Exceptions in Java

Alexandre Bergel
abergel@dcc.uchile.cl
07/06/2010

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

http://java.sun.com/docs/books/tutorial/essential/exceptions

Roadmap

Why an exception mechanism?

What is an exception?

The Catch or Specify Requirement

How to throw exception

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;
    }
}
```

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);
    }
  ...
}
```

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*

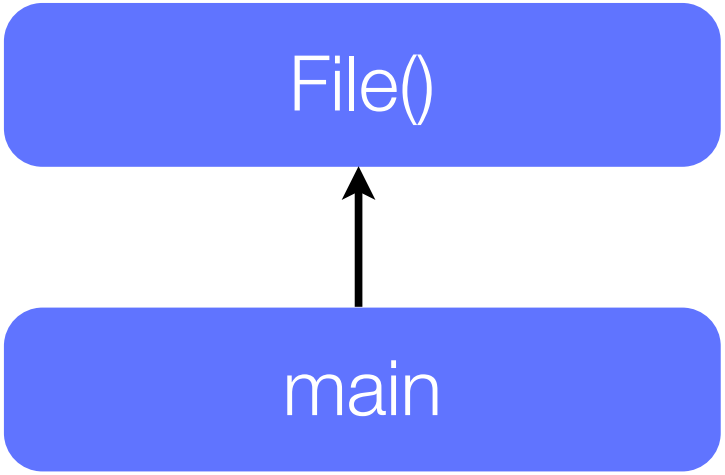
From the Web Server example

```
public static void main(String[] args) {
    try {
        new SimpleWebServer(new File(null), 8000);
    }
    catch (Exception e) {
        System.out.println(e);
    }
}
```

main

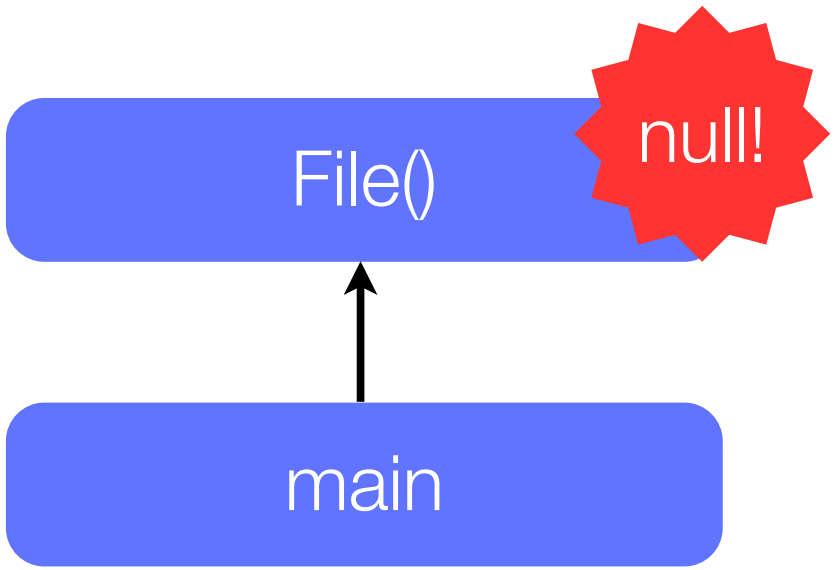
From the Web Server example

```
public static void main(String[] args) {
    try {
        new SimpleWebServer(new File(null), 8000);
    }
    catch (Exception e) {
        System.out.println(e);
    }
}
```



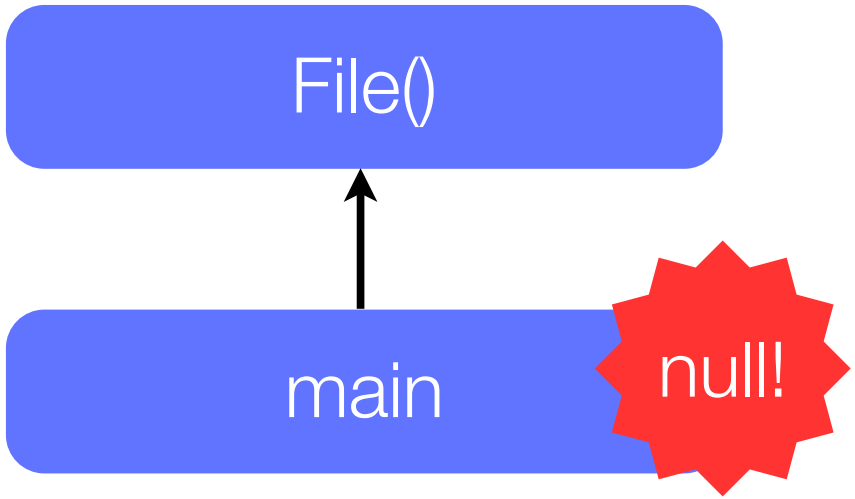
From the Web Server example

```
public static void main(String[] args) {
    try {
        new SimpleWebServer(new File(null), 8000);
    }
    catch (Exception e) {
        System.out.println(e);
    }
}
```



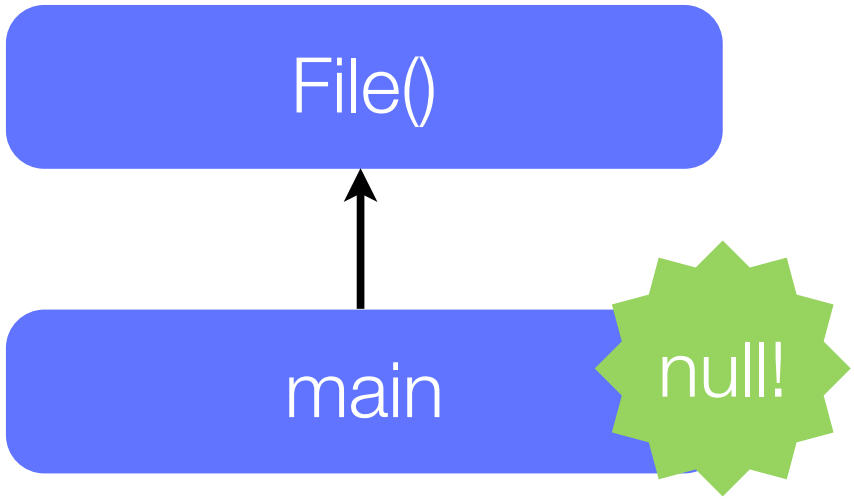
From the Web Server example

```
public static void main(String[] args) {
    try {
        new SimpleWebServer(new File(null), 8000);
    }
    catch (Exception e) {
        System.out.println(e);
    }
}
```



