| [Print](javascript:window.print()) | [Close](javascript:parent.window.close()) |

**Course Transcript**

Getting Started with Software Programming

**Understanding the basics of software programming**

| [1. The Evolution of Computers and Software Programming](http://xlibrary.skillport.com/courseware/Content/cca/sd_spff_a01_it_enus/output/html/course_transcript.html#t3) |

| [2. Software Development Life Cycle](http://xlibrary.skillport.com/courseware/Content/cca/sd_spff_a01_it_enus/output/html/course_transcript.html#t7) |

| [3. Programming Basics](http://xlibrary.skillport.com/courseware/Content/cca/sd_spff_a01_it_enus/output/html/course_transcript.html#t29) |

| [4. Program Development Tools](http://xlibrary.skillport.com/courseware/Content/cca/sd_spff_a01_it_enus/output/html/course_transcript.html#t9) |

| [5. Understanding Programming Basics](http://xlibrary.skillport.com/courseware/Content/cca/sd_spff_a01_it_enus/output/html/course_transcript.html#t33) |

**Introducing algorithms**

| [1. Algorithms Explained](http://xlibrary.skillport.com/courseware/Content/cca/sd_spff_a01_it_enus/output/html/course_transcript.html#t13) |

| [2. Structuring an Algorithm](http://xlibrary.skillport.com/courseware/Content/cca/sd_spff_a01_it_enus/output/html/course_transcript.html#t15) |

| [3. Creating an Algorithm](http://xlibrary.skillport.com/courseware/Content/cca/sd_spff_a01_it_enus/output/html/course_transcript.html#t17) |

The Evolution of Computers and Software Programming

Learning Objectives

*After completing this topic, you should be able to*

* *identify stages in the evolution of software programming languages*
* *recognize the main types of programming languages that evolved*

**1. The evolution of programming**

Computers aren't capable of thought in the same way that humans are. They can do only what they're told to do. A program is a set of step-by-step instructions that tells a computer what to do.

Programs are written in various programming languages, which differ in complexity and function. The language used to create a program depends on what the program will be used for, as well as other factors.

Programming languages have generally evolved with the goal of making it easier for developers to read and write programs. However, the easier a programming language is to read and write, the slower and larger it tends to be. Today, the goal is to create programs that are easy to write and modify, and that run faster and take up less space.

**Drill Down Home Page**

The evolution of programming languages can be described in four stages.

**Page 1 of 4: The first stage - binary code**

The only language that computers actually understand is binary code. Binary code consists of two numbers – 1 and 0. Using this language is known as binary arithmetic.

A computer's processor consists of millions of tiny electronic switches that are either on or off. In binary code, the number 1 represents an on state and the number 0 represents an off state. A typical instruction to a computer consists of a long string of 1s and 0s.

**Page 2 of 4: The first stage - binary code**

Binary language was developed to communicate with first-generation computers. These computers consisted of large components joined by cables. They were so huge that they filled entire rooms.

At first, scientists reprogrammed these computers by physically rearranging their cables and switches. Later they developed ways of rearranging the instructions given to a computer using binary code. This was much easier, and could make the computer behave in different ways.

**Page 3 of 4: The first stage - binary code**

Binary language isn't ideal for programming, for the following reasons:

* it's extremely tedious and time-consuming to write
* it's very easy to type the wrong value by mistake, and
* programs written in binary language can't be transferred between different processors

**Page 4 of 4: The first stage - binary code**

Because of the difficulties associated with binary language, many new programming languages have evolved. However, all instructions written in other languages have to be converted into binary code before a computer can understand them.

**Page 1 of 3: The second stage – assembly languages**

Assembly languages use short commands made up of abbreviated words, such as MOV and ADD, and hexadecimal numbers. Each of these commands can replace more than ten binary commands – and they're much easier to read, write, and modify than binary code.

Code

.model small  
.stack 100h  
  
.data  
msg     db     'Hello, world!$'  
  
.code  
start:  
        mov    ah, 09h   ; Display the message  
        lea    dx, msg  
        int    21h  
        mov    ax, 4C00h  ; Terminate the executable  
        int    21h  
  
end start

**Page 2 of 3: The second stage – assembly languages**

A processor retrieves data from a source such as a hard disk and temporarily stores it in a part of the processor called a register. The processor then edits the data and sends it back to the hard disk or to another location.

Assembly language can be used to tell the processor which data to retrieve, which register to store it in, how to manipulate it, and when to send it back. For example, this code tells the processor to move the hexadecimal number 09h into the register ah.

Code

.model small  
.stack 100h  
  
.data  
msg     db     'Hello, world!$'  
  
.code  
start:  
        mov    ah, 09h   ; Display the message  
        lea    dx, msg  
        int    21h  
        mov    ax, 4C00h  ; Terminate the executable  
        int    21h  
  
end start

Assembly language can be translated and converted into binary code by a special program known as an assembler. Because computers understand only binary code, each program written in assembly language has to be fed through an assembler before it can work.

**Page 3 of 3: The second stage – assembly languages**

Assembly language programs have the following disadvantages:

Code

.model small  
.stack 100h  
  
.data  
msg     db     'Hello, world!$'  
  
.code  
start:  
        mov    ah, 09h   ; Display the message  
        lea    dx, msg  
        int    21h  
        mov    ax, 4C00h  ; Terminate the executable  
        int    21h  
  
end start

* they run more slowly than binary language programs
* they take up more hard-disk space and use more memory than binary programs
* they're system-specific and therefore can't be transferred from one computer to another, and
* they're extremely tedious to write

**Page 1 of 4: The third stage – high-level languages**

High-level languages were developed to solve one of the major problems with assembly language – which is that assembly language programs aren't portable.  
  
High-level languages are also easier to understand than assembly language because they make use of recognizable words such as BEGIN, END, and IF.

Code

IF age < 12 THEN  
  Classification := "child";  
ENDIF

**Page 2 of 4: The third stage – high-level languages**

Just like assembly language, high-level languages can't be understood by processors. Programs written in these languages have to be translated into binary code by either *compilers* or *interpreters*.

Compilers translate large chunks of code into binary at a time. However, the act of compiling can be time-consuming.

Interpreters, on the other hand, translate code into binary line by line when a program is run, making it unnecessary to wait until all code has been translated. The disadvantage of interpreters is that program execution can be slow.

**Page 3 of 4: The third stage – high-level languages**

Various commonly used languages are examples of high-level languages:

* Java   
    
  **Code**  
  public class HelloWorld {  
    public static void main(String[] args) {  
    System.out.println("Hello, world!");  
    }  
  }
* Pascal   
    
  **Code**  
  program HelloWorld;  
  begin  
    writeln('Hello, world');  
  end.
* BASIC, and   
    
  **Code**  
  10 PRINT "Hello, world!"  
  20 END
* C   
    
  **Code**  
  main( ) {  
          printf("hello, world");  
   }

**Page 4 of 4: The third stage – high-level languages**

C is probably the most widely used programming language in history. This is because it can give the programmer access to hardware, and is easier to read and write than assembly language. C has evolved over time, and has laid the groundwork for derivative languages such as C++ and C#.

Code

main( ) {  
        printf("hello, world");  
 }

**Page 1 of 3: The fourth stage – fourth-generation languages**

Fourth-generation programming languages were developed to speed up the application development process by automating tasks and providing tools based on graphic user interfaces, or GUIs. One command in a fourth-generation language can replace up to 50 binary-code commands.

Many fourth-generation languages are designed for specific business purposes. For example, Structured Query Language – or SQL – is used to work with data in databases. For example, the SQL command shown can be used to retrieve the names of all full-time employees from a database.

Code

Select \*  
From Employees  
Where Full\_Time = 'Yes'

Fourth-generation languages make more frequent use of words found in human languages than earlier programming languages. For example, you can use the word Select when retrieving specific records from a database using SQL.

Code

Select \*  
From Employees  
Where Full\_Time = 'Yes'

**Page 2 of 3: The fourth stage – fourth-generation languages**

These are two widely used examples of fourth-generation languages:

* SQL, which is used for database queries, and   
    
  **Code**  
  Select \*  
  From Employees  
  Where Full\_Time = 'Yes'
* ColdFusion, which is used for web development   
    
  **Code**  
  <cfset message = "Hello World!">  
  <cfoutput>#message#</cfoutput>

**Page 3 of 3: The fourth stage – fourth-generation languages**

Various GUIs for generating code using fourth-generation languages are available. For example, you can use a report painter to specify the data required from a database and click a single button to generate the code necessary to compile a report.

Fourth-generation languages also make use of libraries that provide code for common functions.

Question

Sequence the stages in the evolution of software programming languages.

**Options:**

1. Binary code
2. Assembly language
3. High-level languages
4. Fourth-generation languages

Answer

**Correct answer(s):**

**Binary code is ranked**

Binary code consists of 1s and 0s. It was the very first programming language, and is still the only language that a computer processor can understand.

**Assembly language is ranked**

Assembly language uses short commands made up of abbreviated words and hexadecimal numbers. It's simpler for people to use than binary code, and so represents a second stage in the evolution of programming.

**High-level languages is ranked**

High-level languages were developed to make programs portable, and to simplify programming by including recognizable words in code. They overcame some of the limitations associated with assembly language and so represent a third stage in the evolution of programming.

**Fourth-generation languages is ranked**

Fourth-generation languages enable programmers to generate code using high-level commands and, in some cases, a GUI. They represent the fourth, or current stage, in the evolution of programming.

**2. Programming languages**

Many programming languages have been developed over the years, partly because different development situations call for different languages.  
  
For example, JavaScript is best when you want to implement interactive features or add functionality to a web site, whereas Java is best when you need to develop a program quickly and easily. The choice of programming language also depends on the system that will be used and the types of tasks that a program must perform.

Specialized programming languages can be divided into five categories:

**high-level languages**

There are several specialized languages within the category of high-level languages. One of these is FORTRAN, which stands for Formula Translator. This language is best used for creating programs for mathematical or scientific work. Another example is COBOL, which stands for Common Business-Oriented Language and is used primarily in commerce.

**Rapid Application Development – or RAD – languages**

RAD languages were developed to enable developers to create powerful graphic user interfaces, or GUIs, with objects like mouse pointers, windows, scrollbars, and buttons. Examples of RAD languages are Visual Basic, Delphi, and Visual C#.

**web programming languages**

Web programming languages assist programmers in creating web sites. Web programming languages such as PHP and JavaScript enable developers to add interactivity or functionality to web sites.

