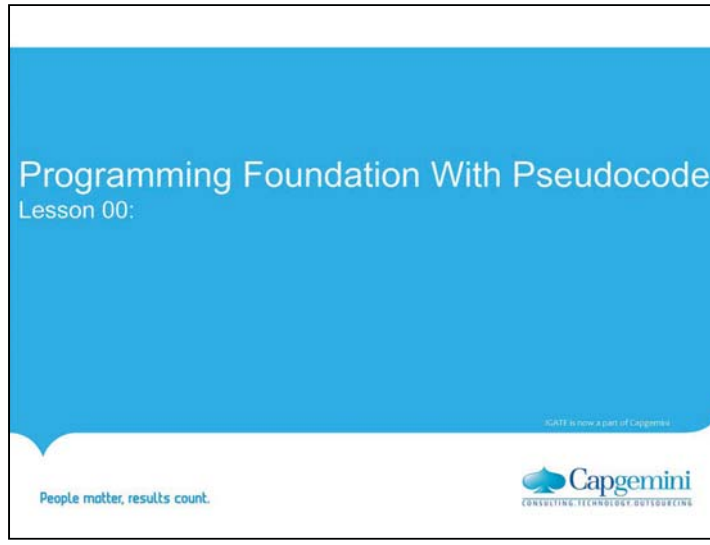


©2016 Capgemini. All rights reserved.
The information contained in this document is proprietary and confidential.
For Capgemini only.

Course Goals and Non Goals

- Course Goals
  - To learn about how to write good program by understanding concepts like
    - Readability
    - Maintainability
    - Modularity
    - Defensive programming
    - Algorithm analysis and design
  - To learn about how to write pseudocode in design phase
  - To develop robust programs by performing Code Reviews and Unit Testing (test cases/results)
  - Understanding Software testing
- Course Non Goals
  - To learn any specific language features in this course. (Language features will be covered in subsequent modules.)

Capgemini

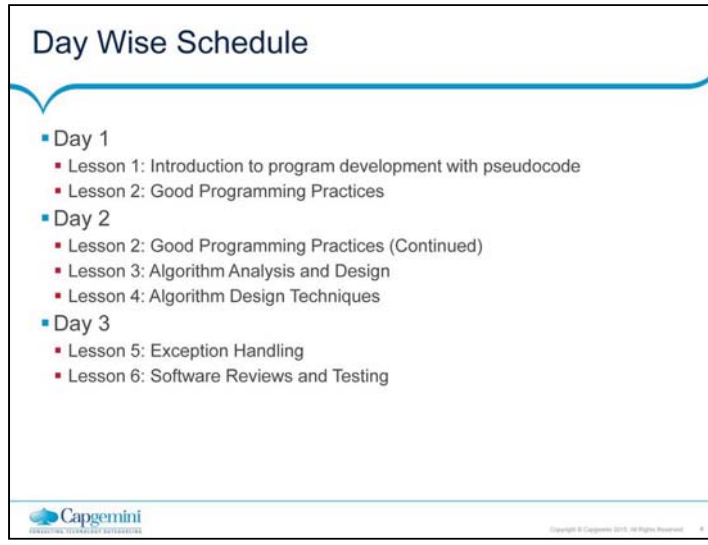
Copyright © Capgemini 2015. All Rights Reserved.

Intended Audience

- Novice Developers

Capgemini

Copyright © Capgemini 2015. All Rights Reserved.


Day Wise Schedule
Day 1
Lesson 1: Introduction to program development with pseudocode
Lesson 2: Good Programming Practices
Day 2
Lesson 2: Good Programming Practices (Continued)
Lesson 3: Algorithm Analysis and Design
Lesson 4: Algorithm Design Techniques
Day 3
Lesson 5: Exception Handling
Lesson 6: Software Reviews and Testing
Capgemini

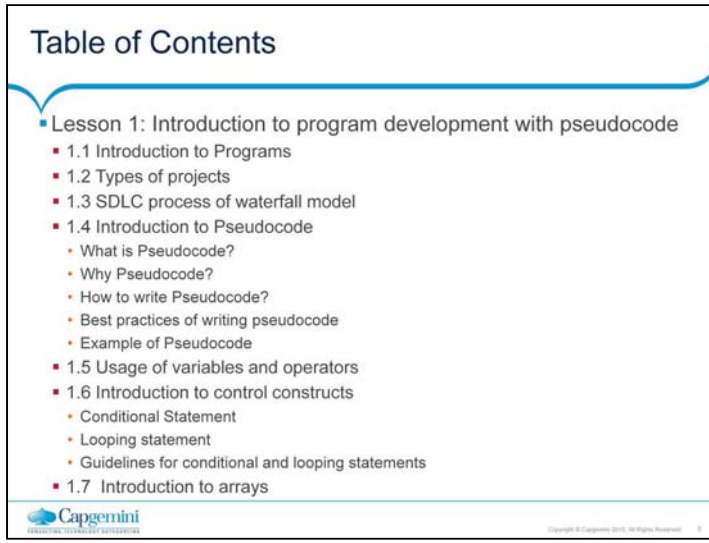


Table of Contents
Lesson 1: Introduction to program development with pseudocode
1.1 Introduction to Programs
1.2 Types of projects
1.3 SDLC process of waterfall model
1.4 Introduction to Pseudocode
What is Pseudocode?
Why Pseudocode?
How to write Pseudocode?
Best practices of writing pseudocode
Example of Pseudocode
1.5 Usage of variables and operators
1.6 Introduction to control constructs
Conditional Statement
Looping statement
Guidelines for conditional and looping statements
1.7 Introduction to arrays
Capgemini

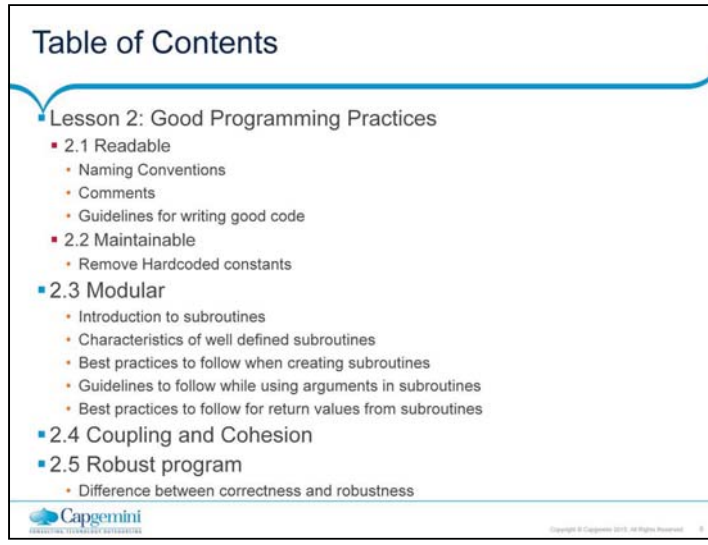


Table of Contents
Lesson 2: Good Programming Practices
2.1 Readable
Naming Conventions
Comments
Guidelines for writing good code
2.2 Maintainable
Remove Hardcoded constants
2.3 Modular
Introduction to subroutines
Characteristics of well defined subroutines
Best practices to follow when creating subroutines
Guidelines to follow while using arguments in subroutines
Best practices to follow for return values from subroutines
2.4 Coupling and Cohesion
2.5 Robust program
Difference between correctness and robustness
Capgemini

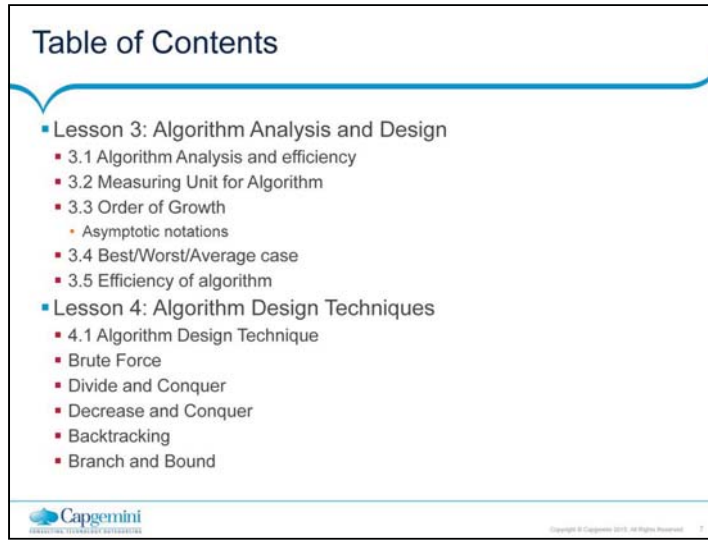


Table of Contents
Lesson 3: Algorithm Analysis and Design
3.1 Algorithm Analysis and efficiency
3.2 Measuring Unit for Algorithm
3.3 Order of Growth
Asymptotic notations
3.4 Best/Worst/Average case
3.5 Efficiency of algorithm
Lesson 4: Algorithm Design Techniques
4.1 Algorithm Design Technique
Brute Force
Divide and Conquer
Decrease and Conquer
Backtracking
Branch and Bound
Capgemini

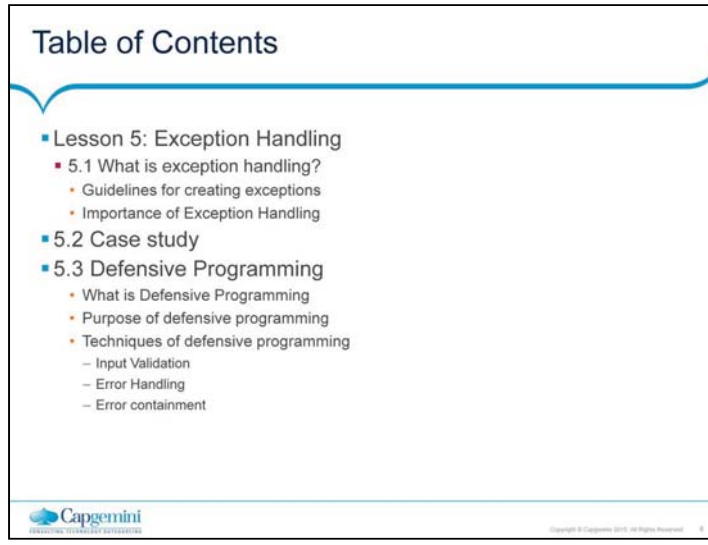


Table of Contents
Lesson 5: Exception Handling
5.1 What is exception handling?
Guidelines for creating exceptions
Importance of Exception Handling
5.2 Case study
5.3 Defensive Programming
What is Defensive Programming
Purpose of defensive programming
Techniques of defensive programming
Input Validation
Error Handling
Error containment
Capgemini

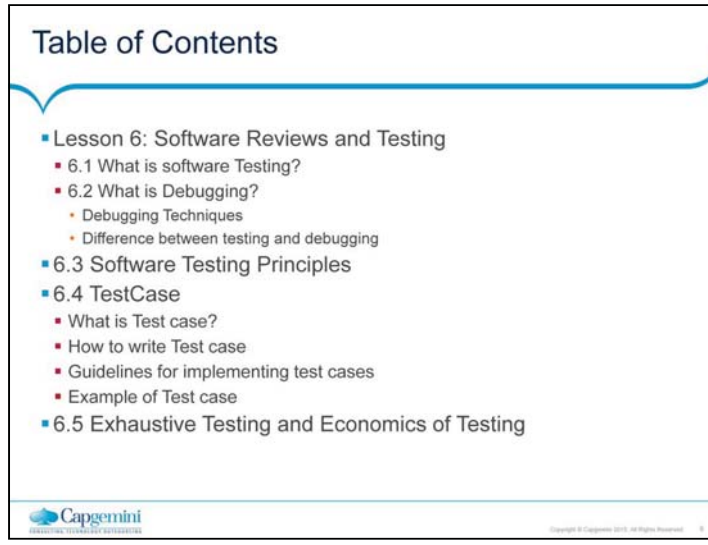


Table of Contents
Lesson 6: Software Reviews and Testing
6.1 What is software Testing?
6.2 What is Debugging?
Debugging Techniques
Difference between testing and debugging
6.3 Software Testing Principles
6.4 TestCase
What is Test case?
How to write Test case
Guidelines for implementing test cases
Example of Test case
6.5 Exhaustive Testing and Economics of Testing
Capgemini

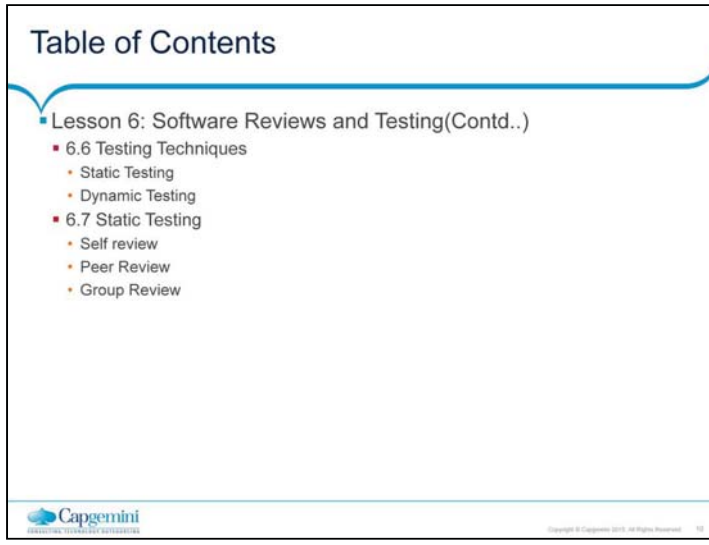


Table of Contents
Lesson 6: Software Reviews and Testing(Contd..)
6.6 Testing Techniques
Static Testing
Dynamic Testing
6.7 Static Testing
Self review
Peer Review
Group Review
Capgemini

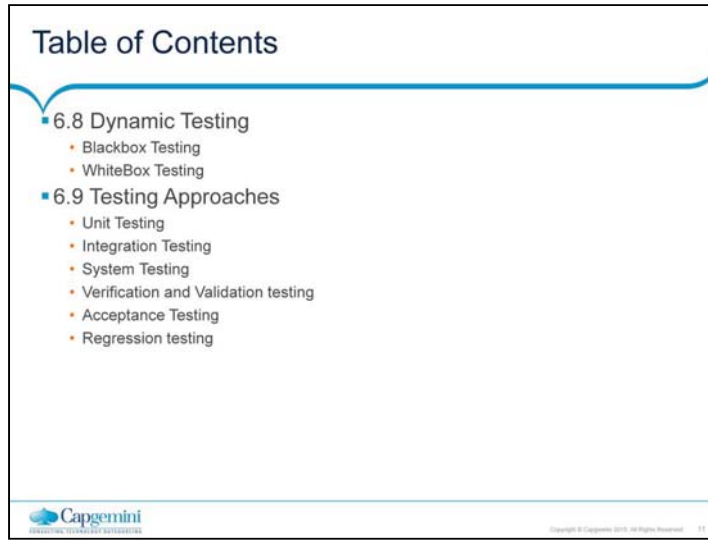
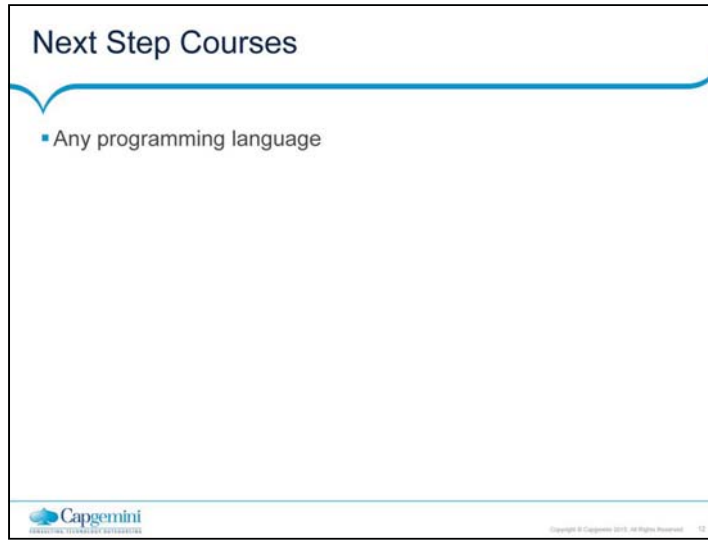
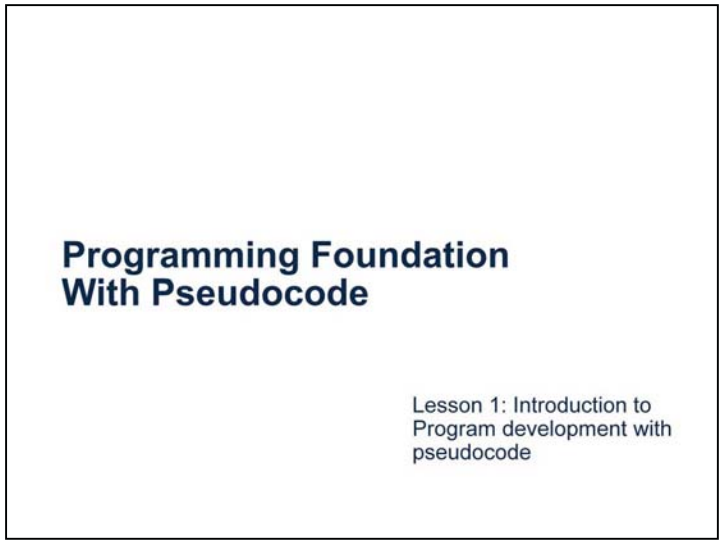


Table of Contents
6.8 Dynamic Testing
Blackbox Testing
WhiteBox Testing
6.9 Testing Approaches
Unit Testing
Integration Testing
System Testing
Verification and Validation testing
Acceptance Testing
Regression testing
Capgemini


Next Step Courses
Any programming language
Capgemini

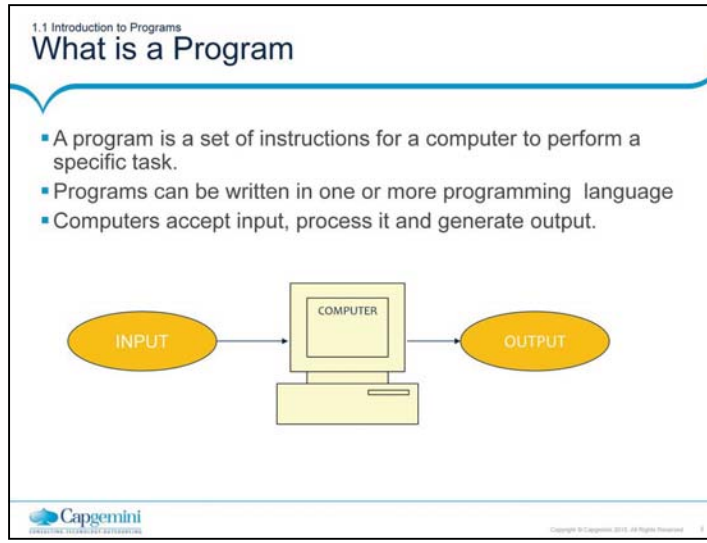

Programming Foundation
With Pseudocode
Lesson 1: Introduction to
Program development with
pseudocode

Lesson Objectives

- To Understand the following concepts
  - Introduction to programs
  - Types of projects
  - SDLC process of waterfall model
  - Introduction to Pseudocode
  - Usage of variables and operators
  - Introduction to control constructs

Capgemini

Copyright © Capgemini 2015. All Rights Reserved



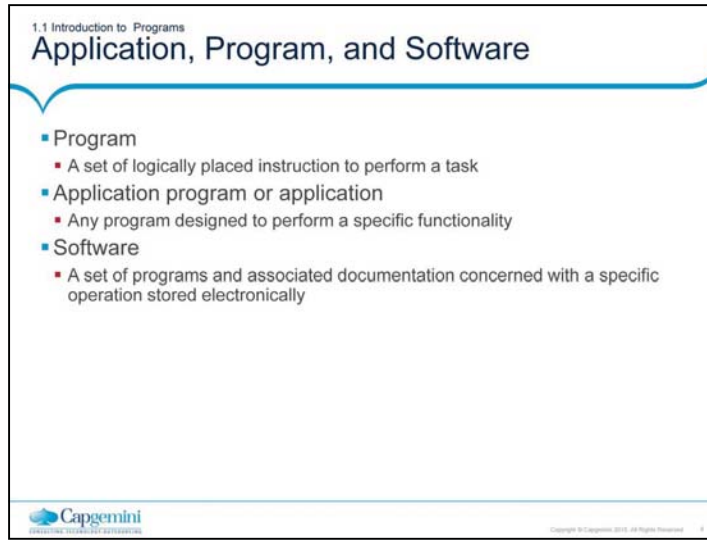
What is a Program?

Set of instructions for a computer to perform a specific task. Programs can be written in one or more programming language.

Example: Program to receive employee details as an input, then calculate and display the gross and net salary of an employee.

What is Programming Language?

- Used to feed instructions to the computer
- Can be categorized as Machine language, Assembly language, Compiled Languages, Interpreted Languages, Object Oriented Languages … etc
- Languages which are more simpler, easier are referred as High-Level Languages
- Low level languages provides little or no abstraction to the internal working of microprocessor



Program:

Set of ordered instructions that enable a computer to carry out a specific task. A collection of instructions that tell the computer what to do.

Ex: Program to find a prime number, Program to print employee pay slip etc..

Application :

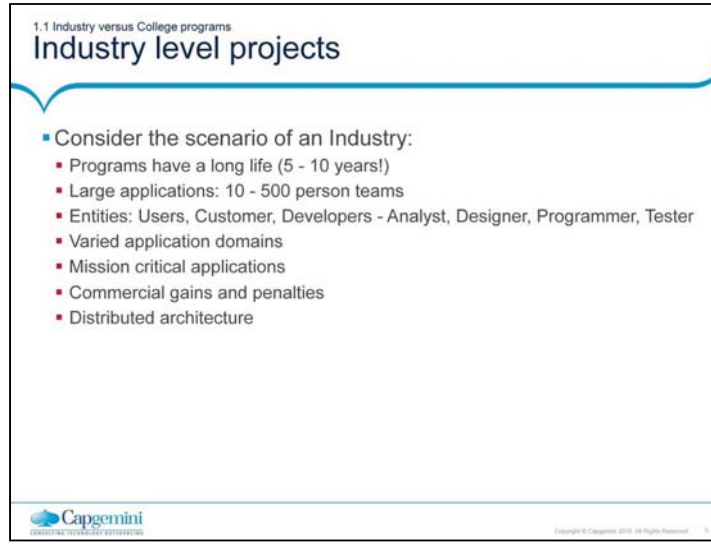
Any program designed to perform a specific function

Ex : Notepad, M S Paint etc

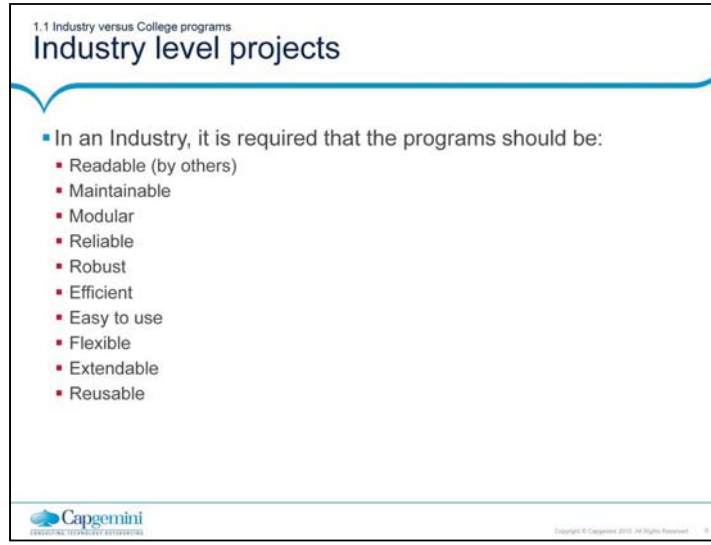
Software:

A set of programs and associated documentation concerned with a specific operation stored electronically

Ex : Microsoft Office, Oracle etc

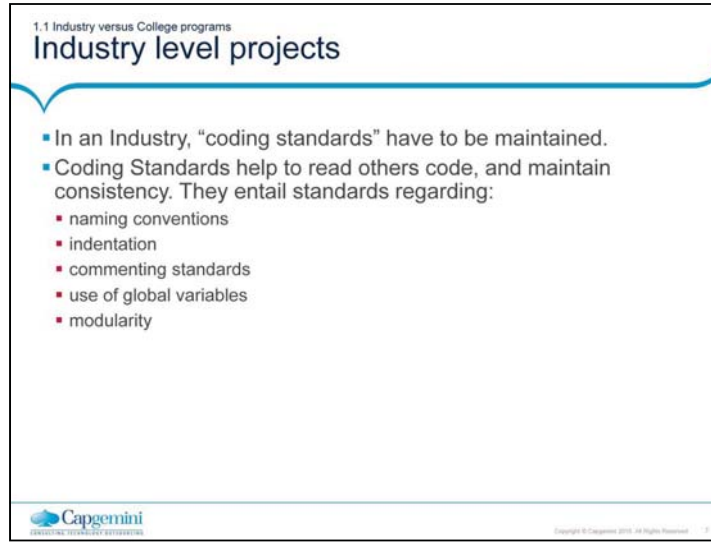


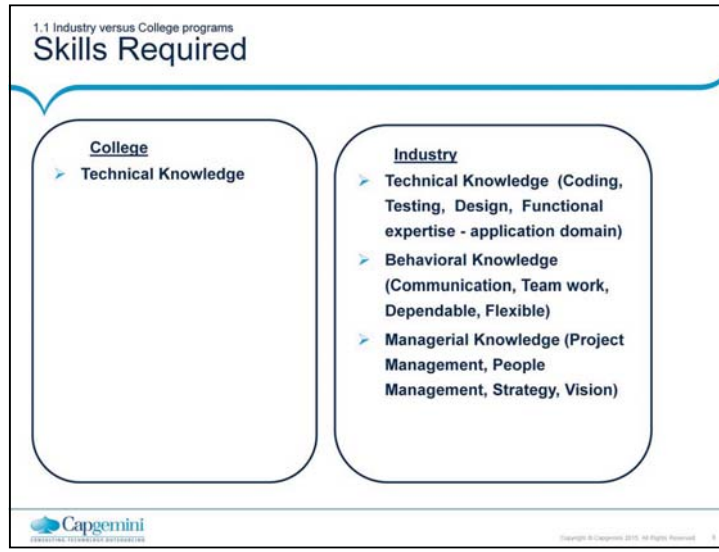
- Consider the scenario of a College:
  - Throw away programs
    - These programs are not used by any one later
  - Same Users / Developers
  - Small assignments: 1-2 person teams
  - No commercial angle
  - Familiar application domain
  - Low criticality
  - Traditional single-machine architecture



Industry level programs should be

- Readable : Easy to read and understand the code at any time since lifetime of projects will be longer in terms of years.
- Maintainable : If the program is easy to understand and If it is easy to modify then the program is called as maintainable
- Modular : A small unit of code for a single purpose
- Reliable : Reliability describes about the ability of a system will work perfectly as stated without failure or error
- Robust : The ability of a system to continue operating despite abnormalities in input, calculations, etc.
- Efficient : A task which gets done in the specified time with desired quality
- Easy to use: Program which is easy to use by the end users
- Flexible
- Extendable : If the additional features of a program is possible to be included without any side effects, then the program is called as extendable.
- Reusable : If the task written in program is called multiple times, then the program is reusable.


1.1 Industry versus College programs
Industry level projects
In an Industry, “coding standards” have to be maintained.
Coding Standards help to read others code, and maintain consistency. They entail standards regarding:
naming conventions
indentation
commenting standards
use of global variables
modularity
Capgemini

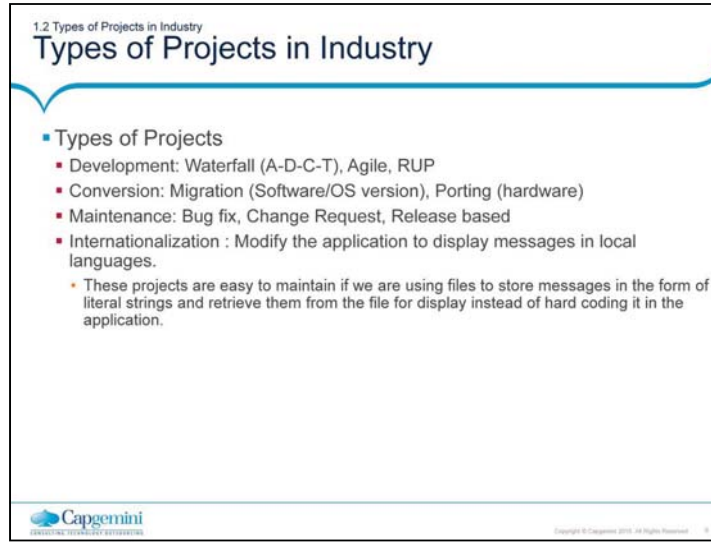


A person require much more technical knowledge for writing industry level applications as compared to writing programs in the college.

In college having good technical knowledge for a particular technology is enough to write program but in industry along with technical knowledge you require some other knowledge like

- What are good programming practices?
- How to write test cases?
- You also should have domain knowledge.
- As we need to work in a team in industry, you also need to improve your communication knowledge. You should be dependable and flexible.
- To reach at managerial level, you need to acquire people management skill, Project management skill over a period of time along with the technical skills

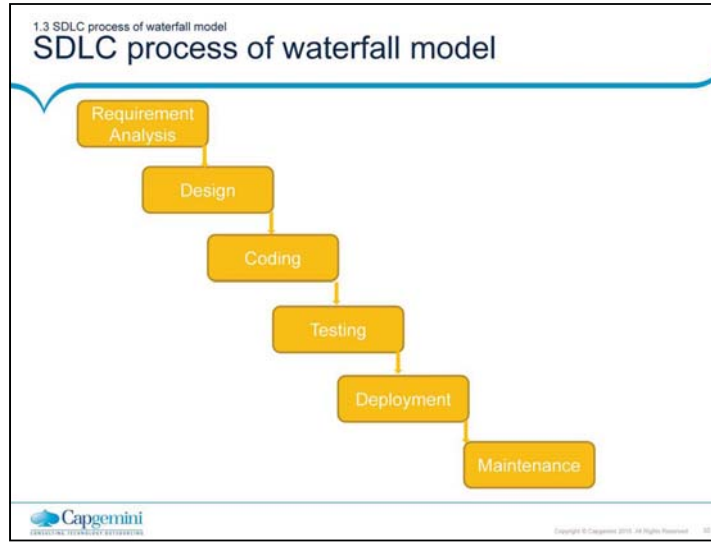
.



