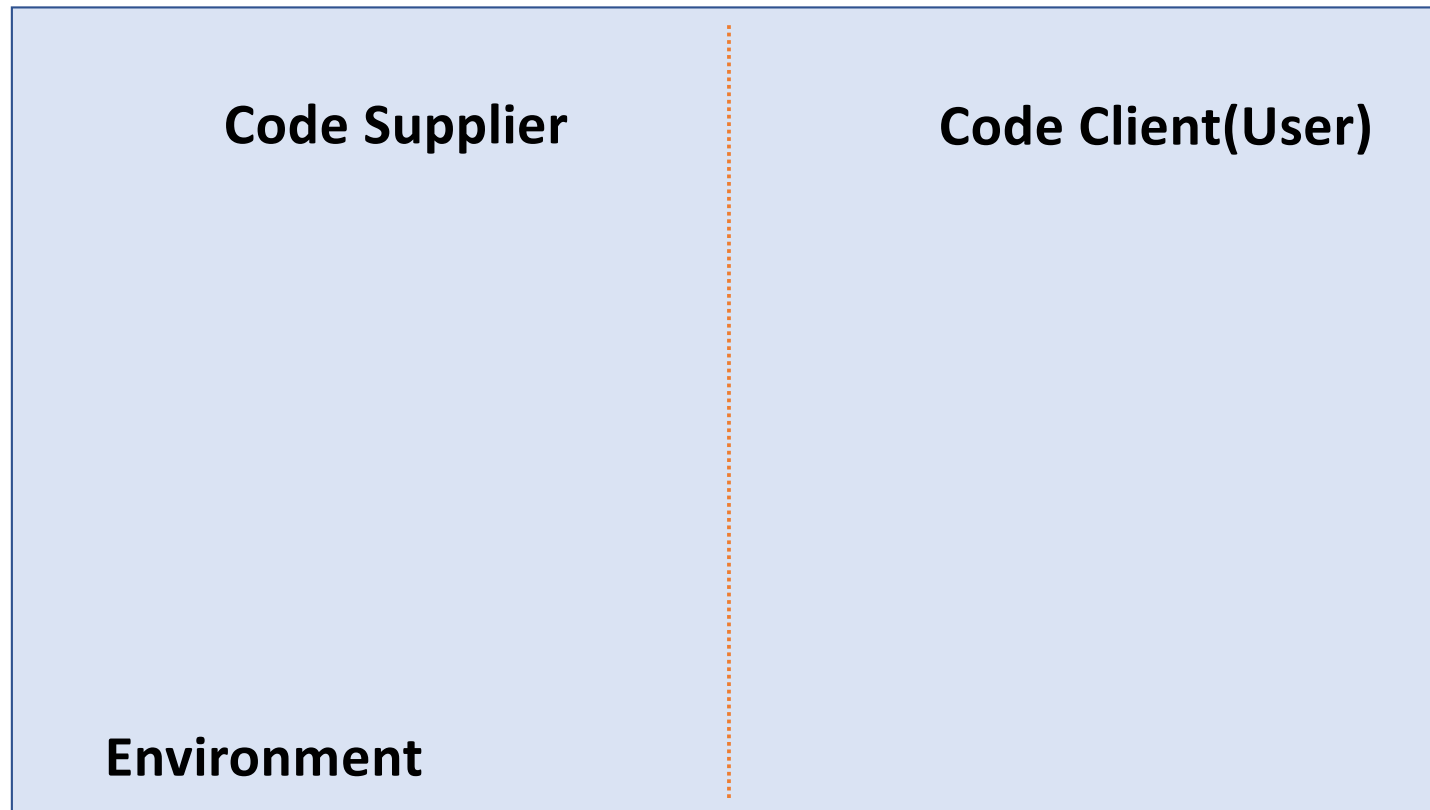


M4 (b) – Design for Robustness

Jin L.C. Guo

This Photo by Unknown Author is licensed under [CC BY-SA](#)

Where can things go wrong?



Java Convention for Checking Preconditions

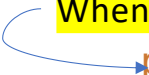
Explicit checks that throw particular, specified exceptions

Use assertion to test a *nonpublic* method's precondition that you believe will be true no matter what a client does with the class.

