Exceptions in Java

Alexandre Bergel
http://bergel.eu
29-11-2021

The Java programming language uses *exceptions*
to handle errors and other exceptional events

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

Source

https://docs.oracle.com/javase/tutorial/essential/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, tacking care of the potential errors clut eadability

Example:

What happens if the file can't be opened?

```
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, tacking care of the potential errors clut ity

Example:

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

```
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, tacking care of the potential errors clut ity

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, tacking care of the potential errors clut ity

Example:

What happens if the read fails?

```
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, tacking care of the potential errors clut ity

Example:

What happens if the file can't be closed?

```
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 (fileOpenFailed) {
       doSomething;
    } catch (sizeDeterminationFailed) {
        doSomething;
    } catch (memoryAllocationFailed) {
        doSomething;
    } catch (readFailed) {
        doSomething;
    } catch (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 a file with an empty path

```
package java.io;

public class File implements Serializable, Comparable<File> {

   public File(String pathname) {

     if (pathname == null) {
         throw new NullPointerException();
     }
     this.path = fs.normalize(pathname);
     this.prefixLength = fs.prefixLength(this.path);
   }
  ...
}
```

Creating a file with an empty path

```
package java.io;

public class File implements Serializable, Comparable<File> {

   public File(String pathname) {

     if (pathname == null) {
         throw new NullPointerException();
     }
     this.path = fs.normalize(pathname);
     this.prefixLength = fs.prefixLength(this.path);
   }
  ...
}
```

new File("/tmp/foo") => OK

new File(null) => throws a NullPointerException

Defining an exception class

```
package java.lang;

public class NullPointerException extends RuntimeException {

   ...

}
```

java.lang

Class NullPointerException

java.lang.Object
  └ java.lang.Throwable
    └ java.lang.Exception
      └ java.lang.RuntimeException
        └ **java.lang.NullPointerException**

All Implemented Interfaces:

Serializable

Looking for an handler

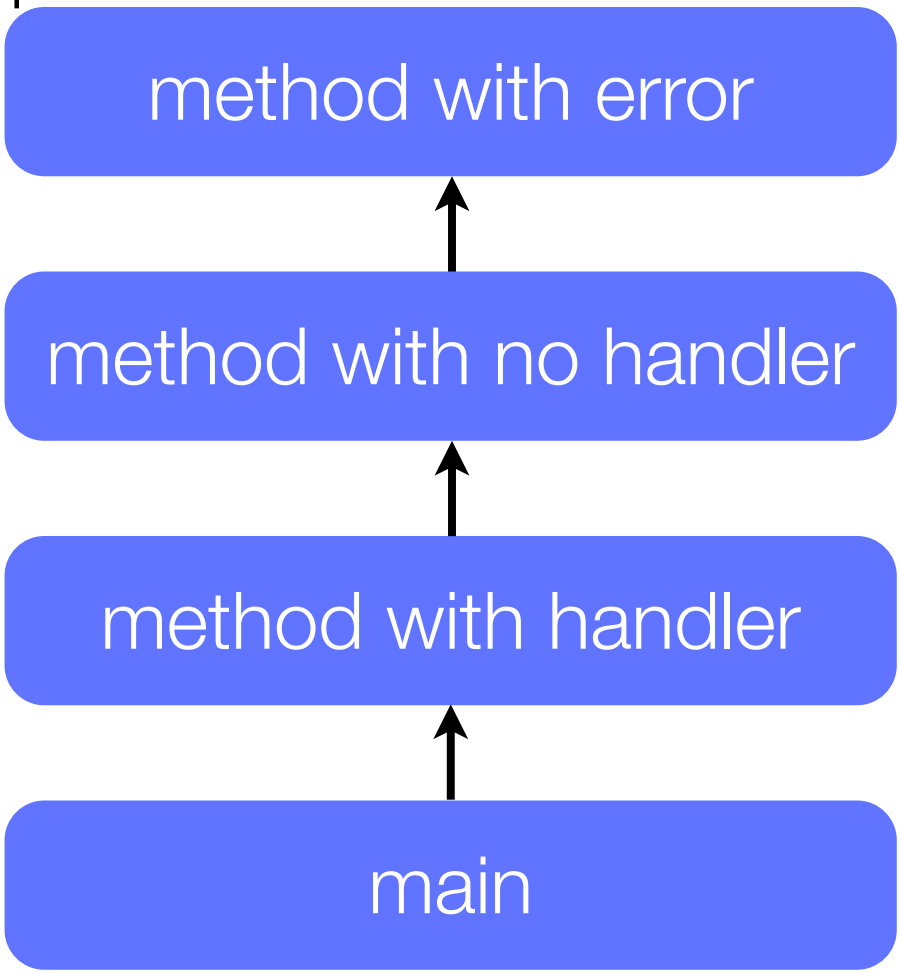
After a method throws an exception, the runtime system *attempts to find something to handle it*

