Table of Contents

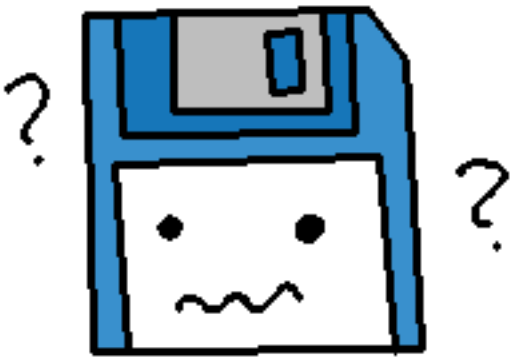
Introduction

Beyond the challenge of learning the syntax and semantics of a programming language (which can be as challenging as learning a human language), another big challenge of programming is the **problem solving aspect**.

Programs need to be written to solve some problem, or to automate some task, or to compute some numbers, but we don't just wave a magic wand and have it solve the problem – we have to tell the computer how to do **each step.**

Even your text editor needs to be told:

- How to I read an existing text file? (This includes the text, colors, images, etc.)
- How do I save a file?
- How do I draw the file contents to the screen?
- How do I draw the rest of the user interface that is used?
- How do I respond to keyboard input?



If you were writing your own text editor – how would *you* begin?

Maybe that question is overwhelming at this point if you have just started programming, or have some experience with making text-based programs, but nothing with a Graphical User Interface. Perhaps you could start with a text-based text editor using just your language's **input** and **output** functionality – OK, but then how do you implement it?

This is where problem solving comes in handy. **Strategy #1: Break down the problem.**

Strategy 1: Define the problem in your own words

The first step is to make sure you understand *what* you're actually supposed to be doing. Is there a program specification, or a specific problem to solve?

If there is a spec, you might have to re-read it several times: skim through first, then investigate each section. To really grasp what is being said, it might be good to be able to reiterate the instructions in your own words.

Specification

If the value of **totalBooks** is equal to 10, then the library is full of books. Display an error, "Library full: Cannot add new books", if this happens, and **return** from the method early.

Otherwise, allow the method to continue:

Prompt the user for the following information: Book title, author, and ISBN. Title and Author are strings, and ISBN is an integer. Store these in temporary variables within the method.

Every time we add a book, we increment **totalBooks** by 1. We can get the first open slot available to us in the array using this variable as well.

Use **books[totalBooks]** to call the **setTitle**, **setAuthor**, and **setIsbn** methods from the **Book** class to set up the book. Afterward, increment **totalBooks** by 1.

My Words

1. Check to see if the list of books is full...
    1. If it is full, leave the method.
2. Otherwise, allow the user to add a book...
    1. Ask for the title
    2. Ask for the author
    3. Ask for the ISBN
    4. Store these values in the "Book" list element.
    5. Add 1 to the total book count.

This also helps you to figure out what kind of questions to ask the person providing you the specification – such as the instructor. If you don't understand a feature, once you know the general layout of the program, it can be much easier to ask, *"I don't understand how the totalBook variable and the books array are connected"*.

Strategy 2: Break it down

Take the problem or the program specification and figure out what the different **parts** of the program will be, as well as sketch out an algorithm in the form of a list of commands written in English.

Figure out the main features of the program, and break down how each of those features work.

It is best to focus on one feature at a time and work your way up. If you only focus on one feature, you can properly test *just that feature*.

Warning!

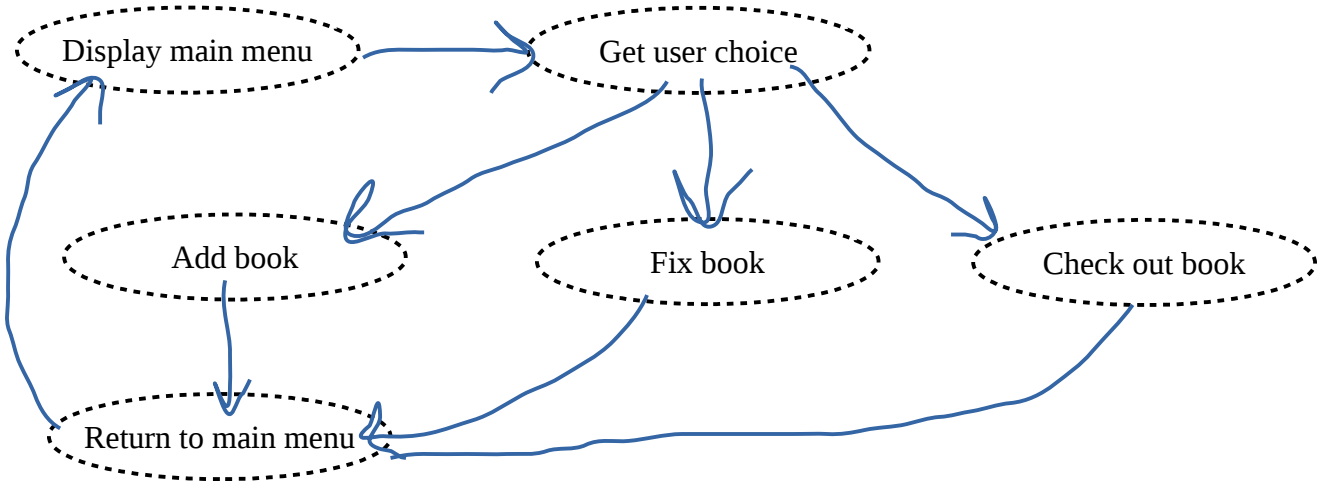
A common newbie mistake is to try to **implement the entire program all at once**.

This makes it harder to track down syntax errors, logic errors, and runtime errors!

