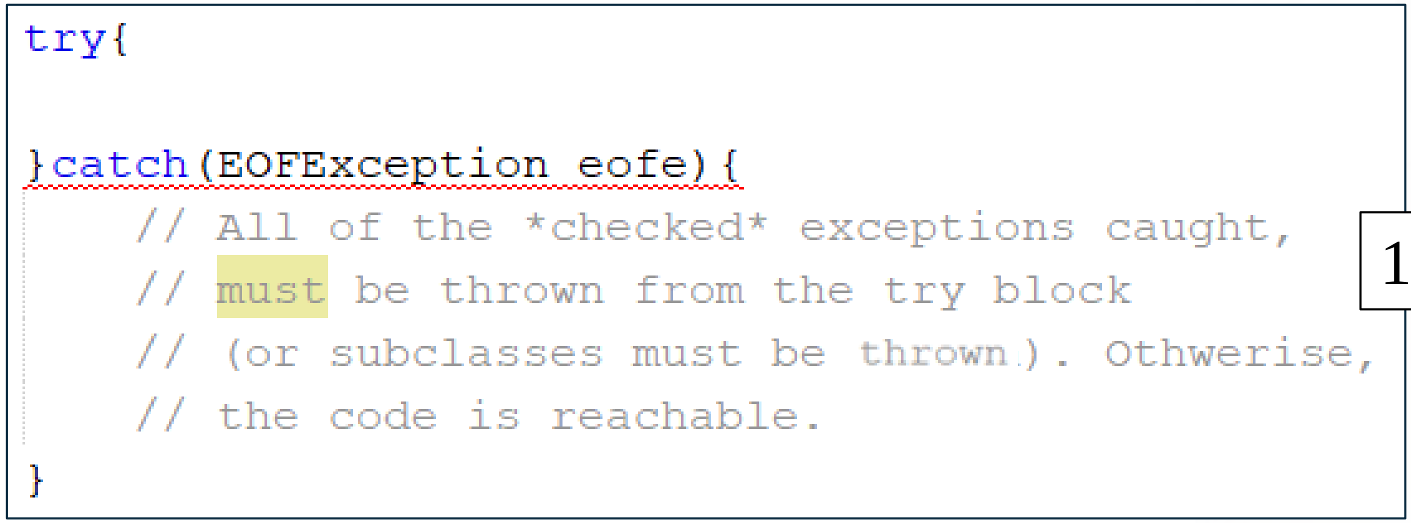
try and *catch* blocks

- All caught <u>checked</u> exceptions must be thrown from the *try* block; otherwise the relevant catch blocks are unreachable.

- Multiple *catch* blocks are allowed and are evaluated in the order that they are coded. Thus, care is required that you do not have “unreachable code” errors i.e. code the *catch* blocks in the order of the most specific to the least specific.

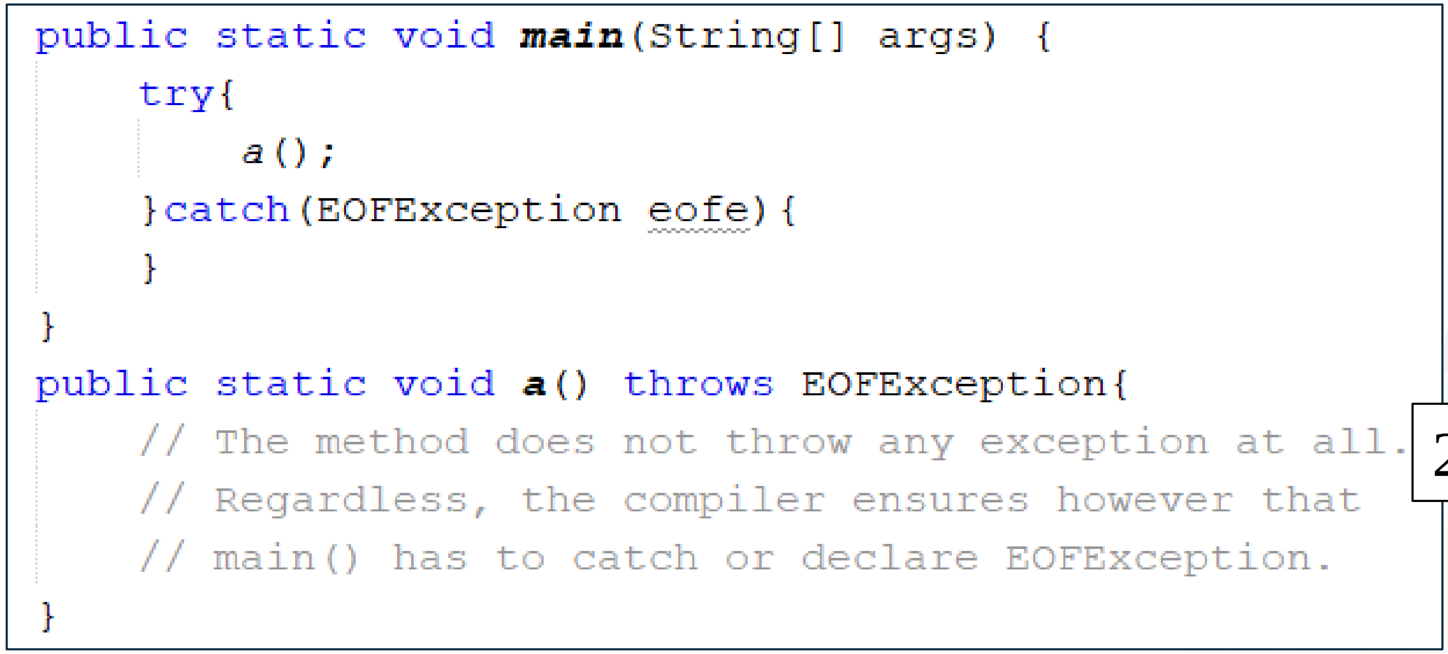
- Multi-*catch* blocks enable unrelated (sibling) exceptions to be handled together, thereby reducing code duplication. The identifier used must appear only once.





```java
try{

}catch(EOFException eofe){
    // All of the *checked* exceptions caught,
    // must be thrown from the try block
    // (or subclasses must be thrown). Othwerise,
    // the code is reachable.
}
```

1.

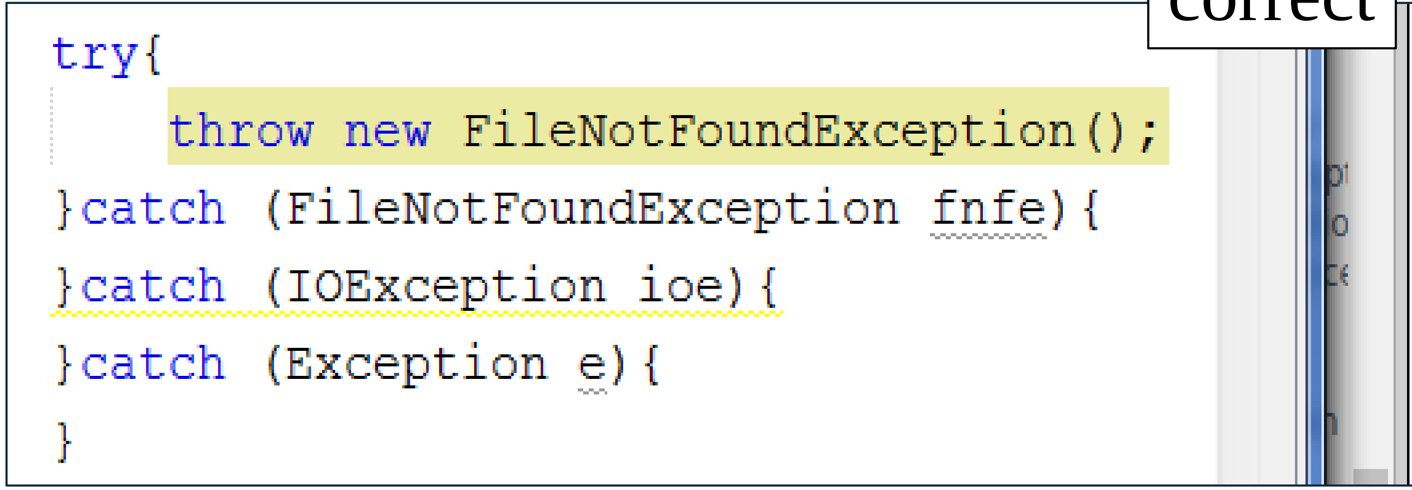


```java
public static void main(String[] args) {
    try{
        a();
    }catch(EOFException eofe){
    }
}
public static void a() throws EOFException{
    // The method does not throw any exception at all.
    // Regardless, the compiler ensures however that
    // main() has to catch or declare EOFException.
}
```

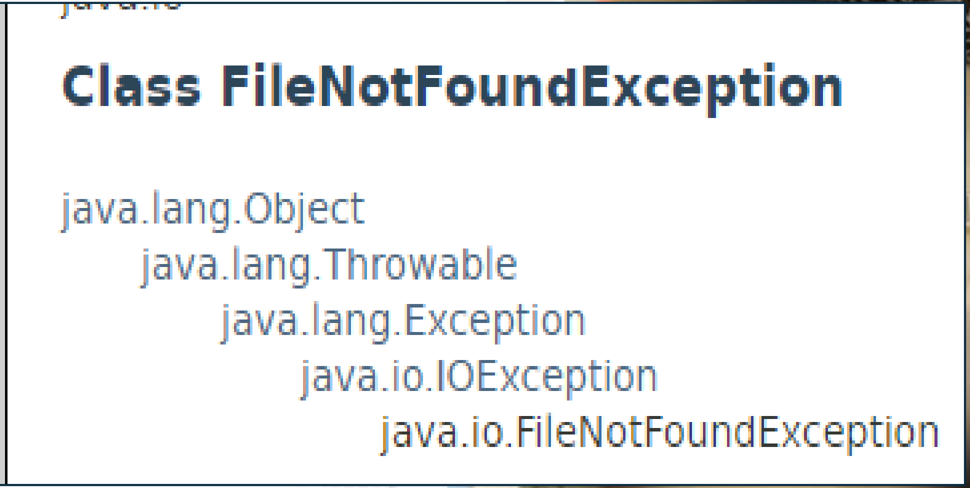
2.

try and *catch* blocks

correct



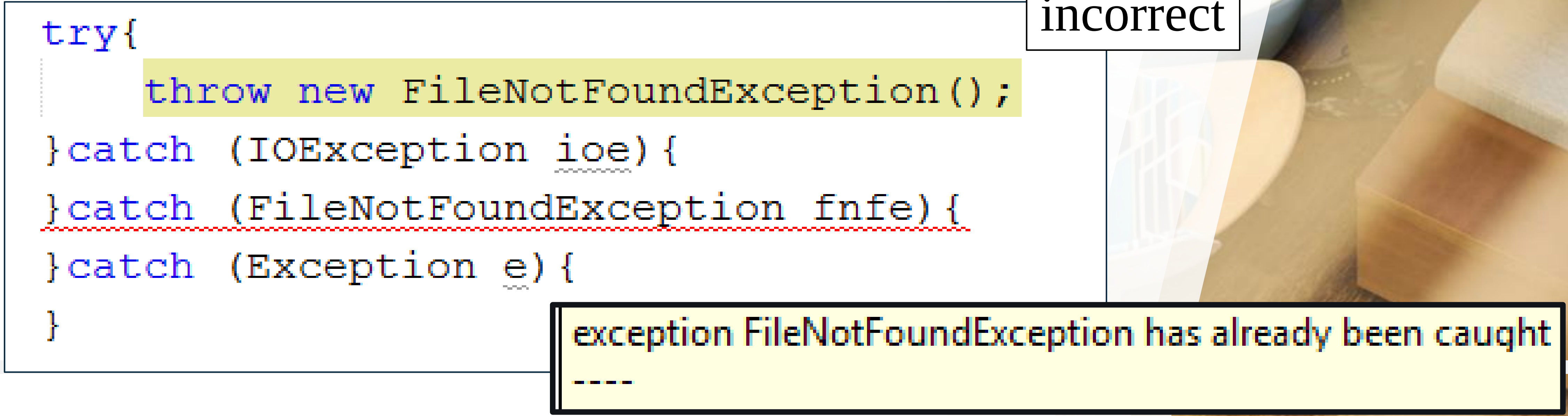
```
try{
    throw new FileNotFoundException();
}catch (FileNotFoundException fnfe){
}catch (IOException ioe){
}catch (Exception e){
}
```