In an industry there are various types of projects like

- **Development Projects:** In these projects we may use different types of life cycles like Waterfall model, Agile, RUP (Rational Unified Process) you need to be familiar with these terms

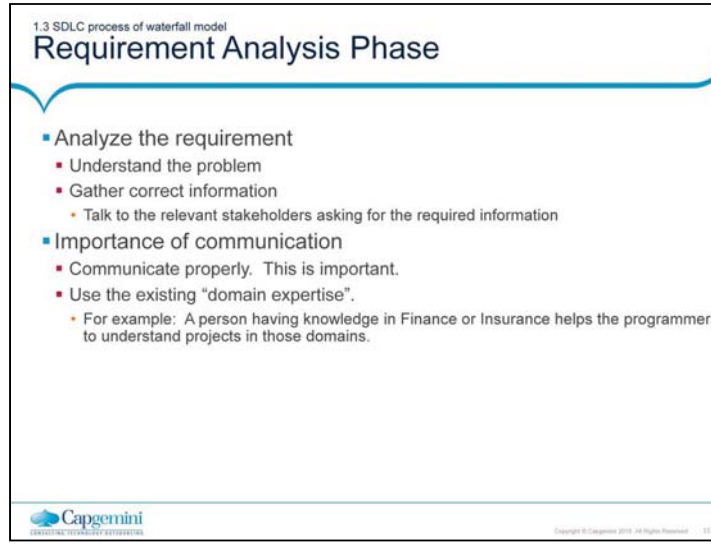
- **Conversion Projects**
  - Migration - It refers to projects like upgrading a software to different version of the OS or DBMS systems
  - Porting – If there is major change in the system like, Hardware Platform has changed.
- **Maintenance projects**
  - Bug fix – If there is an unwanted behavior exists in an existing system due to bug(Problem), then fix the bug by doing changes.
  - Change request – If there are any changes in the functionality are requested from customer such projects are called as changed request type of project.
  - Release base – Products for which periodically the new release of the product is made
- **Internationalization :** Normally most of the applications display messages to user in English but in some cases we may need to change the application to display messages in local language based on the location where we are using it. In such type of projects we need to change the code accordingly. If we have hard coded the messages in the applications then maintenance will be the tedious activity. Hence the good solution for it is store the messages in a file in the form of literal strings and use these files for displaying messages.



Waterfall Model: The waterfall model is a sequential software development model in which development is seen as flowing steadily downwards (like a waterfall) through the phases of requirements analysis, design, coding, testing (validation), deployment, and maintenance.

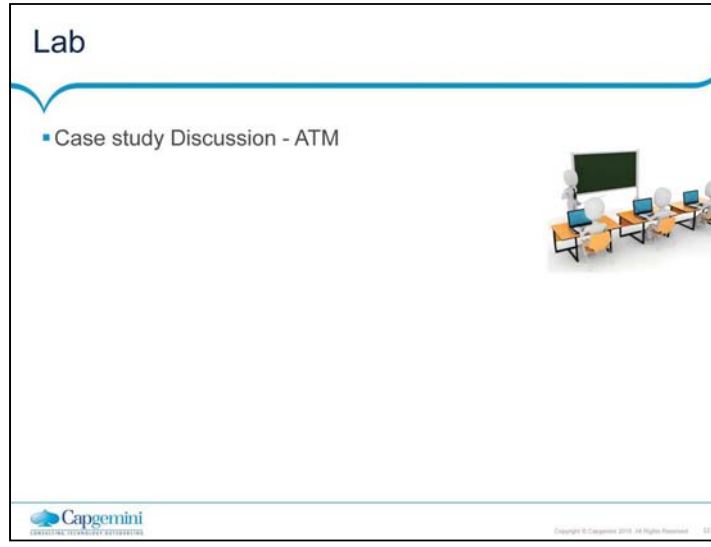
SDLC process of Waterfall Model:

- **Requirement Analysis :** Identify all the requirements. After requirement gathering, analyze the requirements for identifying their validity and the possibility of incorporating the requirements in the system to be developed.
- **Design:** It is a process of creating a detailed specification for a software module . It involves algorithmic design and other implementation specific approaches for a s/w component such as modularity , control hierarchy,, data structures etc. Designers/Technical leads ,senior developers , architects are involved in this phase
- **Coding :** Main objective of this phase is to translate the software design into code , each component identified in design is implemented as a program module following coding guidelines
- **Testing :** Process of checking what's been developed against the requirement.
- **Deployment :** Process of bringing the system into production environment
- **Maintenance :** The maintenance phase involves making changes to hardware, software, and documentation to support its operational effectiveness.



Analyze the requirement:

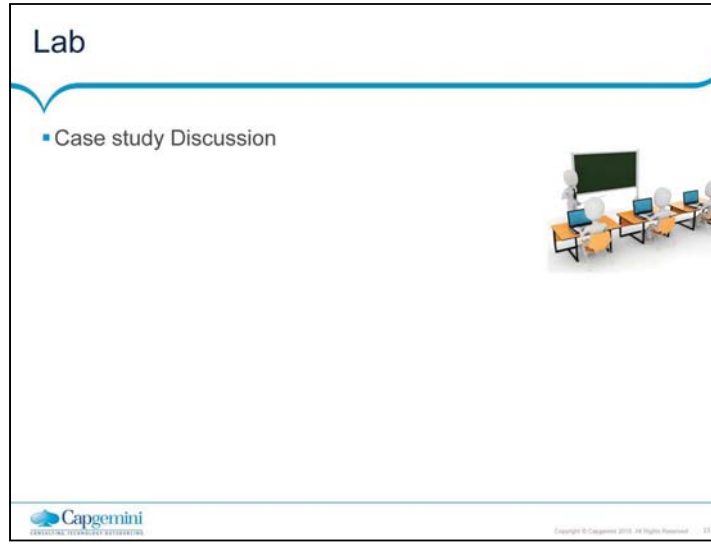
- **Understand the Problem:**
  - Interpret the problem in your own words
  - Determine the outputs required
  - Identify the inputs required to obtain the desired output
  - List out the clarifications required
  - List the assumptions made
  - List the constraints / limitations
- **Gather Correct Information:** Gather the required information from the concern stakeholders by communicating with them.
- **Importance of proper communication:** User talks in language of business, and Programmer talks in the language of technology. Since there is a big gap between these languages, understanding the requirements properly is very important. This is possible through proper communication. The communication exercise is as follows:
  - Consider income tax calculation logic is written by you. If you want to check whether the Income Tax calculation written in the program, is correct as per the current Tax laws? Where can you get this information?
  - Draft an email to the relevant stakeholders, requesting for the required information.



Case Study: The ATM application aims at performing ATM transactions and balance enquiry of an existing account holder in a user friendly way

Following is a list of functionalities of the system. Wherever, the description of functionality is not adequate; you can make appropriate assumptions and proceed.

- The user is requested for the card number and his personal pin number for authentication purpose.
- After authenticating the user, the application requests the user to choose any one of the following options:
    - BALANCE ENQUIRY
    - CASH WITHDRAWL
    - MINI STATEMENT
    - QUIT.
- When the user chooses one of the above options, say '1', the balance of the user is retrieved and displayed. The application further requests the user whether he/she wants the report to be generated and responds accordingly.
- When the user chooses '2', transaction is performed based on the request of the user with the help of the transaction file. Thus after the transaction is complete the user's account is updated.



Case Study – ATM (Contd..)

- When user wants to generate a record for his/her last 5 DAYS transaction, mini statement is opted where details retrieved from Transaction History File.

Updating balance is done when recharge is successful in Master and Transaction File. Thus, the application requests the user for further processing and responds based on the input from the user.

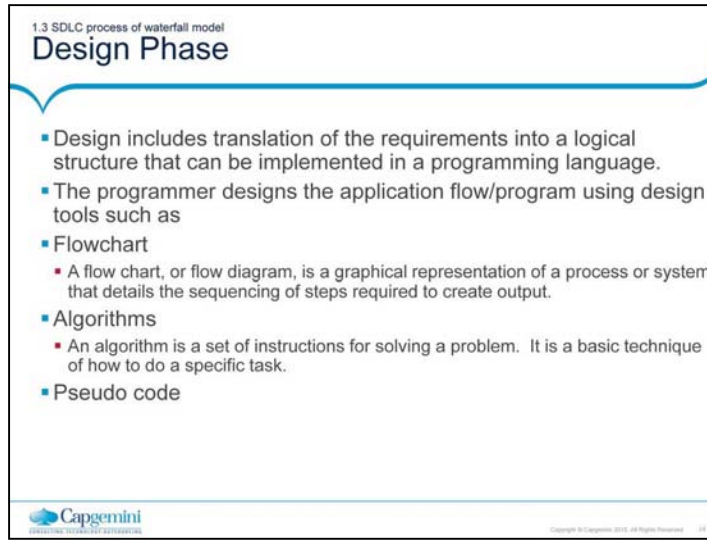
Case Study 1: In a firm there are 10 salespeople and incentive is paid on the portion of sales that exceeds two thirds of the average sales of the group. List the salesperson receiving incentives along with their incentive amount. Incentive amount is 20% the amount of sales that exceeds the two thirds of the average.

Case Study 2: All candidates have to take three tests. Selection for the interview round is done based on the scores of all the three tests. The individual score in each test has to be greater that 75 and the average score across the three tests should be a minimum of 80. An interview call letter is to be sent to candidates who have been selected and a reject letter is to be sent to the others. The interview stage involves two rounds:

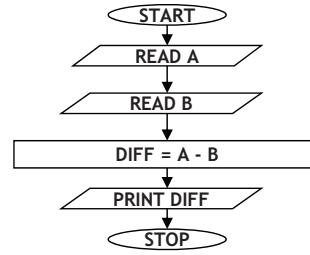
- *Round 1:* qualifying criterion: rating of greater than five on a scale of 1 to 10
- *Round 2:* qualifying criterion: rating of greater than seven on a scale of 1 to 10Candidates go through both rounds of interview.

Selected candidates are sent offer letters and the rest are sent reject letters.

Represent the logic for finding the number of candidates who have been sent interview call letters, who have been sent reject letters and who have been sent offer letters.



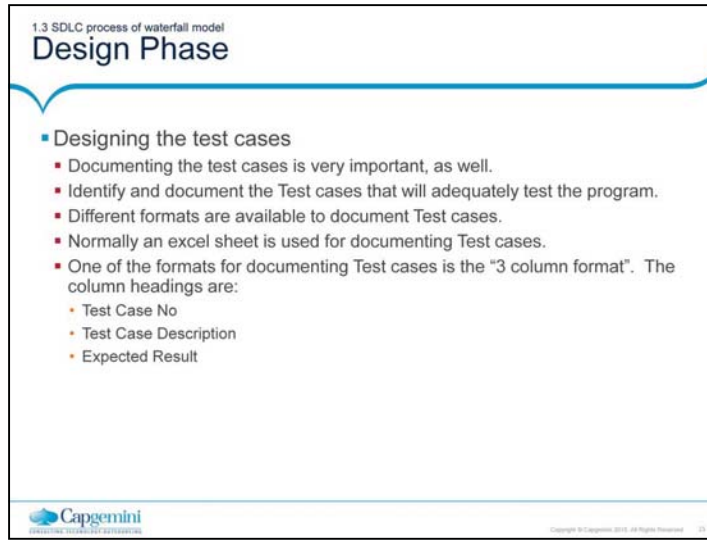
Flowchart: A flow chart, or flow diagram, is a graphical representation of a process or system that details the sequencing of steps required to create output. A typical flow chart uses a set of basic symbols to represent various functions, and shows the sequence and *interconnections* of functions with lines and arrows. Ex: Flow chart to calculate difference b/w two numbers:



Algorithm: An algorithm is a set of instructions for solving a problem. It is a basic technique of how to do a specific task. It takes input, processes it according to a set of instructions, and generates output. An algorithm must provide correct output for every possible input condition. An algorithm must have a definite end point so that when the input has been processed and the desired output achieved, the process stops.

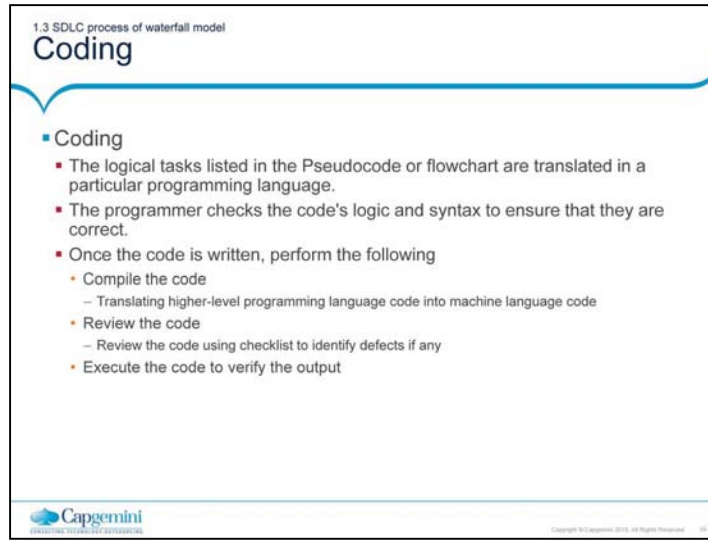
Example: Algorithm to calculate the difference between two numbers

```
Accept two numbers as num1 and num2
Find the difference between num1 and num2
PRINT DIFFERENCE
```



Test case
"A set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement."
See the sample test cases given below for checking valid date.

Test Case ID	Test Condition / Scenario	Test Steps	Input/Test Data	Expected Result
TC_1	To check the valid Date	Enter date	10/04/2005	Date should be accepted.
TC_2	To Check the Date when days are entered as Alphabets	Enter day as alphabets. Enter valid month & year	Ten/04/2005	"Enter Valid date"
TC_3	To Check the Date when days are entered as Special Characters	Enter day as special char. Enter valid month & year	!)/04/2005	"Enter valid date"

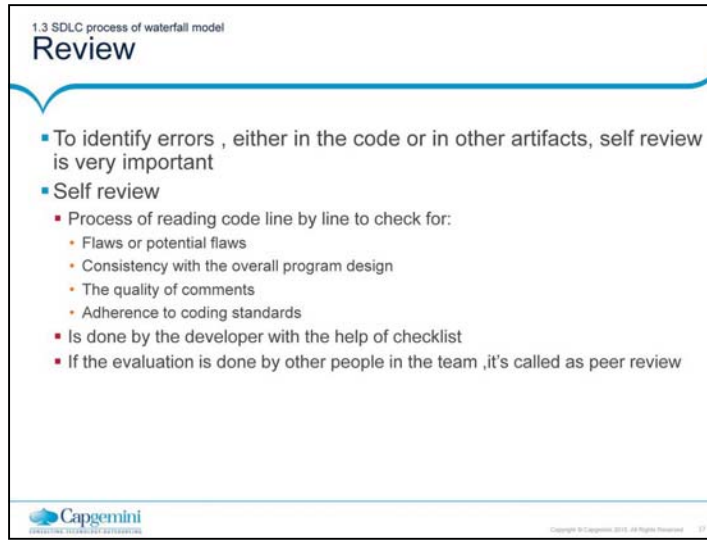


Compiling the code:

- Computers can execute machine language code only.
- The process of translating higher-level programming language code into machine language code is called compiling the program.
  Special programs, called compilers and interpreters, perform this task.
- Code that has errors in it cannot be compiled. The compiler will flag the error, and the programmer must fix the error before trying to compile the code again

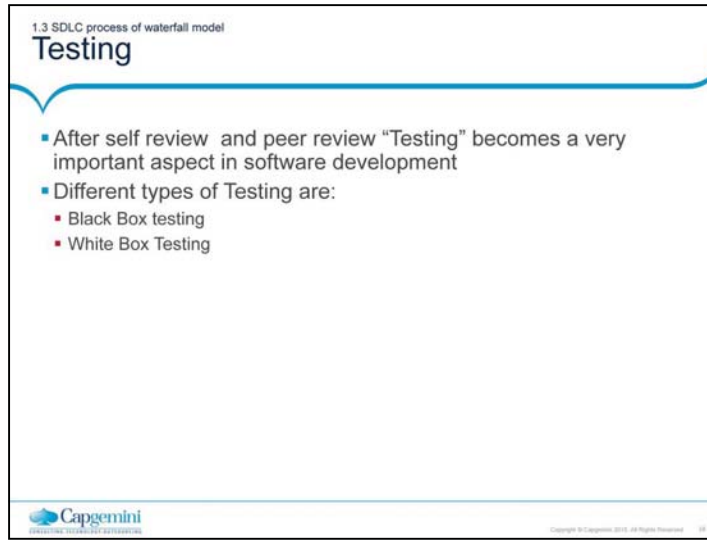
Execute the program and verify the output:

- Once a program is compiled, it can be executed.
- The programmer checks the program's output to ensure that it is correct
- Logic of the program can be checked by using sample inputs and comparing the output obtained with the expected output identified during creation of test cases.



Self review is a process done by the author himself with the aid of tools like checklist, review guidelines, etc..

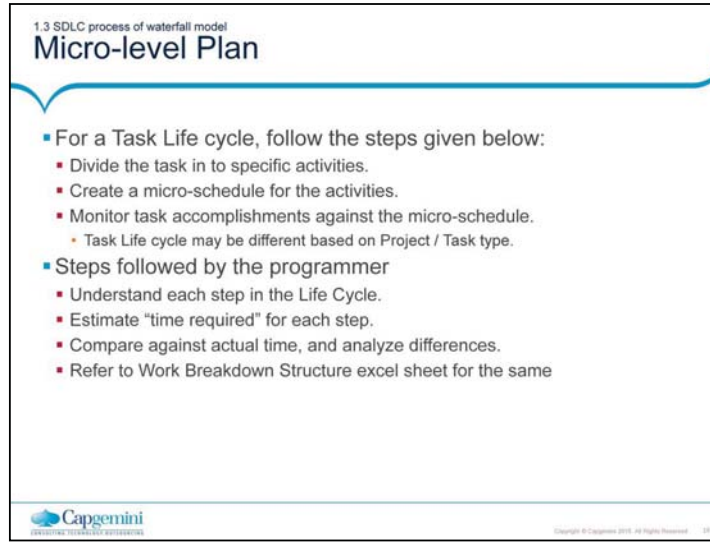
Checklist is a tool with which review can be performed. Checklist is available as an excel document used to verify that all the requirements mentioned in requirement specification is covered and best practices, coding standards are followed.



Different types of testing:

- **Black Box testing** is a software testing method focused on testing whether the input is properly accepted, and output is correctly produced in an application. For an Example, if you want to validate the given input, then use black box testing.

- **White Box testing** is also a software testing method focused on testing the internal structure of the code. For an example, any unreachable path is identifiable using white box testing.

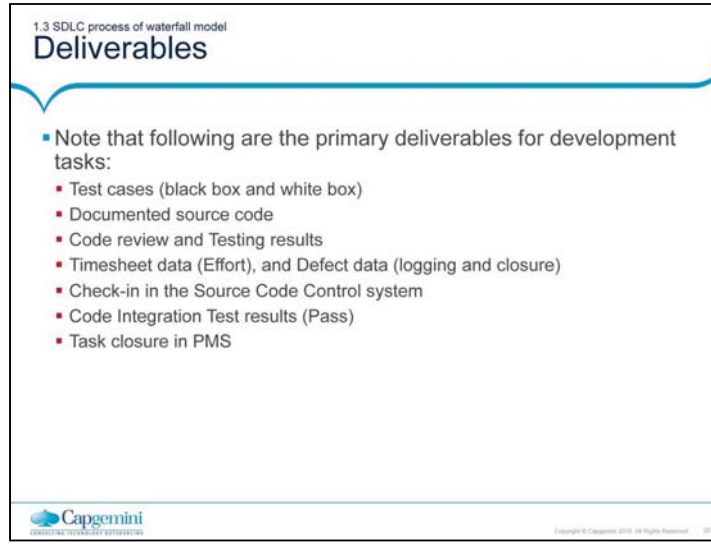


If any of the task is assigned to a programmer then to complete it in time follow the task life cycle.

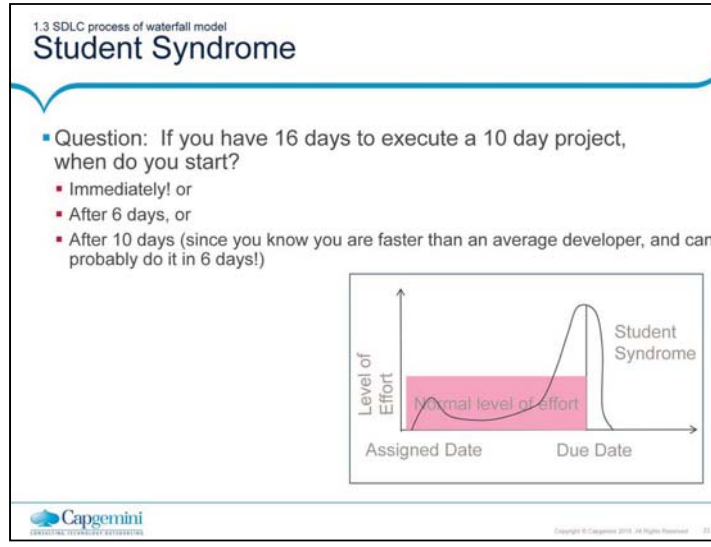
1. The task life cycle means, divide the task in to smaller activities which should be performed in the sequence to complete the task.
2. Decide the time lines for every activity and prepare a micro schedule for the activity
3. Monitor the accomplishment for every activity against the schedule prepared in the previous step. So that we can take care of timely completion of the task. It will also help to improve our ability to estimate the timelines to complete the task.

Task life cycle will be different for different project or different task type

And hence programmer should first understand each step in the life cycle. Estimate the time required for each step and monitor the task completion against the plan.

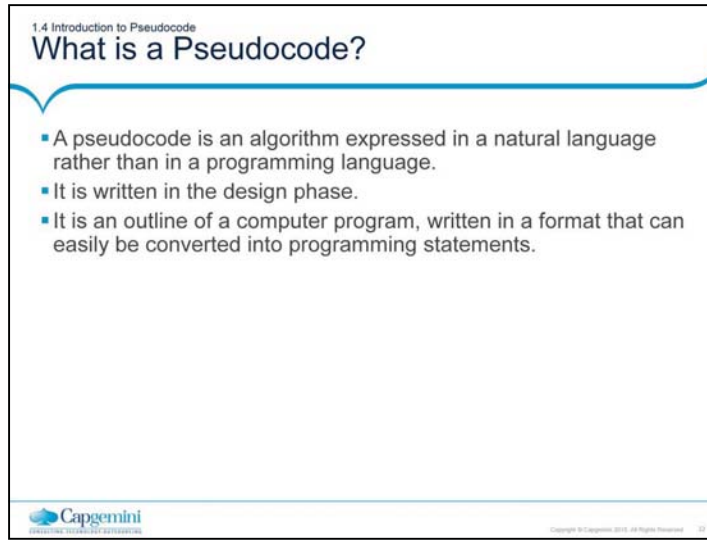


We need to submit above mentioned deliverables.
In defect data we need to log the defect and also mention the closure of it.
Submit the code to configuration management (Check-in in the Source Code Control system). Once the integration test is pass then close the task in PMS (Project Management system). i.e we are ready to work on another task.



Student Syndrome:

- In the above graph. If you observe people are working with less efforts than expected in the beginning and near the dead line people are slogging to complete the work in time.
- This is similar to how a student prepare for the exams for day and night just few days before the exam.
- Avoid student syndromes. While working in project.
- Parkinson's Law:
  - "Work expands to fill (and often exceed) the available time."
- Murphy's Law:
  - "If anything can go wrong, it will!"



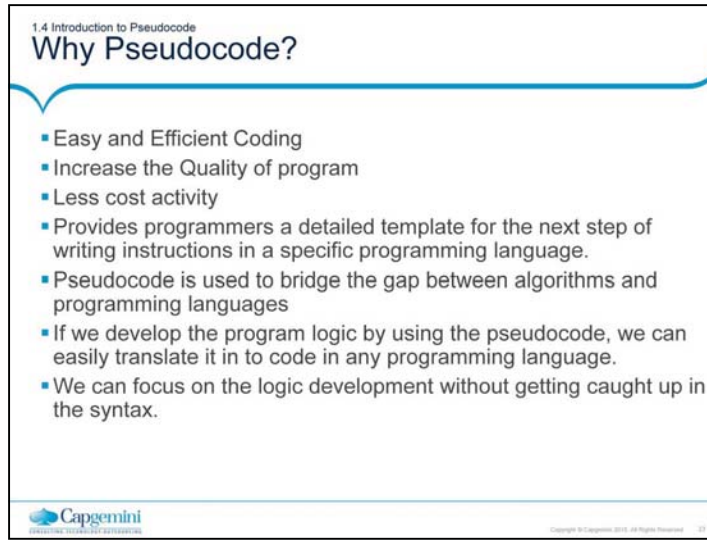
What is Pseudocode?

- Pseudocode is a natural language description of what a computer program must do rather than in a programming language.
- It allows the developer to focus on the logic of the code without being distracted by details of programming language syntax.
- At the same time, the pseudocode needs to be complete. It describe the entire logic of the task so that implementation becomes easier for translating the task line by line into source code.
- The pseudocode describes the logic of the program and acts as a blueprint for the source code to be written by the programmer.

PseudoCode for Finding Whether a number is odd or even:

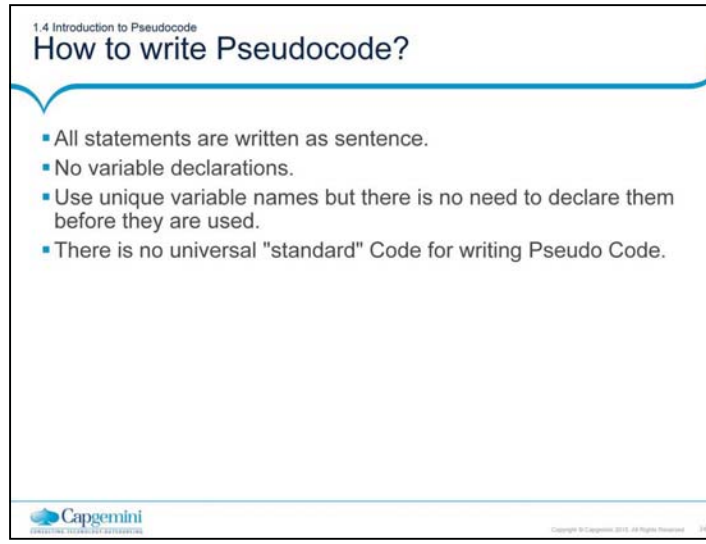
```
BEGIN
        PROMPT "Enter the number" AND STORE IN num
        IF (num MOD 2 == 0) THEN
                DISPLAY "Even Number"
        ELSE
                DISPLAY "Odd Number”
        ENDIF
END
```

PROMPT is a Pseudocode keyword used to take inputs from the keyboard and store in a variable



Why Pseudocode ?

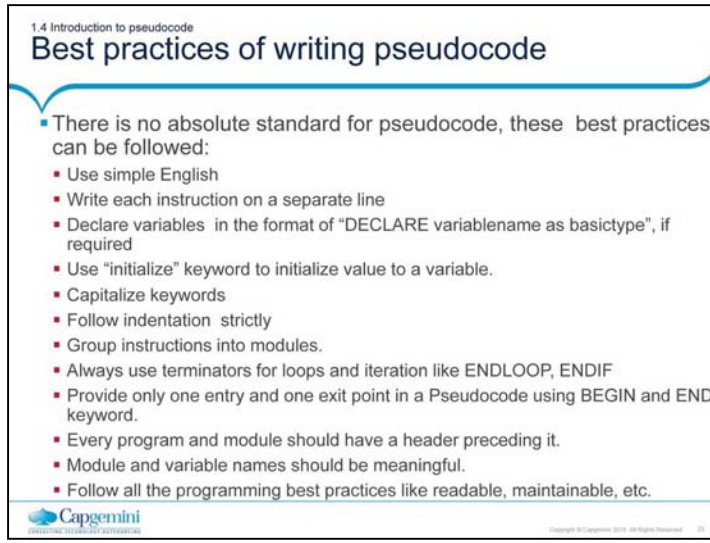
- Easy and Efficient Coding – Easy to solve the task by focusing only on logic of the code with pseudo code rather than any other programming language.
- Increase the Quality of program – Easy way for Analyst to ensure the code matches with design specifications. Once it is matched, then they can easily convert the pseudocode into project specific Language. Thus it helps to ensure requirements are met and that program code meets good software development practice.
- Less Cost activity. Since Catching Logical errors is less tedious than catching them in development process.



Rules to be followed While writing pseudo Code:

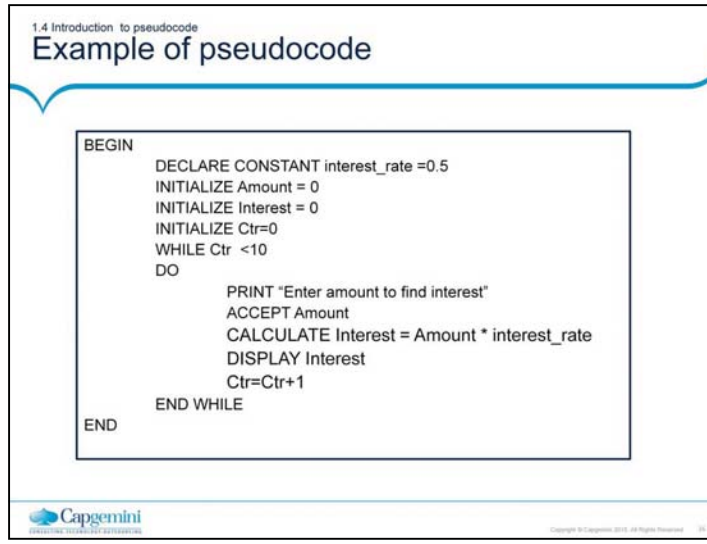
- All statements are written as sentence. Use Words and Phrases which are in line with basic computer operations.
- No variable declarations. Use unique variable names but there is no need to declare them before they are used.
- There is no universal "standard" Code for writing Pseudo Code. But for understandable by others in the project follow the common coding standards specific to the project.
- Some of the Common Coding Notations are

Pseudocode Keyword	Function / Operation
DISPLAY/PRINT	Output to screen
PROMPT/ACCEPT	Display a prompt and store into a variable
EQUALS or =	Assignment operation
READ	Read from data source (File)
WRITE	Write to data source (File)
INITIALIZE/SET	Give data an initial value



Pseudocode Example with best practices:

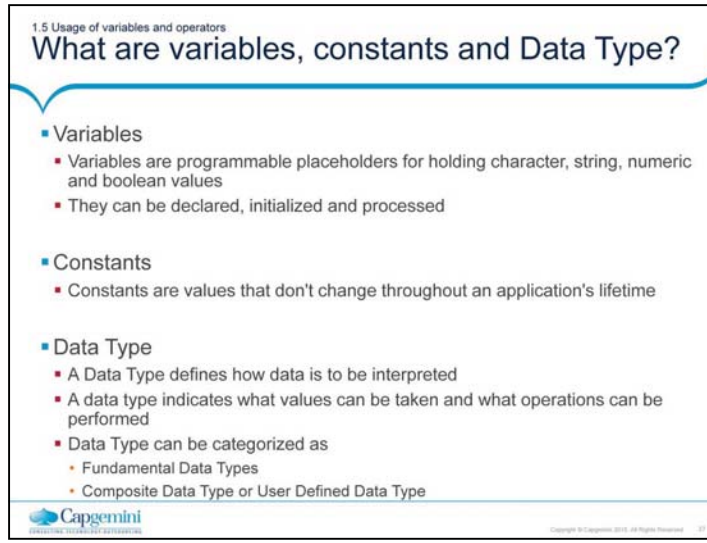
```
BEGIN
        DECLARE num AS INTEGER
         INITIALIZE num TO 0
         PROMPT "Enter the number" AND STORE IN num
         IF(num > 0) THEN
                  DISPLAY "Positive Number"
         ELSE
                  DISPLAY "NegativeNumber"
         ENDIF
END
```



The above pseudocode is used to calculate interest for a given amount based on fixed interest rate. This process will be repeatedly executed for 10 times.