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*

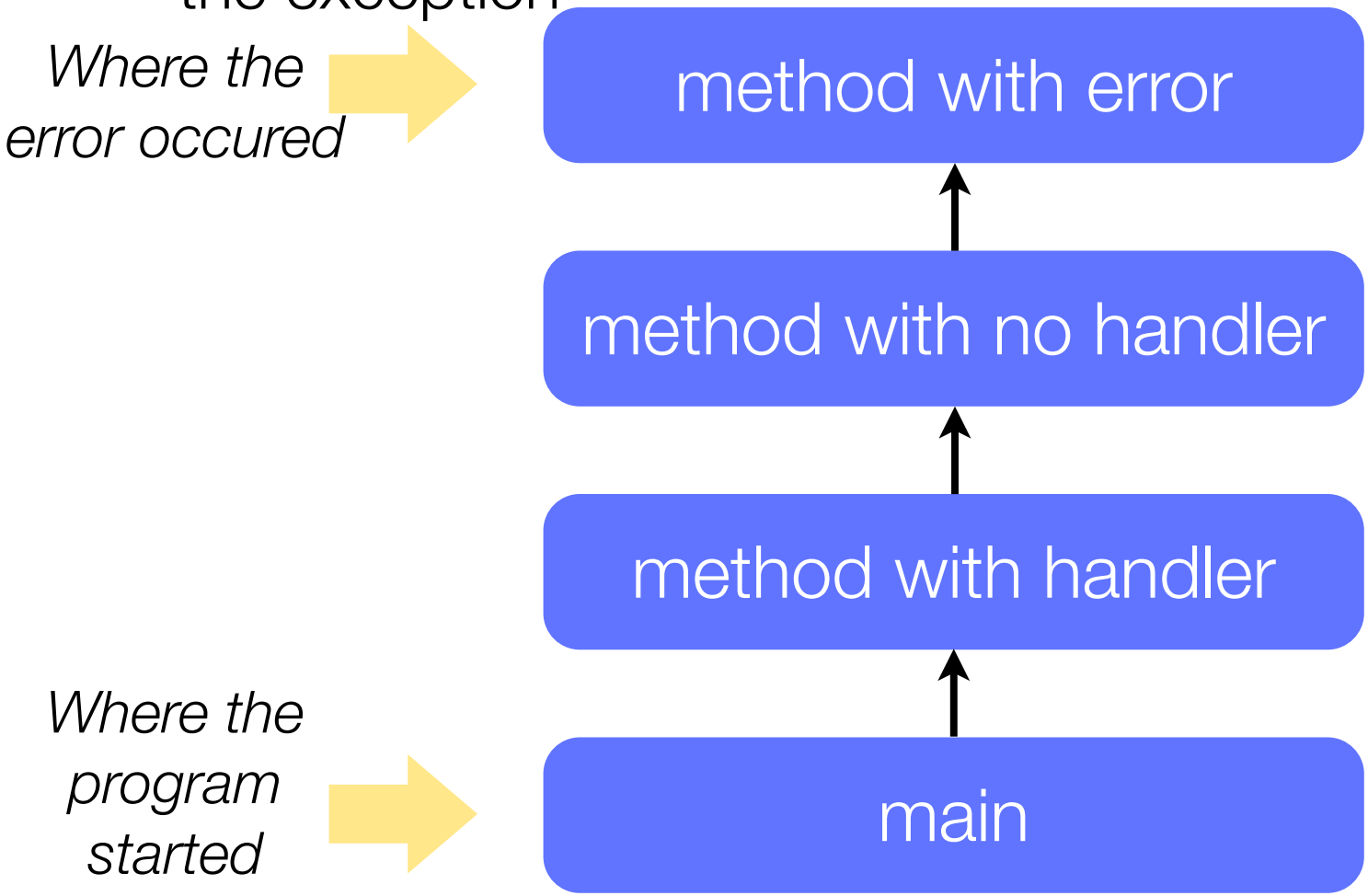
Searching the call stack

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



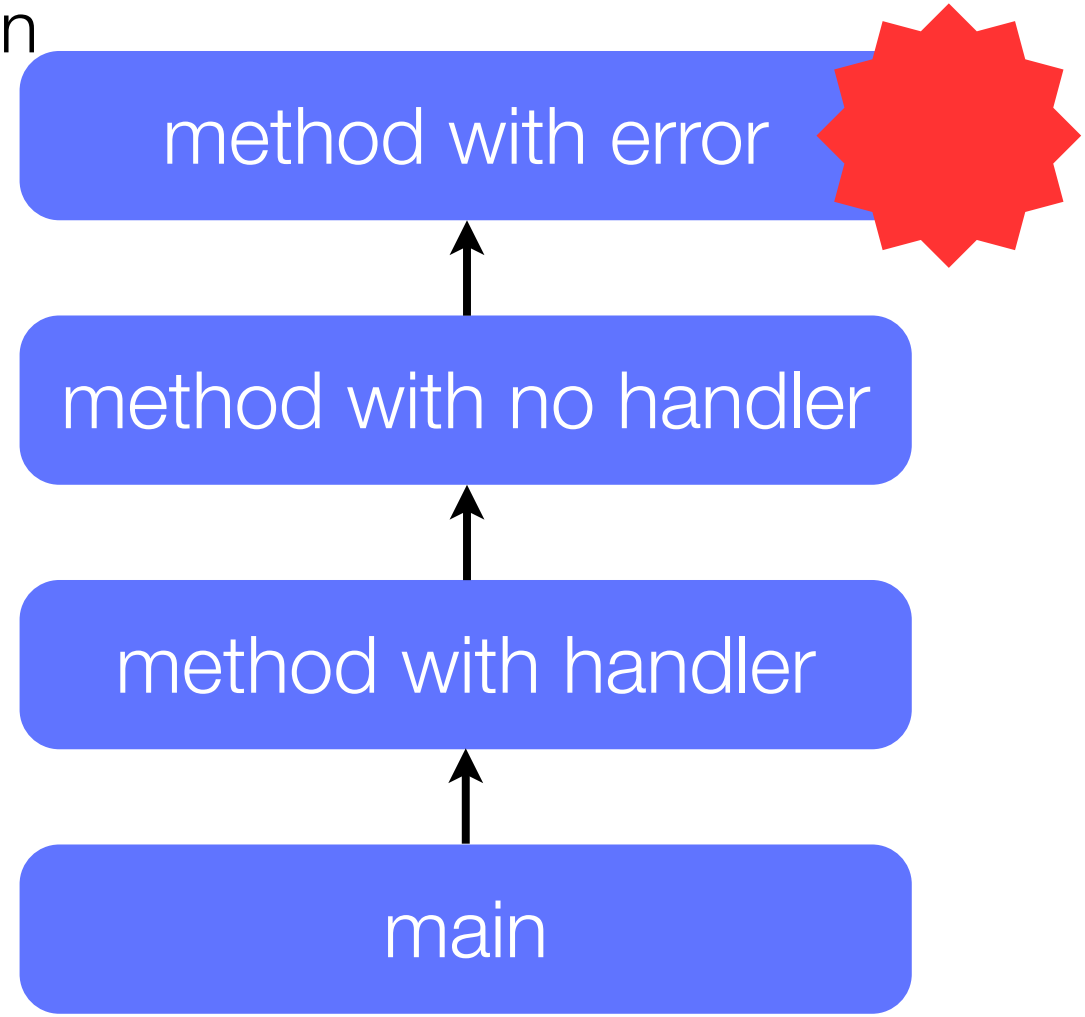
Searching the call stack

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



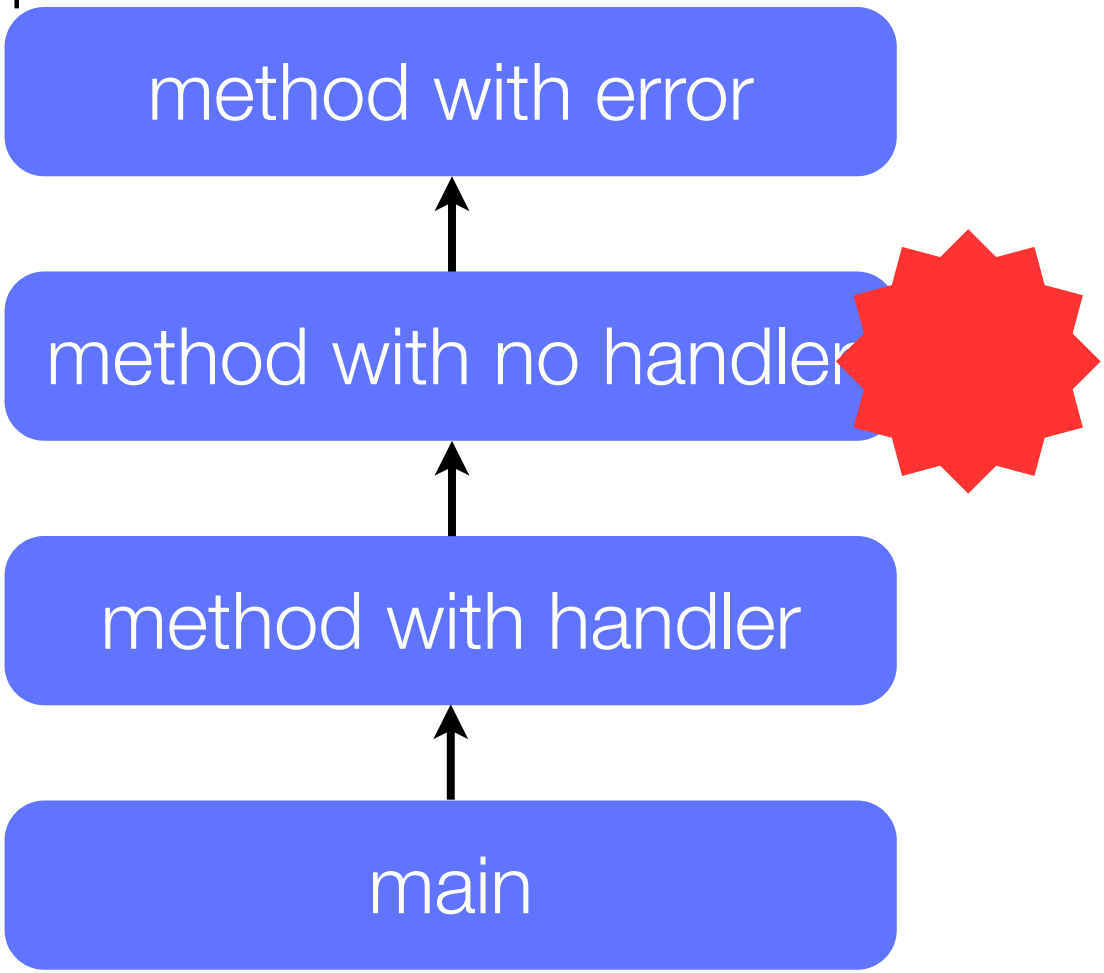
Searching the call stack

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



