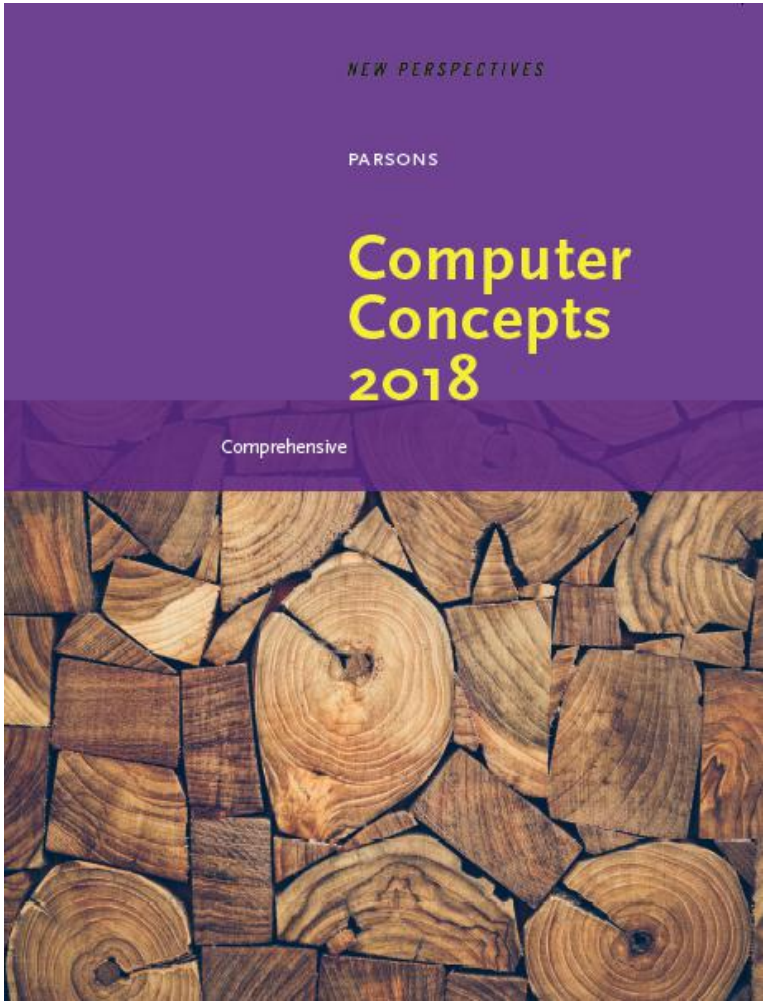
Computer Concepts 2018



Module 11
Programming

Module Contents

- Section A: Program Development
- Section B: Programming Tools
- Section C: Procedural Programming
- Section D: Object-Oriented Programs
- Section E: Declarative Programming

Section A: Program Development

- Programming Basics
- Program Planning
- Writing Programs
- Program Testing and Documentation

Section A: Objectives (1 of 2)

- Describe the difference between programming and software engineering
- List the three core elements of a problem statement and provide an example of each
- Supply at least three examples of projects that would be best tackled using predictive methodology and three projects best tackled using agile methodology
- Describe the difference between constants and variables, and provide an example of how each would be used in a program

Section A: Objectives (2 of 2)

- List three types of errors that might be encountered during program testing
- Explain the significance of formal methods
- Explain the purpose of STRIDE and DREAD
- Explain the significance of defensive programming

Programming Basics (1 of 3)

- **Computer programming** encompasses a broad set of activities that include planning, coding, testing, and documenting
- A related activity, **software engineering**, is a development process that uses mathematical, engineering, and management techniques to reduce the cost and complexity of a computer program while increasing its reliability and modifiability
- Software engineering can be characterized as more formalized and rigorous than computer programming

Programming Basics (2 of 3)

Hashtags indicate comments that are used for documentation.

The program makes calculations using inches and centimeters. These values are initially set to 0.

The program begins by displaying a title.

The program asks the user to enter a length, which is stored in a variable called inches.

The calculation is performed and stored in a variable called centimeters.

The program displays the length in centimeters and then ends.

```
# This program converts inches to centimeters

inches = 0.0
centimeters = 0.0
print ("Convert Inches to Centimeters.")

inches = input("Enter length in inches: ")

centimeters = 2.54 * inches

print ("That is ", centimeters, " centimeters.")
```

Programming Basics (3 of 3)

- Programmers typically specialize in either *application programming* or *system development*
- **Application programmers** create productivity applications such as Microsoft Office
- **Systems programmers** specialize in developing system software such as operating systems, device drivers, security modules, and communications software

Program Planning (1 of 3)

- In the context of programming, a **problem statement** defines certain elements that must be manipulated to achieve a result or goal
- A good problem statement for a computer program has three characteristics:
  - It specifies any assumptions that define the scope of the problem
  - It clearly specifies the known information
  - It specifies when the problem has been solved

Program Planning (2 of 3)

- In a problem statement, an **assumption** is something you accept as true in order to proceed with program planning
- The **known information** in a problem statement is the information that is supplied to the computer to help it solve a problem
- After identifying the known information, a programmer must specify how to determine when the problem has been solved

Program Planning (3 of 3)

- Several software development methodologies exist to help program designers and coders plan, execute, and test software
- Methodologies can be classified as *predictive* or *agile*
  - A **predictive methodology** requires extensive planning and documentation up front; it's used to construct buildings and assemble cars—tasks that are well defined and predictable
  - An **agile methodology** focuses on flexible development and specifications that evolve as the project progresses

Program Coding (1 of 6)

- The core of a computer program is a sequence of instructions
- A **keyword**, or command, is a word with a predefined meaning
- Keywords differ depending on the programming language; there is a basic vocabulary that covers most necessary tasks

Program Coding (2 of 6)

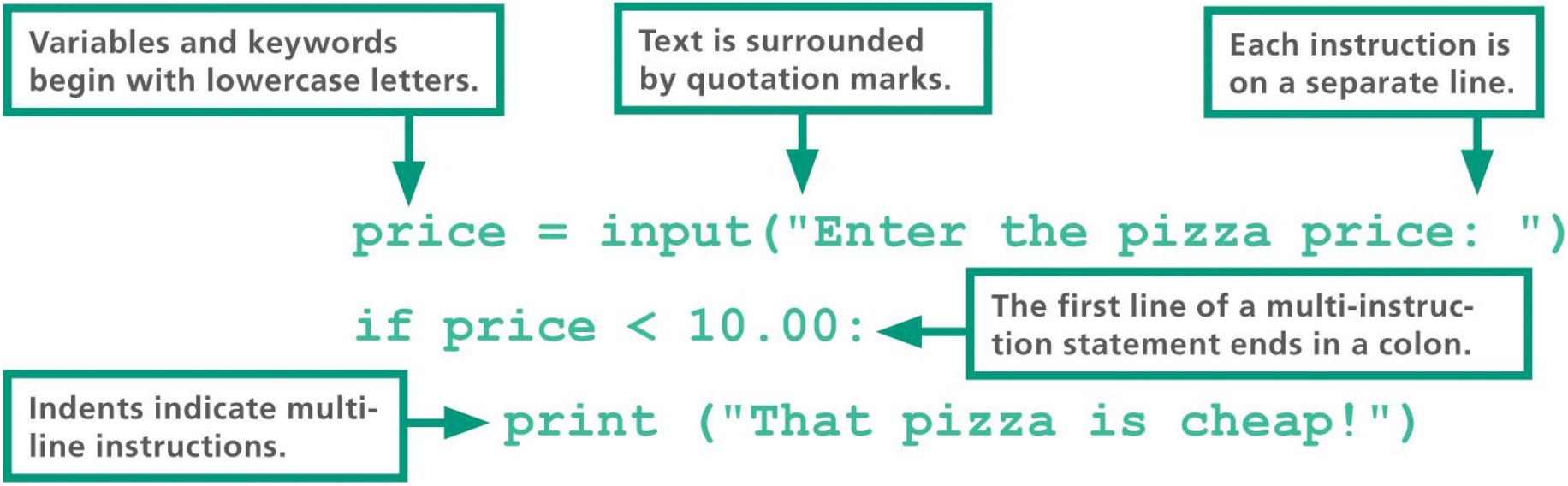
input	Collection information from the program's user.
print	Display information on the screen.
while	Begin a series of commands that will be repeated in a loop.
break	Terminate a loop.
if	Execute one or more instructions only if a specified condition is true.
else	Add more options to extend the if command.
def	Define a series of instructions that become a unit called a function.
return	Transfer data from a function to some other part of the program.
class	Define an object as a set of attributes and methods.