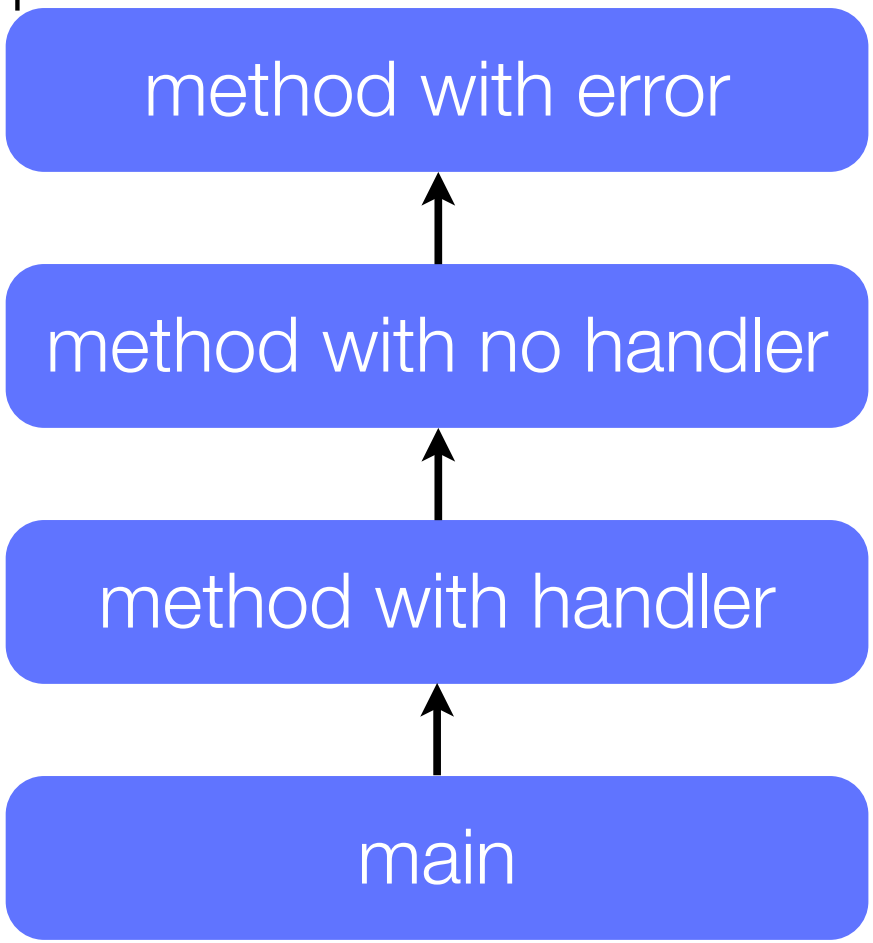
From the Web Server example

```
public static void main(String[] args) {
    try {
        new SimpleWebServer(new File(null), 8000);
    }
    catch (Exception e) {
        System.out.println(e);
    }
}
```