**database programming languages, and**

Database programming languages were developed to fulfill one of the core functions of computers, namely systematic storage of data. Database languages such as SQL can be used to manipulate, retrieve, and store data in a database.

**scripting programming languages**

As users have become more knowledgeable and demand more from software, scripting programming languages were developed. They enable users to modify or build features in existing applications.  
  
Scripting programming languages are based on high-level languages, but aren't exactly the same and typically take longer to master. An example of a scripting language is Visual Basic for Applications, or VBA, which is supported by most Microsoft programs.

The evolution of programming languages has primarily been driven by technological improvements in computers. These improvements have been made in leaps and bounds since the first computer was designed in the 1940s.

The evolution of computers can be described in five stages:

**first generation**

The first generation of computers began in 1945 with the invention of the Electronic Numeric Integrator and Calculator, also known as ENIAC. This computer weighed 30 tons and failed once every seven minutes. Computer programming was first invented to work with the ENIAC.

**second generation**

The second generation of computers began in 1948, when AT&T Bell Telephone Labs developed transistors that could replace the vacuum tubes used in ENIAC. Computers became more widely available, which prompted the development of better programming languages.

**third generation**

The third generation of computers began with the invention of integrated circuits, which grouped all the essential components of a computer into one unit. Computers in this generation took up less space and were more reliable than earlier computers.

**fourth generation, and**

The fourth generation of computers began in 1974, when the first microcomputers were invented. These computers were also known as personal computers, or PCs. The cost of computers decreased, which made them available to more people, and they also became much more powerful with the invention of semiconductor memory.

**fifth generation**

The fifth generation of computers began with chip technology – including technology associated with superconductors, quantum computing, and nanotechnology. It also involved the development of Very Large Scale Integration, or VLSI.  
  
Computers in this generation continue to get smaller and more powerful. They also display some signs of artificial intelligence. An example is games that let users play against the computer.

Question

Match each programming language to the corresponding category.

**Options:**

1. Delphi
2. JavaScript
3. SQL
4. VBA

**Targets:**

1. RAD languages
2. Web programming languages
3. Database programming languages
4. Scripting languages

Answer

*RAD languages are used to develop GUIs with objects such as buttons and scrollbars. An example is Delphi.*

*Web programming languages are used to add interactivity and functionality to web sites. An example is JavaScript.*

*Database programming languages are used to retrieve and manipulate data in databases. An example is SQL.*

*Scripting languages enable users to modify existing programs. An example is VBA.*

**Correct answer(s):**

Target 1 = Option A

Target 2 = Option B

Target 3 = Option C

Target 4 = Option D

**3. Summary**

Computers can only do what they're told to do. They only understand binary code. Assembly language consists of abbreviated words and hexadecimal numbers, but its programs aren't portable. High-level languages make programs portable and use recognizable words. Fourth-generation languages also use recognizable words. All programs must ultimately be translated into binary code before a computer can understand them, and this is done by compilers and interpreters.  
  
Many languages have evolved to fulfill different needs. The technological evolution of computers has also driven the evolution of programming languages. The first generation of computers were extremely large and unreliable, but each subsequent generation has improved on this through the invention of transistors, integrated circuits, and technology such as nanotechnology, artificial intelligence and VLSI.

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/sd_spff_a01_it_enus/output/html/course_transcript.html#top)

Software Development Life Cycle

Learning Objectives

*After completing this topic, you should be able to*

* *recognize the steps in the software development life cycle*
* *identify the differences between the waterfall and extreme programming software development methodologies*

**1. Development methodologies**

Creating a large software program is a complicated process. The size and intricacy of software, as well as the nature of the Information Technology industry, has made it necessary for this process to involve many people.

Various software development methodologies have been developed in an attempt to establish a systematic method for creating software that can be followed by all programmers.

It's advisable to adopt one of these methodologies and follow it throughout the lifetime of a software development project. Doing this helps programmers write software that not only works reliably but can be developed within a reasonable time.

Two very different software development methodologies have emerged as common approaches:

* the waterfall model, and
* extreme programming

The waterfall model divides the software development process into four distinct phases. The idea is that all focus is placed on only one phase at a time. Each part of a phase should be fully completed before a programming team moves on to the next phase.

The four phases in the waterfall model are

**analysis**

Analysis is the first step and involves analyzing requirements to determine exactly what a program is supposed to do.  
  
Customers specify what they require of the program. Programmers then interpret these requirements to create a comprehensive list of program specifications. Once the list is complete and customers and programmers are in agreement, the analysis phase is signed off and won't be revisited.

**design**

In the design phase, programmers plan how they're going to create a program. They decide which programming language, compiler, and other tools to use. They break down the program into sections that can be assigned to individual programmers to create. They also decide on the communication procedures they'll use to ensure completeness and consistency, and to prevent duplication of work.  
  
Once a program has been designed on paper and a deadline has been set, the design phase is complete.

**implementation, and**

The implementation phase is when a program is actually written. The programmers write code for the sections they've been assigned, following the plan created in the Design phase. Once all the sections have been written, they're put together to form the whole program. This is the end of the implementation phase.

**testing**

In the testing phase, the entire program is tested to check if it works properly. The programmers fix any problems they find and test again to ensure that the fixes haven't created new problems in other areas.  
  
The process of fixing and testing continues until all identifiable bugs have been fixed. The testing phase is then over. The program is complete and can be delivered to the customer.

The waterfall model looks good in theory, but in practice its rigidity often causes problems. If one phase is stalled or not completed properly, it has a serious knock-on effect on the following phases.

In particular a maintenance phase is missing from the waterfall model. It should be added to the waterfall model to make a more complete software development life cycle.

The extreme programming method was developed to address the problems associated with the waterfall model. In this method, the phases in the process are integrated instead of distinct and isolated. The process flows back and forth through the phases until a program emerges, complete and free of bugs.

Extreme programming includes four overlapping phases:

* Coding
* Testing
* Listening, and
* Designing

Extreme programming is an adaptive and people-oriented process. Because it's difficult for customers to envision all aspects and features of a program at the start, they're involved throughout the process.  
  
A program slowly evolves as the programmers work and communicate both with each other and the customer. The process constantly adapts according to the customer's needs.

The general process followed in extreme programming includes four main steps:

**the customer communicates requirements**

The customer defines initial requirements for the program. This is a starting point only – the requirements may change as the process continues.

**programmers build a prototype**

A small team of programmers creates a simple prototype version of the program as a model for the customer to study and comment on. This helps the programmers determine whether they've understood the customer's requirements and whether they're on the right track.

**the actual program is built, and**

The team of programmers creates the actual program, constantly testing it, communicating with the customer, and incorporating necessary changes.

**additional features are added**

Once the basic program is complete and working, the team may add additional features requested by the customer.

Question

Match each of the phases to the corresponding methodology. More than one phase may match to each of the methodologies.

**Options:**

1. Analysis
2. Design
3. Implementation
4. Coding
5. Listening

**Targets:**

1. Waterfall model
2. Extreme programming

Answer

*The four phases in the waterfall model are analysis, design, implementation, and testing.*

*The four overlapping phases in the extreme programming method are coding, testing, listening, and designing.*

**Correct answer(s):**

Target 1 = Option A, Option B, Option C

Target 2 = Option B, Option D, Option E

**2. Development life cycle**

Each of the software development methodologies has both advantages and disadvantages. Different models are followed in different organizations, and one model may be modified to suit different needs.

Although different models are used and these are constantly evolving and being adapted, the modern software development life cycle follows a similar course in most cases.

**Drill Down Home Page**

Typically, the modern software development life cycle includes five phases.

**Page 1 of 2: Analysis**

In the Analysis phase of the software development life cycle, developers list the tasks that software must perform. They study the problems to be solved, and their causes and effects.

**Page 2 of 2: Analysis**

The specific task that a program is required to perform is derived from a problem statement, which is a concise description of the problem that a customer needs to address. The software must provide a workable solution to this problem.

**Page 1 of 3: Design**

In the Design phase, a program is designed. The aim is to meet the goals identified during the Analysis phase in a logical, efficient, and economical manner.

The designers consider the existing software and hardware resources they can use to achieve the goal. If these are unsuitable, new software and hardware components may have to be designed.

The overall structure of the program is designed first. This is then broken down into smaller components of a manageable size, known as modules.

**Page 2 of 3: Design**

To ensure that everyone involved can understand the design and visualize the necessary components of a planned software program, it's necessary to model the design. Some of the common modeling methods include flowcharts and process diagrams.

**Page 3 of 3: Design**

Once the visual design of the program has been completed, the developers can continue the design process using pseudocode.

Code

If student's grade is >= 60   
    Print "passed"   
else   
    Print "failed"

Pseudocode is used to bridge the gap between algorithms and programming languages. It provides an outline of a computer program, written in a format that can be easily converted into programming statements.

Pseudocode isn't a specific programming language but is used to mimic any number of computer languages. It's especially useful if the software planners and programmers haven't yet decided which language to use to create software, or if they plan on porting the software to several languages or operating systems.

**Page 1 of 3: Implementation**

During the Implementation phase, a program is written in a particular programming language. Each language has different advantages and disadvantages. For example, FORTRAN is a good language to use for processing numerical data, whereas LISP is good for artificial intelligence applications.

Program code should perform the required tasks, and it must be readable so that other programmers can maintain it.

Programs are often written in separate modules, with each module performing a particular task. Each module must function properly, both independently and in relation to the rest of the program.

**Page 2 of 3: Implementation**

Computer programs often grow to be quite large and the code they contain may include a variety of dependencies.  
  
Because of this, the Implementation phase is often considered to have three functional steps that are cyclical. They are

* coding
* testing and debugging, and
* optimizing code

**Page 3 of 3: Implementation**

The coding step involves writing actual code.

Testing and debugging is on-going. Programmers frequently compile their code while they're creating it to ensure that it uses the proper syntax and functions correctly. This ensures that they detect small code errors before they become larger problems.

During testing and in the process of writing code, bugs are often found and then remedied on the fly. Programmers may also detect inefficiencies or identify more effective ways of making code work. So the process of optimizing code is also on-going.

**Page 1 of 2: Testing**

Once all components of a program have been written and integrated, the software development life cycle reaches the testing phase.

**Page 2 of 2: Testing**

The entire program is tested as whole to check if it runs smoothly and to detect any bugs. Any problems found are fixed and the program is tested again to ensure that the fixes didn't accidentally cause problems in other parts of the program.

