10 OBJECTIVES

Students are introduced sequentially to each new programming construct. We call these stages “objectives” so that the language of learning is familiar.

Each objective builds-on and uses the keywords only introduced in the current and previous objectives to ensure a smooth and gradual learning transition between the objectives for students.

With a functions first approach, inputs are introduced much later than you might expect in favour of structured programming, arguments, and parameters.

Objectives:

1. Learn how to write structured programs.
2. Learn how to use selection.
3. Learn how to use number data types.
4. Learn how to use string data types.
5. Learn how to use counter-controlled iterations.
6. Learn how to use condition-controlled iterations.
7. Learn how to handle user inputs.
8. Learn how to use arrays and lists.
9. Learn how to use serial files.
10. Learn how to master the basics.

Objective 10 is an ever-increasing set of problems with a new one being released periodically.

Students will probably have studied most of these concepts at Key Stage 3, but these resources will not be too easy for them! The level of rigour is higher than that expected at Key Stage 3, especially with a functions first approach and validation of inputs. Repeating known concepts and consolidating knowledge firmly is extremely helpful. Even if students used these resources at GCSE, at A level they should attempt them again from objective 1 using a different language. E.g. Python at GCSE and C# at A level. Understanding the similarities and differences between different procedural languages is beneficial to learners wanting to study Computer Science beyond school.

4 STAGES IN EACH OBJECTIVE

Each objective has four stages for students to complete.

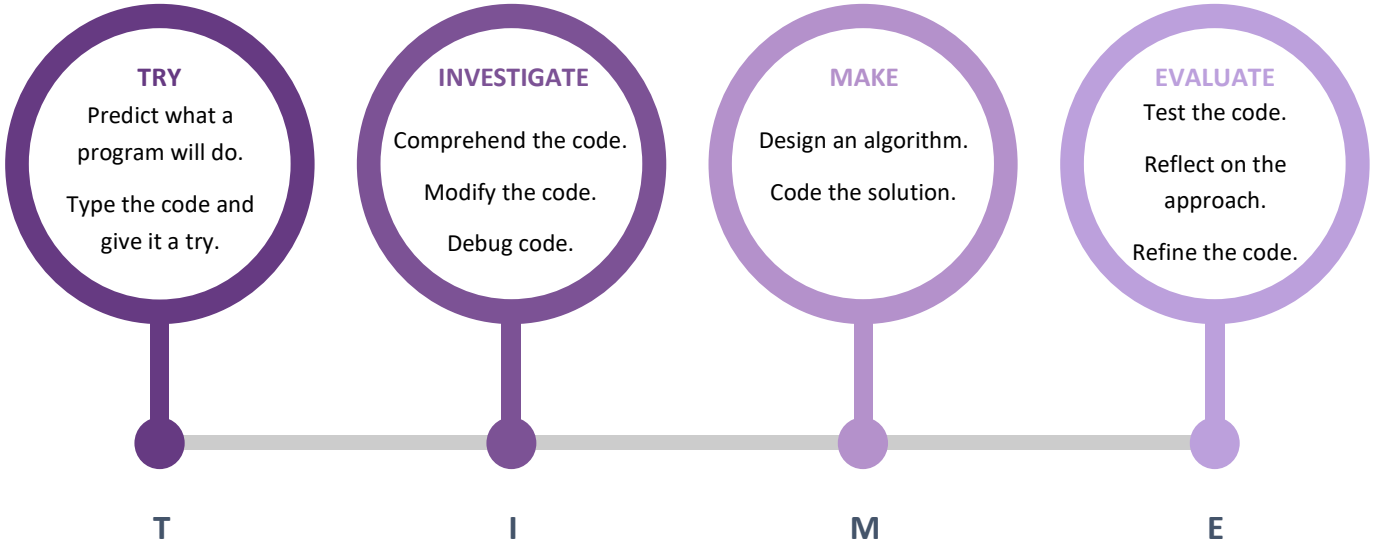
T - Try

I - Investigate

M - Make

E - Evaluate

Craig'n'Dave call this their "TIME" approach to programming. It is based on the PRIMM model proposed by Sue Sentence.



WORKBOOKS

Each objective is presented in a PowerPoint student workbook. Activities for the students are written in the notes section of each slide.

Using PowerPoint enables the teacher to project slides if that is useful and output them in a variety of formats. They also work well with learning platforms, e.g. Google Classroom.

Try stage

Students look at a coded solution and predict what the output will be. Students type in the code to see if they are correct. This approach aims to simulate how self-taught programmers in the 1980s used code listings in magazines and programming manuals to learn to code. Not starting from a blank screen is less daunting for students, provides models and guides student practice.

Investigate stage

Students learn how the sample programs work, understanding the new commands introduced. A set of small tasks instruct students to modify the sample programs they have been given. At the end of the stage keywords and learning points are summarised as a handy reference. The syntax of related additional commands is also presented that were not needed in the sample programs but could be useful to students when writing their own programs. These also include all the commands listed in examination board specifications.

A set of progressive comprehension questions also provide students with an opportunity to consider what terms mean, what commands or blocks of code achieve, how they work and why they are used. A deeper knowledge of syntax and programming constructs allows students to adapt to new commands more easily. This is based on a simplified student-friendly version of the block model of program comprehension proposed by Carsten Schulte.

Make stage

Problems are presented in a mixture of written English, flowcharts, pseudocode, Parsons problems and output focused. Problems increase in difficulty indicated by the number of points. Students are encouraged to use whitespace, indentation, sensible variable names, and subroutines from their very first program.

Students should be using comments throughout their programs to explain the purpose of subroutines, variables, selections, and iterations. This will provide adequate problem decomposition making the explicit design of an algorithm using pseudocode redundant.

Back in the history of Computer Science when IDEs did not exist, programs were written on tape or card and it took a day to execute within a queue of other programs, it made sense to design algorithms to ensure they were robust! This is no longer necessary to become a good programmer. The small bite-sized programs used to teach programming are not large enough to warrant a design stage.

If students struggle to see solutions to the problems and therefore struggle with problem decomposition, they could produce all their comments first and fill in the code required between the comments afterwards, much like using pseudocode.

We suggest students choose the problems they want to solve, aiming for a total of 6 points in each objective. This provides for student choice and differentiation. Students who find programming more difficult could achieve 6 points from: 1 + 1 + 2 + 2-point problems. More able students could achieve 6 points from: 3 + 3-point problems.

Evaluate stage

Objectives will frequently require students to create and use test tables to test their solutions to the problems. This encourages good practice and teaches the importance of robust code.

Once students have finished an objective, they should alert their teacher. This is an opportunity to have a conversation with the student and give oral feedback on the problems attempted. Immediate oral feedback will be far more useful to the student than written feedback.

See overleaf for a framework for feedback conversations.