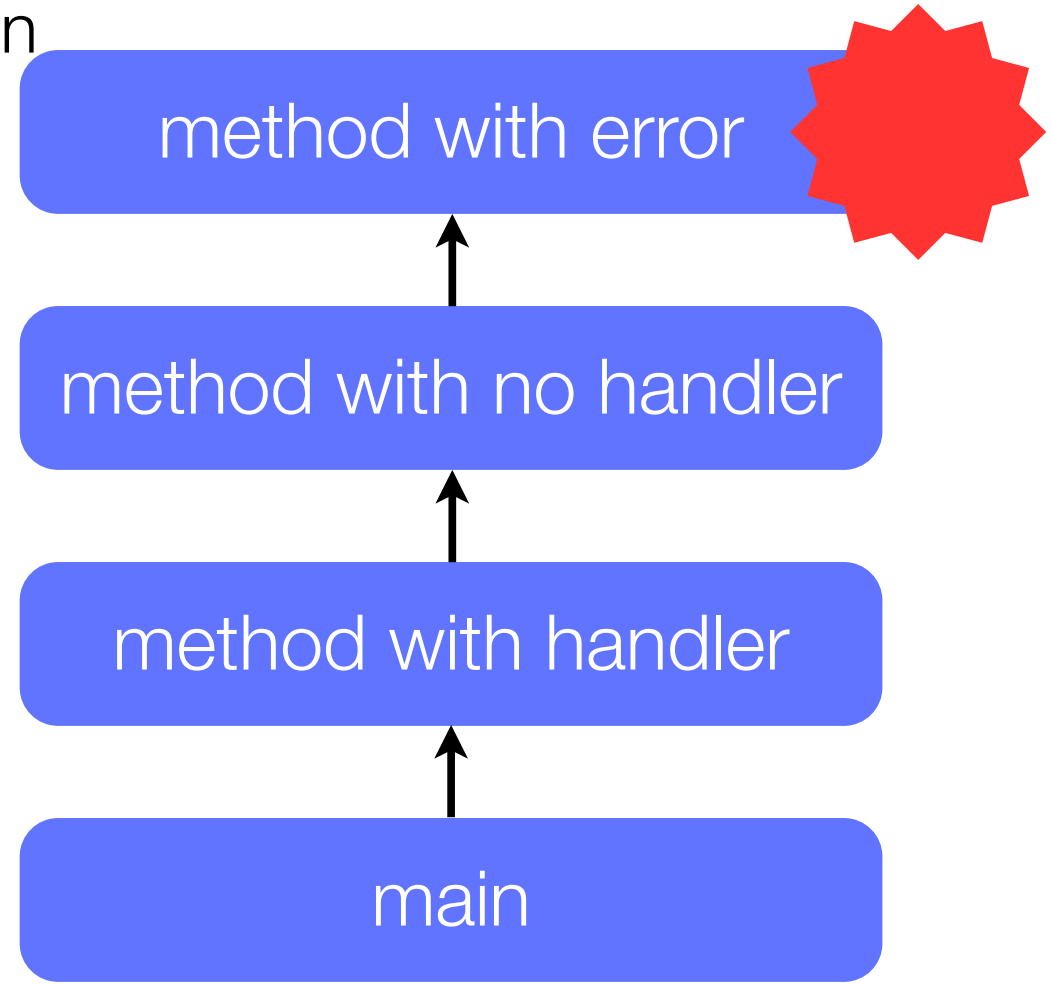
Searching the call stack

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



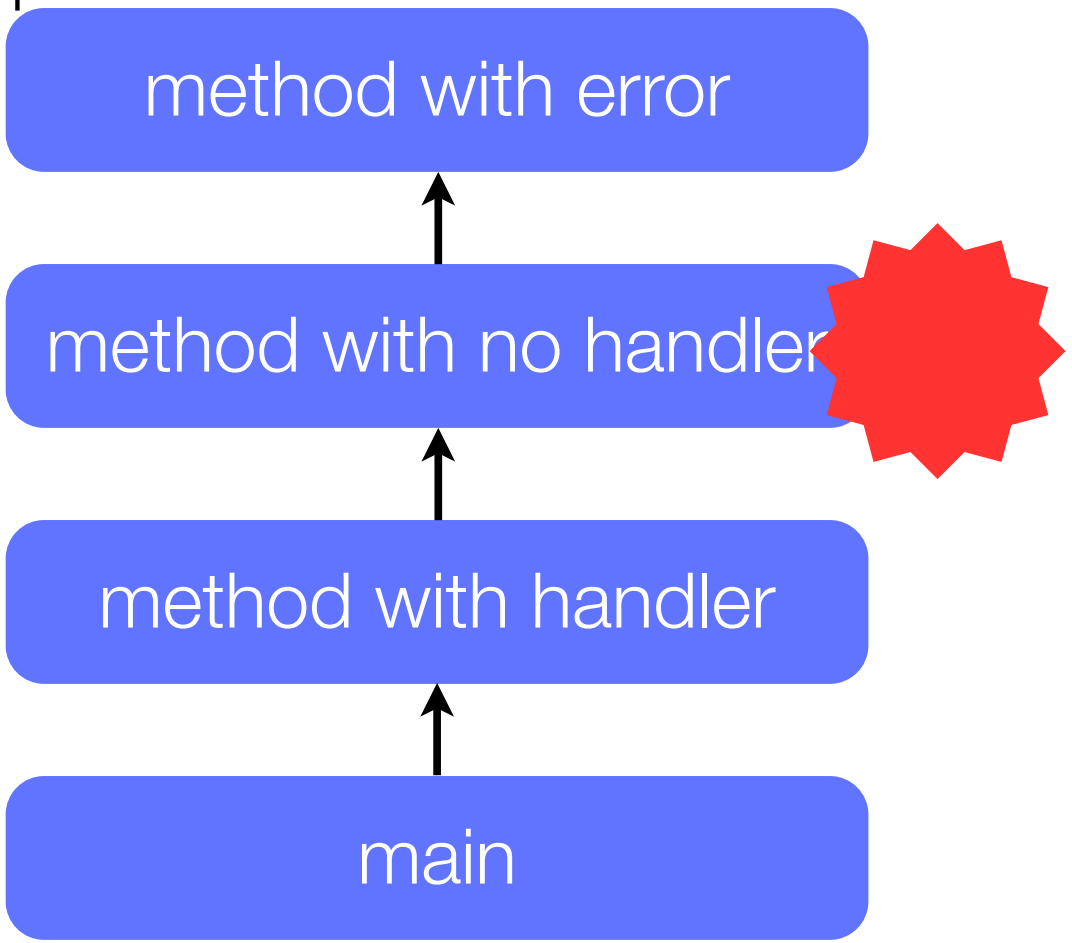
Searching the call stack

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



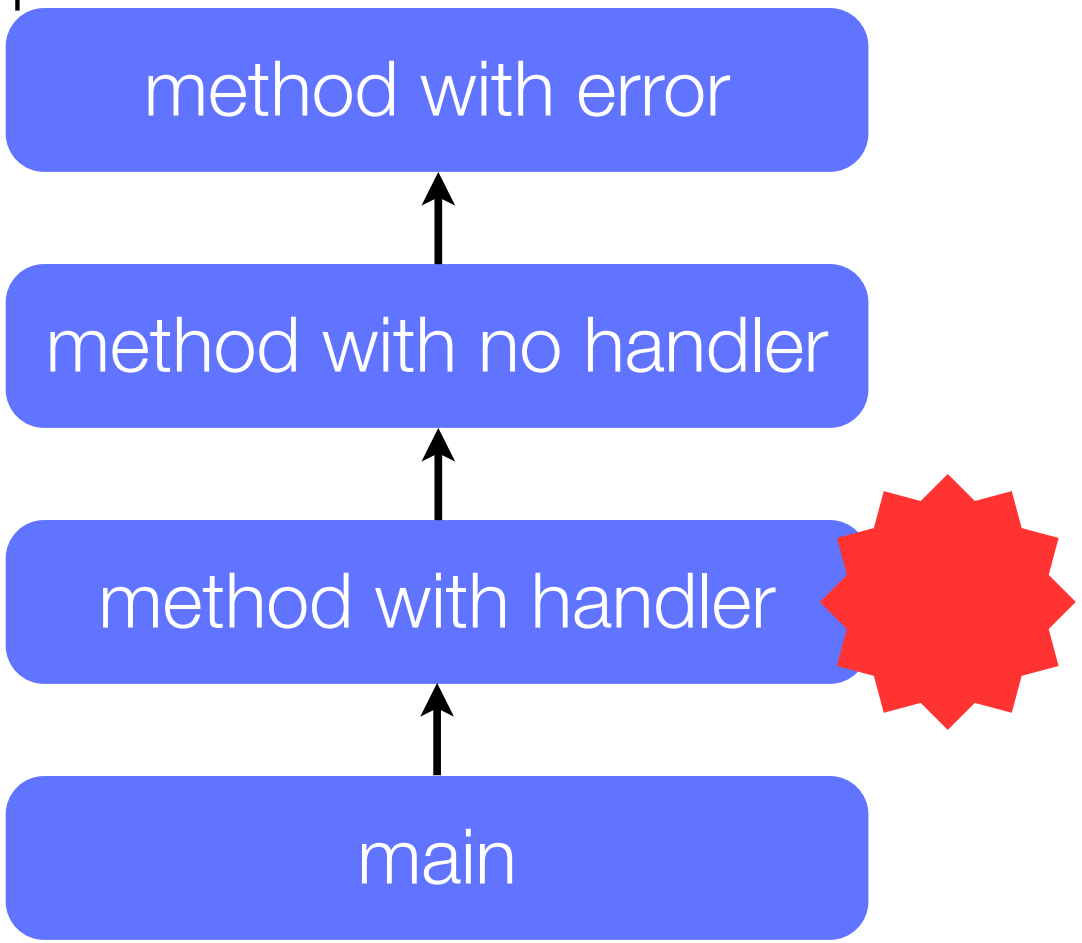
