

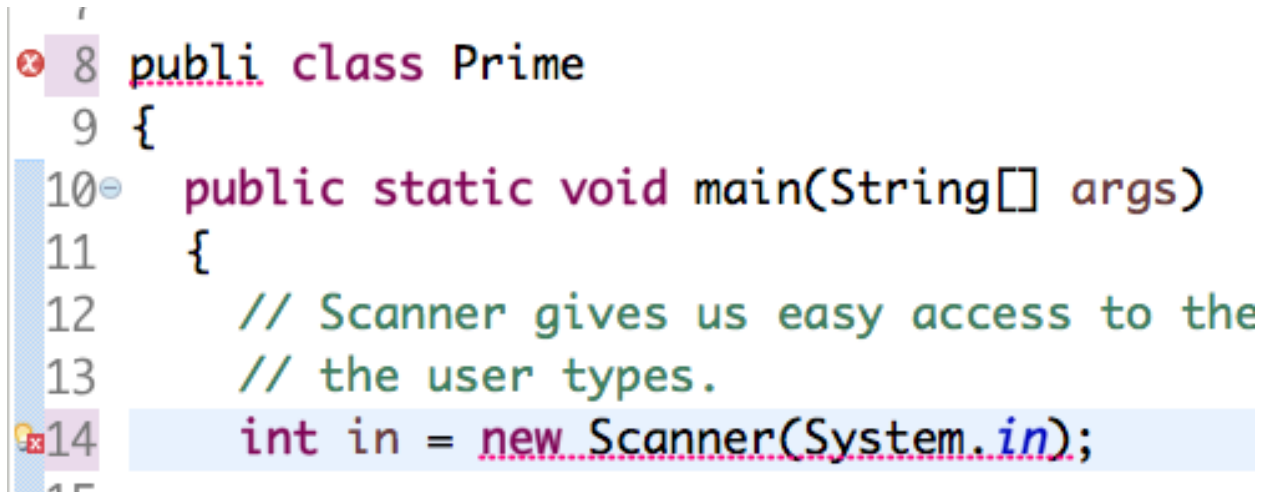
CMSC 206: Data Structures

Lab 2: Errors and Exceptions

This lab covers errors and exception handling in Java. It also covers how to read and write data to/from files using some of the simpler mechanisms provided in Java.

Understanding Errors: Syntax & Runtime Errors

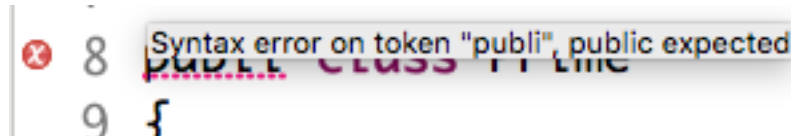
While entering code in Eclipse, you will invariably have typos or misspellings that will lead to syntax errors. Eclipse tries to highlight these using red circles/boxes as shown below:



```
1  
2  
3  
4  
5  
6  
7  
8 publi class Prime  
9 {  
10 public static void main(String[] args)  
11 {  
12 // Scanner gives us easy access to the  
13 // the user types.  
14 int in = new Scanner(System.in);  
15
```

Figure 1: Syntax errors are flagged as you enter your code in Eclipse.

Above, there are two errors flagged. In the first syntax error, the word `public` is misspelled as `publi`. If you move your mouse over the red circle, Eclipse will show you additional information about the error:



```
8 publi class Prime  
9 {
```

Syntax error on token "publi", public expected

Figure 2: Additional information about the error.

It also suggests a correction. If you make the correction, the error will go away. The second error is a *type error*, when you are trying to using a value of one type in a context expecting another. In this case, you've declared your `in` variable to have type `int`, but on the right-hand side of the `=` is something of type `Scanner`. If you point at the error underline, Eclipse will suggest changing the `int` to `Scanner`, which is the correct solution for this problem.

These two errors are fundamentally different: the first is a syntax error, while the second is a type error. To understand this a bit better, consider the following two English sentences:

1. Yesterday, I took my asldkma for a walk in the park.
2. Yesterday, I took my ardently for a walk in the park.

The first has a syntax error: there is a word in there that is not in your brain's dictionary. The second has a grammatical error (the natural-language analogue to a programming language's type error) – an adverb is used where we expect a noun. (Technically, the first error is called a *lexical*

error, distinct from the *parsing error* in a sentence like "Yesterday, I took my dog for a walk in the." We'll lump lexical errors and parsing errors together into the category of syntax errors.)

Both syntax errors and type errors are *compile-time errors*. These errors mean that the program does not make sense to the computer, and it cannot be run until the errors are corrected. When trying to run a program with compile-time errors, Eclipse still offers you the possibility of proceeding regardless; this is generally a bad idea. Fix the errors. Remember: error messages are meant to be helpful hints telling you how to improve your program.