Program Coding (3 of 6)

- Keywords can be combined with specific **parameters**, which provide more detailed instructions for the computer to carry out
- These parameters include *variables* and *constants*
  - A **variable** represents a value that can change
  - A **constant** is a factor that remains the same throughout a program

Program Coding (4 of 6)

- The set of rules that specify the sequence of keywords, parameters, and punctuation in a program instruction is referred to as **syntax**



Program Coding (5 of 6)

- You may be able to use a text editor, program editor, or graphical user interface to code computer programs
- A **text editor** is any word processor that can be used for basic text editing tasks, such as writing email, creating documents, and coding computer programs
- A **program editor** is a type of text editor specially designed for entering code for computer programs
- A **VDE** (visual development environment) provides programmers with tools to build substantial sections of a program by pointing and clicking rather than typing each statement

Program Coding (6 of 6)

- Frequently used controls include labels, menus, toolbars, list boxes, text boxes, option buttons, check boxes, and graphical boxes
- A control can be customized by specifying values for a set of built-in **properties**

Program Testing and Documentation (1 of 9)

- Programs that don't work correctly might crash, run forever, or provide inaccurate results; when a program isn't working, it's usually the result of a *runtime*, *logic*, or *syntax error*
  - A **runtime error** occurs when a program runs instructions that the computer can't execute
  - A **logic error** is a type of runtime error in the logic or design of the program
  - A **syntax error** occurs when an instruction does not follow the syntax rules of the programming language

Program Testing and Documentation (2 of 9)

- Omitting a keyword, such as ELSE
- Misspelling a keyword, such as mistakenly typing PIRN instead of PRINT
- Omitting required punctuation, such as a period, comma, or bracket
- Using incorrect punctuation, such as typing a colon where a semicolon is required
- Forgetting to close parentheses

Program Testing and Documentation (3 of 9)

- The process of finding and fixing errors in a computer program is called **debugging**
- Programmers can locate errors in a program by reading through each line, much like a proofreader
- Programmers also insert documentation called **remarks** (or comments) into the program

Program Testing and Documentation (4 of 9)

- Programs need to meet *performance*, *usability*, and *security* standards
  - **Performance** – programmers need to carry out real-world tests to ensure that programs don't take too long to load
  - **Usability** – programs should be easy to learn and use and be efficient
  - **Security** – program specifications are formulated so that programmers remain aware of security throughout the software development life cycle

Program Testing and Documentation (5 of 9)

- **Formal methods** help programmers apply rigorous logical and mathematical models to software design, composition, testing, and verification
- **Threat modeling** (also called risk analysis) is a technique that can be used to identify potential vulnerabilities by listing the key assets of an application, categorizing the threats to each asset, ranking the threats, and developing threat mitigation strategies that can be implemented during programming

Program Testing and Documentation (6 of 9)

- Spoofing: Pretending to be someone else
- Tampering: Changing, adding, or deleting data
- Repudiation: Covering tracks to make attacks difficult to trace
- Information disclosure: Gaining unauthorized access to information
- Denial of service: Making a system unavailable to legitimate users
- Elevation of Privilege: Modifying user rights to gain access to data

Program Testing and Documentation (7 of 9)

- Damage: How much damage can a particular attack cause?
- Reproduce: Is this attack easy to reproduce?
- Exploit: How much skill is needed to launch the attack?
- Affected: How many users would be affected by an attack?
- Discovered: How likely is it that this attack would be discovered?

Program Testing and Documentation (8 of 9)

- **Defensive programming** (also referred to as secure programming) is an approach to software development in which programmers anticipate what might go wrong as their programs run and take steps to smoothly handle those situations
- Techniques associated with defensive programming include:
  - **Walkthroughs.** Open source software goes through extensive public scrutiny that can identify security holes, but proprietary software can also benefit from a walkthrough with other in-house programmers

Program Testing and Documentation (9 of 9)

- **Simplification.** Complex programs are more difficult to debug than simpler ones. Simplifying complex sections can sometimes reduce a program's vulnerability to attacks
- **Filtering input.** It is dangerous to assume that users will enter valid input. Attackers have become experts at concocting input that causes buffer overflows and runs rogue HTML scripts. Programmers should use a tight set of filters on all input fields

Section B: Programming Tools

- Language Evolution
- Compilers and Interpreters
- Paradigms and Languages
- Toolsets

Section B: Objectives (1 of 2)

- Explain how the concept of abstraction applies to programming languages
- Provide two examples of low-level languages and five examples of high-level languages
- Explain how assemblers are related to compilers
- Describe the difference between compiling a program and using an interpreter
- List and describe three popular programming paradigms
- List at least three legacy programming languages

Section B: Objectives (2 of 2)

- List two programming languages used to program mobile apps
- List three programming languages that are popular for developing dynamic Web sites
- Explain how programmers use IDEs, SDKs, VDEs, and APIs

Language Evolution (1 of 7)

- When applied to programming languages, **abstraction** inserts a buffer between programmers and the chip-level details of instruction sets and binary data representation
- For programming languages, abstraction automates hardware-level details, such as how to move data from memory to the processor
- A **low-level language** has a low level of abstraction because it includes commands specific to a particular CPU or microprocessor family
- A **high-level language** uses command words and grammar based on human languages to provide a level of abstraction that hides the underlying low-level language

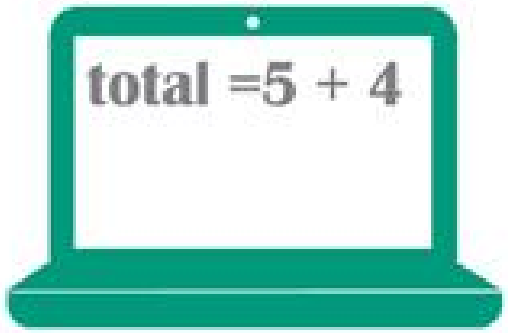
Language Evolution (2 of 7)

Programmers using low-level languages have to deal with hardware-level tasks, such as loading data into registers of the processor with the following code:

MOV REG1

MOV REG2

ADD REG1, REG2



Programmers using high-level languages are buffered from the hardware details by levels of abstraction. Only one instruction is needed, and the programmer does not have to specify the registers where the numbers are located.

