Exception Handling

Java Developer

StayAhead Training; May, 2022

Exceptions

Exception Handling

- An exception is an error that occurs within your code
- Some exceptions can be anticipated, some cannot
- Each exception is accompanied by an object that contains information about what went wrong (or at least it should)
- An exception is thrown from method to method until it is handled or it reaches the JVM at which point your app will crash
- Exceptions are not something to be avoided - they help you to write good quality, robust code

Checked vs. Unchecked Exceptions

Exception Handling

- A checked exception:
  - results from an environmental error, e.g. server down, file moved, bad input
  - can be anticipated
  - must be handled
- An unchecked exception:
  - results from bad code
  - should **not** be handled (the code should be fixed instead)

Checked vs. Unchecked Exceptions

Exception Handling

- Checked exception, e.g.

```
var text = Files.readString(Path.of("my file.txt"));
```

- The readString method will throw a checked exception if the file doesn't exist
- The error is anticipatable
- The compiler will force us to handle it
- By handling it, we're making our code better/more robust

Checked vs. Unchecked Exceptions

Exception Handling

- Unchecked exception, e.g.

```
var nums = new int[] {1, 2, 3};
var num3 = nums[3];
```

- The attempt to read from array index 3 will throw an unchecked exception
- The error is the result of bad code
- The compiler will **not** force us to handle it*
- *The compiler does not check for unchecked exceptions, hence the name

The Call Stack

Exception Handling

- The stack is the part of Java memory that stores methods in execution and their local variables
- The first method to be pushed onto the stack is main
- The main method is likely to call some other method which is in turn pushed onto the stack and so on
- When a method ends it is popped off the stack
- An app is terminated when the main method is popped off the stack