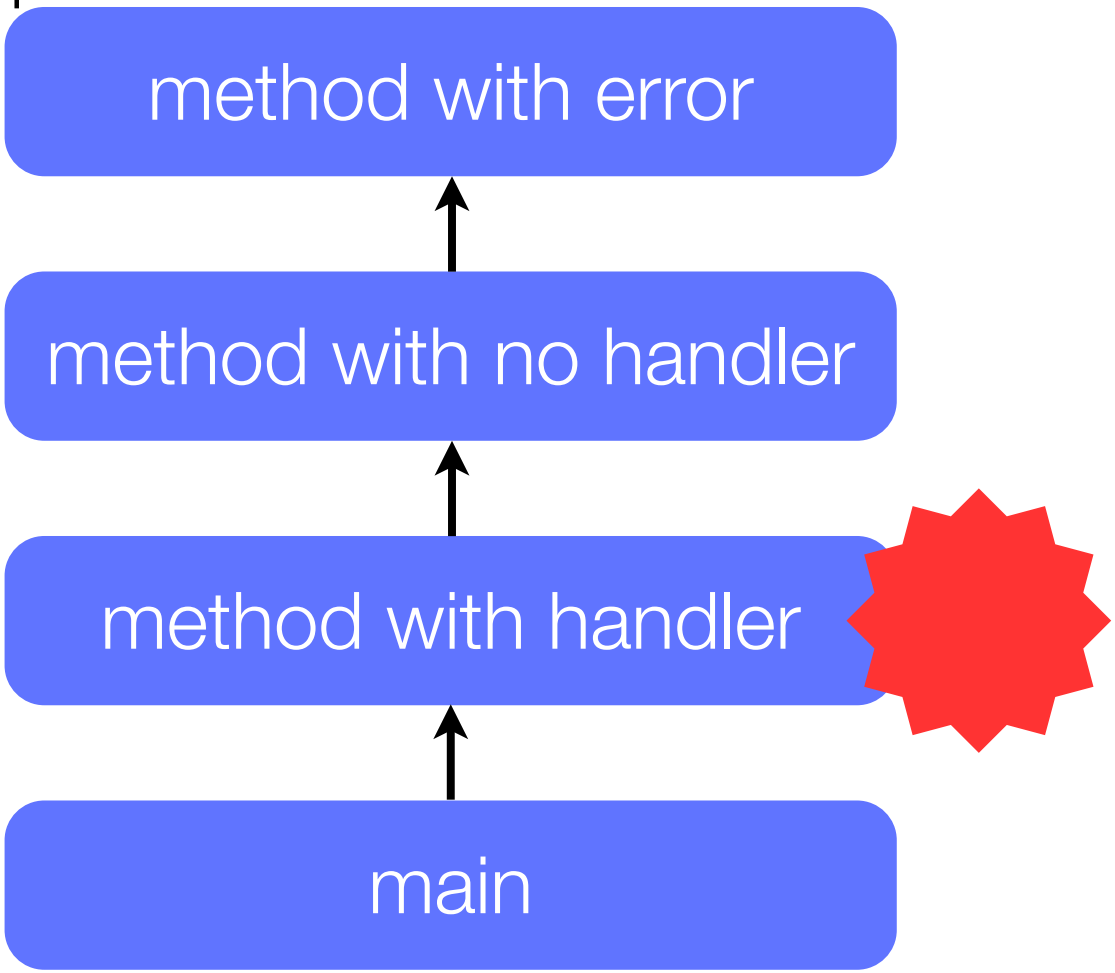
Searching the call stack

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



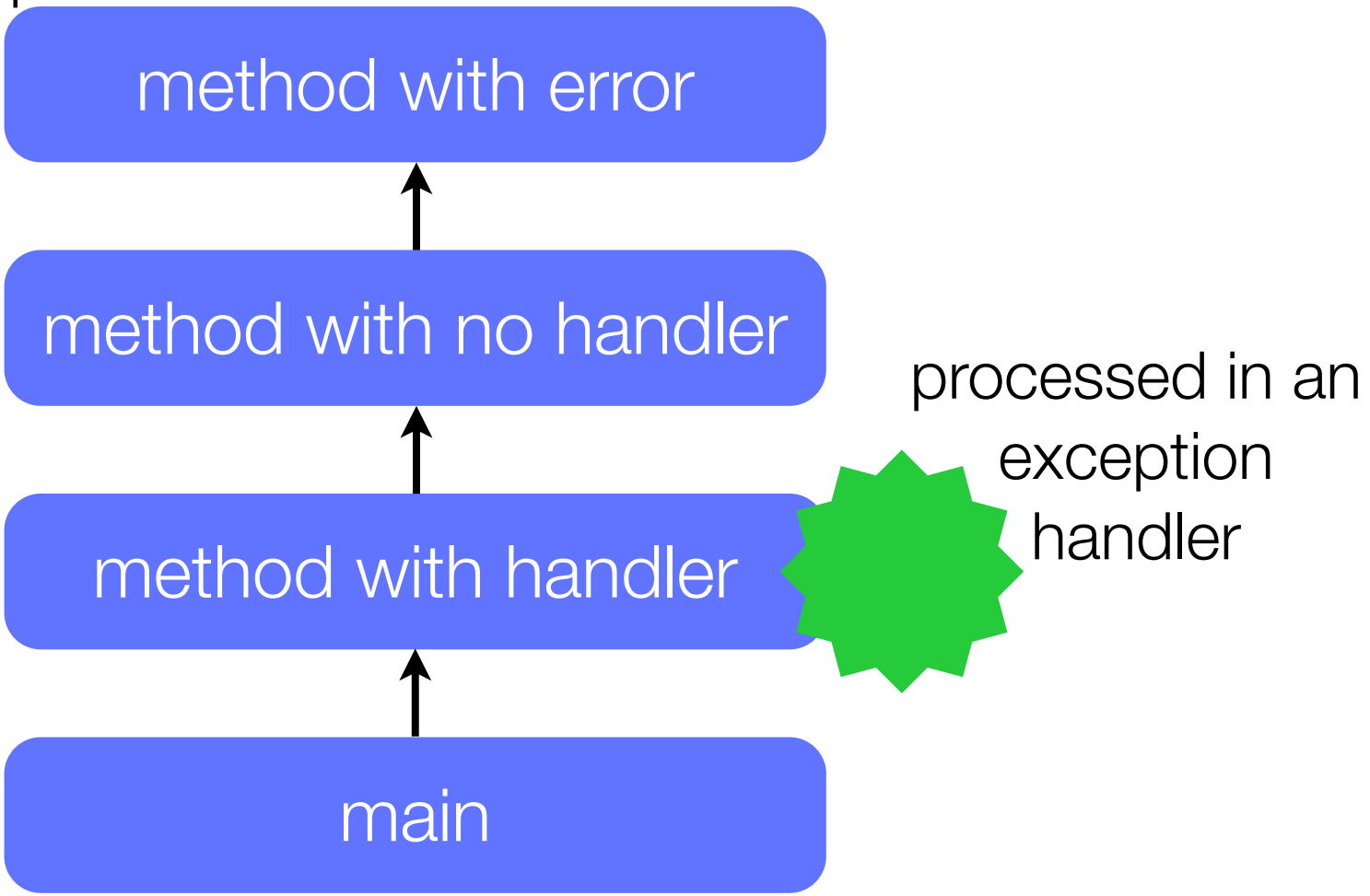
Searching the call stack

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



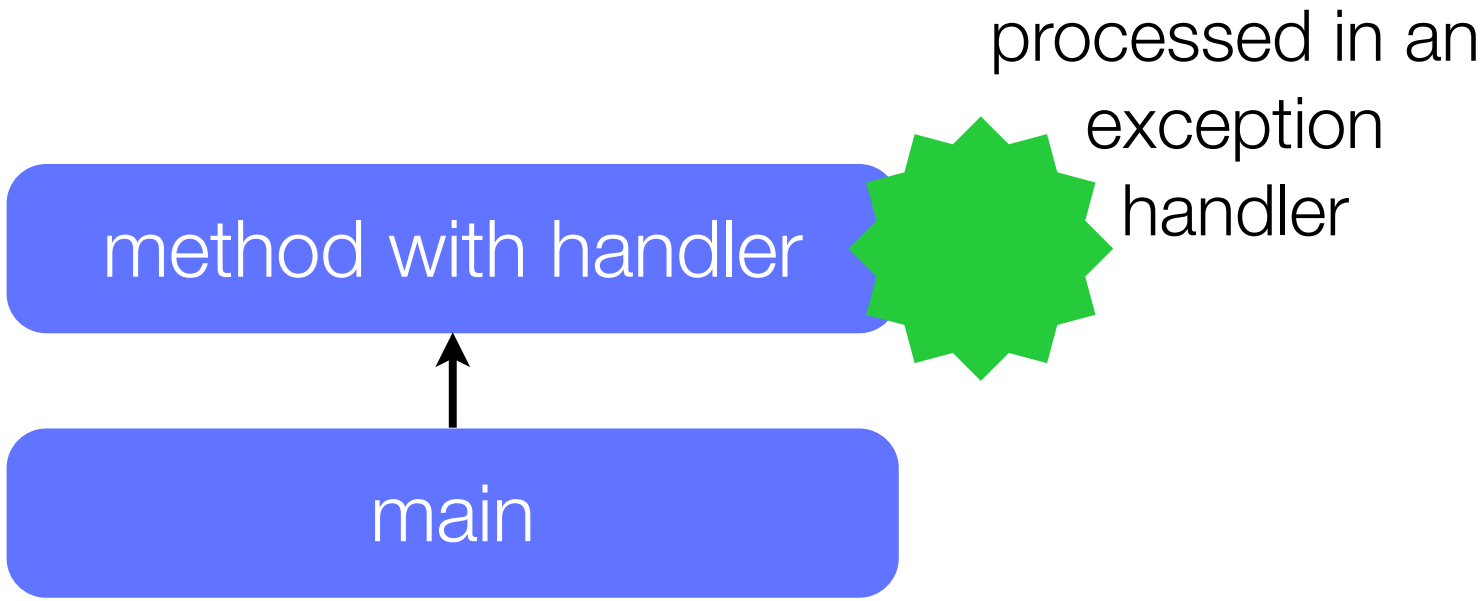
Searching the call stack

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