Language Evolution (3 of 7)

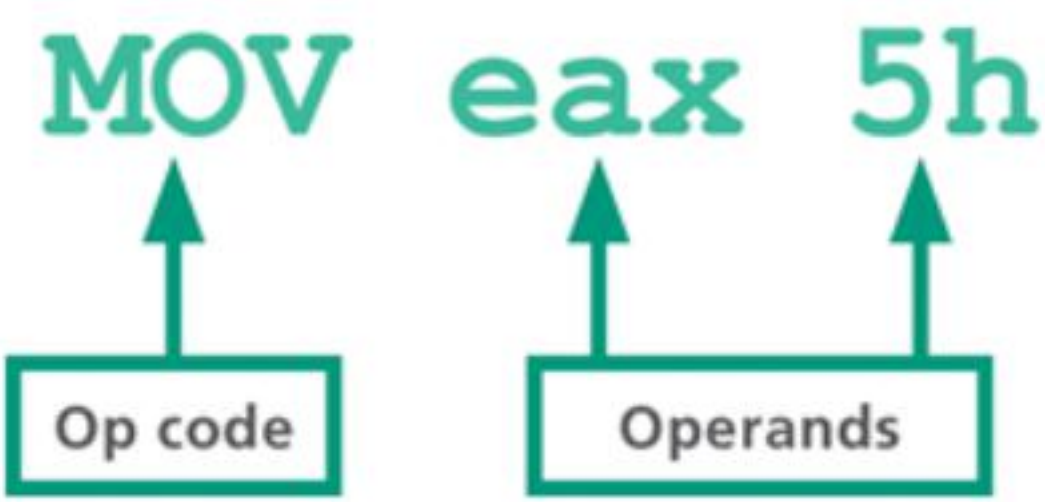
- **First-generation** languages are the first machine languages programmers used
- **Second-generation** languages added a level of abstraction to machine languages by substituting abbreviated command words for binary numbers
- **Third-generation** languages were conceived in the 1950s and used easy-to-remember command words, such as PRINT and INPUT

Language Evolution (4 of 7)

- An **assembly language** is classified as a low-level language because it is machine specific
- An **assembler** typically reads a program written in an assembly language, which has two parts: the *op code* and the *operand*
  - An **op code**, which is short for *operation code*, is a command word for an operation such as add, compare, or jump
  - The **operand** for an instruction specifies the data for the operation

Language Evolution (5 of 7)

- Look at the parts of an assembly language shown in the below figure—consider how tedious it would be to write a program consisting of thousands of these concise, but cryptic, op codes:



Language Evolution (6 of 7)

- **Fourth-generation** languages are considered “high-level” languages and more closely resemble human languages
- The computer language Prolog, based on a declarative programming paradigm, is identified as a **fifth-generation** language—though some experts disagree with this classification

Language Evolution (7 of 7)

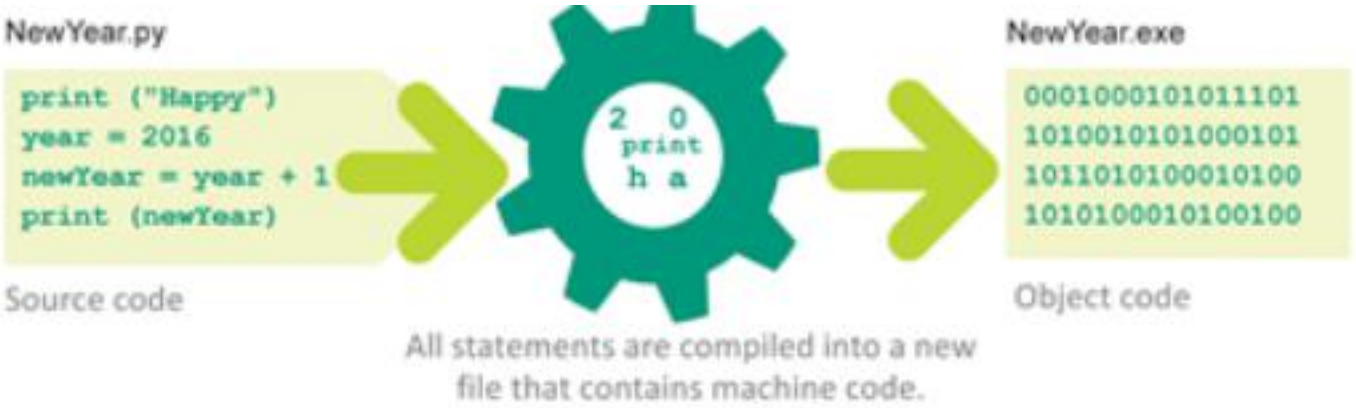
```
SORT TABLE Kids on Lastname
```

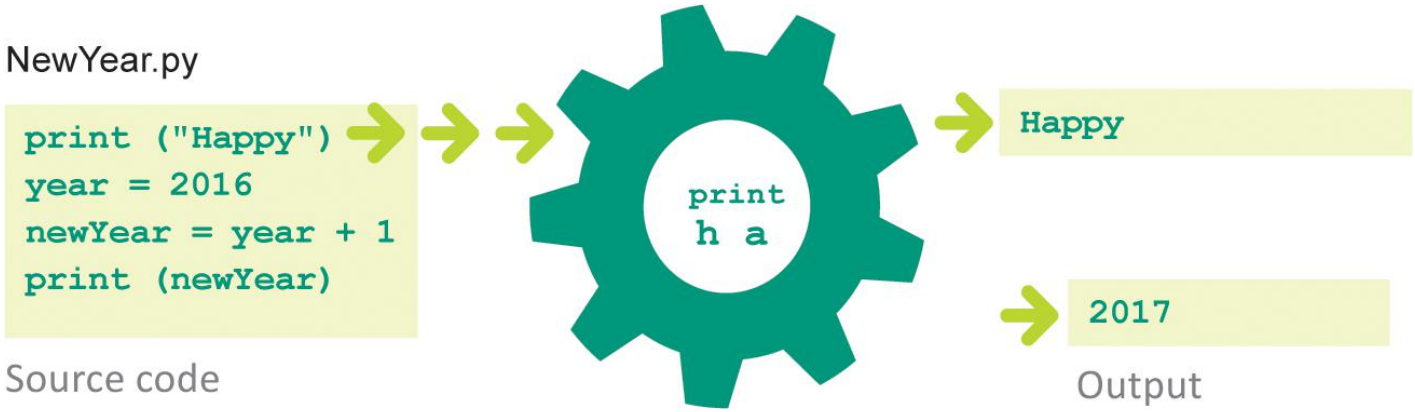
```
PUBLIC SUB Sort(Kids As Variant, inLow As Long, inHi As Long)
    DIM pivot   As Variant
    DIM tmpSwap As Variant
    DIM tmpLow  As Long
    DIM tmpHi   As Long
    tmpLow = inLow
    tmpHi = inHi
     pivot = Kids((inLow + inHi) \ 2)
    WHILE (tmpLow <= tmpHi)
       WHILE (Kids(tmpLow) < pivot And tmpLow < inHi)
          tmpLow = tmpLow + 1
       WEND
       WHILE (pivot < Kids(tmpHi) And tmpHi > inLow)
          tmpHi = tmpHi - 1
       WEND
     IF (tmpLow <= tmpHi) THEN
          tmpSwap = Kids(tmpLow)
          Kids(tmpLow) = Kids(tmpHi)
          Kids(tmpHi) = tmpSwap
          tmpLow = tmpLow + 1
          tmpHi = tmpHi - 1
       END IF
    WEND
    IF (inLow < tmpHi) THEN Sort Kids, inLow, tmpHi
    IF (tmpLow < inHi) THEN Sort Kids, tmpLow, inHi
END SUB
```

Compilers and Interpreters (1 of 3)

- The human-readable version of a program created in a high-level language by a programmer is called **source code**
- Source code must first be translated into machine language using a *compiler* or *interpreter*
  - A **compiler** converts all the statements in a program in a single batch, and the resulting collection of instructions, called **object code**, is placed in a new file
  - An **interpreter** converts and executes one statement at a time while the program is running; once executed, the interpreter converts and executes the next statement

Compilers and Interpreters (2 of 3)





Compilers and Interpreters (3 of 3)

```
 1  import random
 2  min = 1
 3  max = 6
 4
 5  rollAgain = "yes"
 6
 7  while rollAgain == "yes" or rollAgain == "y":
 8      print ("Rolling...")
 9      print ("The values are ...")
10      print (random.randint(man,max))
11      print (random.randint(min,max))
12
13      rollAgain = input("Roll again? ")
```

```
COMPILE ERROR!
Traceback (most recent call last):
File "python", line 10, in <module>
NameError: name 'man' is not defined
```

This program contains an error in line 10. Even though lines 1 through 9 contain no errors, their output is not displayed because the program did not compile without errors.