Fixed Interest Rate is 0.5.

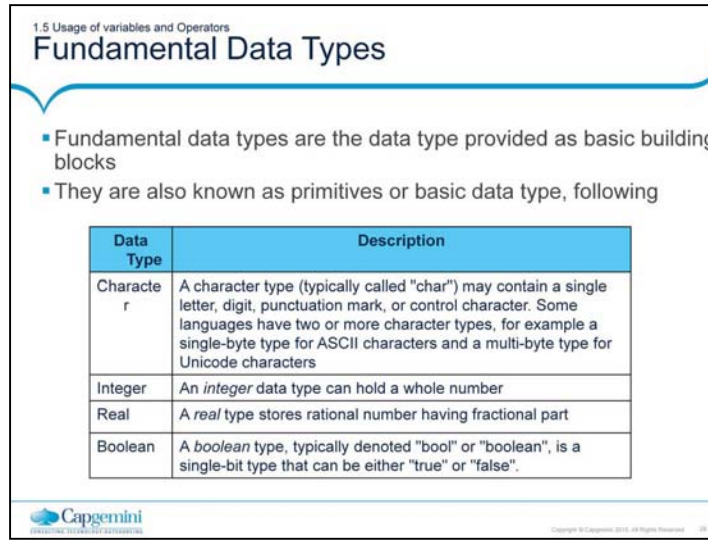
Formulae used to calculate interest is Amount * Fixed Interest Rate.



Variable:

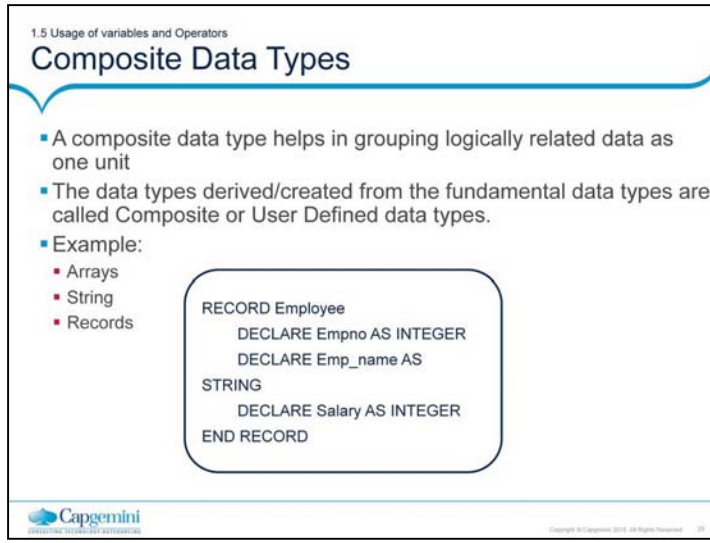
Variable's value can be changed at any point in a program.
Each new value must be of the initial type.
Variable describes data that may change while the program is running

Constants:
Constant describes data that is set before the program is used and will remain the same while the program is running
For example, the mathematical symbols pi and e have well-defined values, which are invariant



Character
The char data type represents single characters, such as ‘m’.
A variable made up of a number of characters is called a string.
Integers
Integers are positive and negative whole numbers
There are different sizes of integers available, including
Short integers and
Long integers
Real
Real data are numbers with a fractional part, for example 8.65 and -0.03
They also include numbers that have no fractional part but are expressed as a whole number with a decimal point, such as 4.0 or -1.0
Boolean
Boolean values are logical values and can be either true or false.

Pointer Data type is integer type
Named Constants are also integers



"Structured data" refers to data that's built up from other types. Use them to clarify relationships between related items

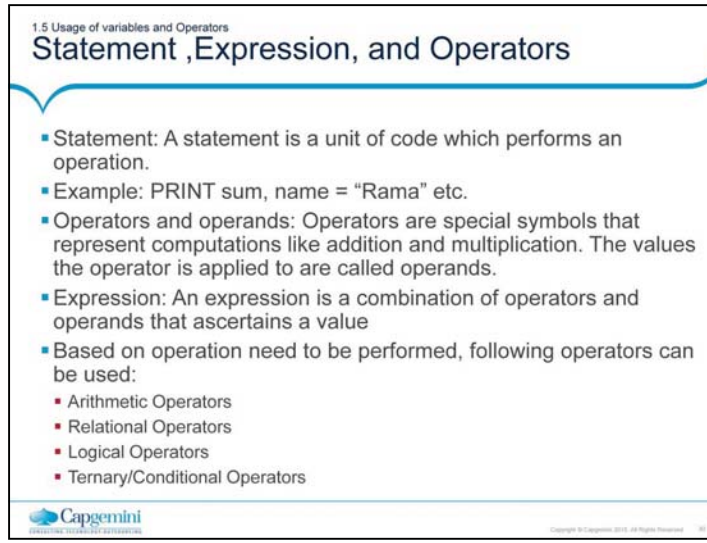
Example :

```
Name  = InputName;
Address =  InputAdress;
Phone = InputPhone;
Title = InputTitle;
Department = InputDepartment
Bonus = InputBonus
```

```
Employee.Name  = InputName;
Employee.Address = InputAddress;
Employee.Phone = InputPhone;
Supervisor.Title = InputTitle;
Supervisor.Department  = InputDepartment;
Supervisor.Bonus = InputBonus;
```

Why to create user defined datatypes?

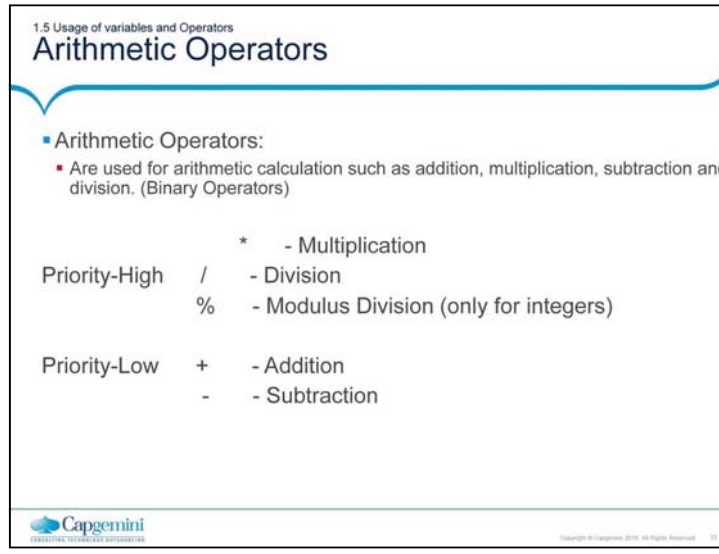
- To increase reliability
  One could specify the range of values that a variable of a User Defined data type can take.
- To make up for language weakness
  If a language does not support a type the user wants, it is possible to create it yourself.
  Example, a student in 'C



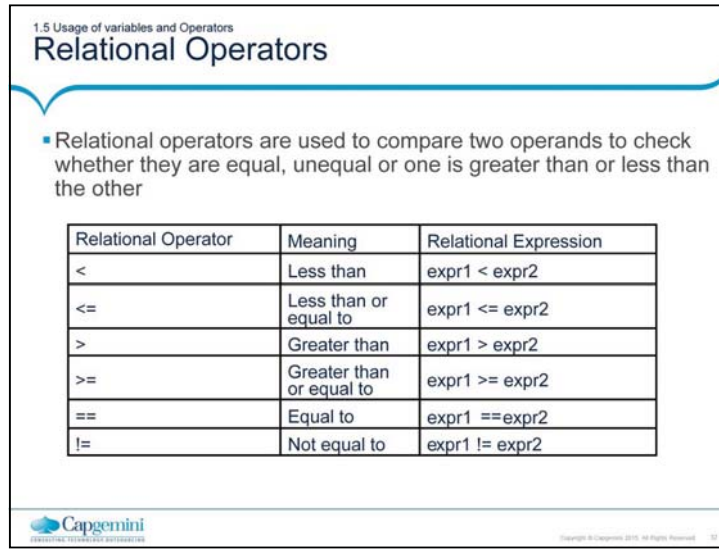
Operators and operands, Expression
An operator is a symbol that represents a specific action that must be performed on data

Ex:In the expression *sum=num1+num2*
sum,num1 and num2 are the *operands* and + is the *operator*

Operators have rules of precedence and associativity that are used to determine how expressions are evaluated.
Expressions:
An Expression is a combination of constants and variables together with the operators.
Constants and variables by themselves are also considered as expressions. An expression that involves only constants is called a **Constant Expression**.
Note: Balanced parentheses can be used in combining constants and variables.
Operators have been classified into categories based on the operation that they perform. Refer the slide for the different categories.



Types of Operators: Arithmetic Operators:
There are five types of arithmetic operators that are used for arithmetic calculations such as, addition, subtraction, multiplication and division.
These five operators are binary operators that is, they require two operands.
Each of these operators work with values of type integers, Real and character.

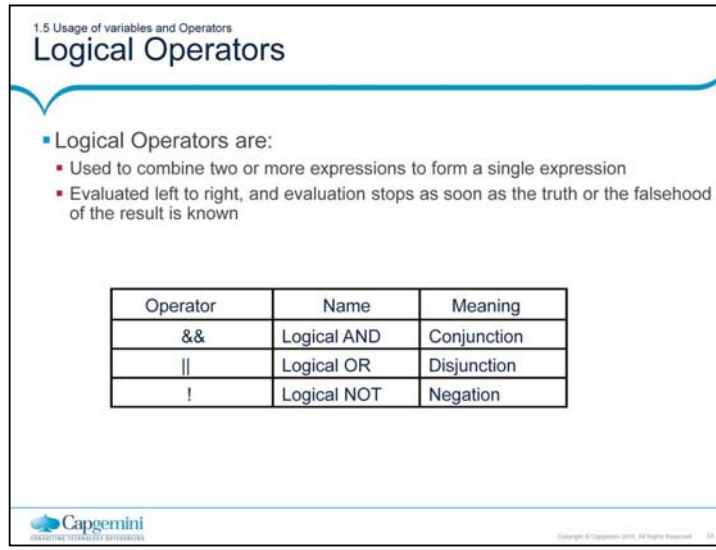


Types of Operators: Relational Operators:

Relational Operators are used to compare two operands to check whether they are equal, unequal or one is less than or greater than the other.

There are six relational operators for comparing the values of two expressions and the expression so formed is called a Relational Expression.

Table provided in the slide shows relational operators and how they can be used to compare expressions expr1 and expr2.



Types of Operators: Logical Operators:
There are three logical operators for combining expressions into logical expressions.
Table in the slide shows available logical operators

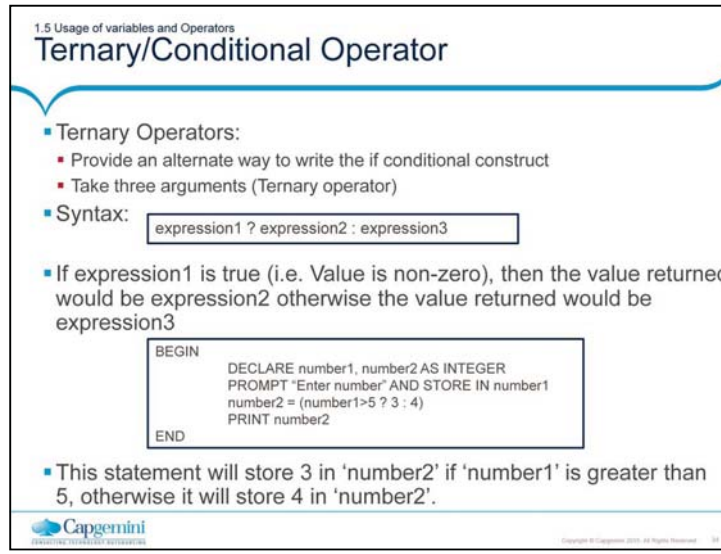
The logical operators **&&** and **||** are binary operators, and ! is a unary operator.
The value of a logical expression is either '1' or '0', depending upon the logical values of the operands. The operands may be of any arithmetic type. The result is always an integer.

Logical AND Operator(&&):
The && operator combines two expressions into a logical expression and has the following operator formation:

expr1 && expr2

An expression of this form is evaluated by first evaluating the left operand. If its value is 0 (false), then right operand is not evaluated and the resulting value is 0 (false).

If the value of left operand is nonzero (true), the right operand gets evaluated. The resulting value is 1 (true) if the right operand has nonzero value (true) and 0 (false) otherwise.



Types of Operators: Ternary Operators:
Simple condition operations can be carried out using the conditional operator (? :). The conditional operator is a ternary operator that is, it takes three arguments. It has following operator formation:

expression1 ? expression2 : expression3

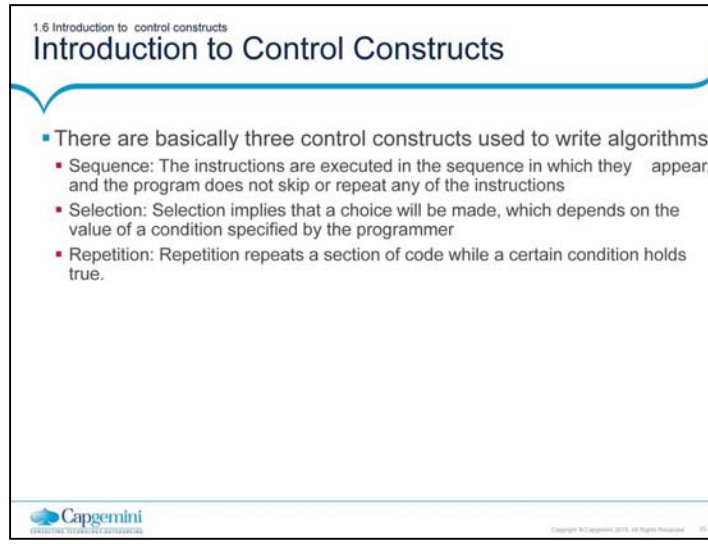
Where ? and : are the two symbols that denote this operator. A conditional expression is evaluated by first evaluating expression1. If the resulting value is true, then expression2 is evaluated and the value of the expression2 becomes the result of the conditional expression. Otherwise, expression3 is evaluated and its value becomes the result.

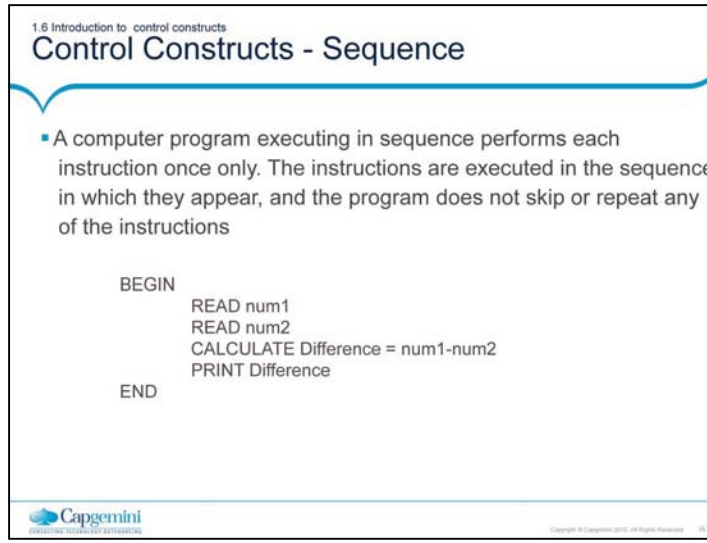
This is used to assign one of the two values to a variable depending upon some condition.

For example:

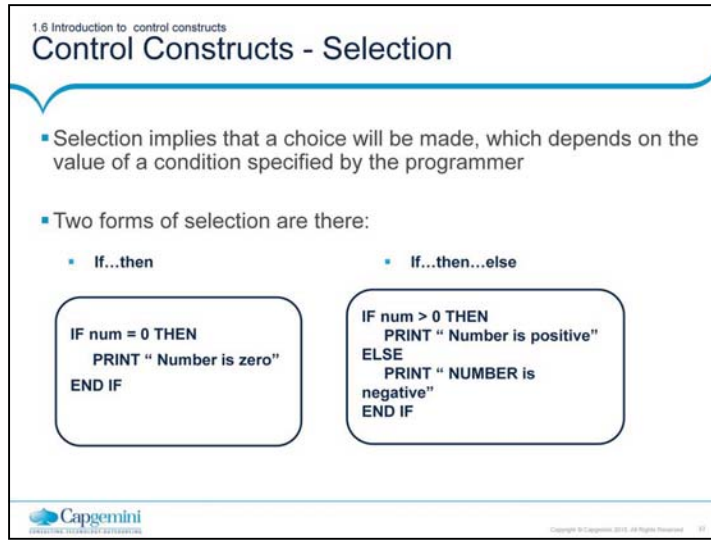
big = num1 > num2 ? num1 : num2 ;

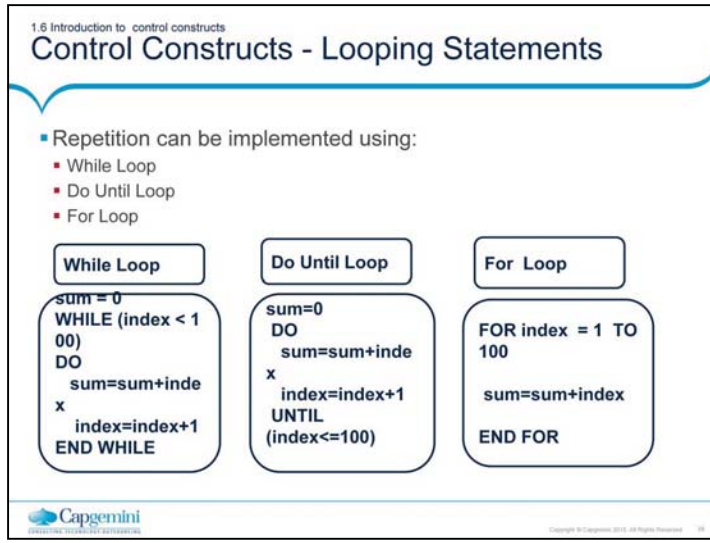
assigns value of num1 to big if num1 is greater than num2, else assigns the value of num2 to big.


1.6 Introduction to control constructs
Introduction to Control Constructs
There are basically three control constructs used to write algorithms:
Sequence: The instructions are executed in the sequence in which they appear, and the program does not skip or repeat any of the instructions
Selection: Selection implies that a choice will be made, which depends on the value of a condition specified by the programmer
Repetition: Repetition repeats a section of code while a certain condition holds true.
Capgemini



Pseudocode given in the slide is used to find the difference between two numbers. In the above pseudocode, each instruction will be executed sequentially.


1.6 Introduction to control constructs
Control Constructs - Selection
Selection implies that a choice will be made, which depends on the value of a condition specified by the programmer
Two forms of selection are there:
If...then
IF num = 0 THEN
PRINT " Number is zero"
END IF
If...then...else
IF num > 0 THEN
PRINT " Number is positive"
ELSE
PRINT " NUMBER is negative"
END IF
Capgemini



Control Constructs – Looping Statements

Do Until Loop: Set of statements inside the block is executed and then the condition is checked. Executed till the condition is true. Block will be executed at least once irrespective of the condition

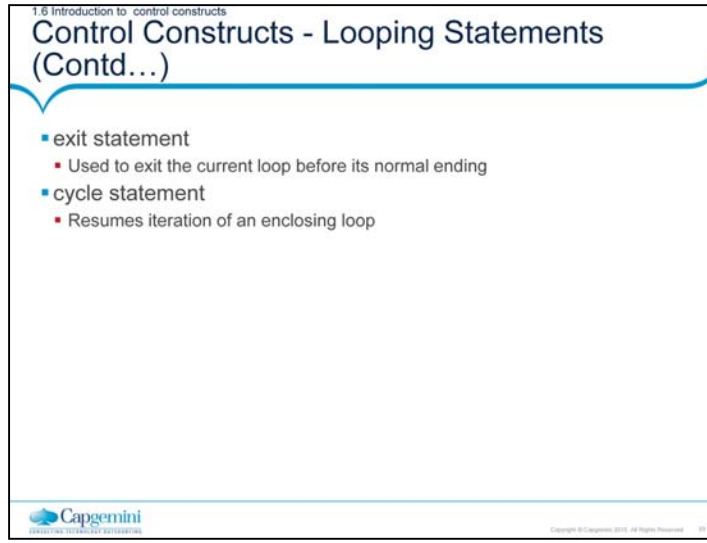
Example:

```
seats_allocated = 0
 DO
  GET booking
  PRINT ticket
  ADD 1 to seats_allocated
UNTIL (seats_allocated < 60)
```

For Loop:Consists of set of statements that are executed for a fixed number of iterations Must specify a starting value, ending value and incrementing value

Example:

```
FOR seats_allocated = 1 to 25
   Get booking
   PRINT ticket
END FOR
```

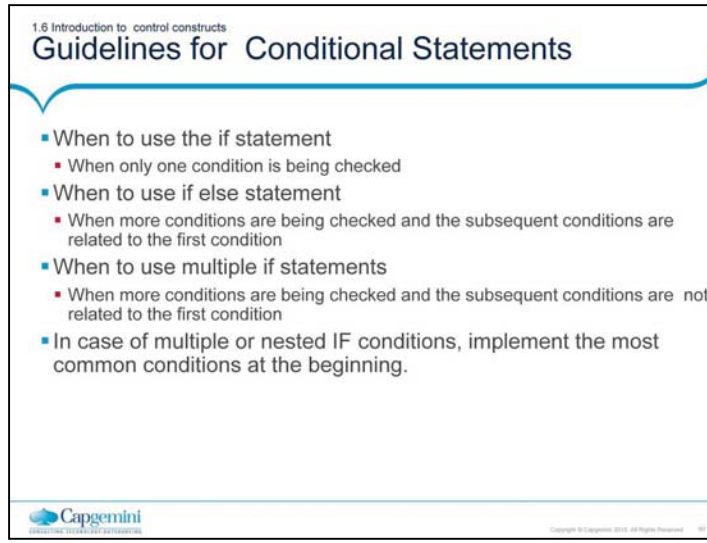


Control Constructs – Looping Statements (contd..)

Example: Cycle and exit statement(check no is odd or even until user wants to stop)

```
BEGIN
   WHILE(TRUE)
      PRINT "Enter a NUMBER"
      ACCEPT num
      IF (REMAINDER of num/2 = 0) THEN
            PRINT "Number is EVEN"
      ELSE
            PRINT "Number is ODD“
      END IF
       PRINT "Enter u'r Choice , Continue with another number?[Y/N]"
       ACCEPT choice
        IF(choice='Y') THEN
                  cycle
        ELSE
                  exit
    END WHILE
END
```

In above example cycle statement will transfer the control to the beginning of the while loop. Exit statement will transfer the control out of while loop.



Consider Pseudocode has to be written for accepting a number from user and need to identify whether the given number is falling under any of the below mentioned category

- zero
- negative
- Positive

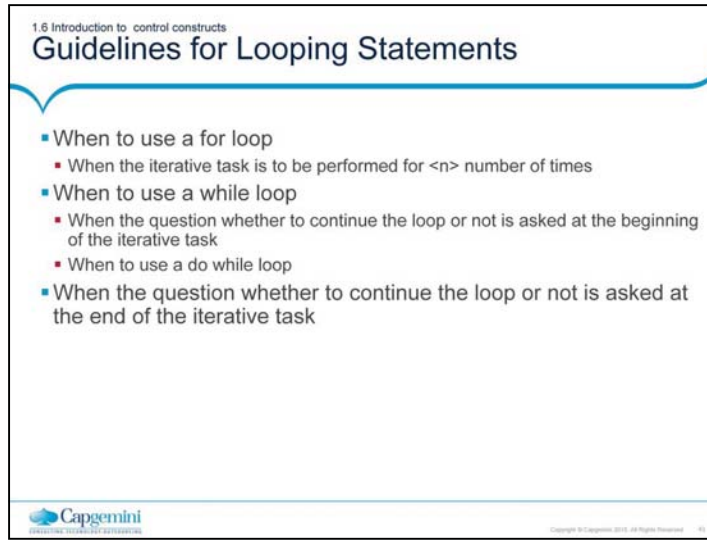
Think which condition statement is more suitable for this?

Answer: As the given number has to be compared against minimum 2 conditions, IF-ELSEIF-ELSE is more suitable

Pseudocode:

```
BEGIN
        DECLARE num AS INTEGER
        PROMPT “Enter a number” AND STORE IN num
        IF ( num== 0 )THEN
                PRINT "The value you entered was zero.“
        ELSE IF ( num< 0 ) THEN
                PRINT  "The value is negative.“
        ELSE
                PRINT  "The value is positive.“
        END IF
END
```

I



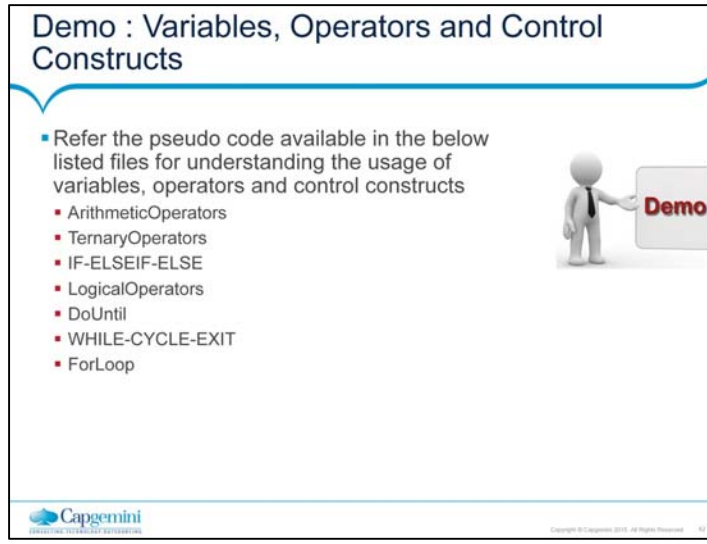
Consider Pseudocode has to be written for finding sum of n numbers by accepting "n" value from the user. For an Example, if given "n" value is 5, then sum of value is 15(1+2+3+4+5).

Think which looping statement is more suitable for this scenario?

Answer: As the given number has to be used as upper limit for process to be executed repeatedly, use for loop to describe lower limit and upper limit.

Pseudocode:

```
BEGIN
        DECLARE num, count, sum AS INTEGER
        PROMPT "Enter the value of n" AND STORE IN num
        FOR COUNT = 1 TO num
                sum+=count;
        END FOR
        PRINT "Sum is " + sum
END
```



Faculty will be sharing the below mentioned files. Refer the same to understand the usage of variables, operators and control constructs

ArithmeticOperators.txt

TernaryOperators.txt

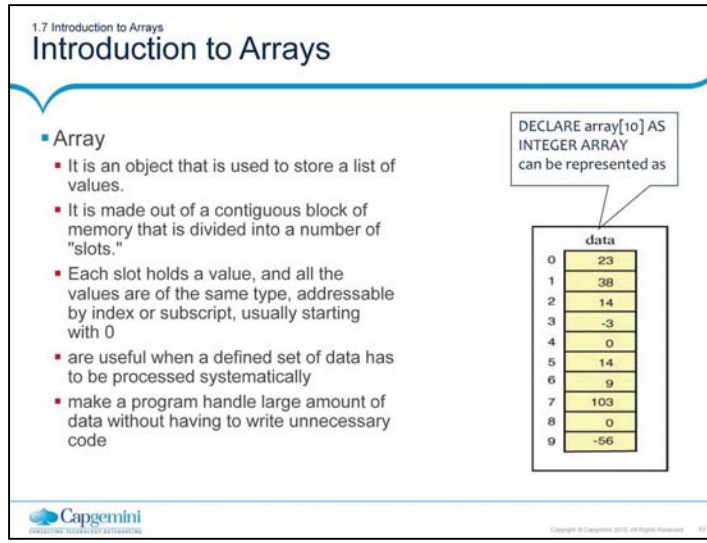
IF-ELSEIF-ELSE.txt

LogicalOperators.txt

DoUntil.txt

WHILE-CYCLE-EXIT.txt

ForLoop.txt



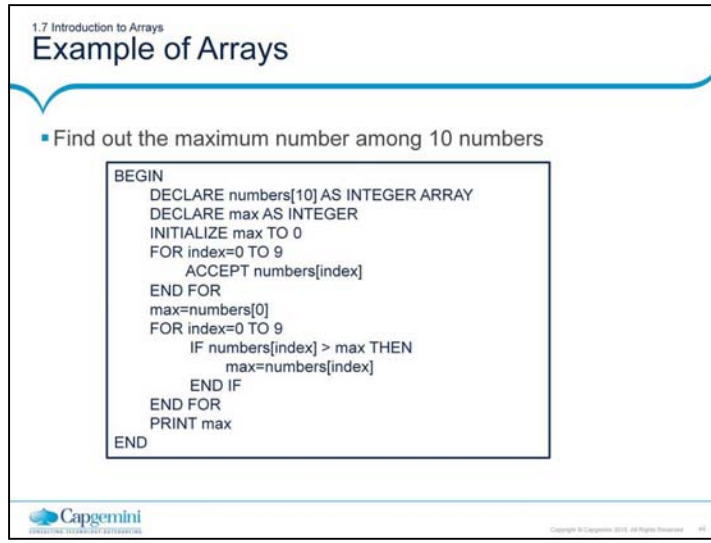
When to go far an Array?

