Exceptions

Nancy Hitschfeld
Matías Toro

Programming languages such as Scala or Java use *exceptions*
to handle errors and other exceptional events

This lecture is about learning
when, *how*, *why* to use exceptions

Roadmap

1. Why an exception mechanism?
2. What is an exception?
3. The Catch or Specify Requirement
4. How to throw exception
5. Operations on an exception
6. Exception to abort recursion

Roadmap

1. **Why an exception mechanism?**
2. What is an exception?
3. The Catch or Specify Requirement
4. How to throw exception
5. Operations on an exception
6. Exception to abort recursion

Why an exception mechanism?

In the C programming language, tacking care of the potential errors clutter the code and reduce readability

Example:

```
readFile {
    open the file;
    determine its size;
    allocate that much memory;
    read the file into memory;
    close the file;
}
```

Why an exception mechanism?

In the C programming language, taking care of the potential errors clutter the code

What happens if the file can't be opened?

Example:

```
readFile {
    open the file;
    determine its size;
    allocate that much memory;
    read the file into memory;
    close the file;
}
```

Why an exception mechanism?

In the C programming language, taking care of the potential errors clutter the code

Example:

```
readFile {
    open the file;
    determine its size;
    allocate that much memory;
    read the file into memory;
    close the file;
}
```

What happens if the length of the file can't be determined?

Why an exception mechanism?

In the C programming language, taking care of the potential errors clutter the code

Example:

What happens if enough memory can't be allocated?

```
readFile {
    open the file;
    determine its size;
    allocate that much memory;
    read the file into memory;
    close the file;
}
```

Why an exception mechanism?

In the C programming language, taking care of the potential errors clutter the code

Example:

```
readFile {
    open the file;
    determine its size;
    allocate that much memory;
    read the file into memory;
    close the file;
}
```

What happens if the read fails?

Why an exception mechanism?

In the C programming language, taking care of the potential errors clutter the code

What happens if the file can't be closed?

Example:

```
readFile {
    open the file;
    determine its size;
    allocate that much memory;
    read the file into memory;
    close the file;
}
```

```
errorCodeType readFile {
    initialize errorCode = 0;

    open the file;
    if (theFileIsOpen) {
        determine the length of the file;
        if (gotTheFileLength) {
            allocate that much memory;
            if (gotEnoughMemory) {
                read the file into memory;
                if (readFailed) {
                    errorCode = -1;
                }
            } else {
                errorCode = -2;
            }
        } else {
            errorCode = -3;
        }
        close the file;
        if (theFileDidntClose && errorCode == 0) {
            errorCode = -4;
        } else {
            errorCode = errorCode and -4;
        }
    } else {
        errorCode = -5;
    }
    return errorCode;
}
```

Without exception

With exception

```
readFile {
    try {
        open the file
        determine its size
        allocate that much memory
        read the file into memory
        close the file
    } catch{
       case fileOpenFailed => doSomething
       case sizeDeterminationFailed => doSomething
       case memoryAllocationFailed => doSomething
       case readFailed => doSomething
       case fileCloseFailed => doSomething
     }
}
```

Roadmap

1. Why an exception mechanism?
2. **What is an exception?**
3. The Catch or Specify Requirement
4. How to throw exception
5. Operations on an exception
6. Exception to abort recursion

What is an exception?

When an error occurs in a method, the method creates an object, and hands it to the runtime system

An *exception* is an event that occurs during the execution of a program that disrupts the normal flow of instructions

Creating an exception object and handling it to the system is called *throwing an exception*

Creating an empty list

```
object Nil extends List[Nothing] {
  override def head: Nothing = throw
     new NoSuchElementException("head of empty list")
  override def tail: Nothing = throw
     new UnsupportedOperationException("tail of empty list")
  override def last: Nothing = throw
     new NoSuchElementException("last of empty list")
  override def init: Nothing = throw
     new UnsupportedOperationException("init of empty list")
   ...
}
```

Creating an empty list

```
object Nil extends List[Nothing] {
  override def head: Nothing = throw
     new NoSuchElementException("head of empty list")
  override def tail: Nothing = throw
     new UnsupportedOperationException("tail of empty list")
  override def last: Nothing = throw
     new NoSuchElementException("last of empty list")
  override def init: Nothing = throw
     new UnsupportedOperationException("init of empty list")
   ...
}
```

List(1,2).head => OK

Nil.head => throws a NoSuchElementException

Defining an exception class

```
class NoSuchElementException extends RuntimeException {
   …
}
```

Looking for an handler

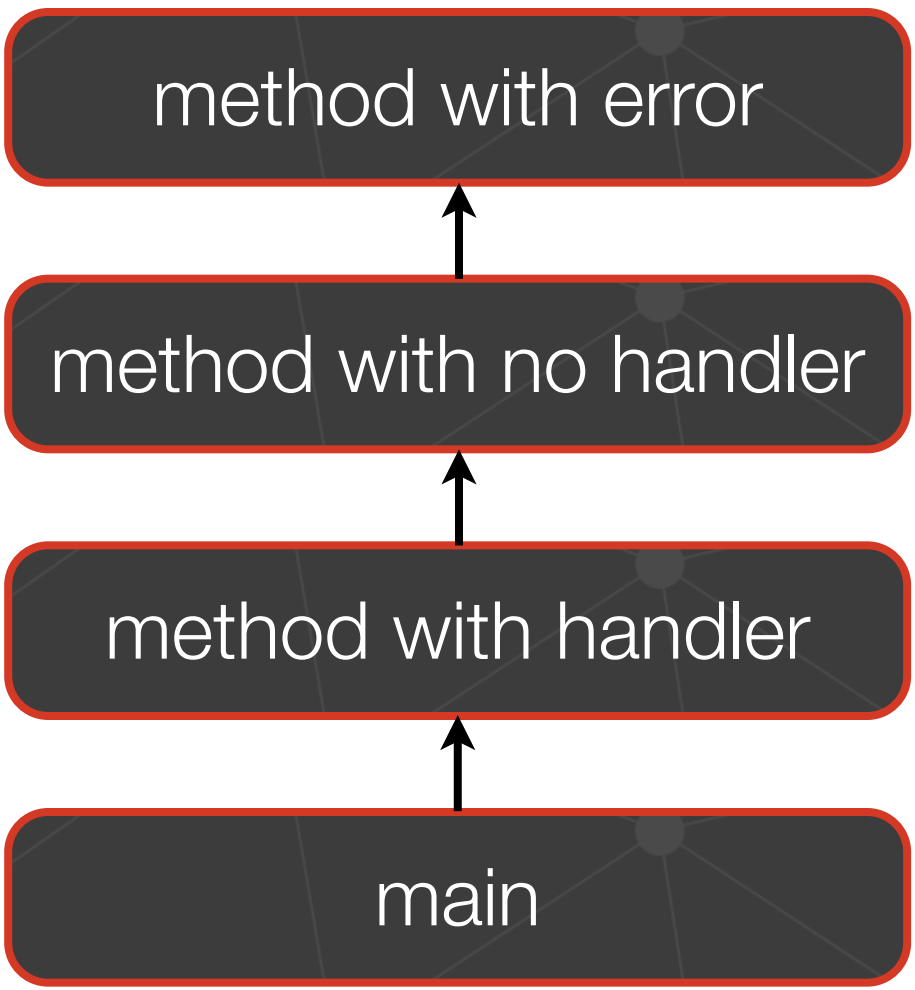
After a method throws an exception, the runtime system *attempts to find something to handle it*

The set of possible "somethings" to handle the exception is the ordered list of methods that had been called to get to the method where the error occurred