Paradigms and Languages (1 of 4)

- The phrase **programming paradigm** refers to a way of conceptualizing and structuring the tasks a computer performs
- A programmer uses a programming language that supports the paradigm
- Other programming languages—referred to as **mulitparadigm languages**—support more than one paradigm

Paradigms and Languages (2 of 4)

PARADIGM	DESCRIPTION
Procedural	Emphasizes linear steps that provide the computer with instructions on how to solve a problem or carry out a task
Object-oriented	Formulates programs as a series of objects and methods that interact to perform a specific task
Declarative	Focuses on the use of facts and rules to describe a problem

Paradigms and Languages (3 of 4)

- Programmers generally find it useful to classify languages based on the types of projects for which they are used
- Some languages are used for Web programming; others for mobile apps, games, and enterprise applications
- Some of the most commonly used programming languages include:
  - Fortran
  - LISP
  - COBOL
  - BASIC
  - C
  - Prolog

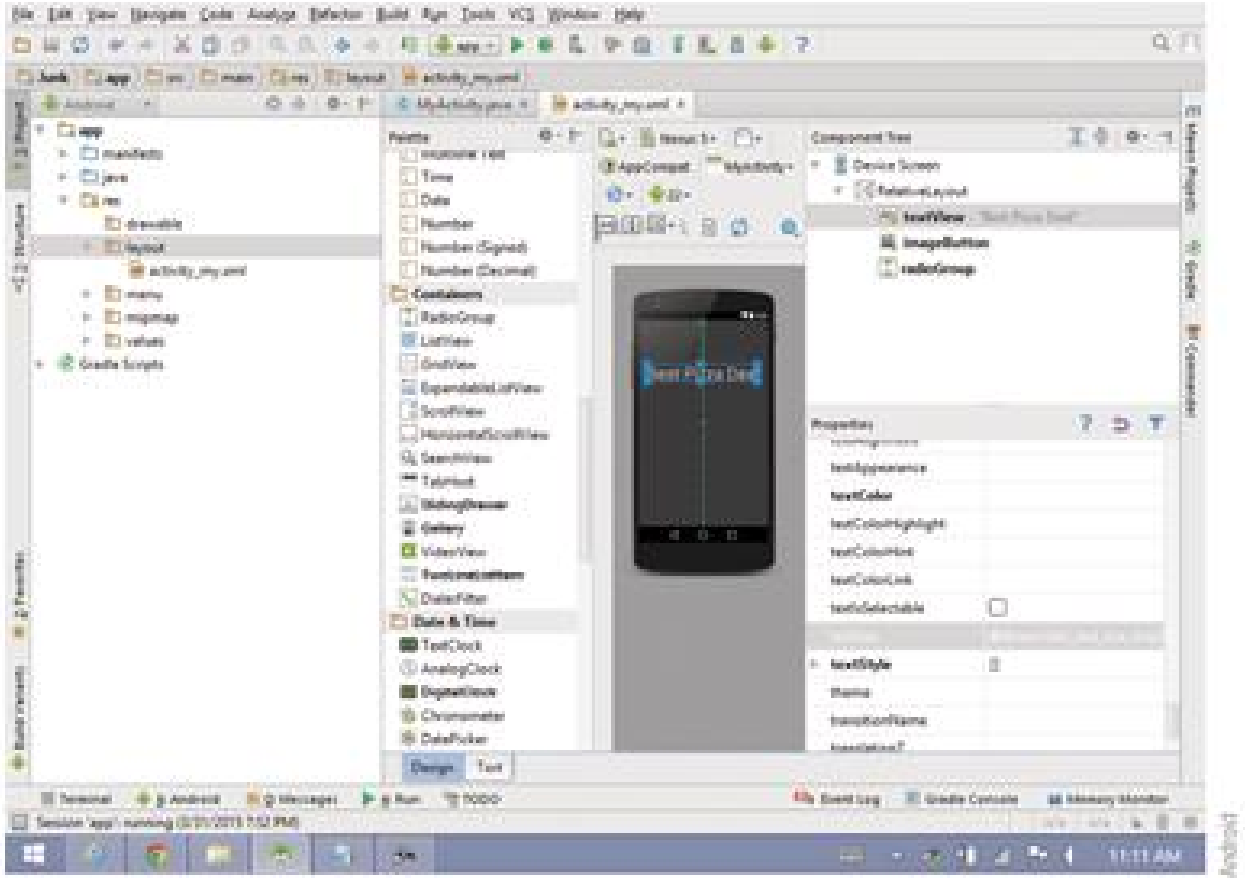
Paradigms and Languages (4 of 4)

- Ada
- C++
- Objective-C
- Perl
- Python
- Visual Basic (VB)
- Ruby
- Java
- JavaScript
- PHP
- C#
- Swift

Toolsets (1 of 2)

- Serious programmers typically download and install programming tools; their toolbox may include a compiler, a debugger, and an editor
- Programmers often download an *SDK* or *IDE* that contains a collection of programming tools
  - An **SDK** (software development kit) is a collection of language-specific programming tools that enables a programmer to develop applications for a specific computer platform
  - An **IDE** (integrated development environment) is a type of SDK that packages a set of development tools into a sleek programming application

Toolsets (2 of 2)



Section C: Procedural Programming

- Algorithms
- Pseudocode and Flowcharts
- Flow Control
- Procedural Applications

Section C: Objectives (1 of 2)

- Explain how algorithms relate to procedural programming
- List three tools that can be used to express an algorithm during the planning phase of program development
- Draw a diagram that illustrates how a function controls program flow
- Draw a diagram that illustrates how a selection control structure affects program flow
- Draw a diagram that illustrates how a repetition control structure affects program flow

Section C: Objectives (2 of 2)

- Describe at least two programming projects that are well suited for the procedural approach
- Explain the advantages and disadvantages of the procedural paradigm

Algorithms (1 of 4)

- The traditional approach to programming uses a **procedural paradigm** (sometimes called an imperative paradigm) to conceptualize the solution to a problem as a sequence of steps
- A programming language that supports the procedural paradigm is called a **procedural language**; these languages are well suited to problems that can easily be solved with a linear, step-by-step algorithm

Algorithms (2 of 4)

- An **algorithm** is a set of steps for carrying out a task that can be written down and implemented
- For example, the algorithm for making macaroni and cheese is a set of steps that includes boiling water, cooking the macaroni in the water, and adding the cheese sauce
- Algorithms are usually written in a format that is not specific to a particular programming language

Algorithms (3 of 4)

An important characteristics of a correctly formulated algorithm is that carefully following the steps guarantees that you can accomplish the task for which the algorithm was designed. If the recipe on a macaroni and cheese package is a correctly formulated algorithm, by following the recipe,, you should be guaranteed a successful batch of macaroni and cheese

Algorithms (4 of 4)

- An algorithm for a computer program is a set of steps that explains how to begin with known information specified in a problem statement and how to manipulate that information to arrive at a solution
- Algorithms are usually written in a format that is not specific to a particular programming language

Pseudocode and Flowcharts (1 of 2)

- You can express an algorithm in several different ways, including *structured English*, *pseudocode*, and *flowcharts*
  - **Structured English** is a subset of the English language with a limited selection of sentence structures that reflect processing activities
  - **Pseudocode** is a notational system for algorithms that is less formal than a programming language
  - A **flowchart** is a graphical representation of the way a computer should progress from one instruction to the next as it performs a task