- When there is a finite set of logically related information that needs similar processing
- For example calculating the grade of all students of a particular class
- All data's are similar and of one data type

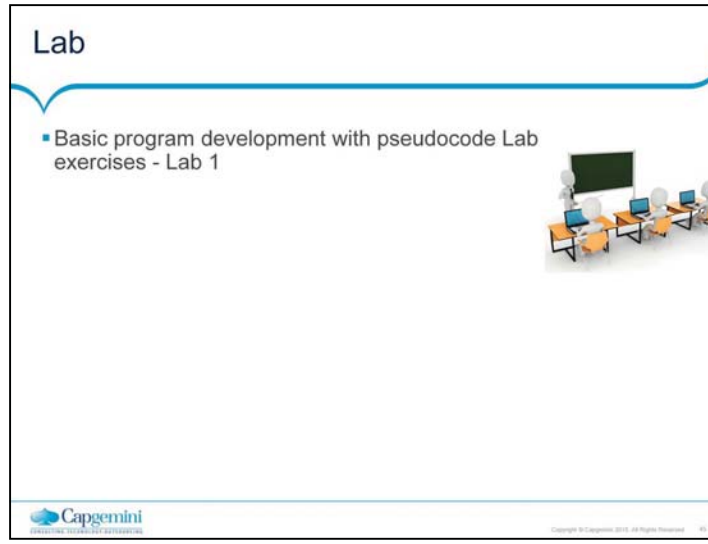
Operations on an array:

The frequent operations which we perform on an array are

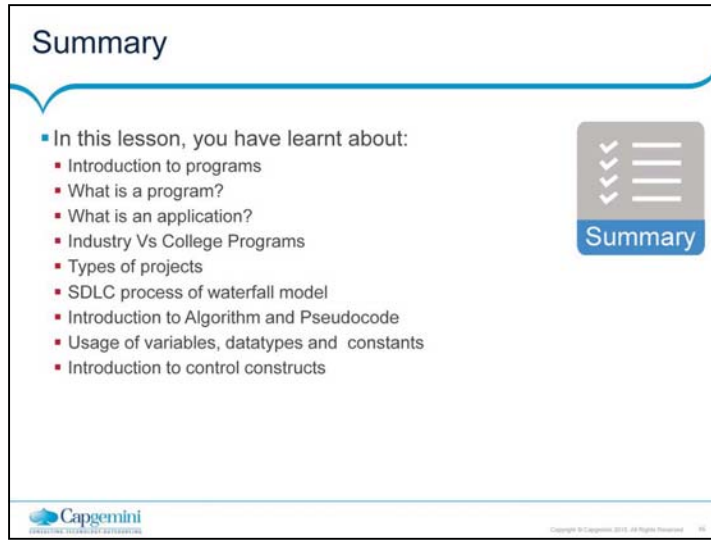
- Searching: searching the array for particular element
- Sorting : arranging data in particular order (ascending/descending)



The above pseudocode logic is used to accept 10 numbers from user and find the largest number among 10 numbers. For storing the accepted 10 numbers of same type, array is used in this pseudocode.



Write pseudocode for all the assignments mentioned in Lab 1 for practicing on the usage of variables, operators, control constructs and arrays.


Summary
In this lesson, you have learnt about:
Introduction to programs
What is a program?
What is an application?
Industry Vs College Programs
Types of projects
SDLC process of waterfall model
Introduction to Algorithm and Pseudocode
Usage of variables, datatypes and constants
Introduction to control constructs
Summary
Capgemini

Review Question

- Question 1: What are different types of testing techniques?
  - A. Self testing
  - B. Black box testing
  - C. Red box testing
  - D. None of the above

- Question 2: A task which gets done in the specified time with desired quality defines ------
  - A. Maintainable
  - B. Efficient
  - C. Robust
  - D. Readable

Capgemini

Copyright © Capgemini 2015. All Rights Reserved

Review Question

- Question 3: There is no absolute standard for pseudocode
  - A. True
  - B. False
- Question 4: Identify guidelines for using variables
  - A. Assigning values to variables just before values are used
  - B. Prefer local variables over global ones
  - C. Initialize variables used in looping constructs at the top of the code block
  - D. Maximize the lifetime of local variables

Capgemini

Copyright © Capgemini 2015. All Rights Reserved

Review Question

- Question 5: How does the following code breach best practice guidelines

```
myarray[10]= new Array{1,1,2,3,5,8,13,21,34,55};
index=0;
while(index<10)
index=index+1;
//dosomething
```

- A. The array should have a clearer name
- B. The index is never incremented
- C. The index Is zero based so the loop passes beyond the last element

Capgemini

Copyright © Capgemini 2015. All Rights Reserved

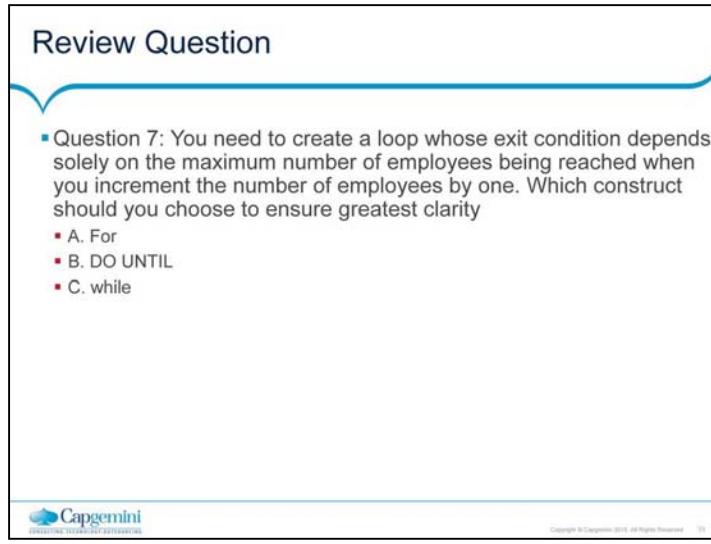
Review Question

- Question 6: Match the looping structure to their definitions

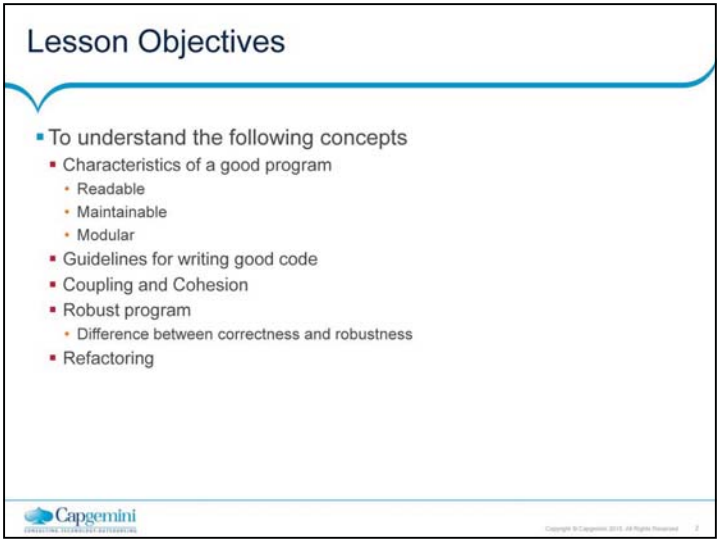
1. For	A. Condition controlled for executing choice once
2. IF	B. Condition controlled for executing choice multiple times
3. While	C. Count controlled

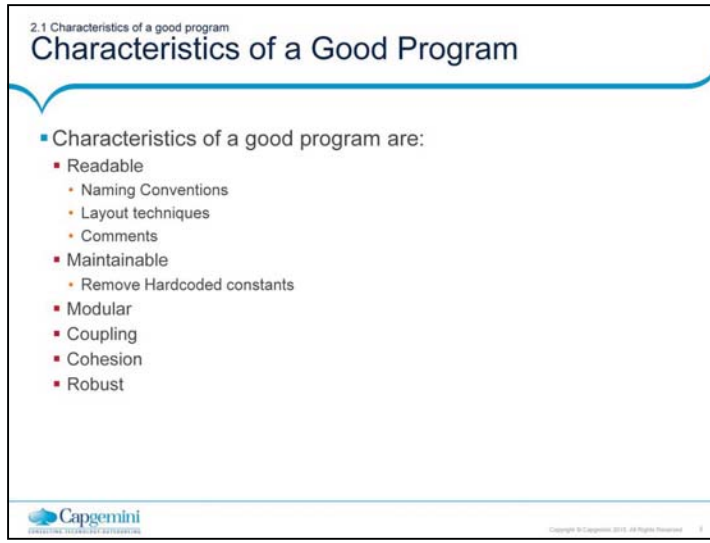
Capgemini

Copyright © Capgemini 2015. All Rights Reserved


Review Question
Question 7: You need to create a loop whose exit condition depends solely on the maximum number of employees being reached when you increment the number of employees by one. Which construct should you choose to ensure greatest clarity
A. For
B. DO UNTIL
C. while
Capgemini


Programming Foundation
With Pseudocode
Lesson 2: Good Programming
Practices


Lesson Objectives
To understand the following concepts
Characteristics of a good program
Readable
Maintainable
Modular
Guidelines for writing good code
Coupling and Cohesion
Robust program
Difference between correctness and robustness
Refactoring
Capgemini



If any program has following features then it is called as good program
1. Readable
   If the names of variables are meaningful and the code is easy to understand

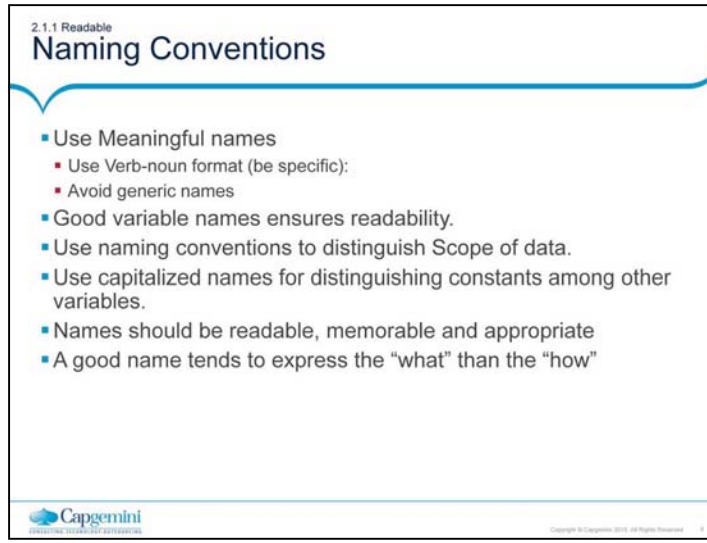
2. Maintainable
   If the program is easy to understand and If it is easy to modify then the program is called as maintainable

3. Modular
   A small unit of code for a single purpose

4. Coupling
   Coupling or Dependency is the degree to which each program module relies on each other.

5. Cohesion
   A cohesion is a measure of how the activities within a single module are related to one another.

6. Robustness
A program is said to be robust when it is fault tolerant. You should test your program extensively with positive and negative test conditions to prove its robustness



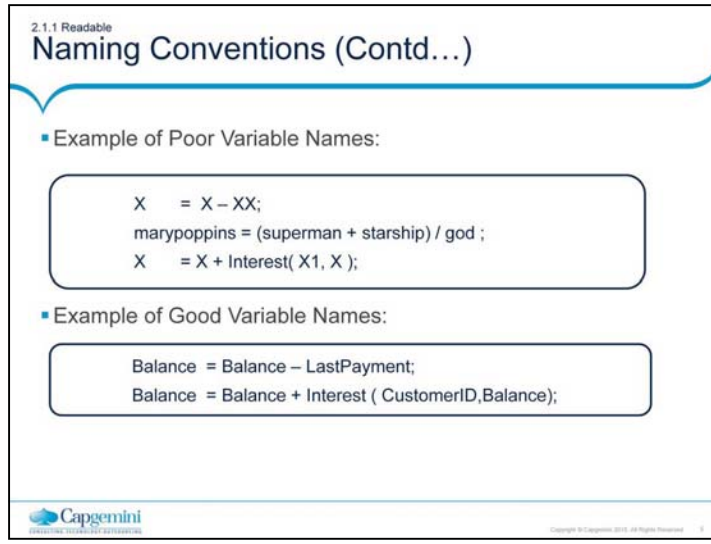
Naming Conventions

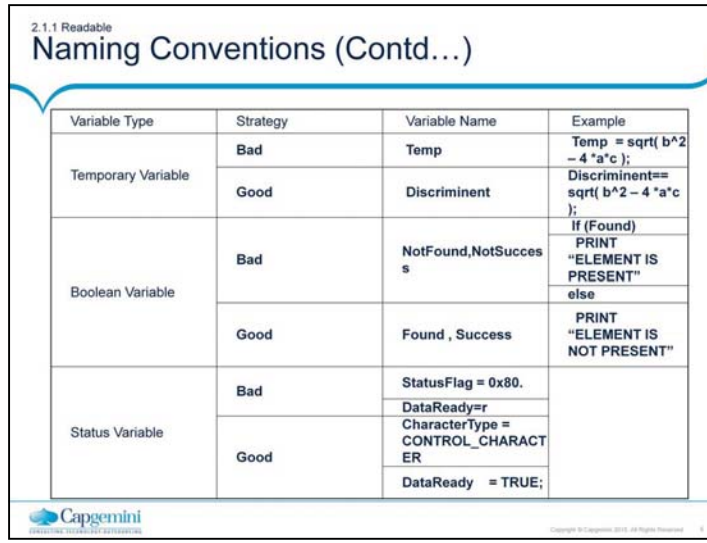
- Meaningful Names:
  - The name fully and accurately describes what the variable represents
  - The name should refer to the real-world problem rather than to the programming-language solution
  - Use Verb-noun format (be specific):

    For example: Read-Employee-Record, Calculate-Deductions, Print-Pay-slip
  - Avoid generic names:

    For example: Process-inputs, Handle-calculations
- Good variable names are a key element of program readability.
- Use naming conventions to distinguish Scope of data like local/global.

  For an Example, MAX_USERS_G variable scope is global
- Use capitalized names for distinguishing among type names, named constants, enumerated types, and variables.


2.1.1 Readable
Naming Conventions (Contd…)
Example of Poor Variable Names:
X = X – XX;
marypoppins = (superman + starship) / god ;
X = X + Interest(X1, X);
Example of Good Variable Names:
Balance = Balance – LastPayment;
Balance = Balance + Interest (CustomerID,Balance);
Capgemini



Naming Status Variable: Think of a better name than "flag" for status variables.

```
if ( DataReady ) …..
  if ( CharacterType & PRINTABLE_CHAR) …
  if ( ReportType == AnnualRpt ) …
  DataReady    = TRUE;
  CharacterType = CONTROL_CHARACTER ;
  ReportType   = AnnualRpt;
  RecalcNeeded = FALSE;
```

CharacterType = CONTROL_CHARACTER is more meaningful than StatusFlag = 0x80.

Naming Temporary Variable: Use meaningful, descriptive names for Temporary variables. Don't use Temp, x or some other vague name.

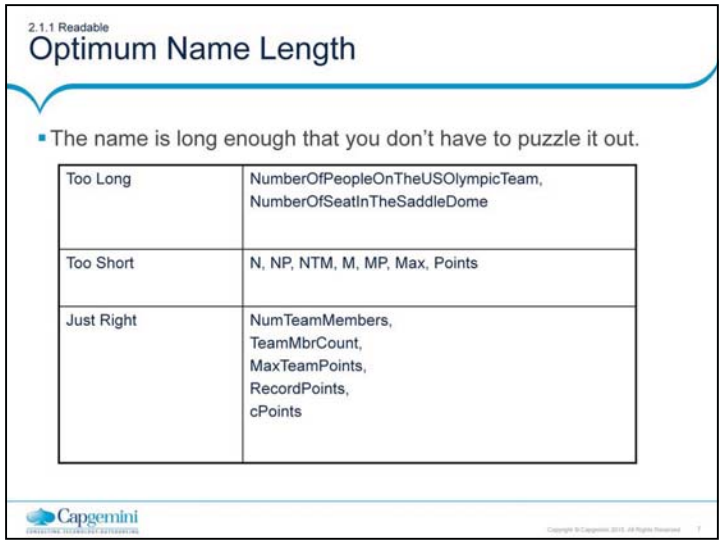
Bad Temporary variable Name

```
Temp  = sqrt( b^2 – 4 *a*c );
```

A Better approach is

```
Discriminant  = sqrt( b^2 – 4 *a*c );
```

Never use a numeric suffix to differentiate variables



Naming Boolean Variables: Use names that imply true or false like done, error, found, success as boolean variables

Avoid using negative names like NotFound, NotDone and NotSuccessful

"If not notFound" is difficult to read

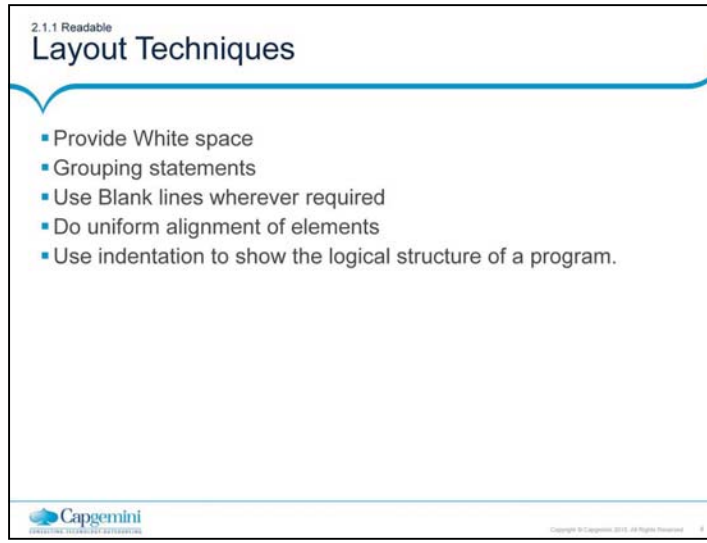
Better way is to use "if found" instead

Give meaningful names for Loop index.

```
i = 0
ACCEPT Emp, Basic
G = B * 1.8 + 1700
PF = 0.12*B
T = ((G*12 - 150000)*0.3 + 19000)/12
N = G - PF - T – 200
PRINT Emp, B, G, PF, T, N
i=i +1
IF i==10  THEN Stop
Continue
```

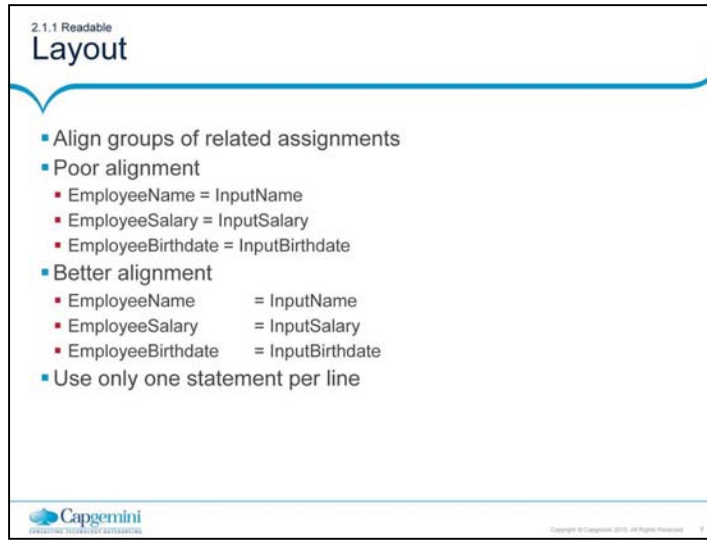
In this example variable 'i' is used to count how many pay slips have been generated.

'*Rec_Count*' can be used instead of 'i'



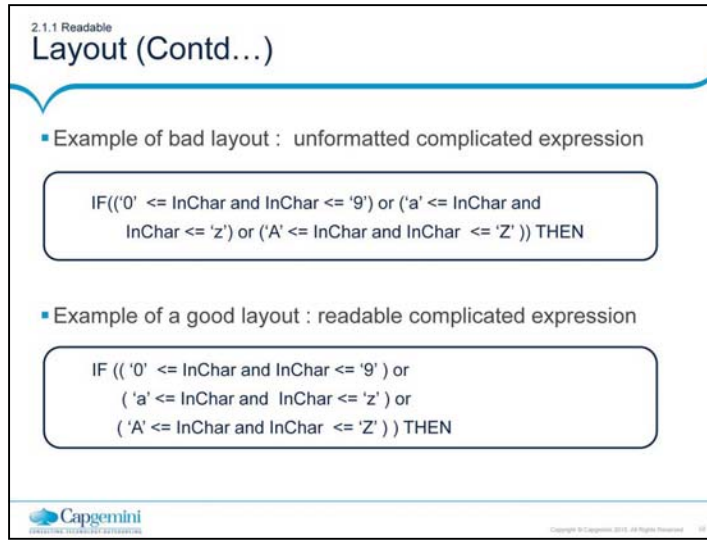
Layout Techniques :

- Use white space to enhance readability.
    - Use vertical and horizontal whitespace generously.
    - Indentation and spacing should reflect the block structure of the code.
    - A long string of conditional operators should be split onto separate lines
- Grouping: ensure related statements are grouped together.
- Blank lines: separate unrelated statements from each other. Use blank lines to divide groups of related statements into paragraphs, to separate routines from one another, and to highlight comments.
- Alignment: align elements that belong together.
- Indentation: use indentation to show the logical structure of a program



Layout :

- Use "=" in a uniform positions in a group of statements.
- Use only one statement per line
- Within a block, Provide tabspace to start with a statement.



Layout Techniques:

- Statement belongs to if/loop should be started after a tab space
- A long string of conditional operators should be split onto separate lines. For example the below if statement

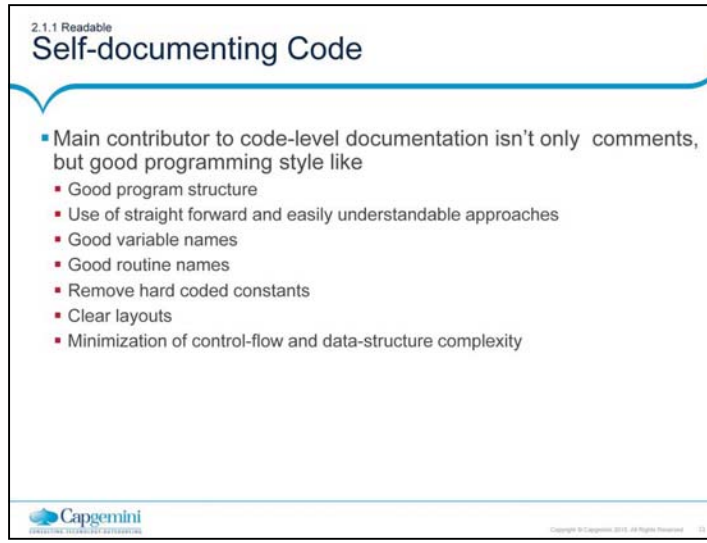
```
if (foo->next==NULL && number < limit && limit
        <=SIZE    && node_active(this_input)) {...
```

might be written better as:

```
if (foo->next == NULL
    && number < limit && limit <= SIZE
   && node_active(this_input))       {        ...
```

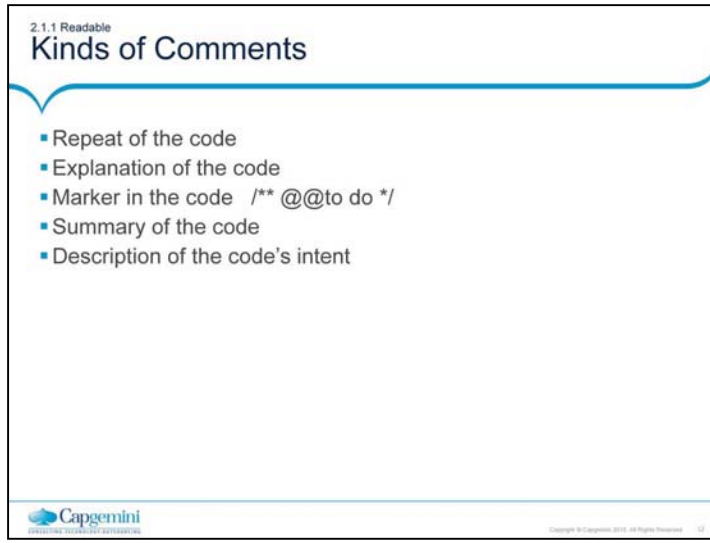
Better version of using For LOOP: Consider for loop have 3 sections

```
FOR (curr = *varp, trail = varp;
     curr != NULL;
     trail = &(curr->next), curr = curr->next )
{
```



Self documenting code:

- Good Program Structure: Follow program structure like modularity
- Use of straight forward and easily understandable approaches.
- Good variable names : Meaningful variable names
- Good routine names : Meaningful routine names
- Remove hard coded constants: Use of named constants instead of literals
- Clear Layout : Follow all the layout techniques



Kinds of comments

- Repeat of the code:

A repetitious comment restates what the code does in different words. It merely gives the reader of the code more to read without providing additional information

- Explanatory Comments:

Explanatory comments are typically used to explain complicated, tricky or sensitive pieces of code. If the code is so complicated that it needs to be explained, its nearly always better to improve the code than it is to add comments. Make the code itself cleared and then use summary comments

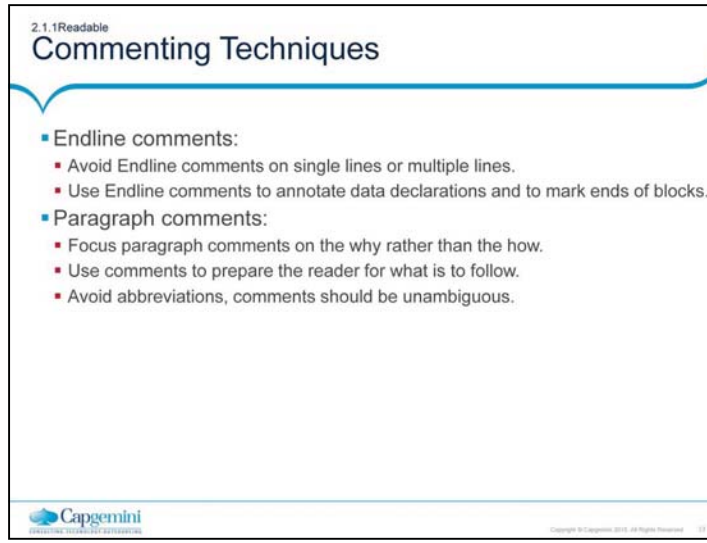
- Marker in the code /** @@todo*/ :

todo is a form of comment used to convey that the code is yet to be completed by replacing todo comment with the functionality logic. For an example, /**@@todo*/

- Summary of the code : Use comment to provide description of the program like header block comment. For an example,

//Program Description:

- Description of the code's intent : Use this comment, to describe the layout used



Commenting Techniques

Commenting is amenable to several different techniques depending on the level to which the comments apply : program, file, routine, paragraph, or individual line

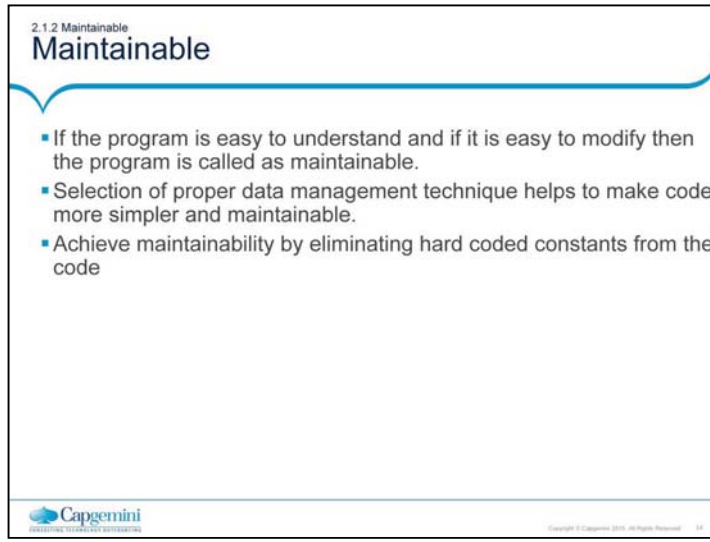
- Endline Comments
  - Endline comments are comments that appear at the ends of lines of code
  - Use endline comments to annotate data declarations like

    Declare index as integer and store 0. //upper index of an array

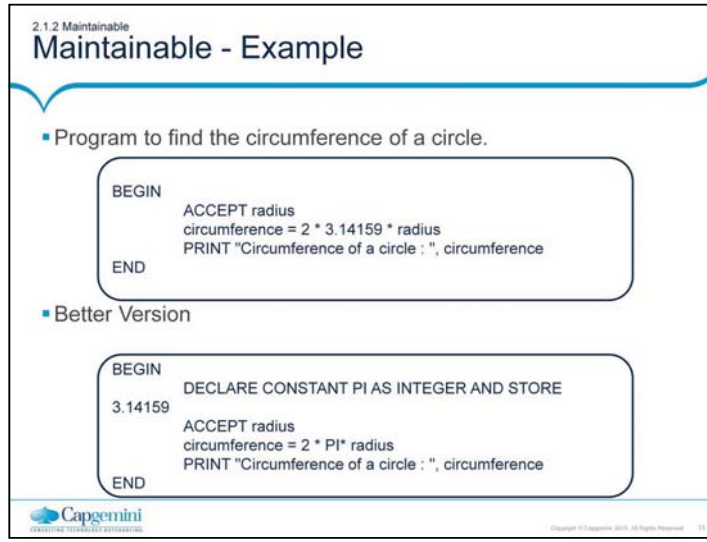
  - Use endline comments to mark ends of blocks

- Paragraph Comments
  - Most comments in a well documented program are one sentence or two sentence comments that describe paragraphs of code
  - Use to describe the purpose of the block of code
  - For an example,

```
/*******************************
*          Search for an employee
* *******************************/
```

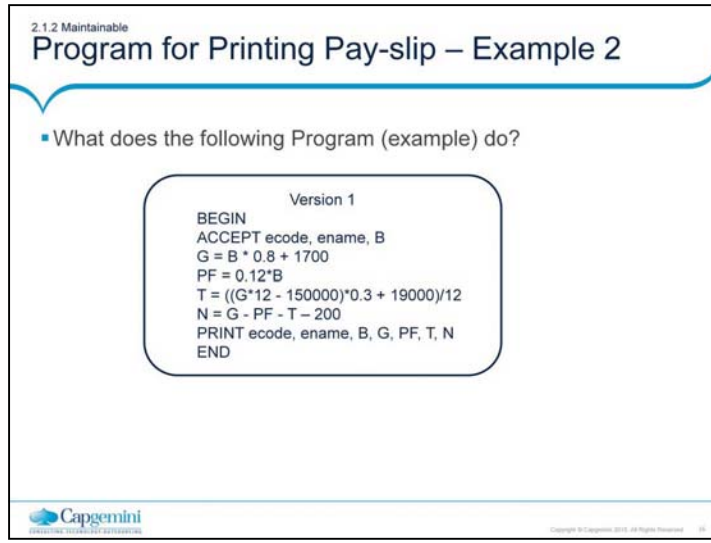


If the program is easy to understand and if it is easy to modify then the program is called as maintainable. Selection of proper data management technique helps to make code more simpler and maintainable. Achieve maintainability by eliminating hard coded constants from the code.



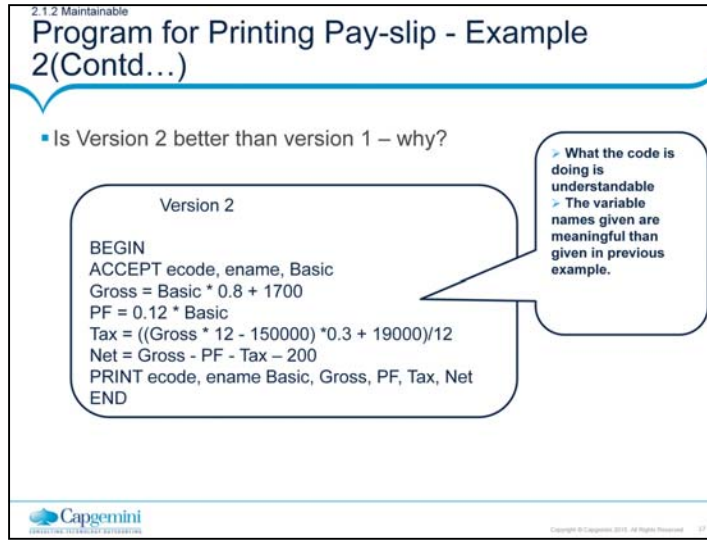
Example:

The given program is used to find the circumference of a circle based on radius. In the given code, hardcoded constant exists which is eliminated by using hard coded constant.