Once you have a complete program free of compile-time errors, it is time to try and run it. No program runs correctly the first time you run it. There are always bugs. These are more technically called *logic errors*. (Continuing the relationship to English sentences: a logic error is analogous to the sentence "Yesterday, I took my god for a walk in the park." The sentence makes sense, but it's probably not what you meant.) Your program may successfully run to completion, but it may produce incorrect results. Or, maybe it runs forever. These problems may be due to faulty logic in your code. You will have to examine the program or its output and fix it. Later, we will learn ways to test your programs to minimize these errors and how to use the Eclipse debugger to dissect a running program.

Often, during program execution, you will run into errors due to which your program will crash! This happens very often during development and it is important to learn to recognize the cause of errors so that you can fix them. Runtime errors that result in program crashes typically print out the cause of the error, including the line on which the error occurred, in the Java Console. These may appear cryptic at first, but it is time to try and understand them.

Writing Code That Crashes!

The best way to understand runtime errors is to write code that crashes!

Exercise 1: Create a new project (call it *Crash1*) and place in it the following program:

```
public class Crash1
{
    private static int[] a = { 10, 20, 30, 40, 50 };
    // this is a convenient (correct) shorthand for
    // initializing an array

    public static void main(String[] args)
    {
        for (int i = 0; i < a.length; i++)
        {
            System.out.println(a[i]);
        }

        System.out.println("Done printing the array!");
    }
}
```

Run the program. This is the correct version. You will get the output shown below.

```
10
20
30
40
50
Done printing the array!
```

Now, let's break it. Change the termination test in the `for`-loop to the following:

```
i <= a.length
```

Can you tell what's going to happen? Why?

Run the program. You should now see the output shown below:

```
10
20
30
40
50
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5
    at Crash1.main(Crash1.java:8)
```

This kind of error is a *runtime error*. After printing out the last element in the array, the error occurred. Why? Did you guess correctly? (The sentence equivalent here might be "Yesterday, I took my dinosaur for a walk in the park." It makes good sense, but it's utterly impossible.)

Let's try and parse the error message. Runtime errors are flagged by *exceptions* in Java. If you read the first line, it is telling you that there was an exception in the thread "main". In computer science, a *thread* is a line of execution that a computer is processing; the control-flow arrows that

I draw on the board represent a thread. At any point there could be several threads of computation running – this is what happens when your computer is running several programs simultaneously. So, this exception occurred in the “main” thread, which is running your main method.

The error message also gives you the name of the exception:

`java.lang.ArrayIndexOutOfBoundsException`. The name says it all: you've accessed an array outside its bounds. And, that is exactly the error we introduced when we changed the termination test. We let the loop run beyond the bounds of the array, to an index 5, which in an array of five elements does not exist. The error message also includes the index number that caused the exception is also shown (5), as well as the line at which the exception occurred:

```
Crash1.main(Crash1.java:8)
```

That is, at line 8 of `Crash1.java` (your source code) was where you tried to access an array element that doesn't exist. This is line 8:

```
System.out.println(a[i]);
```

Since `i` is 5 and `a[5]` does not exist, you get the `ArrayIndexOutOfBoundsException`.

Java includes an elaborate set of classes that define various exceptions. Moreover, when these exceptions occur (technically, when some code *throws* an exception) it also gives you an option to write your programs so that you include code to handle (or *catch*) some exceptions. This is called *exception handling*. In many instances, you are required to provide code for handling anticipated exceptions. We will learn how to handle exceptions next.

Exceptions & Runtime Errors

Java has a class called `Exception` and several dozen types of exceptions that can arise are defined as subclasses of the `Exception` class. Some of the errors occur during input/output (for example, a broken network connection, or a misplaced input file, etc.) and others during the course of executing your logic (for example an out of bounds array access, or trying to access an object before instantiation, etc.). Sometimes, it is possible to recover from an error. To do this, Java provides a way for you to detect and specify corrective measures.

Java provides a `try/catch` statement block to detect and handle exceptions. Its basic syntax is shown below:

```
try {
    ...some code during whose execution errors may occur...
} catch (<some exception> <variable>) {
    ...some corrective action you could perform...
} catch (<some other exception> <variable>) { // optional
    ...some corrective action you could perform...
} // could have more "catch" clauses
```

Essentially, you enclose the code that may (or is likely to) result in an error in a `try/catch` statement block as shown. If, in the course of executing the surrounded code an exception occurs, the code included in the `catch`-block is executed, if the exception named in the `catch` block is the one that happened. The following exercise illustrates this.

Exercise 2: Write a program to input a number and output its square root. Create a new project, call it `Crash2`, and enter the code shown below.


```
import java.util.*;

public class Crash2
{
    public static void main(String[] args)
    {
        Scanner in = new Scanner(System.in);

        while(true)
        {
            System.out.print("Enter a number: ");
            String line = in.nextLine();
            int data = Integer.parseInt(line);

            System.out.println("The square root of " + data +
                               " is " + Math.sqrt(data));
        }
    }
}
```

The code shown above repeatedly prompts the user for a number and then computes and outputs the square root of that number. The number is actually input as a string. You use the Java function `Integer.parseInt` to extract a number from the input string. (Look it up if you're unfamiliar with it!) There is also a `Double.parseDouble` if you want to accept floating-point (decimal) input.