Java Convention for Private Method

```
* ... ...  
* @pre pStudent != null && !isFull()  
* @post aEnrollment.get(aEnrollment.size()-1) == pStudent()  
*/
```

When this is **private or protected**



```
public void enroll(Student pStudent) {  
    assert pStudent != null && !isFull() : this;  
    aEnrollment.add(pStudent);  
}
```

```
public boolean isFull() {  
    return aEnrollment.size() == aCap;  
}
```

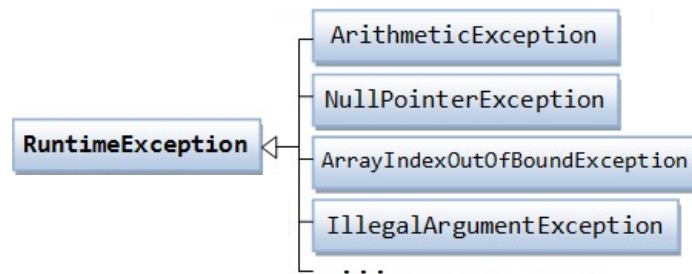
Java Convention for Public Method

```
/**
 * ...
 * @param Student to be enrolled to the Course
 * @throws IllegalStateException if isFull()
 */

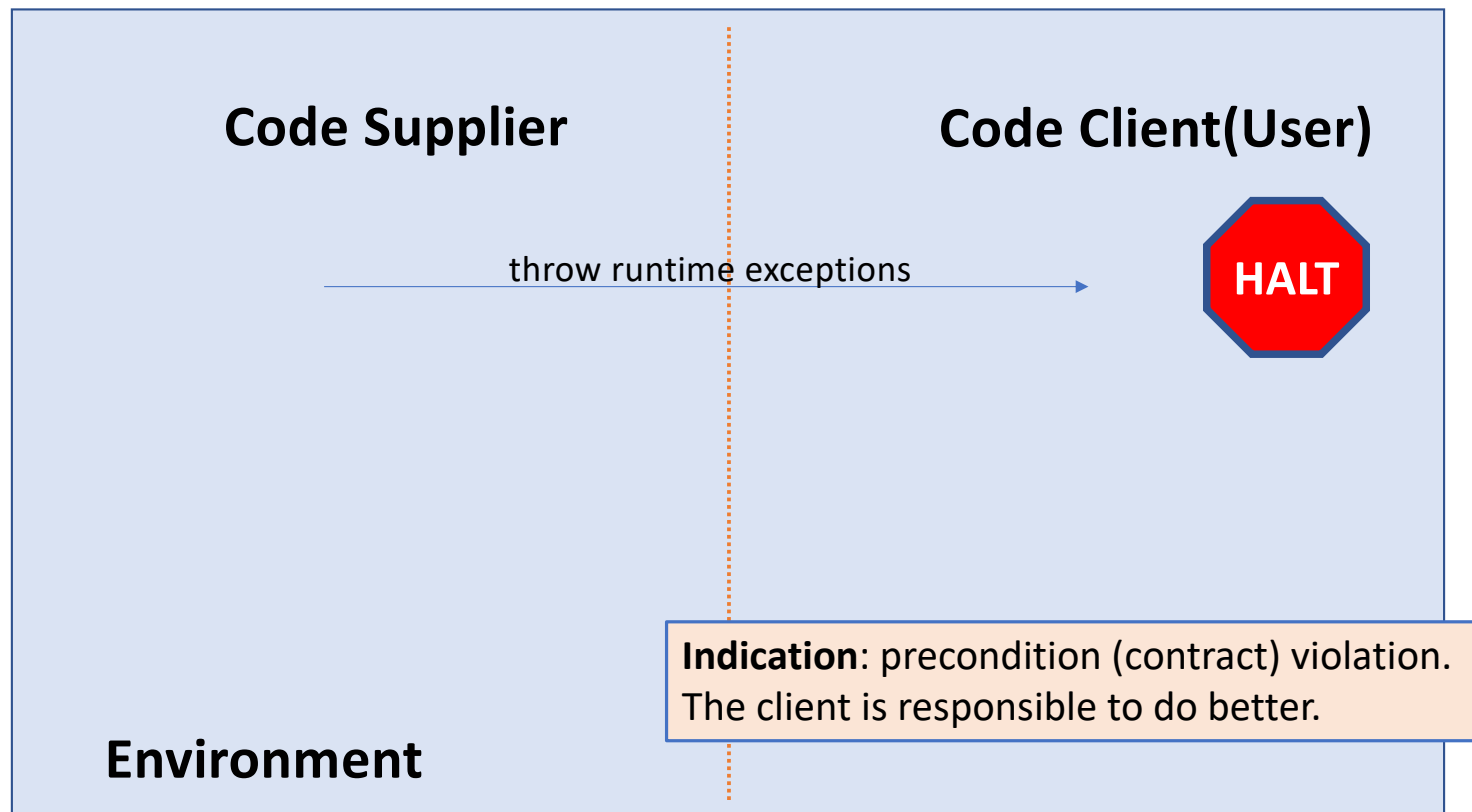
public void enroll(Student pStudent) {

    if (pStudent == null)
        throw new NullPointerException();
    if (isFull())
        throw new IllegalStateException();
    aEnrollment.add(pStudent);
}
```