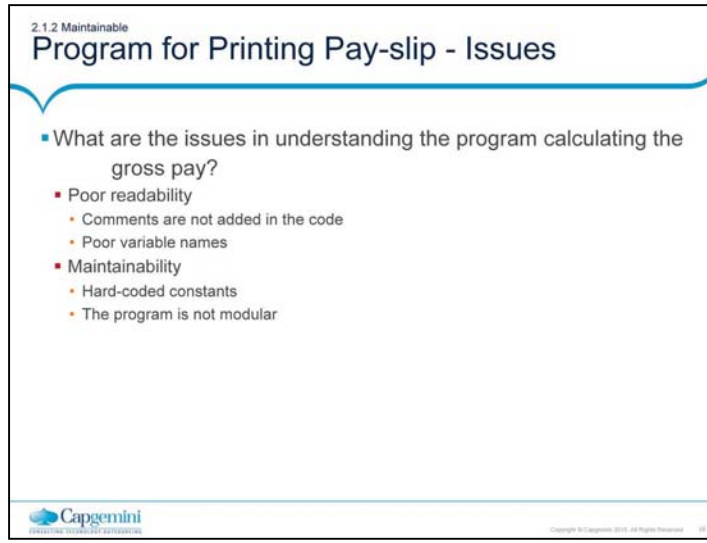
What are problems in the code above:

Variable names are not meaningful. Understanding meaning of variables G, B, T, N etc is difficult. Hence the code is not easily understandable
We don't understand what the given code is doing?



See the above example what it is doing?
It reads employee code, employee name and basic salary from keyboard.
Calculates Gross pay, Provident Fund (PF), Tax and Net pay.
Prints the pay slip

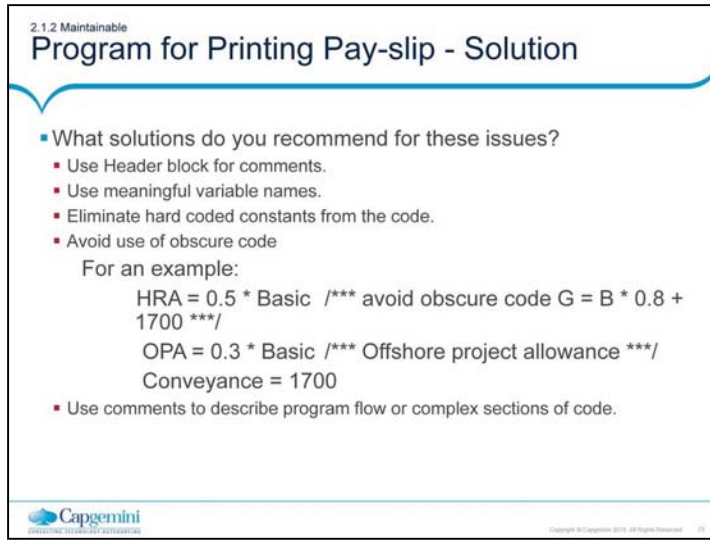
It is understandable because the variable names given are meaningful than given in previous example.
It shows that if you give meaningful variable names then it helps you to understand program better.



Programming standards that are to be followed are given below:

- Use meaningful variable names. It helps to easily understand the program. Avoid using hard-coded constants in the program. Instead, declare them as constants at the beginning of the program, and use these constant names through the program. If value of constant changes later, the value can be modified for the declared constant at one place. The effect will be visible everywhere this constant name is used.
  For example: tax rate

- Include comments at the header and module level
- Don't use literal values in the program instead create constant variable to store fixed literal value for making program to be more maintainable.
- Ensure that you have created more modular program.

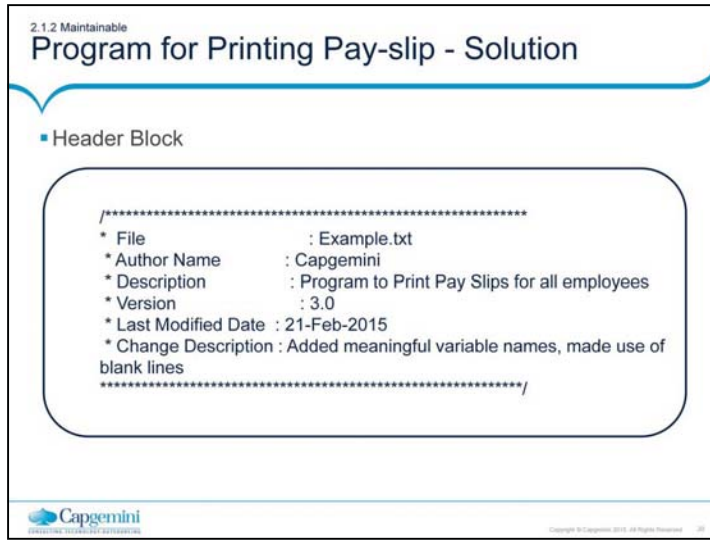


If we use above solution to update the code then the code will become more readable, maintainable.

By reading the code, G = B * 0.8 + 1700, we do not understand what is meant by 1700, nor what is meant by 1.8 in the gross calculation.

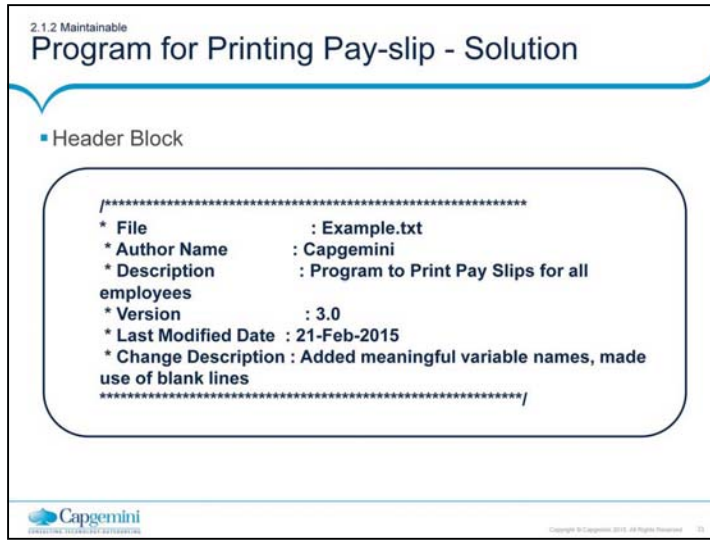
However, the given code explains that Conveyance is 1700. HRA is 50% of Basic. Offshore Project allowance is 30% of Basic. So do not try to club these equations. Otherwise, maintenance of the code will be difficult, and understandability of code will reduce, as well.

Hence avoid such obscure code. Keep it simplified.



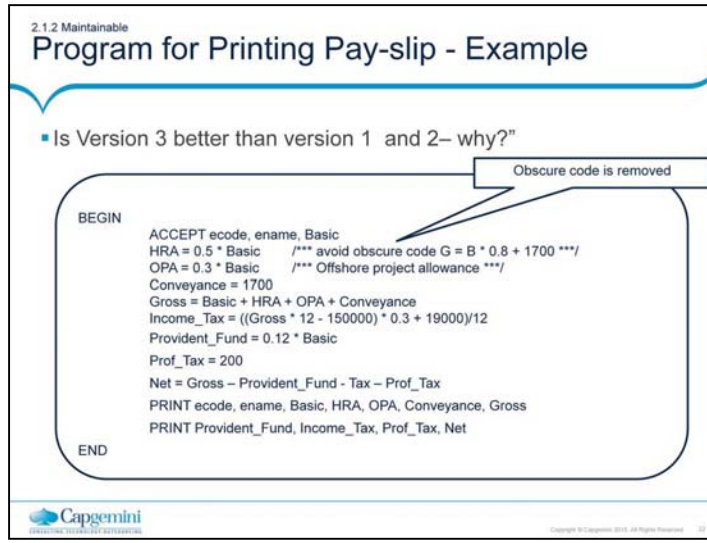
See the given header block of comments
It includes:

- File : Give the name of the file.
- Author name : Provide the author name who involved in the development of program
- Description : Detailed description of the program
- Version : Version of the program
- Last modified date : Date on which the program is last modified
- Change Description: History of changes happened in the program



See the given header block of comments
It includes:

- File : Give the name of the file.
- Author name : Provide the author name who involved in the development of program
- Description : Detailed description of the program
- Version : Version of the program
- Last modified date : Date on which the program is last modified
- Change Description: History of changes happened in the program

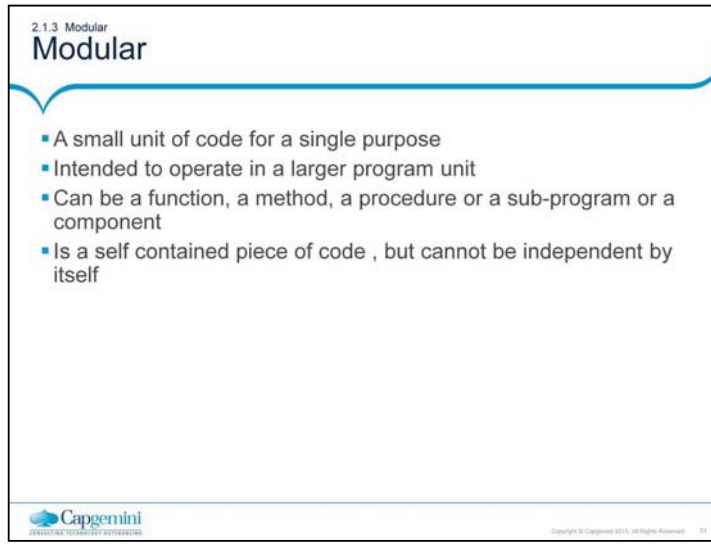


Why version 3 is better than version 1 and 2
1. Hard coded constants are given proper names. E.g – HRA, OPA, Conveyance.
2. The given code is easy to maintain because if conveyance amount , percentage for HRA or OPA changes then we can easily understand where to make the change in the code, which was difficult in version 1 and 2
3. Code is readable as naming conventions are followed and layout is applied

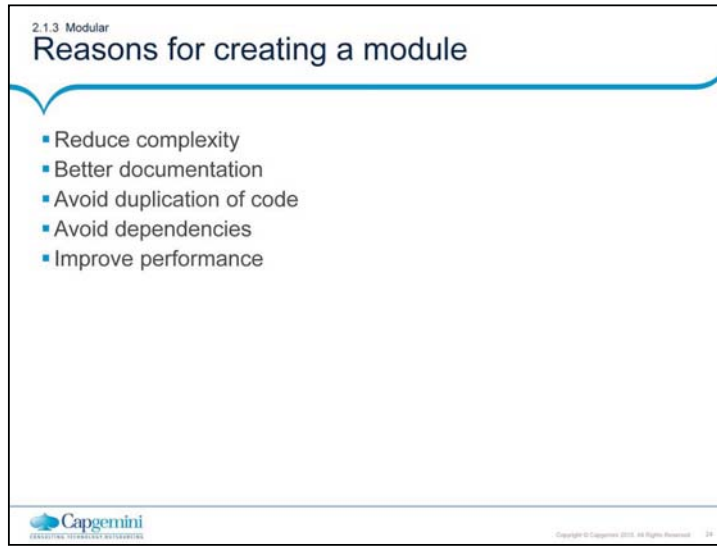
Following improvements are required in the code:
a) If your code includes any complicated calculations It is necessary to document it in the code. In the above example the calculation of income tax includes many operations. What is meaning of it need to be documented in the code.

Steps involved in income tax calculations
1. Calculate annual salary : Gross * 12
2. Calculate taxable amount: 1,50,000 will be subtracted from annual salary
3. Calculate annual tax: Annual Tax = 30% of taxable amount + 19000
4. Calculate monthly tax Monthly tax = Annual tax/12

b) The given code is not modular. It doesn't have any modular structure.
If the code is huge. It is performing various functions then it is better to create separate module for each function which increases reusability of the program.
If any of these modules can be reused in some other application Programmer's efforts and time will be saved.
A good example is login module. Almost every application requires login screen which authenticate users. Such modules can be written once and used in multiple applications.

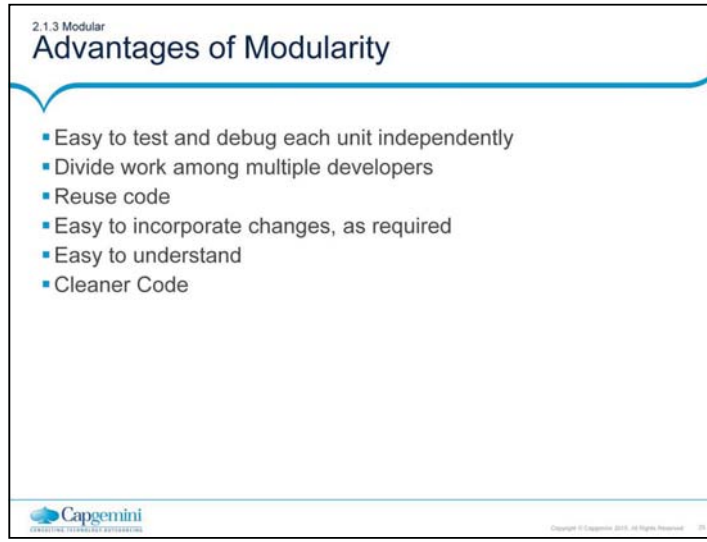


Modular : A small unit of code for a single purpose intended to operate in a larger program unit . It can be a function, a method, a procedure or a sub-program or a component. Module is a self contained piece of code , but cannot be independent by itself



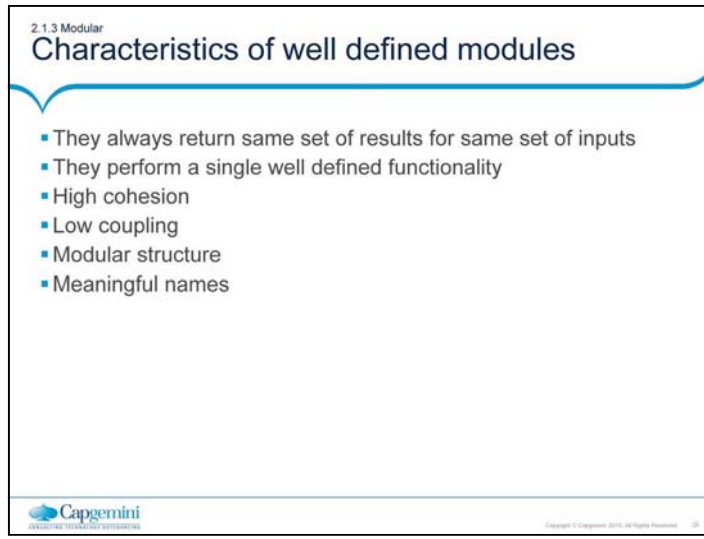
Reasons for creating a module

- Reduce complexity : By using the abstracting power of modules, complex code can be made to appear simpler and easy to understand
- Better documentation : By putting a set code into a well defined module, makes the code self –explanatory
- Avoid duplication of code
- Avoid dependencies : Sections of code that depend on each other makes changes difficult to incorporate
- Improve performance : Easy to test and debug units of code, than a long one



Advantages of Modularity

- Easy to test and debug each unit independently so that defects will be identified at the earlier stage.
- Divide work among multiple developers so that application development time will be reduced.
- Reuse code: Once a module is written, the same module can be reused in another application.
- Easy to incorporate changes, as required so that the code will be more maintainable.
- Easy to understand as the code is written in a single unit called module.
- Cleaner Code



Characteristics of well defined modules

- They always return same set of results for same set of inputs
- They perform a single well defined functionality
- **High Cohesion** – do one thing, and do it well
- **Low Coupling** – reduce dependencies between modules

Ideally when there is “high cohesion” and “low coupling”, one can change the implementation of a routine without impacting the call interface.

For example: A sort routine can change its algorithm from “Bubble” to “Quick sort” – without causing the calling code to break.

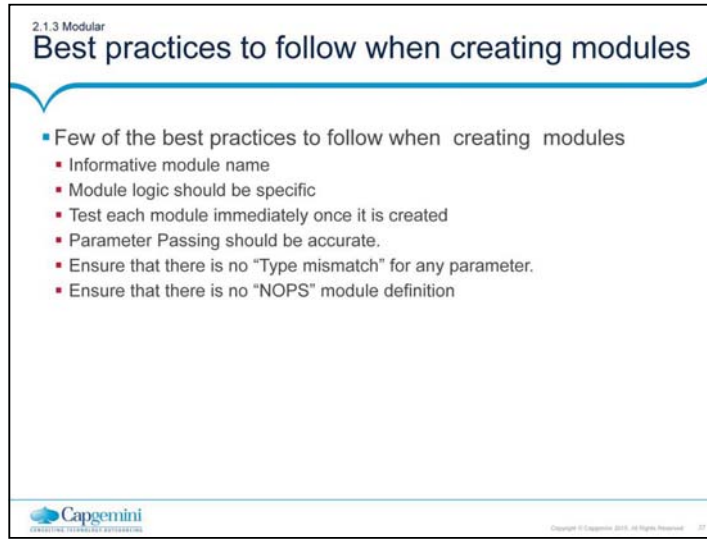
- **Meaningful names**

Use Verb-noun format (be specific):

For example: Read-Employee-Record, Calculate-Deductions, Print-Pay-slip

Avoid generic names:

For example: Process-inputs, Handle-calculations



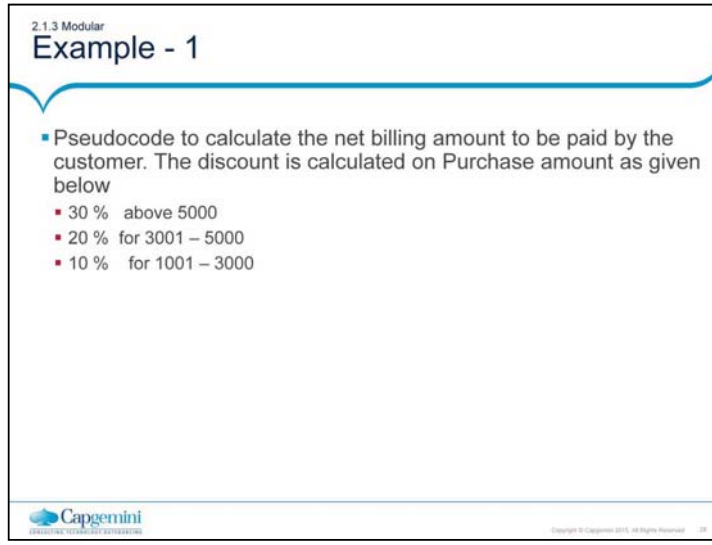
Best Practices to follow when creating modules

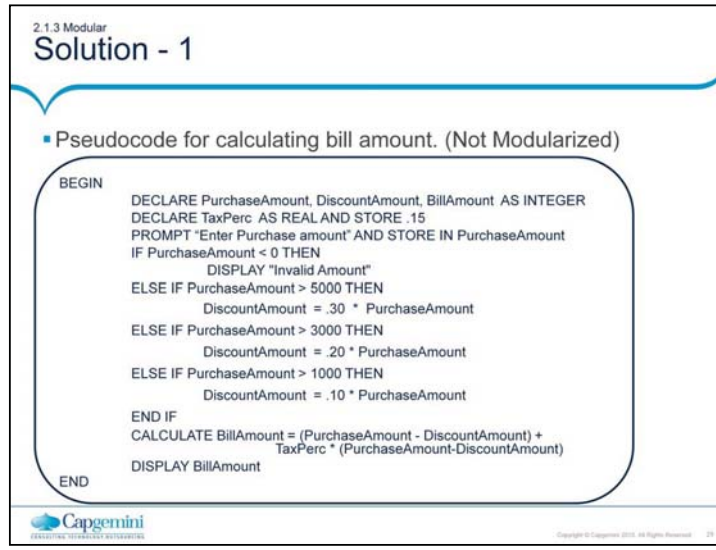
- **Informative Module Name:** Give the module an informative name like getProductPrice, calculateSalary, printDetails.
- **Module logic should be specific: I**dentify a clear purpose for the module before you start writing
- Test each module as it is created by performing unit testing
- Parameter passing should be accurate: Ensure that the number of parameters, and the sequence is correct for the module call.
- Ensure that there is no "Type mismatch" for any parameter.
- Ensure that there is no "NOPS"(No Operation) Module definition. i.e. Empty module definition shouldn't be there.

calculateTotal(Integer price, Integer quantity)

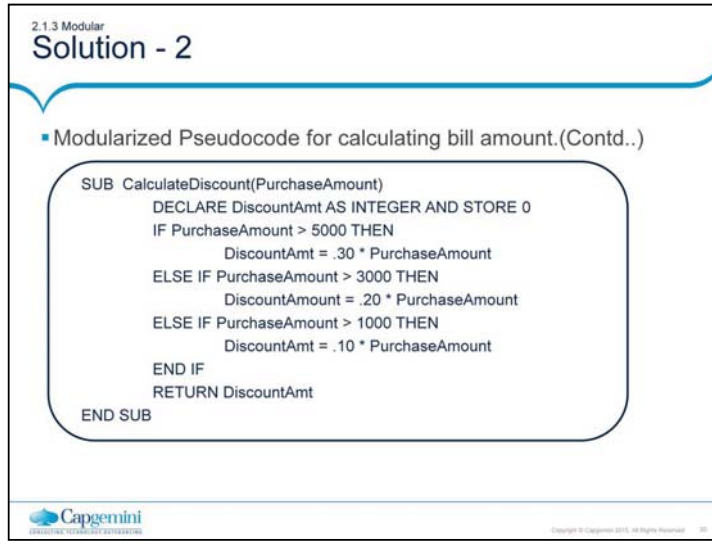
Refer the valid and invalid statements to invoke a module

- **calculateTotal(3,5); //Valid**
- **calculateTotal(4,3,4); //Invalid**
- **calculateTotal('Test',3); //Invalid**

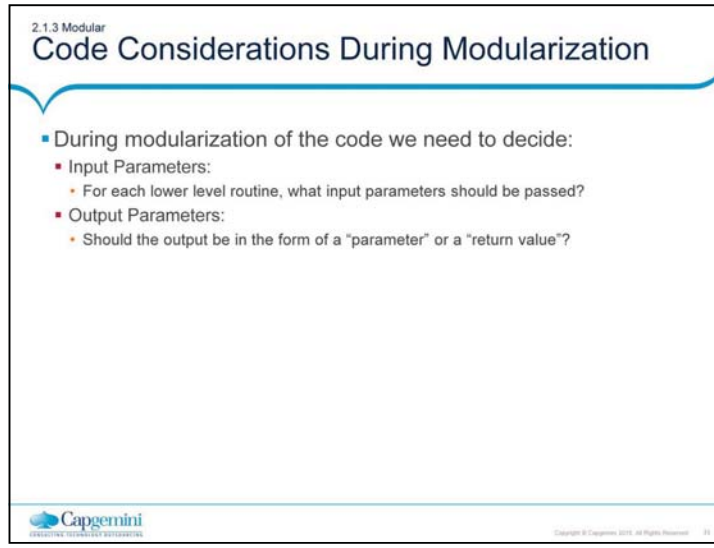

2.1.3 Modular
Example - 1
Pseudocode to calculate the net billing amount to be paid by the customer. The discount is calculated on Purchase amount as given below
30 % above 5000
20 % for 3001 – 5000
10 % for 1001 – 3000
Capgemini



The above pseudocode for calculating bill amount is not modularized



The above pseudocode for calculating bill amount is modularized


2.1.3 Modular
Code Considerations During Modularization
During modularization of the code we need to decide:
Input Parameters:
For each lower level routine, what input parameters should be passed?
Output Parameters:
Should the output be in the form of a "parameter" or a "return value"?
Capgemini

2.1.3 Modular

Code Considerations

- Analyze parameters and return values for Accept_Employee_Details, Compute_Deductions , Compute_Gross_Pay

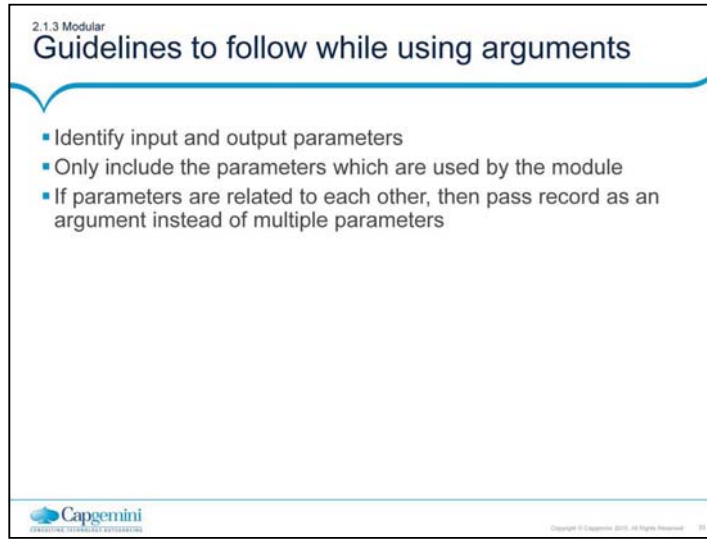
```
SUB Accept_Employee_Details()
      ACCEPT emp_code, Basic
END SUB
```

```
SUB Compute_Deductions(Basic)
  Provident_Fund = 0.12 * Basic
  Prof_Tax = 200
  Compute_Income_Tax (Basic)
END SUB
```

```
SUB Compute_Gross_Pay(Basic)
  HRA = 0.5 * Basic /*** House Rent Allowance ***/
  OPA = 0.3 * Basic /*** Offshore project allowance ***/
  Conveyance = 1700
  Gross = Basic + HRA + OPA + Conveyance
END SUB
```

Capgemini

Copyright © Capgemini 2015. All Rights Reserved.



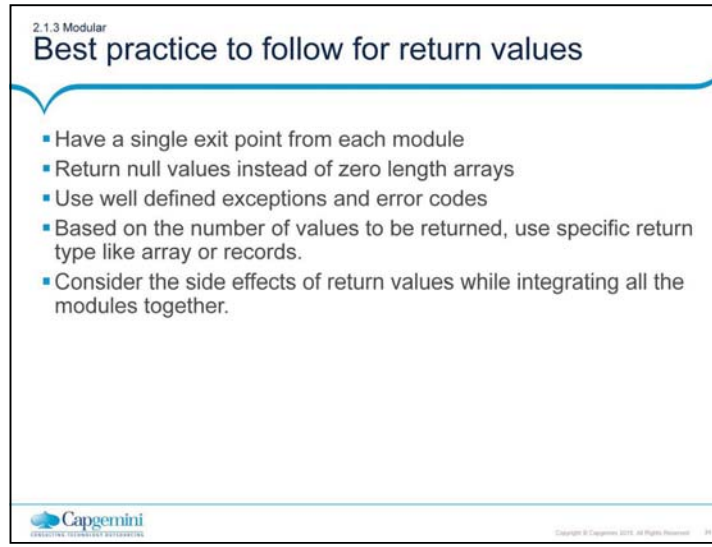
Guidelines to follow while using arguments

- Identify input and output parameters: After analyzing the requirement, identify input and output parameters before writing module code. For an Example, if you want to create a module for implementing a logic related to retrieving product details based on product id, then input parameter is productid and output parameter is variable of type record which contains product details.
- Only include the parameters which are used by the module, never pass unused parameters.
- If parameters are related to each other, then pass record as an argument instead of multiple parameters which strive for high cohesion and low coupling. For an example, refer the below module code snippet to add an employee details

```
SUB addEmployee(empId, name, salary)
END SUB
```

Instead use the below module signature

```
SUB addEmployee(Employee emp) //Consider Employee is a record
END SUB
```



Best Practice to follow for return values

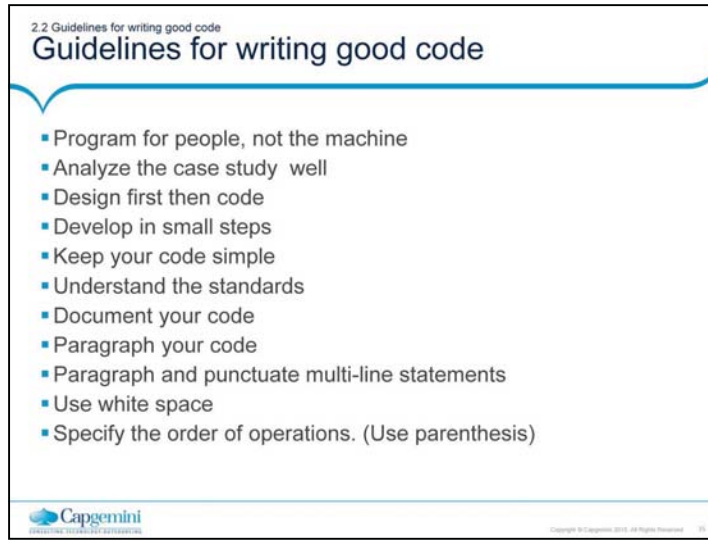
- Use only one return statement in each module as shown below:

```
SUB checkDigit(number)
        DELCARE result AS BOOLEAN
        IF(num>0) THEN
                result=true
        ELSE
                result=false
        END IF
        RETURN result
END SUB
```

- Return null values instead of zero length arrays.
- Return exceptions/error code if an invalid input is accepted.

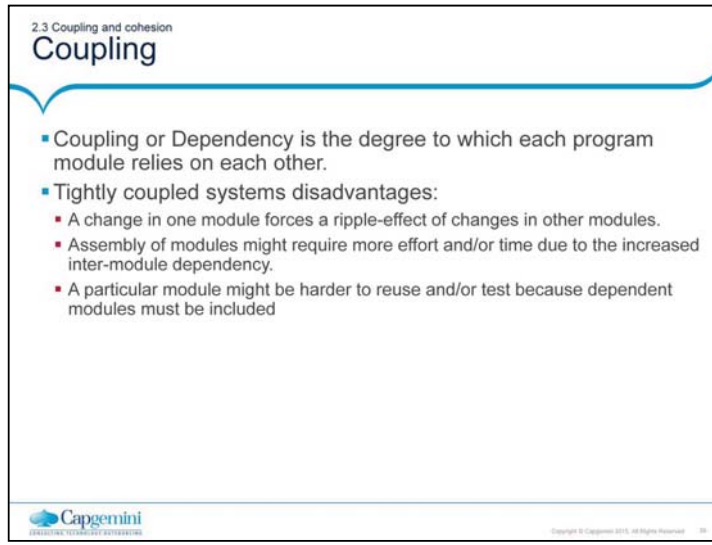
```
SUB getProductPrice(productId)
                DECLARE errorcode AS INTEGER AND STORE 0
                IF(elementfound(productId)) THEN
                      RETURN productPrice
                ELSE
                                errorcode = -1
                                RETURN errorcode;
                END IF
END SUB
```

- Use array as return type if more than one value has to be returned of same type or use record if more than one value has to be returned of different type.