The list of methods is known as *the call stack*

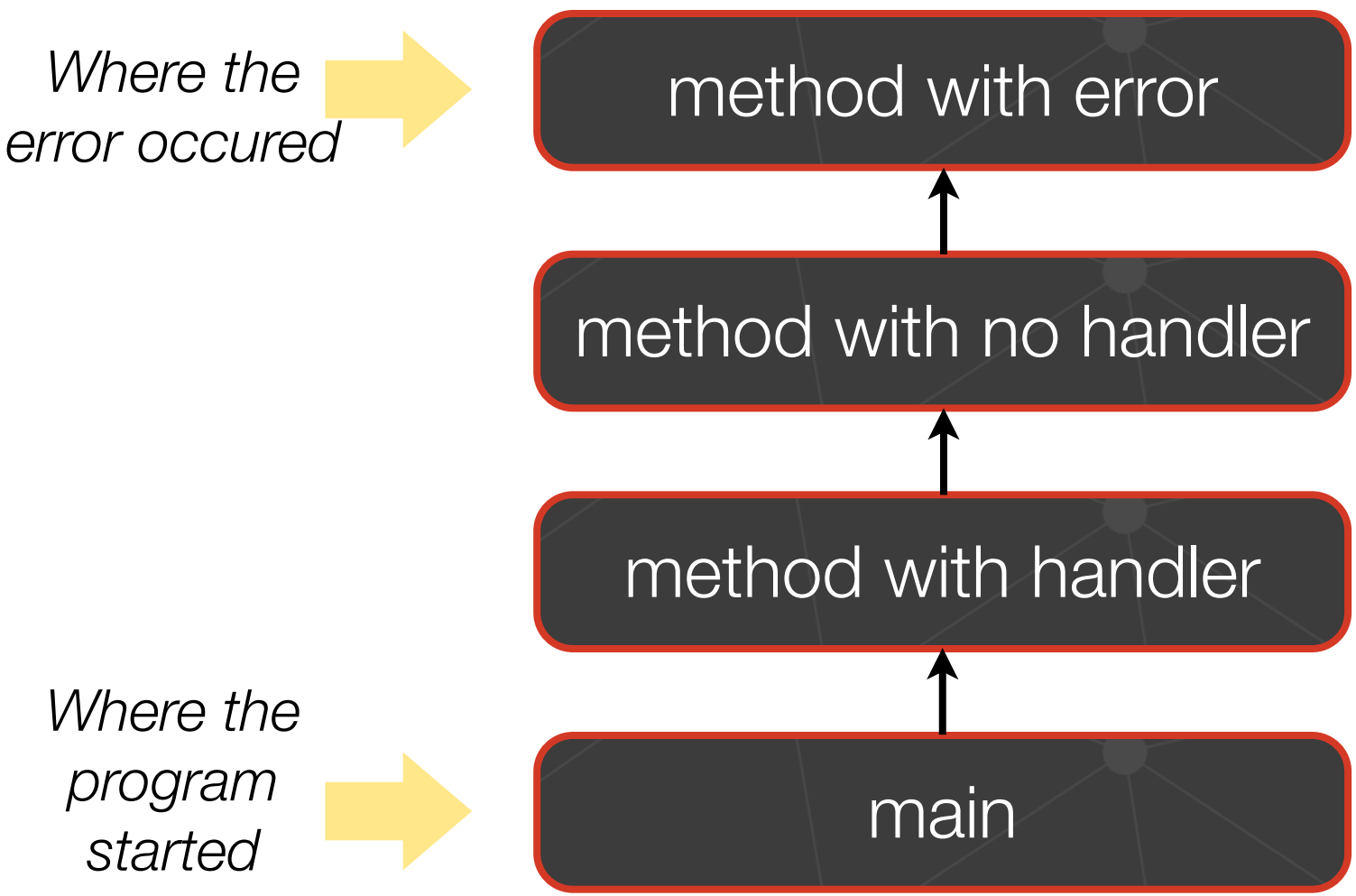
Searching the call stack

The runtime system *searches the call stack* for a *method* that contains a block of code *that can handle* the exception



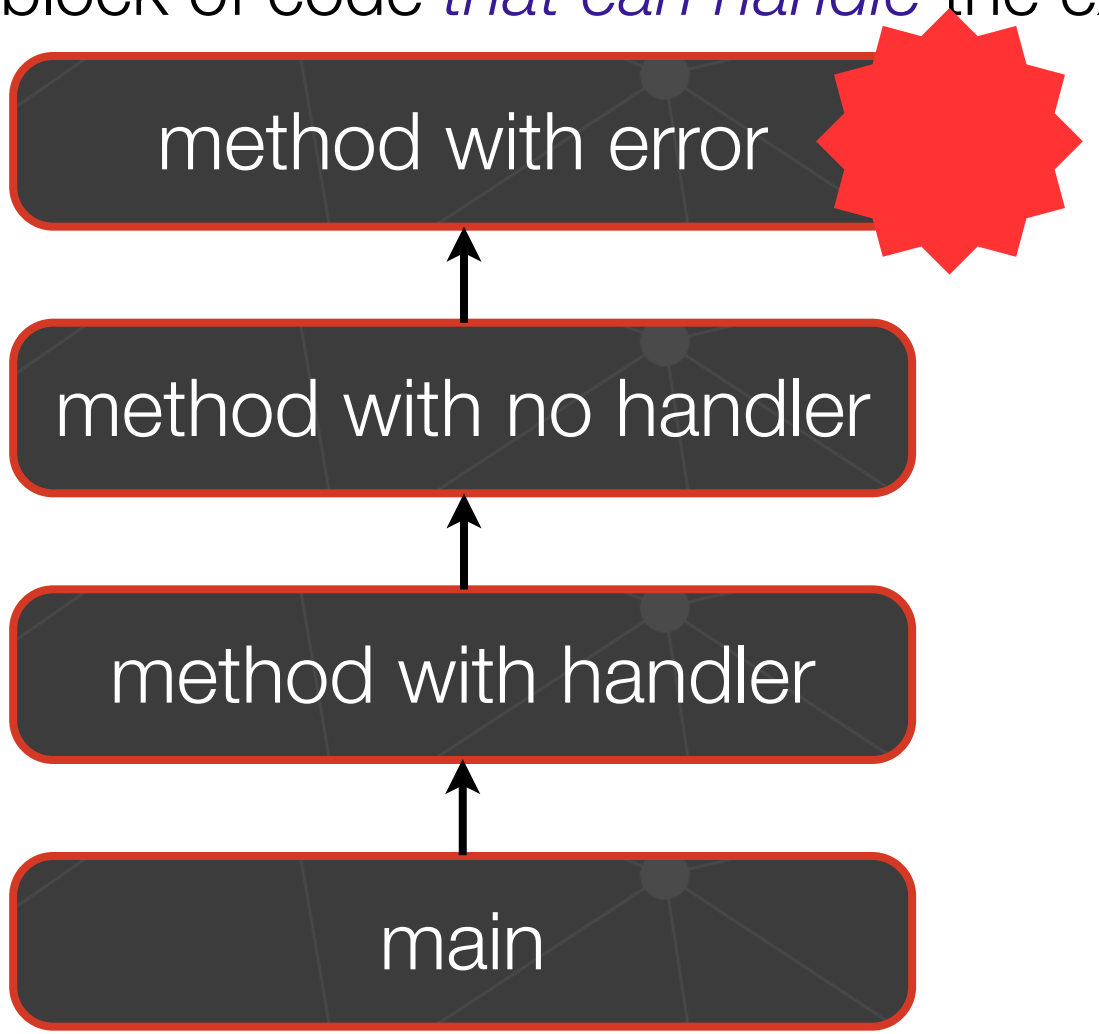
Searching the call stack

The runtime system *searches the call stack* for a *method* that contains a block of code *that can handle* the exception