This process of fixing and testing continues until all the bugs that can be detected have been fixed.

**Page 1 of 3: Maintenance**

During the maintenance phase, any errors and deficiencies in a program are identified and the necessary improvements are made, while preserving the program's integrity.

The use of design notes, well-commented code, and meaningful variable names can all contribute to easier maintenance of a program.

**Page 2 of 3: Maintenance**

The improvements and fixes made during on-going maintenance often mean that new versions of a program are released.

Programs can be developed in minor or sub-versions, each addressing particular bugs or inconsistencies. Or they can be upgraded in major versions that include new functionality and involve code rewrites. These major modifications are often designed to take advantage of new hardware or other compatible software and operating systems that have become available.

Version control is the process of tracking and controlling the major and minor versions of a program. An important aspect of software development is understanding when the different kinds of version updates are necessary.

**Page 3 of 3: Maintenance**

When a software program is released, it's usually packaged with certain software documentation that includes, for example, descriptions, comments, and help files.

The release of new versions of the program should then also be accompanied by the release of new documentation. In the case of minor updates, documents or release notes identifying any changes should be provided. Major upgrades may require completely updated software documentation, with new help files and comments.

Question

Sequence the phases in the software development life cycle.

**Options:**

1. Analysis
2. Design
3. Implementation
4. Testing
5. Maintenance

Answer

**Correct answer(s):**

**Analysis is ranked**

The Analysis phase is the first step and involves determining exactly what a program is supposed to do.

**Design is ranked**

The Design phase is the second phase. It involves determining how a program must be built to meet the goals set in the Analysis phase.

**Implementation is ranked**

Implementation is the third phase, when a program is written in a particular programming language.

**Testing is ranked**

Testing is the fourth phase, during which an entire program is tested to check that it works properly.

**Maintenance is ranked**

Maintenance is the fifth phase. During this phase, any errors and deficiencies in the program are identified and the necessary improvements are made.

**3. Summary**

Various software development methodologies have been developed. Two of the main approaches include the waterfall method and extreme programming. In the waterfall method, analysis, design, implementation, and testing are sequential and completely separate phases. In extreme programming, the phases of coding, testing, listening, and designing all overlap.  
  
A general software development life cycle includes a series of five phases – Analysis, Design, Implementation, Testing, and Maintenance.

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/sd_spff_a01_it_enus/output/html/course_transcript.html#top)

Programming Basics

Learning Objectives

*After completing this topic, you should be able to*

* *recognize the features of common programming methodologies*
* *recognize the features of a good program*

**1. Common programming methodologies**

As customers' demands for program functionality continue to increase, software designers look for methodologies that enable them to create cheap but efficient programs quickly.

Like natural languages, which develop in response to social needs, programming methodologies are developed in response to market needs.

Applications don't only have to perform to specification – they must also be developed in the shortest possible time on the smallest possible budget.

There are two common programming methodologies:

* structured programming, and
* object-oriented programming, or OOP

The structured programming approach was developed after it was discovered in 1966 that any program can be written using just three structures:

**sequence**

The sequence structure is the execution of a set of statements in order.

**selection, and**

The selection structure is the selection of one particular statement, depending on a particular program state.

**repetition**

The repetition structure is the repetition of a statement until a particular program state is detected.

With this discovery came the notion that program code can be thought of as a collection of related components – rather than as a long and linear sequence of commands that must be processed from beginning to end.

Distinctive components of an application can be re-used, making it unnecessary to repeat code. One core component might be used by a number of ancillary components.

Grouping components that perform similar functions also enables modular development of an application, with development of each component constituting a separate development step.

Third-generation programming languages are procedural languages, which means they process instructions step-by-step to arrive at a result.

Procedural languages are considered suitable for a structured programming approach because they support sequence, selection, and repetition, while still representing a linear structure.

OOP was developed in the 1980s. It takes structured programming's notion of modular components further by creating computer abstractions, or objects, to represent real-world objects. So OOP seeks to align the programming logic of an application with its associated real-world logic. This makes code easier to understand and debug.

Each object in OOP has a set of properties that describe it. For example a car\_aerial object might have extended or retracted as a property value.

Each object also has a number of operators or methods associated with it. For example, the car\_aerial object might have two methods – extend and retract.

OOP retains the modular development principle from structured programming to enable easy debugging. Modular development is useful for OOP and structured programming because program errors tend to be associated with a single module.

Structured programming and object-oriented programming can be used by themselves or in combination.

Using a combination of the methods can help overcome shortcomings associated with each of the separate methods.

Question

What are some of the features of structured programming?

**Options:**

1. Any program can be written using just three structures
2. Statements are processed one after the other, from the beginning of a program to the end
3. Each component constitutes a development step
4. Components of the program reflect real-world objects

Answer

***Option 1:****Correct. Structured programming is based on the theory that any program can be written using just three structures – sequence, selection, and repetition.*

***Option 2:****Incorrect. Structured programming overcame the limitation of viewing processing as having to occur in a linear fashion.*

***Option 3:****Correct. Structured programming enables developers to plan a project using the development of each component, or module, as a separate development phase.*

***Option 4:****Incorrect. OOP takes modularity further than structured programming by mapping components and their logic to real-world objects.*

**Correct answer(s):**

1. Any program can be written using just three structures  
3. Each component constitutes a development step

**2. Features of good programs**

Regardless of whether a structured or object-oriented approach is taken, programming involves providing a computer with step-by-step instructions.

A compiler processes the entire program and converts it into executable binary code. This can take some time.

Alternatively, an interpreter executes instructions sequentially – line by line – returning the results for each line as a program runs.

You can read the code for a very small program and follow its logic to determine how it works.

However, most programs are large and can therefore be difficult to decipher. If you aren't the original programmer, this task becomes even more difficult.

A good program has five features:

* it's complete
* it's efficient
* it's usable
* it's easy to maintain, and
* it's readable

A program is complete if all functionality planned in the design phase has been developed and is working properly.

Programs should be written efficiently and sensibly.

Different computers have different hardware specifications, and inefficient or poorly-written code that worked on one computer may produce unexpected results on another.

It's easy to write excessive amounts of code without taking into account how it will affect the performance and stability of a program.

Usable programs are those that the intended end users find easy to operate. Usability depends on factors such as ease of input, interface design, and speed. Informative error messages and help files can also improve usability.

Offline user documentation, such as a printed manual, is often preferred by users because it's easier to read than online help. However, online help files can be distributed and searched more easily, so they're commonly included with software.

Maintaining a program involves making changes to a program after its implementation.

A computer system seldom satisfies the needs of its users indefinitely – they'll soon think of ways to improve it. So any software should be easy to adjust and build on.

Programmers other than those who originally designed a program should be able to understand and modify it, provided they're familiar with the programming language used.

Readable code is easy to follow. You can use indentation to make code clearer. For example, a simple practice like indenting part of an IF statement to the right can make code easier to read.

Code

(indented version)  
IF age < 12 THEN  
     Classification := "child";  
ENDIF  
  
(unindented version)  
IF age < 12 THEN  
Classification := "child";  
ENDIF

When programs contain hundreds or even thousands of lines of code, indentation – or lack of it – will significantly affect readability.

Comments also increase the readability of a program. In all languages, you can ensure that text you add as comments won't be executed as actual code.

Designers insert comments, code hints, or remarks to explain sections of code. For example, you might use a comment to explain the purpose of a section of code, to note where a new approach was tested, or to indicate that a portion of the code can be disabled for testing.

Different languages use different tags to indicate comments. Also, some languages – such as C++ and Java – differentiate between single-line comments and comment blocks.

Code

// This is a comment on a single line.  
...MISSING CODE...  
/\* This is a comment of multiple lines.  
The compiler will ignore everything between the slash on the preceding line and the slash on this line. \*/  
...MISSING CODE...

It's important to know how to identify comments in the programming language you're using. In C++, for example, braces {} indicate a block of commands – whereas in PASCAL, they indicate comments.

Question

What are the features of a good program?

**Options:**

1. It's complete
2. It's efficient
3. It's usable
4. It's easy to maintain
5. It's code is easy for other programmers to read
6. Its code is as short as possible
7. It has a small file size

Answer

***Option 1:****Correct. A good program is one in which all functionality planned in the design phase has been developed and is working properly.*

***Option 2:****Correct. Coding should be efficient so that programs perform the required tasks quickly and are stable.*

***Option 3:****Correct. Usable programs are those that the intended end users find easy to operate.*

***Option 4:****Correct. A good program has code that's easy for any programmer familiar with the programming language to read and follow.*

***Option 5:****Correct. A good program has code that's easy for any programmer familiar with the language that was used to read and follow.*

***Option 6:****Incorrect. If the other characteristics of a good program are present, a program can be considered good even if its code is very long. Also, practices like adding explanatory comments to code are recommended even if they add to overall code length.*

***Option 7:****Incorrect. It isn't always necessary for a program to have a small file size. This will depend on how the program will be distributed and used.*

**Correct answer(s):**

1. It's complete  
2. It's efficient  
3. It's usable  
4. It's easy to maintain  
5. It's code is easy for other programmers to read

**3. Summary**

The first major programming methodology – structured programming – was developed in the late 1960s. It's based on three steps – sequence, selection, and repetition. OOP was developed in the 1980s as the second major programming methodology. It aims to align the programming logic of an application with associated real-world logic.  
  
Computer programs should be complete,efficient, usable, maintainable, and readable. You can help make code more readable by using suitable indentation and comments.

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/sd_spff_a01_it_enus/output/html/course_transcript.html#top)

Program Development Tools

Learning Objective

*After completing this topic, you should be able to*

* *recognize the functions of common program development tools*

**1. Tools for program development**

Programmers can use a range of development tools to simplify essential programming tasks.

An Integrated Development Environment, or IDE, is a set of tools for software development that's integrated in one package.

An IDE can have several functions:

* it can help you write and edit a program's source code
* it may display color-coded syntax for code
* it may display relevant error notifications while you type
* it can automate certain repetitive tasks
* it can help you browse through code modules
* it may compile code, and
* it may enable you to use drag-and-drop functionality to create visual features and connect to databases

Some IDEs provide templates for commonly used modules, objects and code blocks, and include context-sensitive shortcut menus that let you perform common tasks quickly. Some also include auto-complete functions, which can predict and complete functions and methods while you type.