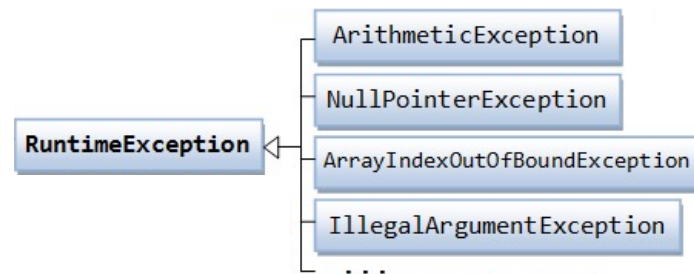
Runtime Exceptions



Code Interaction

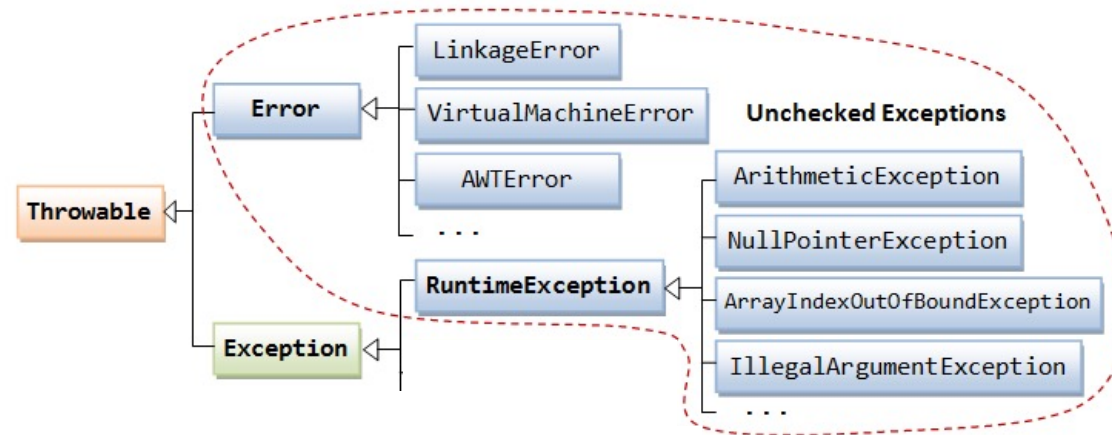


Runtime Exceptions



Unchecked Exceptions

They all cause the program to halt.



