Unit IV: Looking Forward: Careers & Technology

Chapter 17: Programming Concepts

17.1: Goal

In this chapter, you will be introduced to the fundamental concepts of computer programming using Alice, a visual programming environment. You will learn how programs are created using objects, events, methods, variables, and control structures while developing logical thinking and problem-solving skills. Understanding these concepts will give you insight into how software is created and prepare you for future programming experiences.

17.2: Objectives

- Explain what a computer program is.
- Describe the steps used to solve problems computationally.
- Explain the purpose of algorithms.
- Differentiate between pseudocode, flowcharts, and programming languages.
- Describe variables, data types, and basic operations.
- Explain how decisions and repetition control program execution.
- Recognize the importance of testing, debugging, and documentation.
- Describe how AI is changing software development.
- Create a simple program in Alice

17.3: Think About It

Every day, you follow instructions to complete tasks such as cooking a meal, assembling furniture, or driving to work. What happens if the instructions are incomplete, out of order, or contain mistakes?

Computers face the same challenge. They can perform amazing tasks, but only when given clear and logical instructions.

17.4: Opening Scenario

A local animal shelter wants a simple computer program to track pets waiting for adoption. Volunteers need to enter information about each animal, search for available pets, and update records when an adoption occurs. Before anyone writes code, the developers must determine exactly what the program should do. They create step-by-step plans, organize information, test their ideas, and revise the program until it works correctly. This process illustrates that programming is much more than writing code: it is solving problems logically and communicating those solutions to a computer.

17.5: What Is Programming?

Programming is the process of creating instructions that tell a computer what to do. These instructions,

called **code**, are written using a programming language that both programmers and computers can work with. Programs are behind nearly everything we do with technology—from mobile apps and websites to video games, banking systems, medical equipment, and artificial intelligence.

Computers do not independently understand a goal such as “calculate the average grade” or “move a character across the screen.” A programmer must break the task into a series of clear, logical steps. This step-by-step procedure for solving a problem is called an **algorithm**. The algorithm is then translated into instructions the computer can execute.

Programming Languages

A **programming language** provides rules and commands for communicating instructions to a computer. Different languages are designed for different purposes. Python is widely used for data analysis, automation, and artificial intelligence; JavaScript is commonly used to create interactive web pages; and languages such as Java and C++ are used to develop many types of software. C# (C Sharp) is used to create Windows programs with multiple-forms and a graphical user interface.

Some programming environments use visual tools rather than requiring beginners to type every command. **Alice 3**, for example, allows students to create programs by dragging statements into the code that control characters and objects in a 3D world. Students learn important programming concepts such as top down design, variables, control structures, decisions, repetition, procedures and function, and events.

Programming Is Problem-Solving

Learning programming is not simply learning commands. Much of programming involves **problem-solving and logical thinking**. Programmers first determine what they want a program to accomplish, divide the problem into smaller tasks, create instructions for those tasks, and then test the program.

Programs rarely work perfectly the first time. Finding and correcting errors in a program is called **debugging**. Debugging is a normal part of programming and helps programmers understand how their instructions are actually being interpreted and executed.

From an Idea to a Program

Most programs begin with a simple question: **What do I want the computer to do?** The programmer then develops a solution, writes or assembles the instructions, tests the program, and makes corrections as needed. Whether students eventually become programmers or not, learning basic programming helps develop **logical thinking, creativity, persistence, and problem-solving skills** that are useful in many careers.

17.6: Why This Chapter Matters

Even if you never become a software developer, programming concepts help you think logically, solve problems, and better understand the technology you use every day.

Many careers—including healthcare, business, education, engineering, public safety, and manufacturing—benefit from computational thinking. Understanding how software works also helps people use AI tools more effectively and communicate with technical teams.

17.7: Computational Thinking and Problem Solving

Computational thinking is a problem-solving approach that involves breaking a complex problem into smaller, manageable parts and developing a logical process for solving it. Although the term is closely associated with computer programming, computational thinking can be used to solve problems in many areas of everyday life and work. It focuses on understanding the problem before trying to write the program.

Four important elements of computational thinking are **decomposition, pattern recognition, abstraction, and algorithm design**.