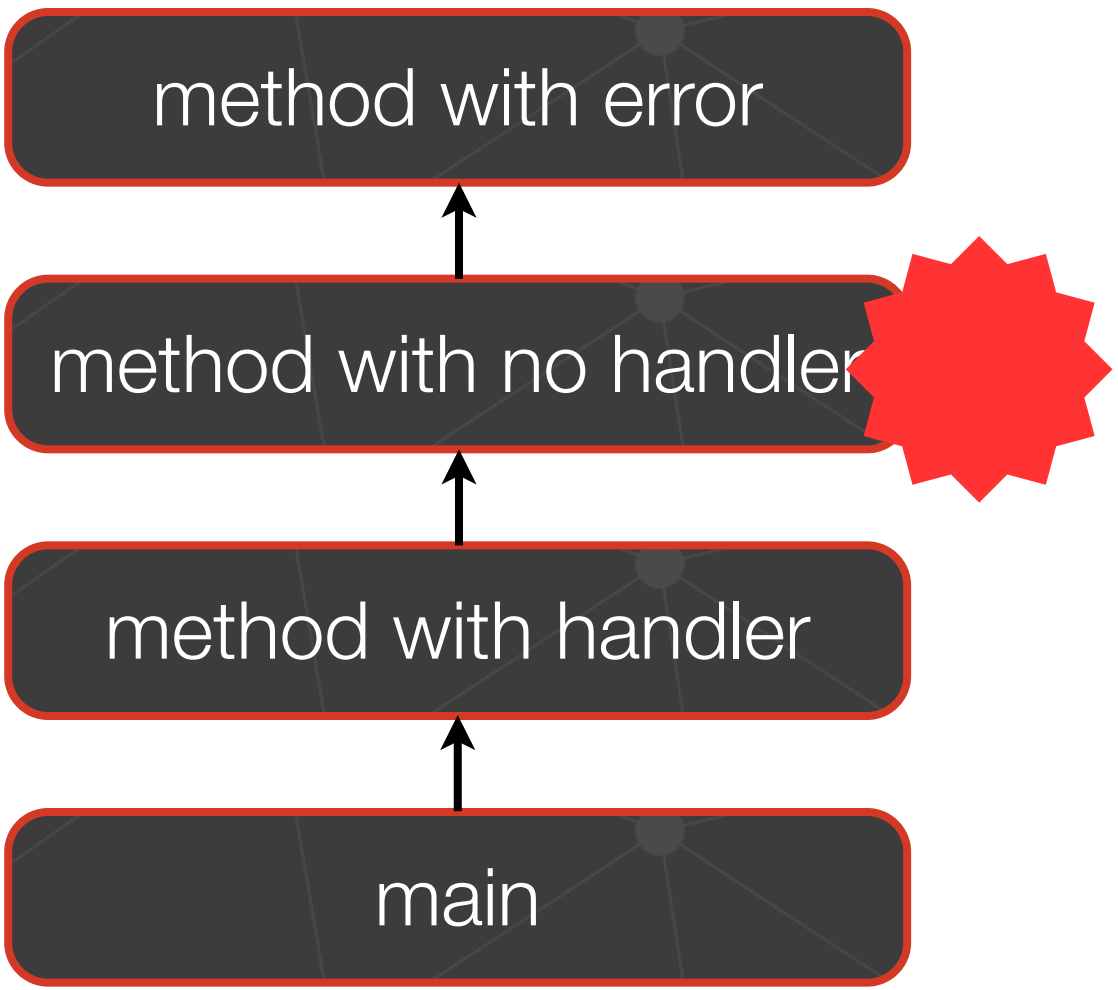
Searching the call stack

The runtime system *searches the call stack* for a *method* that contains a block of code *that can handle* the exception



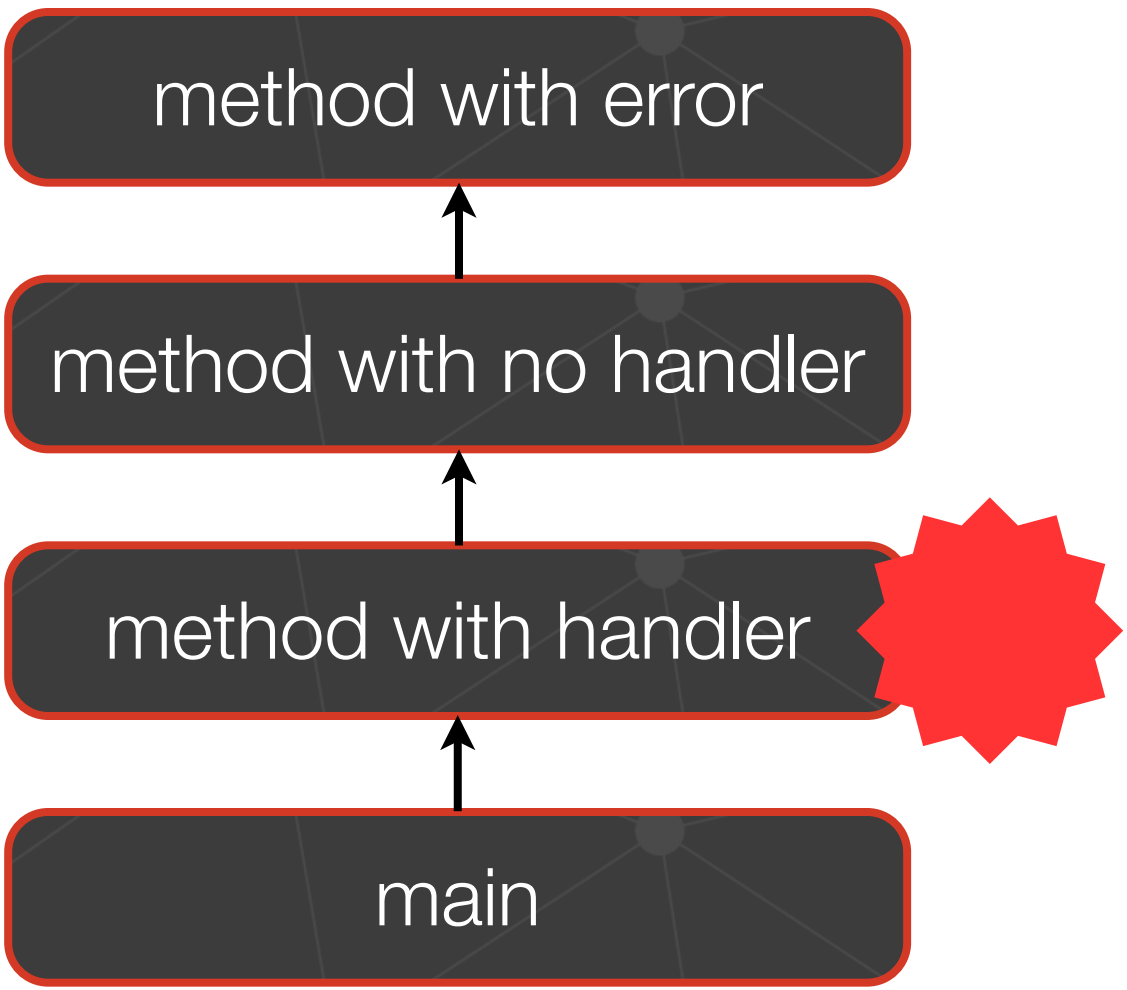
Searching the call stack

The runtime system *searches the call stack* for a *method* that contains a block of code *that can handle* the exception



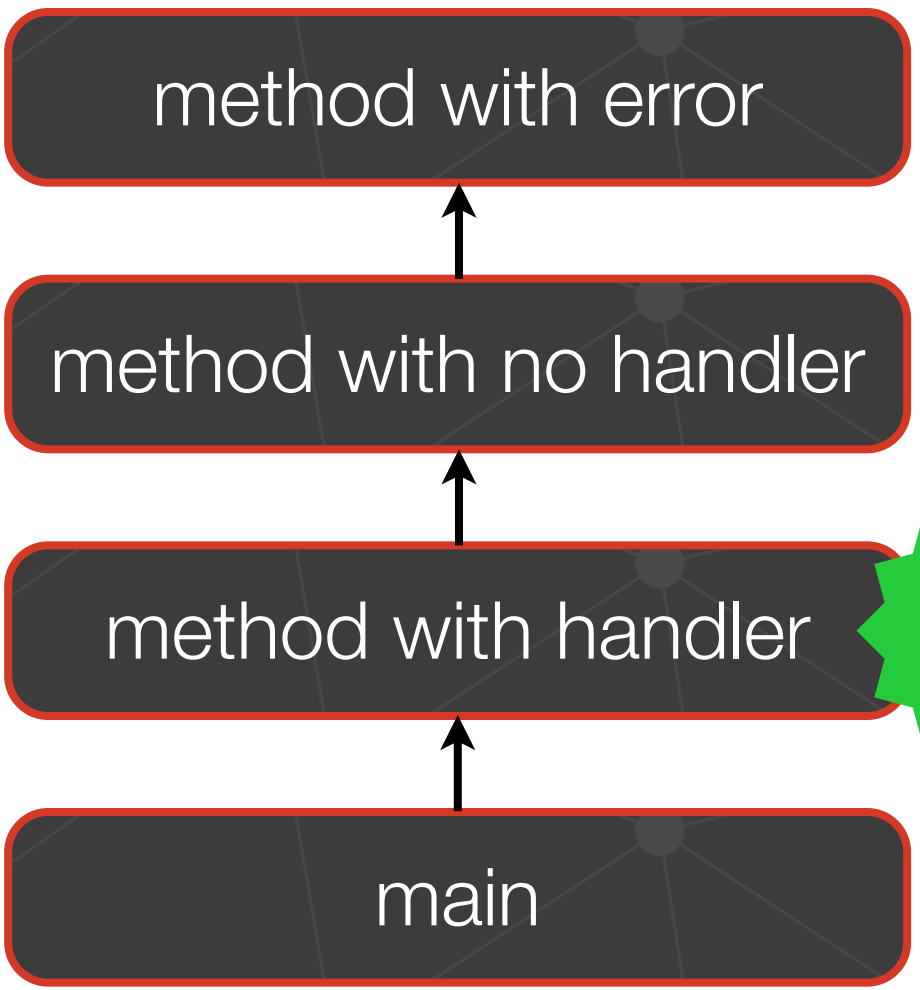
Searching the call stack

The runtime system *searches the call stack* for a *method* that contains a block of code *that can handle* the exception