Searching the call stack

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



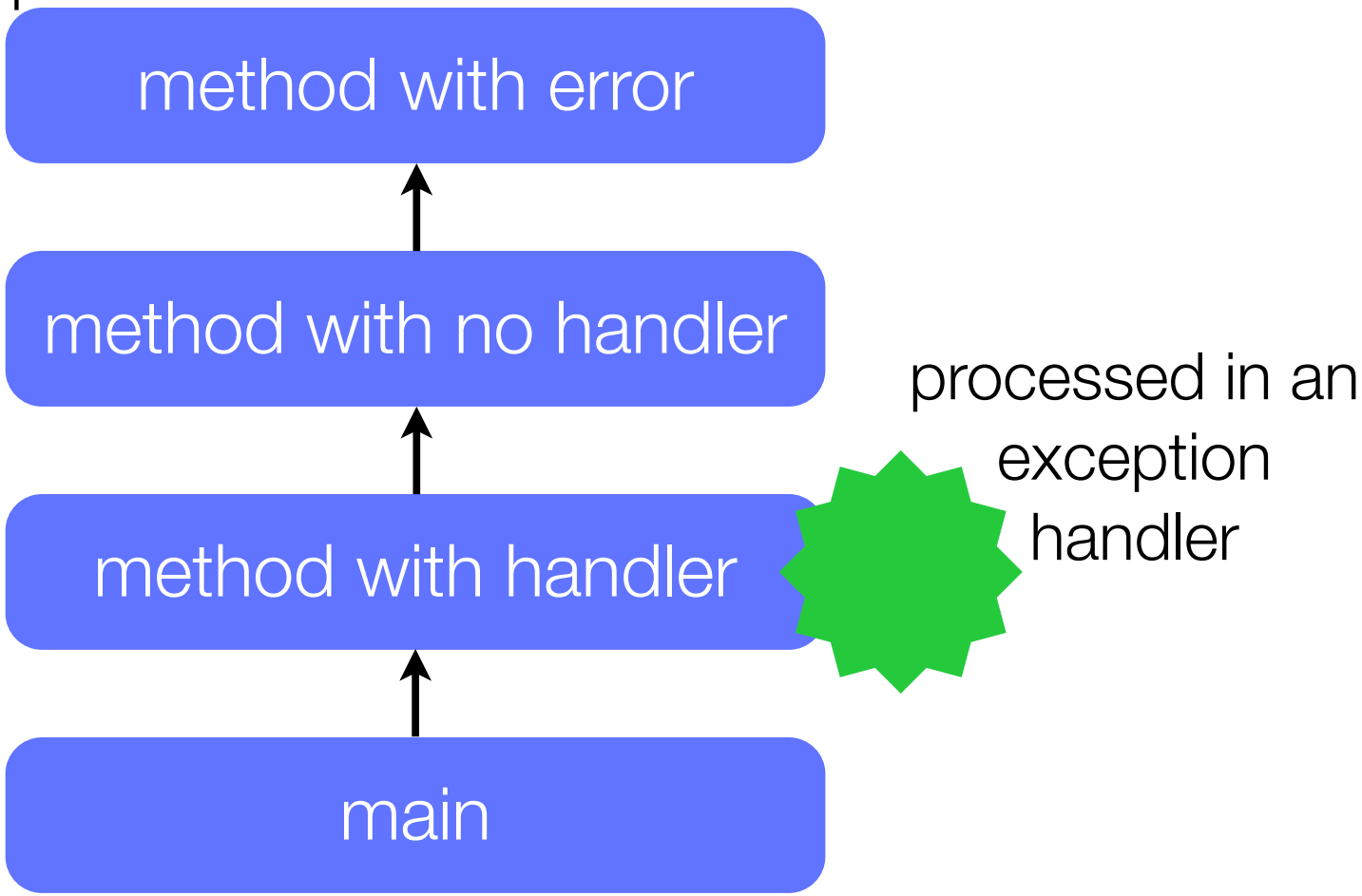
Searching the call stack

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



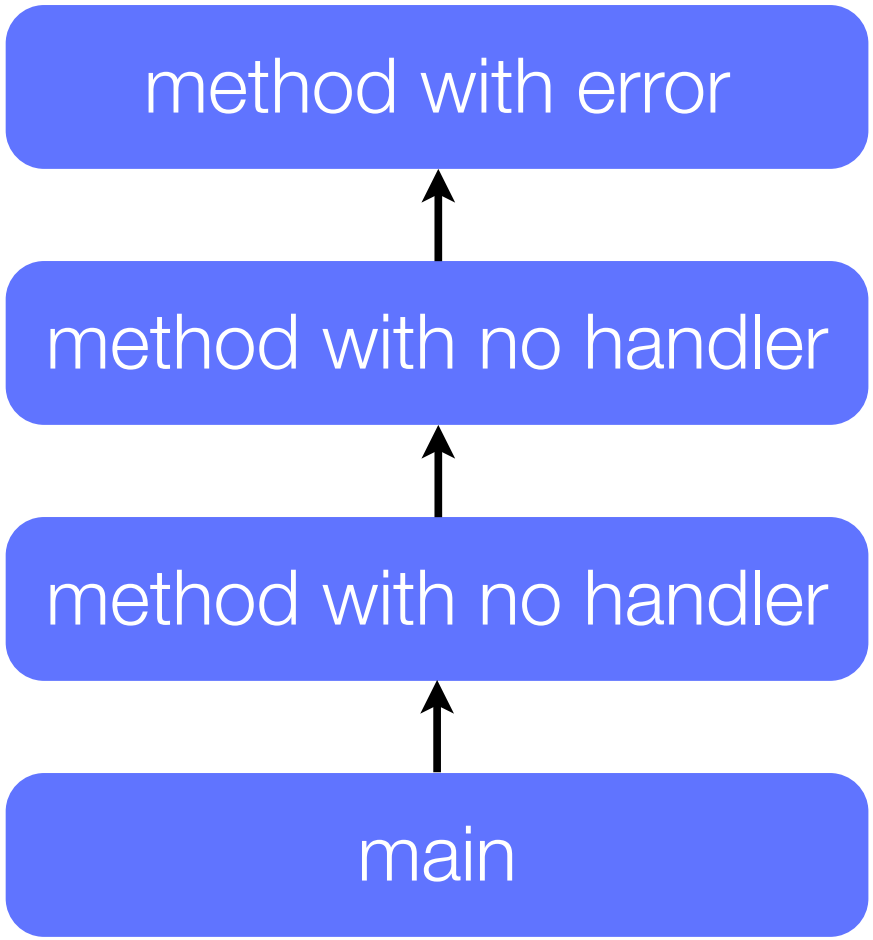
Searching the call stack