Decomposition

Decomposition means breaking a large problem into smaller problems that are easier to understand and solve. For example, suppose you want to create an Alice animation in which a character walks across a room, picks up an object, and returns to the starting position. Instead of treating this as one complicated task, you can divide it into smaller tasks:

. Move the character toward the object.
2. Have the character reach for the object.
3. Pick up the object.
4. Turn around.
5. Return to the starting position.

Each smaller task can be developed and tested separately.

Pattern Recognition

Pattern recognition involves identifying similarities or repeated elements in a problem. Recognizing patterns can help programmers reuse solutions rather than solving the same problem repeatedly.

For example, if several characters in an animation need to perform the same greeting, the programmer may recognize that the actions—turning toward another character, waving, and speaking—follow the same pattern. Instead of creating completely different instructions each time, the programmer can develop a reusable solution.

Abstraction

Abstraction means concentrating on the important information needed to solve a problem while ignoring details that are not currently relevant.

Imagine programming a car to move through an Alice scene. You may need to know the car's location, direction, and speed. You probably do not need to know the color of its seats or the shape of its headlights. Abstraction helps programmers focus on the information that affects the solution.

Algorithm Design

An **algorithm** is a step-by-step procedure for completing a task or solving a problem. After a problem has been broken into smaller pieces and the important information has been identified, the programmer can develop an algorithm describing the actions the computer should perform.

For example, an algorithm for determining whether a student passed a course might be:

1. Obtain the student's final grade.
2. Compare the grade with the minimum passing grade.
3. If the grade is equal to or higher than the minimum, report **Pass**.
4. Otherwise, report **Not Pass**.

Algorithms can be written in ordinary language, represented with a **flowchart**, or expressed as **pseudocode** before they are converted into programming instructions.

Testing and Refining a Solution

Problem solving does not end when the first solution is created. Programmers **test** their programs to determine whether they work as expected. When a problem is discovered, they identify its cause, make a correction, and test again. This process of finding and correcting errors is called **debugging**.

Programmers may also refine a working solution to make it clearer, faster, easier to maintain, or easier for someone else to understand.

Think Like a Programmer

Computational thinking can be summarized with four questions:

Can I break the problem into smaller parts?
Do I recognize any patterns?
Which details are important?
What steps will solve the problem?

Learning to ask these questions is one of the most valuable parts of learning programming. The goal is not simply to tell a computer what to do—it is to learn how to approach a problem **systematically and logically**.

17.8: Algorithms, Flowcharts, and Pseudocode

Before writing a program, programmers often plan how the program will solve a problem. Three useful tools for planning a solution are **algorithms, flowcharts, and pseudocode**. Each provides a different way to organize the steps that a computer will eventually perform. Planning first can make programs easier to create, test, and debug.

Algorithms

An **algorithm** is a precise, step-by-step procedure for completing a task or solving a problem. We use algorithms in everyday life even when we do not call them algorithms. A recipe, driving directions, and instructions for assembling furniture are all examples of procedures consisting of ordered steps.

For example, an algorithm for calculating the average of two test scores could be:

. Get the midterm and final exam scores.
2. Add the scores together.
3. Divide the total by 2.
4. Display the average.

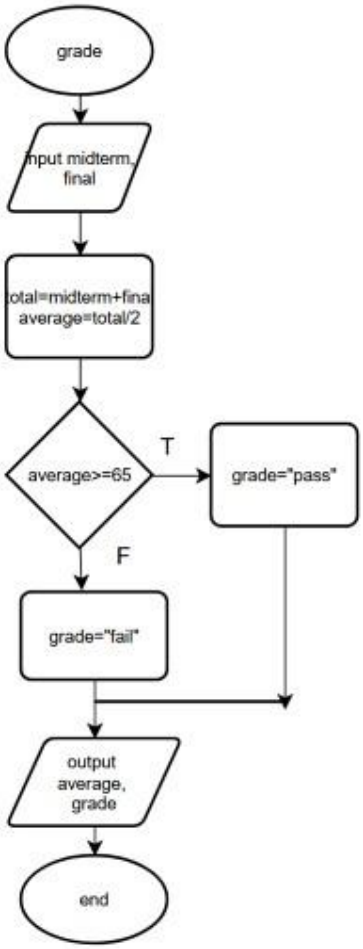
The **order of the steps matters**. A computer follows instructions exactly as they are given, so an algorithm must be clear, complete, and arranged in the correct sequence.