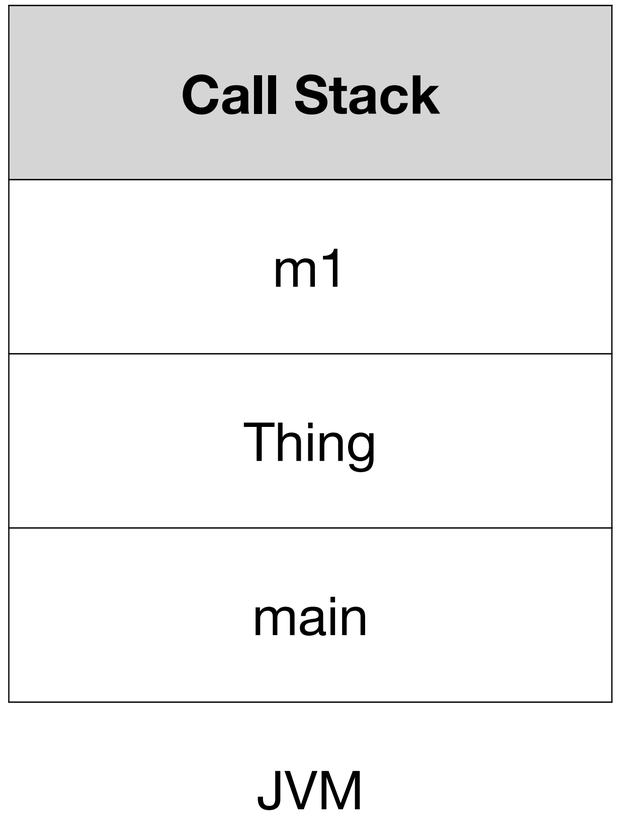
The Call Stack

Exception Handling

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

Thing() {
   m1();
}

void m1() {
   ...
}
```



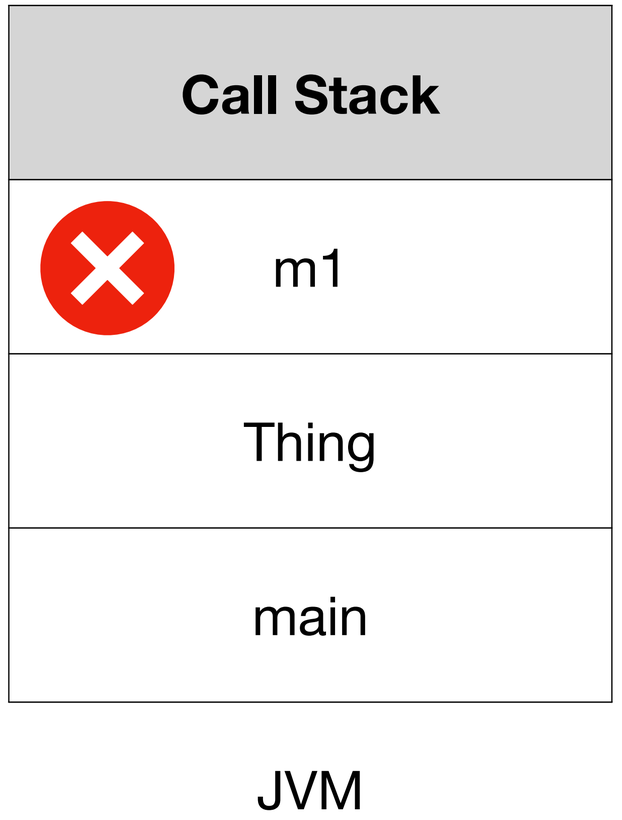
The Call Stack

Exception Handling

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

Thing() {
   m1();
}

void m1() {
   ... // exception occurs
}
```



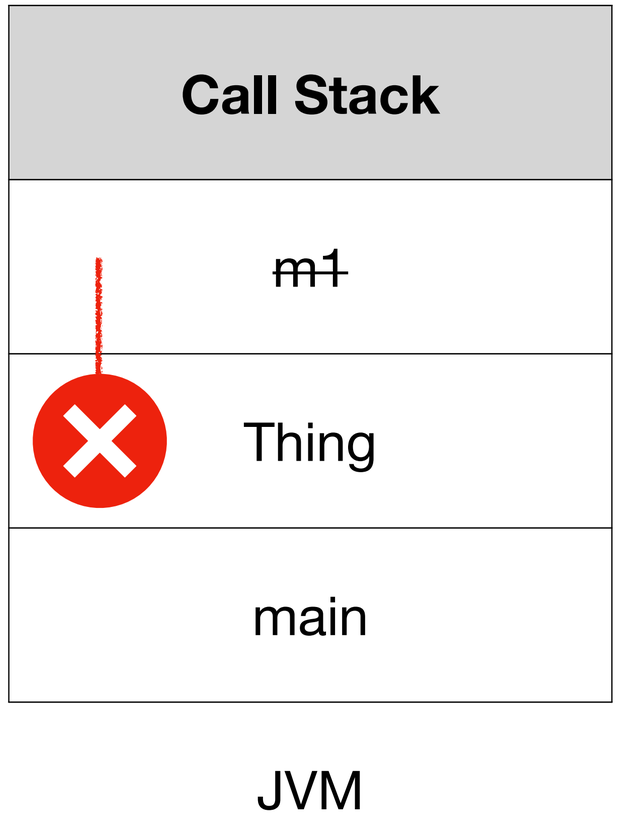
The Call Stack

Exception Handling

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

Thing() {
   m1();
}

void m1() {
   ... // exception occurs; is not handled
}
```



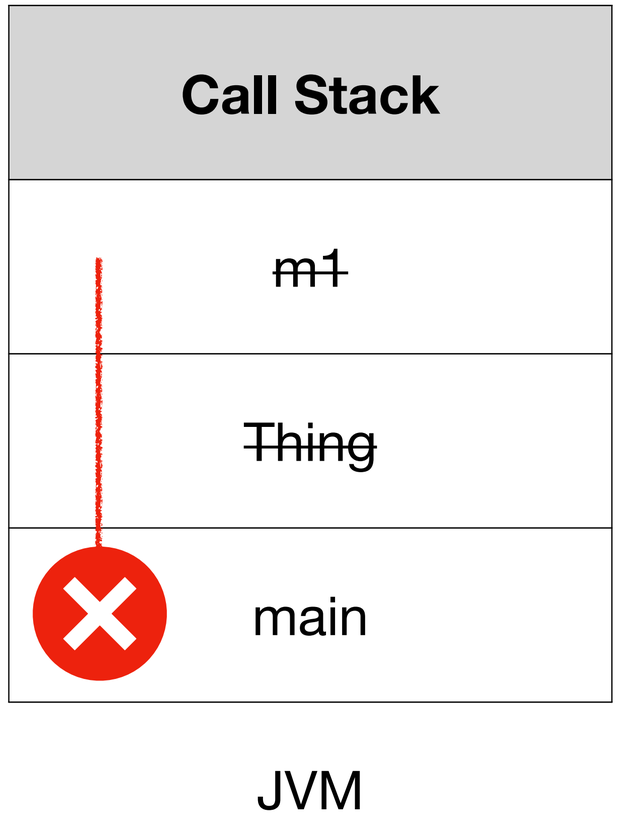
The Call Stack

Exception Handling

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

Thing() {
   m1(); // not handled here either
}

void m1() {
   ... // exception occurs; is not handled
}
```