Class FileNotFoundException

```
java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.io.IOException
                java.io.FileNotFoundException
```

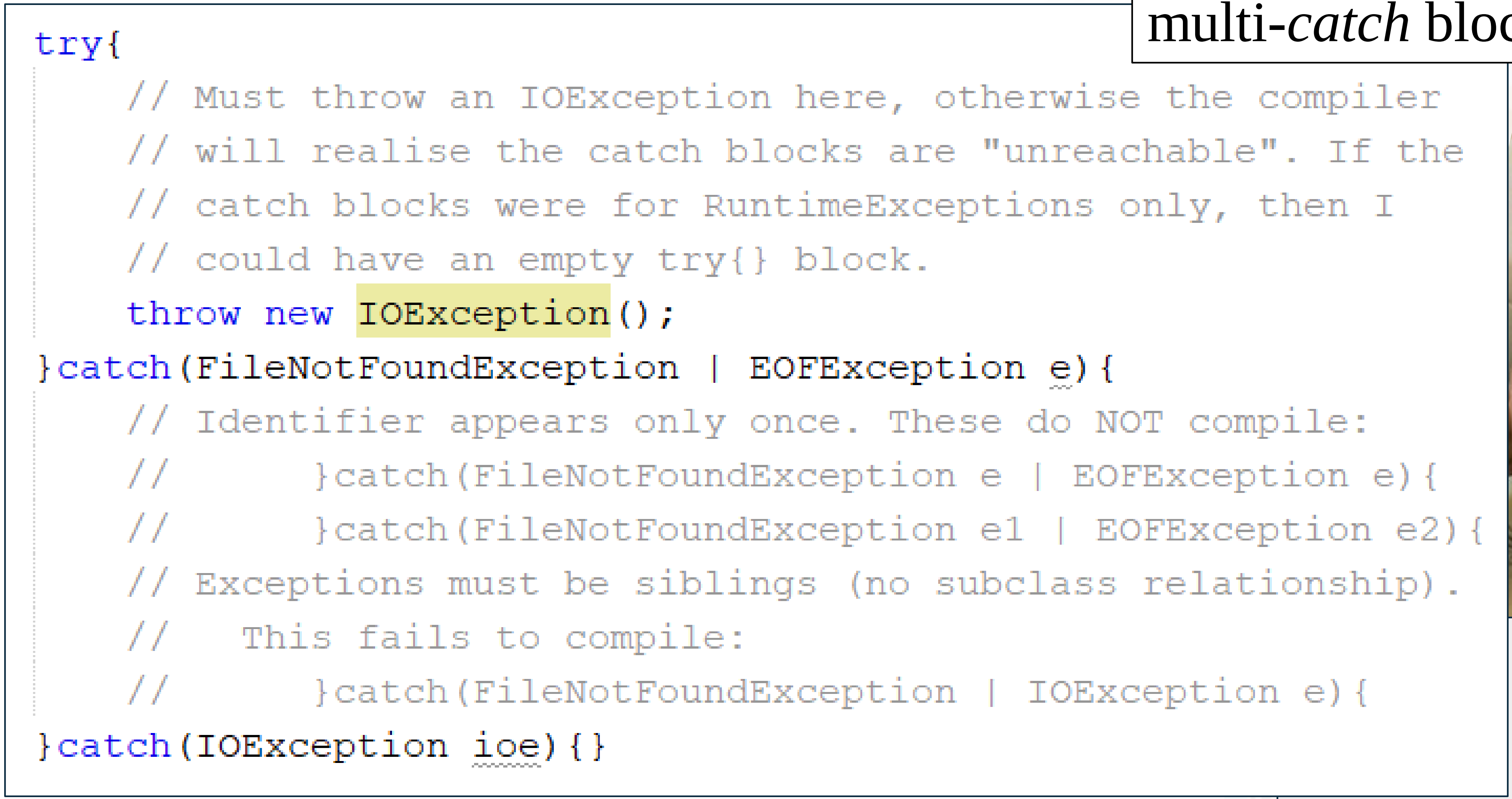
incorrect



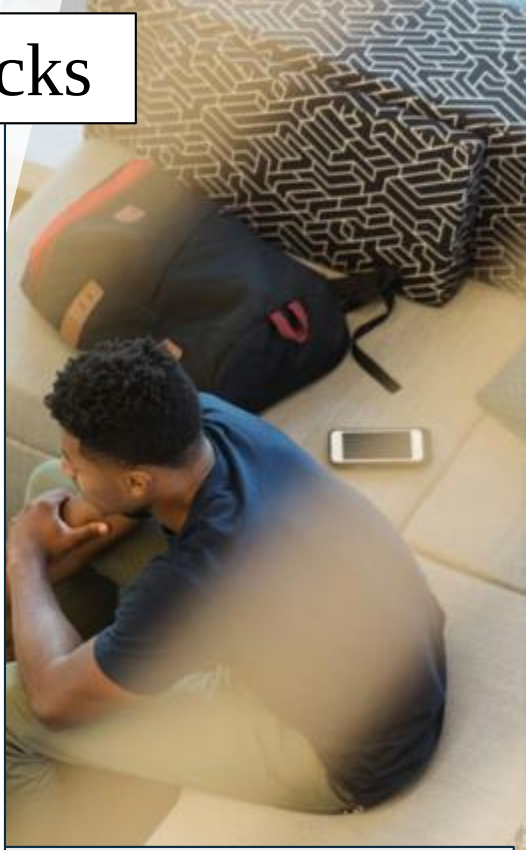
```
try{
    throw new FileNotFoundException();
}catch (IOException ioe){
}catch (FileNotFoundException fnfe){
}catch (Exception e){
}
```

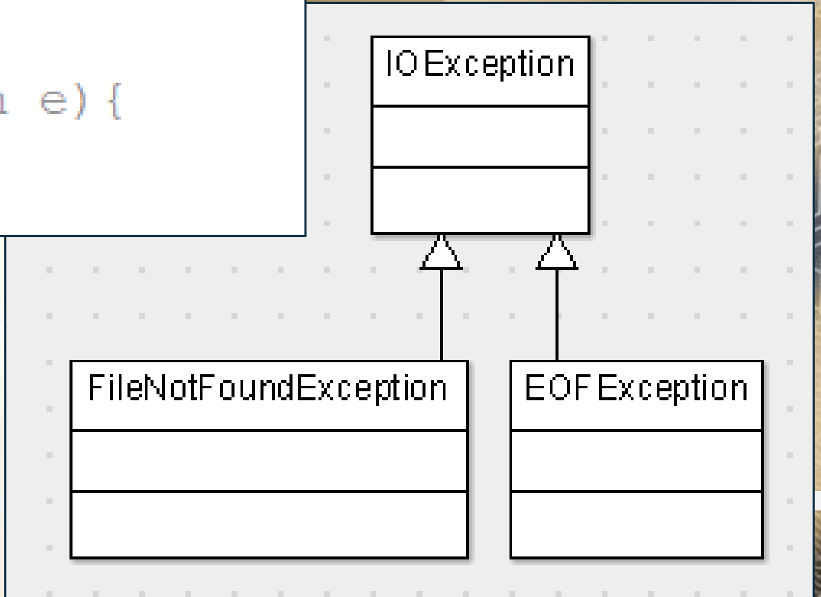
exception FileNotFoundException has already been caught
----

multi-*catch* blocks



```
try{
    // Must throw an IOException here, otherwise the compiler
    // will realise the catch blocks are "unreachable". If the
    // catch blocks were for RuntimeExceptions only, then I
    // could have an empty try{} block.
    throw new IOException();
}catch(FileNotFoundException | EOFException e){
    // Identifier appears only once. These do NOT compile:
    //      }catch(FileNotFoundException e | EOFException e){
    //      }catch(FileNotFoundException e1 | EOFException e2){
    // Exceptions must be siblings (no subclass relationship).
    //   This fails to compile:
    //      }catch(FileNotFoundException | IOException e){
}catch(IOException ioe){}
```



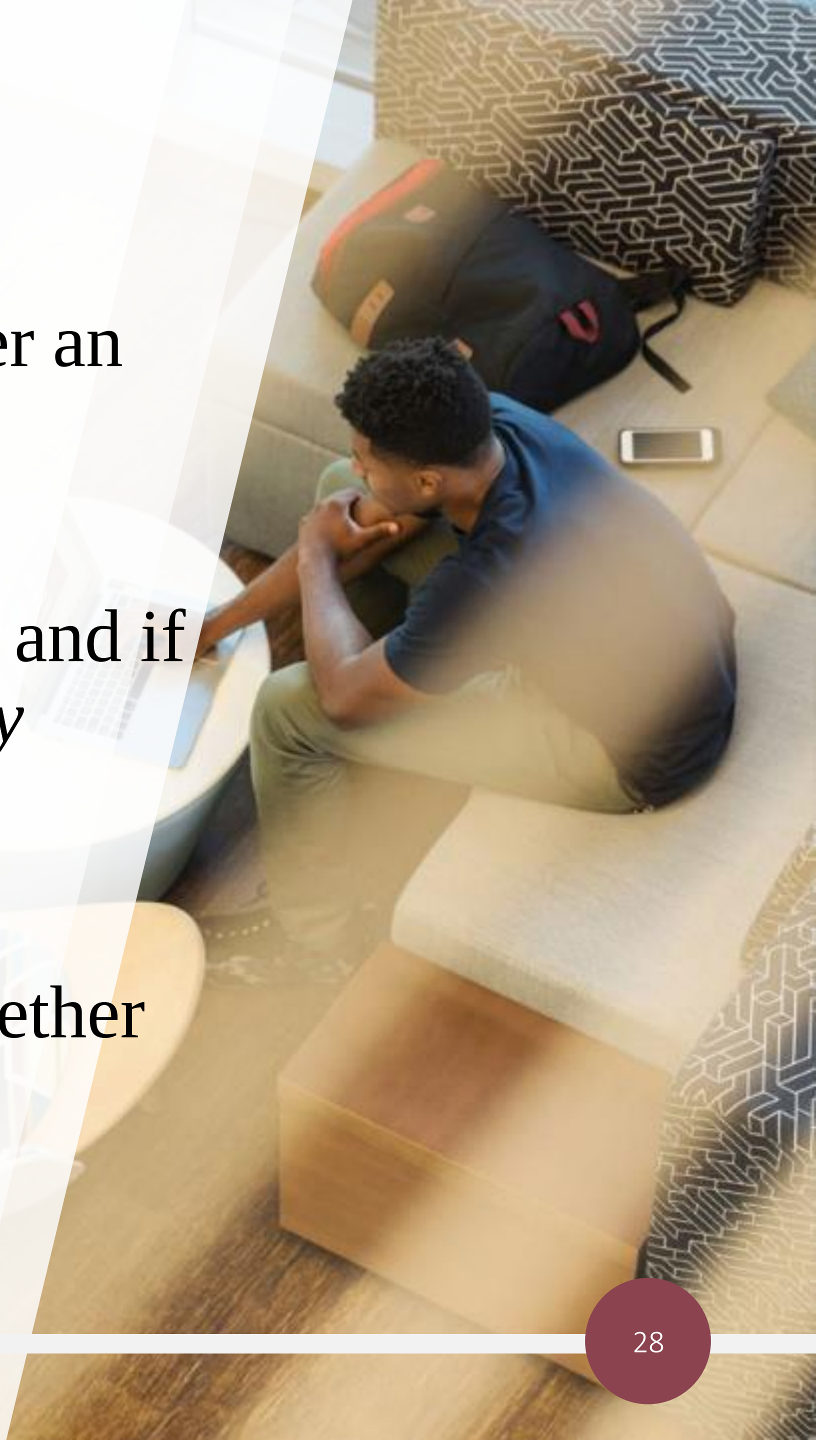


finally blocks

- The *finally* block is designed for tidying up resources (e.g. file and database connections) regardless of whether an exception occurs or not.