Pseudocode

Pseudocode is an informal way of writing an algorithm using words and statements that resemble programming instructions. It is designed for people to read, so it does not have to follow the exact syntax of a particular programming language.

For example, an algorithm for calculating the average of two test scores could be:

. input midterm and final exam scores.
2. total=midterm + final
3. average=total/2
4. output average.



Flowcharts

A **flowchart** is a visual representation of an algorithm. It uses standardized shapes connected by arrows to show the sequence in which actions and decisions occur. Flowcharts can be particularly helpful when a program contains decisions or several possible paths.

Common flowchart symbols include:

Symbol	Purpose
Oval	Start or end of the algorithm
Rectangle	Process or action
Diamond	Decision or question
Parallelogram	Input or output
Arrow	Direction of program flow

We have added displaying "pass" or "fail", so a flowchart is useful here.

For example, a program might ask for a student's grade and determine whether the student passed. A **diamond** could contain the decision *Grade ≥ 65?* One arrow would represent **True** and lead to the action for a passing grade, while another would represent **False** and lead to a different action.

Three Ways to Represent the Same Solution

An algorithm, flowchart, and pseudocode can all describe the same process. The difference is how that process is represented. An **algorithm** describes the steps, a **flowchart** shows the steps visually, and **pseudocode** expresses the steps in a form similar to programming code.

Once the solution has been planned, the programmer can translate it into a programming language or construct it using a visual programming environment such as Alice. Planning the logic before programming often makes it easier to identify missing steps and mistakes before they become part of the program.

17.9: Variables, Data, and Operations

Computer programs work with **data**. A program might use numbers, words, dates, images, or other types of information. To work with this information, programmers need ways to store data, identify what kind of data it is, and perform operations on it. Three fundamental programming concepts are **variables, data types, and operations**.

Variables

A **variable** is a named storage location that holds a value while a program is running. The value stored in a variable can change as the program executes.

For example, a program that keeps track of a student's score might use a variable named score:

```
score = 85
```

The "=" assigns a value to the variable. Later, if the student earns five additional points, the program could change the value: `score = score + 5`

The variable score is assigned a new value equal to its current value plus 5; score would now contain the value **90**.

Variables should have meaningful names whenever possible. A name such as studentAge is easier to understand than a name such as x. Descriptive variable names make programs easier to read, debug, and maintain.

Data Types

A **data type** identifies the kind of information being stored. Programming languages may use different names for data types, but common types include:

Data Type	Description	Examples
Integer	Whole numbers	5, 27, -10
Decimal/Real/Double	Numbers containing decimal values	3.14, 72.5
String	Text or a sequence of characters	"Hello", "Maria"
Character	A single letter, number, or symbol	"A", "?"
Boolean	A value representing true or false	true, false

The data type matters because it determines what operations can be performed on the data. For example, numbers can be added mathematically, while strings can be joined together to create longer pieces of text.

Arithmetic Operations

Programs frequently perform mathematical calculations. Common **arithmetic operators** include:

Operator	Operation	Example

+	Addition	5 + 3
-	Subtraction	10 - 4
*	Multiplication	6 * 2
/	Division	20 / 4

A program calculating the cost of several items might use:

```
totalCost = price * quantity
```

The computer calculates the result and stores it in the variable totalCost.

Comparison Operations

Programs also need to compare values in order to make decisions. A **comparison operator** compares two values and produces a Boolean result—either **true** or **false**.

Examples include: `age >= 18` `score == 100`

```
temperature < 32
```

The expression age >= 18, for example, asks whether the value stored in age is greater than or equal to 18.

Logical Operations

Logical operators allow programmers to combine conditions. Three common logical operations are **AND, OR, and NOT**.

- **AND** requires both conditions to be true.
- **OR** requires at least one condition to be true.
- **NOT** reverses a true or false value.

For example: age >= 18 AND hasLicense == true

Both conditions must be true for the entire expression to be true.

Variables in Alice

In Alice, variables can store information that an animation or program needs while it runs. For example, a variable might keep track of a **score, number of attempts, speed, distance, or whether an event has occurred**. Objects can also have properties that describe information about them.