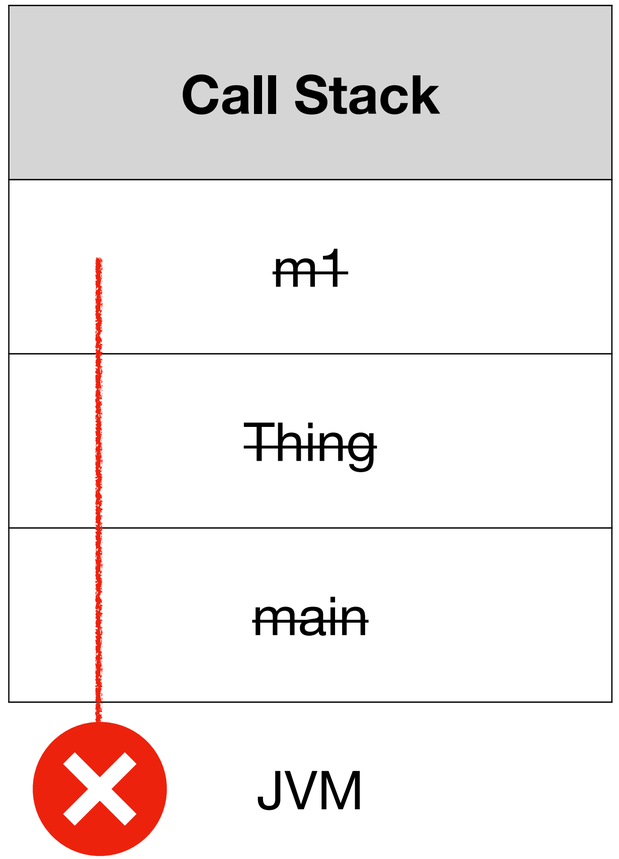
The Call Stack

Exception Handling

```
public static void main(String[] args) {
  new Thing(); // still not handled
}

Thing() {
  m1(); // not handled here either
}

void m1() {
  ... // exception occurs; is not handled
}
```



The Call Stack

Exception Handling

- One of the methods on the call stack **must** handle the exception
- But which one?
- An exception should be handled by whichever method is best able to do so
- For example, if the exception occurs because of bad user input, then the method that prompts the user for said input is best placed to handle it - it can prompt the user to try again

try and catch
Exception Handling

- A try catch block is the mechanism used to handle exceptions
- A try block wraps some code that may throw a checked exception
- The associated catch block is executed if and when the exception is thrown
- If an exception is thrown control passes immediately to the catch block (remaining statements in the try block are skipped)
- The catch block has access to the Exception object
- Execution of the catch block does **not** cause the method to be terminated

try and catch

Exception Handling

- Example 1: no exception is thrown

```
try {
   var text = Files.readString(Path.of("myfile.txt"));
   System.out.println(text);
} catch (NoSuchFileException e) {
   // exception handling code
}
// additional method code
```

try and catch
Exception Handling

- Example 2: an exception is thrown

```
try {
   var text = Files.readString(Path.of("myfile.txt"));
   System.out.println(text);
} catch (NoSuchFileException e) {
   // exception handling code
}
// additional method code
```

try and catch

Exception Handling

- The catch block is passed the exception object, e.g.

```
catch (NoSuchFileException e) {
  // exception handling code
}
```

- The code inside the parentheses is effectively a variable declaration
- e is the variable that references an object of type NoSuchFileException