Searching the call stack

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



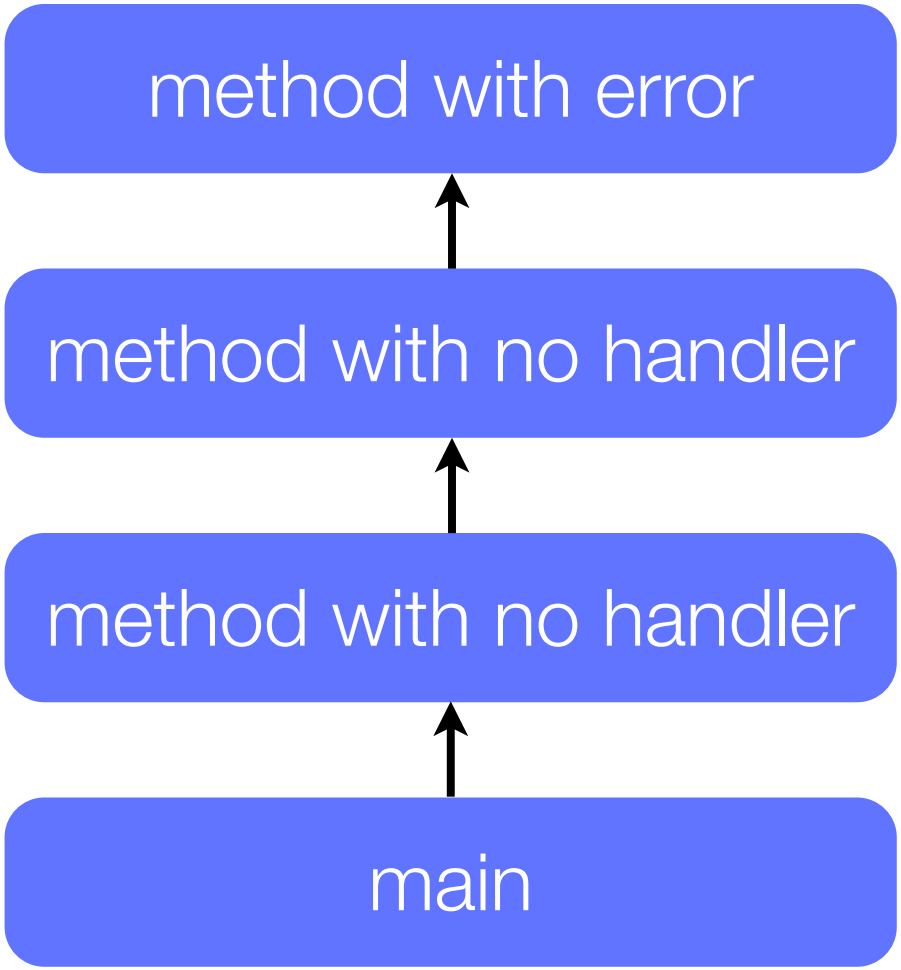
Searching the call stack

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

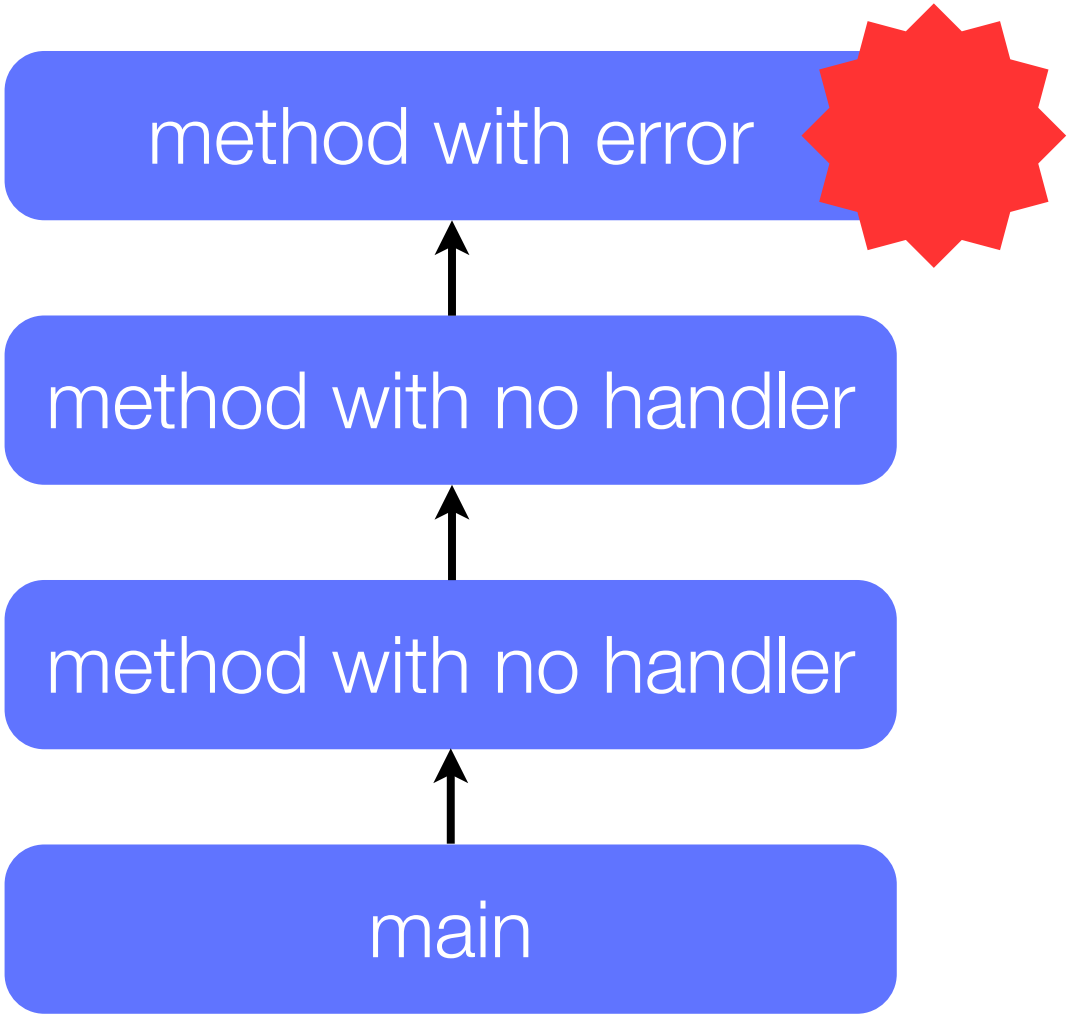
The search begins with the method in which the error occurred and proceeds through the call stack in the reverse order in which the methods were called

