CS 315 – Programming Languages
Exceptions

Exception Handling in PLs is an error handling technique. An Exception is an occurrence of an anomalous condition during the execution of the program. Handling the exception involves executing an exception handling procedure, which typically changes the program flow.

Before we take a deeper look at exceptions, let us consider the alternative. Note that not all PLs have exceptions (think C). The most commonly used error handling technique besides exceptions is *error codes*. These are error indicating codes returned from functions to indicate problematic situations.

Consider the following code:

```
ErrorCode res = foo();
if (res==ErrorCode.ABC) {
      …
} else if (…) {
      …
}
```

The downside is that, you cannot make use of the return variable. So, you either make the error code or the return value an out argument. Both scenarios are bad from the perspective of code clarity. An even more important limitation of error codes is that, they cannot decouple the location of the error from the location of the error handling code.

Now let us study exceptions using the Java language as an example.

```
0     …
      try {
1           …
2           call(); // assume this throws
3           …
      } catch(MatchingException e) {
4           …
      } finally {
5           …
      }
6     …
```

Execution order is: 0, 1, 2 (incomplete), 4, 5, 6
In this example, the exception was handled, so the execution continues to 6.

Let us consider another example:

```
0     …
      try {
1           …
2           call(); // assume this throws
3           …
      } catch(NonMatchingException e) {
4           …
}     } finally {
5           …
      }
6     …
```

Execution order is: 0, 1, 2 (incomplete), 5, (exceptions is thrown up to the caller)
In this example, the exception was not handled, so the control goes to the caller. Depending on whether the caller has a try/catch block, the control flow will proceed accordingly. Note that the code in finally block always gets executed.

Let us consider yet another example:

```
0       …
        try {
1               …
2               call(); // assume this throws
3               …
        } catch(MatchingException e) {
4               …
5               throw e;
}       } finally {
6               …
        }
7       …
```

Execution order is: 0, 1, 2 (incomplete), 4, 5, 6, (exceptions is thrown up to the caller)
In this example, the exception was first handled, but then thrown again. Since it is thrown, the exception was not fully handled. So again, the control goes to the caller. Depending on whether the caller has a try/catch block, the control flow will proceed accordingly. Note that the code in finally block always gets executed.

Checked and Unchecked Exceptions (Java)

If a checked exception can escape a method, then it must be listed in the exception specification of the method header (aka. the *throws clause*), otherwise a compile error is generated. The exception specifications enable the caller of a method to decide whether they want to handle the error cases (use a try/catch block) or themselves propagate it to their caller (again by listing in the exception specification).

Example:

```
// an IOException may escape this method,
// so it is listed in the throws clause
void foo(String filename) throws IOException
{
      // the below line may throw
      FileReader reader = new FileReader(new File(filename));
      ...
}
```

Alternative 1:

```
void bar() throws IOException
{
      // the below line may throw
      foo("test.txt");
}
```

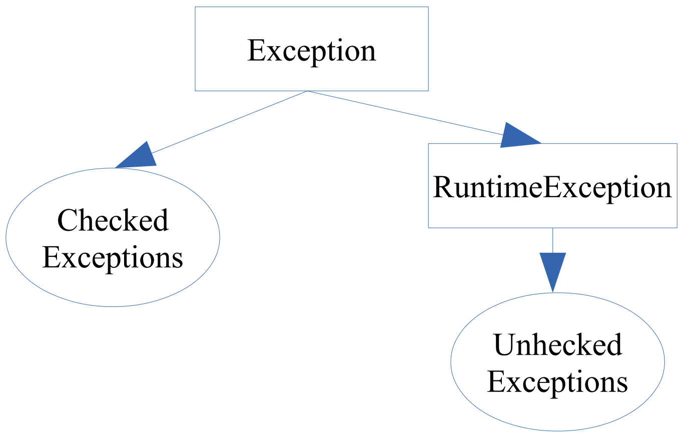
Alternative 2:

```
void bar()
{
      try {
            foo("test.txt");
      } catch(IOException) {
            … // recover
      }
}
```

- NullPtr Exception
- ArrayOutOfBoundsException
- DividedByZeroException

Unlike checked exceptions, unchecked exceptions are not listed in the throws clause clause. In other words, even if an unchecked exception may escape a method, it will not be listed in the throws clause. Unchecked exceptions are those exceptions that are prevalent. For instance, any place where there is a method call on an object, there is a possibility of getting a NullPointerException. Or anywhere where you may use array indexing, there is a possibility of getting an ArrayIndexOutOfBoundsException. If such exceptions were made checked exceptions, they would appear in so many throws clauses that it would have defeated the purpose.

