Exceptions

CPSC 1181 – O.O.

Jeremy Hilliker

Summer 2017

Langara.
THE COLLEGE OF HIGHER LEARNING.

Overview

- Exceptions
- Language construct
- How to use
- Advantages

- Akn: Lesson: Exceptions (The Java™ Tutorials > Essential Classes)
- https://docs.oracle.com/javase/tutorial/essential/exceptions

Program Flow

- So far, methods of flow control (procedural):
  - Linear execution
  - Conditional Branches (if, else)
  - Loops
  - Switch / Case
  - Method calls / return
    - Object-oriented
    - IoC
    - Callbacks
    - Event-driven

Exceptions

- The term “exception” is shorthand for “exceptional event”
- An ***exception*** is an event that disrupts the program’s normal execution flow.

- Alternative: error checking

Error Checking

- Check for some error condition after method calls with potential errors.

```c
#include <stdlib.h>

int main() {
    FILE * fp; // a "file pointer"

    fp = fopen ("file.txt", "w+");
    // What if this failed?  Returns null.
    // Note: this was not checked

    fprintf(fp, "%s %s %s %d", "We", "are", "in", 2012);
    // ^^ Segmentation Fault -> program dies a fiery death

    fclose(fp);
    return(0);
}
```

```
// A
if ((fp = fopen(file, "r")) == NULL)
    return;

// B
#include <stdio.h>
int main () {
  FILE * pFile;
  pFile = fopen ("myfile.txt","w");
  if (pFile!=NULL)
  {
    fputs ("fopen example",pFile);
    fclose (pFile);
  }
  return 0;
}

// C
#include <cstdio>
#include <cstdlib>
int main() {
    FILE* fp = std::fopen("test.txt", "r");
    if(!fp) {
        std::perror("File opening failed");
        return EXIT_FAILURE;
    }

    int c; // note: int, not char, required to handle EOF
    while ((c = std::fgetc(fp)) != EOF) { // standard C I/O file reading loop
       std::putchar(c);
    }

    if (std::ferror(fp))
        std::puts("I/O error when reading");
    else if (std::feof(fp))
        std::puts("End of file reached successfully");

    std::fclose(fp);
}
```

Error Checking Problem

- Often forgotten
- Sometimes you don't know what to check
- Have to explicitly check. Every. Single. Time.
  - P.S.: don't ever forget!
- What if we cant reasonably handle the error here?
  - IE: this was part of a much larger operation
  - Must explicitly propagate the error through the call stack with a special return value…
    - That has to be checked at. Every. Single. Level.
    - P.S.: don't every forget!
    - We'll lose info about where the error occurred as it propagates up the call stack

Result

- Instead of programming for success:

```
x.doAmazingThings();
```

- We always have to be programming for failure:

```
if(!x.doAmazingThings()) return ERR_NOT_AMAZING;
```

- Always.
- Always.
- Don’t ever forget!

- What if the language
  - enforces error handling consideration for some errors,
  - and provides an automatic propagation mechanism

Exceptions

- The term “exception” is shorthand for “exceptional event” / “exceptional circumstance”
- An ***exception*** is an event that disrupts the program’s normal execution flow.

- Execution flow is immediately transferred to an exception handler

- We say that a method ***throws*** an ***exception***
- And that an exception handler ***catches*** an exception

