What we will cover:
Types of errors: Syntax, runtime, and logic
Debugging Techniques

Types of Errors
There are three categories of errors: syntax, runtime, and logic. Errors are unavoidable while programming, but you can minimize the time lost to finding and fixing them by being familiar with error types and characteristics.

Syntax Errors
Detected, at the latest, at compile time, syntax errors are the easiest errors to find. Syntax errors are errors in your code, misspellings, omitted semicolons or braces, or method or class signature errors. You should get used to decoding and understanding the verbiage that the compiler uses to describe errors. Listed below are common syntax errors and the associated compiler output.

NOTE: Some of these errors come from the javac command-line compiler. Eclipse indicates syntax errors in real-time and has no distinct compilation process, only a “Run” process which gives slightly different error messages. You can click on the red X (sometimes a light bulb) in the gutter of the Eclipse editing window for descriptions of the Eclipse error.

Misnamed/wrong-case class declaration
Remember that class names must match, exactly, the file name. In a file named ExampleProgram.java:

```
public class exampleprogram {
    public static void main(String[] args) {
    }
}
```

C:\errors\src>javac ExampleProgram.java
ExampleProgram.java:1: class exampleprogram is public, should be declared in a file named exampleprogram.java
public class exampleprogram {
    ^
1 error

Undeclared/un-typed variables
Declare, identify, and give variables a data type before use. Neglecting any of those items will cause an error. Make sure that you use the same name later in the program when referring to a variable.

```
public class ExampleProgram {
    public static void Main(String[] args) {
        i = 12;
        System.out.println("i: " + i);
    }
}
```

C:\errors\src>javac ExampleProgram.java
ExampleProgram.java:3: cannot find symbol
    symbol : variable i
    location: class ExampleProgram
              i = 12;
              ^
ExampleProgram.java:4: cannot find symbol
    symbol : variable i
    location: class ExampleProgram
              System.out.println("i: " + i);
              ^
2 errors

Errors cascade downward as shown. Always start at the first error, which is probably causing some of the other errors, and fix that one first. Recompile the program and see which errors are still there.
Missing semicolon/brace/parentheses
Leaving one of these important symbols out is often a little trickier to find because the compiler reads statements by finding the semicolon. In some languages, and earlier versions of Java, the location indicated would sometimes be after the location of the actual error.

```java
public class ExampleProgram {
    public static void main(String[] args) {
        int i = 12;
        System.out.println("i: " + i)
    }
}
```

C:\errors\src>javac ExampleProgram.java
ExampleProgram.java:4: ';' expected
    System.out.println("i: " + i)
^
1 error

Runtime Errors
Runtime errors are, obviously, errors that happen when attempting to run a program. This will often lead to the early termination of the program. Getting input from a user is often a source of runtime errors. If you are expecting an integer, but the user enters a double, the program will generate an error and stop running.

```java
import javax.swing.JOptionPane;
public class ExampleProgram {
    public static void main(String[] args) {
        String temp = JOptionPane.showInputDialog(null, "Please enter an integer");
        int i = Integer.parseInt(temp);
        JOptionPane.showMessageDialog(null, "Number of seconds: " + i);
    }
}
```

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "3.0"
at java.lang.NumberFormatException.forInputString(Unknown Source)
at java.lang.Integer.parseInt(Unknown Source)
at java.lang.Integer.parseInt(Unknown Source)
at ExampleProgram.main(ExampleProgram.java:6)
```

There are no syntax errors in this program, but it does expect an integer value and exits ungracefully when given a double.

When the main method is declared incorrectly, a subtle error can occur. Java allows a programmer to create their own methods and overload existing methods, a process of adding extra functionality to an existing method. If you do not specify the correct method signature for the `main` method, then the compiler treats it like a different method and then cannot find a `main` method to run the program.

```java
public class ExampleProgram {
    public void main(String[] args) {
    }
}
```
Eclipse catches this error and prevents the program from running. javac does not catch this error and compiles the program successfully. At runtime, you get this error:
C:\errors\src\java ExampleProgram
Exception in thread "main" java.lang.NoSuchMethodError: main

Division by zero is another common runtime error. These errors are easy to check for and prevent with conditional checks, but they could be a side effect or the result of a logic error. You could initialize a variable early in a program and forget to assign it a value, or it may be assigned a value of 0 at some point in the program.