Searching the call stack

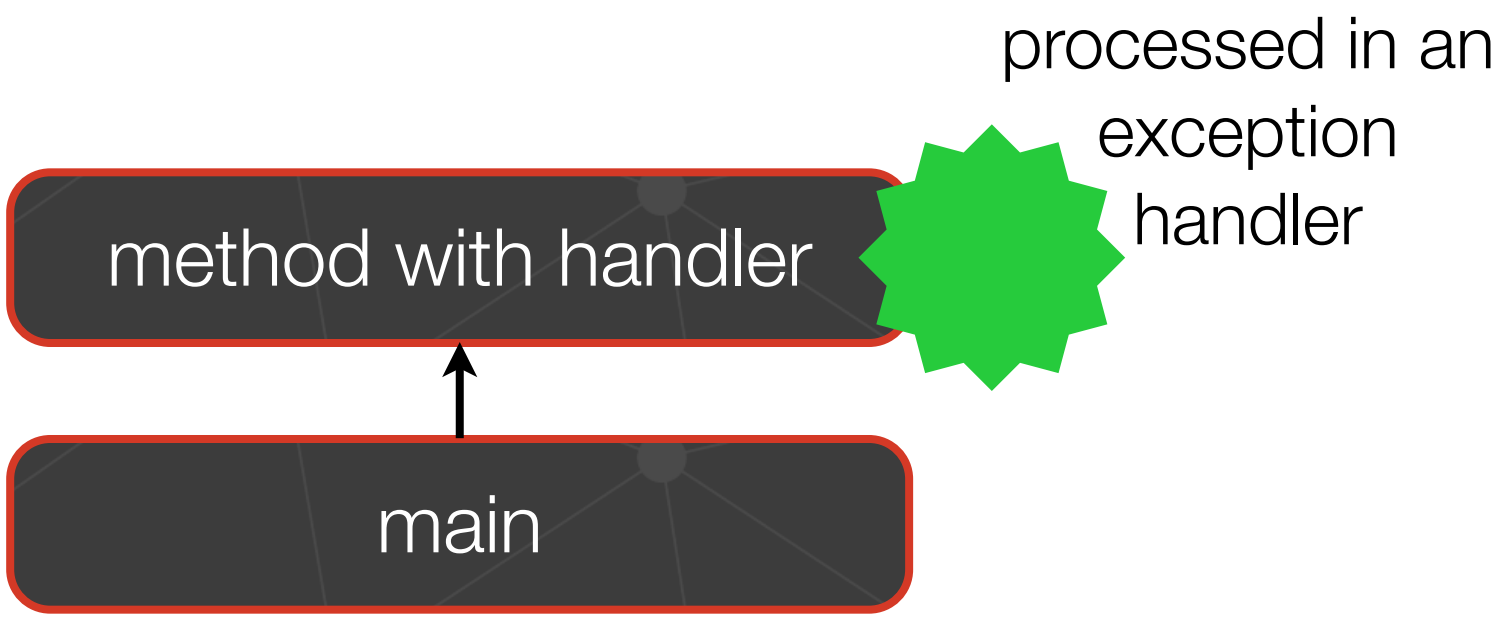
The runtime system *searches the call stack* for a *method* that contains a block of code *that can handle* the exception



processed in an exception handler

Searching the call stack

The application execution continue in the frame that contains the handler. The frames above the handler are discarded.



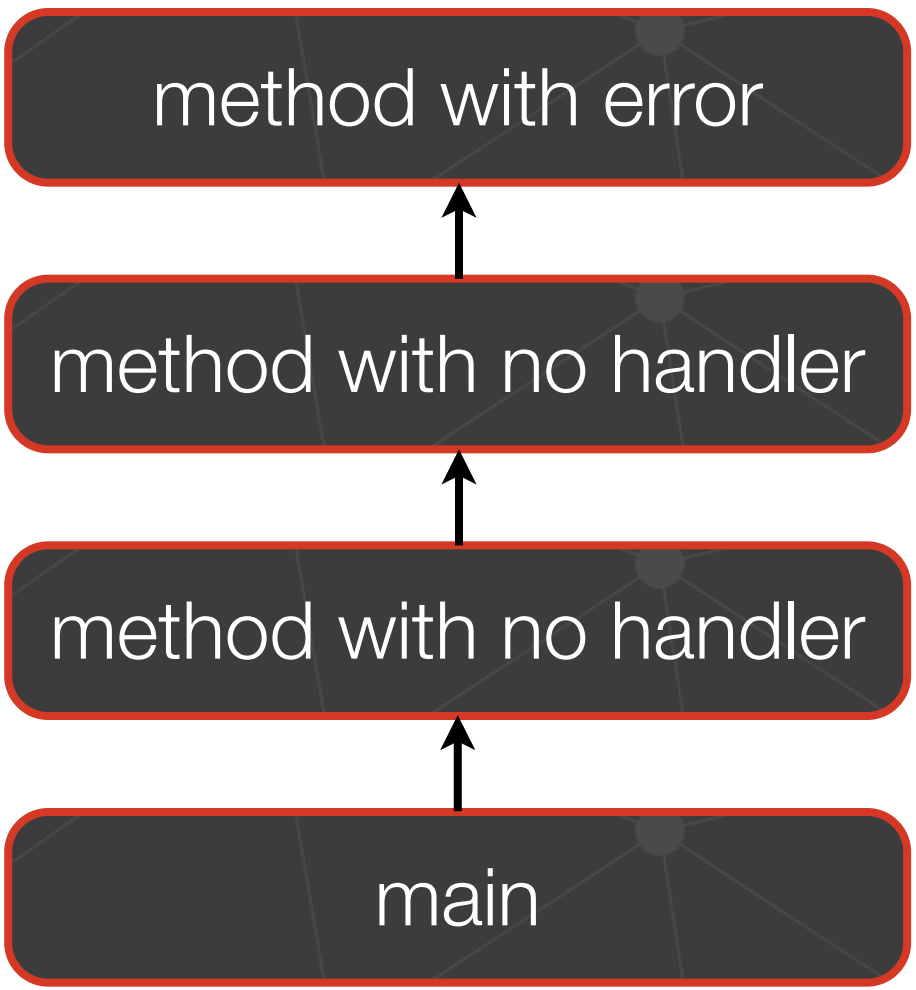
Searching the call stack

The block of code handling an exception is called an *exception handler*

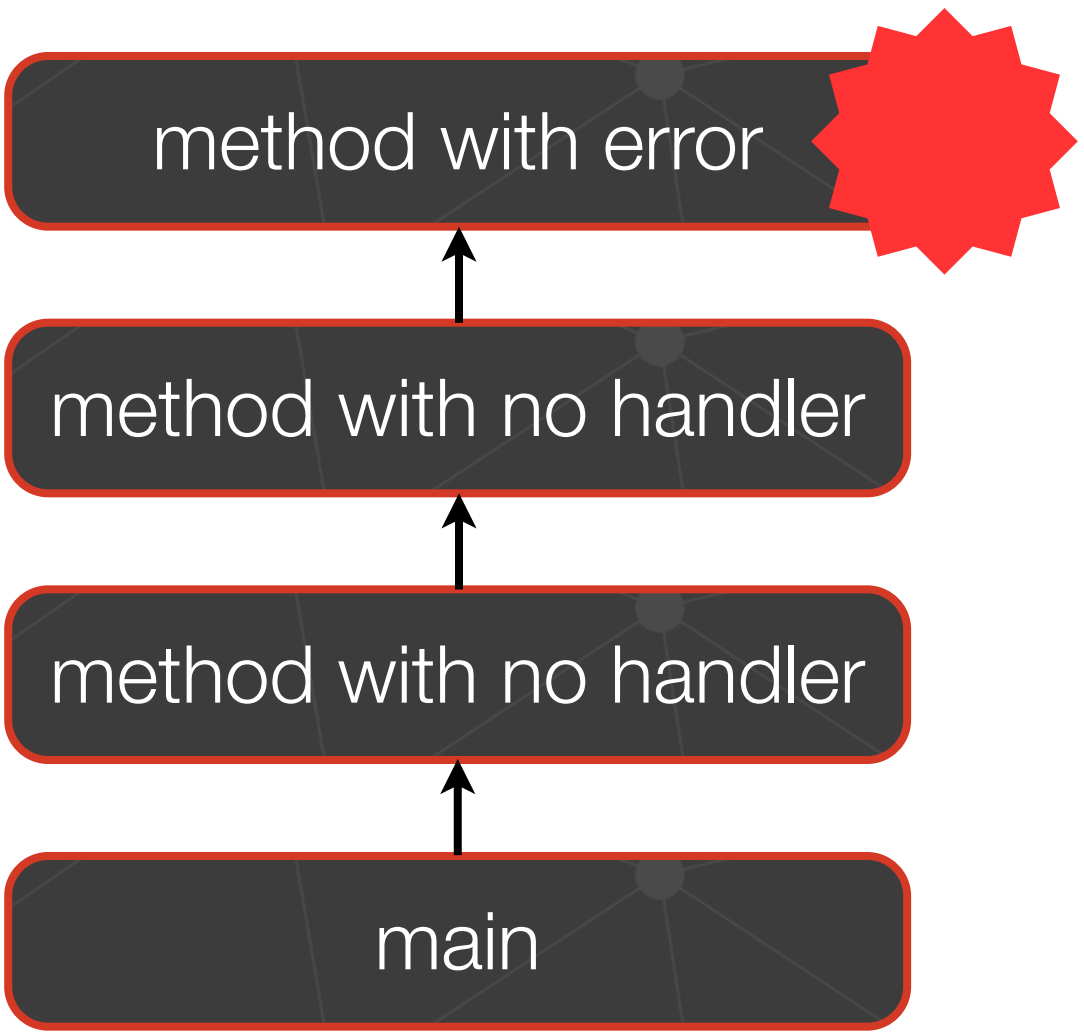
The search begins with the method in which the error occurred and proceeds through the call stack in the reverse order in which the methods were called