A popular IDE is Microsoft Visual Studio, which supports among others, development in C#, Java, and Visual Basic. Visual Studio also enables you to integrate testing of an application into the development life cycle, which improves overall productivity.

Various tools are considered important in any software development cycle:

* editors
* compilers and interpreters
* a debugger
* help file generators
* an installation program
* a profiler, and
* binary, decimal, and hex converters

Editors may be simple text editors that are like word-processing applications but don't wrap words to the next line.

Source code can be saved in ASCII files and you can even use Microsoft Word to write these files. However, word processors usually include formatting features and other hidden characters that aren't needed in the source files.

Dedicated programming editors include additional features. For example, many code editors include line numbers that help you navigate through code. They may also offer automatic tabulation, color-coding of various elements, pop-up help information, and auto-complete functions.

The first editors that were used were line editors. These were simple editors that accepted one line of code at a time. This had advantages on systems with limited amounts of memory, but programmers weren't able to navigate through and change existing code. An example of this kind of editor can be found in UNIX.

Full-screen editors followed the line editors. These enable navigation and alteration of code. They're still in use today and are considered standard for software development.

Some of the features offered by dedicated editors are

**syntax auto-completion**

Some editors can recognize a programming language and suggest possible commands, methods, or functions to complete entries you've started to type. You can then select the correct option to complete a command or statement. For example, if you choose an IF-THENstatement, a generic structure is created for you.

**syntax highlighting**

Some editors highlight completed commands, methods, or functions to help you distinguish them from the rest of your code. This makes the source much easier to read, understand, and navigate.

**the ability to view and edit multiple files**

Some editors enable you to open a number of files that form part of the same project at the same time. You can then switch between the open files, editing and saving them.

**stacked undo and redo commands**

Most editors create a stack of changes that you've made and let you undo or redo these changes so you can test different code.

**support for macros, and**

Some editors enable you to create macros to automate tasks you perform often. For example if you want certain commands to be shown in uppercase, you can create a macro that will do this automatically.

**project management**

Large application projects usually consist of a number of files containing source code. Most editors have features that enable you to manage these files by showing them in a hierarchical tree structure. This organizes large projects and can help you manage and keep track of the files.

Computer hardware can understand only machine language that consists of a set of binary instructions. Programming languages have evolved to the point where they're very natural and very much like human languages. But they still have to be translated into machine language before computer hardware can use them.

To facilitate this, you can use either a compiler or interpreter to translate the program into machine language. Of the two, interpreters are less commonly used, although they still perform an important role in modern programming.

A compiler reads the entire source code and translates it as a whole into machine language. It then stores this in a single file, usually an executable or .exe file. At this point you can distribute the program to users who can simply run the file to use the program.

One reason that compilers are a more popular choice than interpreters is that the source code is hidden from users. Once a program is compiled you can use the program without needing the source code, but you'll have to recompile the program if you make any changes to the code.

It's important to note that a program compiled for a Windows-based system will not work on a Macintosh-based system. This is because the compiled machine language contains instructions specific to the microprocessor. If you need a Mac version of the program, you'll need to recompile it specifically for Mac.

An interpreter reads and converts code to machine language one line at a time. The interpreted code is stored in a computer's memory. Because computer memory is volatile, the interpreted code is discarded whenever the computer is shut down.

This means that whenever you want to run the program, the source code has to be re-interpreted. So when you want to distribute a program to users, you have to give them access to the source code. This is the main reason why interpreters are rarely used commercially.

Interpreters are popular in web programming, however. For example, web browsers use interpreters to read JavaScript. The reason for this is that not all web users use the same systems, so programs compiled in a specific machine language won't be accessible to everyone.

Debugging code involves removing errors from it. All errors must be removed before a program can be said to be complete.

The two main types of errors in code are

**syntax errors, and**

Syntax errors occur when a programmer writes code that doesn't conform to the rules of the programming language. These are like grammar errors in human languages.  
  
If a compiler or an interpreter detects a syntax error, it will cease operation and report the error.

**logic errors**

Logic errors are errors in the design of a program. They occur when a line of code conforms to the rules of the language, but the code is based on flawed logic.  
  
Logic errors are less obvious than syntax errors, so programs may compile or be interpreted without these types of errors being reported. The errors are often discovered only when a program runs and it performs incorrectly or halts unexpectedly.

A debugger runs a program one line at a time so that the programmer can examine it for errors. This is particularly useful for picking up and repairing logical errors.

Code should be readable and easy to understand so that it can be maintained. Comments can be written into the code to ensure that it meets these requirements.

You can use various methods to debug a program, one of which is called *stepping*. When you step through a program, each line is executed and you can also see which line of code is being read. If an error occurs, you can see which command caused it.

You can step through code at the source level, but also view the machine code for each line as it executes. Stepping line by line is a practical way of debugging small programs that don't contain a lot of source code.

For larger programs, you can use different methods to save time or to focus on particular parts of the source code.

You can use *breakpoints* to skip over parts of code you know to be working correctly. You add a breakpoint to the first part of code where you suspect the problem could lie. When you do this and run a debugger, the debugger skips over the first part of the program and halts at the first breakpoint. You can then step through the next section of the program line by line.

When you're debugging a large program with multiple subprograms, you can use two techniques to handle the source code more efficiently:

**step over, and**

When you mark a subprogram with a step over command, the debugger executes the subprogram without debugging it line by line. You can use this method when you're confident the subprogram is working correctly.

**step out**

When you're stepping through a subprogram line by line and want to stop, you can use the step out command to do so. The debugger will then immediately stop stepping through the subprogram and continue with the code that follows it.

The step method of debugging enables you to check the logical and working structure of a program. However, when you want to focus on checking the data used by a program, you can use the *watching* method.

The watching method enables you to check the variables used in the program and to change the variable data to manipulate or test the results.

This is especially helpful if a program is designed to handle and manipulate user data or calculate mathematical expressions.

Creating good help files is an essential step in the development of most software. Help files are generally more accessible than user manuals because they're modeled as miniature web pages. This makes information and solutions to problems much easier to find.

Programs similar to word processors can help in the creation of help files. These programs let you write, edit, organize, and format text, and create hyperlinks to different parts of a help file.

To help users set up and run the programs you create, you can use installation programs to create installers for software.

Programs often contain a number of different files, which, if bundled together, would prove cumbersome for users. You can use installation programs to specify exactly which files are needed by your program and where these files should be stored on disk. An installer configures the registry settings a user's computer will need to find and access all these files, and stores the files in the correct folders.

You can also use a program installer to display user license agreements and graphics promoting your company's brand to users.

You can use a *profiler* to optimize a program that consists of a number of parts. A profiler examines a program and identifies the parts of it that are used most frequently.

These are the parts that should be optimized.

Say a program accepts users' login credentials and then allows them to sort and search through a number of records. The login procedure will occur only once in a session but the search and sort procedures may be called a number of times. So it's the search and sort procedures that should be optimized.

Profilers use two methods to examine a program:

**sampling, and**

Sampling involves examining a program at fixed intervals and flagging the parts of the program that are used most often. These parts are called *hotspots*.

**instrumentation**

After sampling finds the hotspots, you use instrumentation to check each line of code in the hotspots in detail. You then concentrate on optimizing them to speed up the overall performance of the program.

Binary, decimal, and hexadecimal converters are often required during program development. For example, color references are often in hexadecimal format, so a converter is useful in design applications. Similarly, a converter that converts between binary and decimal numbers may be useful in scientific applications.

Graphic

*Binary: 11111010000000011000  
Decimal: 1024024.24  
Hexadecimal: FA018*

You can change the calculator bundled with Windows to Scientific view to access these types of converters. Similar tools are also available to Mac and Linux users.

Graphic

*The Windows calculator includes Hex, Dec, Oct, and Bin radio buttons.*

Question

Match each programming tool to its function.

**Options:**

1. Editor
2. Compiler or interpreter
3. Debugger
4. Help file generators
5. Profiler

**Targets:**

1. A program for typing instructions for computers
2. Converts programming languages to binary code or machine language
3. Helps find and remove errors
4. Provides guidance to users
5. Examines code for the purpose of optimization

Answer

*You use editors to type and edit code instructions, which you save to a file.*

*You use compilers and interpreters to convert programming code to binary or machine language so that it can be understood by computers.*

*You use a debugger to find and fix or remove erroneous code from a program.*

*You create help files to guide users in using a particular program.*

*You use a profiler to examine code to determine which parts are executed most often. You can then focus on optimizing these parts of the code.*

**Correct answer(s):**

Target 1 = Option A

Target 2 = Option B

Target 3 = Option C

Target 4 = Option D

Target 5 = Option E

**2. Summary**

Programmers can use a number of tools to simplify the development process. A programming environment that provides a set of these tools is called an IDE. You use an editor to write source code, and convert programming language to machine language using a compiler or interpreter. You can run a debugger to check code for errors and use a program similar to a word processor to create help files for users. You can also create an installation for a program and use a tool called a profiler to optimize frequently used parts of the program.

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/sd_spff_a01_it_enus/output/html/course_transcript.html#top)

Understanding Programming Basics

Learning Objectives

*After completing this topic, you should be able to*

* *demonstrate basic knowledge of computer languages, programs and programmer tools*
* *demonstrate knowledge of software methodologies and programming approaches*

**1. Exercise overview**

To perform this exercise you need to understand the basics of software programming.

This involves the following tasks:

* understanding computer languages, programs and programmer tools and
* understanding software methodologies and programming approaches

**2. Understanding programming basics**

Question

Computers have continued to evolve since they were first introduced. With this, programming languages have also evolved.  
  
Sequence the stages in the evolution of programming languages.

**Options:**

1. Binary code
2. Assembly language
3. High-level languages
4. Fourth-generation languages

Answer

**Correct answer(s):**

**Binary code is ranked**

Binary code, which consists of 1s and 0s, represents the first stage in the evolution of programming languages. It's the basic code that computers understand.

**Assembly language is ranked**

Assembly language consists of abbreviated words and hexadecimal numbers, and represents the second stage in the evolution of programming languages.

**High-level languages is ranked**

High-level languages represent the third stage in the evolution of programming languages. They use recognizable words, and they solved the non-portability problem associated with assembly language.

**Fourth-generation languages is ranked**

Fourth-generation languages represent the fourth stage in the evolution of programming languages. They contain recognizable words and can be used to generate code based on high-level specifications.

Question

Match each type of programming language to its description.

