Java Exceptions

- One example of a deep inheritance hierarchy in the Java Class Library is the set of classes for **Exceptions**
- An Exception is a runtime error
- We generally see three kinds of errors in programming:
  1. Syntax Errors
  2. Logical Errors
  3. Runtime Errors

Motivation for Java Exceptions

- Exceptions provide graceful way to handle (**catch**) errors as a program is executing.
- They provide a means to specify distinct error-handlers over a wide range of code.
- They provide standard way by which to generate (**throw**) an error state.
- They represent fixable errors (**exceptions**) in a program as objects
- They create an extensible **inheritance hierarchy** of exception classes for precise error handling.

Types of Program Exceptions/Errors

- Exceptions (can be handled by program):
  - I/O errors (keyboard / mouse / disk)
  - Math errors (e.g., divide by zero)
  - Network errors (internet, LAN)
  - Illegal casting, object dereferencing (null), math
  - Array / collection index out of bounds
- Errors (cannot be handled reasonably by program):
  - Computer out of memory
  - Java Virtual Machine bug / error / crash
  - Corrupt Java classes required by program

Exception Inheritance Hierarchy

- `java.lang.Throwable`
  - `Error`
    - `InternalError`
    - `StackOverflowError`
    - `VirtualMachineError`
    - `OutOfMemoryError`
    - `UnknownError`
  - `Exception`
    - `AWTException`
    - `IOException`
      - `FileNotFoundException`
      - `MalformedURLException`
      - `RemoteException`
      - `SocketException`
    - `RuntimeException`
      - `ArithmeticException`
      - `ClassCastException`
      - `IllegalArgumentException`
      - `IndexOutOfBoundsException`
      - `NullPointerException`
      - `UnsupportedOperationException`

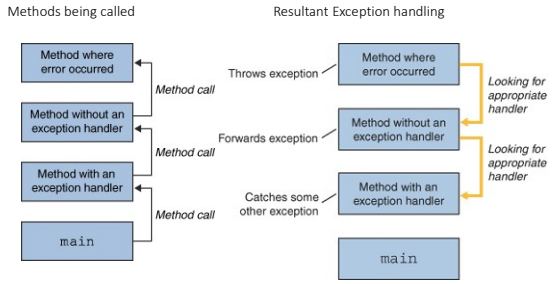
Methods of Exception

- The Exception class has several useful methods:
  - `Exception(String message)` - Constructor to associates a message with this exception that can be recovered later
  - `getMessage()` – Returns an informative string associated with a given instance of Exception
  - `printStackTrace()` – Prints the stack of method calls that led up to the point at which this instance of Exception was thrown to the System.err print stream

Checked vs. Runtime Exceptions

- Checked (not a subclass of `RuntimeException`):
  - Could have been caused by something out of your program's control
  - Must be dealt with by your code, or else the program will not compile
  - Your code must say throws … if it doesn't explicitly handle checked exceptions that could be thrown.
- Unchecked (subclass of `RuntimeException`):
  - Your fault!!
  - (Probably) could have been avoided by "looking before you leap" in your code (testing for such errors)
  - Need not be handled, but will crash program if the exception is not handled

Exceptions and the Call Stack



Throwing Runtime Exceptions

- May be thrown at any point in code, at programmer's discretion
- Need not be caught (handled) by caller

```
public Object get(int index) {
        // check argument for validity
        if (index < 0) {
                throw new IndexOutOfBoundsException("neg. index!");
        } else {
                return myStuff[index];
        }
}
```

Throwing Checked Exceptions

- Method header **must** specify all types of checked exceptions that it can throw.
- Anyone who calls the "throwing" method must now handle its exceptions, or re-throw them (pass the buck).

```
public void readFile(String fileName) throws IOException {
  if (!canRead(fileName)) {
        throw new IOException("Can't read file! ");
  } else {
        whatever();
  }
}
```

Catching (Handling) Exceptions

```
try {
        codeThatMightCrash();
}
catch (KindOfException exceptionVarName) {
        // code to deal with index exception

}
// optional:
finally {
        // code to execute after the try code,
        // or exception's catch code, has finished running
}
```

`finally`

- The `finally` block always executes when the `try` block exits.
- Allows the programmer to avoid having cleanup code accidentally bypassed by a `return`, `continue`, or `break`.
- Putting cleanup code in a `finally` block is always a good practice, even when no exceptions are anticipated.
  - Don't have to have any `catch`es to use `finally`

Cleaning Up Afterwards

```
static String readFirstLineFromFileWithFinallyBlock(String path)
                throws IOException {
    BufferedReader br = new BufferedReader(new FileReader(path));
    try {
        return br.readLine();
    } finally {
        if (br != null) br.close();
    }
}
```

Try-with-resources (Java 7+)

```
static String readFirstLineFromFile(String path)
                throws IOException {
    try (BufferedReader br = new BufferedReader(
                        new FileReader(path)))
   {
        return br.readLine();
    }
}
```

- A try-with-resources statement can have catch and finally blocks just like an ordinary try statement.
  - Any catch or finally block is run after the resources declared have been closed.