Pseudocode and Flowcharts (2 of 2)

```
display prompts for entering shape, price, and size
input shape1, price1, size1
if shape1 = square then
        squareInches1 ← size1 * size1
if shape1 = round then
        squareInches1 ← 3.142 * (size1 / 2) ^2
squareInchPrice1 ← price1 / squareInches1
display prompts for entering shape, price, and size
input shape2, price2, size2
if shape2 = square then
        squareInches2 ← size2 * size2
if shape2 = round then
        squareInches2 ← 3.142 * (size2 / 2) ^2
squareInchPrice2 ← price2 / squareInches2
if squareInchPrice1 < squareInchPrice2 then
        output "Pizza 1 is the best deal."
if squareInchPrice2 < squareInchPrice1 then
        output "Pizza 2 is the best deal."
if squareInchPrice1 = squareInchPrice2 then
        output "Both pizzas are the same deal."
```

Flow Control (1 of 7)

- The key to a computer's ability to adjust to so many situations is the programmer's ability to control the *flow* of a program
- **Flow control** refers to the sequence in which a computer executes program statements
- During **sequential execution**, the first statement in the program is executed first, then the second statement, and so on, to the last statement in the program

Flow Control (2 of 7)

- Here is a simple program written in the Python programming language that outputs **This is the fist line.** and then outputs **This is the next line.:**

```
print (“This is the first line.”)

print (“This is the next line.”)
```

Flow Control (3 of 7)

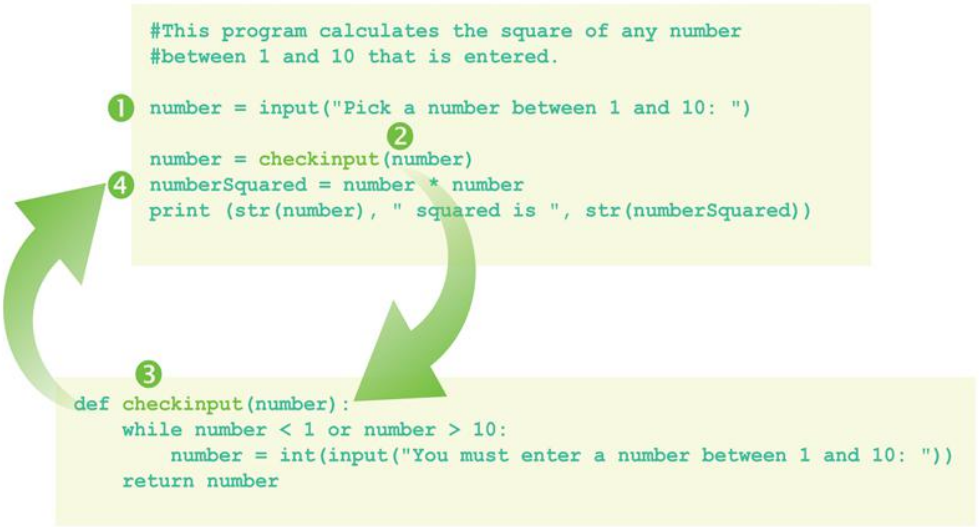
- **Control structures** are statements that specify the sequence in which a program is executed
- A **sequence control structure** changes the order in which instructions are carried out by directing the computer to execute an instruction elsewhere in the program
- In the following simple program, a **goto** command tells the computer to jump directly to the instruction labeled “Widget”:

```
print (“This is the first line.”)
goto Widget
print (“This is the next line.”)
Widget: print (“All done!”)
```

Flow Control (4 of 7)

- A **function** is a section of code that is part of a program but is not included in the main sequential execution path
- A sequence control structure directs the computer to the statements contained in a function—when the statements have been executed, the computer returns to the main program

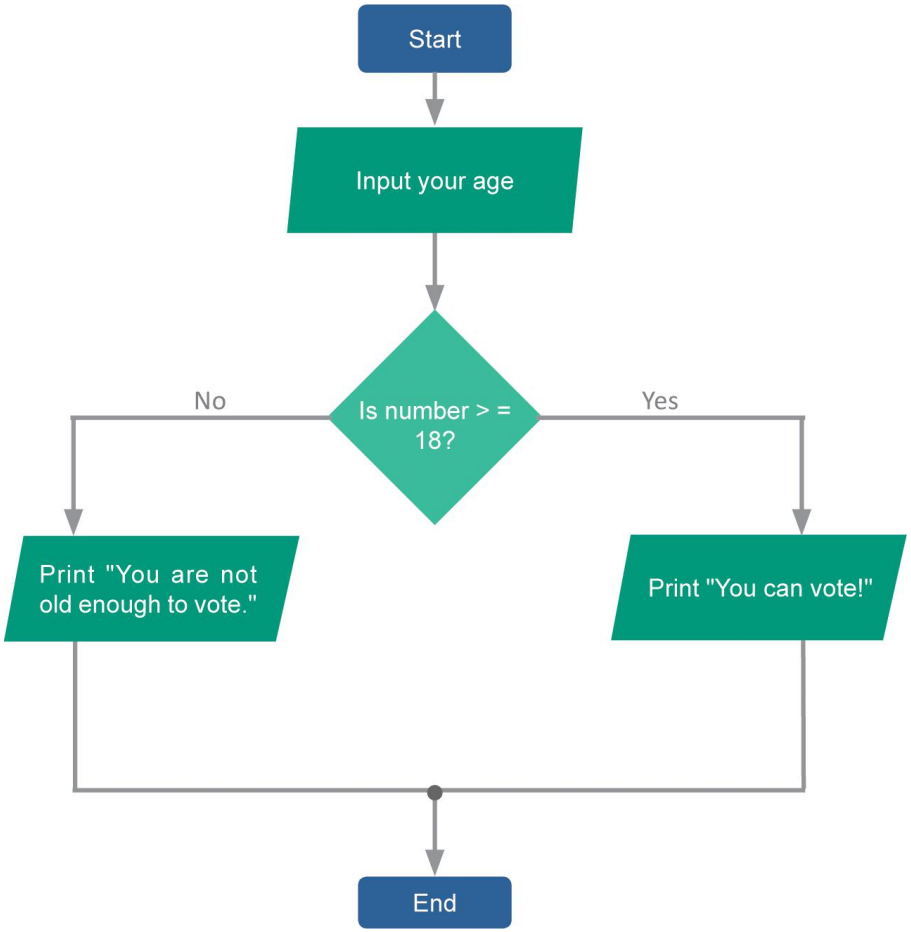
Flow Control (5 of 7)



1. The program asks the user to enter a number
2. Next, the program takes the number a jumps to the check input function
3. In the check input function, the program makes sure the number is between 1 and 10
4. A valid number between 1 and 10 is returned to the main program, where the number is squared and displayed

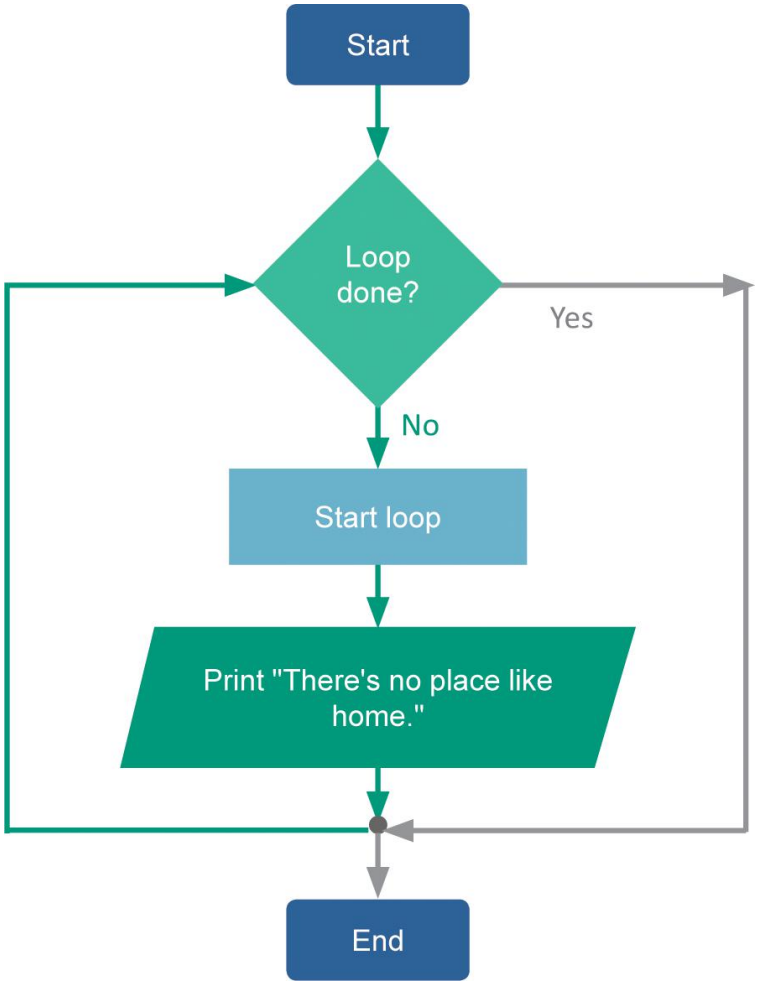
Flow Control (6 of 7)