The exception handler chosen is said to *catch the exception*

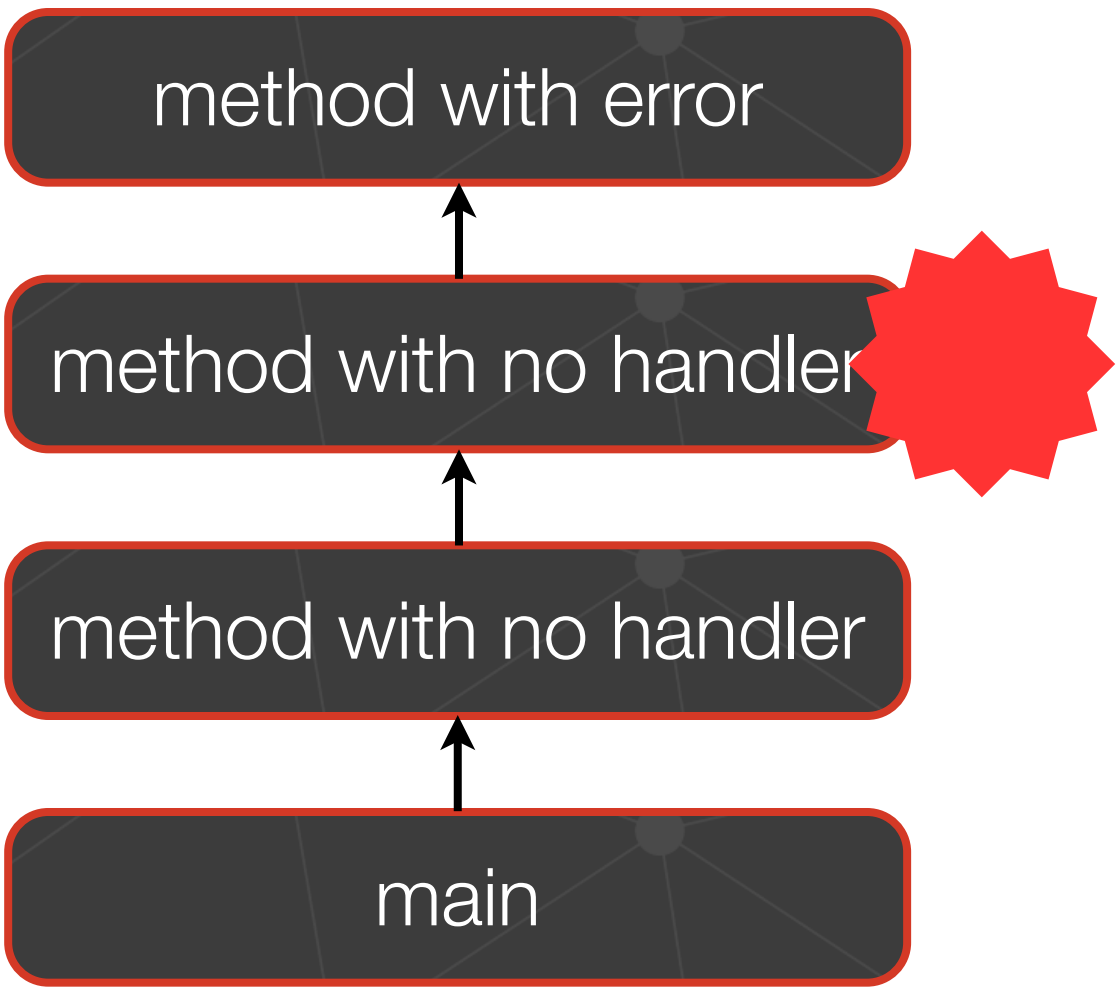
And if there is no handler?



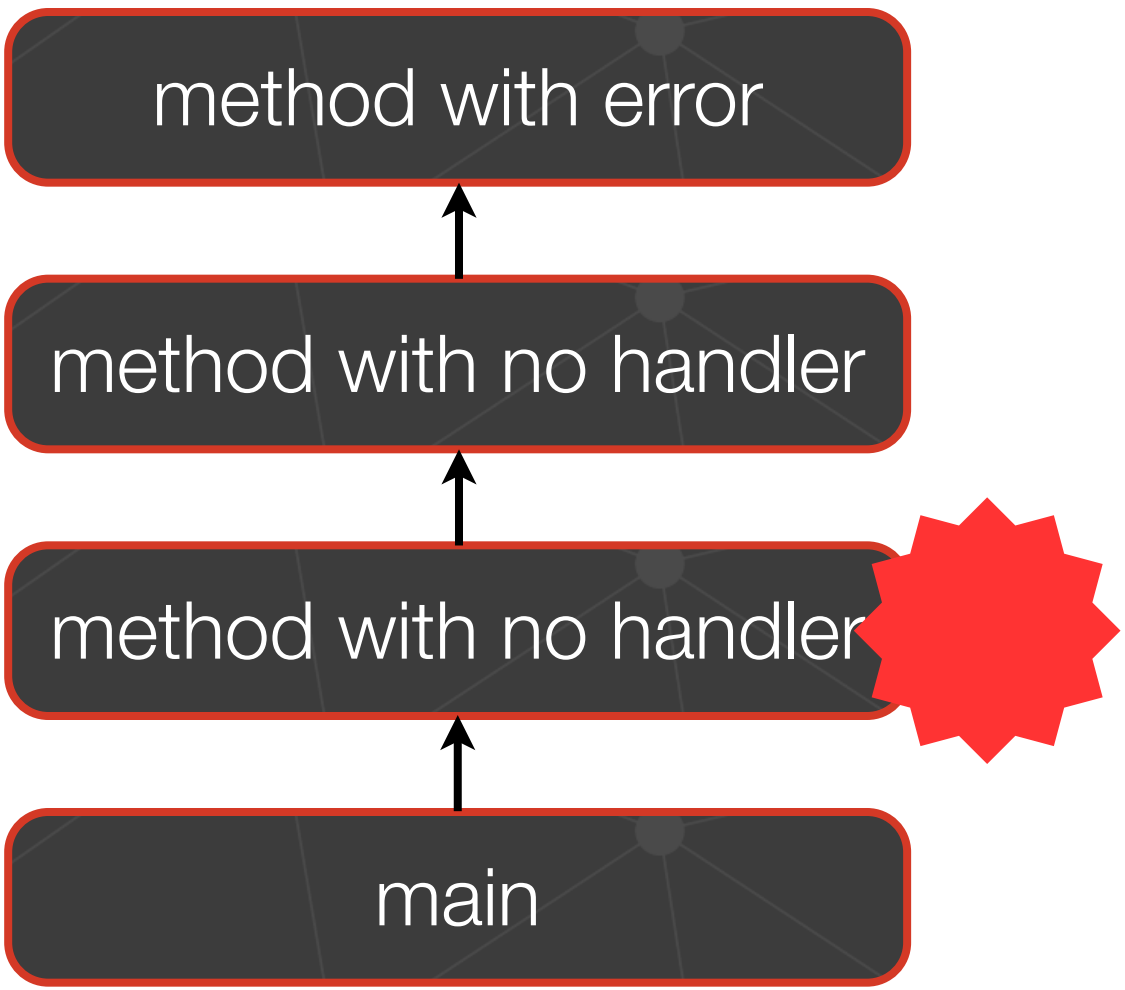
And if there is no handler?