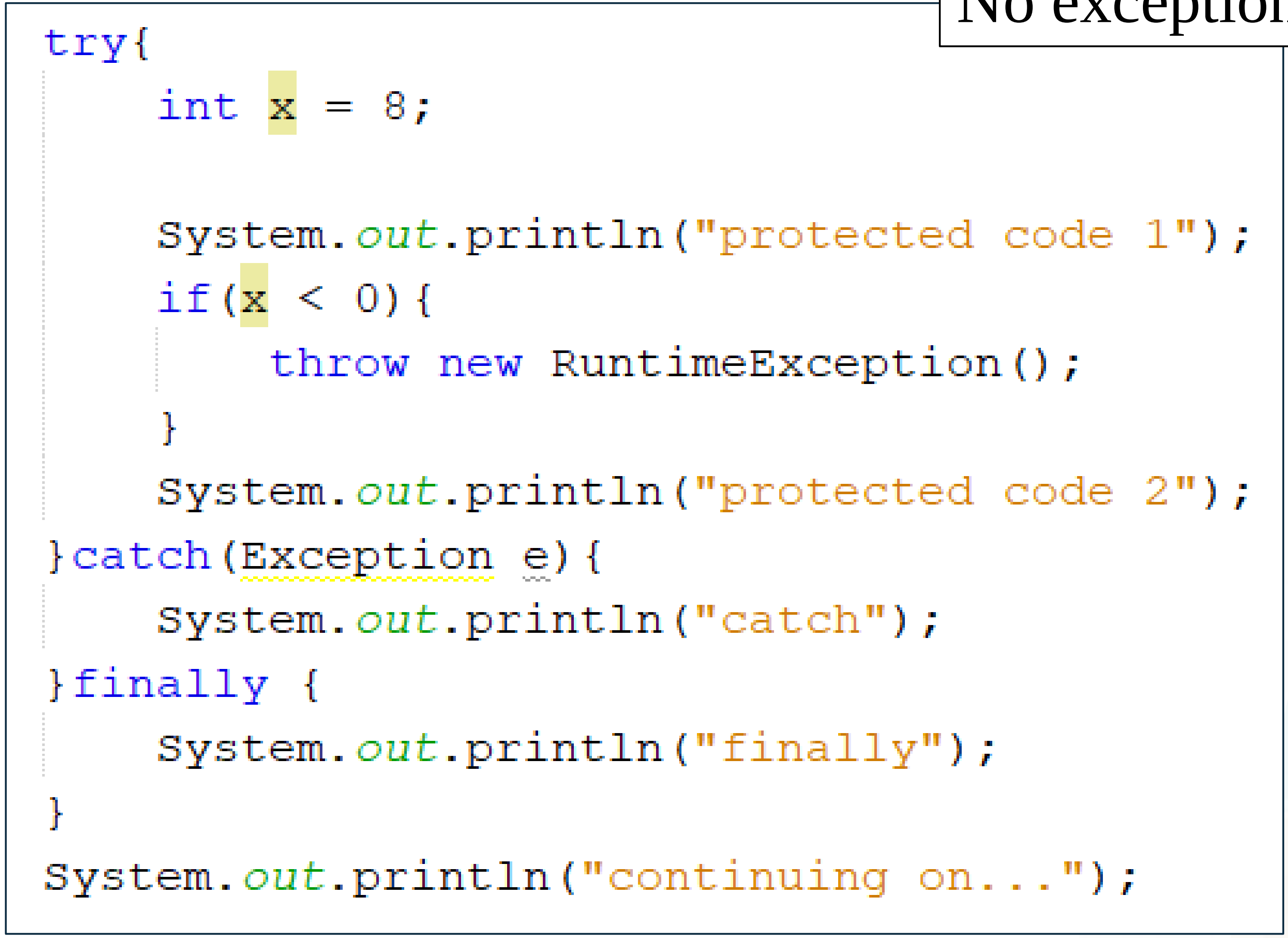
- If present, the *finally* block is after the last *catch* block and if there are no *catch* blocks, the *finally* block is after the *try* block.

- The *finally* block is **always** executed, regardless of whether an exception is thrown or not.

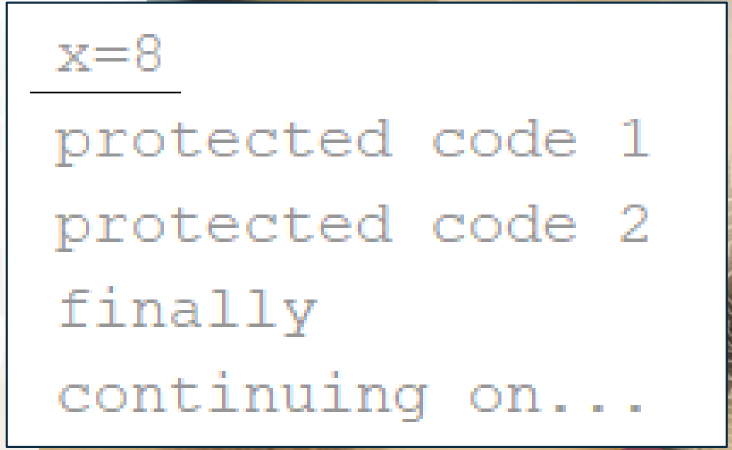


finally blocks

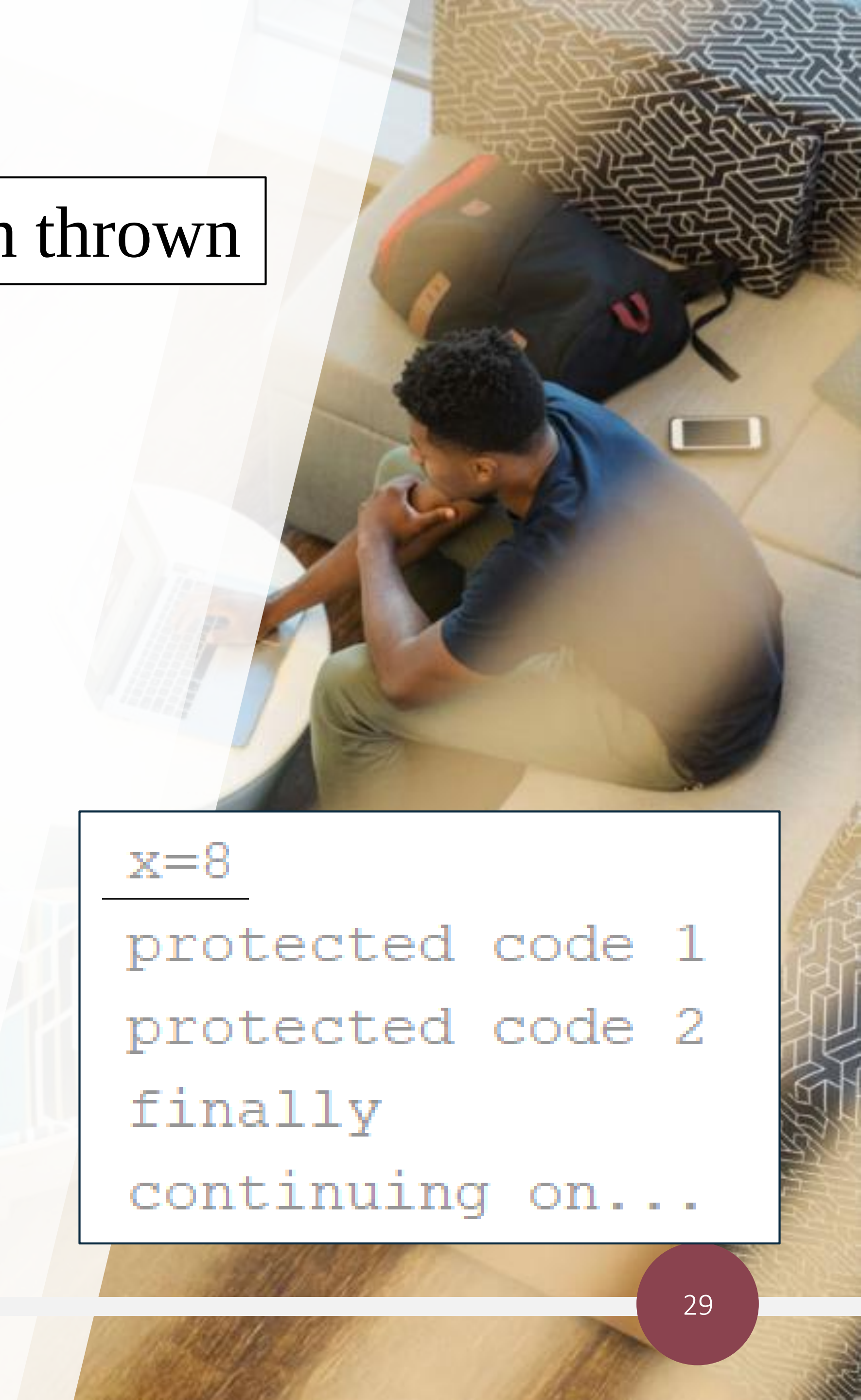
No exception thrown



```
try{
    int x = 8;

    System.out.println("protected code 1");
    if(x < 0){
        throw new RuntimeException();
    }
    System.out.println("protected code 2");
}catch(Exception e){
    System.out.println("catch");
}finally {
    System.out.println("finally");
}
System.out.println("continuing on...");
```

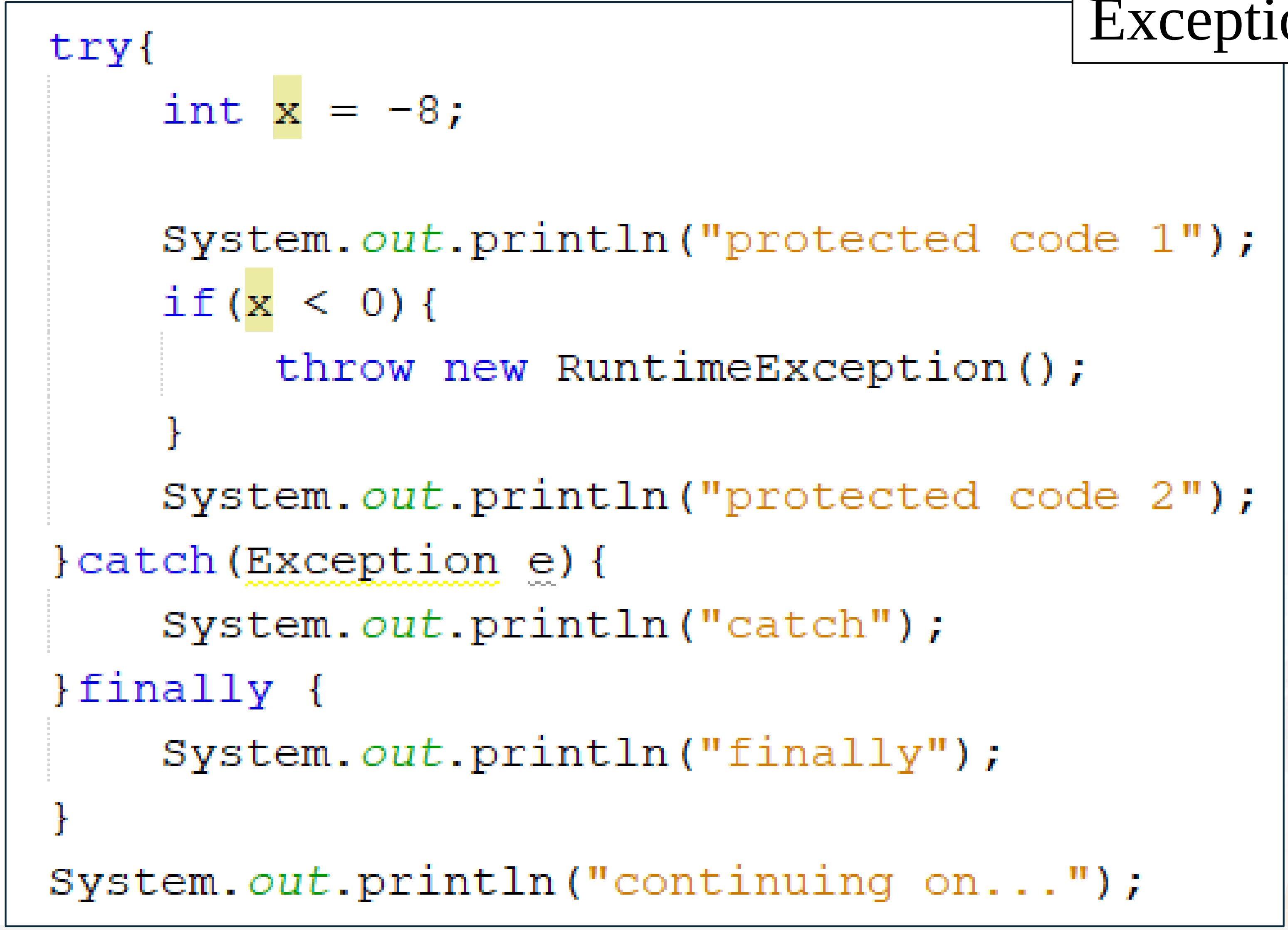


```
x=8
protected code 1
protected code 2
finally
continuing on...
```

