WIT COMP1000

Exception Handling

Errors

- There are two main types of errors that programmers have to handle
- Compile/build errors occur when the compiler converts source code into byte code and are often the result of syntax errors in the source code
- Runtime errors occur when the program is executing and something Bad happens
- Runtime errors generally result in an *exception*

Exceptions

- An exception is *thrown* by a method or statement to indicate that an error has occurred
  - » Throwing an exception is similar to returning a value from a method, but exceptions are used only to communicate errors
- If the exception is not specifically handled by the program, the program will immediately terminate
- Examples: `InputMismatchException`, `ArrayIndexOutOfBoundsException`, `ArithmeticException`

Example: InputMismatchException

```
import java.util.Scanner;

public class ClassExamples {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        int inputValue;

        System.out.print("Enter an integer: ");
        inputValue = input.nextInt();

        System.out.printf("%d^2=%d%n", inputValue, inputValue*inputValue);
    }
}
```

If the user types in a value other than an integer, then an exception will be thrown

Unhandled Exceptions

- Unhandled exceptions result in program termination
- The JVM will output the type of exception and some information about the exception to the screen when the program terminates due to an exception
  - » These messages are unlikely to be useful to anyone other than the programmer
  - » There are mechanisms that we can use to check for exceptions in order to react to these cases and respond appropriately

Handling Exceptions

- In Java, `try` and `catch` blocks are used to handle exceptions within the program
- Any statements that might result in an exception should be placed inside a `try` block
- Every `try` block will be followed by one or more `catch` blocks
  - One `catch` block for each type of exception that needs to be handled from that `try` block

Example: Handling `InputMismatchException`

```
import java.util.InputMismatchException;
import java.util.Scanner;

public class ClassExamples {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        int inputValue = 0;

        try {
            System.out.print("Enter an integer: ");
            inputValue = input.nextInt();
        }
        catch (InputMismatchException ex) {
            System.out.println("Error!  An integer is required!");
            System.exit(0);
        }

        System.out.printf("%d^2=%d%n", inputValue, inputValue*inputValue);
    }
}
```

Note that we had to add an import line for the exception type used

Generic Form of `try` and `catch`

```
try {
    STATEMENTS THAT MIGHT THROW EXCEPTIONS
}
catch (EXCEPTION_TYPE1 EXCEPTION_VARIABLE1) {
    STATEMENTS THAT HANDLE EXCEPTION_TYPE1
}
catch (EXCEPTION_TYPE2 EXCEPTION_VARIABLE2) {
    STATEMENTS THAT HANDLE EXCEPTION_TYPE2
}
…
```

- If the same statements might throw more than one exception, you must have separate `catch` statements for each exception type
- You can also use more than one `try`/`catch` to handle exceptions from different statements

Exercise

- Write a program that asks the user for two integers, x and y, and then outputs the remainder of x divided by y
- Your program must use `try`/`catch` to print out useful error messages if the user does not follow directions (that is, if they enter a value that isn't an integer)

Answer

```
import java.util.InputMismatchException;
import java.util.Scanner;

public class ClassExamples {
	public static void main(String[] args) {
		Scanner input = new Scanner(System.in);
		int x = 0, y = 0, result = 0;

		try {
			System.out.print("Enter x: ");
			x = input.nextInt();
			System.out.print("Enter y: ");
			y = input.nextInt();
		}
		catch (InputMismatchException ex) {
			System.out.println("Must enter integers!");
			System.exit(0);
		}

		result = x % y;
		System.out.printf("%d mod %d = %d%n", x, y, result);
	}
}
```

Example: Two `try/catch` Blocks

```
import java.util.InputMismatchException;
import java.util.Scanner;

public class ClassExamples {
	public static void main(String[] args) {
		Scanner input = new Scanner(System.in);
		int x = 0, y = 0, result = 0;

		try {
			System.out.print("Enter x: ");
			x = input.nextInt();
			System.out.print("Enter y: ");
			y = input.nextInt();
		}
		catch (InputMismatchException ex) {
			System.out.println("Must enter integers!");
			System.exit(0);
		}

		try {
			result = x % y;
		}
		catch (ArithmeticException ex) {
			System.out.println("Can't divide by zero!");
			System.exit(0);
		}
		System.out.printf("%d mod %d = %d%n", x, y, result);
	}
}
```

`ArithmeticException` Notes

- `ArithmeticException` exceptions will only catch division by zero for integers, not doubles
  - This "correctly" produces NaN (not a number)
- Depending on the situation, it may be better to simply check for zero-valued divisors with an `if` statement
  - That will work for both numeric types
- You also don't need to import anything for the `ArithmeticException` type

Exceptions Thrown from Methods

- Often methods that you create will have the potential to throw exceptions as well
- You can catch the exceptions in `main()` by putting the method call in a **`try`** block
  - Note that you can also catch the exception in the method itself
- If you know that a method can throw an exception, then you should declare it as part of the method signature using the **`throws`** keyword

Example: Thrown from a Method

```
import java.util.InputMismatchException;
import java.util.Scanner;

public class ClassExamples {
	public static void main(String[] args) {
		Scanner input = new Scanner(System.in);
		int[] values = new int[5];

		try {
			fillArray(input, values);
		}
		catch (InputMismatchException ex) {
			System.out.println("Must enter integers!");
			System.exit(0);
		}
		printArray(values);
	}
	public static void fillArray(Scanner s, int[] a) throws InputMismatchException {
		System.out.print("Enter " + a.length + " integers: ");
		for (int i = 0; i < a.length; i++) {
			a[i] = s.nextInt();
		}
	}
	public static void printArray(int[] a) {
		for (int i = 0; i < a.length; i++ ) {
			System.out.println(a[i]);
		}
	}
}
```

Catching the exception in `main()`

Declaring that `fillArray()` might throw an `InputMismatchException`

Exception Propagation

- Exceptions have the potential to modify the standard control flow of programs
- If an exception isn't handled within the method that caused the error to occur, then that method will immediately end (with no return value provided to the caller)
- If the calling method doesn't handle the exception, then it will also end immediately
- This repeats until a method catches and handles the exception or `main()` is terminated

Wrap Up

- Exceptions are generated when an error occurs
- `try`/`catch` blocks are used to check for and handle exceptions appropriately
  - » To avoid having the program terminate without useful error messages to the user
- If an exception might occur within a method and you do not catch and handle it within the method, then use the `throws` keyword in the method signature to indicate which exceptions might be thrown
- We will not cover it here, but you can also create your own exceptions for extra error processing