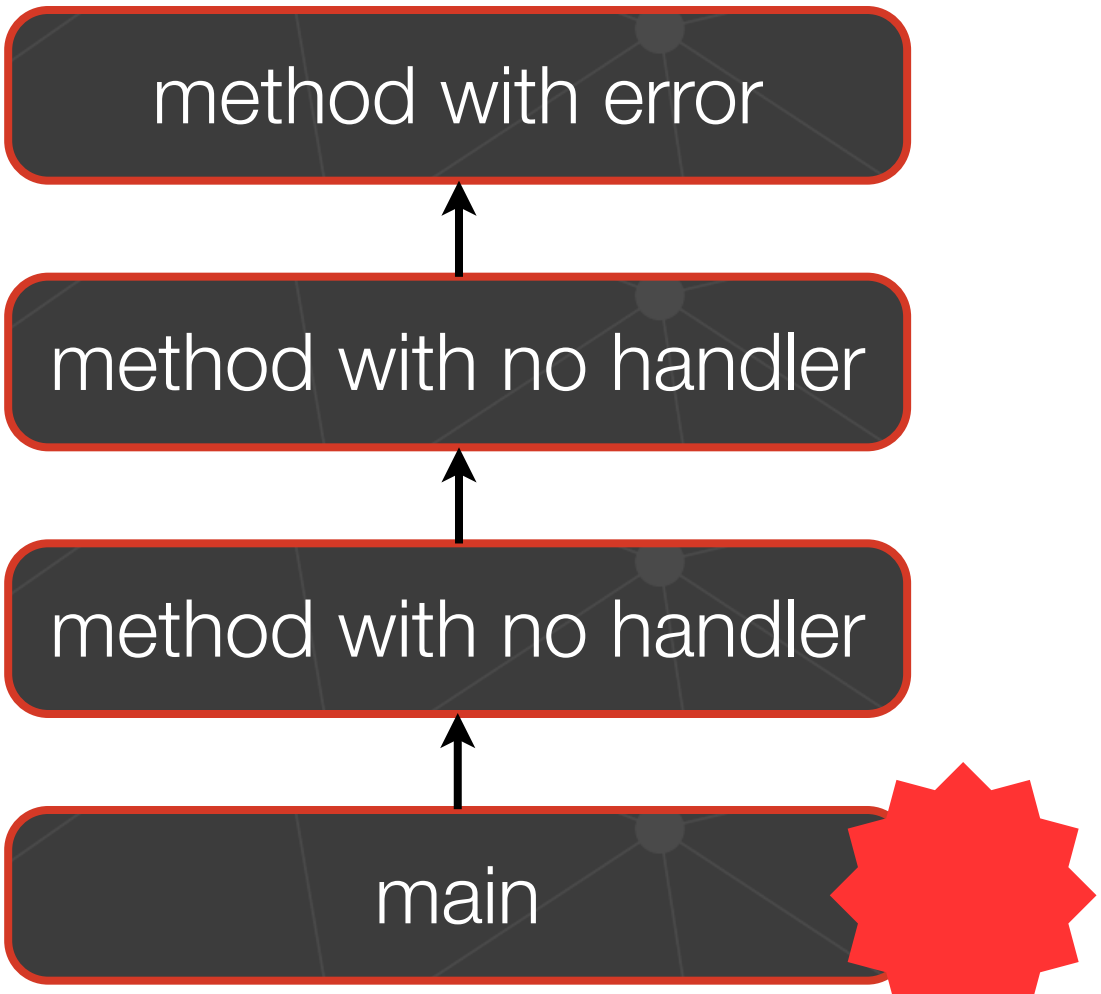
And if there is no handler?



And if there is no handler?