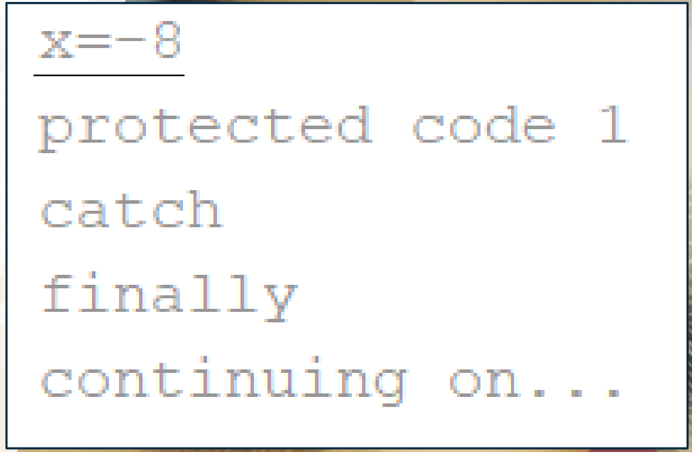


finally blocks

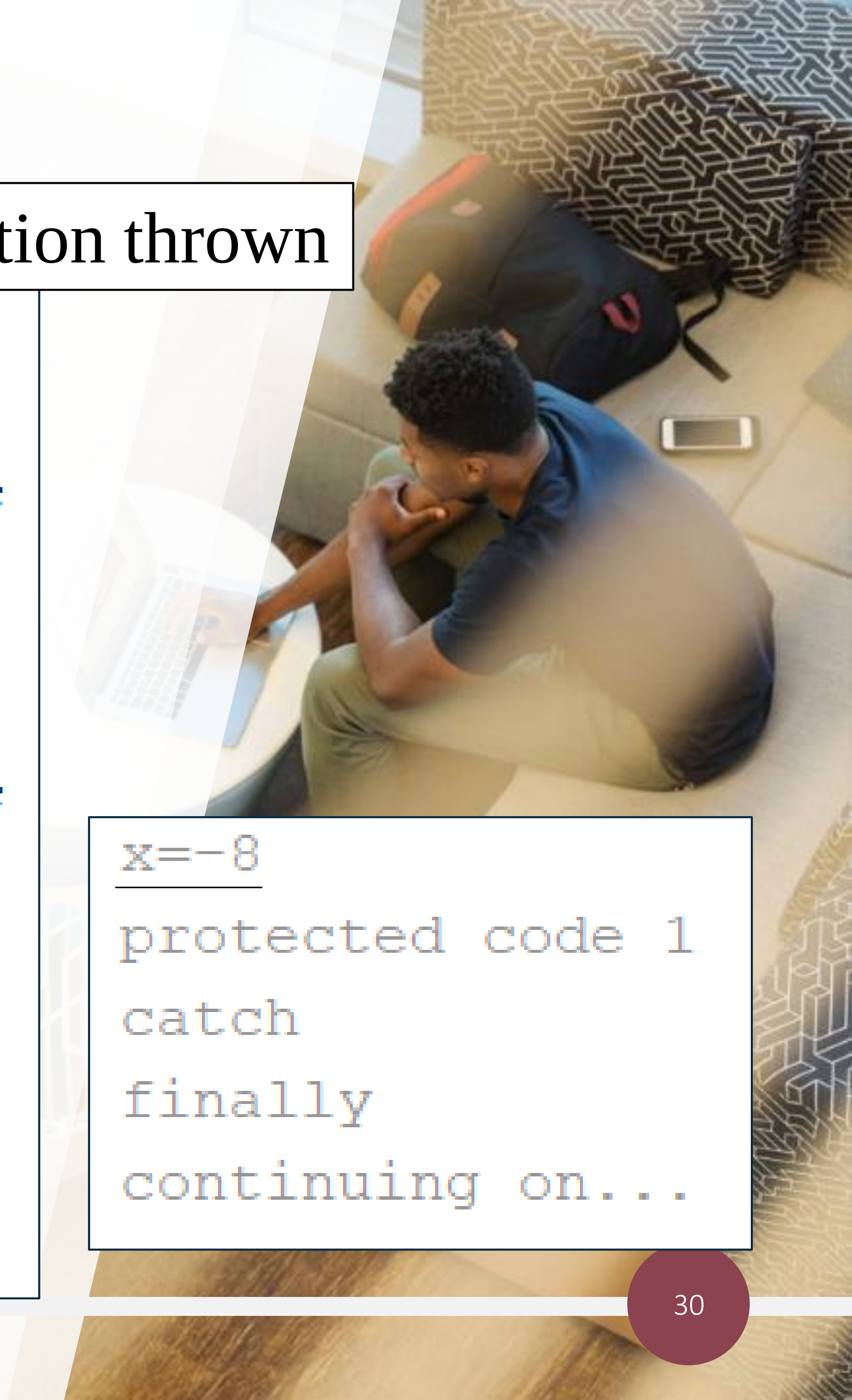
Exception thrown



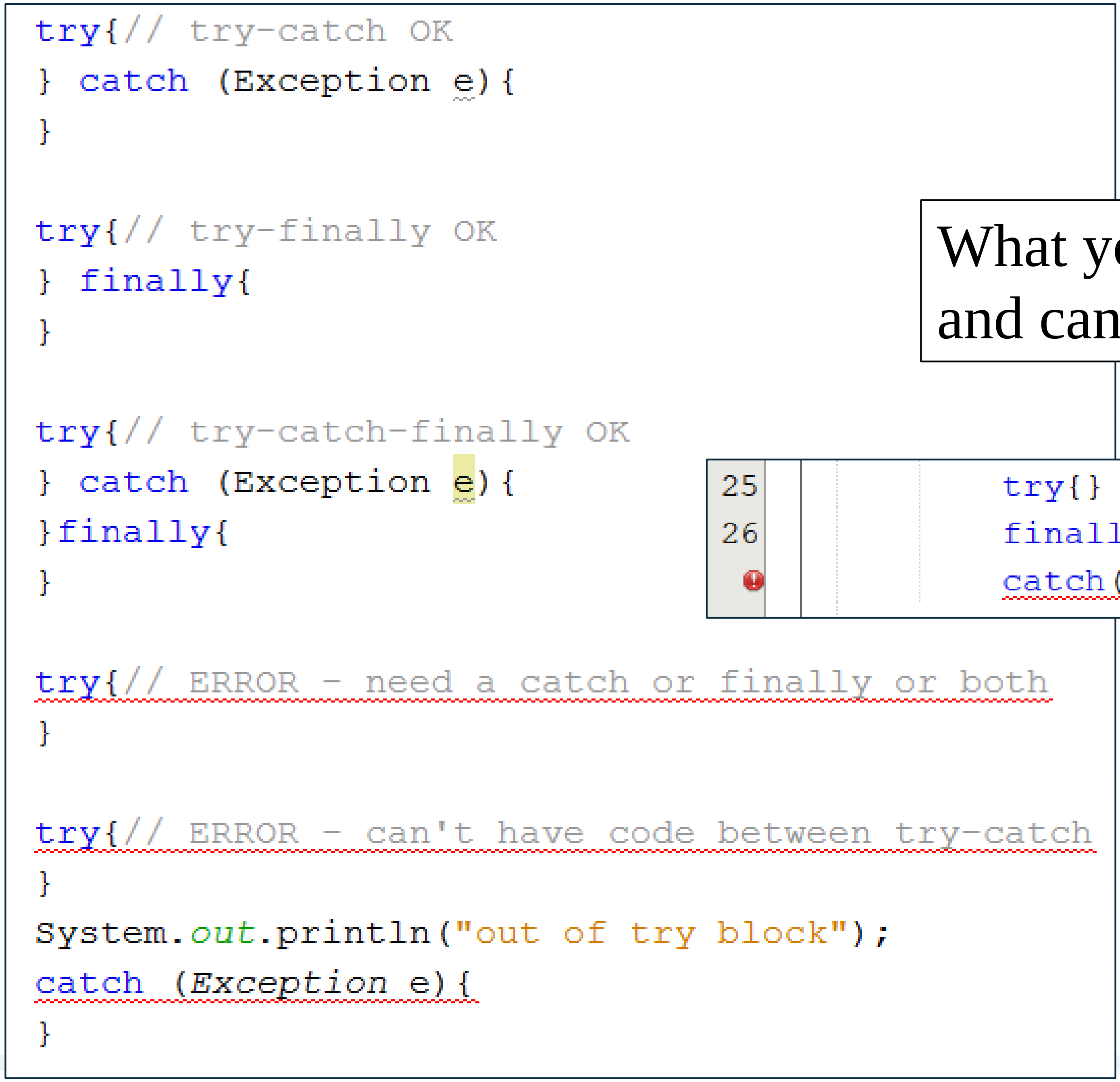
```
try{
    int x = -8;

    System.out.println("protected code 1");
    if(x < 0){
        throw new RuntimeException();
    }
    System.out.println("protected code 2");
}catch(Exception e){
    System.out.println("catch");
}finally {
    System.out.println("finally");
}
System.out.println("continuing on...");
```



```
x=-8
protected code 1
catch
finally
continuing on...
```