Catching Multiple Exceptions

```
try {
        codeThatMightCrash();
        moreBadCode();
} catch (IndexOutOfBoundsException ioobe) {
        // code to deal with index exception
} catch (IOException ioe) {
        // optional; code to deal with i/o exception
} catch (Exception e) {
        // optional; code to deal with any other exception
} finally {
        // optional; code to execute after the try code,
        // or exception's catch code, has finished running
}
```

Catching Multiple Exceptions

```
try{
    //call some methods that throw IOException's
} catch (FileNotFoundException e){

} catch (IOException e){

}
```

- The **first** catch-block a thrown exception matches will handle that exception

Polymorphism

- We must be careful with that ordering due to our inheritance hierarchy
  - FileNotFoundException "IsA" IOException
- By specifying the base class type, we might also accidentally (or deliberately) catch any subclasses
- The idea that a derived class can be used anywhere a base class is specified is important

Liskov Substitution Principle

- BaseClass b = new DerivedClass();
- This is legal because by extension, we promised that everything the BaseClass did, the DerivedClass does as well (no deletion, only addition)
- Thus, anywhere we see BaseClass, we can get a DerivedClass without the semantic correctness being wrong
  - Legal, but could have been a logical mistake

Catching Multiple Exceptions (Java 7+)

```
catch (IOException | SQLException ex) {
    logger.log(ex);
    throw ex;
}
```

Try/Catch Example 1

```
try {
        readFile("hardcode.txt");
} catch (IOException ioe) {
        // code could throw IOException; must catch.
        // I'll handle it by printing an error
        System.out.println("Unable to read file!");
} finally {
        // whether it succeeded or not,
        // close the file
        closeFile();
}
```

Try/Catch Example 2

```
while (true) {
        int index = kb.readInt();

        try {
                Object element = myCollection.get(index);
                break;  // if I get here, it must have worked!

        } catch (IndexOutOfBoundsException ioobe) {
                // Wouldn't have to catch this
                // since it's a runtime exception...
                System.out.print("Bad index; try again.");
         }
}
```

Good Ways to Handle Exceptions

- Print error message (`System.err.println`).
- Pop up error box (in GUI programs).
- Re-prompt user (for keyboard errors).
- Try operation again (for I/O problems).
- Fix or correct the error yourself (not always possible).
- Re-throw exception (if you shouldn't handle it, but perhaps someone else should).
- Throw more general exception (more abstract).

Where to Catch an Exception

- What is the logical definition of the error?
- When do you have the information to handle the error?

Poor Exception Handling

- Tiny try block (micro-management)
- Huge try block
- Over-general catch clause (catch Exception)
- Ineffectual catch body (e.g. {})
- Catching a runtime exception where it could have been prevented by simple checking (e.g. null, index)

Creating Your Own Exception Class

```
public class NoSignalException extends Exception {
        public NoSignalException(String message) {
                super(message);
        }

        public String toString() {
                return "Your phone is out of range:"
                + getMessage();
        }
}
```

- Note: getMessage is a method of Throwable.

Using super() constructor calls

- Just like `this` represents the current instance, `super` represents the superclass.
  - We can use it for constructors and for overriding methods by calling the base class's method
- The call to `super()` or the call to `this()` **must** be the first line of any constructor, if it is used.
  - This is true for both keywords, so you cannot combine them in the same constructor

Exception Causes

- All exceptions and errors are subclasses of Throwable.
- Often an exception is generated as a direct result of some other exception, perhaps one thrown by a lower-level API.
- Constructors of Throwable (and hence of Exception) take an optional "cause" which specifies the Throwable that caused this one.

```
try {
        readFile("hardcode.txt");
} catch (IOException ioe) {
        // I'll handle this exception by throwing another,
        // application-specific exception.
        throw new DatabaseReadException("…", ioe);
}
```

Some Points to Note

- Shouldn't catch an Error. Why?
- Exceptions occur all over the place, in
  - I/O
  - Networking / internet
  - Remote code invocation
  - Reflection
- Making methods throw checked exceptions forces them to be used more carefully.
- Cute debugging technique:
  new RuntimeException().printStackTrace();
- You can catch Exceptions in Eclipse:
  - Run -> Add Java Exception Breakpoint

To Use Exceptions, Or Not To Use Exceptions?

- Exceptions can improve readability, reliability and maintainability.
  - When used improperly, they can have the opposite effect.
- Use exceptions only for **exceptional** conditions. They should never be used for ordinary control flow.
  - In other words, avoid spaghetti code.

To Use Exceptions... (Contd.)

- Use checked exceptions for conditions from which the caller can reasonably be expected to recover.
- This example forces the caller to handle the problem in the calling code.

```
public void parseFile(File file) throws IOException, ParseException {
      if( ! formatCorrect(file)) {
            throw new ParseException ("Bad Format");
      } ...
```

To Use Exceptions... (Contd.)

- Use runtime exceptions to indicate programming (caller) errors.
- Convention dictates that errors are reserved for use by the JVM.
- In this example, we use NullPointerException which extends RuntimeException.

```
public void parseFile(File file) throws IOException, ParseException {
      if( file == null ) {
            throw new NullPointerException("null file passed.");
      } ...
```