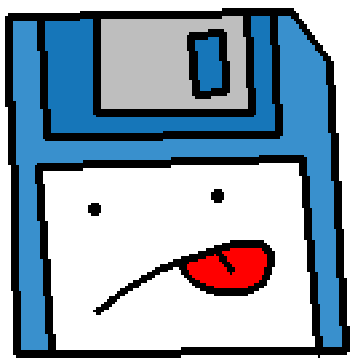
Table of Contents

Introduction

Bugs happen. No matter how good you are at programming, or what you're working on, there is bound to be some bugs. It comes with the territory.

Since you will always have to deal with bugs, it is good to have some knowledge of what kind of bugs there are, and how to diagnose and fix these bugs.

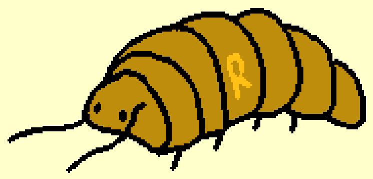


Know your types of bugs

Syntax Error

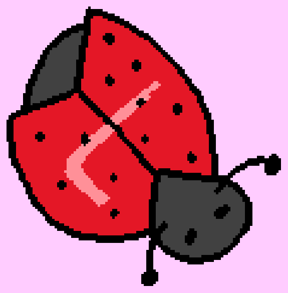
A Syntax Error is when something in the language has been incorrectly typed.

The compiler can easily detect these errors, and you will receive a compile-time error.



Run-time Error

A run-time error is something that is not “grammatically incorrect” in your program, but something breaks (or doesn't work as intended) while the program is running.



Logic Error

A logic error is an error that doesn't break the program, but doesn't work as intended – it is an error in logic. It could simply be getting a formula wrong, or it could be a typo that results in unintended consequences.

You can find the reason behind most compile errors with a search

Sometimes, compiler build errors are useful. Sometimes, they're not. In the times where the message is vague or misleading, you might try to do a web search for the verbatim error message. Often, others have run up against the same kind of error messages, and helpful veterans to coding will help out with figuring out what is going wrong.

Debugging tools help you with run-time and logic errors!

Usually the IDEs (Integrated Development Environments) used to build software contain debugging tools. These might be features like pausing the program to see the variable values at that snapshot in time, setting breakpoints to step through code line-by-line, and a stack trace, which will list all the functions that have been called and in the order they were called.

Debugging Tips

Reproduce the error

Bugs are much easier to go after if you're able to come up with a set of steps that reliably reproduce the error. If you can create the same error every time, you can better know where to look in the code.

You might also try to come up with several variations of steps to come up with the same error as well, which will further help you figure out which functions are at fault, and which are not.

Diagnose the error

Once you know how the error is made from the user's point-of-view, locate the functionality in the code that is run during the reproduction steps.

Check your code with a read-through, but if it isn't immediately obvious what is going wrong, you might want to use some debugging techniques to figure out where the logic is failing.

Tips to track down the problem code

Output debugging

A quick-and-dirty way to try to find out problem parts of your program, and to see the program flow, is to add **print** or **output** statements in your program.

For example, putting "**functionName begin**" and "**functionName end**" at the beginning and ending of all of your functions will help you see *where* in the program execute it may be crashing.

By displaying the values of your variables, you can make sure you have the correct inputs and outputs that you expect at each step of the code.

Debug tools

Visual Studio and other development environments have debugging tools such as **breakpoints**, **immediates**, **locals**, and **stack trace**.

- Breakpoints – allow you to set points in the program where execution will pause and return to the IDE. Then, you can use buttons like **step over** and **step into** to go through the program one line at a time.
- Immediates – This is usually a terminal where you can type in some code, like a variable name, to see what a variable's value is at that point in time.
- Locals – This is a table that will display all the variables and their values within the current function.
- Stack Trace – This will list out all the functions that have been called at the point in the program, from most to least recent.