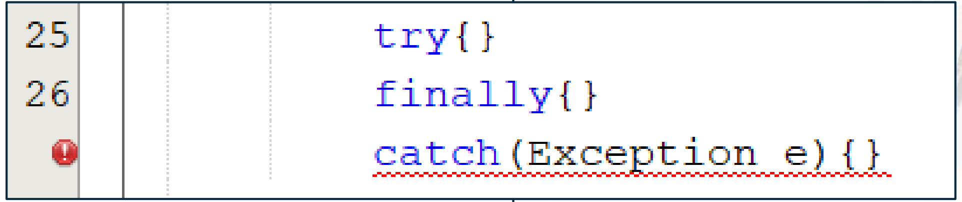


finally blocks



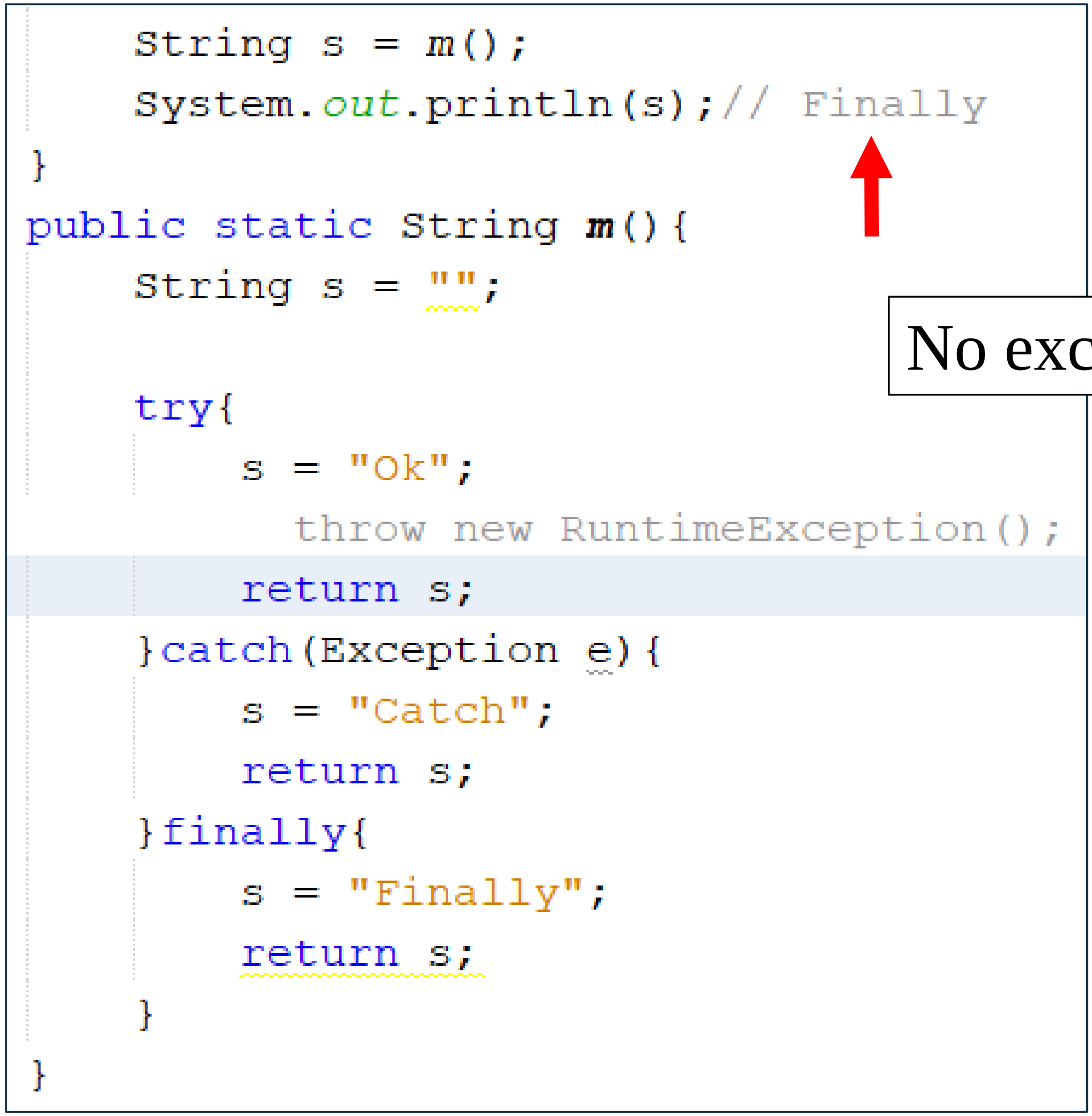
```
try{// try-catch OK
} catch (Exception e){
}

try{// try-finally OK
} finally{
}

try{// try-catch-finally OK
} catch (Exception e){
}finally{
}

try{// ERROR - need a catch or finally or both
}

try{// ERROR - can't have code between try-catch
}
System.out.println("out of try block");
catch (Exception e){
}
```

What you can and cannot do.



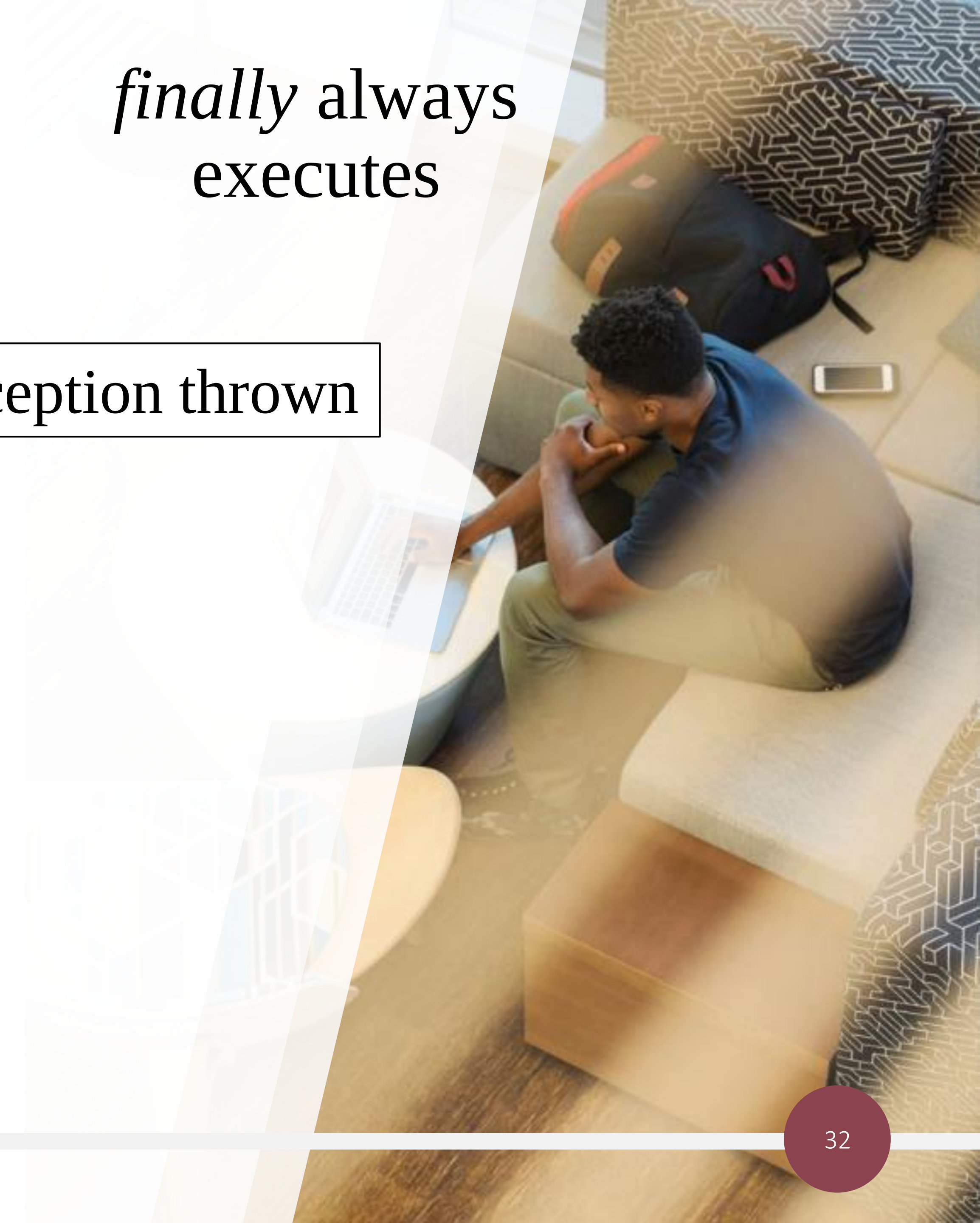
```
try{}
finally{}
catch(Exception e){}
```

finally always executes

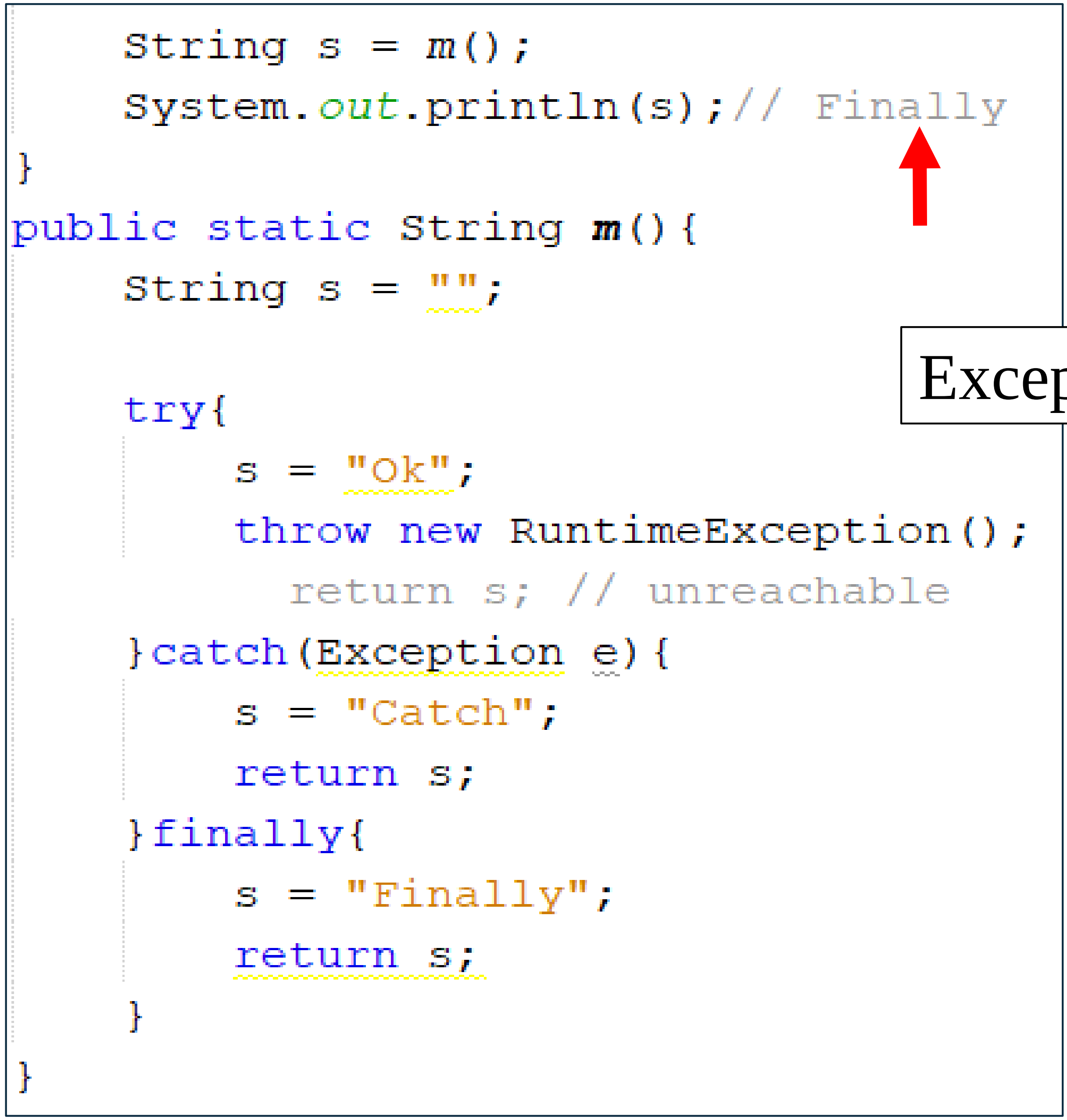


```
    String s = m();
    System.out.println(s);// Finally
}
public static String m(){
    String s = "";

    try{
        s = "Ok";
          throw new RuntimeException();
        return s;
    }catch(Exception e){
        s = "Catch";
        return s;
    }finally{
        s = "Finally";
        return s;
    }
}
```

No exception thrown



finally always executes



```
    String s = m();
    System.out.println(s);// Finally
}
public static String m(){
    String s = "";

    try{
        s = "Ok";
        throw new RuntimeException();
          return s; // unreachable
    }catch(Exception e){
        s = "Catch";
        return s;
    }finally{
        s = "Finally";
        return s;
    }
}
```

Exception occurs

try-with-resources statement

- Closing resources in the *finally* block can get lengthy as the closing of these resources needs to be wrapped in *try-catch* blocks also.

- Java introduced the *try-with-resources* statement for this reason.

- Resources opened in the *try* clause are automatically closed by Java in an **implicit** *finally* block.

- You can code your own explicit *finally* block but the implicit one will be called first.

try-with-resources statement

- The rule that a *try* requires a *catch* or *finally* still applies; however, given that the *finally* block is implicit, the *catch* is no longer mandatory.

- This is only for the *try-with-resources* statement. The traditional *try* block requires a *catch* or *finally* (or both) .