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

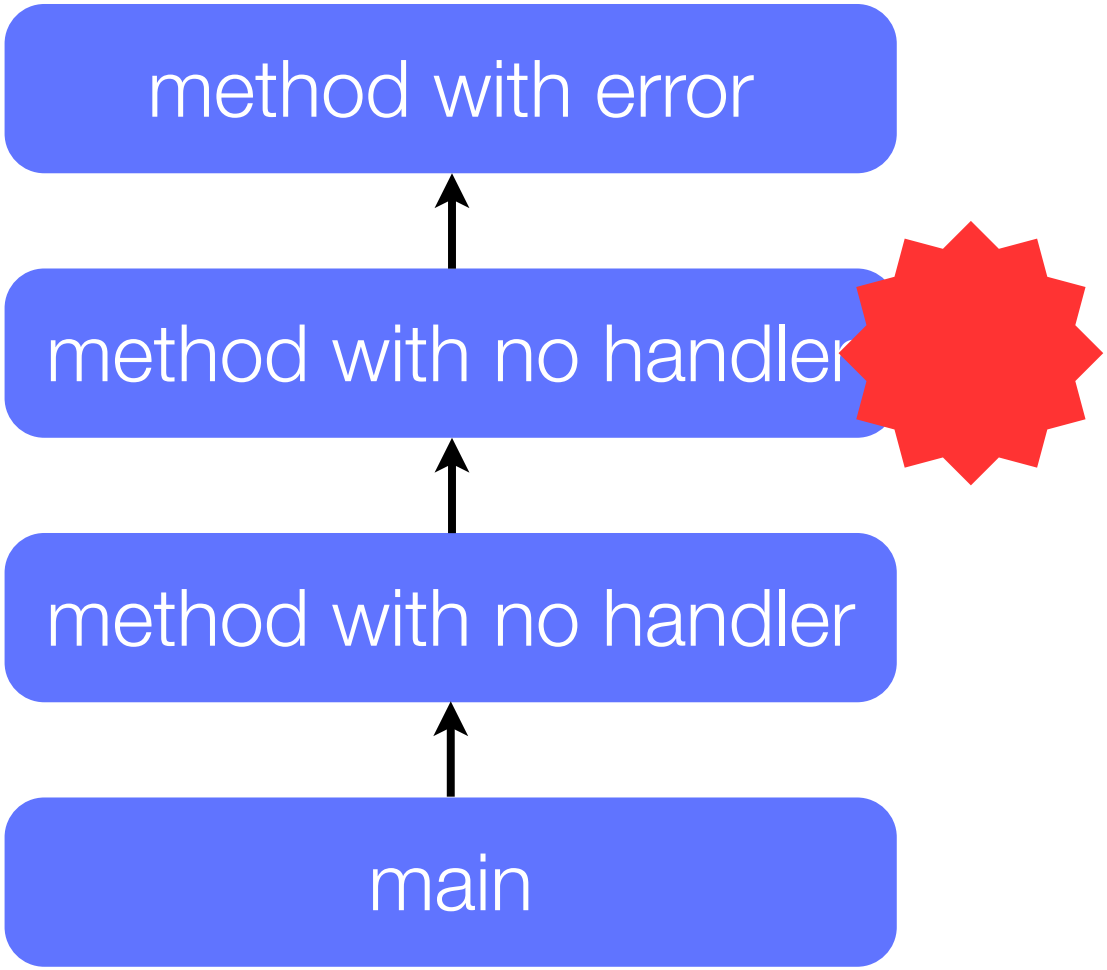
And if there is no handler?



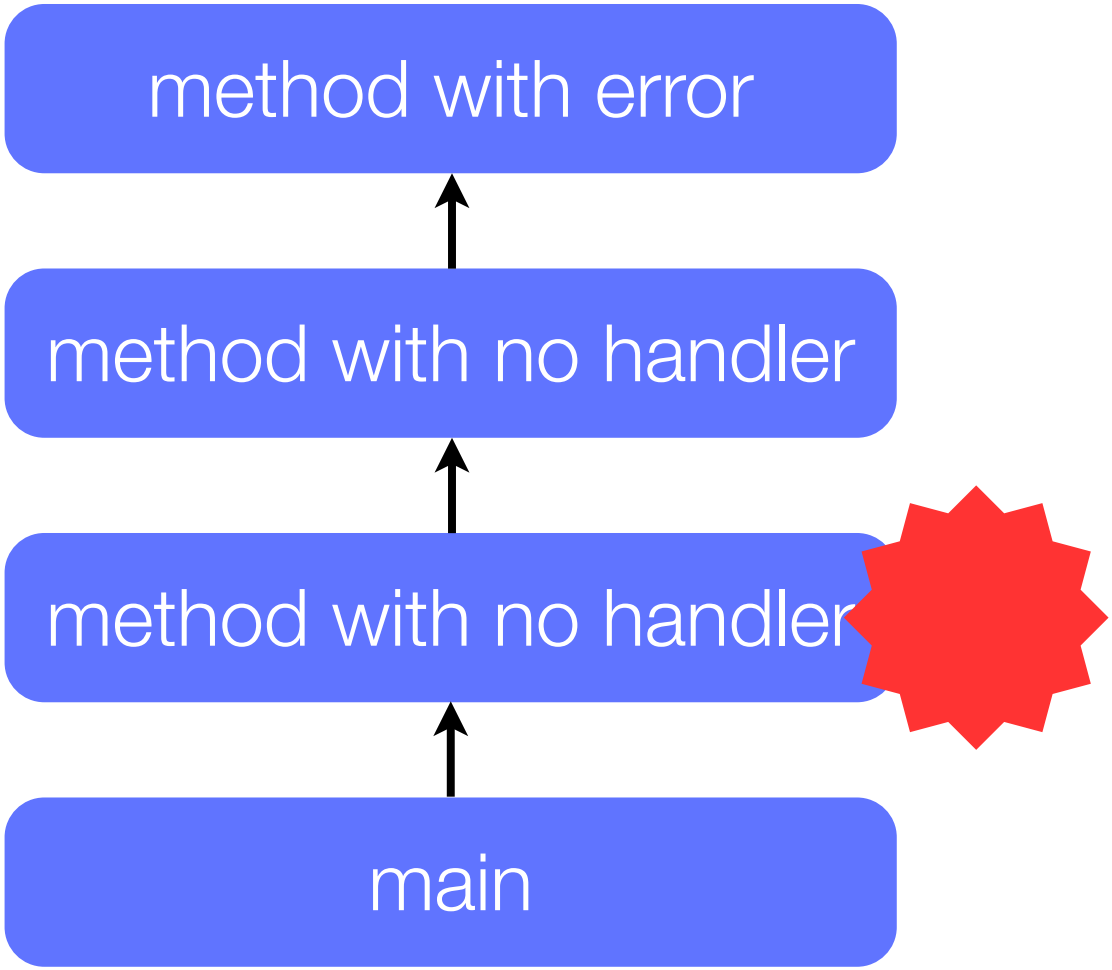
And if there is no handler?