**Options:**

1. Rapid Application Development, or RAD languages
2. Web programming languages
3. Database programming languages
4. Scripting languages

**Targets:**

1. Used to create applications with graphic user interfaces
2. Used to create graphics and interactivity for web sites
3. Used to manipulate and retrieve data
4. Used by users to modify existing applications

Answer

*RAD languages such as Visual C can be used to create powerful graphic user interfaces – or GUIs – for applications.*

*Web programming languages such as JavaScript and PHP were developed to add graphics and interactivity to web sites.*

*Database programming languages such as SQL were developed for storing, manipulating, and retrieving data in databases.*

*Users can use scripting languages to add or modify features in an existing application.*

**Correct answer(s):**

Target 1 = Option A

Target 2 = Option B

Target 3 = Option C

Target 4 = Option D

Question

What are the features of a good program in any language?

**Options:**

1. It's complete
2. It's efficient
3. It's easy to use
4. It's easy to maintain
5. It's readable
6. It's basic
7. It's short

Answer

***Option 1:****Correct. A good program is a program that includes all the required functionality and that has been completely tested.*

***Option 2:****Correct. Programs should be written so that they perform efficiently. Poorly designed code can hamper performance.*

***Option 3:****Correct. Usable programs are those that the intended users find easy to operate.*

***Option 4:****Correct. Good programs are easy to maintain. So programmers can easily make changes to them after their implementation.*

***Option 5:****Correct. Good code is easy for other programmers to read and follow.*

***Option 6:****Incorrect. Complicated programs can still be considered good if they're complete, efficient, usable, maintainable, and readable.*

***Option 7:****Incorrect. If the other characteristics of a good program are present, a program can be considered good even if it's long.*

**Correct answer(s):**

1. It's complete  
2. It's efficient  
3. It's easy to use  
4. It's easy to maintain  
5. It's readable

Question

Match commonly used development tools with their functions.

**Options:**

1. Editor
2. Compiler or interpreter
3. Debugger
4. Help file creator
5. Profiler
6. Installer

**Targets:**

1. Used for typing code instructions into a computer
2. Converts a programming language to binary code
3. Helps identify incorrect code
4. Assists in creating user guides
5. Studies source code for the purpose of optimization
6. Creates a file users run to set up the program on their computers

Answer

*You can use an editor to type and edit code instructions into a computer, and also save the code to a file.*

*You use compilers and interpreters to convert programming code to binary to make the code understandable to the computer.*

*You use a debugger to identify and correct or remove code errors in a program.*

*You use a help file creator to generate help files to guide users in using a program.*

*You use a profiler to examine source code and determine which parts are executed most often. You can then focus on optimizing these parts of the code.*

*You use an installer to create a single file with a set of instructions and program files that will store a program in the correct folders on a user's hard drive.*

**Correct answer(s):**

Target 1 = Option A

Target 2 = Option B

Target 3 = Option C

Target 4 = Option D

Target 5 = Option E

Target 6 = Option F

**3. Understanding programming methodologies**

Question

Which of these are phases in extreme programming?

**Options:**

1. Coding
2. Testing
3. Listening
4. Designing
5. Implementation
6. Analysis

Answer

***Option 1:****Correct. Coding is the phase in extreme programming when programmers write the code.*

***Option 2:****Correct. Testing is the phase in extreme programming when programmers test a program and fix errors. This occurs at various stages throughout development.*

***Option 3:****Correct. Listening is the phase in extreme programming when programmers get feedback from the customer. It occurs at various stages throughout development.*

***Option 4:****Correct. Designing is the phase in extreme programming when programmers design the final program.*

***Option 5:****Incorrect. Implementation is a phase in the waterfall model, rather than in extreme programming.*

***Option 6:****Incorrect. Analysis is a phase in the waterfall model, rather than in extreme programming.*

**Correct answer(s):**

1. Coding  
2. Testing  
3. Listening  
4. Designing

Question

Sequence the phases in the waterfall model.

**Options:**

1. Analysis
2. Design
3. Implementation
4. Testing

Answer

**Correct answer(s):**

**Analysis is ranked**

Analysis is the first phase. It involves determining exactly what a program is supposed to do.

**Design is ranked**

Design is the second phase. It's when programmers plan how they're going to develop a program.

**Implementation is ranked**

Implementation is the third phase, when a program is actually written.

**Testing is ranked**

Testing is the fourth phase, when an entire program is tested to see if it works properly.

Question

Which step is often added to the waterfall model to make a more complete software development life cycle?

**Options:**

1. Debugging
2. Maintenance
3. Coding
4. Listening

Answer

***Option 1:****Incorrect. Debugging is part of the Testing phase in the waterfall model.*

***Option 2:****Correct. The Maintenance phase was added to the waterfall model to make a more complete software development life cycle.*

***Option 3:****Incorrect. Coding is a phase in extreme programming.*

***Option 4:****Incorrect. Listening is a phase in extreme programming.*

**Correct answer(s):**

2. Maintenance

Question

What are some of the features of structured programming?

**Options:**

1. Any program can be written using just three structures
2. Each component constitutes a development step
3. Statements are processed one after the other from the beginning of a program to the end
4. Components of the program reflect real-world objects

Answer

***Option 1:****Correct. Structured programming breaks all programming into three basic structures – sequence, selection, and repetition.*

***Option 2:****Correct. The development of each component of a program in structured programming can serve as a separate development step.*

***Option 3:****Incorrect. Structured programming overcomes the need to process an entire program from beginning to end.*

***Option 4:****Incorrect. Object-oriented programming, or OOP, uses objects that reflect real-world objects.*

**Correct answer(s):**

1. Any program can be written using just three structures  
2. Each component constitutes a development step

Question

What are some of the features of object-oriented programming – or OOP?

**Options:**

1. Programming logic is aligned with its associated real-world logic
2. Objects consist of properties and methods
3. Statements are processed one after the other from the beginning of a program to the end
4. The entire application is consumed by a compiler at runtime

Answer

***Option 1:****Correct. OOP takes modularity further than structured programming by mapping components and their logic to real-world objects.*

***Option 2:****Correct. Objects in OOP consist of properties that describe it and methods that perform actions.*

***Option 3:****Incorrect. Structured programming overcame the need to process an entire document.*

***Option 4:****Incorrect. Interpreters can test single objects within an application without consuming the entire program.*

**Correct answer(s):**

1. Programming logic is aligned with its associated real-world logic  
2. Objects consist of properties and methods

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/sd_spff_a01_it_enus/output/html/course_transcript.html#top)

Algorithms Explained

Learning Objectives

*After completing this topic, you should be able to*

* *recognize factors to consider when writing an algorithm using the top-down approach*
* *recognize how to structure algorithms*

**1. The top-down approach**

A computer is essentially an unintelligent machine. It can't perform tasks by itself – instead it simply processes sets of instructions. It steps through the instructions one at a time.

The set of instructions followed by the computer to resolve a particular problem is known as an *algorithm*.

A computer is a system that receives input, processes it, and generates output. To use an analogy, consider the process for making a cake. The problem to solve would be needing a cake for someone's birthday. You take an ingredient mix and bake it in an oven to create a cake.

Graphic

*Three part process:  
1. input  
2. process  
3. output*

The input in the baking system is the ingredient mix, the process is baking the ingredient mix in the oven, and the output is a cake. If the input or the process isn't correct, a suitable cake won't be made and the problem of needing a birthday cake will persist.

Graphic

*Three part process:  
1. input - ingredients  
2. process - bake  
3. output - cake*

The algorithm is the part that performs the processing – for example, combining the ingredients and then baking at 350 degrees for 40 minutes.

The algorithm for making cake is simple. However, an algorithm can become extremely complex if a system must be able to accept a lot of input, perform a complicated process, and return output. It might consist of thousands of lines of code.

The top-down approach to algorithm writing addresses this problem by segmenting an algorithm into manageable parts, or modules.

Each part is then further distilled into a number of subparts. This continues until the steps in the algorithm can't be further broken down into constituent steps.

There are a number of important points to remember when writing algorithms:

* you should include all the steps
* all input conditions should be addressed
* no step should be ambiguous, and
* steps should result in output at some point

First it's crucial to ensure that you include all the required steps in an algorithm. After the steps are divided into modules, you must ensure that no module or constituent step has been accidentally left out.

Some valid output should always be rendered for any input, even if it's merely a message indicating that the input is insufficient.

The process should take into account all the possible input conditions. So in an algorithm for boiling water, for example, you would include sub-steps for what to do if the kettle is already full or if the water is already hot.

No step in an algorithm can be ambiguous or it won't be clear whether the step has been performed properly.

Code

if X=10  
then A  
if X=9+1  
then B

The steps should account for what to do when the desired output is generated. This means you need to put an end point in the list of instructions.

Consider a text-based description of the steps in an algorithm for printing invoices.

Graphic

*The listed steps include retrieve customer details, retrieve all order details for that customer, and calculate the total invoice amount.*

You need to ensure you include all the steps. So you need to further distill the calculating step into its constituent steps.

Graphic

*You add four sub-steps to the calculate the total invoice amount step: add 5% interest to orders one month or more overdue, add 10% to orders two months and over overdue, sum the orders, and add sales tax.*

You also need to ensure there is an end point. So you add the final step to print the invoice.

Graphic

*You add print the invoice as the final step.*

The two steps related to adding interest include some ambiguity. Should orders that are two months overdue be charged an extra 10%, or an extra 15%?

Graphic

*The first sub-step of calculate the total invoice amount is indicated: add 5% interest to orders one month or more overdue.*

So you change the step to add 5% only where the order is one month overdue, and 10% only when the order is two or more months overdue.

Graphic

*The first sub-step of the calculate the total invoice amount step is changed to add 5% interest to orders one month overdue.*

All input conditions have been considered. For example an account that is three months overdue is handled as well as an account that isn't overdue.

Question

What are some of the guidelines for writing algorithms?

**Options:**

1. All the constituent steps should be present
2. All input conditions should be considered
3. Only particular inputs should be anticipated
4. No end point to the instruction should be defined

Answer

***Option 1:****Correct. You should distill the steps into their constituent parts until you can no longer break them up.*

***Option 2:****Correct. You should aim to receive valid output for any input condition.*

***Option 3:****Incorrect. All input conditions should be addressed.*

***Option 4:****Incorrect. An end point should be defined in the instruction for when the output conditions are met.*

**Correct answer(s):**

1. All the constituent steps should be present  
2. All input conditions should be considered

**2. Pseudo-code and algorithms**

After breaking down the steps of an algorithm to their most basic form, you can begin to think about converting the algorithm into a programming language.