- try{
  }catch{
  }

  try(*AutoCloseable* resources){
  }

try-with-resources statement

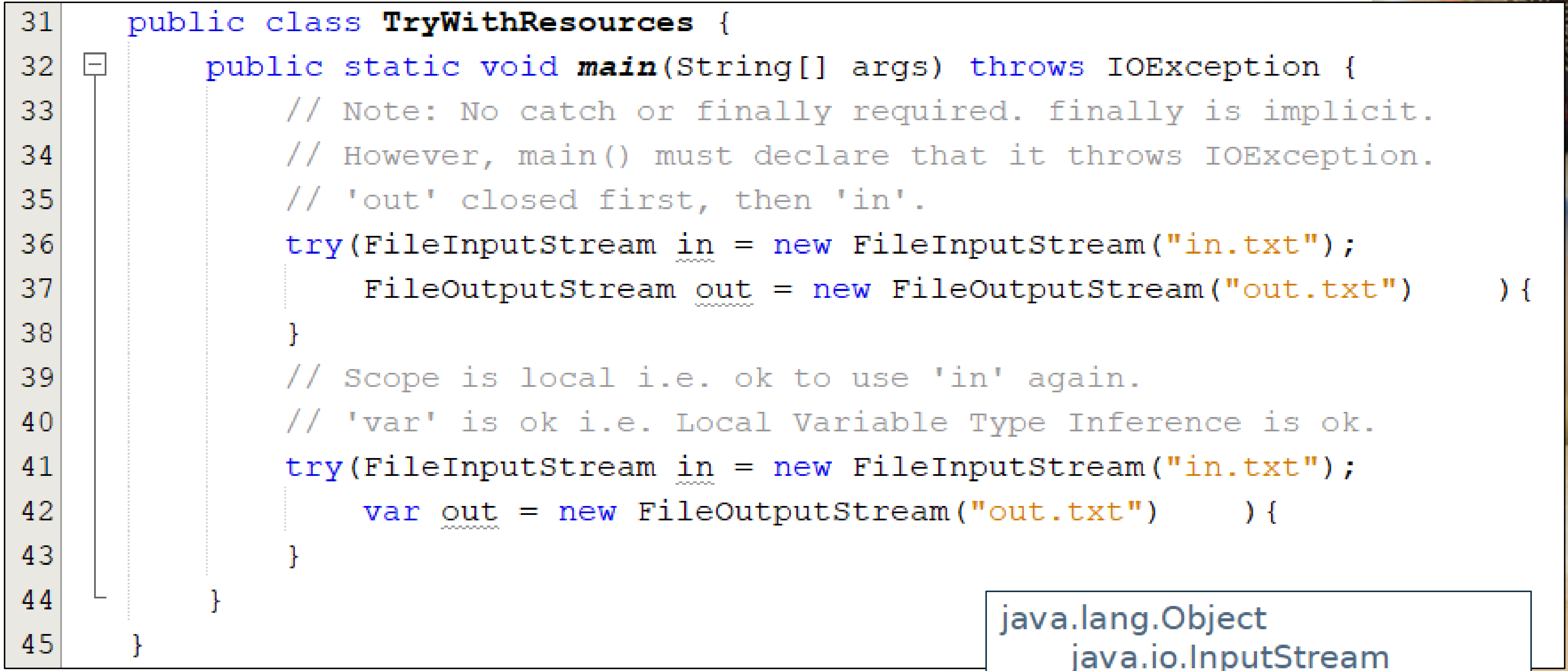
- Separate the resources in the *try* clause with semi-colon i.e. ;

- The resources created in the *try* clause are local to the *try* block i.e. they are only in scope for the *try* block. Do not try to access these resources in a *catch* or (explicit) *finally* block (if coded).

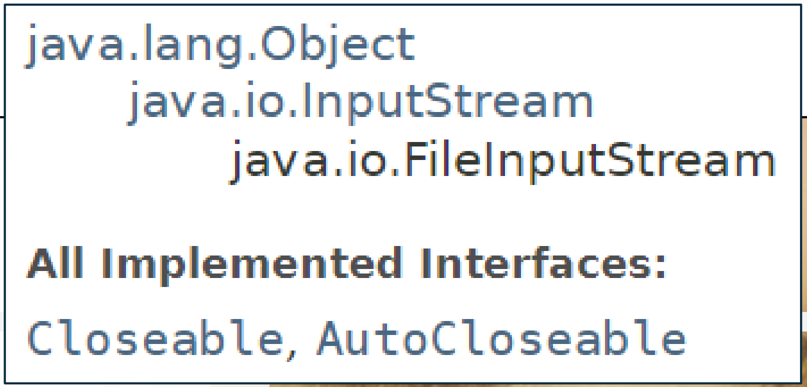
- As they are local, *var* can be used.

- Resources are closed in reverse order to the order they are created.

try-with-resources statement



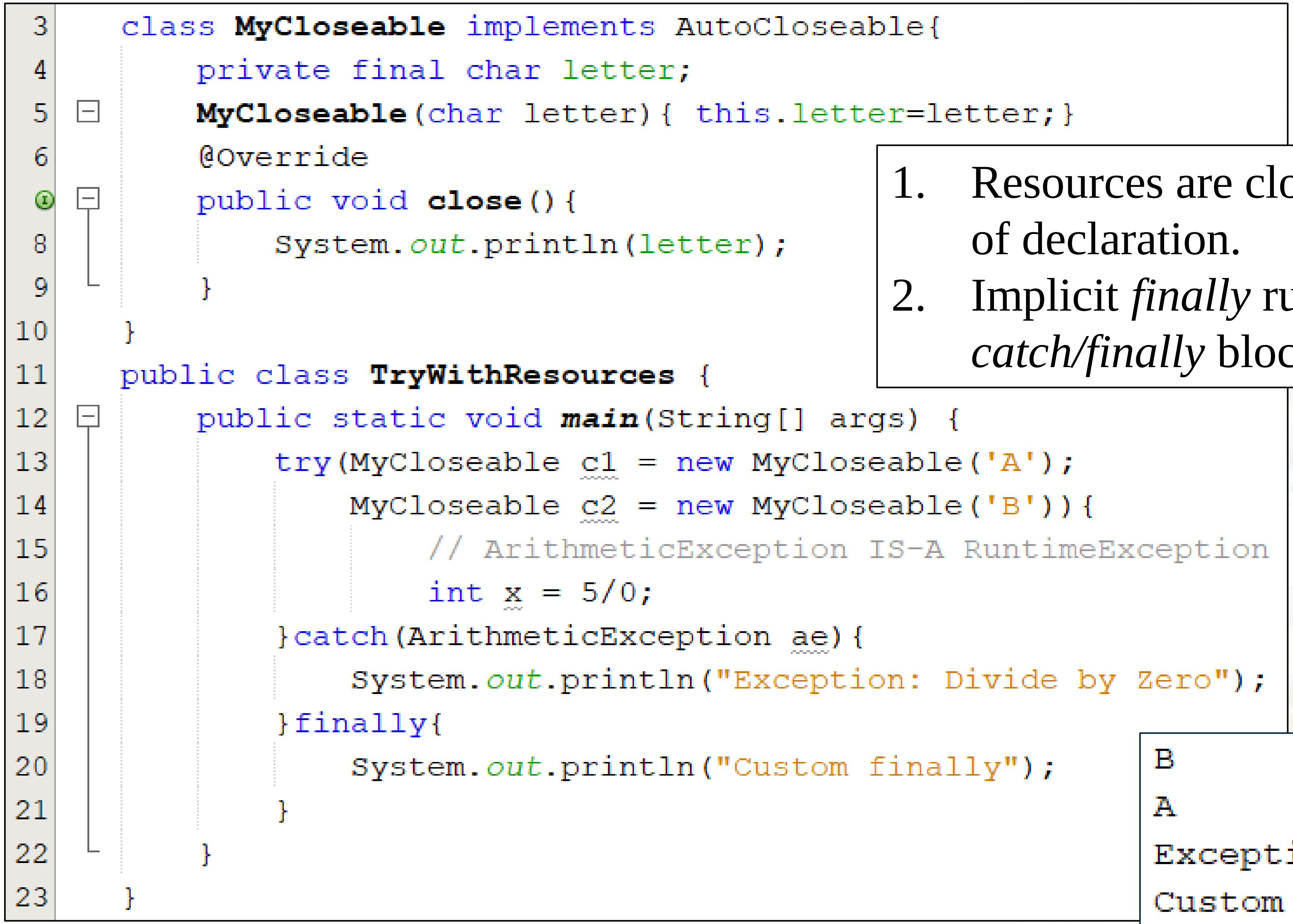
```java
public class TryWithResources {
    public static void main(String[] args) throws IOException {
        // Note: No catch or finally required. finally is implicit.
        // However, main() must declare that it throws IOException.
        // 'out' closed first, then 'in'.
        try(FileInputStream in = new FileInputStream("in.txt");
            FileOutputStream out = new FileOutputStream("out.txt")     ){
        }
        // Scope is local i.e. ok to use 'in' again.
        // 'var' is ok i.e. Local Variable Type Inference is ok.
        try(FileInputStream in = new FileInputStream("in.txt");
            var out = new FileOutputStream("out.txt")     ){
        }
    }
}
```



java.lang.Object
    java.io.InputStream
        java.io.FileInputStream

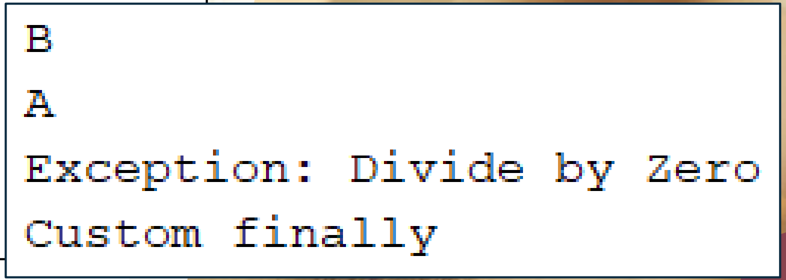
All Implemented Interfaces:

Closeable, AutoCloseable



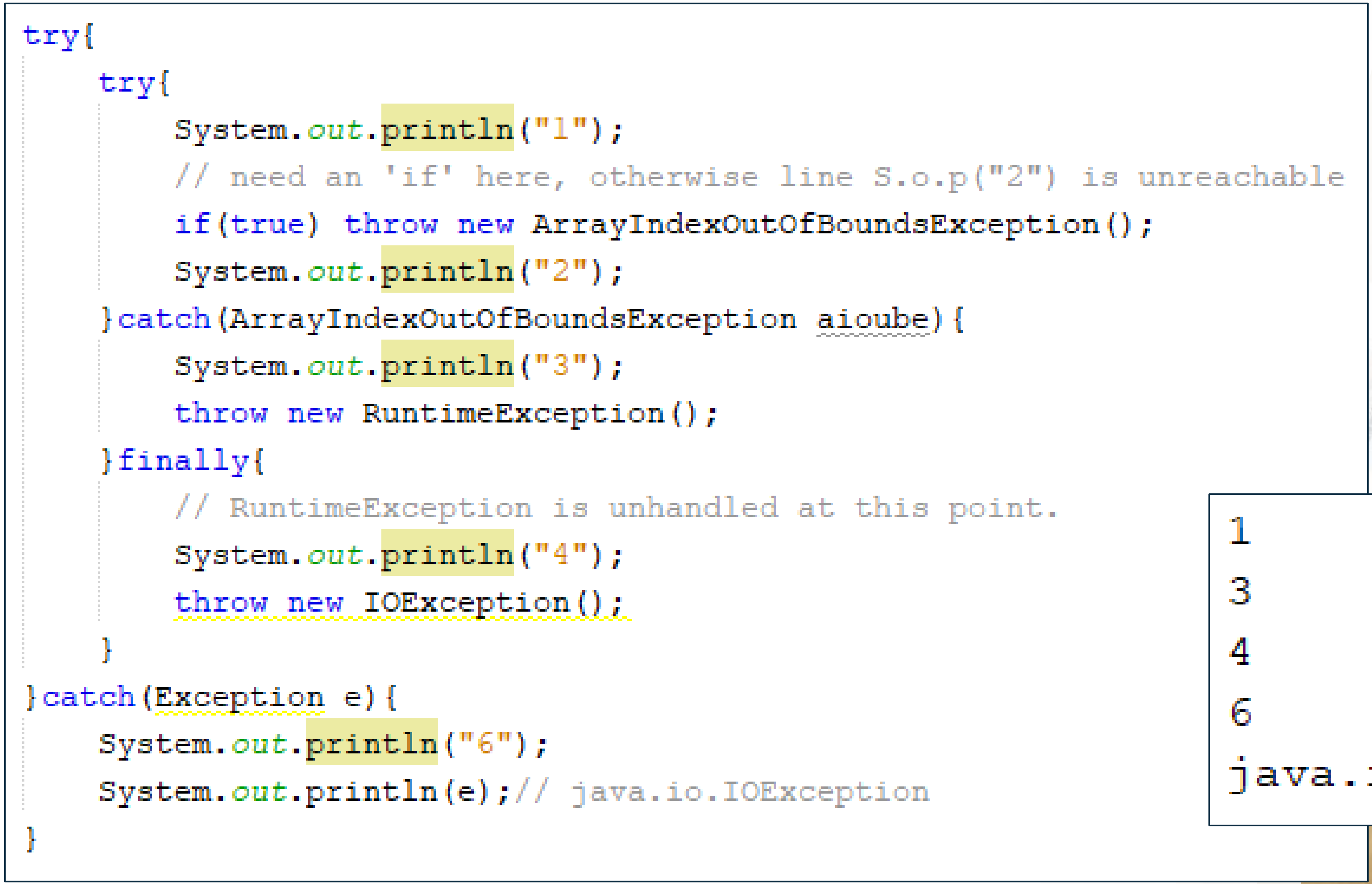
```java
class MyCloseable implements AutoCloseable{
    private final char letter;
    MyCloseable(char letter){ this.letter=letter;}
    @Override
    public void close(){
        System.out.println(letter);
    }
}
public class TryWithResources {
    public static void main(String[] args) {
        try(MyCloseable c1 = new MyCloseable('A');
            MyCloseable c2 = new MyCloseable('B')){
                // ArithmeticException IS-A RuntimeException
                int x = 5/0;
        }catch(ArithmeticException ae){
            System.out.println("Exception: Divide by Zero");
        }finally{
            System.out.println("Custom finally");
        }
    }
}
```