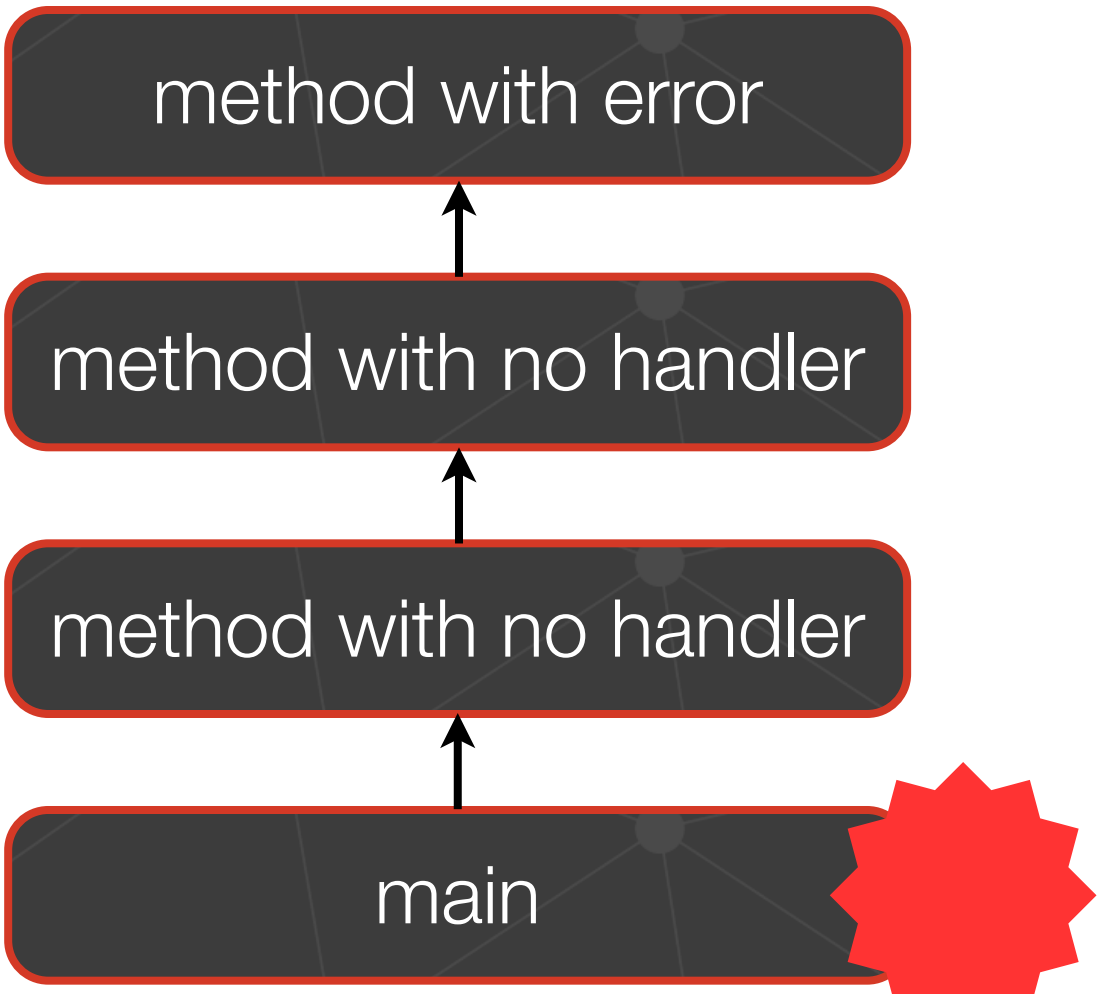


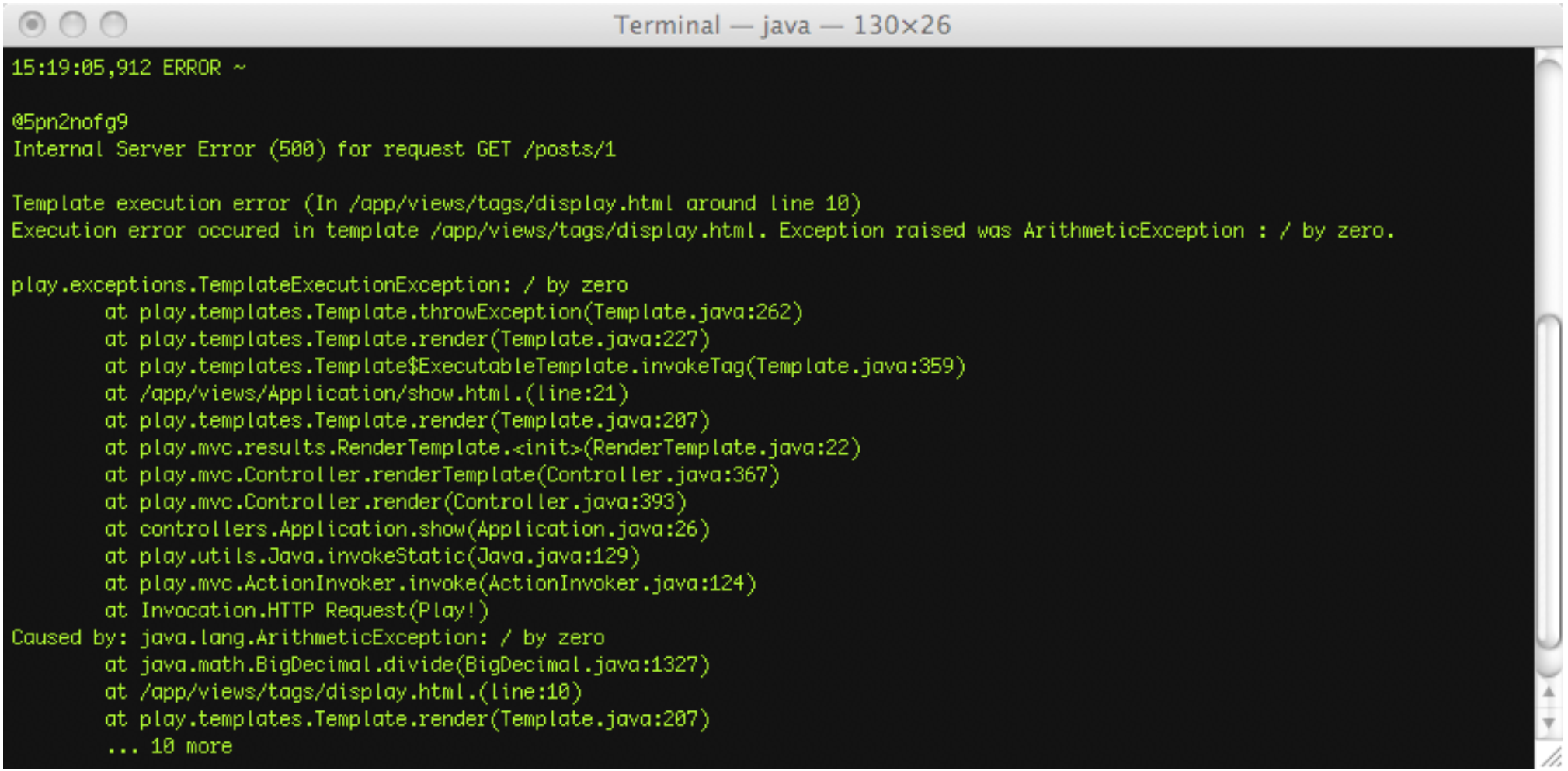
And if there is no handler?



And if there is no handler?

The program prints the stack and terminates




Terminal — java — 130×26
15:19:05,912 ERROR ~
@5pn2nofg9
Internal Server Error (500) for request GET /posts/1
Template execution error (In /app/views/tags/display.html around line 10)
Execution error occured in template /app/views/tags/display.html. Exception raised was ArithmeticException : / by zero.
play.exceptions.TemplateExecutionException: / by zero
at play.templates.Template.throwException(Template.java:262)
at play.templates.Template.render(Template.java:227)
at play.templates.Template$ExecutableTemplate.invokeTag(Template.java:359)
at /app/views/Application/show.html.(line:21)
at play.templates.Template.render(Template.java:207)
at play.mvc.results.RenderTemplate.<init>(RenderTemplate.java:22)
at play.mvc.Controller.renderTemplate(Controller.java:367)
at play.mvc.Controller.render(Controller.java:393)
at controllers.Application.show(Application.java:26)
at play.utils.Java.invokeStatic(Java.java:129)
at play.mvc.ActionInvoker.invoke(ActionInvoker.java:124)
at Invocation.HTTP Request(Play!)
Caused by: java.lang.ArithmeticException: / by zero
at java.math.BigDecimal.divide(BigDecimal.java:1327)
at /app/views/tags/display.html.(line:10)
at play.templates.Template.render(Template.java:207)
... 10 more

And if there is no handler?

If the runtime system *exhaustively* searches all the methods on the call stack *without finding* an *appropriate exception handler* the runtime system (and, consequently, the program) *terminates*.

Roadmap

1. Why an exception mechanism?
2. What is an exception?
3. **The Catch or Specify Requirement**
4. How to throw exception
5. Operations on an exception
6. Exception to abort recursion

The Catch or Specify Requirement

Valid **Java** programming language code must honor *the Catch or Specify Requirement*

This means that code that might throw certain exceptions must be enclosed:

- *a try statement that catches the exception*. The try must provide a handler for the exception
- *a method that specifies that it can throw the exception*. The method must provide a throws clause that lists the exception

Code that fails to honor the Catch or Specify Requirement will not compile

Marked as “throws”

```
package java.io;
public abstract class OutputStream implements Closeable, Flushable {
   ...

    public void write(byte b[]) throws IOException {
        write(b, 0, b.length);
    }

  public void write(byte[] b, int off, int len) throws IOException {
     ... throw new IOException() ...
  }
}
```

The Three Kinds of Exceptions

Not all exceptions are subject to the Catch or Specify Requirement

1 - Checked exception

exceptional condition that a well-written *application* should *anticipate* and *recover from*

subject to the catch or specify requirement

all exceptions are checked exceptions, except for those indicated by Error, RuntimeException, and their subclasses

Need to specify the exception in a throws clause when defining the method that can throw it

The Three Kinds of Exceptions

2 - Error

exception conditions that are external to the application

the application usually cannot anticipate or recover from

e.g., hardware or system malfunction, java.lang.IOError

Error are not subject to the Catch or Specify Requirement

No need to specify the exception when defining the method

Classes that models errors are subclasses of java.lang.Error

The Three Kinds of Exceptions

3 - Runtime exception

exceptional conditions that are internal to the application

the application usually cannot anticipate or recover from

e.g., bugs, logic error, improper use of an API, NullPointerException

The application can catch this exception, but it makes more sense to eliminate the bug that caused the exception to occur

Runtime exceptions are not subject to the Catch or Specify Requirement

Runtime exceptions are those indicated by RuntimeException and its subclasses.

Errors and runtime exceptions are collectively known as unchecked exceptions.

Scala doesn’t use checked exceptions

In Scala we also throw errors similarly to Java, but the handling is simplified:

Code that might throw certain exceptions ***can*** be enclosed in *a try statement that catches the exception*. The “try" ***can*** provide (optionally) a handler for the exception

Roadmap

1. Why an exception mechanism?
2. What is an exception?
3. The Catch or Specify Requirement
4. **How to throw exception**
5. Operations on an exception
6. Exception to abort recursion

Throwing an exception

Use the Scala keyword “throw”

Throwing an exception is realized with the instruction:

```
throw object
```

where `object` is an object having the type Throwable

Catching an exception

Use the Scala keyword “catch”, followed by a list of cases.

```
var s = …
try {
  Integer.parseInt(s.trim)
} catch {
  case e: Exception => …
}
```

Creating a variable for this case…

… whose type is Exception

More than one catch is possible

```
var text = ""
try {
  text = openAndReadAFile(filename)
} catch {
  case fnf: FileNotFoundException => fnf.printStackTrace()
  case ioe: IOException => ioe.printStackTrace()
}
```

The catch clauses are ordered. The first handler that matches for the class of the exception is used.