Students may encounter variables while creating animations, games, and interactive stories. A game could begin with: `score = 0`

Each time the player completes a task:

```
score = score + 1
```

Although Alice provides a visual programming environment, the underlying idea is the same as in text-based programming languages: **data is stored, changed, compared, and used to control what the program does.**

17.10: Decisions and Repetition

Programs become much more useful when they can **make decisions and repeat actions**. Instead of always performing exactly the same sequence of instructions, a program can choose what to do based on a condition or perform an action several times. These concepts are called **selection** and **repetition**, and they are fundamental to nearly every programming language.

Making Decisions with If Statements

A program makes a decision by evaluating a **condition**. A condition is an expression that produces a result of either **true or false**.

An **if statement** tells the computer to perform an action only when a particular condition is true.

For example:

```
IF temperature < 32      DISPLAY "Freezing"  END IF
```

The program checks whether the temperature is below 32. If the condition is true, the message is displayed. If it is false, the program skips the instruction.

If/Else Statements

Sometimes a program needs to choose between two alternatives. An **if/else statement** specifies what should happen when a condition is true and what should happen when it is false.

```
IF grade >= 60      DISPLAY "Pass"  ELSE      DISPLAY "Not Pass"  END IF
```

Only one of the two actions will occur.

In an Alice program, a decision might determine what a character does:

```
IF distanceToDoor < 2      character opens door  ELSE      character moves toward door  END
IF
```

This allows the animation to respond to what is happening rather than simply following the same instructions every time.

Repetition and Loops

Repetition, also called **iteration**, occurs when a program performs a set of instructions more than once. A programming structure that controls repetition is called a **loop**.

Imagine an Alice character that needs to jump five times. Instead of giving the same command five times, the programmer could use a loop:

```
REPEAT 5 TIMES      character jumps  END REPEAT
```

Loops make programs shorter, easier to understand, and easier to modify.

Count-Controlled Loops

A **count-controlled loop** repeats instructions a specific number of times.

For example:

```
REPEAT 10 TIMES      DISPLAY "Welcome!"  END REPEAT
```

The program knows before the loop begins that the instructions will be performed ten times.

In Alice, a count-controlled loop can be useful for repeating movements, sounds, or other actions in an animation.

Condition-Controlled Loops

Sometimes a programmer does not know exactly how many times an action will need to repeat. Instead, the repetition continues **while a particular condition remains true**.

```
WHILE score < 10      continue game  END WHILE
```

The game continues as long as the score is less than 10. Once the condition becomes false, the loop ends.

Be Careful with Infinite Loops

A loop must eventually have a way to stop unless continuous repetition is intentional. An **infinite loop** occurs when the condition that controls a loop never becomes false.

For example:

```
WHILE score < 10

   DISPLAY score

END WHILE
```

If nothing inside the loop changes score, the condition may remain true forever. Thinking carefully about how and when a loop will end is an important part of programming.

Sequence, Selection, and Repetition

Three basic structures form the foundation of programming:

Structure	Purpose	Example
Sequence	Performs instructions in order	Move forward, turn, speak
Selection	Chooses between actions	If the door is closed, open it
Repetition	Repeats instructions	Jump five times

These concepts are sometimes called the **three basic control structures**. By combining sequence, selection, and repetition, programmers can create programs that perform complex tasks, respond to changing conditions, and interact with users.

In Alice, students can see these concepts come to life as characters **move in sequence, make decisions, and repeat actions**, making it easier to visualize how a program controls what happens. You can also specify **do together** to turn both the hip and knee at the same time to step.

17.11: Testing and Debugging

Programs rarely work perfectly the first time they are created. Programmers must test their programs to determine whether they behave as expected and correct any problems they discover. **Testing** is the process of running a program under different conditions to determine whether it works correctly. **Debugging** is the process of finding and correcting errors, or **bugs**, in a program.

What Is a Bug?

A **bug** is an error or problem that causes a program to behave incorrectly. A bug might cause a program to produce the wrong answer, perform an unexpected action, stop running, or fail to run at all. Programming errors are commonly divided into several categories:

Type of Error	Description	Example
Syntax Error	An instruction does not follow the rules of the programming language.	Missing punctuation or an incorrectly written command
Logic Error	The program runs, but the instructions produce the wrong result.	Multiplying two numbers when they should be added
Runtime Error	A problem occurs while the program is running.	Attempting to divide a number by zero

In a visual programming environment such as Alice, some syntax errors are prevented because students select and arrange valid programming statements. However, **logic errors can still occur**. A character might turn in the wrong direction, move too far, perform actions in the wrong order, or respond incorrectly to a condition.