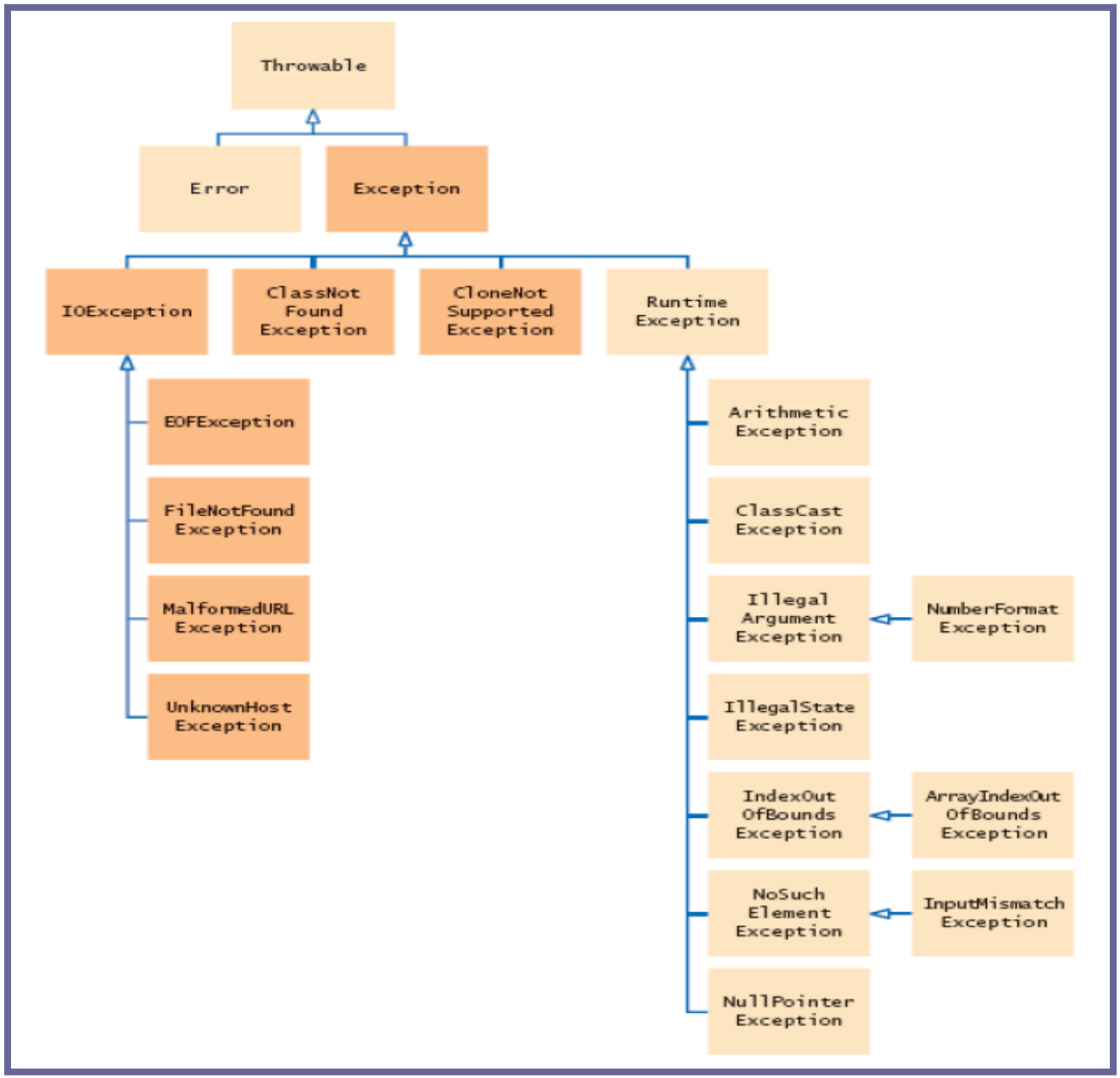
Ex: Writing to a file

```
 2  public void save(File f, byte[] data) {
 3      // throws FileNotFoundException
 4      OutputStream out = new FileOutputStream(file);
 5      // throws IOException
 6      out.write(data);
 7      // throws IOException
 8      out.close();
 9
10      // none of this will compile!
11  }
```

```
Exceptionzz.java:7: error: unreported exception FileNotFoundException; must be caught or declared to be thrown
                OutputStream out = new FileOutputStream(f);
                                   ^
Exceptionzz.java:9: error: unreported exception IOException; must be caught or declared to be thrown
                out.write(data);
                          ^
Exceptionzz.java:11: error: unreported exception IOException; must be caught or declared to be thrown
                out.close();
                          ^
3 errors
```

Three types of Exceptions

- Error
  - A horrible thing that should never* be recovered from
  - Typically comes from the VM or the machine
- Checked Exception
  - Something that the compiler forces you to consider
    - Like our examples on previous slide
- Unchecked Exception
  - Typically the result of the programmer's error
  - Not enforced by compiler
    - IllegalArgumentException, NullPointerException, ArrayIndexOutOfBoundsException, etc.