Pseudocode is a way to start converting your natural language instructions into a programming language.

It can't be executed by a computer but it provides a framework on which to start writing the program code.

Pseudocode uses natural language that conforms to certain broad rules of syntax.

Guidelines for writing pseudocode are to

* use simple English
* use a new line for each instruction
* use indentation
* provide one entry and one exit point for the algorithm, and
* group instructions into modules

You should use simple English to make pseudocode clear and concise. For example, the algorithm instruction find the difference between numbers A and B could be rewritten as difference = A-B in pseudocode.

You should also use only one line for each instruction. Putting multiple instructions on one line makes the algorithm harder to follow. This should be re-written.

Code

BEGIN GET A GET B CALCULATE difference = A-B PRINT difference END

Each step should start on a new line.

Code

BEGIN  
GET A  
GET B  
IF B EQUALS 10  
THEN CALCULATE THE DIFFERENCE BETWEEN A AND B  
END

You should use indentation to indicate the logical hierarchy between the instructions. For example, a THEN statement should be indented from its corresponding IF statement.

Code

BEGIN  
    GET DAY  
        ASSIGN DISCOUNT VALUE  
            IF DAY = MONDAY  
                DISCOUNT = 15%  
            IF DAY = TUESDAY  
                DISCOUNT = 10%  
            IF DAY = WEDNESDAY  
                DISCOUNT = 5%  
            ELSE DISCOUNT = 0%  
        END ASSIGN DISCOUNT VALUE  
END

Instead of using BEGIN and END tags to indicate the start and end of an algorithm, you can provide other entry and exit points.

Code

INPUT A  
INPUT B  
IF B = 10  
    THEN C = A - B  
ENDIF

In this case, INPUT A is the entry point and ENDIF is the exit point.

You should group instructions into modules that can each perform a distinct operation. You could also use indentation to mark appropriate modules.

Graphic

*This code is indicated:  
IF B = 10  
   THEN C = A - B  
ENDIF*

Code

INPUT A  
INPUT B  
IF B = 10  
    THEN C = A - B  
ENDIF

Instead of re-writing modules when they're needed again, you can write them just once and simply call them as reusable objects when they're needed.

Once you've written the pseudocode for a program, you can start to turn each instruction into actual code.

Code

INPUT A  
INPUT B  
IF B = 10  
    THEN C = A - B  
ENDIF

In this example, the C programming language is used.

Code

int a = 15;  
int b = 10;  
if (b == 10) {  
    c = a - b;  
}

Note

*In this case, the = sign is an assignment operator, and == is a comparison operator.*

Question

What are some of the guidelines for writing pseudocode?

**Options:**

1. Use simple programming language
2. Use a new line for each module
3. Use indentation
4. Provide one entry and one exit point for the algorithm
5. Group instructions into modules

Answer

***Option 1:****Incorrect. You should use simple natural language to write pseudocode.*

***Option 2:****Incorrect. You should use a new line for each instruction, rather than for each module.*

***Option 3:****Correct. You make pseudocode easier to read by using indentation.*

***Option 4:****Correct. The pseudocode should provide for single entry and exit points.*

***Option 5:****Correct. You should group related instructions into modules, which can be called in other parts of the code if they're needed more than once.*

**Correct answer(s):**

3. Use indentation  
4. Provide one entry and one exit point for the algorithm  
5. Group instructions into modules

Most programming languages use certain conventions. You can apply these to pseudocode to make it easier to convert into actual executable code.

In particular, conventions relate to

* getting input
* performing operations and assigning values, and
* providing output

There are a number of common commands for handling input. You can use the GET command to acquire input from the keyboard.

Code

BEGIN  
    GET A  
    GET B  
    IF B EQUALS 10  
        THEN CALCULATE THE DIFFERENCE BETWEEN A AND B  
END

And you can use the READ command to acquire input from other sources, such as a file.

Processing of instructions in an algorithm involves arithmetic functions and assigning values to variables. In the case of pseudocode, values are assigned to nouns.

Graphic

*Two examples of assigning values to pseudo-code nouns are indicated:  
t=100  
temperature=100*

For example, in this pseudocode A and B were the nouns. You now give values to the variables A and B.

Code

INPUT A  
INPUT B  
IF B = 10  
    THEN C = A - B  
ENDIF

You can use the common arithmetic operator conventions in pseudocode:

**+**

You can use the plus sign to add values. For example, the code total=total+1 adds 1 to the value of total and assigns the new value to the total variable.

**-**

You can use the minus sign to subtract values. For example the code total=total-1subtracts 1 from the value of total and assigns the new value to the total variable.

**\***

You can use the asterisk to multiply values. For example, the code total=total\*2 multiplies the value of total by 2 and assigns the new value to the total variable.

**/**

You can use the forward slash to divide values. For example, the code total=total/2divides the value of total by 2 and assigns the new value to the total variable.

**()**

You can use parentheses to indicate the order of precedence, where the content of the brackets is calculated first. For example, in the code total=3\*(2+4), the addition will be performed before the multiplication function, resulting in total of 18 rather than 10.

There are a number of ways to assign values to variables. Depending on the programming language used, you could use VAR or LET statements, or simply assign values.

Code

total = 10

Note

*You use the = arithmetic operator to assign a value to a variable.*

For example, this BASIC programming language command stores the value 10 in the totalvariable.

Code

LET total = 10

This JavaScript programming language command stores the value 4 in the total variable.

Code

var total = 4

There are a number of ways to handle output, depending on which language you use. Each language uses its own syntax. in BASIC for example, you can use PUT, OUTPUT, DISPLAY, WRITE, and PRINT statements to handle output.

The PUT, OUTPUT, and DISPLAY statements send output to the computer screen.

Code

DISPLAY total

You use the WRITE statement to store data in a file on the computer.

Code

WRITE total

In BASIC, you can use the PRINT statement to output data to the display. This statement will print the total variable.

Code

PRINT total

To print the word "total" instead of the value of the variable that has that name*,* you enclose the word in single quotation marks.

Code

PRINT 'total'

Question

Match the examples of commands to their functions.

**Options:**

1. GET
2. READ
3. PRINT
4. WRITE
5. LET
6. VAR

**Targets:**

1. Input
2. Output
3. Assign values and arithmetic operations

Answer

*GET and READ are input statements.*

*PRINT and WRITE are output statements.*

*LET and VAR assign values to variables.*

**Correct answer(s):**

Target 1 = Option A, Option B

Target 2 = Option C, Option D

Target 3 = Option E, Option F

**3. Evaluating pseudocode**

Say you've written an algorithm in pseudocode that calculates the discount offered at a restaurant depending on the day of the week.

Code

BEGIN  
    GET DAY  
        ASSIGN DISCOUNT VALUE  
            IF DAY = MONDAY  
                DISCOUNT = 15%  
            IF DAY = TUESDAY  
                DISCOUNT = 10%  
            IF DAY = WEDNESDAY  
                DISCOUNT = 5%  
            ELSE DISCOUNT = 0%  
        END ASSIGN DISCOUNT VALUE  
END

The algorithm should include all the steps and can be broken into modules. This includes distilling each step into its constituent parts. In this case, there appears to be no way to further distill the restaurant algorithm. The ASSIGN DISCOUNT VALUE block can be considered a module, as can the entire code sample.

Code

BEGIN  
    GET DAY  
        ASSIGN DISCOUNT VALUE  
            IF DAY = MONDAY  
                DISCOUNT = 15%  
            IF DAY = TUESDAY  
                DISCOUNT = 10%  
            IF DAY = WEDNESDAY  
                DISCOUNT = 5%  
            ELSE DISCOUNT = 0%  
        END ASSIGN DISCOUNT VALUE  
END

All possible input conditions need to be addressed and there should be a single entry point. The possible input conditions are the days of the week, or an empty value. In all these cases, the IFstatements handle each eventuality and a catch-all ELSE statement handles any other condition. The entry point is the GET DAY statement.

Code

BEGIN  
    GET DAY  
        ASSIGN DISCOUNT VALUE  
            IF DAY = MONDAY  
                DISCOUNT = 15%  
            IF DAY = TUESDAY  
                DISCOUNT = 10%  
            IF DAY = WEDNESDAY  
                DISCOUNT = 5%  
            ELSE DISCOUNT = 0%  
        END ASSIGN DISCOUNT VALUE  
END

Each instruction should be free of ambiguity. All possible inputs should be accommodated, leading to valid error-free output.

Code

BEGIN  
    GET DAY  
        ASSIGN DISCOUNT VALUE  
            IF DAY = MONDAY  
                DISCOUNT = 15%  
            IF DAY = TUESDAY  
                DISCOUNT = 10%  
            IF DAY = WEDNESDAY  
                DISCOUNT = 5%  
            ELSE DISCOUNT = 0%  
        END ASSIGN DISCOUNT VALUE  
END

The steps in an algorithm need to terminate at some point to result in output, and there should be a single point of output. In this case, each output is generated by populating the DISCOUNT variable with a value. This variable is the termination point of the algorithm.

Code

BEGIN  
    GET DAY  
        ASSIGN DISCOUNT VALUE  
            IF DAY = MONDAY  
                DISCOUNT = 15%  
            IF DAY = TUESDAY  
                DISCOUNT = 10%  
            IF DAY = WEDNESDAY  
                DISCOUNT = 5%  
            ELSE DISCOUNT = 0%  
        END ASSIGN DISCOUNT VALUE  
END

Pseudocode should use simple, natural language. In this case, basic English words have been used.

Code

BEGIN  
    GET DAY  
        ASSIGN DISCOUNT VALUE  
            IF DAY = MONDAY  
                DISCOUNT = 15%  
            IF DAY = TUESDAY  
                DISCOUNT = 10%  
            IF DAY = WEDNESDAY  
                DISCOUNT = 5%  
            ELSE DISCOUNT = 0%  
        END ASSIGN DISCOUNT VALUE  
END

A new line should be used for each step in the instruction, and sub-instructions should be indented. In the example, each instruction is on a new line and the child elements are indented.