This block of code that can handle an exception is called an *exception handler*

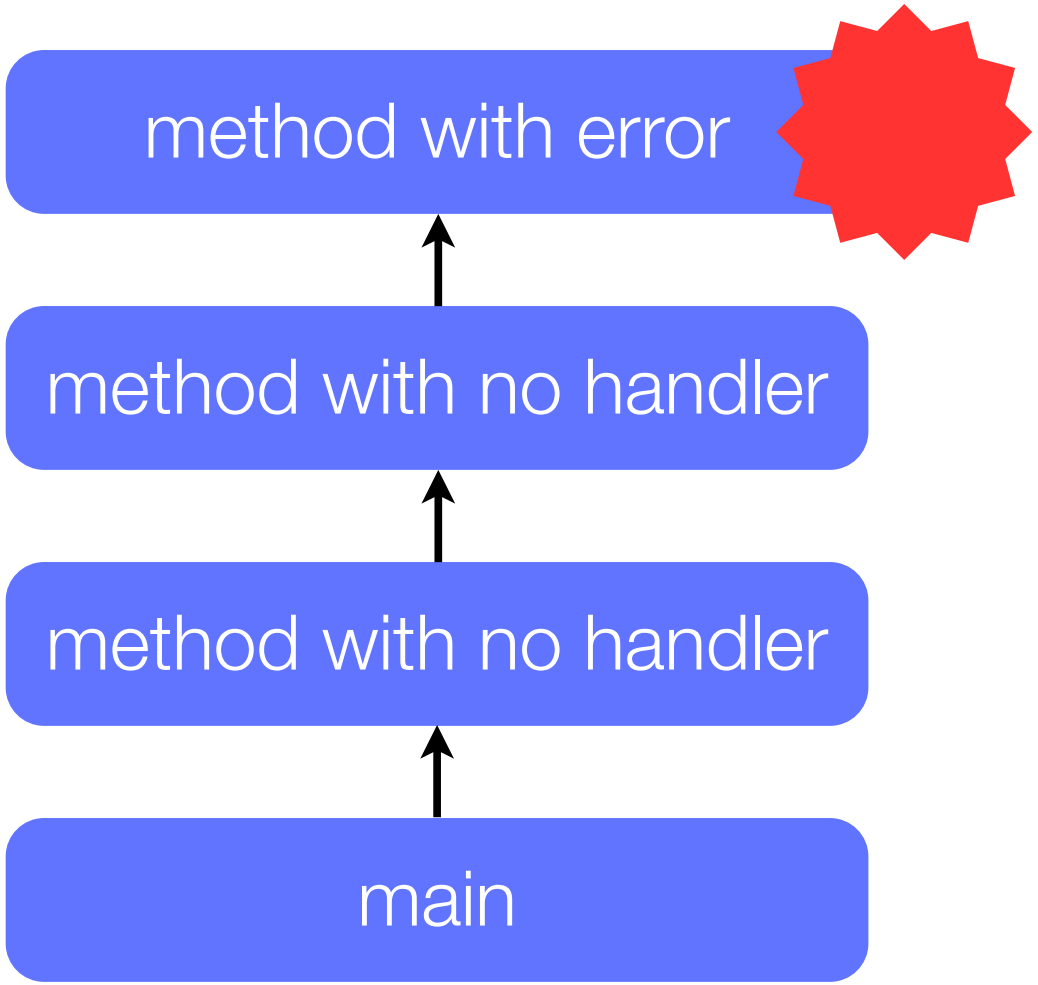
The search begins with the method in which the error occurred and proceeds through the call stack in the reverse order in which the methods were called