Throwable
Error
Exception
IOException
ClassNot Found Exception
CloneNot Supported Exception
Runtime Exception
EOFException
FileNotFound Exception
MalformedURL Exception
UnknownHost Exception
Arithmetic Exception
ClassCast Exception
Illegal Argument Exception
NumberFormat Exception
IllegalState Exception
IndexOut OfBounds Exception
ArrayIndexOut OfBounds Exception
NoSuch Element Exception
InputMismatch Exception
NullPointer Exception

Ex: Writing to a file

```
 2  public void save(File f, byte[] data) {
 3      // throws FileNotFoundException
 4      OutputStream out = new FileOutputStream(file);
 5      // throws IOException
 6      out.write(data);
 7      // throws IOException
 8      out.close();
 9
10      // none of this will compile!
11  }
```

```
Exceptionzz.java:7: error: unreported exception FileNotFoundException; must be caught or declared to be thrown
                OutputStream out = new FileOutputStream(f);
                                   ^
Exceptionzz.java:9: error: unreported exception IOException; must be caught or declared to be thrown
                out.write(data);
                          ^
Exceptionzz.java:11: error: unreported exception IOException; must be caught or declared to be thrown
                out.close();
                          ^
3 errors
```

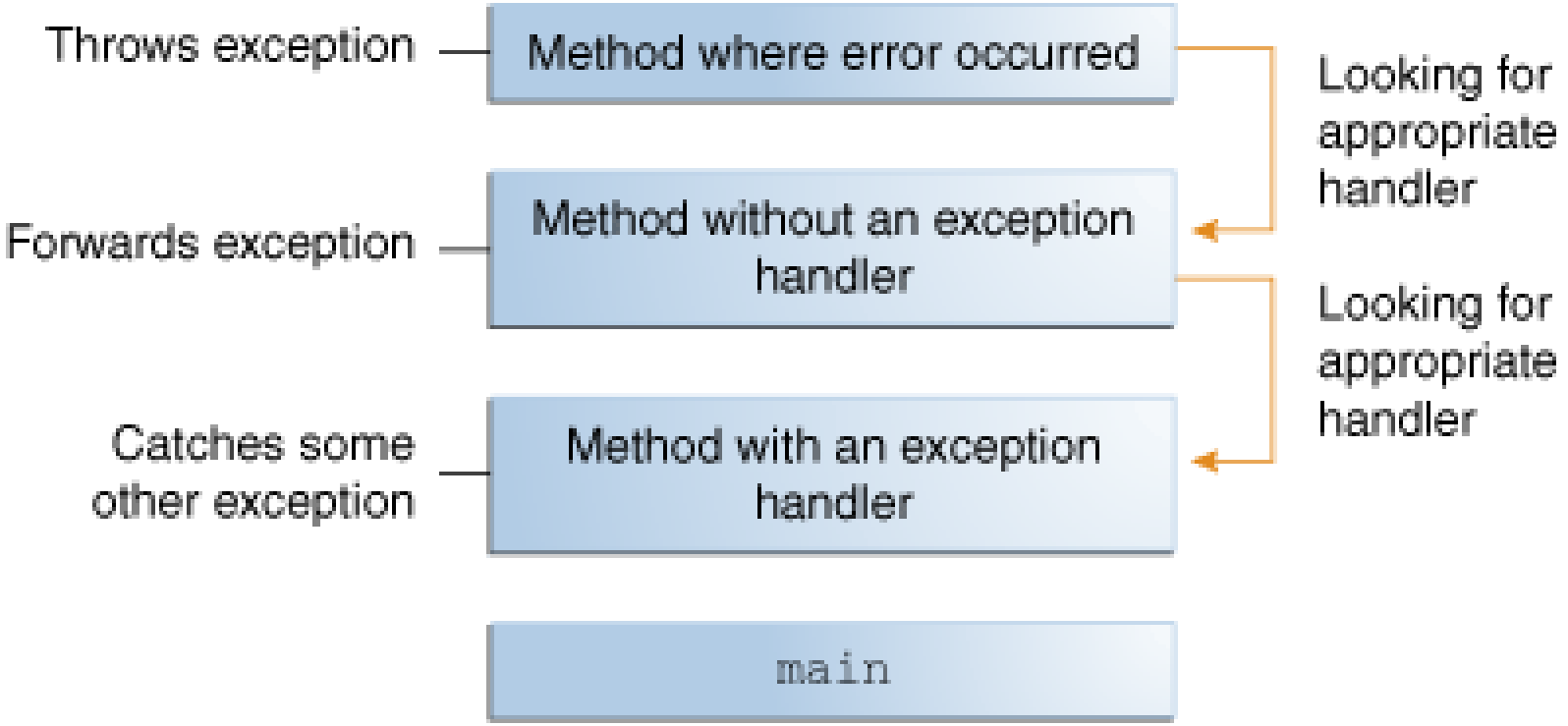
Checked Exception

1. Catch and handle the exception ourselves

2. Let it propagate
   - Up the call stack to the next exception handler

- If we choose 2),
  - because the exception is checked,
  - we must tell the compiler that:
    - We are aware that we did not handle the error
    - And we have to declare that we throw an exception

Exception Propagation



2) Propagate Exception

```
12      public void save(File f, byte[] data) throws IOException {
13          // throws FileNotFoundException
14          OutputStream out = new FileOutputStream(f);
15          // throws IOException
16          out.write(data);
17          //throws IOException
18          out.close();
19      }
```

- Probably best choice
  - Our caller will want to know if a “save” failed

- There’s a bug
  - If the .write() call throws an exception
    - Don’t reach the .close() line
    - Leaves the stream open

try-finally

```
12      public void save(File f, byte[] data) throws IOException {
13          try {
14              // throws FileNotFoundException
15              OutputStream out = new FileOutputStream(f);
16              // throws IOException
17              out.write(data);
18          } finally {
19              //throws IOException
20              out.close();
21          }
22      }
```

- “finally” block is always executed
  - Can throw its own exceptions too
- Pattern: the thing that opens something is responsible for closing it

try-with-resources

```
12      public void save(File f, byte[] data) throws IOException {
13          // can be done with all things that are-a "java.lang.AutoCloseable"
14          // including all things that are-a "java.io.Closeable"