Code

BEGIN  
    GET DAY  
        ASSIGN DISCOUNT VALUE  
            IF DAY = MONDAY  
                DISCOUNT = 15%  
            IF DAY = TUESDAY  
                DISCOUNT = 10%  
            IF DAY = WEDNESDAY  
                DISCOUNT = 5%  
            ELSE DISCOUNT = 0%  
        END ASSIGN DISCOUNT VALUE  
END

**Case Study: Question 1 of 2**

Scenario

*For your convenience, the case study is repeated with each question.*

You're writing an algorithm that calculates the tip amount that should be given at a restaurant, depending on the level of service. To do this, you need to get the total bill, subtract the sales tax from the total before establishing the tip percentage – which is determined based on the level of service, and calculate the tip amount based on the subtotal and level of service.

Answer the questions in order.

Question

How is this pseudocode deficient?

**Code**  
BEGIN GET BILL TOTAL GET SUBTOTAL BY SUBTRACTING TAX FROM BILL TOTAL  
INPUT "How was the service (poor, good or excellent)?"  
IF SERVICE WAS POOR  
THEN TIP AMOUNT EQUALS 10%  
IF SERVICE WAS GOOD  
THEN TIP AMOUNT EQUALS 15%  
IF SERVICE WAS EXCELLENT  
THEN TIP AMOUNT EQUALS 20%

**Options:**

1. Each instruction should be on a new line
2. The instruction shouldn't be ambiguous
3. Instructions should be grouped in modules
4. Input should be handled

Answer

***Option 1:****Correct. GET BILL and GET SUBTOTAL should be on separate new lines.*

***Option 2:****Incorrect. The pseudocode handles all possible input conditions without ambiguity.*

***Option 3:****Incorrect. The full pseudocode sample is itself a module.*

***Option 4:****Incorrect. The pseudocode handles all possible input conditions.*

**Correct answer(s):**

1. Each instruction should be on a new line

**Case Study: Question 2 of 2**

Scenario

*For your convenience, the case study is repeated with each question.*

You're writing an algorithm that calculates the tip amount that should be given at a restaurant, depending on the level of service. To do this, you need to get the total bill, subtract the sales tax from the total before establishing the tip percentage – which is determined based on the level of service, and calculate the tip amount based on the subtotal and level of service.

Answer the questions in order.

Question

How is this pseudocode deficient?

**Code**  
BEGIN  
GET BILL TOTAL  
GET SUBTOTAL BY SUBTRACTING TAX FROM BILL TOTAL  
INPUT "How was the service (poor, good or excellent)?"  
IF SERVICE WAS POOR  
THEN TIP AMOUNT EQUALS 10%  
IF SERVICE WAS GOOD  
THEN TIP AMOUNT EQUALS 15%  
IF SERVICE WAS EXCELLENT  
THEN TIP AMOUNT EQUALS 20%

**Options:**

1. The pseudocode should use indentation
2. There is no end point
3. Each instruction should be on a new line
4. Instructions should be grouped in modules

Answer

***Option 1:****Correct. You should indent instructions that are sub-steps of a parent line. THENstatements should be indented from their IF statements.*

***Option 2:****Correct. The pseudocode still needs to calculate and print the tip amount, which is the subtotal divided by the tip amount.*

***Option 3:****Incorrect. Each new instruction is already on a new line.*

***Option 4:****Incorrect. The entire sample constitutes a single module.*

**Correct answer(s):**

1. The pseudocode should use indentation  
2. There is no end point

**4. Summary**

A computer program receives input, processes it, and generates output to solve a problem. The algorithm is what performs the processing. When there is a lot of input and a system has to perform a complicated process, keeping track of the output can be challenging. The top-down approach to algorithm writing addresses this problem. Algorithms should include all the steps, all input conditions should be addressed, no step should be ambiguous, and steps should terminate to result in output at some point.  
  
You can use pseudocode to start converting natural language instructions into an algorithm in preparation for a programming language. Pseudocode should use simple language, use a new line for each instruction, use indentation, provide one entry and one exit point for the algorithm, and group instructions into modules. You can apply conventions that most programming languages use to make pseudocode easier to convert into executable code.

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/sd_spff_a01_it_enus/output/html/course_transcript.html#top)

Structuring an Algorithm

Learning Objectives

*After completing this topic, you should be able to*

* *recognize examples of sequence, selection, and repetition*
* *express inputs, processes and outputs as an algorithm*

**1. Sequence, selection, and repetition**

In the early 1960s, the coding method used by programmers was less logical and linear than it's today. This led to the method being called *spaghetti code*.

Spaghetti code relied on the GOTO command to skip unnecessary lines of code in a program. The GOTO command recognized lines of code by a name or number, depending on the programming language.  
  
For example, if a certain criterion was met in a line of code, the GOTO command could then move the program along to a later line of code.

Graphic

*Five lines of code are shown, code skips from 3 to 5 as follows:  
1: CODE  
2: CODE  
3: CODE GO TO 5  
4: CODE  
5: CODE*

The GOTO command resulted in a lot of movement backwards and forwards through the code, which was difficult to follow logically. If a program used too many GOTO commands, it would end up with a complex structure that was prone to errors.

Consider writing a program to print address labels. You would need the program to understand the difference between a mailbox and a street address.

The example code shows how spaghetti code would rely on the GOTO statement to differentiate between the two address types.

Code

21: PRINT name  
22: IF address IS empty GOTO 25  
23: PRINT address  
24: GOTO 26  
25: PRINT postbox  
26: PRINT city  
27: PRINT state  
28: GOTO 17

Structured programming on the other hand would use IF, THEN, and ELSE statements in a linear and logical sequence.

Code

IF street IS empty    
    THEN      
      PRINT postbox    
    ELSE      
      PRINT address  
ENDIF  
PRINT city  
PRINT state

The end result of both methods is the same, but it's far easier to follow the structured programming method due to its linearity.

Structured programming uses three control structures as the basis for writing a program:

**sequence**

The sequence control structure ensures that a program runs instructions in sequence and performs each instruction once only. Instructions aren't repeated or skipped, unless repetition control is applied.

**selection, and**

The selection control structure is used when a choice has to be made. These choices depend on the conditions defined by the programmer.

**repetition**

The repetition control structure is used to repeat a certain section of code until a specific condition is met.

Consider sequence control in an example of code for printing information. Each command runs only once to perform the necessary print action.  
  
The sequence of the printed address in this case is the sequence the programmer specifies – the recipient's name, followed by the recipient's street address, and so on.

Code

21: PRINT name  
22: PRINT address  
23: PRINT postbox  
24: PRINT city  
25: PRINT state

The most commonly used commands for capturing information for selection control are IF, THEN, and ELSE.

Code

IF street IS empty    
    THEN   
        PRINT postbox    
    ELSE   
        PRINT street

Some programming languages require that you end any IF statement with a keyword, such as ENDIF.

Code

IF street IS empty    
    THEN   
        PRINT postbox    
    ELSE   
        PRINT street  
ENDIF  
PRINT city  
PRINT state

In certain scenarios, you may not need to use the ELSE statement if there is no alternative action to take.

Code

    ELSE 

In this example, you're creating a list of adults. Persons under the age of 18 shouldn't be added to the list. There is no ELSE statement because there is no action to take if a person's age is under 18.

Code

IF age >= 18 THEN  
    STORE name IN adult\_list  
ENDIF

This is known as a null ELSE statement.

The repetition control structure is used to repeat a section of code until a specific condition is met. This is also known as looping.

The commands used to create a loop are FOR, WHILE, and DO. FOR is the most common of these commands and is used across multiple languages.  
  
In this example, a WHILE loop is terminated using the ENDWHILE command. Different programming languages may use different methods to terminate loops.

Code

seats\_allocated = 0  
WHILE seats\_allocated < 5 DO     
    GET booking     
    PRINT ticket     
    ADD 1 to seats\_allocated  
ENDWHILE

This code is used to book seats on a minibus. The seats\_allocated value is defined before the loop starts because the loop needs a value to compare seats\_allocated to. The code runs until the value of seats\_allocated reaches 5, at which point the ENDWHILE command runs.

Question

Consider the code example.  
  
Which type of control structure is being used?

**Code**  
GET name  
PRINT name  
GET address  
PRINT address  
GET postbox  
PRINT postbox  
GET city  
PRINT city  
GET state  
PRINT state

**Options:**

1. Sequence
2. Selection
3. Repetition

Answer

***Option 1:****Correct. The sequence control structure ensures that instructions are run once in sequence. No instructions are repeated or skipped.*

***Option 2:****Incorrect. Selection control is used when a choice needs to be made. This code represents a sequence control in which instructions are run one at a time in the specified order.*

***Option 3:****Incorrect. Repetition control repeats a certain part of the code until a specific condition is met. This code represents a sequence control in which instructions are run one at a time in the specified order.*

**Correct answer(s):**

1. Sequence

**2. Inputs, processes, and outputs**

An *algorithm* is a set of instructions used to solve a problem. Creating this is the first step a programmer will take when developing a computer program. The programmer constructs algorithms using simple language before beginning to use any actual computer code.

An algorithm takes an input, processes it, and produces an output. These different parts of an algorithm are written using different types of words. The processing in an algorithm is usually described using verbs such as "find", "calculate", or "sort." The inputs or outputs are described using nouns, adjectives, or adverbs. Examples are "time", "speed", or "distance."

It's common to use a defining table, or input-process-output table, as the basis for developing an algorithm.

Graphic

*A table with three columns and one row is labelled Defining table: the column headers are Input, Process, and Output. The row details are Input: Get booking, Process: Add 1 to seats allocated, and Output: Print ticket.*

The process elements become statements in the algorithm.

The input and output elements become arguments for the statements in the algorithm. An argument is similar to the object of a sentence. In this example, "booking" is an argument.

Say you need to calculate the speed that a courier must drive in order to cover a certain distance in a specific time. You have to take time, speed, and distance into account, and then create an algorithm that will process all these variables.

To create a defining table, you'd need to identify what your input, process, and output are. In this case, the inputs are distance and time, the process can be defined as "calculate", and the output is speed. Using this table, you can create your algorithm.

Once you have calculated the output, you need to specify a process or operation to make the output visible. An example is an instruction to display the speed.

Code

BEGIN  
Convert kilometers\_to\_miles    
    DISPLAY 'Enter the distance in kilometers.'    
    GET kilometers    
    miles = kilometers \* 0.625    
    DISPLAY kilometers 'kilometers = ' miles 'miles.'  
END

Once you've defined your input, process, and output using the defining table, you can start to create the algorithm.

Question

You're developing an algorithm to print shipping labels for certain packages in a post office.  
  