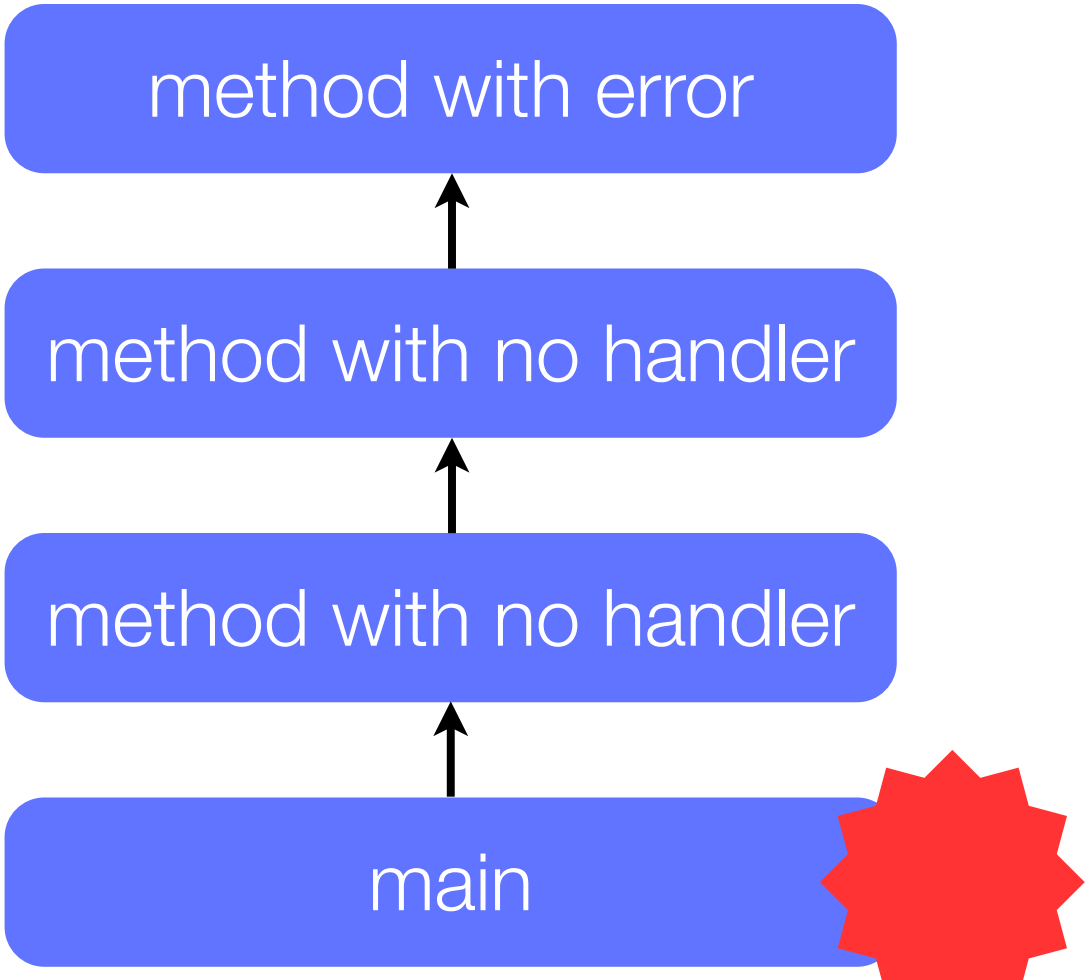
And if there is no handler?



And if there is no handler?