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

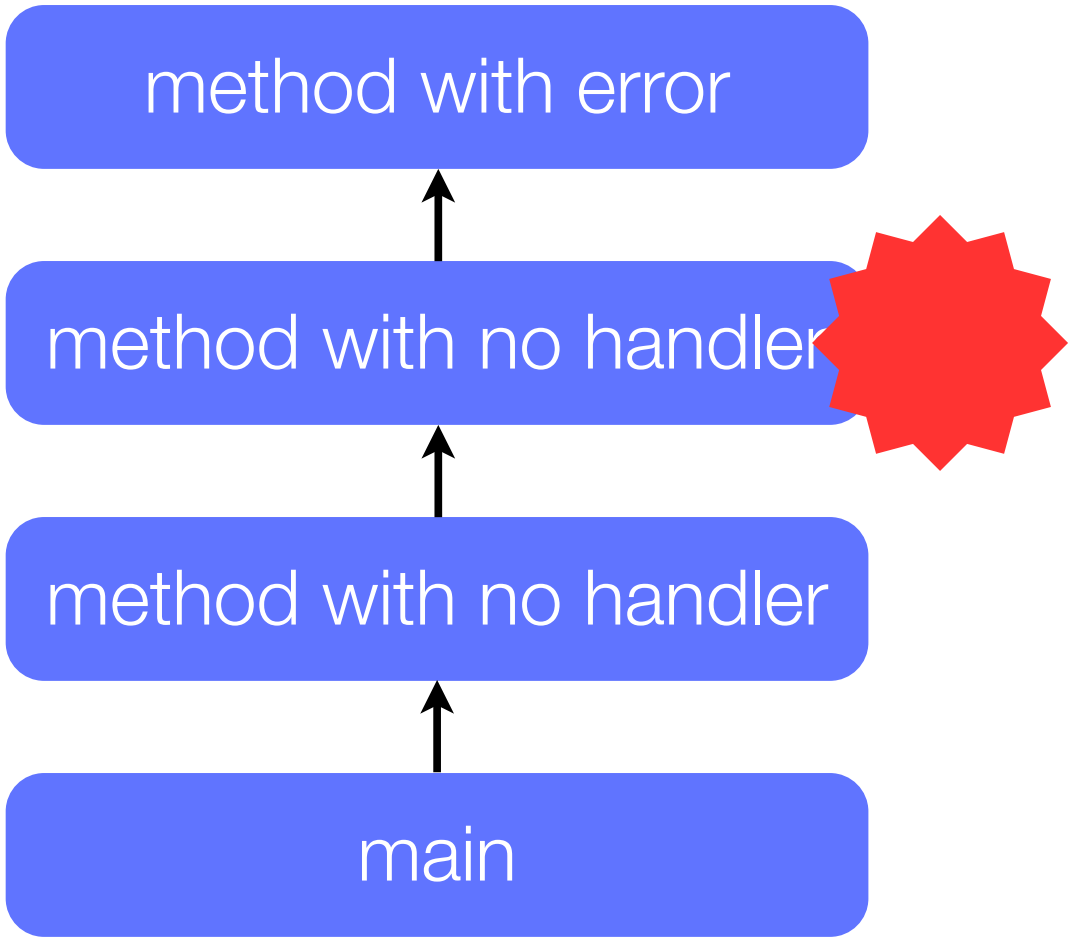
And if there is no handler?



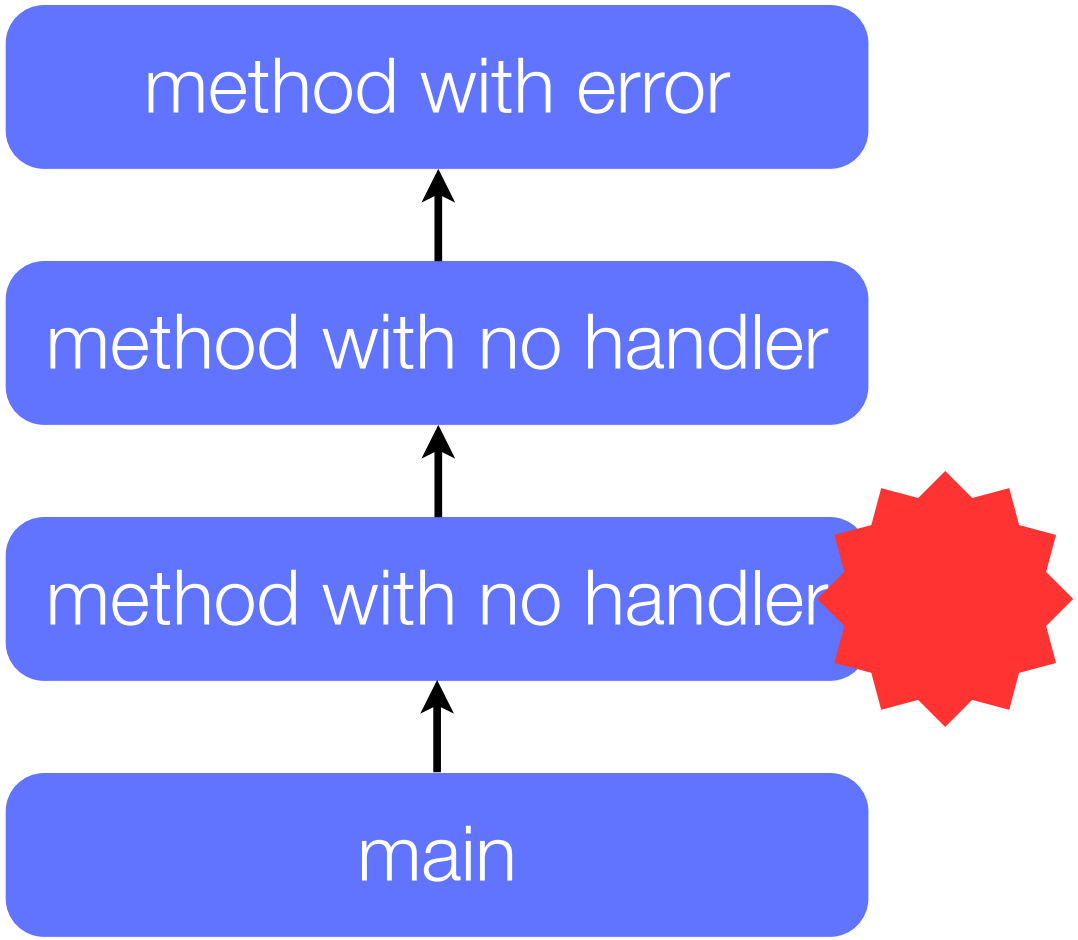
And if there is no handler?