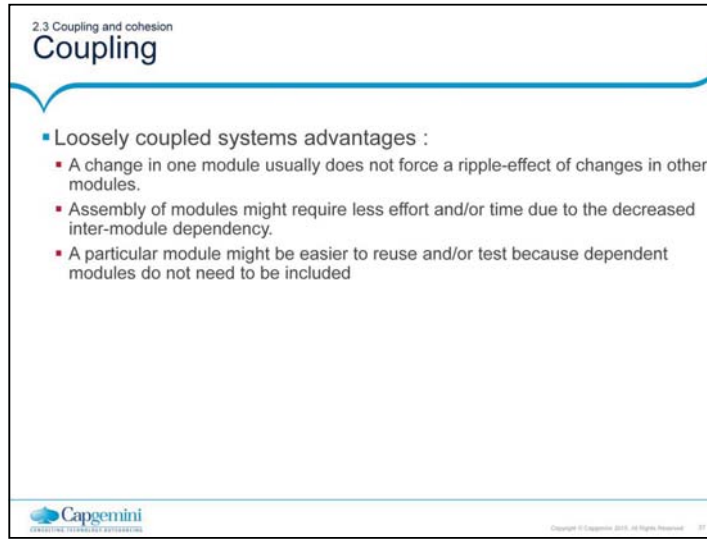
Guidelines for writing good code

- While writing program, keep in your mind that program will be used by people, so make program to be user friendly.
- Before starts with development , analyze the case study well to incorporate all the requirements.
- Do the high level (like database design) and low level designing(pseudocode) of an application before working in coding phase.
- Create program in an incremental approach, so that after implementation of each logic it can be easily tested for finding defects.(As finding defects in earlier stage, decreases cost and save development time).
- Make your code to more simple by avoiding the usage of complex data structure/constructs
- Understand the standards:
    - All the coding standards as well as processed should be understandable and apply the same in your code.
    - Believe in them
    - Make them part of your quality assurance process
    - Adopt the standards that make the most sense for you
- Document your code using comments for making it to be more readable
- Paragraph your code by applying modularity to make code reusable.
- Use whitespace as a layout technique to make code more readable
- Specify the order of operations using parenthesis for prioritizing



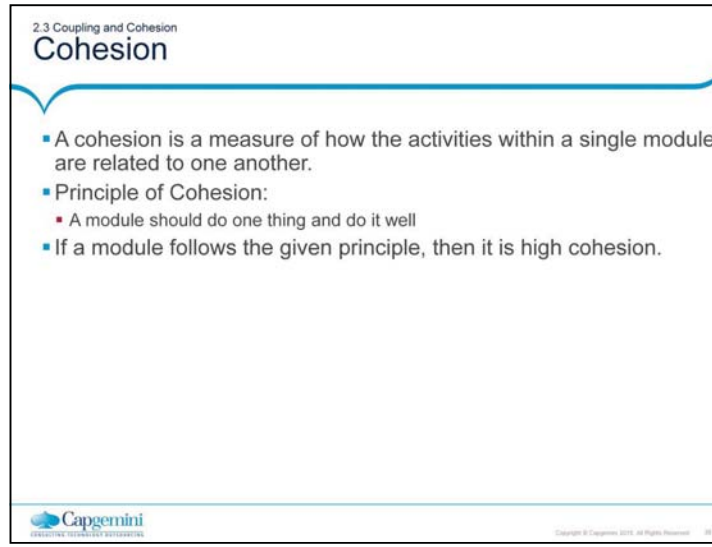
If a module does only one thing, we need to pass less parameters.
If a module does too many things, we need to pass more parameters.
Passing more parameters means high coupling.

We should strive for low coupling.



Low Coupling – reduce dependencies between modules

- Note that changes in one routine should not normally impact other routines, as long as the interface is the same.
- Remember that, in case too many things are done in one routine, a lot of data needs to be shared. This increases the dependencies, and also the chances of defects
- Consider “Smaller interface” (low coupling) versus “long list of parameters” (high)
- Consider data sharing through “parameters” (low) versus “global data” or “global files” (high)
- Note that passing “flags” that control the processing implies High coupling.



Writing function with high cohesion is always good practice.
Ideally, we should strive for **High Cohesion** and **Low Coupling**.

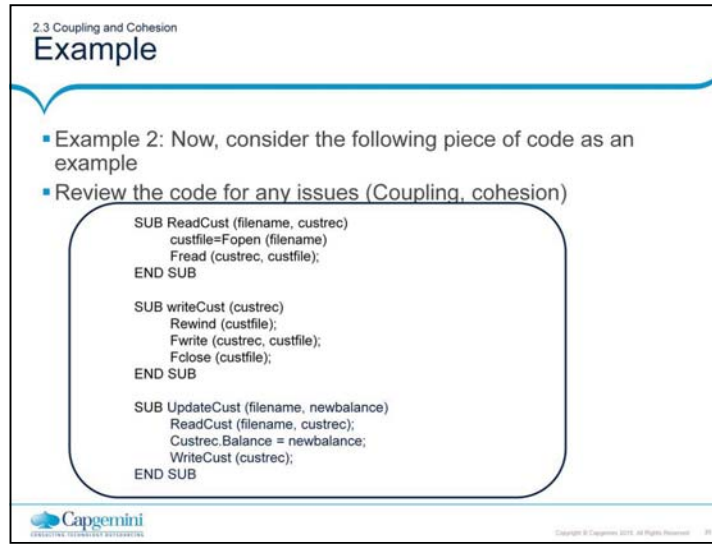
For example:
1. Sin (x); Cos (x); Tan (x); (High Cohesion, Low Coupling)

Note:
A good program requires high cohesion and low coupling.

2.Trig (Type, x)
where type is Sin, Cos, or Tan; (Less Cohesion, High Coupling)

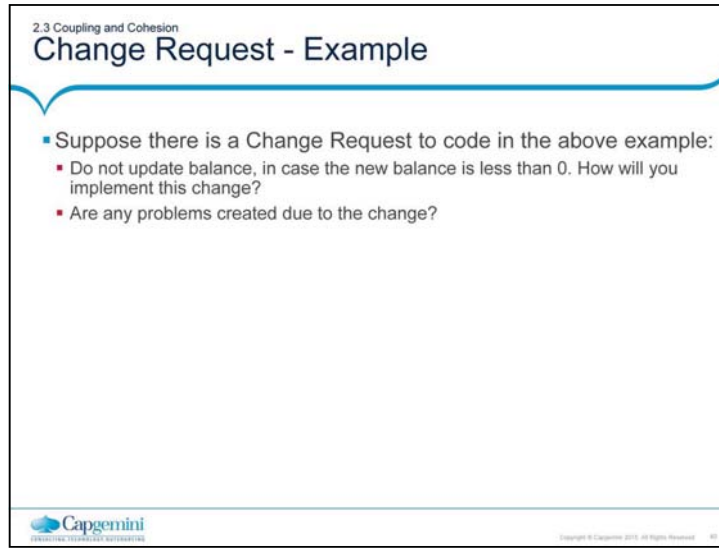
Note:
Since in function trig we are trying to add all three functions together, we require to pass one extra parameter i.e. Type. This parameter indicates whether to calculate Sin, Cos, or Tan.

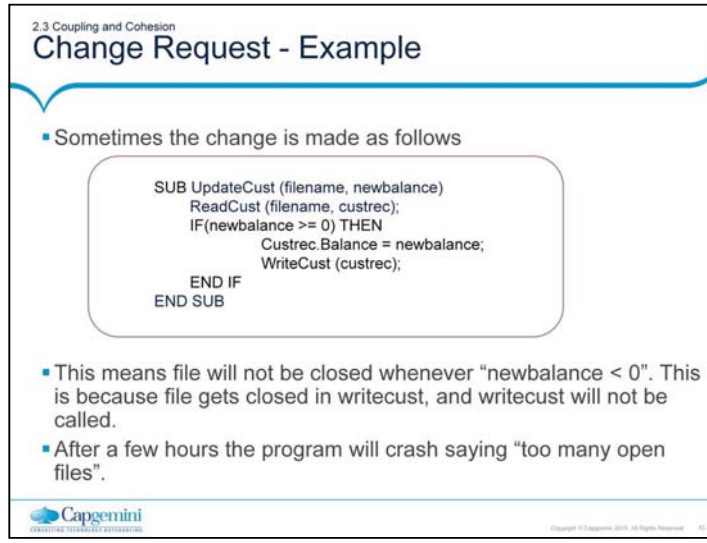
a. More number of parameters are required to pass, so it is High Coupling.
b. b. The function is not performing just a single task, so it is Low Cohesion.

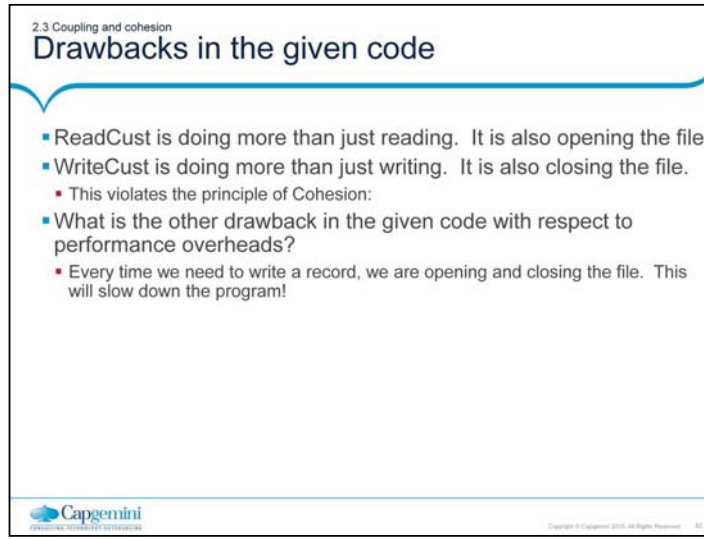


Consider Fopen, Fwrite, Rewind, Fclose are predefined functions.

Fopen : To open a file
Fwrite : To write data to a file
Fclose : To close the opened file
Rewind :Moving cursor back to the first position of a file.

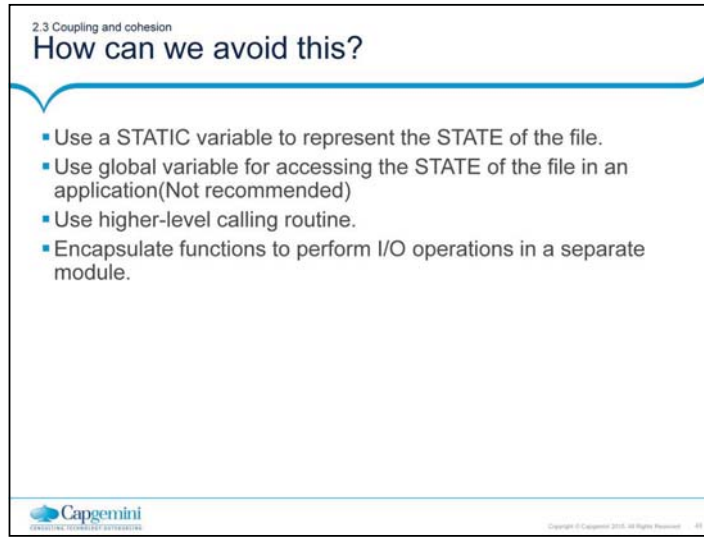

2.3 Coupling and Cohesion
Change Request - Example
Suppose there is a Change Request to code in the above example:
Do not update balance, in case the new balance is less than 0. How will you implement this change?
Are any problems created due to the change?
Capgemini


2.3 Coupling and Cohesion
Change Request - Example
Sometimes the change is made as follows
SUB UpdateCust (filename, newbalance)
ReadCust (filename, custrec);
IF(newbalance >= 0) THEN
Custrec.Balance = newbalance;
WriteCust (custrec);
END IF
END SUB
This means file will not be closed whenever “newbalance < 0”. This is because file gets closed in writecust, and writecust will not be called.
After a few hours the program will crash saying “too many open files”.
Capgemini



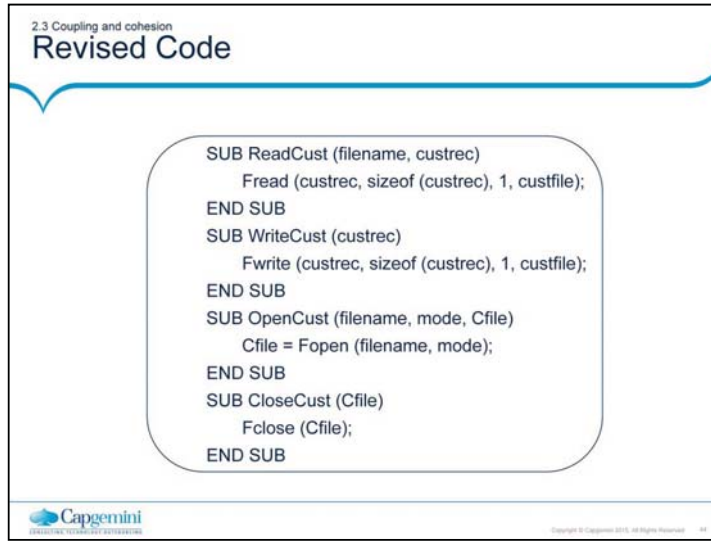
Cohesion:

- It means, the function should do one thing only, and the code should not be mixed up.
- As a result, the code becomes more readable and more maintainable.

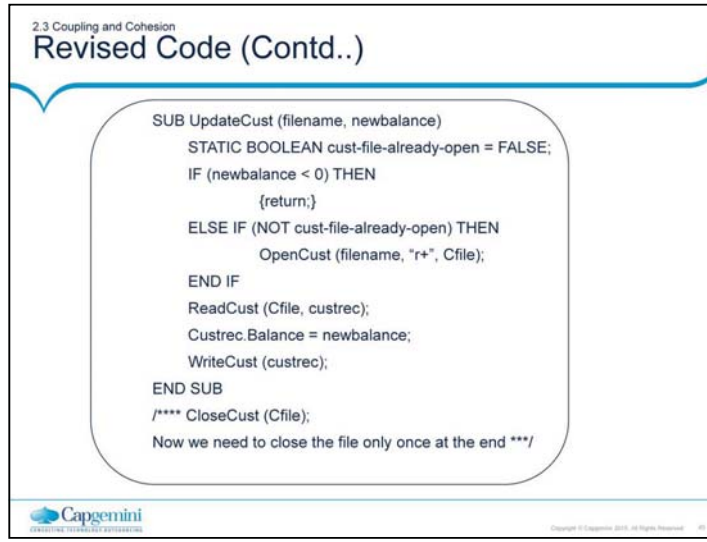


Steps to be taken care for avoiding drawbacks on the implementation of cohesion and coupling:

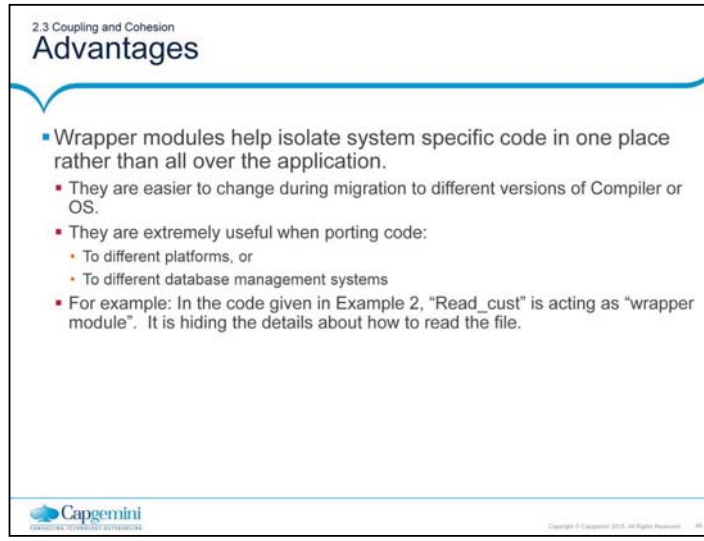
- Use a STATIC variable(cust-file-already-open) to represent the STATE of the file as OPEN or not
- We cannot access the STATIC variable cust-file-already-open outside this routine
- We will be able to access it outside the routine if we declare it as GLOBAL. However, that is not a good practice.
- It is better to open and close the cust-file in the higher-level calling routine. The routine should call UpdateCust only to write the record.
- The calls to fread / fwrite / fopen / fclose are encapsulated or wrapped in the modules ReadCust / WriteCust / OpenCust / CloseCust respectively.

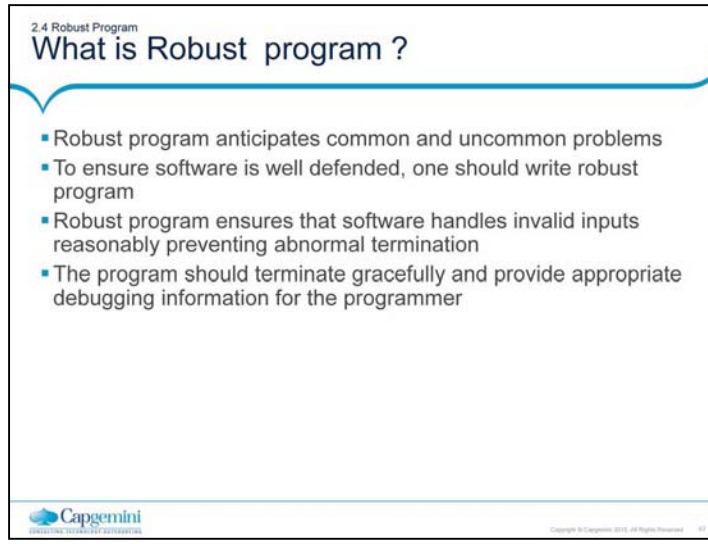


The calls to fread / fwrite / fopen / fclose are encapsulated or wrapped in the modules ReadCust / WriteCust / OpenCust / CloseCust respectively.

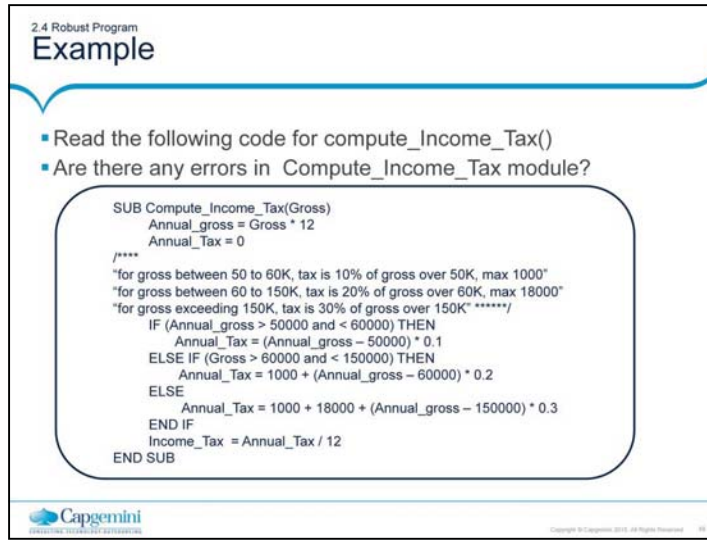


Used STATIC variable(cust-file-already-open) to represent the STATE of the file as OPEN or not


2.3 Coupling and Cohesion
Advantages
Wrapper modules help isolate system specific code in one place rather than all over the application.
They are easier to change during migration to different versions of Compiler or OS.
They are extremely useful when porting code:
To different platforms, or
To different database management systems
For example: In the code given in Example 2, "Read_cust" is acting as "wrapper module". It is hiding the details about how to read the file.
Capgemini

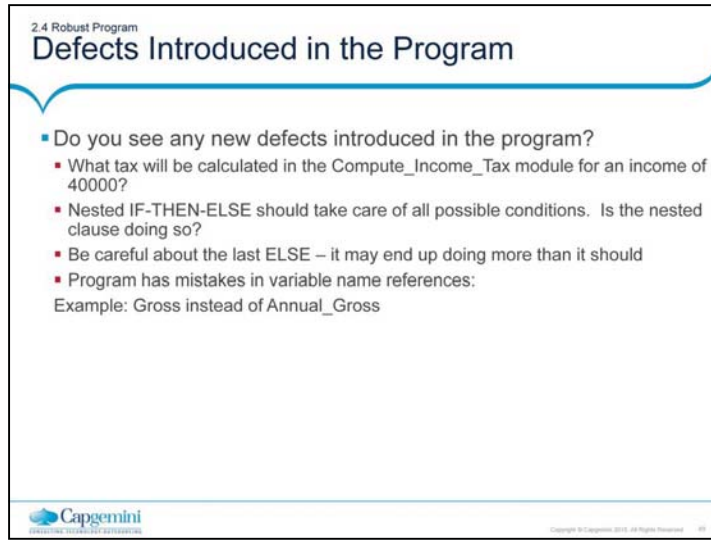


Robust program ensures that software handles invalid inputs reasonably preventing abnormal termination. Robust program anticipates common and uncommon problems. To ensure software is well defended, one should write robust program. The program should terminate gracefully and provide appropriate debugging information for the programmer

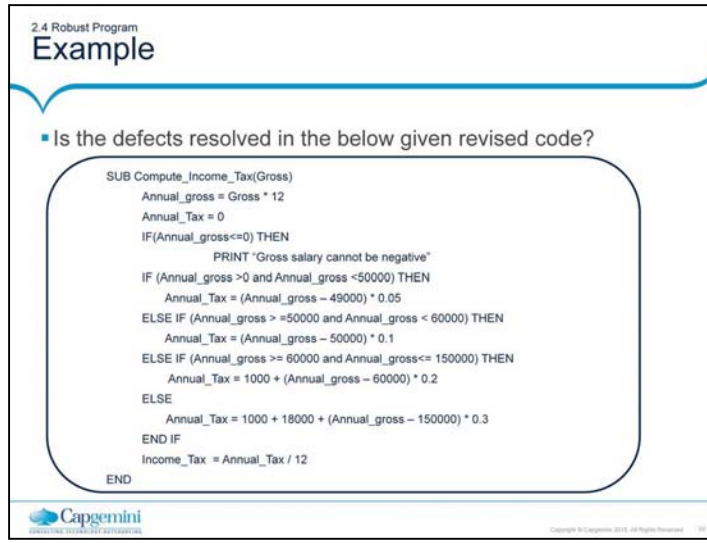


In the above program we are not considering a case where annual_gross is less than 40000. As a result, in case the statements at the bottom of the program are executed, we will get wrong result.
Note: When we use a nested IF, be careful while writing the last ELSE. This is because, the program will be executed for all unhandled conditions, as well.

The above program has one more mistake. We have used Gross instead of Annual_Gross. This error will not be detected by the compiler because we are using Gross as a variable form during calculation of monthly gross.
This will result in wrong output.



For resolving all the defects mentioned in the slide, rework on the code.



In the above program we have considered a case where annual_gross is less than 40000. We have renamed, Gross as Annual_Gross. The statements in ELSE block will be executed only if Annual_Gross value is greater than 150000.

Annual Gross salary is calculation is taken care based on the below data

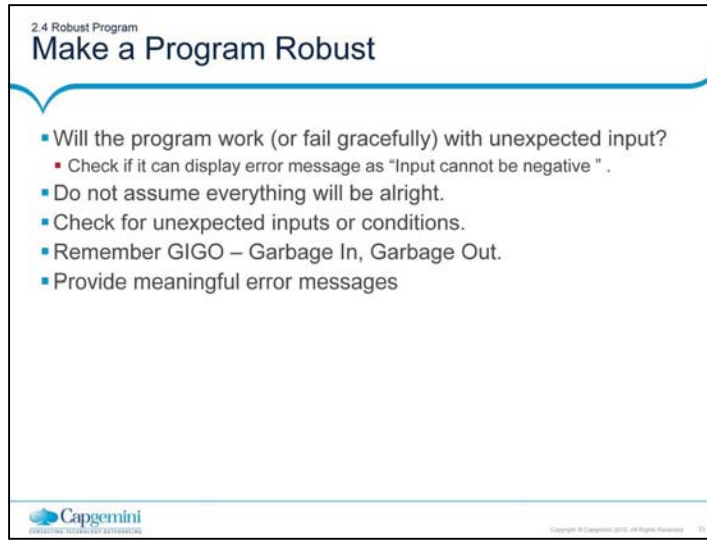
/****

"for gross less than 50K, tax is 5% of gross over 49K"

"for gross between 50 to 60K, tax is 10% of gross over 50K, max 1000"

"for gross between 60 to 150K, tax is 20% of gross over 60K, max 18000"

"for gross exceeding 150K, tax is 30% of gross over 150K" ******/



Is the program robust?

- We test the program with different expected values, and it works fine as per our expectation.
- However, what if somebody provides an unexpected input?
  - will the program give proper error message and stop gracefully, or
  - will the program fail and give some garbage output
- Hence to make the program robust, add appropriate messages to handle unexpected input as mentioned in the slide.
- How can we make the program robust?

  Check if it can display error message as "Input cannot be negative"

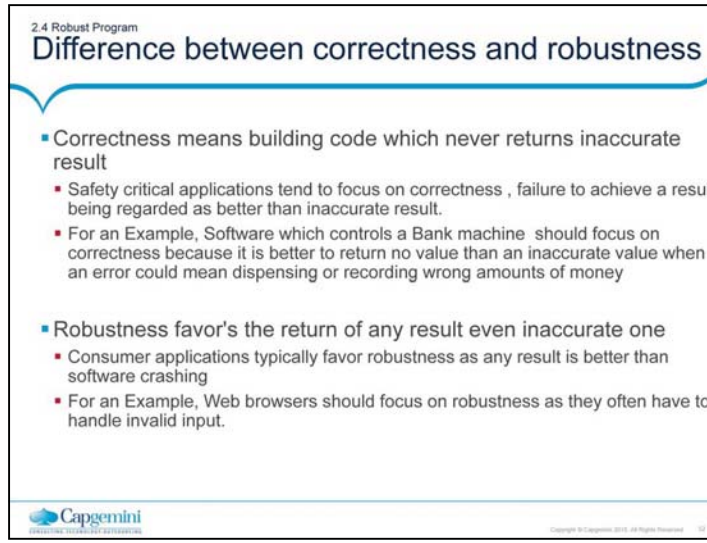
  Check if it can accept extreme values for inputs.

  For example: Basic = 1 crore

  Check if it can handle Output / Printer related problems.
- Check whether:

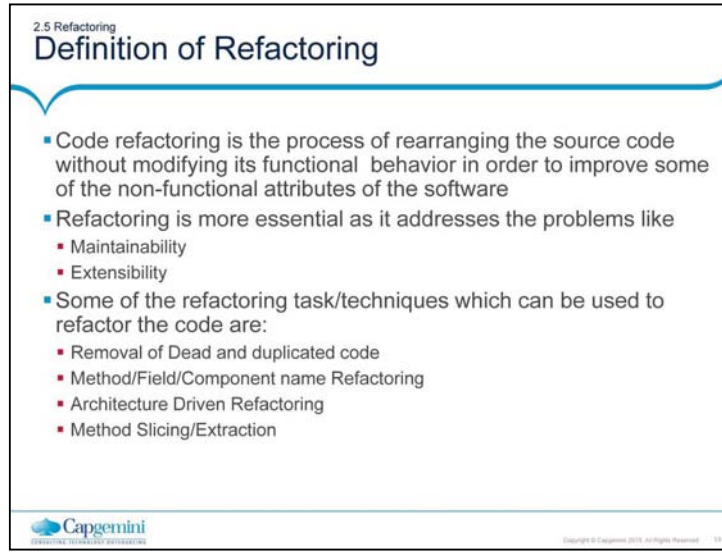
  "Can the Tax calculated be negative?" or

  "Can the Net Pay calculated be negative?"



First determine whether you want the program to primarily offer robustness or correctness

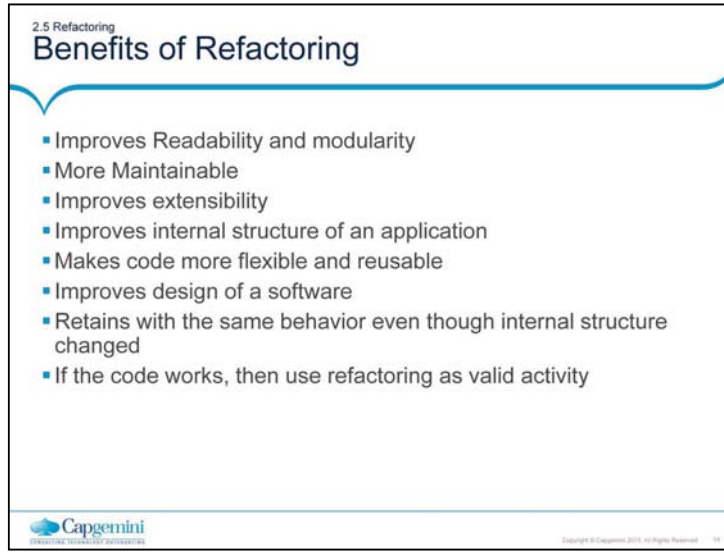
- Examples of Robustness
    - Consumer applications typically favor robustness as any result is better than software crashing
    - You may want a word processing program to sometimes display unwanted characters rather than to shut down when it detects them
    - Web browsers should focus on robustness as they often have to handle invalid input.
- Examples of Correctness
    - Safety critical applications tend to focus on correctness , failure to achieve a result being regarded as better than inaccurate result.
    - E,g software which controls radiation equipment for patients is best shut down if it receives bad input for a radiation dosage.
    - Software which controls a Bank machine should focus on correctness because it is better to return no value than an inaccurate value when an error could mean dispensing or recording wrong amounts of money



Refactoring is the process of clarifying and simplifying the design of existing code, without changing its behavior. Refactoring requires care because it can introduce bugs to the code

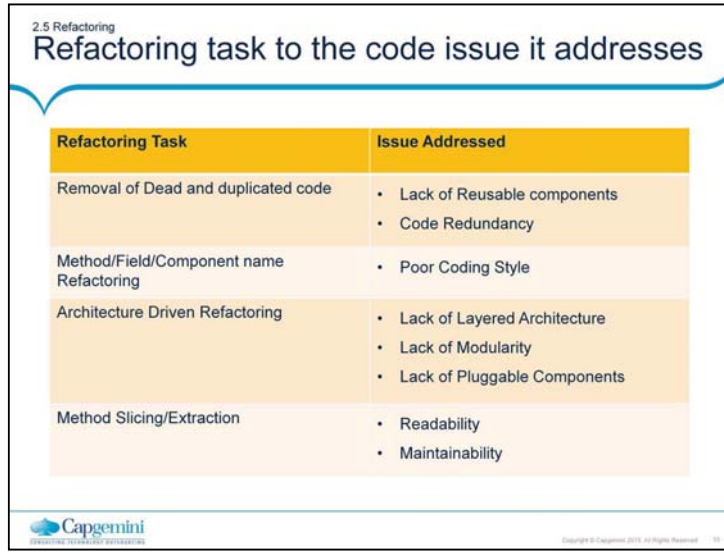
Refactoring is more essential as it addresses the problems like
- Maintainability – Code is not maintainable
- Extensibility – Very difficult to add new feature or upgrade an existing feature

Refactoring task/techniques are:
- Removal of Dead and duplicated code - Remove the unused variables/code, unreachable statements and duplicated code.
- Method/Field/Component name Refactoring – Name of a method/field/component might be confusing, provide a meaningful name for the same and ensure the changes are reflected in other references wherever the variable/field/component is used.
- Architecture Driven Refactoring - Few functionalities in an application can be potentially reused in other applications. In order to reuse such a functionality, separate the code in a separate component by adhering to the architecture design.
- Method Slicing/Extraction - Longer method needs to be broken up into smaller ones to enhance readability and maintainability. Also need to achieve cohesion by implementing only one logic in a method with more suitable name.