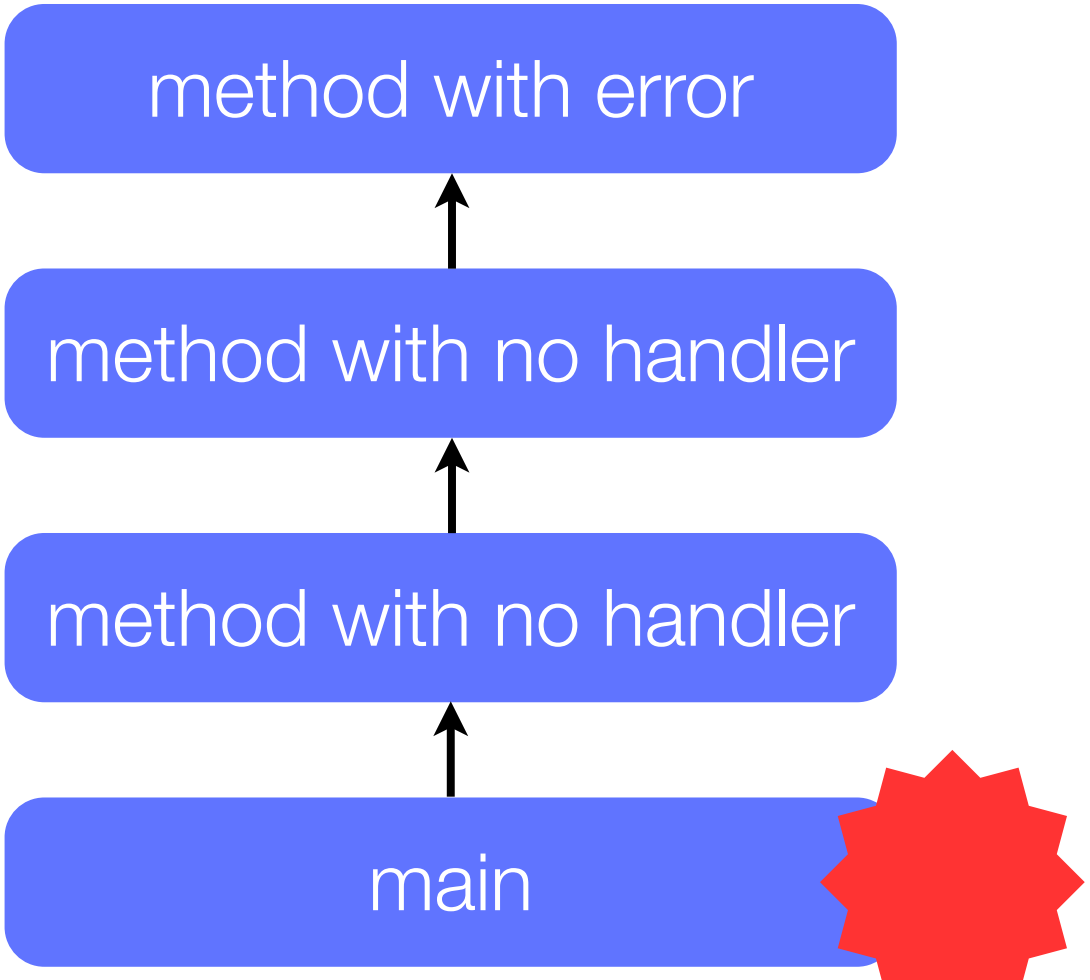


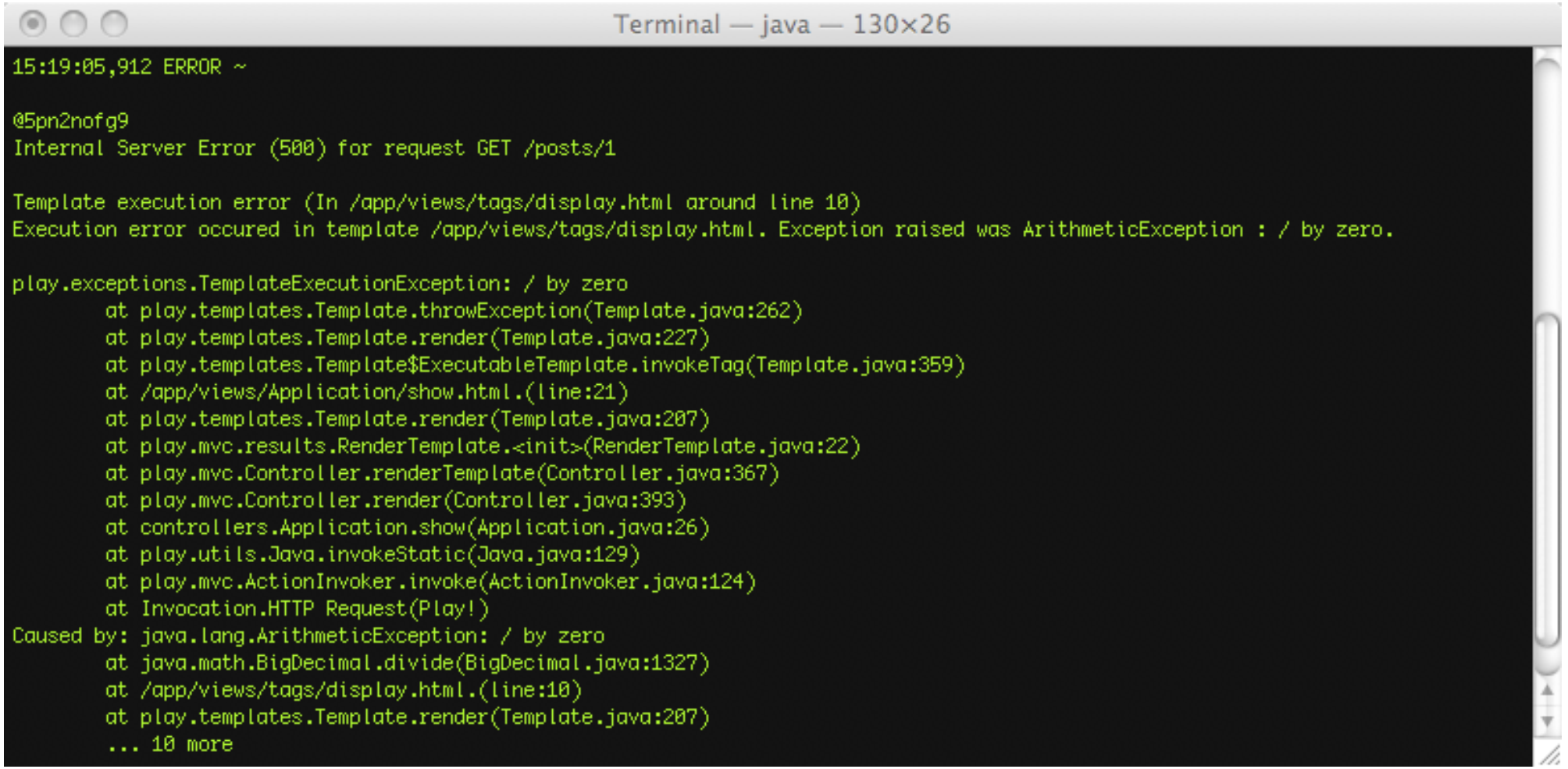
And if there is no handler?



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.

Which kind of exception should I use?

It is likely that you have to use checked exception in your application

- Create a subclass of `java.lang.Exception`
- Use `throw`, `try` and `catch` to raise an exception and add the proper handler in case of something goes as not expected

Except in some rare cases, you can define an handler for error (i.e., a subclass of Error)

- This could be the case for example that in your application you consume a lot of memory, and telling the user when no memory is left

You will probably never have to create a subclass of **`Error`** or **`RuntimeError`**

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 Java keyword “throw”

Throwing an exception is realized with the instruction:

```
throw object
```

where `object` is an object having the type Throwable

More than one catch is possible

```
try {
   ...
} catch (ExceptionType1 name) {
    ...
} catch (ExceptionType2 name) {
    ...
}
```

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

Example

```
class Example {
 static class E extends Exception {}
 static class E2 extends E {}
 static void foo() throws E {
   throw new E();
 }
 public static void main(String[] argv) {
   try {
     foo();
   }
   catch(E2 e){System.out.println("Handler E2");}
   catch(E e){System.out.println("Handler E");}
 }
}
```

Dynamic type of the exception is used to look for the hander

```
class Example {
 static class E extends Exception {}
 static class E2 extends E {}
 static void foo() throws E {
   throw new E();
 }
 public static void main(String[] argv) {
   try {
     foo();
   }
   catch(E2 e){System.out.println("Handler E2");}
   catch(E e){System.out.println("Handler E");}
 }
}
```

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

Not compiling example

```
class Example {
 static class E extends Exception {}
 static class E2 extends E {}
 static void foo() throws E {
   throw new E();
 }
 public static void main(String[] argv) {
   try {
     foo();
   }
   catch(E e){System.out.println("Handler E");}
   catch(E2 e){System.out.println("Handler E2");}
 }
}
```

Not compiling example

```
class Example {
 static class E extends Exception {}
 static class E2 extends E {}
 static void foo() throws E {
   throw new E();
 }
 public static void main(String[] argv) {
   try {
     foo();
   }
   catch(E e){System.out.println("Handler E");}
   catch(E2 e){System.out.println("Handler E2");}
 }
}
```

E is caught before E2

Not compiling example

```
class Example {
 static class E extends Exception {}
 static class E2 extends E {}
 static void foo() throws E {
   throw new E();
 }
 public static void main(String[] argv) {
   try {
     foo();
   }
   catch(E e){System.out.println("Handler E");}
   catch(E2 e){System.out.println("Handler E2");}
 }
}
```

This code does not compile because catch(E2) cannot be used. The handler catch(E) comes before catch(E2), and E is a supertype of E2

Dynamic type of the exception is used to look for the hander

```
class Example {
 static class E extends Exception {}
 static class E2 extends E {}
 static void foo() throws E {
   throw new E2();
 }
 public static void main(String[] argv) {
   try {
     foo();
   }
   catch(E2 e){System.out.println("Handler E2");}
   catch(E e){System.out.println("Handler E");}
 }
}
```

The execution prints “Handler E2” because foo() throws an instance of E2, independently if foo is declared as “throws E”

Dynamic type of the exception is used to look for the hander

```
class Example {
 static class E extends Exception {}
 static class E2 extends E {}
 static void foo() throws E {
   throw new E2();
 }
 public static void main(String[] argv) {
   try {
     foo();
   }
   catch(E2 e){System.out.println("Handler E2");}
   catch(E e){System.out.println("Handler E");}
 }
}
```

The execution prints “Handler E2” because foo() throws an instance of E2, independently if foo is declared as “throws E”

throws and try/catch combined

```
class Ex1 extends Exception {}
class Ex2 extends Exception {}
class Example3 {
  static void foo() throws Ex1 { throw new Ex1();}
  static void bar() throws Ex2 { throw new Ex2();}

  static void zork() throws Ex2 {
    try {
      foo();
      bar();
    }
    catch(Ex1 e) { }
  }
}
```

throws and try/catch combined

```
class Ex1 extends Exception {}
class Ex2 extends Exception {}
class Example3 {
  static void foo() throws Ex1 { throw new Ex1();}
  static void bar() throws Ex2 { throw new Ex2();}

  static void zork() throws Ex2 {
    try {
      foo();
      bar();
    }
    catch(Ex1 e) { }
  }
}
```

Sequentially calling foo and bar may throws exception Ex1 and Ex2

throws and try/catch combined

```
class Ex1 extends Exception {}
class Ex2 extends Exception {}
class Example3 {
  static void foo() throws Ex1 { throw new Ex1();}
  static void bar() throws Ex2 { throw new Ex2();}

  static void zork() throws Ex2 {
    try {
      foo();
      bar();
    }
    catch(Ex1 e) { }
  }
}
```

Ex2 is may be thrown. The caller of zork() should take care of this then

Ex1 is caught

Exception may be thrown again

```
class Ex1 extends Exception {}
class Ex2 extends Exception {}
class Example3 {
  static void foo() throws Ex1 { throw new Ex1();}
  static void bar() throws Ex2 { throw new Ex2();}

  static void zork() throws Ex2, Ex1 {
    try {
      foo();
      bar();
    }
    catch(Ex1 e) { throw e; }
  }
}
```

Exception may be thrown again

```
class Ex1 extends Exception {}
class Ex2 extends Exception {}
class Example3 {
  static void foo() throws Ex1 { throw new Ex1();}
  static void bar() throws Ex2 { throw new Ex2();}

  static void zork() throws Ex2, Ex1 {
    try {
      foo();
      bar();
    }
    catch(Ex1 e) { throw e; }
  }
}
```

zork() may throw either a Ex1 or an Ex2

We throw the same exception 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 {
  ...
}
catch (ExceptionType1 name) {}
catch (ExceptionType2 name) {}
finally {
  // cleaning code here
}
```

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...