- A **selection control structure** tells a computer what to do based on whether a condition is true or false. A simple example of a selection control structure is the if…else command



Flow Control (7 of 7)

- A **repetition control structure** directs the computer to repeat one or more instructions until a certain condition is met
- The selection of code that repeats is usually referred to as a **loop** or an **iteration**



Procedural Applications

- **Procedural languages** encourage programmers to approach problems by breaking the solution down into a series of steps; the earliest programming languages were procedural
- The procedural approach is best used for problems that can be solved by following a step-by-step algorithm
- The procedural approach and procedural languages tend to produce programs that run quickly and use system resources efficiently
- The procedural paradigm is quite flexible and powerful, which allows programmers to apply it to many types of problems

Section D: Object-Oriented Code

- Objects and Classes
- Inheritance
- Methods and Messages
- OO Program Structure
- OO Applications

Section D: Objectives (1 of 2)

- Explain the significance of objects and classes within the object-oriented paradigm
- Define an example class called People with at least four attributes
- Create two subclasses of People called Students and Instructors that inherit at least two attributes from the superclass
- Draw a UML diagram that illustrates the concept of inheritance
- Explain the relationship between methods and messages in an object-oriented program

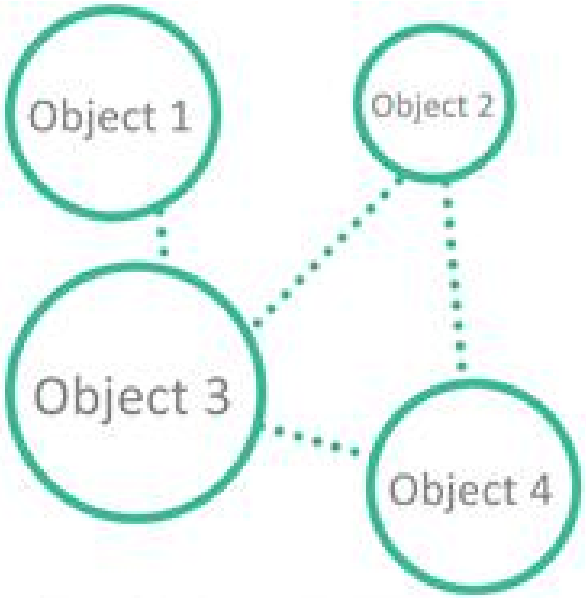
Section D: Objectives (2 of 2)

- Provide an example of polymorphism that relates to classes called People, Students, and Instructors
- Explain the significance of main() in an object-oriented program
- List at least three object-oriented programming languages
- Explain how the concept of encapsulation relates to abstraction

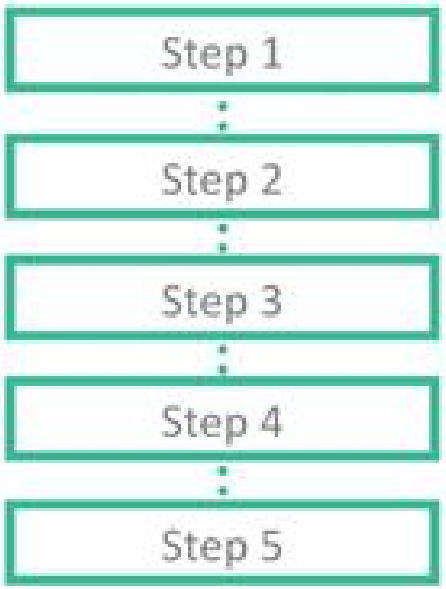
Objects and Classes (1 of 4)

- The **object-oriented (OO) paradigm** is based on objects and classes that can be defined and manipulated by a program's algorithms
- It is based on the idea that the solution for a problem can be visualized in terms of objects that interact with each other
- Rather than envisioning a list of steps, programmers envision a program as data objects that essentially network with each other to exchange data

Objects and Classes (2 of 4)



OBJECT-ORIENTED PARADIGM



PROCEDURAL PARADIGM

Objects and Classes (3 of 4)

- In the context of the OO paradigm, an **object** is a unit of data that represents an abstract or real-world entity, such as a person, place, or thing
- Whereas an object is a single instance of an entity, a **class** is a template for a group of objects with similar characteristics

Objects and Classes (4 of 4)

- A **class attribute** defines the characteristics of a set of objects
- Each class attribute generally has a name, scope, and data type; its scope can be defined as *public* or *private*
  - A **public attribute** is available for use by any routine in the program
  - A **private attribute** can be accessed only from the routine in which it is defined

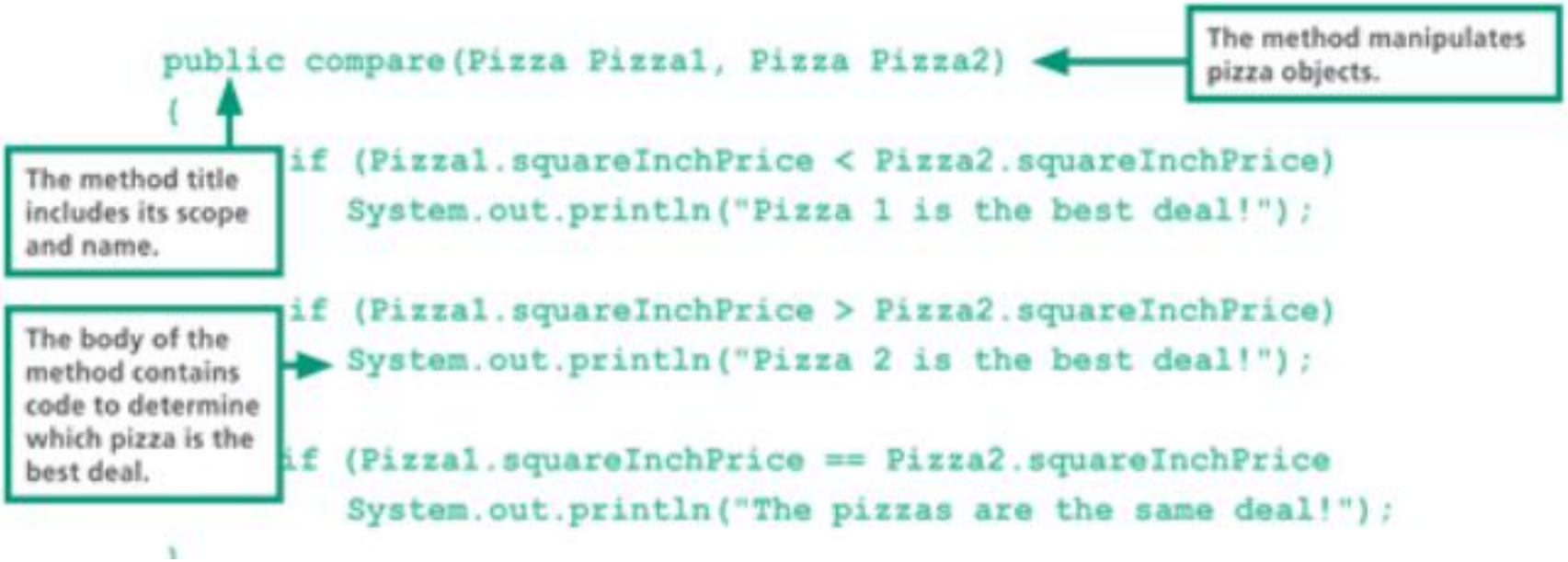
Inheritance

- In OO jargon, **inheritance** refers to passing certain characteristics from one class to other classes
- The process of producing new classes with inherited attributes creates a class hierarchy that includes *superclass* and *subclasses*
  - A **superclass** is any class from which attributes can be inherited
  - A **subclass** (or derived class) is any class that inherits attributes from a superclass