Common errors

1 Typos

Double-check the name of a variable, method, or data-type to make sure it is spelled right, and the casing is correct (upper-case vs. lower-case letters.)

```
double priceWithTax;
// variable name is spelled wrong
PriceWithTax = price + price * tax;
```

2 Double-declarations

Make sure you're not *redeclaring* a variable instead of simply using it. For example, a class might have a member variable called *bookCount*, but if you declare a new variable with the same name within a function, you will be updating that new local variable, **not** the member variable!

```
class Pet
{
    String name;

    String getName()
    {
        // Redeclaring name
        String name;
        return name;
    }
}
```

3 No return-type

When you have a non-void return type in a function/method, you need to make sure to return a value from that method. If your method contains an "if" statement with no "else", it is possible that there is some program state that will result in the function not returning a value.

```
double Price( int age )
{
    if ( age < 20 )
        return 9.99;

    // missing return if
    // age is not less than 20.
}
```

4 Not storing the return-value

When a function returns a value, you need to make sure you're storing its return value at the **caller** level. In other words, if you're calling a function, but not assigning it to a variable, you will lose that return value.

```
class Pet
{
    private String catName;

    public String getName()
    {
        return catName;
    }
}

// Elseware...
String name;

// Return value is not being
// stored anywhere.
myPet.getName();

// Nothing is stored in name.
System.out.println( "Name: " + name );
```

5 Semi-colons at the end of if statements, while loops, etc.

If you put a semi-colon at the end of an if statement line, then none of the code within the { } code-block will be executed: it will assume that the semi-colon is all that needs to be executed if the condition is true.

```
if ( age < 18 );
    System.out.println( "Can't vote!" );
```

6 Single equals instead of double equals in a conditional

Remember that the single equal sign is the **assignment operator**, while the double equal sign is the **equality operator**. If you accidentally put a single-equals in an if statement, your statement is essentially asking, "*Can* I assign this value to this variable?" rather than "Are these two values equal?"

`// Can I assign "admin" to username?`
`if ( username = "admin" )`
	`// Is username equal to "admin"?`
	`if ( username == "admin" )`

7 Not instantiating a variable whose data type is a class

For Java, when you declare a variable and the data-type of that variable is a **class**, it needs to be instantiated with the **new** keyword. If you don't, the program won't compile.

In C++, you don't need to do this if you're declaring a variable whose data-type is a class.

```
// needs to be instantiated
// Random random = new Random();
Random random;
int num = random.nextInt();
```

8 Not instantiating an array

When you declare an array in Java, you have to use the **new** keyword to instantiate it.

In C++, you don't need to do this.

```
// Should be
// double[] prices = new double[5];
double[] prices;
prices[0] = 9.99;
```

9 Not instantiating the elements of an array

In Java, when you declare an array and that array has a data-type that is a class, you have to instantiate **each element of the array** as well as the array itself.

If you don't instantiate the elements, you will get the error:

java.lang.NullPointerException

You do not have to do this for C++.

```
// No:
MyCat[] catList;
catList[0].setName( "Fluffy" );

// Yes:
MyCat[] catList = new MyCat[3];
catList[0] = new MyCat;
catList[1] = new MyCat;
catList[2] = new MyCat;
catList[0].setName( "Fluffy" );
```

10 Going outside of the bounds of an array

If an array is of size 5, the valid indexes are: 0, 1, 2, 3, and 4. If you try to access the element at position 5, then your program will crash.

Usually this error occurs when working with for-loops and not specifying the correct start or end points for the array indices.

```
String[] names = new String[3];

// Should access 0, 1, and 2, but not 3.
for ( int i = 0; i <= 3; i++ )
{
    names[i] = "default";
}
```

11 Bad boolean logic

Remember to validate your boolean logic, otherwise you could end up writing yourself into an infinite while-loop (or just some code that never executes!) For example, with this code:

```
if ( x < 0 && x > 10 )
{
    System.out.println( "invalid entry!" );
}
```

It is impossible for x to be both less than 0 AND greater than 10. The programmer probably meant to put OR!