1. Resources are closed in the reverse order of declaration.
2. Implicit *finally* runs before any explicit *catch/finally* blocks.



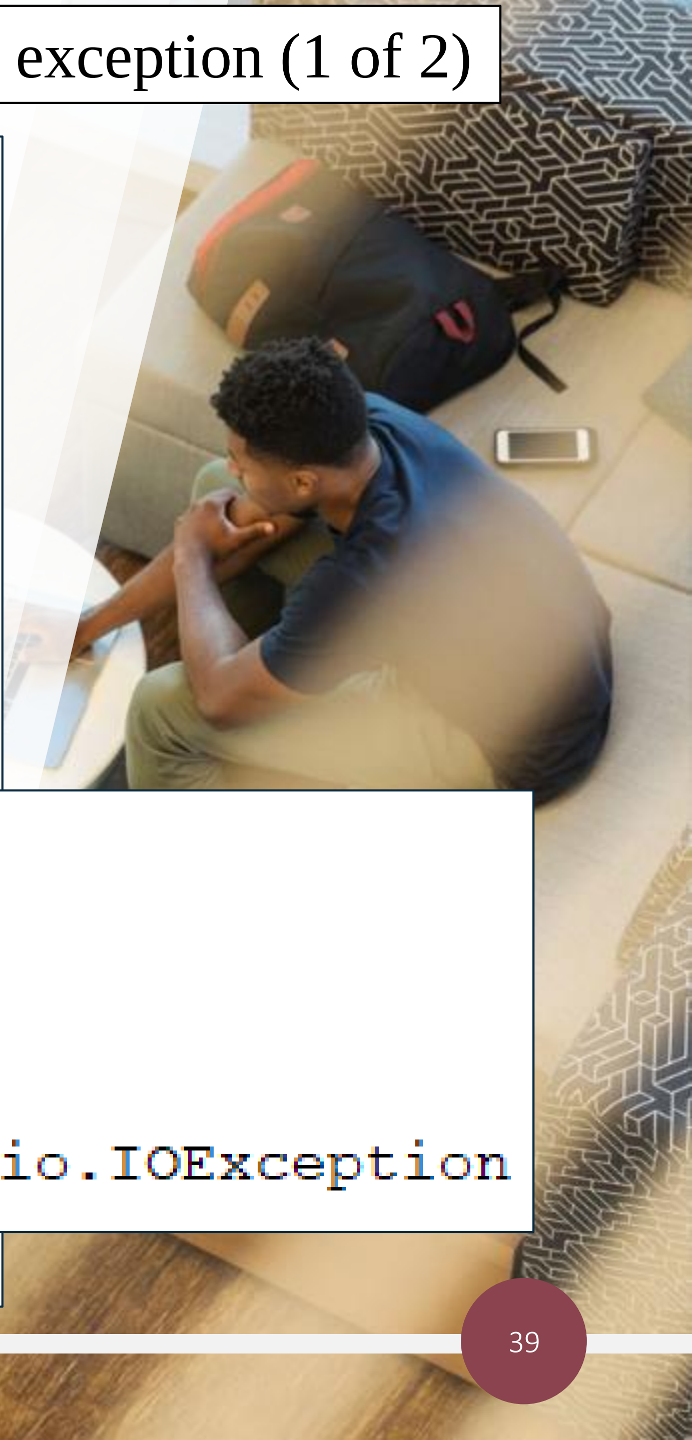
```
B
A
Exception: Divide by Zero
Custom finally
```

finally throws an exception (1 of 2)

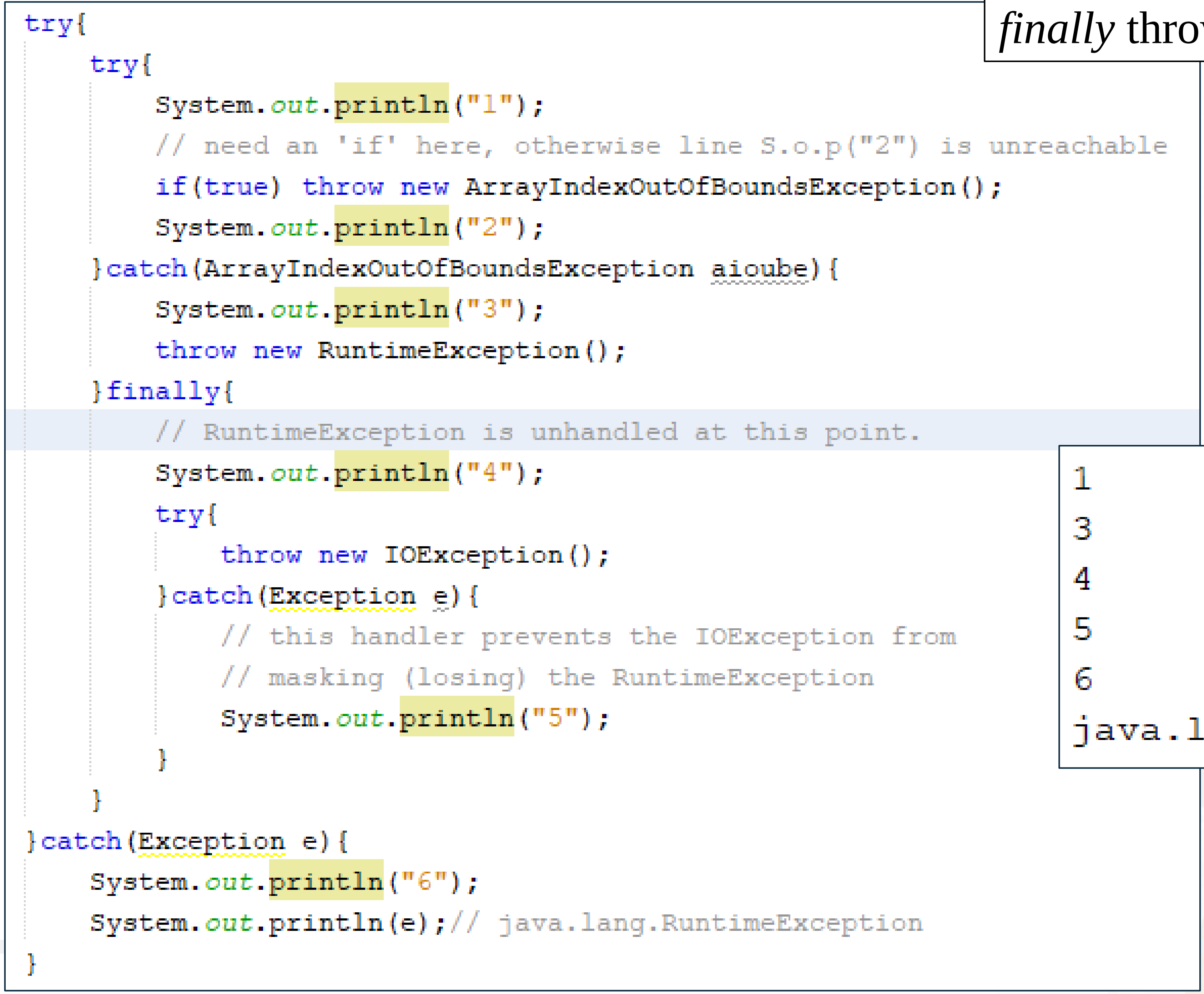


```
try{
    try{
        System.out.println("1");
        // need an 'if' here, otherwise line S.o.p("2") is unreachable
        if(true) throw new ArrayIndexOutOfBoundsException();
        System.out.println("2");
    }catch(ArrayIndexOutOfBoundsException aioube){
        System.out.println("3");
        throw new RuntimeException();
    }finally{
        // RuntimeException is unhandled at this point.
        System.out.println("4");
        throw new IOException();
    }
}catch(Exception e){
    System.out.println("6");
    System.out.println(e);// java.io.IOException
}
```

```
1
3
4
6
java.io.IOException
```