```java
public class ExampleProgram {
    public static void main(String[] args) {
        int numerator = 0, denominator = 0;
        int result = 0;
        numerator = 5;
        result = numerator / denominator;
        System.out.println(numerator + " / " + denominator + " = " + result);
    }
}
```

Exception in thread "main" java.lang.ArithmeticException: / by zero
at ExampleProgram.main(ExampleProgram.java:8)

This also brings up the issue of integer division. As shown in previous lectures, care must be taken to ensure that you are not performing integer division even when your data types are declared directly as doubles or floats.

Another error you will see occurs when trying to access an index or element of an array that does not exist. We will illustrate this with a String.

```java
public class ExampleProgram {
    public static void main(String[] args) {
        String test = "Test";
        System.out.println(test.charAt(5));
    }
}
```

Exception in thread "main" java.lang.StringIndexOutOfBoundsException: String index out of range: 5
at java.lang.String.charAt(Unknown Source)
at ExampleProgram.main(ExampleProgram.java:6)

Logic Errors
Logic errors are what we typically think of when referring to a bug in a program. A logic error occurs when the program does not perform in the way that the developer intended. This usually indicates a problem with the way the program has been written or structured. Remember that you should always have an expectation of how the program is going to behave before you run it. If something different happens and there are no errors, you should examine your code.

```java
import javax.swing.JOptionPane;
public class ExampleProgram {
    public static void main(String[] args) {
        String temp = JOptionPane.showInputDialog(null, "Please enter a number");
        double number = Double.parseDouble(temp);
        double result = number + number;
        JOptionPane.showMessageDialog(null, "The square of " + number + " is " + result);
        System.exit(0);
    }
}
```
Debugging
Debugging is the process of finding and fixing errors. There are a number of ways to track down errors and you will probably use a mix of them based on the type and severity of error that you encounter. Caught by the compiler or IDE, syntax errors generally include a line number where the error occurs. Runtime errors will occur only at runtime, but will still generally include a line number. The hardest type of error to diagnose is the logic error. This is where early planning and design, along with proper expectations will come in handy.

Debugging Techniques
The technique of debugging can range from simple actions to complex ones involving other software to examine your software.

Tracing
Tracing is the simplest of all debugging techniques. Hand tracing typically involves reading the program and executing it in your head or on paper to see what the code is doing. Another common approach to this is to add print statements to your code to output the value of variables at various places in the code’s execution or to indicate which conditional branches the code is following. Before the advent of built-in debuggers and other complex tools, adding print statements was (and continues to be) a popular method for finding errors.

By exploiting cognitive dissonance, rubber ducking is another popular debugging technique that is closely related to tracing. Many people have noticed that when explaining a problem to someone, they will often stumble upon the answer themselves, without the other person’s help. Your brain is uncomfortable holding competing ideas at the same time. Often, knowing what the program should be doing prevents you from seeing what it is actually doing. Voicing the idea aloud often exposes this difference and allows you to notice the error. It has been noted that, instead of wasting another person’s time, this idea can be implemented by speaking to an inanimate object, hence, rubber ducking. This also works if you read the program line-by-line and say, aloud, what the program is doing. This has a tendency to expose errors by showing where the code is doing something that is not solving your current problem.

Test Cases
You should work out what the results should be for certain sets of input. Knowing what results you should obtain from your program lets you know immediately if something has gone wrong. In addition, you should test boundary or edge cases, or even cases where you know the program should fail to make sure it behaves correctly. Once we start using error checking conditional statements, this will become even more important. You should always test all possible branches of a program to ensure that all of your code is correct, not just the code that is executed because of poor test case selection. Test cases can also test completely rewritten sections of code to ensure that old, correct behavior has been replicated in new code.

Code Removal/Commenting/Extraction
The reason we compile and test early and often is to expose errors while working on the bare bones of the problem. Once you have a fully fleshed-out, complex program with error checking, input validation, and other features, debugging can become a tedious task. Commenting code that is not working directly on the data in which your error is occurring can help you track down the offending line(s) of code. Essentially, this method tries to ensure that you are only dealing with code that is solving your problem and not code that is providing other functionality. An alternative to this is to create a new class that only contains the tiny bit of code that is causing problems.
Built-in debuggers
Most IDEs have robust and feature-rich internal debuggers that will allow you to move step-by-step through your code and examine the values of variables and other items during the execution of your program. Setting breakpoints tells the debugger to stop executing the code automatically and to hand over control of execution to the developer. One common option includes stepping into a function, which will take you into the inner workings of that statement. At a breakpoint, you get a snapshot of the variables and their values. You can examine the variables at that point and determine if your programming is behaving correctly.