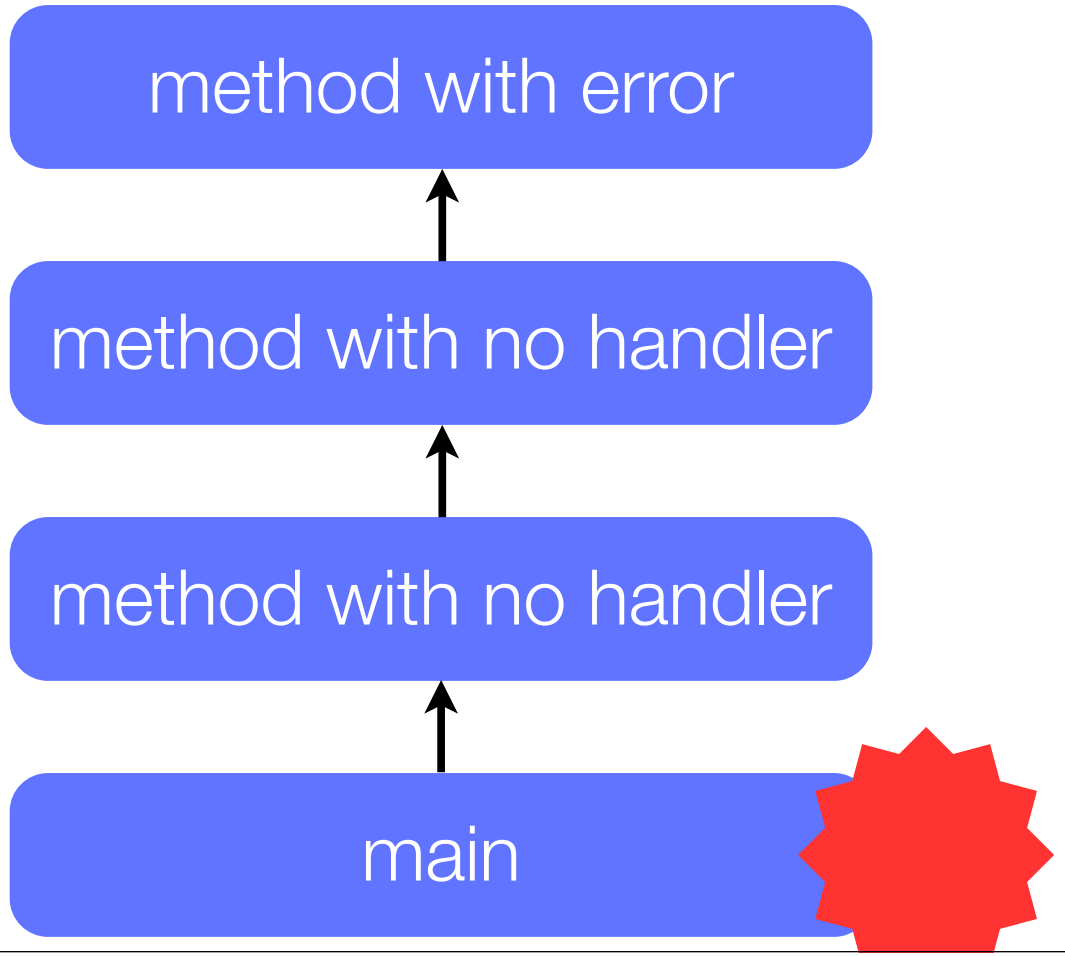
And if there is no handler?



And if there is no handler?

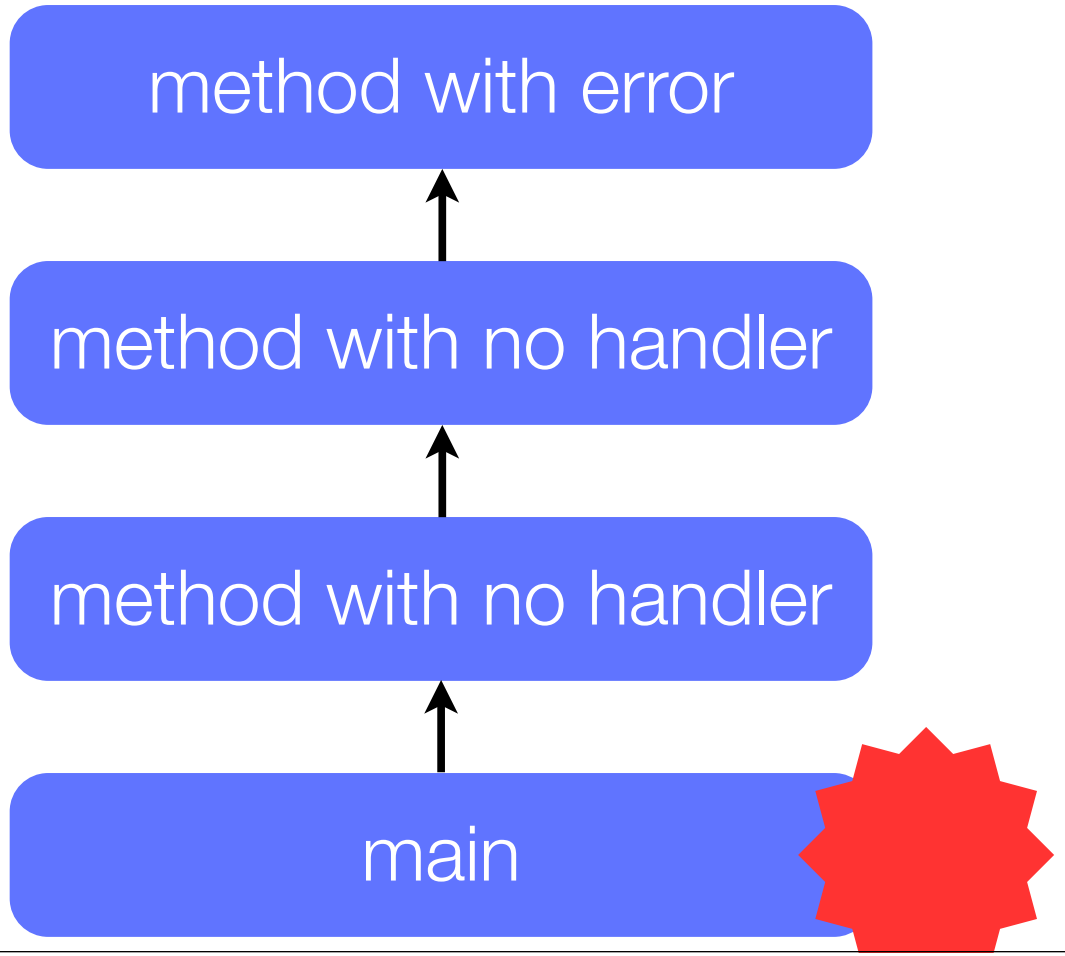


And if there is no handler?