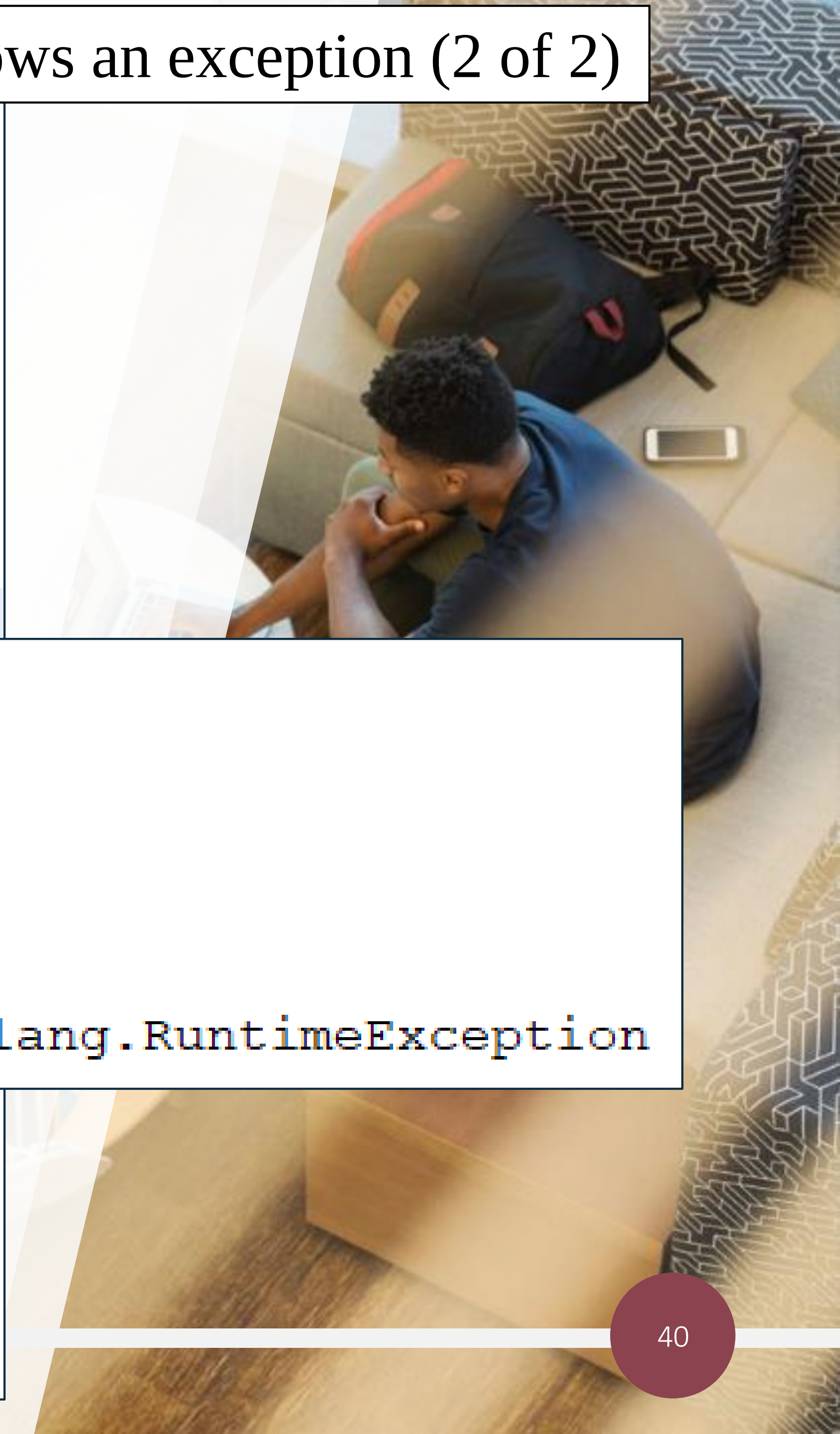


finally throws an exception (2 of 2)



```
try{
    try{
        System.out.println("1");
        // need an 'if' here, otherwise line S.o.p("2") is unreachable
        if(true) throw new ArrayIndexOutOfBoundsException();
        System.out.println("2");
    }catch(ArrayIndexOutOfBoundsException aioube){
        System.out.println("3");
        throw new RuntimeException();
    }finally{
        // RuntimeException is unhandled at this point.
        System.out.println("4");
        try{
            throw new IOException();
        }catch(Exception e){
            // this handler prevents the IOException from
            // masking (losing) the RuntimeException
            System.out.println("5");
        }
    }
}catch(Exception e){
    System.out.println("6");
    System.out.println(e);// java.lang.RuntimeException
}
```

```
1
3
4
5
6
java.lang.RuntimeException
```

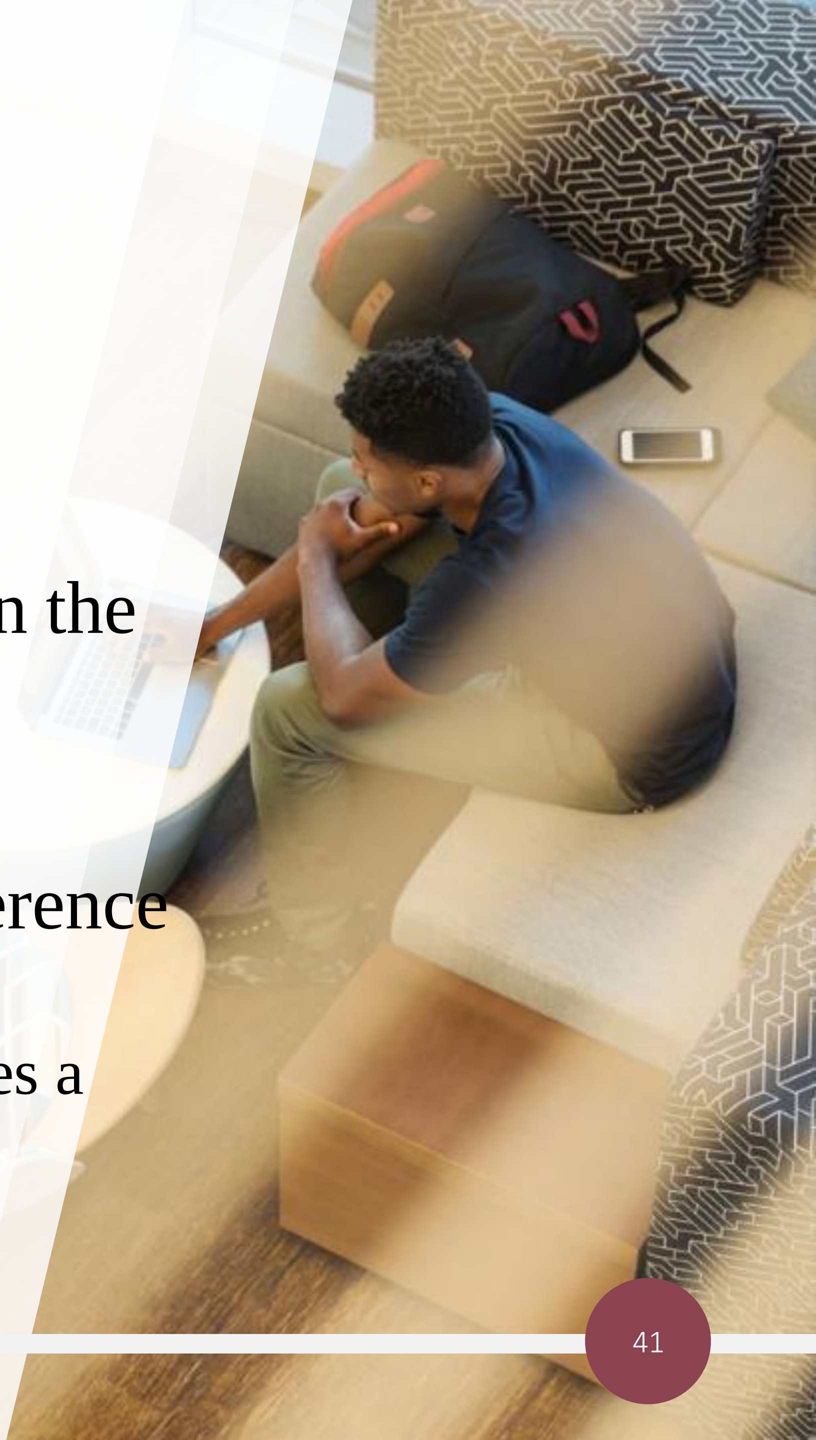


Exception signatures when overriding

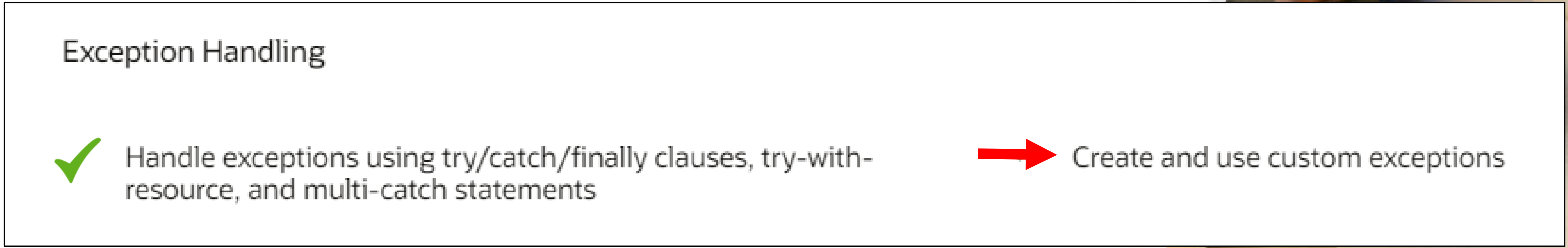
- Overriding a method can be done in two ways:
  - extending a class or
  - implementing an interface

- Either way, you cannot add extra checked exceptions in the overriding method signature.

- This is because the compiler is concerned with the reference type
  - if the reference type is e.g. extended and the subtype overrides a parent method then the compiler still only checks the parent signature. Thus, the overridden code cannot polymorphically generate exceptions that there is no code for…



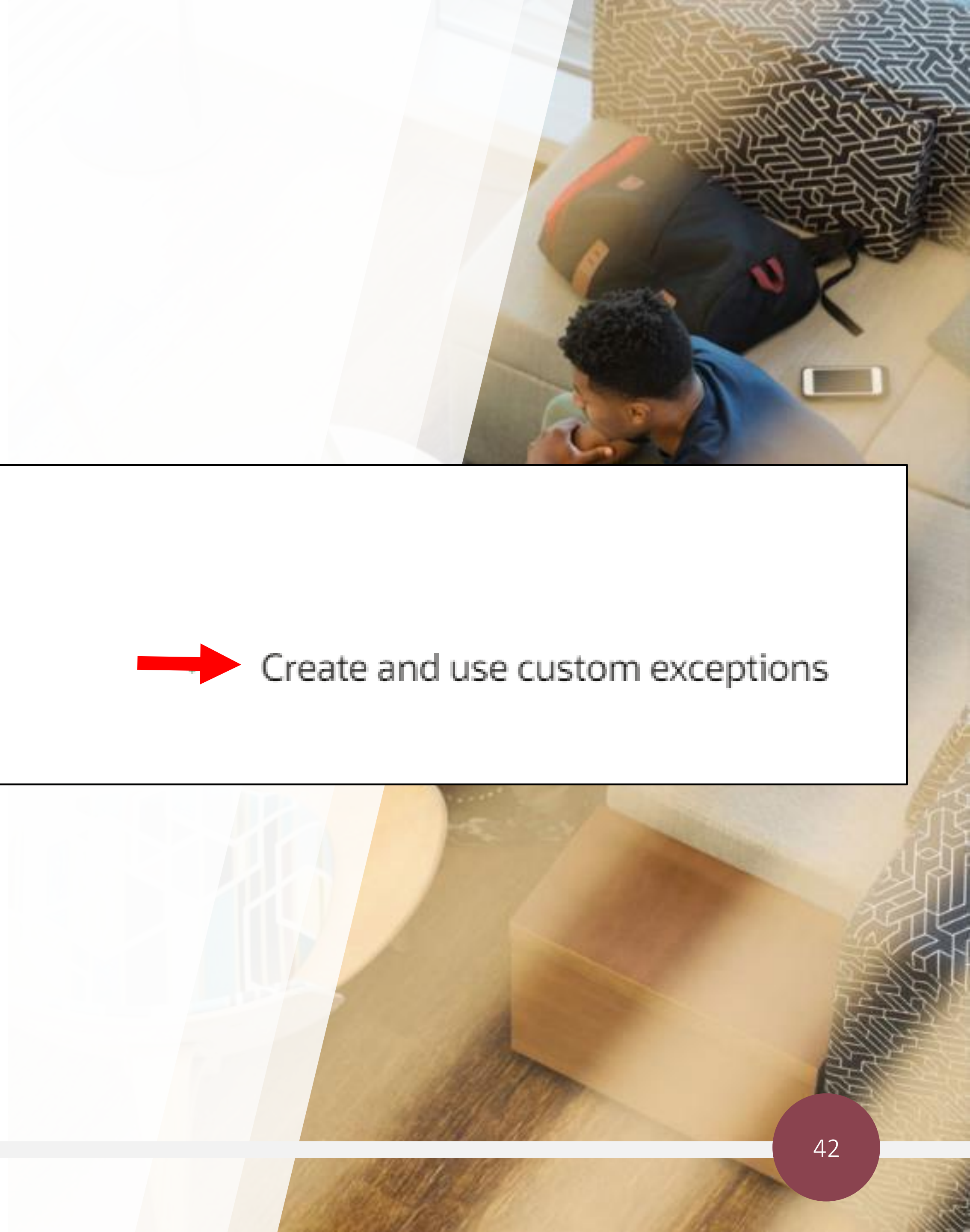
Exception Handling



Exception Handling

✓ Handle exceptions using try/catch/finally clauses, try-with-resource, and multi-catch statements

→ Create and use custom exceptions



Custom Exceptions

- Required you need an exception that is not already provided by the Java API.

- When you want to hide the exception from the caller method i.e. if returning the Java API exception type, would expose too much information about your implementation.

- We define our custom exception classes by extending the relevant API exception and providing our own constructors
  - ➢ no-arg constructor
  - ➢ constructor that takes an *Exception*
  - ➢ constructor that takes a *String*