The whole hierarchy

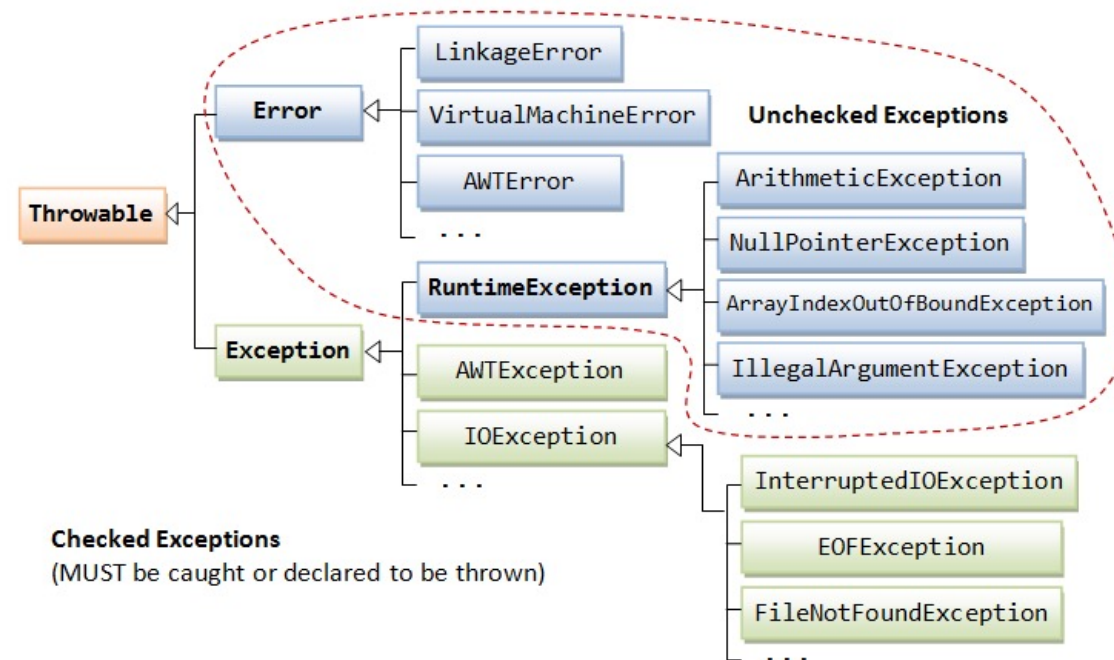
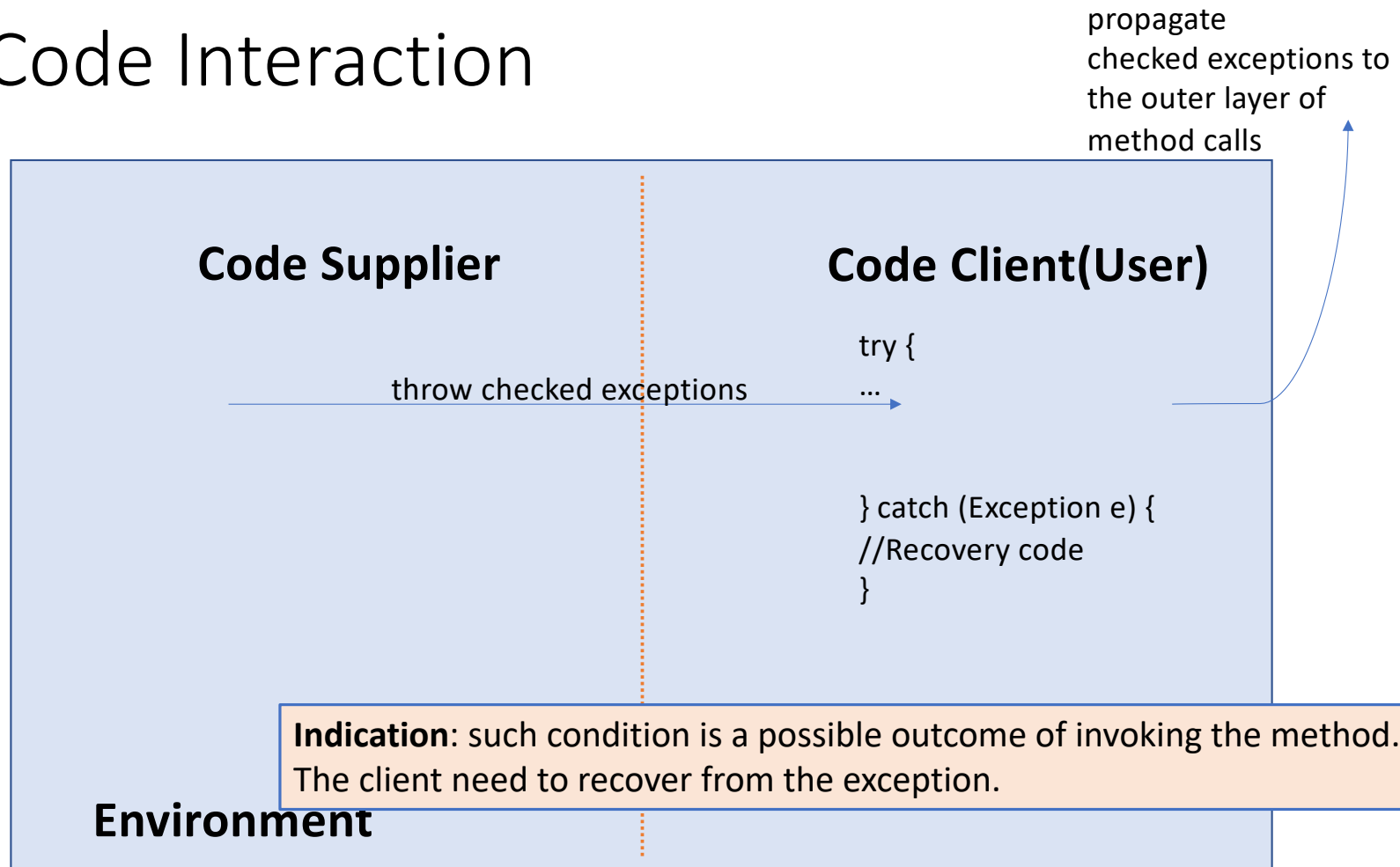


image source: http://www.ntu.edu.sg/home/ehchua/programming/java/images/Exception_Classes.png