Methods and Messages (1 of 4)

- In an OO program, the objects interact; programmers specify how they interact by creating methods
- A **method** is one or more statements that define an action; the names of methods end in a set of parenthesis, such as compare() or getArea()
- The code that is contained in a method may be a series of steps similar to code segments in procedural programs

Methods and Messages (2 of 4)



Methods and Messages (3 of 4)

- A method is activated by a **message**, which is included as a line of program code that is sometimes referred to as a call
- In the OO world, objects often interact to solve a problem by sending and receiving messages
- **Polymorphism**, sometimes called overloading, is the ability to redefine a method in a subclass. It allows programmers to create a single, generic name for a procedure that behaves in unique ways for different classes

Methods and Messages (4 of 4)

OO Program Structure

- For classes and methods to fit together they must be placed within the structure of a Java program, which contains class definitions, defines methods, initiates the comparison, and outputs results
- The computer begins executing a Java program by locating a standard method called main(), which contains code to send messages to objects by calling methods

OO Applications

- In 1983, OO features were added to the C programming language, and C++ emerged as a popular tool for programming games and applications
- Java was originally planned as a programming language for consumer electronics, but it evolved into an OO programming platform for developing Web applications
- Most of today's popular programming languages, such as Java, C++, Swift, Python, and C#, include OO features

Section E: Declarative Programming

- The Declarative Paradigm
- Prolog Facts
- Prolog Rules
- Interactive Input
- Declarative Logic
- Declarative Applications

Section E: Objectives

- Describe how the declarative paradigm differs from the procedural and object-oriented paradigms
- Identify the predicate and arguments in a Prolog statement
- Explain the difference between a Prolog fact and a Prolog rule
- Identify constants and variables in a Prolog statement
- Explain how Prolog uses goals
- Draw a diagram to illustrate the concept of instantiation
- List two types of projects, other than those mentioned in the text, that would be good candidates for a declarative language such as Prolog

The Declarative Paradigm (1 of 2)

- The **declarative paradigm** describes aspects of a problem that lead to a solution
- Programmers using declarative languages write code that declares, or states, facts pertaining to a program

Procedural paradigms	Object-oriented paradigm:	Declarative paradigm:
Programs detail how to solve a problem	Programs define objects, classes, and methods	Programs describe the problem
Very efficient for number-crunching tasks	Efficient for problems that involve real- world objects	Efficient for processing words and language

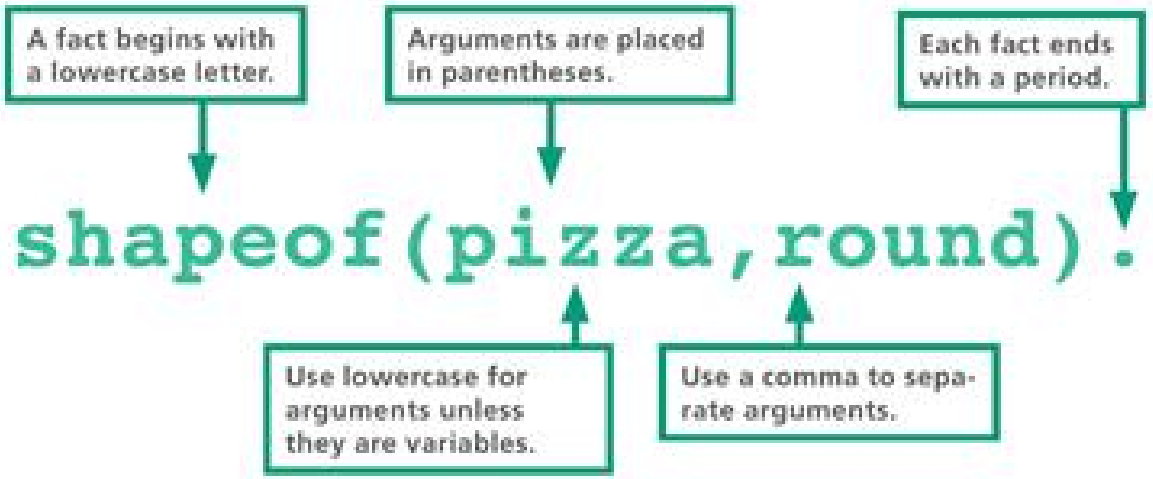
The Declarative Paradigm (2 of 2)

- The programming language Prolog uses a collection of facts and rules to describe a problem
- In the context of a Prolog program, a **fact** is a statement that provides the computer with basic information for solving a problem. A **rule** is a general statement about the relationship between facts

Prolog Facts (1 of 5)

- Prolog programming is easy to use; the punctuation mainly consists of periods, commas, and parentheses, so programmers don't have to track levels and levels of curly brackets
- The words in the parentheses are called **arguments**, which represent one of the main subjects that a fact describes

Prolog Facts (2 of 5)



Prolog Facts (3 of 5)

- The word outside the parentheses is called a **predicate** and describes the relationship between the arguments

`hates(joe,fish).`
Joe hates fish.

`playscardgame(joe,fish).`
Joe plays a card game called fish.

`name(joe,fish).`
Joe is the name of a fish.

Prolog Facts (4 of 5)