Example #1

```
object Example {
  class E extends Exception
  class E2 extends E

  def foo(): Unit = throw new E

  def main(argv: Array[String]): Unit = {
    try{
      foo()
    }
    catch {
      case e: E2 => println("Handler E2")
      case e: E => println("Handler E")
    }
  }
}
```

The execution prints “Handler E” because foo() throws an instance of E

Example #2

```
object Example {
  class E extends Exception
  class E2 extends E

  def foo(): Unit = throw new E

  def main(argv: Array[String]): Unit = {
    try{
      foo()
    }
    catch {
      case e: E => println("Handler E")
      case e: E2 => println("Handler E2")
    }
  }
}
```

E is caught before E2

Example #3

```
object Example {
  class E extends Exception
  class E2 extends E

  def foo(): Unit = throw new E2

  def main(argv: Array[String]): Unit = {
    try{
      foo()
    }
    catch {
      case e: E2 => println("Handler E2")
      case e: E => println("Handler E")
    }
  }
}
```

The execution prints “Handler E2” because foo() throws an instance of E2

Example #4: Exceptions and sub typing

```
object Example {
  class E extends Exception
  class E2 extends E

  def foo(): Unit = throw new E2

  def main(argv: Array[String]): Unit = {
    try{
      foo()
    }
    catch {
      case e: E => println("Handler E")
      case e: E2 => println("Handler E2")
    }
  }
}
```

The execution prints “Handler E” throws an E2, and E2 is an E

Example #5: Exception may be thrown again

```
object Example {
  class E extends Exception
  class E2 extends E

  def foo(): Unit = throw new E

  def main(argv: Array[String]): Unit = {
    try{
      foo()
    }
    catch {
      case e: E => throw e
      case e: E2 => println("Handler E2")
    }
  }
}
```

Caught exception is thrown again

The Finally block

The finally block *always* executes when the try block exits

Putting *cleanup code in a finally block* is always a good practice, even when no exceptions are anticipated

```
try {
  text = openAndReadAFile(filename)
} catch {
  case fnf: FileNotFoundException => fnf.printStackTrace()
  case ioe: IOException => ioe.printStackTrace()
} finally {
  // close your resources here
  println("Came to the 'finally' clause.")
}
```

The Finally block

The *finally* block is a key tool for preventing *resource leaks*

When closing a file or otherwise recovering resources, place the code in a finally block to ensure that resource is always recovered

Exiting the try block...

```
object Example2 {
  def foo: Int = {
    try {
      println("try")
      5
    }
    finally {
      println("finally")
      10
    }
  }

  def main(argv: Array[String]): Unit = {
    println(foo)
  }
}
```

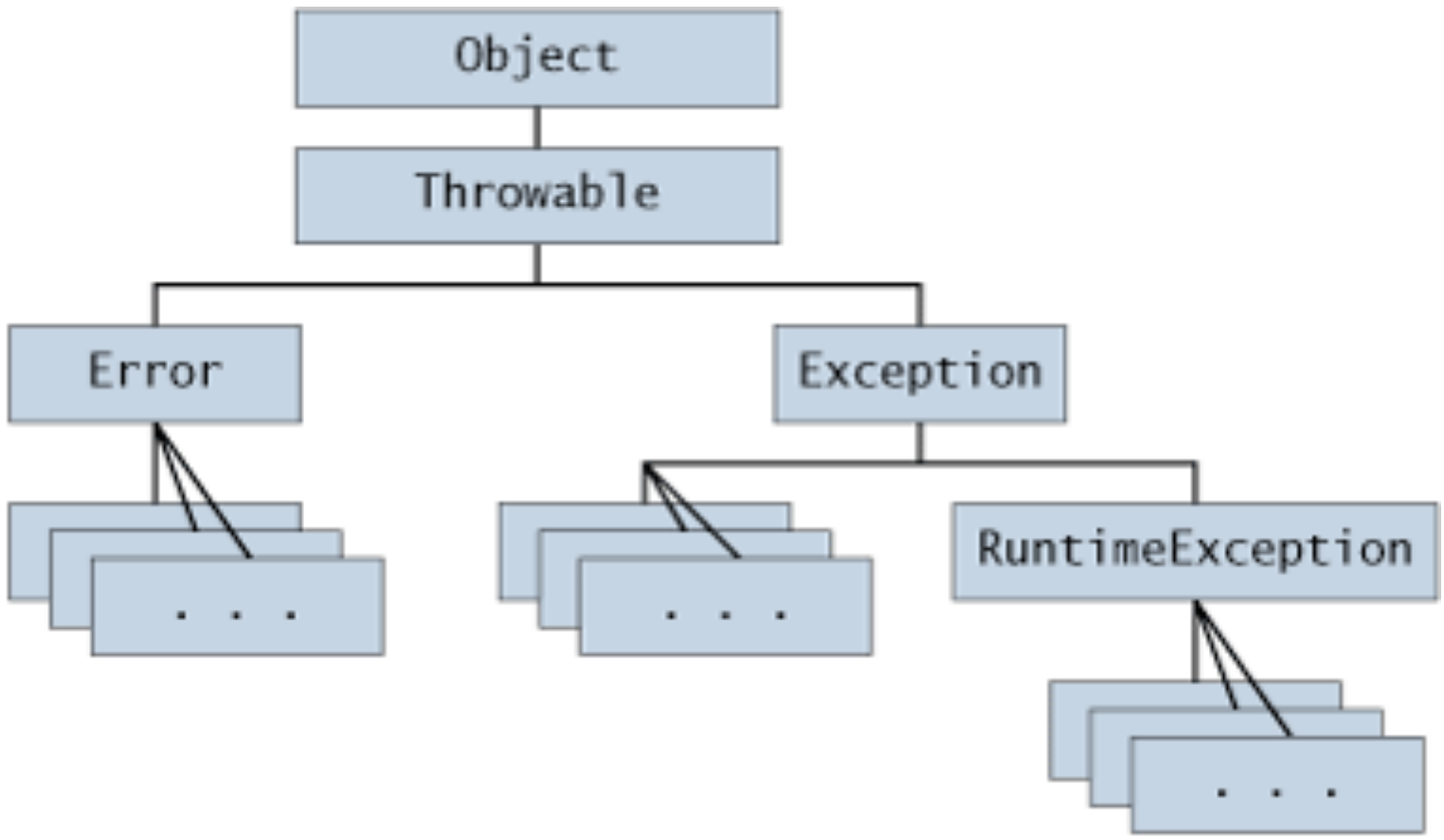
The finally block is evaluated only for side effects; the value of the block as a whole is the value of the last expression in the try (if no exception was thrown) or catch (if one was).

It prints:
try
finally
5

What does the following print?

Roadmap

1. Why an exception mechanism?
2. What is an exception?
3. The Catch or Specify Requirement
4. How to throw exception
5. **Operations on an exception**
6. Exception to abort recursion


Object
Throwable
Error
Exception
. . .
. . .
RuntimeException
. . .

Operations on an exception

Defined in the Throwable class

Throwable	**fillInStackTrace**() Fills in the execution stack trace.
Throwable	**getCause**() Returns the cause of this throwable or `null` if the cause is nonexistent or unknown.
String	**getLocalizedMessage**() Creates a localized description of this throwable.
String	**getMessage**() Returns the detail message string of this throwable.
StackTraceElement[]	**getStackTrace**() Provides programmatic access to the stack trace information printed by printStackTrace().
Throwable	**initCause**(Throwable cause) Initializes the *cause* of this throwable to the specified value.
void	**printStackTrace**() Prints this throwable and its backtrace to the standard error stream.

Roadmap

1. Why an exception mechanism?
2. What is an exception?
3. The Catch or Specify Requirement
4. How to throw exception
5. Operations on an exception
6. **Exception to abort recursion**

Aborting recursion

Exiting deep recursions may be complicated time to time

Checks may be necessary at different places

Aborting recursion

```
class Matrix3D() {
  val line0: Array[Int] = Array(0, 0, 0)
  val line1: Array[Int] = Array(1, 0, 0)
  val mat1: Array[Array[Int]] = Array[Array[Int]](line0, line0, line0)
  val mat2: Array[Array[Int]] = Array[Array[Int]](line1, line0, line0)
  val mat3: Array[Array[Int]] = Array[Array[Int]](line0, line1, line0)

  private var table = Array[Array[Array[Int]]](mat1, mat2, mat3)

  …
}
object Matrix3D {
  def main(args: Array[String]): Unit = {
    val m = new Matrix3D()
    println(m.numberOf2DMatricesWith(1))
  }
}
```

Aborting recursion

```
def numberOf2DMatricesWith(v: Int): Int = {
  var nbOfMatching = 0
  for(z <- 0 to table.length-1) {
    var doesContain = false
    for(y <- 0 to table.length-1) {
      for (x <- 0 to table(y).length-1) {
        if (table(z)(y)(x) == v) doesContain = true
      }
    }
    if (doesContain) nbOfMatching += 1
  }
  nbOfMatching
}
```

What do you think about this method?

Aborting recursion

```
def numberOf2DMatricesWith(v: Int): Int = {
  var nbOfMatching = 0
  for(z <- 0 to table.length-1) {
    try{
      for(y <- 0 to table.length-1) {
        for (x <- 0 to table(y).length-1) {
          if (table(z)(y)(x) == v) throw new Throwable()
        }
      }
    } catch{
      case e: Throwable => nbOfMatching += 1
    }
  }
  nbOfMatching
}
```

With this version, no unnecessary iteration is done

Aborting recursion

Could be handy in some case.

But don’t abuse it!

Things we did not see

Try with resources

Multiple exceptions declaration

What you should know

Why an *exception mechanism* help managing errors?

How to *throw* an exception?

What are the *different kinds* of exceptions?

How does the system look for an handler?

What is the difference between a *checked* and *unchecked* exceptions?

Why the finally block is appropriate for *clean-up* code?

License