Code Interaction



Another design of the `enroll` method

Assume `CourseFullException` is a Checked Exception

```
/**
 * Enroll the student to the course if the course currently is not full
 * @param pStudent to be enrolled to the Course
 * @throws CourseFullException if isFull()
 */
public void enroll(Student pStudent) throws CourseFullException {
    if (pStudent == null)
        throw new NullPointerException();
    if (isFull())
        throw new CourseFullException();
    aEnrollment.add(pStudent);
}
```

Impact to the Client

The client is not obliged to check `isFull()` anymore. However...


```
Course comp303 = new Course("COMP 303", 1);
Undergrad s1 = new Undergrad("00009", "James", "Harris");
Undergrad s2 = new Undergrad("00002", "Benny", "Will");

comp303.enroll(s1);
comp303.enroll(s2);

System.out.println("Done with enrolling students.");
comp303.printEnrolledStudent();
```

Impact to the Client

They have to catch the potential exception or propagate it



```
Course comp303 = new Course("COMP 303", 1);
Undergrad s1 = new Undergrad("00009", "James", "Harris");
Undergrad s2 = new Undergrad("00002", "Benny", "Will");
try {
    comp303.enroll(s1);
    comp303.enroll(s2);
    System.out.println("Done with enrolling students.");
} catch (CourseFullException e){
    ... // Handle the exception
    e.printStackTrace();
}
comp303.printEnrolledStudent();
```

Summary: Checked vs Unchecked Exception

- Checked Exceptions

Code supplier needs to declare in the method signature.

Code client needs to catch or declare.

Used for abnormal cases but can be recovered at runtime

- Unchecked Exceptions

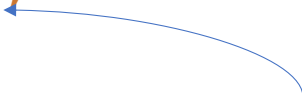
Code supplier does **not** have to declare it

Code client does **not** have to catch nor declare it.

Used for programming errors or things should not happen at runtime.

Any problem with this method?

```
public void writeToFile(Course pCourse, String pFilePath) {  
    File file = new File(pFilePath);  
  
    try {  
        FileWriter fileWriter = new FileWriter(file);  
        for (Student s : pCourse) {  
            fileWriter.write(s.toString());  
            fileWriter.write("\n");  
        }  
        fileWriter.close();  
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
}
```



If exceptions happen here

The `final` block

```
public void writeToFile(Course pCourse, String pFilePath) {
    File file = new File(pFilePath);
    FileWriter fileWriter = null;
    try {
        fileWriter = new FileWriter(file);
        for (Student s : pCourse) {
            fileWriter.write(s.toString());
            fileWriter.write("\n");
        }
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        try {
            fileWriter.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Alternative: try-with-Resources statement