Testing a Program

Testing should involve more than running a program once. Programmers try different values and situations to determine whether the program behaves correctly.

For example, suppose a program determines whether a student passed based on a minimum grade of 60. The programmer might test:

- 85: clearly above the passing grade
- 40: clearly below the passing grade
- 60: exactly at the boundary
- 59: just below the boundary

Testing values near important limits is called **boundary testing** and can reveal errors that ordinary test values might miss.

The Debugging Process

Debugging is often a step-by-step process:

. **Identify the problem.** Determine what the program is doing incorrectly.
2. **Locate the cause.** Find the instruction or logic responsible for the problem.
3. **Correct the error.** Modify the program.
4. **Test again.** Run the program to determine whether the correction worked.
5. **Check for new problems.** Make sure the change did not cause an error somewhere else.

This process may need to be repeated several times before the program works correctly.

Debugging in Alice

Testing is especially useful in Alice because students can **see the results of their instructions**. Suppose a character is supposed to wave. If the character rubs his belly instead, the student can clearly see that there is an error in the program.

Testing an Alice animation a little at a time can make errors easier to find. Rather than creating an entire animation and testing it only at the end, students can test each section as they build it.

Errors Are Part of Programming

Finding an error does not mean that programming has failed. **Testing and debugging are normal parts of software development.** Professional programmers spend a significant amount of time testing programs, investigating unexpected behavior, and improving their solutions.

Debugging also develops an important problem-solving habit: instead of simply asking **"Why doesn't it work?"**, programmers learn to ask **"What is happening, what did I expect to happen, and where do those two begin to differ?"**

17.12: Programming Today

Programming has changed significantly since the earliest computers, but its basic purpose remains the same: **giving computers instructions to solve problems and perform useful tasks**. Today, programming is used in nearly every field, including business, healthcare, education, entertainment, science, manufacturing, transportation, and government. Programming is no longer limited to professional software developers; people in many careers use programming, automation, and related tools to work with data and improve everyday tasks.

Programming Languages Today

Thousands of programming languages have been developed, although a smaller number are widely used. Different languages are suited to different kinds of tasks. **Python**, for example, is commonly used for data analysis, automation, scientific computing, and artificial intelligence. **JavaScript** is central to interactive websites, while languages such as **Java, C++, C#, Swift, and Kotlin** are used for applications, games, mobile devices, and other software.

Programmers often learn more than one language during their careers. The specific commands may change from one language to another, but fundamental concepts such as **variables, decisions, loops, methods, algorithms, and debugging** apply across many languages.

Visual and Low-Code Programming

Not all programming requires typing large amounts of code. **Visual programming environments** allow programmers to create programs by selecting, arranging, and connecting commands. Alice is an example of this approach.

Low-code and no-code platforms take this idea further by allowing people to create applications and automated workflows using graphical interfaces and prebuilt components. These tools can make software development accessible to people who are not professional programmers, although more complex applications may still require traditional programming skills.

Programming and Artificial Intelligence

Artificial intelligence is changing how software is developed. Generative AI tools can help programmers **explain code, suggest solutions, generate portions of programs, locate errors, create documentation, and learn unfamiliar programming concepts**.

However, AI-generated code is not automatically correct or secure. A programmer still needs to understand the problem, evaluate the suggested solution, test the program, and correct errors. This makes the programming concepts students learn, such as algorithms, variables, decisions, loops, testing, and debugging, important even when AI assists with writing the code.

Programming Is Everywhere

Many devices people use every day contain software. Cars contain programs that control numerous systems. Smartwatches monitor activity and display notifications. Stores use programs to manage inventory and process purchases. Hospitals use software to manage patient information and medical equipment. Streaming services use algorithms to organize and recommend content.

Programming is also important in careers that may not have **programmer** in the job title. Scientists may write programs to analyze experimental data, accountants may automate repetitive tasks, researchers may process large datasets, and artists may use programming to create interactive media.

Why Learn Programming Concepts?

Most students in an introductory computer course will not become professional programmers. However, understanding programming helps students understand **how software works and how computers follow instructions**. It also develops computational thinking, logical reasoning, creativity, and problem-solving skills.

As AI and automation become more common, knowing how to **describe a problem clearly, develop a logical solution, evaluate results, and recognize errors** may become even more valuable than memorizing the commands of a particular programming language.