Exceptions

Exception handling
- Allows programmers to catch and handle error conditions that occur throughout a program.

Exception Object
- A java object that represents a certain error condition.
- Many types of Exceptions can occur (IOException, NullPointerException, ArithmeticException, …)
  - Java will create and "throw" an exception when it occurs for certain cases.
  - All exceptions are inherited from the Exception Java object.
  - Programmers can manually create and throw their own Exception object.
    - Can contain information or perform certain actions.
  - All Java Exceptions carry information such as the name of the Exception and stack trace.

Try / Catch
- Exceptions that are thrown need to be handled somewhere in your code.
  - An exception "bubbles up" your code
  - If it's unhandled in the method where it occurs, it has to be thrown to the caller. If the caller doesn't handle it, then the Exception gets thrown to its caller, and so on.
  - If the Exception reaches main and is not handled there, then your program will crash.
- Try / catch blocks are used to handle an Exception.
  - If an exception occurs within a try block, you can catch the Exception and execute code in a corresponding catch block.
- If a catch block is executed, it resumes functionality after the try/catch block.

Example

```
public class SomeClass {
      private Scanner s;
      public SomeClass() {
            s = new Scanner(System.in);
      }

      public int getInt() {
            int value = 0;
            System.out.print("Enter a positive number: ");
            try {
                  value = Integer.parseInt(s.nextLine());
            } catch (Exception e) {
                  // What exceptions can happen based on the user?
                  System.out.println("Caught Exception: " + e);
            }
            s.close();
            return value;
      }
}
////
public static void main(String args[]) {
      SomeClass x = new SomeClass();
      System.out.println("x = " + x.getInt());
}
```

Some cases

```
User Enters Valid Integer: 10
Output: x = 10

User Enters Invalid Integer: abcd
Caught Exception: java.lang.NumberFormatException: For input string:
"abcd"
Output: x = 0

User Enters an Integer not in range: 9999999999
Caught Exception: java.lang.NumberFormatException: For input string:
"9999999999"
Output: x = 0
```

Catching any Exception in Main

- … is generally not a good idea.
    - Not common to have the same reaction for all types of possible Exceptions that can occur.

Handling Multiple Exceptions

- When catching an Exception, you can execute different actions based on what Exception was caught.
- Java matches goes through each catch block one-by-one until a compatible Exception is matched.
- To catch any Exceptions, you can catch an Exception type (since all Exceptions are inherited from the Exception class).

Example

```
public static void main(String args[]) {
      SomeClass x = new SomeClass();
      x.updateArray(100, 100);
}
// In SomeClass.java
int[] array;


public SomeClass() {
      // Comment this out see a NullPointerException with a pos [0-99]
      array = new int[100];
}
```

```
public void updateArray(int pos, int value) {
      try {
            array[pos] = value;
      } catch (IndexOutOfBoundsException e) {
            System.out.println("In IndexOutOfBoundsException");
            System.out.println(e);
      } catch (NullPointerException e) {
            System.out.println("In NullPointerException");
            System.out.println(e);
      }
}
```

Cases

- If you call `x.updateArray(100,100)`
  - IndexOutOfBoundsException is thrown and handled in the corresponding catch block.
- If you comment out the line allocating the array
  - NullPointerException is thrown and handled in the corresponding catch block.

Finally Block

- Regardless of what happens in a try/catch block, the code in "finally" will always execute.

Example

```
// Change updateArray to take in user input
public void updateArray() {
      int pos = 0;
      int value = 0;
      try {
            System.out.print("Enter position: ");
            pos = Integer.parseInt(s.nextLine());
            System.out.print("Enter value: ");
            value = Integer.parseInt(s.nextLine());
      } catch (NumberFormatException e) {
            System.out.println("In NumberFormatException block");
            return;
      } catch (IndexOutOfBoundsException e) {
            System.out.println("In IndexOutOfBoundsException block");
            return;
      } catch (NullPointerException e) {
            System.out.println("In NullPointerException block");
            return;
      } catch (Exception e) {
            System.out.println("In Exception block");
            return;
      } finally {
            System.out.println("In finally block");
            s.close();
      }
      array[pos] = value;
}
```

Combining Exceptions into one block

- In Java versions < 7, each type of exception had to have its own catch block
- In Java versions >= 7, multiple exceptions can have the same catch block

```
public static void main(String args[]) {
      SomeClass x = new SomeClass();

      try {
            x.updateArray();
      } catch (IndexOutOfBoundsException | NullPointerException e) {
            System.out.println("In multiple exception catch block");
      }
}
```

Throwing Exceptions

- If your method handles the exception with a try/catch block, then it will throw an Exception (even if your catch block does nothing).
- If your method does not handle the exception with a try/catch block, then it's the responsibility of the caller of your method to handle it.