Exception Objects*

Exception Handling

- Courtesy of inheritance (of which more later) every exception object has certain methods, including:
  - **printStackTrace** writes the stack trace to stderr (like stdout)
  - **getMessage** returns a String that may describe the error
- NB: simply writing the stack trace or exception message to stdout does not qualify as handling the exception - something has gone wrong and your app is unlikely to work properly if you don't do something about it

Stack Trace*

Exception Handling

- A stack trace is a list of methods through which the exception passed before it was handled (or not), e.g.

```
java.lang.Exception
  at Thing.m1 (Thing.java:8)
  at Thing.<init> (Thing.java:4)
  at App.main (App.java:5)
```

- NB: the stack trace mirrors the call stack and includes filenames and line nos
- The stack trace is intended to help the developer identify the point at which the error occurred - it is not intended for consumption by the end user

finally

Exception Handling

- The finally block:
  - is optional
  - comes after the catch block(s)
  - will be executed regardless of whether an exception is thrown
- Historically, resources like file handles and DB connections, would be closed inside the finally block
- The finally block has been mostly superseded by try with resources

finally

Exception Handling

- Example 1: no exception is thrown

```
try {
   var text = Files.readString(Path.of("myfile.txt"));
   System.out.println(text);
} catch (NoSuchFileException e) {
   // exception handling code
} finally {
   // code to be executed exception or no
}
// additional method code
```

finally

Exception Handling

- Example 2: an exception is thrown

```
try {
   var text = Files.readString(Path.of("myfile.txt"));
   System.out.println(text);
} catch (NoSuchFileException e) {
   // exception handling code
} finally {
   // code to be executed exception or no
}
// additional method code
```

Handling Multiple Exception Types

Exception Handling

- Some methods may throw one of several different types of exception depending on what goes wrong
- For example, attempting to read from a file may fail because:
    A. the file does not exist (NoSuchFileException)
    B. the user does not have permission to read it (IOException)
- We can either handle multiple exception types in the same way, or handle each one differently

Handling Multiple Exception Types

Exception Handling

- Handling multiple exception types in the same way, e.g.

```
try {
  var text = Files.readString(Path.of("myfile.txt"));
  System.out.println(text);
} catch (NoSuchFileException | IOException e) {
  // code to handle either exception
}
// additional method code
```

Handling Multiple Exception Types

Exception Handling

- Handling each exception type in a different way, e.g.

```
try {
  var text = Files.readString(Path.of("myfile.txt"));
  System.out.println(text);
} catch (NoSuchFileException e) {
  // code to handle NoSuchFileException
} catch (IOException e) {
  // code to handle IOException (permission error)
}
// additional method code
```

try with resources

Exception Handling

- try with resources is a try block in which a resource, e.g. a file reader/writer, is opened and which will be closed automatically regardless of whether an exception is thrown, e.g.

```
// the FileReader instance will be closed automatically
try (var reader = new FileReader("myfile.txt")) {
  // TODO
} catch (IOException e) {
  // exception handling code
}
```

throw and throws

Exception Handling

- An exception should be handled by whichever method is best able to do so
- If a method cannot handle the exception it **must** throw it to the caller, e.g.

```
void writeToStdout(Path path) throws Exception {
  var text = Files.readString(path);
  System.out.println(text);
}
```

- NB: don't throw exceptions from the main method

throw and throws

Exception Handling

- Returning a success/failure indicator from a method is bad practice, e.g.

```
boolean doSomething() {
  // TODO
  // the method was unsuccessful
  return false;
}
```

- This requires the caller (and the caller's caller etc.) to code an if statement
- It also obfuscates the return value, which should represent the result of the method's work

throw and throws

Exception Handling

- If a method might not succeed in its task it should throw an exception, e.g.

```
void doSomething() throws Exception {
  // TODO
  // the method was unsuccessful
  throw new Exception("error msg");
}
```

- This requires the caller (and the caller's caller etc.) to either handle the exception (with a try catch block) or to throw it

Custom Exceptions

Exception Handling

- The java.lang.Exception class is the parent to all exception types
- As a rule, Exception type exceptions should not be thrown because they are too generic (the name, Exception, doesn't mean anything)
- We can exploit inheritance to create our own custom exception types, e.g.

```
class InvalidCredentialsException extends Exception {}
```

- The class need not have any fields, constructors, or methods - it's the name that provides the meaning