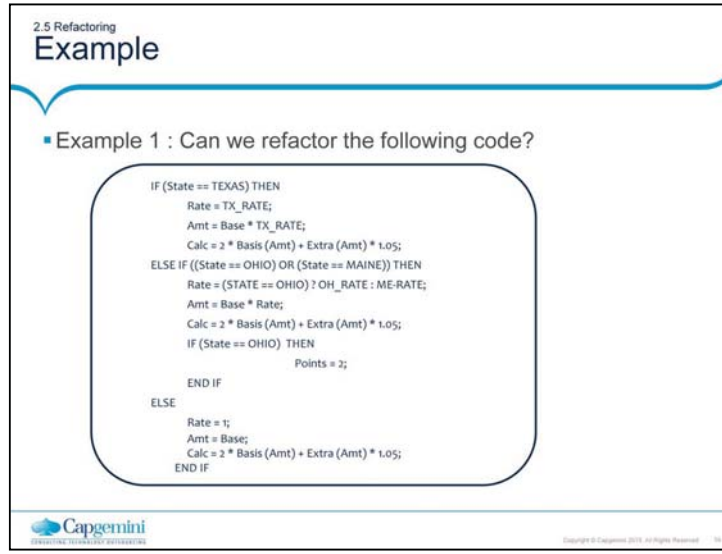


Benefits of Refactoring:

- Improves code readability and modularity by improving code structure and design
- Reduced complexity to improve the maintainability of the source code
- Improves extensibility: Easier to add new features
- Refactoring involves improving internal structure of a software without altering its external behavior
- Goal of refactoring is to make the software more flexible and reusable
- It involves changing the design of the software after it has been coded to improve the design of a software. Refactoring is mainly used to improve badly designed software.
- Ensure software's external behavior remains the same after refactoring
- Refactoring is a valid activity only when the code is already in working state



Each refactoring task is addressed to the code issue it addresses.

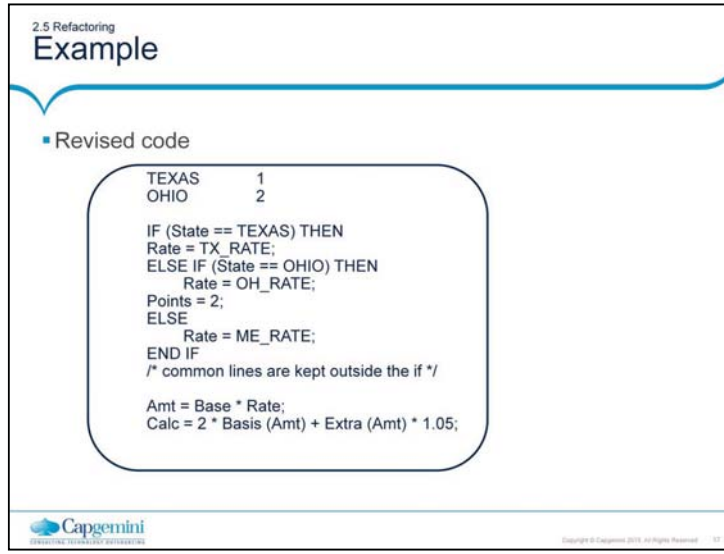


The above code is used to calculate tax based on the state. Tax rate varies for each state but similarity exists in the tax calculation.

Rate, Amt and Calc variables are assigned values in each if-else condition.

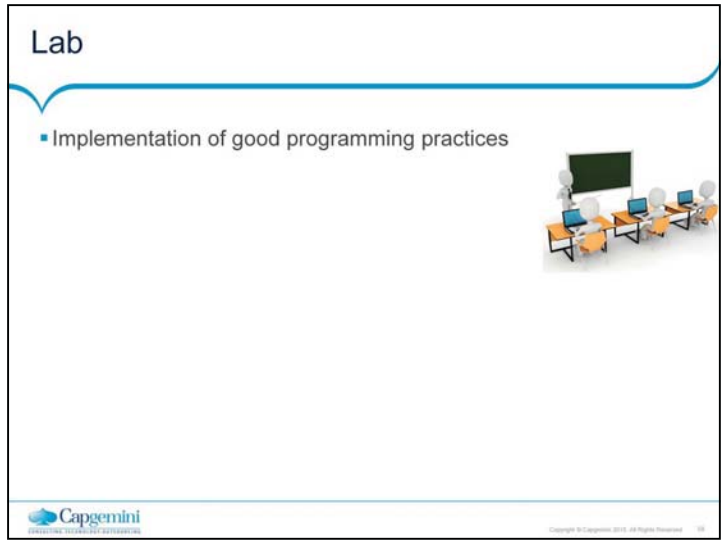
Use Refactoring for performing the below task:

- Avoid Duplicate code in each if else condition
- Common code should be kept outside if condition

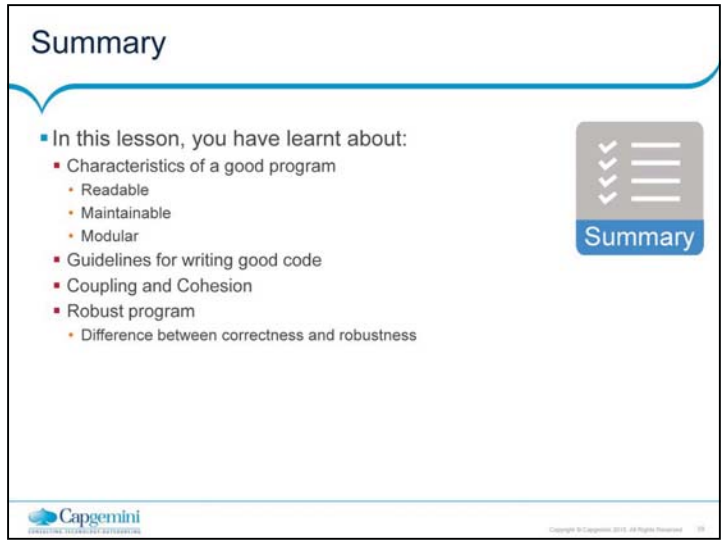


Note: The code is more efficient and readable as compared to the previous code.

Are any further improvements possible?
If the program is used for three states, it is likely to be used for other states, as well, in the future.
Instead of using a nested IF ELSE statement, using a table or an array for tax rates of all states will make it easy to add more states later on.



Write pseudocode for all assignments in lab 2 to apply the characteristics of good programming practices like modularity approach.


Summary
In this lesson, you have learnt about:
Characteristics of a good program
Readable
Maintainable
Modular
Guidelines for writing good code
Coupling and Cohesion
Robust program
Difference between correctness and robustness
Summary
Capgemini

Review Question

- Question 1: Why should you make code self documenting ?
  - A. To help the review process
  - B. To improve the quality of the code
  - C. To make it easier to debug
  - D. To meet international standards

- Question 2: How can you minimize hard coding ?
  - A. By changing named constants in several places in your code
  - B. By using enumerated types instead of named constants
  - C. By using named constants to replace Booleans when you require more answers than true or false

Capgemini

Copyright © Capgemini 2015. All Rights Reserved

Review Question

- Question 3: What are the objectives of good layout ?
  - A. An accurate logical structure
  - B. To improve compatibility with compiler
  - C. To improve readability
  - D. To withstand modifications

- Question 4 : What are the guidelines for good layout ?
  - A. Use braces as boundaries to demarcate block of code
  - B. Use indentation to show logical structure of code
  - C. Use line breaks for statements over 80 characters
  - D. Use white space sparingly

Capgemini

Copyright © Capgemini 2015. All Rights Reserved

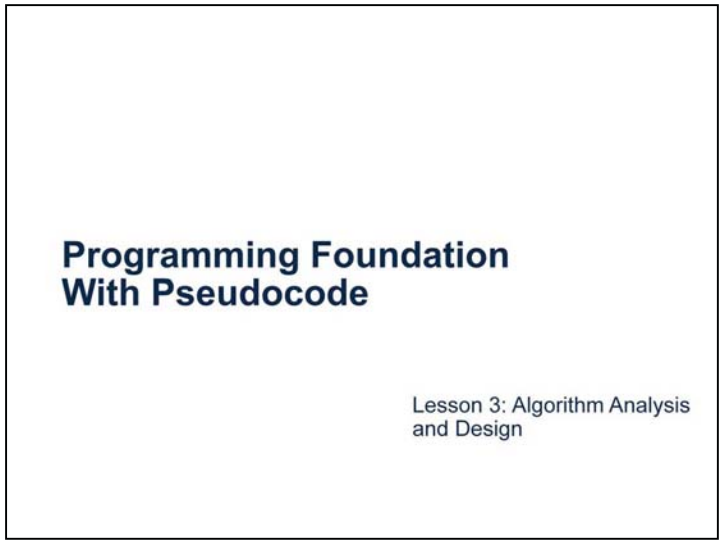
Review Question

- Question 5 : You are writing a module to update the visitor counter on a web page ,what steps can you take to make this module as strong as possible ?
  - A. Call the module pageCountUpdate()
  - B. Call the module webPage6()
  - C. Test the module as soon as it is complete
  - D. Wait until all modules have been added to the program before testing it

- Question 6 : Which of these applications should be robust ?
  - A. Bank machine software
  - B. Radiation machine software
  - C. Web browser software
  - D. Word processing software

Capgemini

Copyright © Capgemini 2015. All Rights Reserved

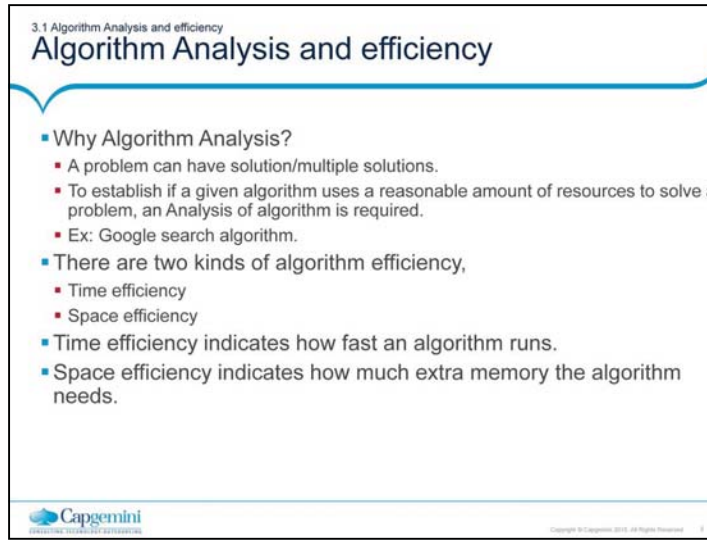

Programming Foundation
With Pseudocode
Lesson 3: Algorithm Analysis
and Design

Lesson Objectives

- To understand the following concepts
  - Algorithm Analysis and efficiency
  - Measuring Unit for Algorithm
  - Order of Growth
    - Asymptotic notations
  - Best/Worst/Average case
  - Example

Capgemini

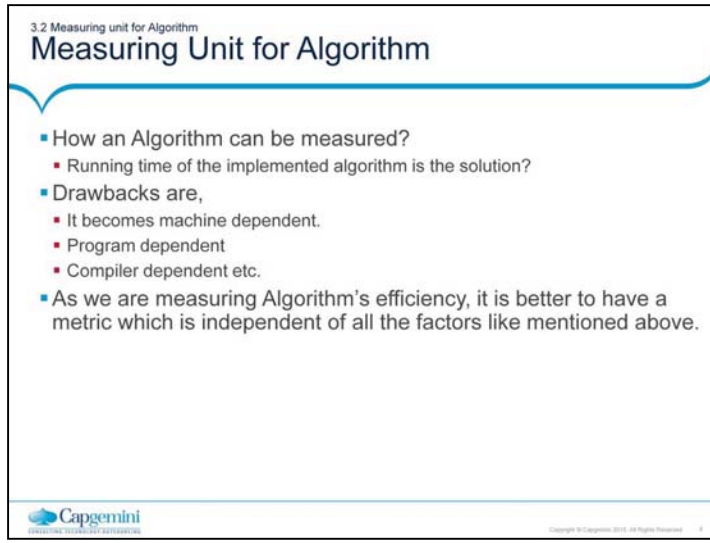
Copyright © Capgemini 2015. All Rights Reserved



Algorithm is a step by step instruction to solve a given problem. Algorithm can have multiple solutions.
For Ex: We have multiple solutions to find GCD of two numbers. Hence we need a method to compare/analyze between multiple solutions/algorithms and find the efficient solution. This chapter explains how to analyze and find efficiency of computer algorithms.

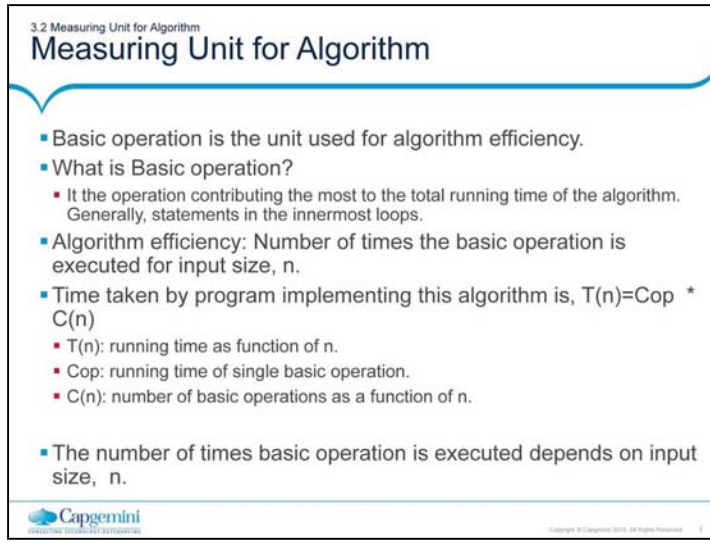
Why Algorithm Analysis?

An analysis is useful to establish if a given algorithm uses a reasonable amount of resources to solve a problem. If the algorithm needs too many resources then, even if it is correct, it cannot be used in practice. Such an algorithm is not an efficient one. When we are talking about efficiency we can refer to space efficiency (it deals with the space required by the algorithm) or to time efficiency (it indicates how fast an algorithm runs).
One must usually make compromise between time and space efficiency. Compromise usually depends on the situation or application demands/requirements. One can be achieved with the trade off of another. No doubt, space was very much an issue in earlier days but even today for mobile and embedded applications, we will be working with limited memory.

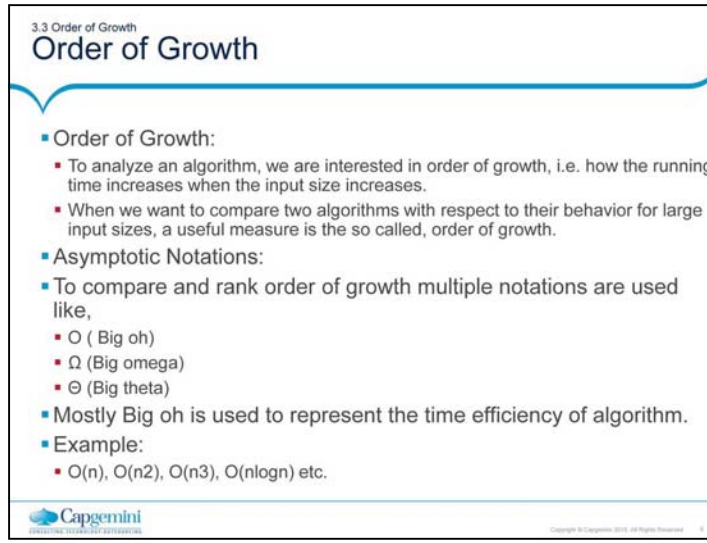


When we talk about comparison, we need some unit. So when we talk about Algorithms measurement, time cannot be a good unit. Drawbacks are,

- It becomes machine dependent.
- Program dependent
- Compiler dependent etc.



Hence a proper measuring unit will be basic operation, the operation contributing the most to the total running time.
Basic operation: Maximum times executing statement in algorithm will be considered as basic operation. Usually statements in the inner loop are executed maximum number of times.
Other thing to note is that, there is no meaning in comparing the algorithms with lower input size. There is no much difference. Algorithms vary in efficiency only when their input size is very large. Hence while discussing further with the input size, n, in this chapter, we mean a large value for n.
To calculate efficiency of algorithm, compute the number of times the basic operation is executed on input size of n.
Algorithm efficiency is the number of times basic operation is executed for the given input size, n.



Asymptotic Notations:

➢O : A function t(n) is said to be in O(g(n)), if t(n) is bounded above by some constant multiple of g(n) for all n, i.e. if there exist some positive constant c and some nonnegative integer n_0 such that

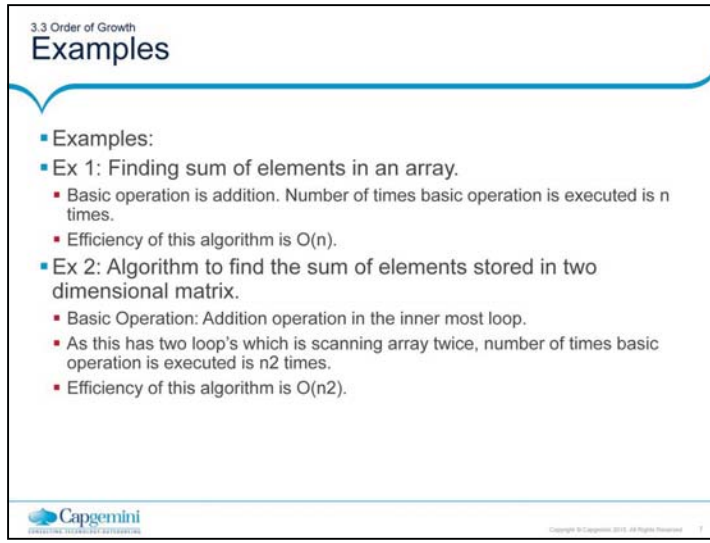
$t(n)<=c(g(n))$ for all $n>=n_0$

➢Ω : A function t(n) is said to be in Ω(g(n)), if t(n) is bounded below by some constant multiple of g(n) for all n, i.e. if there exist some positive constant c and some nonnegative integer n_0 such that

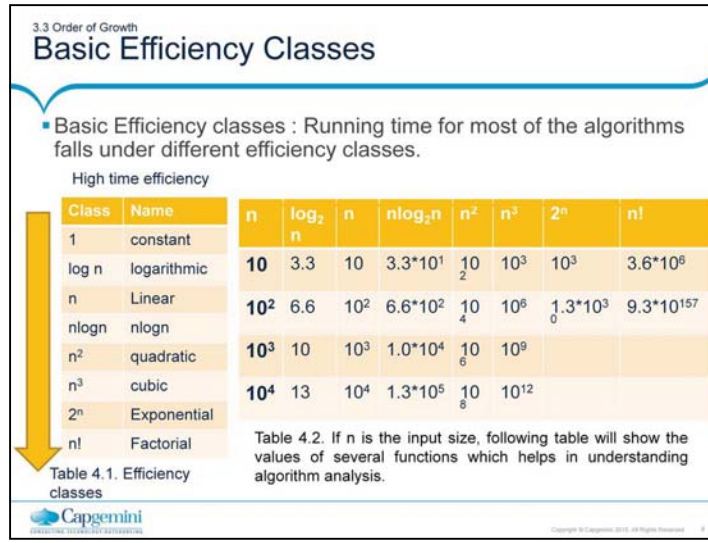
$t(n)>=c(g(n))$ for all $n>=n_0$

➢Θ : A function t(n) is said to be in Θ(g(n)), if t(n) is bounded both above and below by some constant multiple of g(n) for all n, i.e. if there exist some positive constant c1 and c2 and some nonnegative integer n0 such that

$c_2g(n)<=t(n)<=c_1(g(n))$ for all $n>= n_0$



Write an algorithm for the above examples before finding its efficiency.



Constant Time: When instructions of program are executed once or at most only a few times , then the running time complexity of such algorithm is know as constant time. it is independent of the problem's size. It is represented as O(1). For example, linear search best case complexity is O(1)

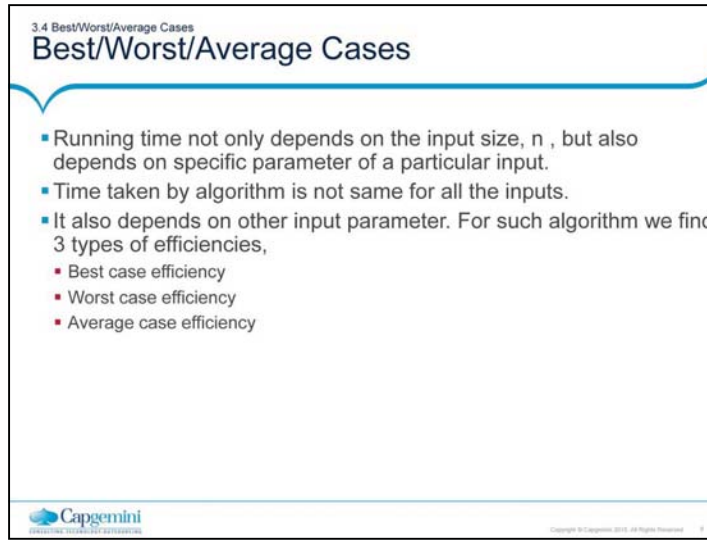
Logarithmic: The running time of the algorithm in which large problem is solved by transforming into smaller sizes sub problems is said to be Logarithmic in nature. In this algorithm becomes slightly slower as n grows. It does not process all the data element of input size n. The running time does not double until n increases to n2 . It is represented as O(log n). For example binary search algorithm running time complexity is O(log n).

Linear: In this the complete set of instruction is executed once for each input i.e input of size n is processed. It is represented as O(n). This is the best option to be used when the whole input has to be processed. In this situation time requirement increases directly with the size of the problem. For example linear search Worst case complexity is O(n).

Quadratic : Running time of an algorithm is quadratic in nature when it process all pairs of data items. Such algorithm will have two nested loops. For input size n, running time will be O(n2). Practically this is useful for problem with small input size or elementary sorting problems. In this situation time requirement increases fast with the size of the problem. For example insertion sort running time complexity is O(n2).

Exponential: Running time of an algorithm is exponential in nature if brute force solution is applied to solve a problem. In such algorithm all subset of an n element set is generated. In this situation time requirement increases very fast with the size of the problem. For input size n, running time complexity expression will be O(2n).For example Boolean variable equivalence of n variables running time complexity is O(2n). Another familiar example is Tower of Hanoi problem where running time complexity is O(2n).

Factorial: Typical for algorithms that generate all permutations of n-element set.



For few algorithms, running time not only depends on the input size, n , but also depends on specific parameter of a particular input.

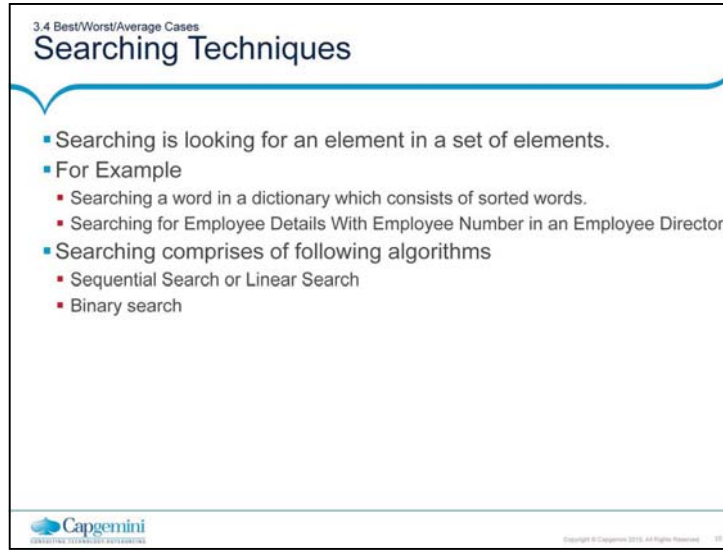
Time taken by algorithm is not same for all the inputs. Hence time taken by algorithm may be different for the same size of input, n. Because It also depends on other input parameter.

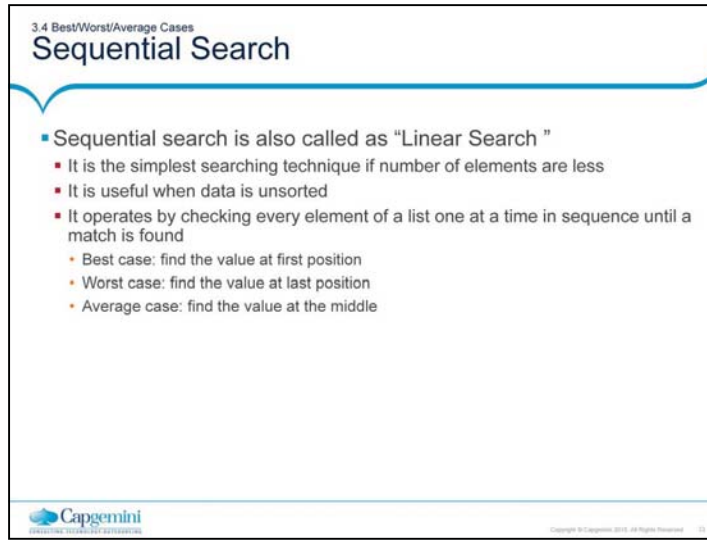
For example, Consider that one is looking for a certain card in a deck of unsorted cards. One card is opened at a time from the deck. Now there are multiple cases,

- We can find the card for the first time.
- We can never find the card.
- We can find the card at the last.
- Or we can find the card somewhere in between.

So here, efficiency differs for different scenarios. This is because, it doesn't matter, how many total cards were found, but it depends on how soon the card is found. Hence for such algorithms we find,

- Best case
- Worst case
- Average case


3.4 Best/Worst/Average Cases
Searching Techniques
Searching is looking for an element in a set of elements.
For Example
Searching a word in a dictionary which consists of sorted words.
Searching for Employee Details With Employee Number in an Employee Directory
Searching comprises of following algorithms
Sequential Search or Linear Search
Binary search
Capgemini



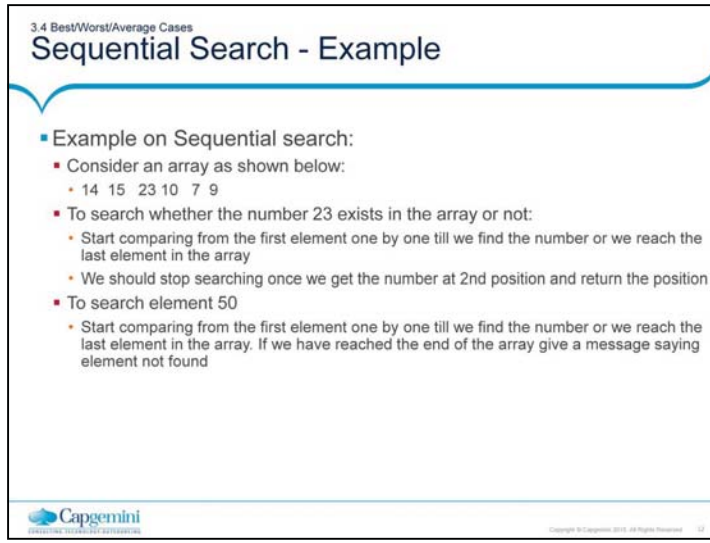
Linear search:

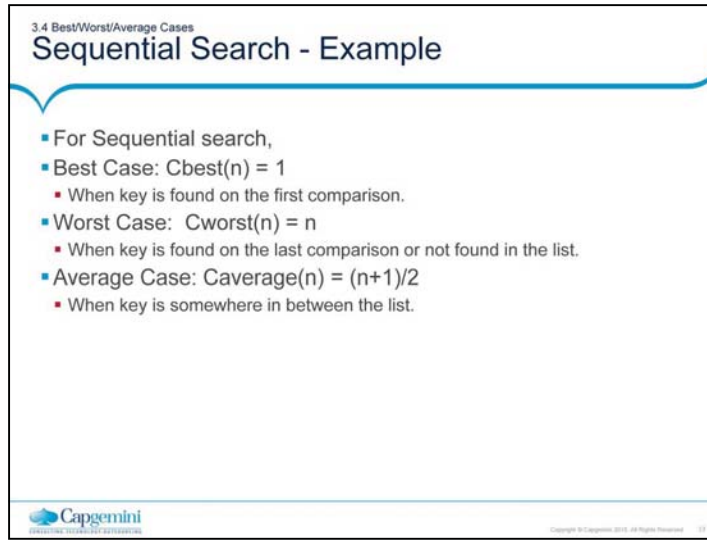
The Linear search technique is normally used when data is not in the sorted order. Linear search is another name for “sequential search”.

If we find the value at first position, then it is “best case” because search requires to do only one comparison.

However, if we find the value at the end, then this is “worst scenario” because we have to search for all the n values sequentially till last value

If we find data at the middle then we require to do only n/2 comparisons.


3.4 Best/Worst/Average Cases
Sequential Search - Example
Example on Sequential search:
Consider an array as shown below:
14 15 23 10 7 9
To search whether the number 23 exists in the array or not:
Start comparing from the first element one by one till we find the number or we reach the last element in the array
We should stop searching once we get the number at 2nd position and return the position
To search element 50
Start comparing from the first element one by one till we find the number or we reach the last element in the array. If we have reached the end of the array give a message saying element not found
Capgemini

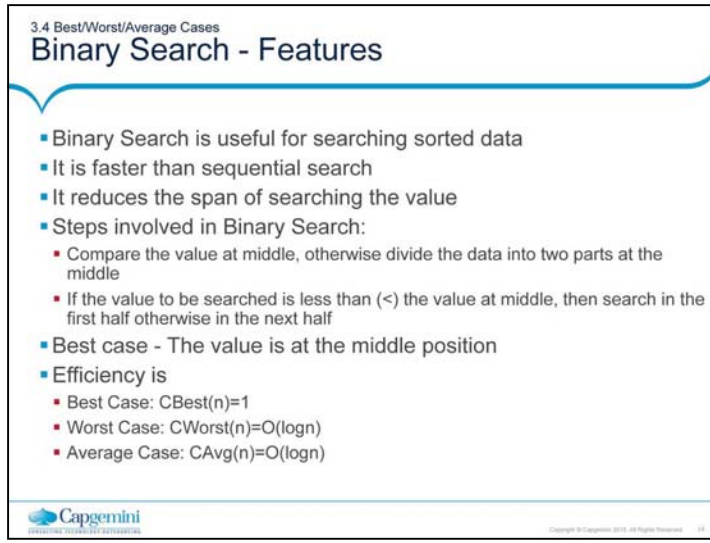


➢For Sequential seacrch,

➢Best Case: $C_{best}(n) = 1$: Here key is found in the fist place of the list. Hence only one comparison is done.