15          try (OutputStream out = new FileOutputStream(f)) {
16              // throws IOException
17              out.write(data);
18          }
19          // note: no finally
20      }
```

```
Exception in thread "main" java.io.FileNotFoundException: \notReal\nope.txt
        at java.io.FileOutputStream.open0(Native Method)
        at java.io.FileOutputStream.open(FileOutputStream.java:270)
        at java.io.FileOutputStream.<init>(FileOutputStream.java:213)
        at java.io.FileOutputStream.<init>(FileOutputStream.java:162)
        at Exceptionzz.save(Exceptionzz.java:15)
        at Exceptionzz.main(Exceptionzz.java:9)
```

1) Catch and Handle

```
3  public class Exceptionzz {
4
5      public static void main(String[] args) {
6          File f = new File("/notReal/nope.txt");
7          byte[] data = new byte[4096];
8
9          new Exceptionzz().save(f, data);
10     }
11
12     public void save(File f, byte[] data) {
13         try {
14             // throws FileNotFoundException
15             OutputStream out = new FileOutputStream(f);
16             // throws IOException
17             out.write(data);
18             // throws IOException
19             out.close();
20         } catch (IOException e) {
21             System.err.println("Exception writing to file: " + e);
22         }
23     }
24 }
```

```
Exception writing to file: java.io.FileNotFoundException: \notReal\nope.txt (The system cannot find the path specified)
```

- Program flow resumes normally after exception is caught

Comments

- Probably not a good idea to catch here
  - Our caller probably wants to know if the save failed

- There's a bug
  - If we failed *while* writing, we left the stream open
  - The close() line is never reached

try-catch-finally

```
12      public void save(File f, byte[] data) {
13          OutputStream out = null;
14          try {
15              // throws FileNotFoundException
16              out = new FileOutputStream(f);
17              // throws IOException
18              out.write(data);
19          } catch (IOException e) {
20              System.err.println("Exception writing to file: " + e);
21          } finally {
22              if(out != null) {
23                  try {
24                      // throws IOException
25                      out.close();
26                  } catch (IOException e) {
27                      System.err.println("Exception closing file: " + e);
28                  }
29              }
30          }
31      }
```

- Contents of finally block always executed

try-with-resources

```
12      public void save(File f, byte[] data) {
13          // can be done with all things that are-a "java.lang.AutoCloseable"
14          // including all things that are-a "java.io.Closeable"
15          try (OutputStream out = new FileOutputStream(f)) {
16              // throws IOException
17              out.write(data);
18          } catch (IOException e) {
19              System.err.println("IOEXception while writing: " + e);
20          }
21          // note: no finally
22      }
```