```
class Example2 {
    static int foo() {
        try {
            return 5;
        }
        finally {
            return 10;
        }
    }
    public static void main(String[] argv) {
        System.out.println(foo());
    }
}
```

What does the following print?

Exiting the try block...

```
class Example2 {
    static int foo() {
        try {
            return 5;
        }
        finally {
            return 10;
        }
    }
    public static void main(String[] argv) {
        System.out.println(foo());
    }
}
```

It prints 10, since the finally is executed after the control flow has exited the try block

Specifying the Exceptions Thrown by a Method

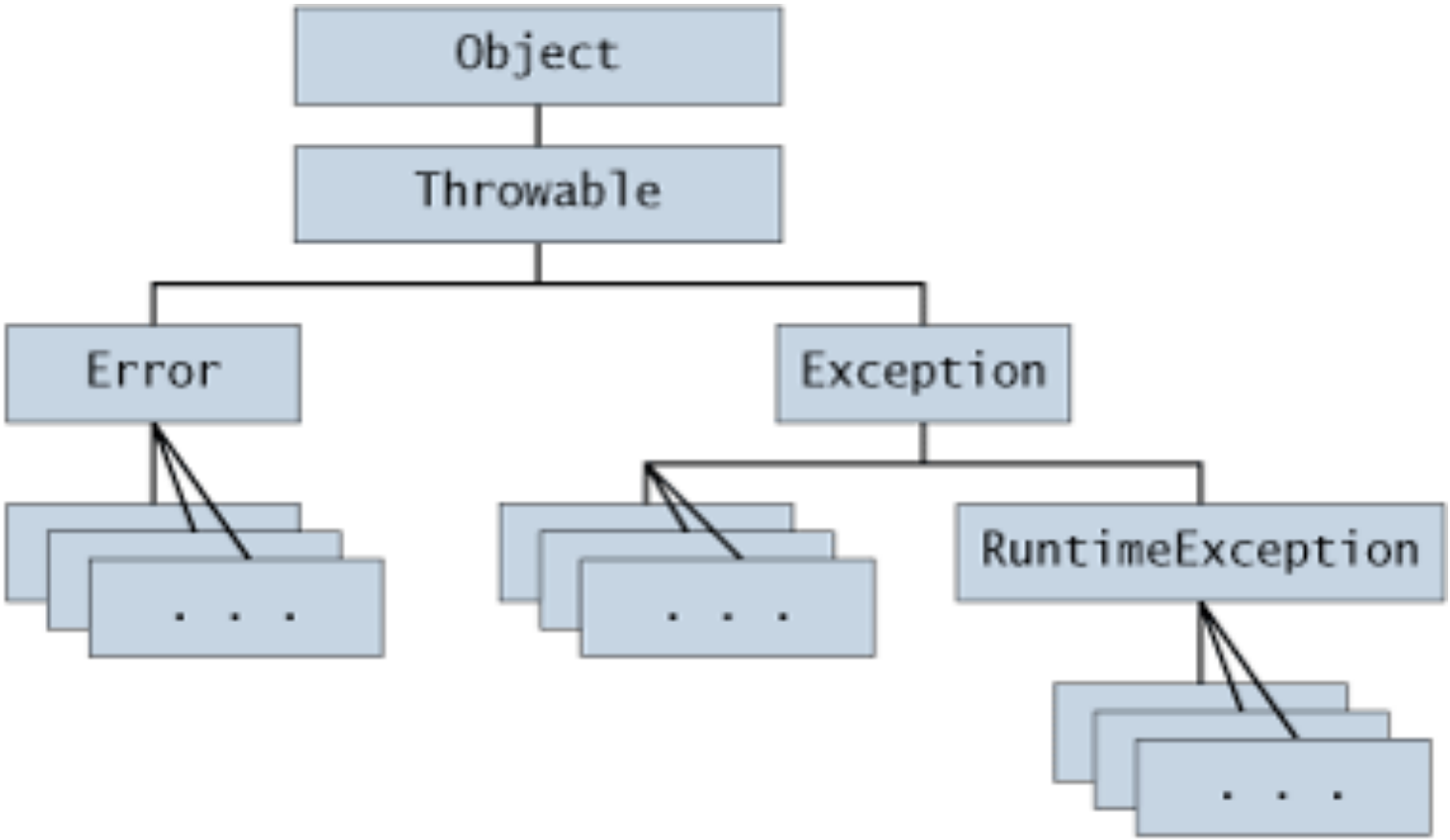
It is appropriate to early catch exceptions

Sometimes, however, it's better to let a method further up the call stack handle the exception

You need to use the *throws* keyword to delegate the responsibility of handling the error

Exception are thrown using the *throw* keyword

throw takes an expression as argument


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

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

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.

Creating a file with an empty path

```
package java.io;

public class File implements Serializable, Comparable<File> {

   public File(String pathname) {

     if (pathname == null) {
         throw new NullPointerException();
     }
     this.path = fs.normalize(pathname);
     this.prefixLength = fs.prefixLength(this.path);
   }
  ...
}
```

new File("/tmp/foo") => OK

new File(null) => throws a NullPointerException

Why having this check in File(String)?

Operations on an exception

In Java, exceptions can only be thrown, caught and re-thrown

Java is quite limited in that respect

Other operations are possible

For example, in Pharo

retry: to re-evaluate the protected block

retryUsing: to provide a value in place and re-evaluate the protected block

resume: resume the execution at the failure point

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

```
public class Matrix3D {
      private int[][][] table;

      Matrix3D() {
            int[] line0 = { 0, 0, 0 };
            int[] line1 = { 1, 0, 0 };
            int[][] mat1 = new int[][] { line0, line0, line0 };
            int[][] mat2 = new int[][] { line1, line0, line0 };
            int[][] mat3 = new int[][] { line0, line1, line0 };
            table = new int[][][] { mat1, mat2, mat3 };
      }
      …
      public static void main(String[] argv) {
            Matrix3D m = new Matrix3D();
            System.out.println(m.numberOf2DMatricesWith(1));
      }
}
```

Aborting recursion

```
int numberOf2DMatricesWith(int v) {
      int nbOfMatching = 0;
      for (int z = 0; z < table.length; z++) {
            boolean doesContain = false;

            for (int y = 0; y < table.length; y++) {
                  for (int x = 0; x < table[y].length; x++) {
                        if (table[z][y][x] == v)
                              doesContain = true;
                  }
            }
            if (doesContain)
                  nbOfMatching++;
      }
      return nbOfMatching;
}
```

What do you think about this method?

Aborting recursion

```
int numberOf2DMatricesWith(int v) {
      int nbOfMatching = 0;
      for (int z = 0; z < table.length; z++) {
            try {
                  for (int y = 0; y < table.length; y++) {
                        for (int x = 0; x < table[y].length; x++) {
                              if (table[z][y][x] == v)
                                    throw new Throwable();
                        }
                  }
            } catch (Throwable e) {
                  nbOfMatching++;
            }
      }
      return 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* exception?

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

Can you answer these questions?

Why the static type of the *throw* exception is not taken into account when looking for a handler?

Can you provide an example for each the 3 kind of exceptions?

How to decide the kind of exception when designing a class exception?

How exceptions and the program execution flow are related?

License

http://creativecommons.org/licenses/by-sa/2.5

creative commons

COMMONS DEED

Attribution-ShareAlike 2.5

You are free:

- to copy, distribute, display, and perform the work
- to make derivative works
- to make commercial use of the work

Under the following conditions:

Attribution. You must attribute the work in the manner specified by the author or licensor.

Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.

- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.