Checked vs. Unchecked Exceptions

- There are two classifications of Exceptions:
- CheckedExceptions
  - Checked during compile time.
  - A method **MUST** declare this in the signature if it is unhandled within the method.
- Unchecked Exceptions
  - Checked during runtime.
  - A method **CAN** (but doesn't have to) declare this in the signature if it is unhandled within the method.
  - Usually good to do so (for commenting purposes).
- For an illustration of checked and unchecked Exceptions see: http://www.programcreek.com/2009/02/diagram-for-hierarchy-of-exception-classes/

Example

```
// update main with the following…
public static void main(String[] args) {
      SomeClass x = new SomeClass();

      try {
            x.readFile("fjksl.txt");
      } catch (Exception e) {
            System.out.println("in main catch block");
            System.out.println(e);
      }
}
```

```
// in SomeClass
public void readFile(String filename) throws FileNotFoundException {
      Scanner inFile;
      try {
            inFile = new Scanner(new File(filename));
      } catch (Exception e) {
            // Change exception to NullPointerException
            System.out.println(e);
      }
}

// FileNotFound Exception is an example of a checked exception.
// You MUST handle it in the method or explicitly throw it.
```

Creating Your Own Custom Exception

- An Exception type can be declared as simple as defining the Class.
- You can create, throw, catch this type in your code depending on some action.
- You can also maintain additional information or provide your own methods if needed.

Example

```
// Create an Exception called MyException extending the Exception class
public class MyException extends Exception {}

// Catch or throw your exception whenever you want.
// Example of throwing the Exception to the caller.
public void someMethod(int x) throws MyException {
      if (x == 0) {
            throw new MyException();
      }
}
```

Testing

Complete Test

- Testing every possible path in every possible situation.
- Complete tests are infeasible!
- The best we can hope for is trying to approximate a Complete Test by testing various types of cases.
- The more rigorous testing a program can "pass", the more confidence is gained that the program "bulletproof".

Test Suite

- A program containing various tests confirming certain behavior.

Example

```
public static void main(String args[]) {
      // normal test cases
      runTestCase(1,2,3,4,4);
      runTestCase(1,2,4,3,4);
      runTestCase(1,1,3,2,3);
      runTestCase(-1,-2,-3,-1,-1);

      // Error Test cases:

      // Boundary Cases
      runTestCase(Integer.MAX_VALUE, 3,4,5,Integer.MAX_VALUE);
      runTestCase(3,Integer.MAX_VALUE,4,5,Integer.MAX_VALUE);
      runTestCase(Integer.MIN_VALUE,Integer.MIN_VALUE, -1, 0, 0);
      //...
}

public static void runTestCase(int a, int b, int c, int d,
                               int expected) {
   int result = getBiggestInt(a, b, c, d);
   if (result != expected) {
      System.out.println("result: " + result + " != expected: " +
                         expected);
   } else {
      System.out.println("result: " + result + " == expected: " +
                        expected);
   }
}

public static int getBiggestInt(int a, int b, int c, int d) {
   if (a >= b && a >= c && a >= d) {
      return a;
   } else if (b >= a && b >= c && b >= d) {
      return b;
   } else if (c >= a && c >= b && c >= d) {
      return c;
   } else {
      return d;
   }
}
```

Types of Test Cases

- Normal Cases: Any expected or "normal" input cases that you can reasonably expect from the user.
- Error Cases: Any case where it's not expected, but is possible
    - Passing in a null object, having a bad file name, disabling network connectivity for something that requires it…
- Boundary Cases: Testing the borders of the input.
    - Testing both max / min value for int parameters.

JUnit

- JUnit allows a programmer to perform automated tests.
- Runs one or more methods on the objects you are testing and checks the results using assertions.
- Tests are run one after another and will produce a graphical report on the cases that passed, the tests that encountered an error (i.e. crashed), and the tests that didn't crash, but failed an assertion.

JUnit Methods

```
assertEquals(expected_value, result_of_test);
assertFalse(boolean_result);
assertTrue(boolean_result);
assertNull(some_object);
assertArrayEquals(array, expected_array);
```

- And many more!

JUnit Example

- Make sure JUnit Library is part of your project's build path
- Configure Build Path → Libraries → Add Library → JUnit4

```
public class Tester {

      private int[] array;
      private Example b; // contains getBiggestInt method defined above

      @Before // Executed before each test in this class
      public void executeBeforeEachTest() {
            System.out.println("@Before: see before every test");
            array = new int[10];
            b = new Example();
      }

      @Test
      public void testInsertItem() {
            array[0] = 0;
            array[1] = 1;
            array[2] = 2;
            array[3] = 3;

            assertEquals(array[0], 0);
            assertEquals(array[1], 1);
            assertEquals(array[2], 2);
            assertEquals(array[3], 3);
      }

      @Test
      public void getBiggestTest() {
            assertEquals(b.getBiggestInt(1,2,3,4), 4);
            assertEquals(b.getBiggestInt(1,2,4,3), 4);
      }
```

```
	@Test
	public void getBiggestBoundaryTest() {
		assertEquals(b.getBiggestInt(Integer.MAX_VALUE, 3,4,5),
				Integer.MAX_VALUE);
		assertEquals(b.getBiggestInt(3,Integer.MAX_VALUE,4,5),
				Integer.MAX_VALUE);
	}

	// Testing an Expected Exception was thrown
	@Test(expected=ArithmeticException.class)
	public void testExceptionThrown() {
		int x = 0;
		int y = 1;
		int z = y / x;
		// Should crash, but doesn't since we're telling the test
		// that an exception should happen.
	}

	@After
	public void executeAfterTest() {
		System.out.println("@After: See this after every test");
	}

	@AfterClass
	public static void executeAfterAllTests() {
		System.out.println("@AfterClass: See this once after all
tests");
	}

	@BeforeClass
	public static void executeBeforeAllTests() {
		System.out.println("@BeforeClass: See this once before all
tests");
	}
}
```