The `while`-loop ensures that the program does this forever. Run the program and enter some numbers to try it out. Enter only positive integer values at first. Then, after a few successful tries, type in something that's not a number. What happens? (Recall that you can press the  "terminate" button, above the Console, to stop a program that loops forever.)

The program crashes with the following error:

```
Exception in thread "main" java.lang.NumberFormatException: asdf
    at java.lang.Integer.parseInt(Integer.java:454)
    at java.lang.Integer.parseInt(Integer.java:527)
    at Crash2.main(Crash2.java:13)
```

Can you guess why that happens? Note that the error is occurring at line 13 of your program. That is the line where you are trying to convert the string to an integer. The exception `java.lang.NumberFormatException` says that there was a problem in the number format: you typed in something other than a number.

To handle this possibility, we need Java's exception handling mechanism.

The `try/catch` Statement

The code shown below uses a `try/catch` statement to address the input of a non-numeric string in the above program.

Exercise 3: Read and understand the code below to see how a `try/catch` block is structured and used. Modify your program in Eclipse to work this way and experiment with it.

```
import java.util.*;

public class Crash2
{
    public static void main(String[] args)
    {
        while(true)
        {
            Scanner in = new Scanner(System.in);

            try
            {
                System.out.print("Enter a number: ");
                String line = in.nextLine();
                double data = Double.parseDouble(line);
                System.out.println("The square root of " + data +
                                   " is " + Math.sqrt(data));
            }
            catch(NumberFormatException e)
            {
                System.out.println("That's not a number!");
            }
        }
    }
}
```

Run the above program. Enter integer values, floating point values (note that this version changes `Integer.parseInt` to `Double.parseDouble` to allow it to accept a greater range of input), as well as other garbage. Notice how your program is now robust enough to do the right thing when a number is input and recovers in situations when nothing or some non-numeric input is provided. It does not crash!

File Input/Output (I/O): The `Scanner` class

Assignments in this course will typically deal with lots of data. This data will often be read by your program from a data file. Java has several ways to access files and read data from them. The `Scanner` class is one of the simplest.

Exercise 5: Write a program that reads a text file and prints out its contents in the Java Console. First, create a new project (call it *TextIO*). Next, create a small data file. To do this, right-click on the project folder, and then navigate to *New... -> Folder* option. Name the folder *Data* and *Finish*. Next, right-click on the *Data* folder and navigate to *New... -> Untitled File*. In the new editor window that pops up, enter a few lines of text. I used these, from a random tweet:

@LiamNeeson

I always wanted to be a cowboy
And Jedi Knights

Are basically cowboys in space

Right?

Save the file (I named mine *LiamNeeson.txt*). Make sure that now you have a *Data* folder in the *TextIO* project folder, and in the *Data* folder you have the text file you created.

Now that you have a text file to read, we can build a program that reads and prints out its contents. As explained in Appendix A of your text (see pages 662-665), the `Scanner` class in Java is defined in the `java.util` package and enables the reading of structured data from an input source (the keyboard, or a file, or even another source). In order to read text from a file, one line at a time, you do the following:

1. Import the `java.io` and `java.util` packages:

```
import java.io.*;
import java.util.*;
```

2. Create a new `Scanner` object linked to the data file using the `File` class in Java:

```
Scanner input = new Scanner (new File(<name of input file>));
```

3. Use the `hasNextLine()` and `nextLine()` methods to look for and read the next line:

```
while(input.hasNextLine())
{
    String line = input.nextLine();
    ...
}
```

4. Close the input stream:

```
input.close();
```

Whenever you access files in a program, many, many things can go wrong. An easy example is that a file might not exist, but other problems can occur, too. Accordingly, these operations force you to enclose the entire operation in a `try/catch` block. The program below puts all these together to show how to read text from a file and then output its contents to the Console.

```
import java.io.*;
import java.util.*;

public class TextIO
{
    public static void main(String[] args)
    {
        String inFileName = "Data/LiamNeeson.txt";
        Scanner input;
        String line;
```

```

try
{
    // Create a new Scanner for the input file
    input = new Scanner(new File(inFileName));

    // test if there is a line to read
    while (input.hasNextLine()) {

        // read the next line
        line = input.nextLine();

        // output it to Console
        System.out.println(line);
    }

    // Close the input stream
    input.close();
}
catch (FileNotFoundException e)
{
    System.out.println("Error in opening the file: " +
                      inFileName);
    System.exit(1);
}
}

```

Run the program. Did you see the contents of the text file in the Console? If not, what went wrong? Once it is running correctly, deliberately change the name of the file and run the program. What do you see? Exceptions at work!

(What does that `System.exit` do? Look it up in the Java API, linked from the main course webpage. The `System` class is in the `java.lang` package.)