Features

1. Display a main menu to help the user navigate the program
2. Be able to add a new book to the library
3. Be able to update an existing book's title
4. Be able to set a book as "checked out"

It might also help to come up with a flow-chart to structure out the program's flow before getting into the coding. A flowchart can be as in-depth or as simple as you'd like. For example:



"Weeks of programming can save you hours of planning."
-- unknown

Strategy 3: Figure out what to tackle first

After you've fleshed out the features of the program, figure out what to tackle first. Usually, this might be something like the main menu or other code that is required in order to implement the other features of the program.

```
public void addBook()
{
    System.out.println(
    "Add: Not yet implemented" );
}

public void updateBook()
{
    System.out.println(
    "Update: Not yet implemented" );
}
```

You could also partially implement each feature, leaving a shell for each feature (like a “to be implemented” message) just to get all the core methods set up and the program flow working. Then you could test it to make sure the program follows your flow-chart, and from there implement each feature in more detail.

Strategy 4: Write, Build, Test

As you're building your program, make sure you **build/compile regularly**. Perhaps every micro-feature! Did you write some code to get the user input? Build. Did you write a loop to validate their input? Build. *Build build build build!*

It is must easier to find **syntax errors** – which the compiler will find for you because it's cool like that – when you're building often, after every few changes. Take advantage of the compiler!

Second, **test after every feature, or portion of a feature!** It is important to guard against **logic errors** and **run-time errors** by testing small portions of the program at a time.

For example: Maybe your “addBook” method keeps overwriting the *first book* instead of adding new books. If you catch it ***while*** writing the addBook method, you can fix it no problem.

If you wait until the program is all written, you'll have to check *all methods* to try to pinpoint why your program isn't adding new books!

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

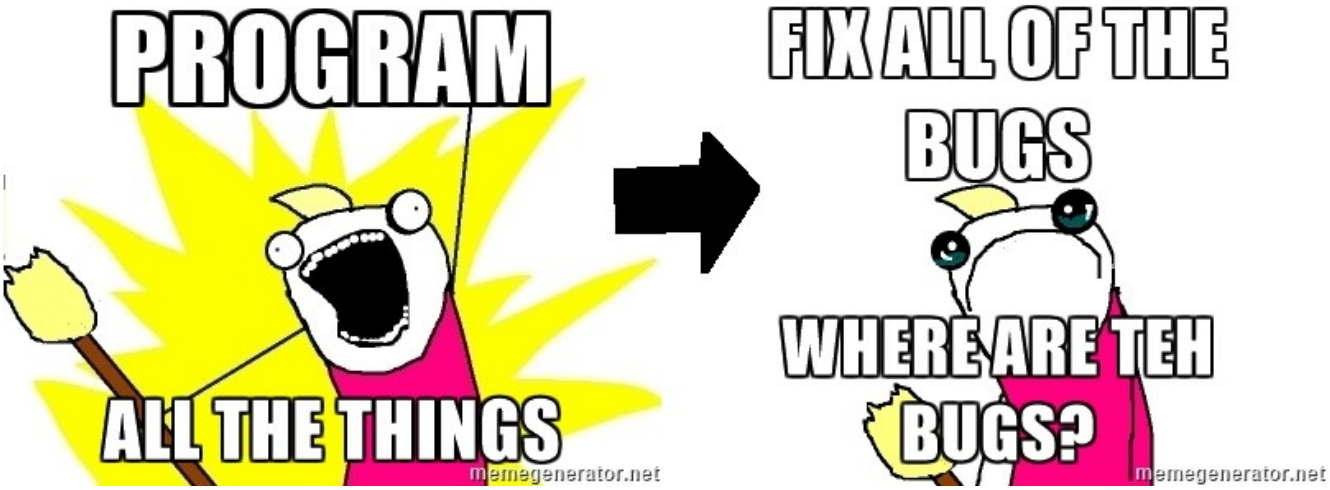
The compiler can easily detect these errors, and you will receive a compile-time error.

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.

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.

Strategy 5: Only work on one feature at a time

Never jump into your program by implementing everything at once, **no building, testing, nada!**



This is a common, and **very avoidable** mistake that new programmers make. You will have a much smoother time with building the features of your program, debugging, and avoiding errors in general if you take it a step at a time.

Never write the entire program all at once, without doing any compiling or testing!

Future-you is your friend. Be nice to future-you. Don't treat future-you like poop by not testing your code. Otherwise, future-you will hate past-you. Future-you will never forgive past-you.

"Inside every well-written large program is a well-written small program."
-- Charles Antony Richard Hoare, computer scientist

Strategy 6: Be patient

Programming is about problem solving. It can take time to figure out the solution to a problem or build beautiful code, but it is worth it.

Remember that everybody gets stuck, everybody gets frustrated, everybody runs into bugs and problems. That's just the nature of programming.

Nobody is born speaking C++. Programming is a skill that we learn and continue learning throughout our lives.

If you keep working at it, you'll notice some level of *fluency* develop. Just like knowing how to conjugate your verbs and use the proper prepositions in your human language, you'll eventually find yourself able to visualize problems and solutions in your mind before writing them on the computer.

There will always be challenges, but the more you practice, the better you get.

Strategy 7: Stay ahead of schedule

Sometimes, you have to give it a break – working while burnt out doesn't help your problem-solving skills. Sometimes, you will figure out the solution to your problem while baking an apple pie or while watching [insert timely popular Netflix show here].

If you wait until the last minute, you don't give yourself that break-time that you might need after a difficult problem. By continuing to program when you feel like you're hitting your head on a wall, you're not going to get much done, and your code might turn out worse.

Make sure to give yourself adequate time to work on the program so that you can give yourself a breather, have time to ask questions and do research, and get clarifications about the requirements.