```
public void writeToFile2(Course pCourse, String pFilePath) {  
    File file = new File(pFilePath);  
    try (FileWriter fileWriter = new FileWriter(file)){  
        for (Student s : pCourse) {  
            fileWriter.write(s.toString());  
            fileWriter.write("\n");  
        }  
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
}
```

`close()` will be called when the try block exits.

public interface **AutoCloseable**

An object that may hold resources (such as file or socket handles) until it is closed.

The [`close\(\)`](#) method of an AutoCloseable object is called automatically when exiting a try-with-resources block for which the object has been declared in the resource specification header. This construction ensures prompt release, avoiding resource exhaustion exceptions and errors that may otherwise occur.

All Known Subinterfaces:

AsynchronousByteChannel, AsynchronousChannel, BaseStream<T,S>, ByteChannel, CachedRowSet, CallableStatement, Channel, Clip, Closeable, Connection, DataLine, DirectoryStream<T>, DoubleStream, EventStream, ExecutionControl, FilteredRowSet, GatheringByteChannel, ImageInputStream, ImageOutputStream, InterruptibleChannel, IntStream, JavaFileManager, JdbcRowSet, JMXConnector, JoinRowSet, Line, LongStream, MappedMemorySegment, MemorySegment, MidiDevice, MidiDeviceReceiver, MidiDeviceTransmitter, Mixer, ModuleReader, MulticastChannel, NetworkChannel, ObjectInput, ObjectOutput, Port, PreparedStatement, ReadableByteChannel, Receiver, ResultSet, RMIConnection, RowSet, ScatteringByteChannel, SecureDirectoryStream<T>, SeekableByteChannel, Sequencer, SourceDataLine, StandardJavaFileManager, Statement, Stream<T>, SyncResolver, Synthesizer, TargetDataLine, Transmitter, WatchService, WebRowSet, WritableByteChannel

All Known Implementing Classes:

AbstractInterruptibleChannel, AbstractSelectableChannel, AbstractSelector, AsynchronousFileChannel, AsynchronousServerSocketChannel, AsynchronousSocketChannel, AudioInputStream, BufferedInputStream, BufferedOutputStream, BufferedReader, BufferedWriter, ByteArrayInputStream, ByteArrayOutputStream, CharArrayReader, CharArrayWriter, CheckedInputStream, CheckedOutputStream, CipherInputStream, CipherOutputStream, DatagramChannel, DatagramSocket, DataInputStream, DataOutputStream, DeflaterInputStream, DeflaterOutputStream, DigestInputStream, DigestOutputStream, DirectExecutionControl, FileCacheImageInputStream, FileCacheImageOutputStream, FileChannel, FileImageInputStream, FileImageOutputStream, FileInputStream, FileLock, FileOutputStream, FileReader, FileSystem, FileWriter, FilterInputStream, FilterOutputStream, FilterReader, FilterWriter, Formatter, ForwardingJavaFileManager, GZIPInputStream, GZIPOutputStream, HttpExchange, HttpsExchange, ImageInputStreamImpl, ImageOutputStreamImpl, InflaterInputStream, InflaterOutputStream, InputStream, InputStreamReader, JarFile, JarInputStream, JarOutputStream, JdiDefaultExecutionControl, JdiExecutionControl, JShell, LineNumberInputStream, LineNumberReader, LocalExecutionControl, LogStream, MemoryCacheImageInputStream, MemoryCacheImageOutputStream, MLet, MulticastSocket, ObjectInputStream, ObjectOutputStream, OutputStream, OutputStreamWriter, Pipe.SinkChannel, Pipe.SourceChannel, PipedInputStream, PipedOutputStream, PipedReader, PipedWriter, PrintStream, PrintWriter, PrivateMLet, ProgressMonitorInputStream, PushbackInputStream, PushbackReader, RandomAccessFile, Reader, Recording, RecordingFile, RecordingStream, RemoteExecutionControl, RMIConnectionImpl, RMIConnectionImpl_Stub, RMIConnector, RMIIIOpsServerImpl, RMIIJRMPServerImpl, RMIServerImpl, Scanner, SctpChannel, SctpMultiChannel, SctpServerChannel, SelectableChannel, Selector, SequenceInputStream, ServerSocket, ServerSocketChannel, Socket, SocketChannel, SSLServerSocket, SSLSocket, StreamingExecutionControl, StringBufferInputStream, StringReader, StringWriter, SubmissionPublisher, URLClassLoader, Writer, XMLDecoder, XMLEncoder, ZipFile, ZipInputStream, ZipOutputStream

```
public static void writeToFileZipFileContents(String zipFileName,
                                             String outputFileName)
    throws java.io.IOException {

    java.nio.charset.Charset charset =
        java.nio.charset.StandardCharsets.US_ASCII;
    java.nio.file.Path outputPath =
        java.nio.file.Paths.get(outputFileName);

    // Open zip file and create output file with
    // try-with-resources statement

    try (
        java.util.zip.ZipFile zf =
            new java.util.zip.ZipFile(zipFileName);
        java.io.BufferedWriter writer =
            java.nio.file.Files.newBufferedWriter(outputPath, charset)
    ) {
        // Enumerate each entry
        for (java.util.Enumeration entries =
            zf.entries(); entries.hasMoreElements();) {
            // Get the entry name and write it to the output file
            String newLine = System.getProperty("line.separator");
            String zipEntryName =
                ((java.util.zip.ZipEntry)entries.nextElement()).getName() +
                newLine;
            writer.write(zipEntryName, 0, zipEntryName.length());
        }
    }
}
```

Case study:

```
if(!comp303.isFull())  
    comp303.enroll(s2);
```

VS

```
try {  
    comp303.enroll(s2);  
} catch (CourseFullException e){  
    ... .. // Handle the exception  
}
```

When Not to use Exceptions

- For ordinary control flow