Match each statement to the correct column in an input-process-output table.

**Options:**

1. INPUT "More [Y/N]?"
2. WHILE answer = "Y"
3. PRINT Label

**Targets:**

1. The input column
2. The process column
3. The output column

Answer

*INPUT "More [Y/N]?" is an input statement. Users have to input Y or N, depending on whether they want to print labels.*

*After confirming that the label is needed, the algorithm processes any statements inside the WHILE command. So the WHILE command belongs in the process column.*

*A label is created as output using the PRINT statement.*

**Correct answer(s):**

Target 1 = Option A

Target 2 = Option B

Target 3 = Option C

An algorithm usually follows a certain structure. It begins with a name that describes the algorithm's function.

Code

CALCULATE overdraft\_interest

Once the algorithm has been named, any instructions that occur between the name and the ENDstatement must be indented. This makes it easier to identify each of the instructions.

Code

        WHILE answer="Y" DO   
        READ name        
        READ address        
        PRINT label        
        DISPLAY "Print another?"     
        ENDWHILE

You need to indent every new control structure that is introduced. The indentation will return to its previous level at the end of a control structure.

Code

CALCULATE overdraft\_interest     
    WHILE answer="Y" DO         
        READ name        
        READ address        
        PRINT label        
        DISPLAY "Print another?"     
    ENDWHILE

The different components of the defining table can be broken down into one or more pseudocode instructions.

Graphic

*A table with three columns and one row is labelled Defining table: the column headers are Input, Process, and Output. The row details are Input: GET info, Process: Do action, and Output: DISPLAY result.*

The components of the input and output columns are used as arguments for the instructions in an algorithm.

Say you need to convert kilometers to miles. You can create an algorithm to express this.

You begin by defining the inputs, processes, and outputs.

The input is the distance in kilometers, the process is the conversion of miles to kilometers, and the output is the distance in miles.

Graphic

*A table with three columns and one row is labelled Defining table: the column headers are Input, Process, and Output. The input is "GET kilometers", the process is "CALCULATE miles = kilometers \* 0.625", and the output is "DISPLAY enter distance in km" and   
"DISPLAY miles."*

You can then use the table to create an algorithm that converts kilometers to miles.

Graphic

*A table with three columns and one row is labelled Defining table: the column headers are Input, Process, and Output. The input is "GET kilometers", the process is "CALCULATE miles = kilometers \* 0.625", and the output is "DISPLAY enter distance in km" and   
"DISPLAY miles."*

Code

BEGIN Convert kilometers\_to\_miles  
  DISPLAY 'Enter the distance in kilometers.'  
  GET kilometers  
  miles = kilometers \* 0.625  
  DISPLAY kilometers 'kilometers = ' miles 'miles.'  
END

Even with a defining table to help you create the algorithm, you'll probably need to refine it several times.

Question

You need to calculate the speed a courier must drive to cover a given distance within a given time.  
  
Identify the correct pseudocode algorithm to express this.

**Options:**

1. Get distance  
   Get time  
   Calculate speed by dividing distance by time  
   Display speed
2. Get distance  
   Calculate speed by dividing distance by time  
   Display speed
3. Get distance  
   Get time  
   Calculate speed by adding distance to time  
   Display speed

Answer

***Option 1:****Correct. This algorithm can correctly calculate speed as it has the requisite distance and time as inputs.*

***Option 2:****Incorrect. This algorithm can't calculate speed without a value input for time.*

***Option 3:****Incorrect. This algorithm can't correctly calculate speed as it uses the incorrect process of adding distance to time.*

**Correct answer(s):**

1. Get distance  
Get time  
Calculate speed by dividing distance by time  
Display speed

**3. Summary**

You can use the three control structures – sequence, selection, and repetition – to help structure an algorithm you're creating.  
  
An algorithm can also be broken down into three different components – input, output, and process. Once you've identified the different components, you can begin to build the algorithm.

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/sd_spff_a01_it_enus/output/html/course_transcript.html#top)

Creating an Algorithm

Learning Objectives

*After completing this topic, you should be able to*

* *create a letter-counting algorithm*
* *create a tax-calculating algorithm*

**1. Exercise overview**

In this exercise, you're required to identify the input, process, and output elements of problem statements in order to construct an algorithm.

This involves the following tasks:

* using a defining table to create a letter-counting algorithm, and
* using a defining table to create a tax-calculating algorithm

**2. Creating a letter-counting algorithm**

You want to write an algorithm that asks the user to enter two words and then adds the number of letters in the first word to the number of letters in the second word, and then displays total number of letters in both words.

Question

To help create the algorithm, you decide to use a table to identify the input and output elements and the order in which they should appear. The algorithm has to be designed to display a result as output immediately after a user enters a word. It also has to return the total number of characters in two words.   
  
Assign each step, which is explained using pseudocode, to the appropriate column.

**Options:**

1. word1\_length
2. total
3. word2\_length
4. word1
5. word2

**Targets:**

1. Input
2. Output

Answer

*The word1 step is the first input parameter. The word2 step is the second input parameter. These are all the inputs that the process requires.*

*The word\_length1 step corresponds to the number of letters in the first input , word1. The word\_length2 step is the number of letters in the second input , word2. And total is the sum of the number of letters in both words. These are the three outputs of the process.*

**Correct answer(s):**

Target 1 = Option D, Option E

Target 2 = Option A, Option C, Option B

In the algorithm, the numbers of characters are the required output. In this case, the algorithm is symmetric – meaning that you can re-order the inputs to get the same total.  
  
If the algorithm is well designed, it will shut down if it encounters a problem such as invalid or missing input. A good algorithm doesn't assume the availability of required elements.  
  
You could extend the algorithm to receive a variable number of words.

Code

Input 1: word1  
Input 2: word2  
...PROCESS...  
Output 1: word1\_length  
Output 2: word2\_length  
Output 3: total

Question

Next you want to structure the processes that change the input elements to the desired output elements.  
  
Place the processes – which are described in pseudocode – in the order in which they should appear in the algorithm.

**Options:**

1. GET word1  
   GET word2
2. CALCULATE word\_length1  
   CALCULATE word\_length2
3. ADD word\_length1+word\_length2=total
4. PRINT total

Answer

**Correct answer(s):**

**GET word1  
GET word2 is ranked**

The first step is to retrieve, or "GET", the relevant parameters – in this case, the two words entered by a user.

**CALCULATE word\_length1  
CALCULATE word\_length2 is ranked**

As the second step, the algorithm must calculate the length of each of the two words entered by the user.

**ADD word\_length1+word\_length2=total is ranked**

As the third step, you add the number of characters in each of the two words to calculate a total. You can do this using an ADD operation. Note that this operation can easily be expanded to accept more than two inputs, making the algorithm more dynamic and flexible.

**PRINT total is ranked**

The final step is to print the result, which is the total number of characters in the two words a user has entered. Every programming language has an equivalent of a PRINToperation that prints to the standard output device – usually the screen. Standard output can be redirected to a printer, a file, or some other device.

The defining table has been completed and you can now create the algorithm.

Graphic

*A table with three columns and one row is labelled Defining table: the column headers are Input, Process, and Output. The input is GET word1 and   
GET word2, the process is CALCULATE word\_length1, CALCULATE word\_length2, and ADD word\_length1+word\_length2=total, and the output is PRINT total.*

Question

Which is the correct algorithm for the defining table you just created?

**Options:**

1. GET word1  
   GET word2  
   CALCULATE word1\_length  
   CALCULATE word2\_length  
   ADD word1\_length TO word2\_length  
   PRINT total
2. GET word1  
   CALCULATE word1\_length  
   CALCULATE word2\_length  
   ADD word1\_length TO word2\_length  
   PRINT total
3. GET word1  
   GET word2  
   CALCULATE word1\_length and word2\_length  
   ADD word1\_length TO word2\_length  
   PRINT total

Answer

***Option 1:****Correct. You GET the input, CALCULATE and ADD the numbers of characters as the process, and print the total as the output.*

***Option 2:****Incorrect. Input for word2 has not been handled in this example.*

***Option 3:****Incorrect. There is ambiguity here about whether the CALCULATE function applies to the word2\_length.*

**Correct answer(s):**

1. GET word1  
GET word2  
CALCULATE word1\_length  
CALCULATE word2\_length  
ADD word1\_length TO word2\_length  
PRINT total

**3. Creating a tax-calculating algorithm**

You now want to create an algorithm that calculates and prints the tax payable on an invoice total. The tax and total should then be passed back to the calling routine.

Question

To create the algorithm, you use a defining table to identify the input and output elements and the order in which they should appear.  
  
Match each option with the correct column.

**Options:**

1. GET invoice total
2. CALCULATE tax  
   final\_amount = invoice total + tax
3. PRINT tax  
   PRINT final\_amount

**Targets:**

1. Input column
2. Process column
3. Output column

Answer

*GET invoice total is an input command. The invoice total is required for the process to work.*

*CALCULATE tax and final\_amount = invoice total + tax are process commands. The invoice total is added to a tax value to result in the final\_amount.*

*PRINT tax and PRINT final\_amount are output commands. The process results in values for taxand for the final\_amount.*

**Correct answer(s):**

Target 1 = Option A

Target 2 = Option B

Target 3 = Option C

Question

Identify the correct algorithm for the defining table you created.

**Options:**

1. GET invoice total  
   CALCULATE tax   
   final\_amount = invoice total + tax  
   PRINT tax  
   PRINT final\_amount   
   RETURN tax AND final\_amount TO CALLING ROUTINE
2. GET invoice total  
   CALCULATE tax  
   final\_amount = invoice total + tax  
   RETURN tax AND final\_amount TO CALLING ROUTINE  
   PRINT tax  
   PRINT final\_amount

Answer

***Option 1:****Correct. This is the correct algorithm to calculate and print the tax payable on an invoice total.*

***Option 2:****Incorrect. RETURN tax AND final\_amount TO CALLING ROUTINE will finish before the algorithm is complete. This code is written using the spaghetti programming method.*

**Correct answer(s):**

1. GET invoice total  
CALCULATE tax   
final\_amount = invoice total + tax  
PRINT tax  
PRINT final\_amount   
RETURN tax AND final\_amount TO CALLING ROUTINE

Two algorithms have been created – one to count letters in words and the other to calculate tax.

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/sd_spff_a01_it_enus/output/html/course_transcript.html#top)

© 2016 Skillsoft Ireland Limited