- Program flow resumes normally after exception is caught

Notes:

- Can have multiple resources:

```
try (
    java.util.zip.ZipFile zf =
         new java.util.zip.ZipFile(zipFileName);
    java.io.BufferedWriter writer =
        java.nio.file.Files.newBufferedWriter(outputFilePath, charset)
) {
```

- A method can throw multiple exceptions
  - foo() throws Exception1, Exception2, Exception3

- You can have multiple catch block
  - They are evaluated in order
  - The first one that matches is chosen
  - If none match, the exception propagates

```
18          } catch (LoveException e) {
19              // I'm checked first
20          } catch (IOException e) {
21              // I'm only checked if the one before me didnt match
22          } finally {
23              // I always run, exception ot not1
24          }
```

- New in Java 7: catch more than one exception type with one handler

```
22          } catch (NullPointerException | ArrayIndexOutOfBoundsException e) {
23              // I can catch multiple types
24          } finally {
```

Throwing an Exception

- You can throw your own exceptions
- They’re just objects
  - Make one
  - And throw it

Ex: IllegalArgumentException

- “Thrown to indicate that a method has been passed an illegal or inappropriate argument.”

```
throw new IllegalArgumentException();
```

```
2  public class BankAccount {
3      //...
4      public void withdraw(int amount) {
5          if(amount > balance) {
6              throw new IllegalArgumentException(
7              "Withdrawl of " + amount +
8              " exceeds the balance of " + balance);
9          }
10
11         balance -= amount;
12         assert balance >= 0;
13     }
14 }
```

- Note: no changes to method signature.
- IllegalArgumentException is an unchecked exception

Making your own Exceptions

- Exception is just a class
  - You can extend it, or any of its subclasses
- Extend something that is-a `RuntimeException` to get an unchecked exception
- Extend something that is-not-a `RuntimeException` to get a checked exception
- Don’t extend Error or Throwable
  - Don’t catch them either…

Ex: Creating InsufficientFundsException

```
1  public class InsufficientFundsException
2      extends RuntimeException {
3
4      public InsufficientFundsException(String msg) {
5          super(msg);
6      }
7  }
```

```
10 public class BankAccount {
11     //..
12     public void witdraw(int amount) {
13         if (amount > balance) {
14             throw new InsufficientFundsException("...");
15         }
16
17         balance -= amount;
18     }
19 }
```

Chained Exceptions

- You can respond to an exception by throwing a different exception
- It's very helpful to know what the original exception was

```
29      public void doAwesome() throws NotAwesomeException {
30          File f = getAwesomeFile();
31          try {
32              // throws FileNotFoundException
33              InputStream in = new BufferedInputStream(new FileInputStream(f));
34              // ...
35          } catch (FileNotFoundException e) {
36              throw new NotAwesomeException("Bummer", e);
37          }
38      }
```

Advantages 1)

- Separate error-handling from normal code

```
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;
}
```

```
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;
    }
}
```

Advantages 2)

- **Propagating Errors Up the Call Stack**

```
method1 {
    call method2;
}

method2 {
    call method3;
}

method3 {
    call readFile;
}
```

```
method1 {
    errorCodeType error;
    error = call method2;
    if (error)
        doErrorProcessing;
    else
        proceed;
}

errorCodeType method2 {
    errorCodeType error;
    error = call method3;
    if (error)
        return error;
    else
        proceed;
}

errorCodeType method3 {
    errorCodeType error;
    error = call readFile;
    if (error)
        return error;
    else
        proceed;
}
```

```
method1 {
    try {
        call method2;
    } catch (exception e) {
        doErrorProcessing;
    }
}

method2 throws exception {
    call method3;
}

method3 throws exception {
    call readFile;
}
```

Advantages 3)

- Grouping error types
- FileNotFoundException extends IOException
  - Both can be caught by one catch block
  - Q: what's its type?

Recap

- Exceptions
  - Intro
  - Motivation
- Three types
  - Error
  - Checked
  - Unchecked
- Propagation
- Language constructs
  - Method -> throws()
  - try-finally
  - try-catch
  - try-catch-finally
  - try-with-resources
  - throw e;
- Extending
- Chaining
- Advantages