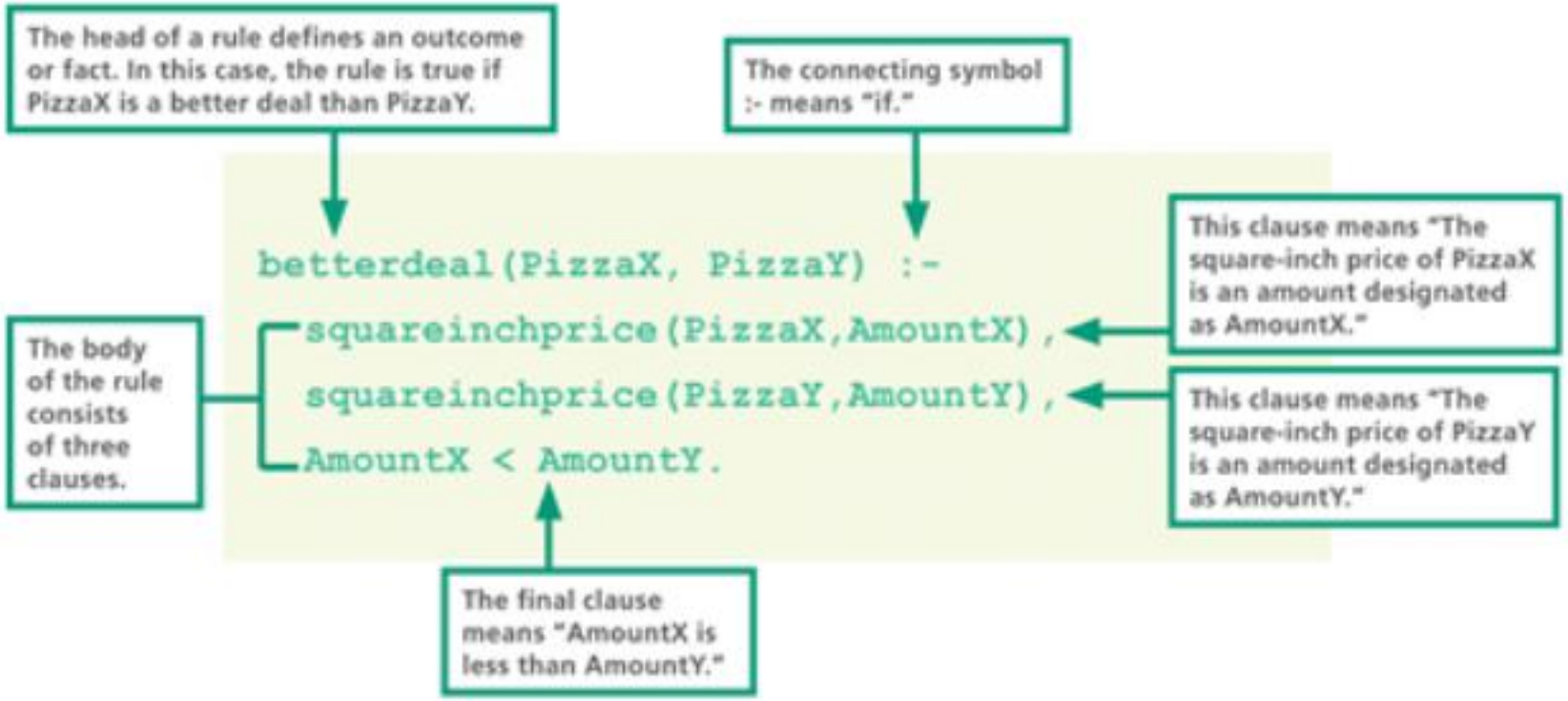
- Each fact in a Prolog program is similar to a record in a database, but you can query a Prolog program's database by asking a question, called a **goal**
- As an example, the following facts can easily be queried by entering goals:

```
priceof(pizza1,10).
sizeof(pizza1,12).
shapeof(pizza1,square).
priceof(pizza2,12).
sizeof(pizza2,14).
shapeof(pizza2,round).
```

Prolog Rules

- With just facts and goals, Prolog would be nothing more than a database
- The addition of rules gives programmers a set of tools to manipulate the facts
- Unlike other programming languages, the order or sequence of rules in a Prolog program is usually not critical to making sure the program works

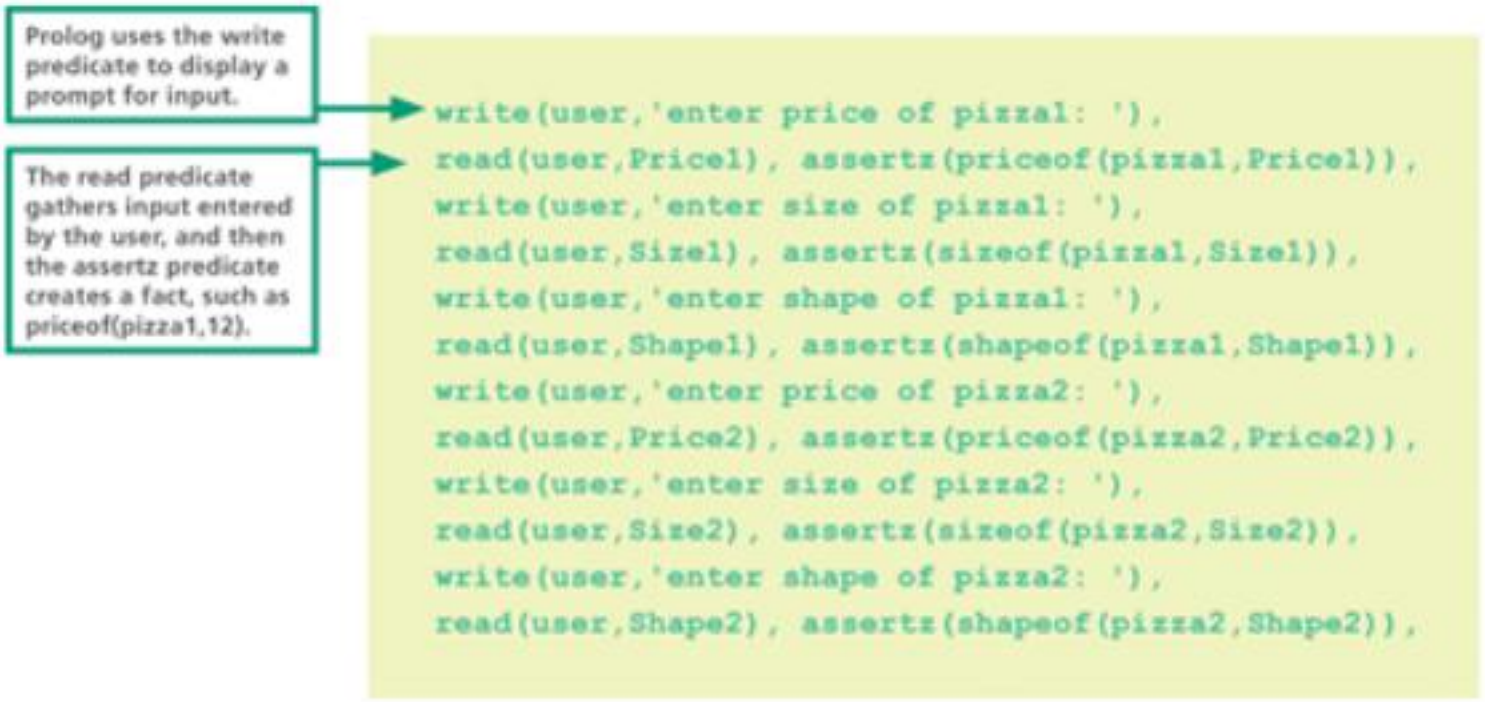
Prolog Facts (5 of 5)



Interactive Input (1 of 2)

- In order for programmers to collect input from the user, they can use *read* and *write* statements
- Read and write predicates collect user input
- Prolog uses the *write* predicate to display a prompt for input
- The *read* predicate gathers input entered by the user, and then creates a fact

Interactive Input (2 of 2)



Declarative Logic (1 of 2)

- Programmers need to determine how many conditions will apply to a program before starting to code facts and rules
- A **decision table** is a tabular method for visualizing and specifying rules based on multiple factors
- The decision table lays out the logic for the factors and actions and allows the programmer to see the possible outcomes

Declarative Logic (2 of 2)

Lowest Price?	Y	N	Y	N	Y	N	Y	N
Delivery Available?	Y	Y	N	N	Y	Y	N	N
Ready in less than 30 minutes?	Y	Y	Y	Y	N	N	N	N
Buy it?	Y	Y	N	N	Y	N	N	N

1. Each factor that relates to the pizza purchase is listed in the first column of the upper part of the table.
2. The remaining cells in the upper section of the table describe every possible combination of factors. This table has three factors for the decision. That means the table needs eight columns to cover all the combinations. This number is calculated as $2^{\text{number of factors}}$. In this case, there are three factors, so 2^3 is 2 * 2 * 2, or 8.
3. The lower part of the table lists actions that are taken based on the factors. The programmer looks at each column of Ys and Ns to decide if the action should be taken. For example, in the column filled with Ys, the action would be to buy the pizza

Declarative Applications

- As a general rule, declarative programming languages are most suitable for problems that pertain to words and concepts rather than to numbers
- Declarative languages offer a highly effective programming environment for problems that involve words, concepts, and complex logic
- One of the disadvantages of declarative languages is that they are not commonly used for production applications—today's emphasis on the OO paradigm has pushed declarative languages out of the mainstream, both in education and in the job market