Topic 9: Exception Handling

Reading: JBD Section 11.1
Exceptions

- An exception is an unusual condition that may arise during execution of a program.
- In Java, exceptions are represented by objects of type Exception or Error.
- Terminology:
  - To throw an exception means to interrupt normal execution and to generate an object that describes the exception.
  - To catch an exception means to handle the unusual condition by taking some action based on the information in the thrown object.
The Class Hierarchy of Throwable Objects

```
 Throwable
  `-- Exception
  `-- Error
     `-- (Specific kinds of exceptions)
     `-- (Specific kinds of errors)
```
Throwable Objects

- All throwable objects contain a "message" field
- All throwable objects implement these methods:
  - String getMessage()
    returns the message associated with this exception
  - String toString()
    returns a printable description of the exception
  - void printStackTrace()
    prints the stack of active methods at the point where the
    exception occurred
  - etc. (see Java documentation)
Default Exception Handling

- When an exception is encountered, execution halts and Java prints out a stack trace.
- The exception is "caught" by the Java interpreter and "handled" by calling `printStackTrace()`.
Writing your own exception handlers

- If a section of code might throw an exception, you can embed it in a "try-catch" statement:

```java
try {
    // code that might raise an ArithmeticException
    // or an InputMismatchException
}
catch(ArithmeticException e1) {
    // code that handles an ArithmeticException
}
catch(InputMismatchException e2) {
    // code that handles an InputMismatchException
}
finally {
    // code that is executed last of all, even if there are no exceptions
}
```
Semantics of "try-catch"

- A try-block may have zero, one, or more catch-blocks
- The finally-block is optional (unless there's no catch)
- When an exception occurs in a try-block, execution is interrupted and an exception object is thrown
- Java searches "upward" for the nearest catch-block that handles that exception class (or a superclass)
  - Could be in an enclosing block, or in a calling method
  - If found, that catch-block is executed
  - If not found, the program exits with a stack trace
- Execution then resumes after the end of the try-catch statement that handled the exception (unless the catch-block throws another exception)
What does a catch-block know?

- It knows the type of exception(s) that it handles
  
  ```java
  catch(IOException e) { ... }
  ```

- It can call the methods of the Exception object to find out more
  
  ```java
  e.getMessage()
  ```
The "finally" clause

```java
try { ... }
catch(This e) { ... }
catch(That e) { ... }
finally { ... }
```

- If a finally-clause is present:
  - If there are no exceptions, the finally-clause is executed after the try-block
  - If an exception is handled locally, the finally-clause is executed after the catch-clause
  - If an exception is handled at a higher level, the finally-clause is executed before the higher-level catch-clause
  - The finally-clause usually contains "cleanup" code (closes files, etc.)
  - The only way to circumvent a finally-clause is to call `System.exit()`
Declaring what a method "throws"

- Certain kinds of exceptions are *checked exceptions*
- Examples (indentation denotes subclasses)
  ```java
  java.io.IOException
    EOFException
    FileNotFoundException
  ```
- If your method might raise a checked exception, then it must:
  - Declare "`throws` ExceptionClass" in the function header
  - OR provide a handler for the checked exception (then it is no longer thrown)
- Methods do not need a `throws` clause for unchecked exceptions (*NullPointerException*, etc.)
- If you call a method that throws an exception, you must either handle it or declare that you throw it
Examples of Exceptions

- Indentation denotes subclasses

- Checked exceptions
  - `java.io.IOException`
  - `EOFException`
  - `FileNotFoundException`

- Unchecked exceptions
  - `java.lang.RuntimeException`
  - `NoSuchElementException`
  - `NullPointerException`
  - `IndexOutOfBoundsException`
  - `ArithmeticException`
  - `InputMismatchException`
  - `java.lang.OutOfMemoryError`
  - `java.lang.AssertionError`
Example of Default Exception Handling

```java
public static void main(String[] args)
    throws FileNotFoundException
{
    Scanner scan =
        new Scanner(new File("deductions.txt"));
    double myDeductions = scan.nextDouble();
    System.out.println("My deductions are "+myDeductions);
}
```

- If Java can't find the `deductions.txt` file,
  - The program terminates
  - You see a `FileNotFoundException` and a stack trace
Example of Explicit Exception Handling

```java
public static void main(String[] args) {
    double myDeductions;
    try {
        Scanner scan =
            new Scanner(new File("deductions.txt"));
        myDeductions = scan.nextDouble();
    }
    catch (FileNotFoundException e) {
        System.out.println("No deductions file");
        System.out.println("Using std deduction");
        myDeductions = 10000;
    }
    System.out.println("My deductions are " + myDeductions);
}
```

"throws" not needed
Another Example: Input Validation

- This code forces the user to enter a valid `double`

```java
double number = 0;
Scanner scan = new Scanner(System.in);
System.out.println("Type a number and hit enter");
boolean stillTrying = true;
while (stillTrying) {
    try {
        number = scan.nextDouble();
        stillTrying = false;
    }
    catch (InputMismatchException e) {
        String throwAway = scan.next();
        System.out.println("Invalid, try again");
    }
}
System.out.println("Your number is "+ number);
```
Scope of Variables

- Variables declared in a try-block or catch-block are local to that block

- If you want a variable to be seen in both a try-block and a catch-block, declare it global to both:

```java
int x;
try {
    x = 5 / 0;
}
catch(ArithmeticException e) {
    x = 99;
}
```
Throwing Your Own Exceptions

- You can make up your own types of exceptions

```java
class BoxTooBigException extends Exception {
    Box box;      // instance variable
    BoxTooBigException(Box b) {   // constructor
        box = b;
    }
}
```

- You can throw them whenever you like

```java
if (b.size > THRESHOLD)
    throw new BoxTooBigException(b);
```
Handling Your Own Exceptions

- You can write a handler for your new exception-type

```java
try { ... }
    catch(BoxTooBigException e) {
        System.out.println("Box " + e.box.name
                        + " is too big");
        System.out.println("Size reduced to "+ THRESHOLD);
        e.box.shrink(THRESHOLD);
    }
```

- Why not just deal with the situation inline?
  - You may want to interrupt processing and handle the exception at a higher level in the call stack
Assertions: a useful kind of exception

- Example of an assert statement:
  ```java
  assert reactor.state == State.SAFE : reactor;
  ```

- Equivalent:
  ```java
  if (!(reactor.state == State.SAFE))
      throw new AssertionError(reactor);
  ```

- This invokes the constructor `AssertionError(Object)`

- By default, the program terminates with an error message: `AssertionError: Reactor@1712193` and a stack trace

- You can write a handler that catches `AssertionErrors` and prints out diagnostics on the failing reactor