And if there is no handler?

The program terminates



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*.

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

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

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.

In the Web server application

```
public SimpleWebServer(File rootDir, int port) throws IOException
{
    _rootDir = rootDir.getCanonicalFile();
    if (!_rootDir.isDirectory()) {
        throw new IOException("Not a directory.");
    }
    _serverSocket = new ServerSocket(port);
    start();
}
```

In the Thread class

```
public class Thread implements Runnable {
   ...

  public static native void sleep(long millis)

     throws InterruptedException;

    ...

 }
```

In the Thread class

```
public class Thread implements Runnable {
   ...

  public static native void sleep(long millis)

     throws InterruptedException;

    ...
 }
```

The native keyword informs the Java compiler that the implementation for this method is provided in another programming language

Catching and Handling

A try block looks like the following:

```
try {
    code that could throw an exception
}
catch and finally blocks ...
```

```
public static void main(String[] args) {
     try {
          new SimpleWebServer(new File("..."), 8000);
     }
     ...
 }
```

Catching and Handling

If an exception occurs within the try block, that exception is handled by an exception handler associated with it

```
try {
  ...
} catch (ExceptionType name) {
   ...
}
```

Catching and Handling

If an exception occurs within the try block, that exception is handled by an exception handler associated with it

```
try {
  ...
} catch (ExceptionType name) {
  ...
}
```

```
public static void main(String[] args) {
    try {
        new SimpleWebServer(new File("..."), 8000);
    }
    catch (IOException e) {
        System.out.println(e);
    }
}
```

Catching and Handling

More than one catch is possible

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

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

Specifying the Exceptions Thrown by a Method

Sometimes, it's appropriate for code to catch exceptions that can occur within it

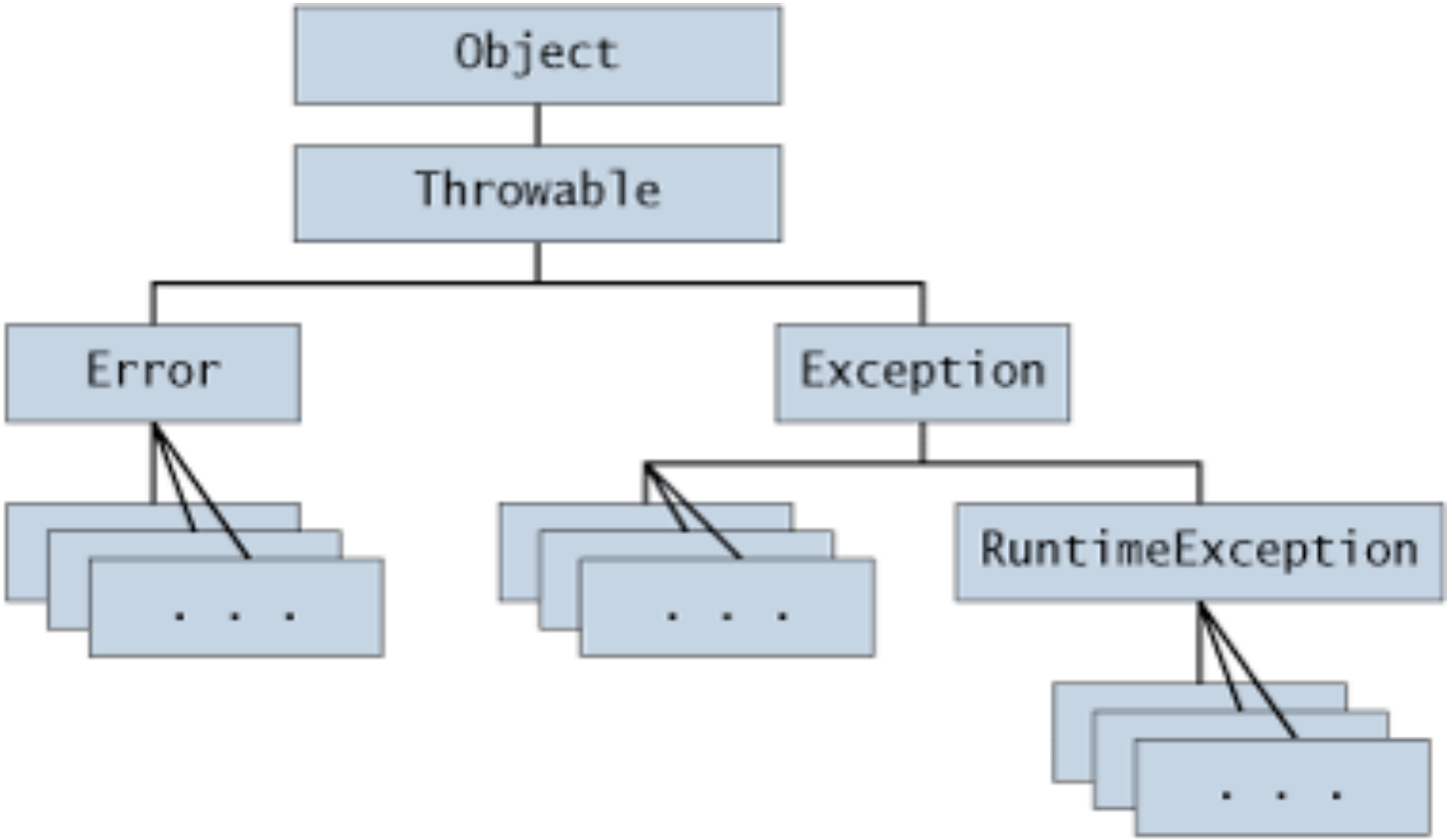
In other cases, however, it's better to let a method further up the call stack handle the exception

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

```
public SimpleWebServer(File rootDir, int port) throws IOException {
    _rootDir = rootDir.getCanonicalFile();
    if (!_rootDir.isDirectory()) {
        throw new IOException("Not a directory.");
    }
    _serverSocket = new ServerSocket(port);
    start();
}
```


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

Conclusion

Software errors need to be managed using Exceptions

Different kinds of exception

The exception mechanism may be abused

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