In Java, exception that derive from RuntimeException are unchecked exceptions. Those derive from Exception are unchecked exceptions.



In order to catch an exception, the class used in the catch clause can be the same as the class of the exception at hand, or it could be one of its ancestors in the class hierarchy. For instance, a FileNotFoundException extends from an IOException, which extends from an Exception. As a result, an instance of the FileNotFoundException can be caught using one of these three exceptions. E.g.:

```
try {
      call(); // assume this throws FileNotFoundException
} catch(IOException e) {
      …  // catches the exception successfully
}
```

The catch clauses are considered in order:

```
try {
      call(); // assume this can throw various IOExceptions
} catch(FileNotFoundException e) {
      …  // catches FileNotFound exception
} catch(IOException e) {
      …  // catches other IOExceptions
}
```

To create your own exceptions, simplt extend from one of the existing ones:

```
public class MyException extends Exception { // a checked exception
      public MyException() { super("My exception message"); }
}

public class MyOtherException extends RuntimeException { // an unchecked exception
      public MyOtherException() { super("My other exception message"); }
}
```

Now let us consider the following exercise. We want to write a function called slurp, which reads a file's contents into a String. We want to write two versions of this, one that returns an empty string if something goes wrong with reading the file (not necessarily a good design), and one that throws an exceptions if something goes wrong with reading the file. The following methods can throw IOExceptions: FileReader constructor, readLine method, and close method. Note that we need to close the file even if something goes wrong, as otherwise an OS resource would leak.

Case 1:
```
public String slurp(String filename)
{
      StringBuilder strBuilder = new StringBuilder();
      String line = null;
      BufferedReader reader = null;
      File file = new File(file);
      try {
            reader = new BufferedReader(new FileReader(file));
            while ( (line = reader.readLine()) != null )
                  strBuilder.append(line);
      } catch(IOException e) {
            return "";
      } finally {
            if (reader != null) {
                  try {
                        reader.close();
                  } catch(IOException e) {
                        return "";
                  }
            }
      }
      return strBuilder.toString()
}
```

Case 2:
```
public String slurp(String filename) throws IOException
{
      StringBuilder strBuilder = new StringBuilder();
      String line = null;
      BufferedReader reader = null;
      File file = new File(file);
      try {
            reader = new BufferedReader(new FileReader(file));
            while ( (line = reader.readLine()) != null )
                  strBuilder.append(line);
      } finally {
            if (reader != null)
                  reader.close();
      }
      return strBuilder.toString()
}
```

Exercise: Read about Java's try-with-resource clause added in Java 7.

```
public String slurp(String filename) throws IOException
{
      StringBuilder strBuilder = new StringBuilder();
      String line = null;
      File file = new File(file);
      try (BufferedReader reader = new BufferedReader(new FileReader(file))) {
            while ( (line = reader.readLine()) != null )
                  strBuilder.append(line);
      }
      return strBuilder.toString();
}
```

A complete exercise:

```
public static void main(String[] args) {
      System.out.println(“MS”);
      level0();
      System.out.println(“ME”);
}

public static void level0() {
      System.out.println(“L0S”);
      try {
            level1();
            System.out.println(“L1M”);
      } catch(RuntimeException e) {
            System.out.println(“L0X”);
      } finally {
            System.out.println(“L0F”);
      }
      System.out.println(“L0E”);
}

public static void level1() {
      System.out.println(“L1S”);
      try {
            level2();
            System.out.println(“L1M”);
      } catch(ArithmeticException e) {
            System.out.println(“L1X”);
            throw e;
      }
      System.out.println(“L1E”);
}

public static void level2() {
      System.out.println(“L2S”);
      try {
            level3();
            System.out.println(“L2M”);
      } catch(NullPointerException e) {
            System.out.println(“L2X”);
      } finally {
            System.out.println(“L2F”);
      }
      System.out.println(“L2E”);
}

public static void level3() {
      System.out.println(“L3S”);
      int n = 3/0;
      System.out.println(n);
      System.out.println(“L3E”);
}
```

The output:

```
MS
L0S
L1S
L2S
L3S
L2F
L1X
L0X
L0F
L0E
ME
```