➢Worst Case: $C_{worst}(n) = n$: Worst case scenario for sequential search is, when key is found on the last comparison or not found in the list. So if the list contains n elements the n number of comparisons is done.

➢Average Case: $C_{average}(n) = (n+1)/2$: When key is somewhere in between the list. (n+1)/2 is derived from the mathematical computation with probability concepts.



Binary search:

It is useful to search information from the sorted data. It is faster than “sequential search”.

Steps involved in Binary search:

Compares the value at mid position, otherwise divides the data into two parts at the middle.

If the value to be searched is less than (<) the value at middle, then search in the fist half otherwise in the next half.

Repeat the steps 1 and 2 for the selected half till we get the value to be searched.

Example:

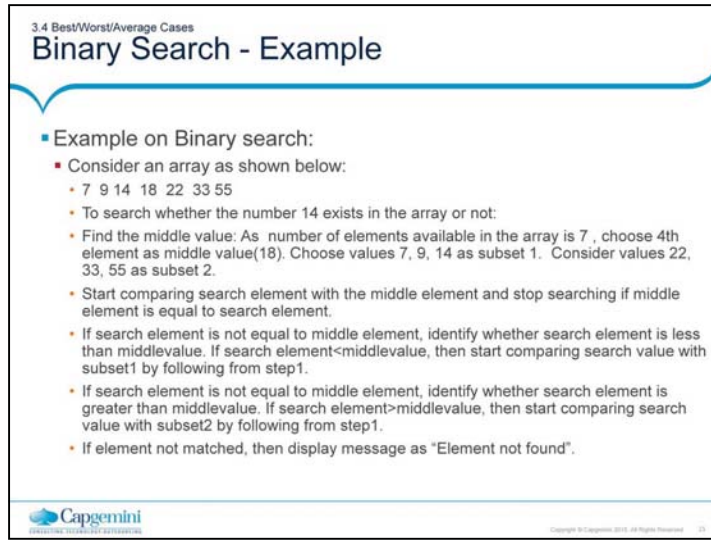
Suppose you are searching for the word ‘psychology’ in the dictionary, while searching if you find the word “Quoits” then we will definitely search in previous pages. In this example too we are reducing span of searching because dictionary is sorted alphabetically. Similar technique is used in binary search.

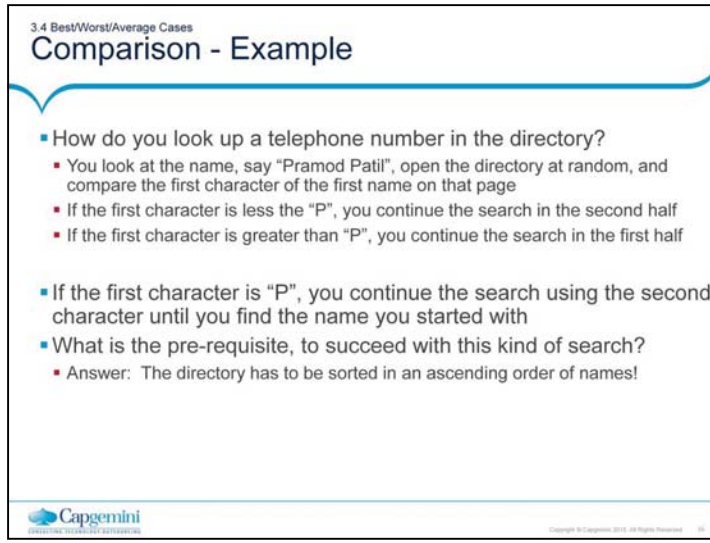
Consider an array of numbers:

14 15 18 24 25 78 82 85 89 92 100

Search whether number 15 is in the array or not.

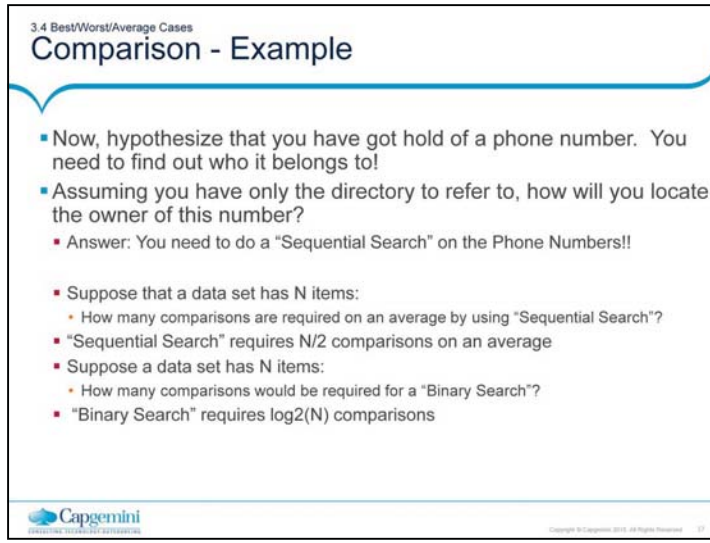
Since the array is in the sorted order, we can use binary search algorithm.

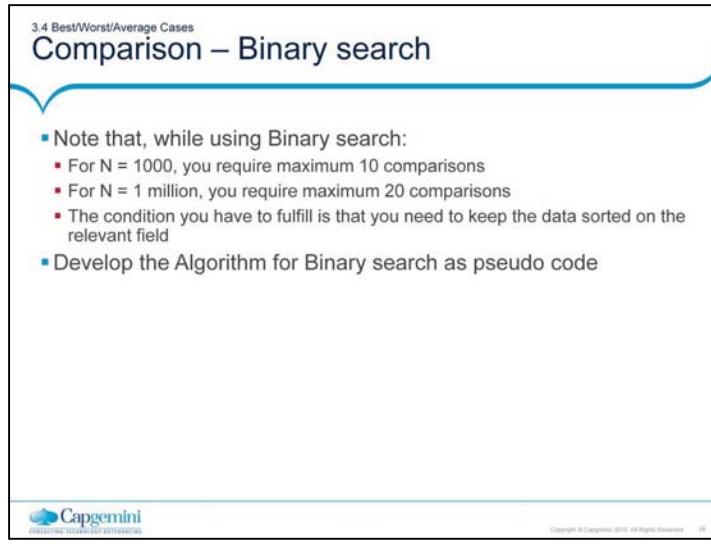

3.4 Best/Worst/Average Cases
Binary Search - Example
Example on Binary search:
Consider an array as shown below:
7 9 14 18 22 33 55
To search whether the number 14 exists in the array or not:
Find the middle value: As number of elements available in the array is 7 , choose 4th element as middle value(18). Choose values 7, 9, 14 as subset 1. Consider values 22, 33, 55 as subset 2.
Start comparing search element with the middle element and stop searching if middle element is equal to search element.
If search element is not equal to middle element, identify whether search element is less than middlevalue. If search element<middlevalue, then start comparing search value with subset1 by following from step1.
If search element is not equal to middle element, identify whether search element is greater than middlevalue. If search element>middlevalue, then start comparing search value with subset2 by following from step1.
If element not matched, then display message as "Element not found".
Capgemini

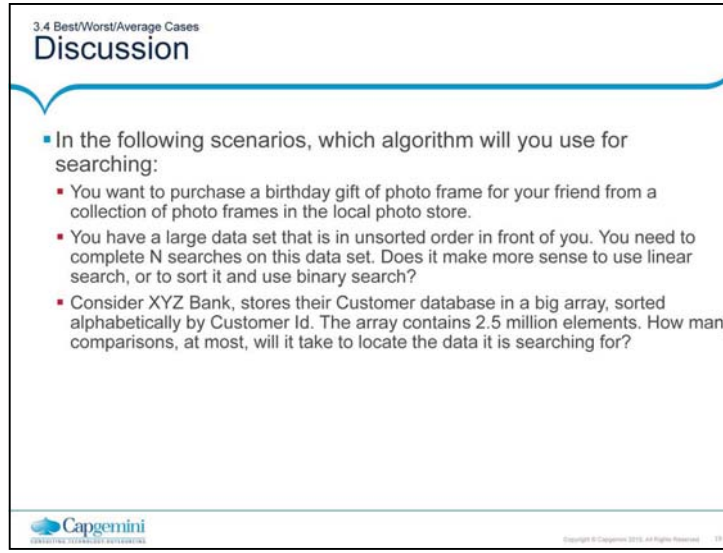


Steps:
The number at the middle is 78 != 15 but 15 < 78, hence search in the first half.
The value at the middle is 18 !=15 but 15 < 18, hence search it in the first half.
The value at the middle is 15, which is equal to the number we are searching for.
Display message, Number found at 2nd position or return the position.

Search whether number 10 is in the array or not.
Steps:
The number at the middle is 78 != 10 but 10 < 78, hence search in the first half.
The value at the middle is 18 !=10 but 10 < 18, hence search it in the first half.
The value at the middle is 15 !=10, hence search it in the first half.
The value at the middle is 14 !=10, At this point we have reached the first element's position beyond which we cannot move.
Display message, Number not found


3.4 Best/Worst/Average Cases
Comparison - Example
Now, hypothesize that you have got hold of a phone number. You need to find out who it belongs to!
Assuming you have only the directory to refer to, how will you locate the owner of this number?
Answer: You need to do a "Sequential Search" on the Phone Numbers!!
Suppose that a data set has N items:
How many comparisons are required on an average by using "Sequential Search"?
"Sequential Search" requires N/2 comparisons on an average
Suppose a data set has N items:
How many comparisons would be required for a "Binary Search"?
"Binary Search" requires log2(N) comparisons
Capgemini


3.4 Best/Worst/Average Cases
Comparison – Binary search
Note that, while using Binary search:
For N = 1000, you require maximum 10 comparisons
For N = 1 million, you require maximum 20 comparisons
The condition you have to fulfill is that you need to keep the data sorted on the relevant field
Develop the Algorithm for Binary search as pseudo code
Capgemini



Answer for Scenario 1:

• Sequential Search need to be used, Since the frames are not located in any order.

• Starting at the first frame, examine each frame (Without skipping) until you find the frame you want.

Answer for Scenario 2:

•It makes more sense to sort it and use binary search.

•Sorting of a data set can be done in ***O(nlogn)*** time.

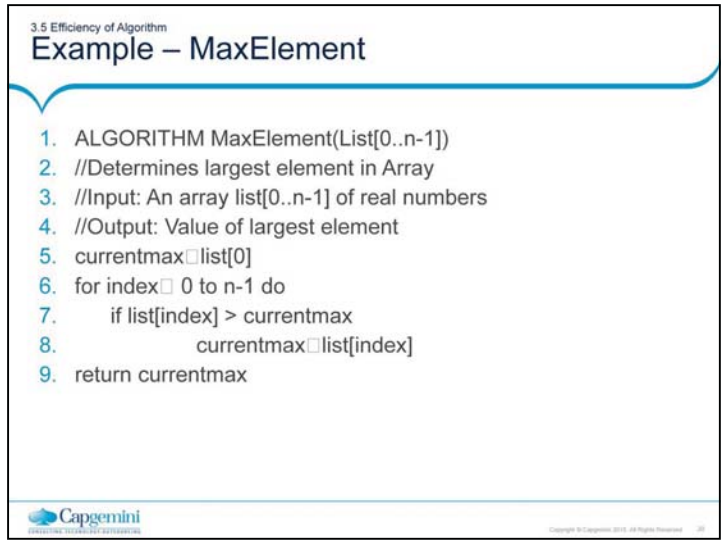
•To do n linear searches will take n*O(n) = = O(n 2) time.

•To sort it and do n binary searches will take O(nlogn) + n*O(logn) = = O(nlogn) time

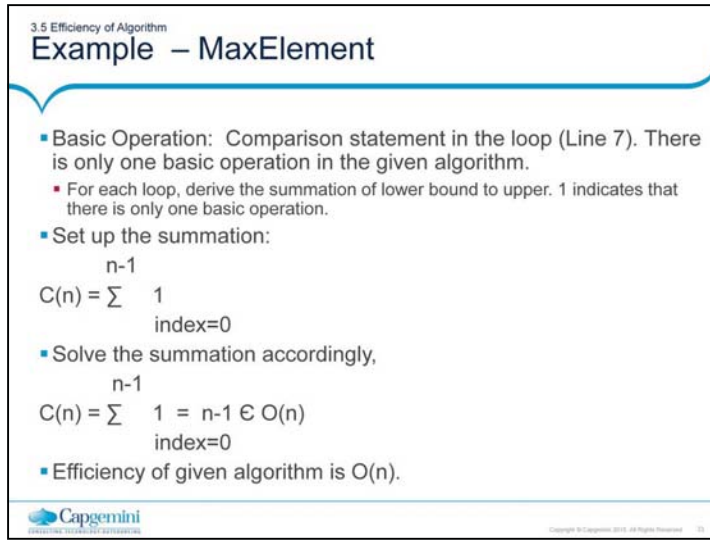
Answer for Scenario 3:

• Since the array is sorted, Binary search algorithm needs to be used.

• It will take at most 22 comparisons; ***ceil*** **(*log*(2, 500, 000)) = = 22** .



Consider the above MaxElement algorithm. It also depicts proper way of writing algorithm. It should always start with a proper name and input as a parameter. First comment describes the algorithm. Second comment explains the input and third comment explains the output. Proper indentation must be maintained in the algorithm.



➢How it works?

➢Basic Operation: Comparison statement in the loop.

➢Summation of lower bound to upper bound. For each loop derive the summation. Hence as given algorithm has only one loops, we end up with 1 summation. 1 indicates that there is only one basic operation.

➢Now solving the above expression with proper formula's, we end up with n-1. We are more interested in efficiency class. Hence constants can be ignored and so efficiency is O(n). i.e Linear.

Summation formula used:

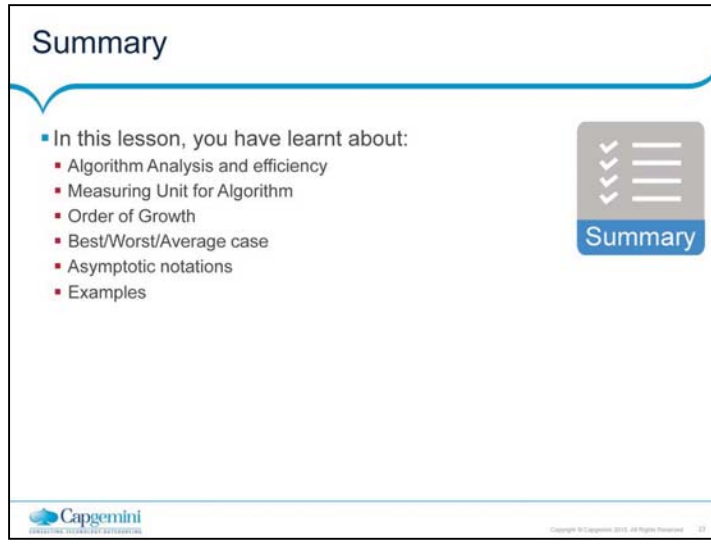
$$\sum_{i=l}^{u} 1 = u-l+1 \quad \text{where } l<=u \text{ are lower and upper integer limits.}$$

Lab

- Analyzing Algorithms Lab 3

Capgemini

Copyright © Capgemini 2015. All Rights Reserved


Summary
In this lesson, you have learnt about:
Algorithm Analysis and efficiency
Measuring Unit for Algorithm
Order of Growth
Best/Worst/Average case
Asymptotic notations
Examples
Summary
Capgemini

Review Question

- Question 1: If efficiency of Algorithm X is $O(n^2)$ and Algorithm Y is O(n logn), Which is more efficient ?
  a. Algorithm X
  b. Algorithm Y
  c. Both are same
  d. Depends on input type.
- Question 2: The time factor when determining the efficiency of algorithm is measured by
  a. counting micro seconds
  b. counting the number of key operations
  c. counting the number of statements
  d. counting the kilobytes of algorithm

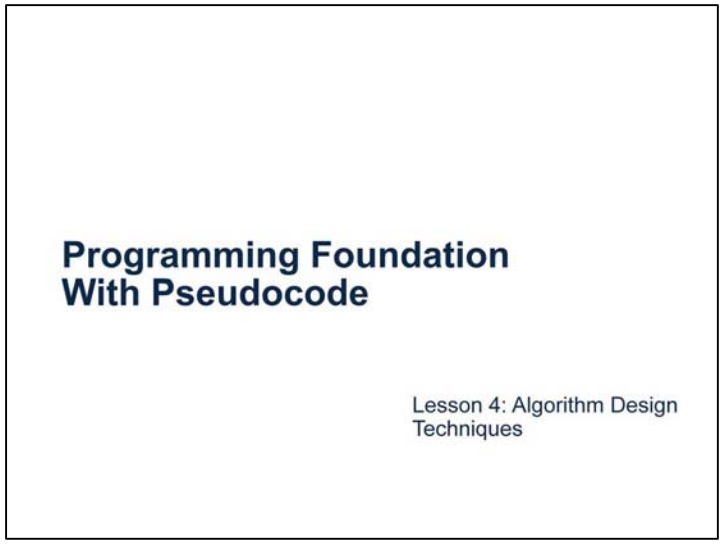
Review Question

- Question 3: Two main measures for the efficiency of an algorithm are
  - a. Processor and memory
  - b. Complexity and capacity
  - c. Time and space
  - d. Data and space

- Question 4: If an algorithm's running time depends on not only the input size n, but also depends on other parameter, then we need to find Best, Worst and Average case efficiencies.
  - a. True
  - b. False

Capgemini

Copyright © Capgemini 2015. All Rights Reserved

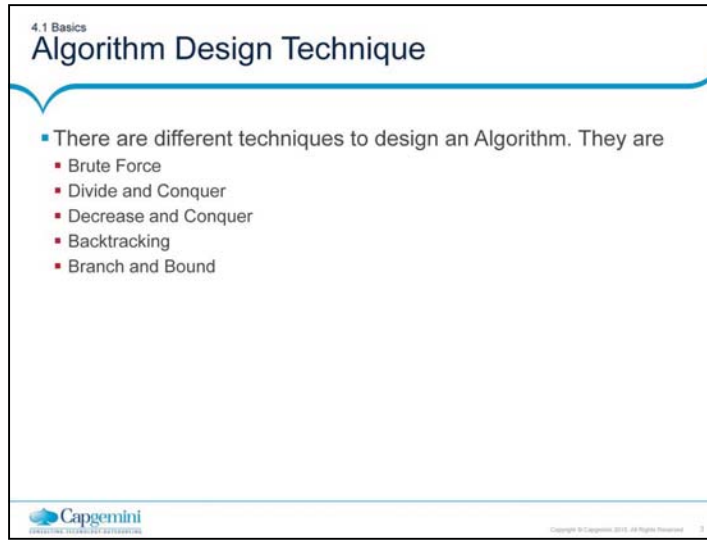

Programming Foundation
With Pseudocode
Lesson 4: Algorithm Design
Techniques

Lesson Objectives

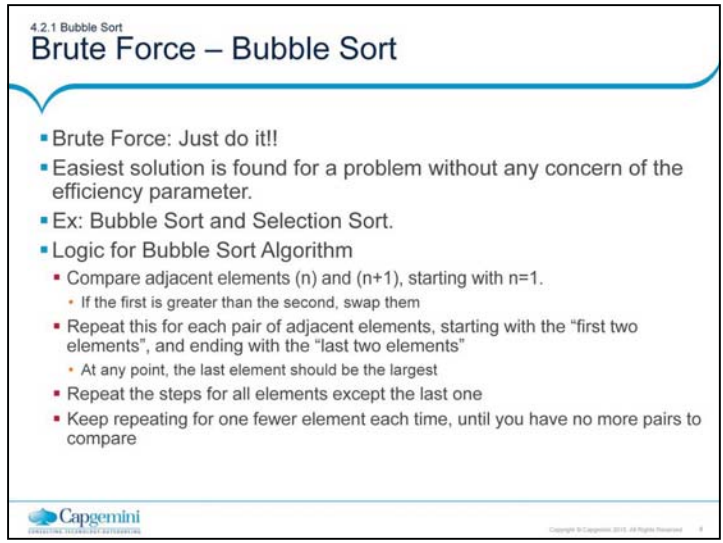
- To enhance the logic building by designing algorithms efficiently for the given problem.
- To understand and compare different Sorting methods under different design techniques:
  - Bubble Sort
  - Merge Sort
  - Insertion Sort
- To understand and compare different algorithms designed for a given problem under different design techniques:

Capgemini

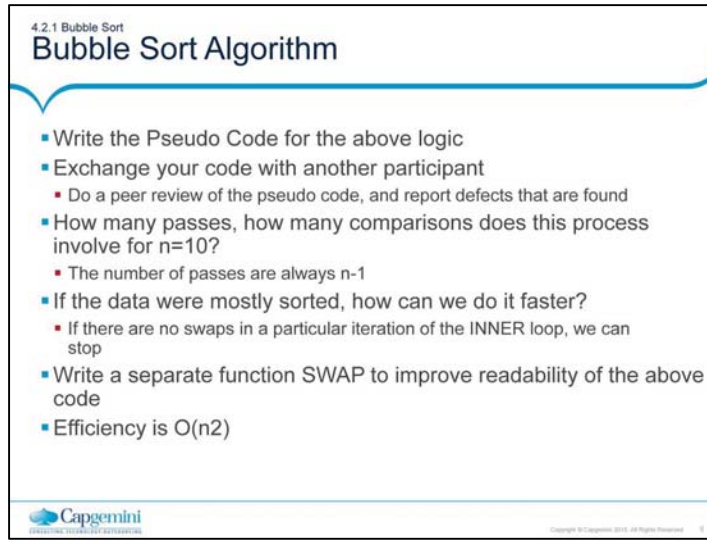
Copyright © Capgemini 2015. All Rights Reserved



Algorithm Design Techniques provides various different approaches to solve a given problem. Various design techniques are available as listed in the above slide.
Brute Force: This is “Just do it” type. Easiest solution is found for a problem without any concern of the efficiency parameter.
Divide and Conquer: Here the task is divided, solution is found for the smallest unit and the solutions are conquered to derive the final result.
Decrease and conquer: It is based on exploiting the relationship between a solution to a given instance of a problem and a solution to a smaller instance of the same problem.
Backtracking: Backtracking is a technique used to solve problems with a large search space, by systematically trying and eliminating possibilities.



Brute force is a straightforward approach to solving a problem, usually directly based on the problem statement and definitions of the concepts involved. "Just do it!" would be another way to describe the prescription of the brute-force approach. Brute Force solutions will not have concerns for the efficiency parameters.
Example for Bubble sort is explained in detail in the above slides.

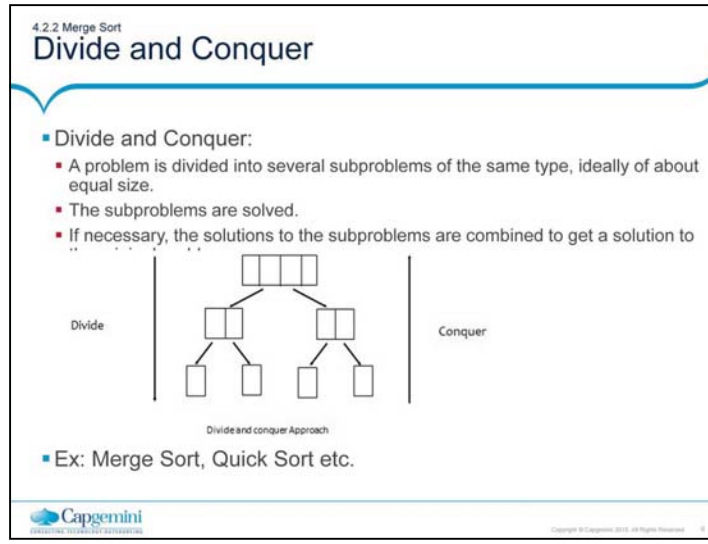


ALGORITHM *BubbleSort(List[0..n-1])*
//Sorts a given array using bubble sort
//Input: An array list[0..n-1] of orderable elements
//Output: Array list[0..n-1] sorted in ascending order
for itr← 0 to n-2 do
for index←0 to n-2-itr do
if list[index+1] < list[index] swap list[index] and list[index+1]

Efficiency of the given algorithm is $\Theta(n^2)$.

$$\sum_{itr=0}^{n-2} \sum_{index=0}^{n-2-itr} 1 = \sum_{itr=0}^{n-2} [(n-2-itr) - 0 + 1]$$

$$= \sum_{itr=0}^{n-2} (n-1-itr) = n(n-1)/2 \in \mathbf{O(n^2)}$$

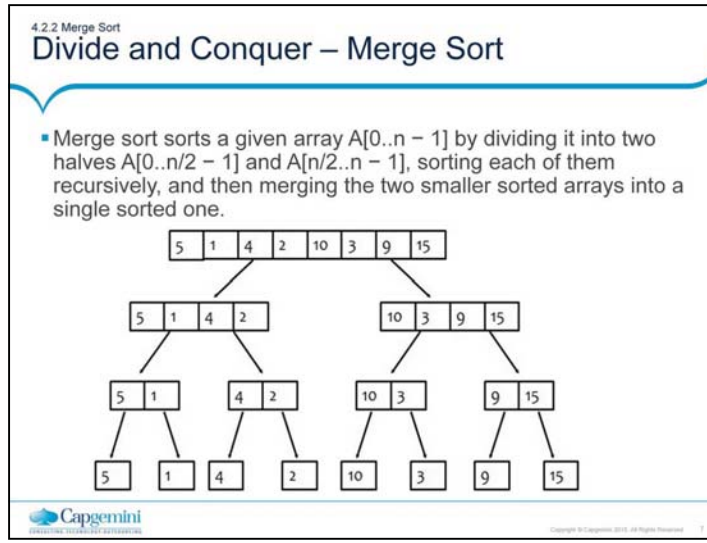


The *divide-and-conquer* technique involves solving a particular computational problem by dividing it into one or more subproblems of smaller size, recursively solving each subproblem, and then "merging" or "marrying" the solutions to the subproblem(s) to produce a solution to the original problem.

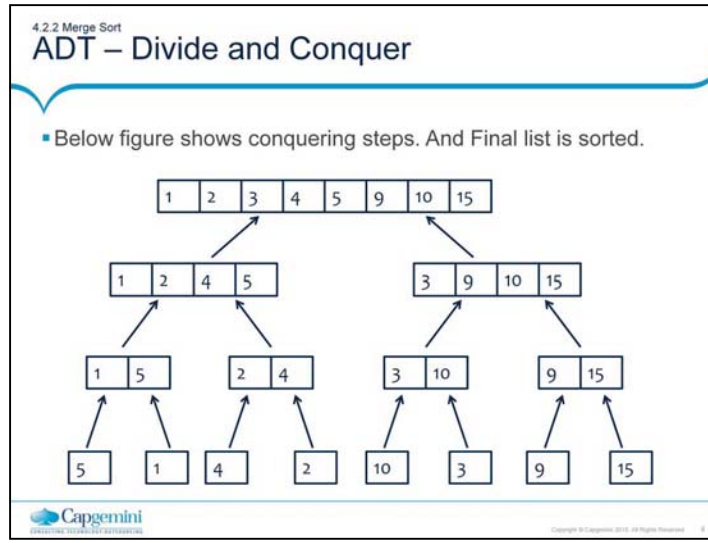
Quite a few very efficient algorithms are specific implementations of this general strategy. Divide and conquer technique work according to the following steps,

- A problem is divided into several subproblems of the same type, ideally of about equal size.
- The subproblems are solved.
- If necessary, the solutions to the subproblems are combined to get a solution to the original problem.

Note: Not every divide-and-conquer algorithm is necessarily more efficient than even a brute-force solution.



- Consider above given input, 5,1,4,2,10,3,9,15.
- First the task is divided in smallest unit as shown below.

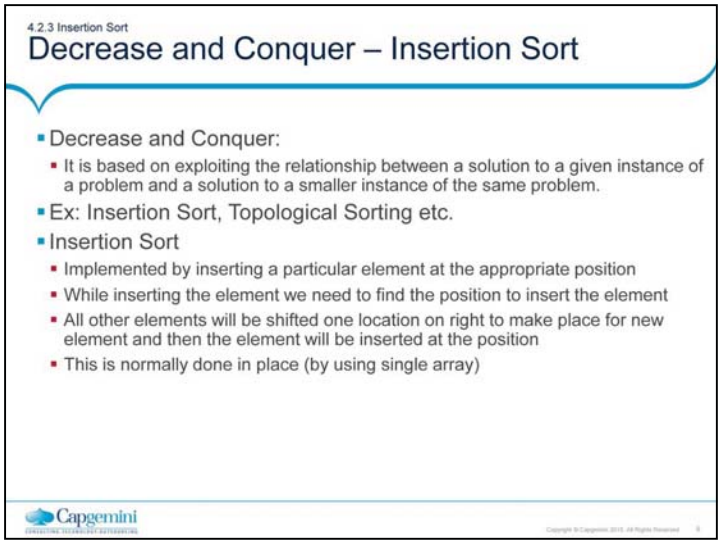


Efficiency of Merge Sort

- $C_{Best}(n)=O(n\log n)$
- $C_{Worst}(n)=O(n\log n)$
- $C_{Avg}(n)=O(\log n)$

Efficiency of Quick Sort

- $C_{Best}(n)=O(n\log n)$
- $C_{Worst}(n)=O(n^2)$ [Depends on the pivot element chosen]
- $C_{Avg}(n)=O(1.38n\log n)$

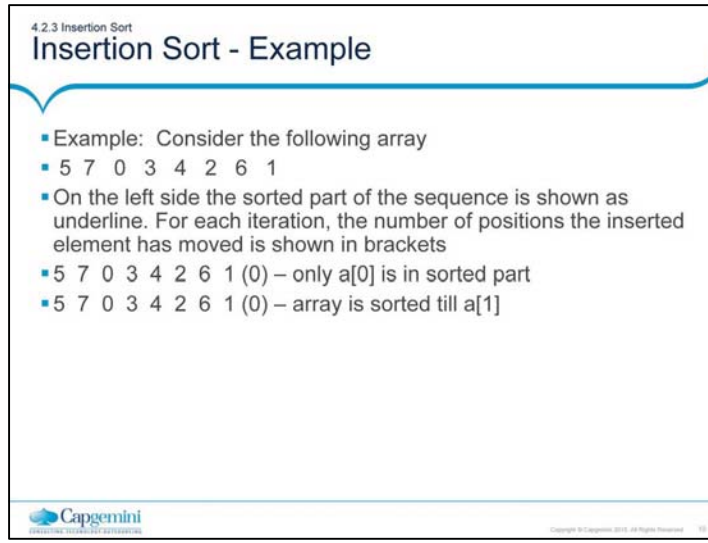


Decrease and Conquer:

It is based on exploiting the relationship between a solution to a given instance of a problem and a solution to a smaller instance of the same problem.

There are three major variations of decrease-and-conquer:

- decrease by a constant : In the ***decrease-by-a-constant*** variation, the size of an instance is reduced by the same constant on each iteration of the algorithm. Typically, this constant is equal to one
- decrease by a constant factor: The ***decrease-by-a-constant-factor*** technique suggests reducing a problem instance by the same constant factor on each iteration of the algorithm.
- variable size decrease: The size-reduction pattern varies from one iteration of an algorithm to another.



Example: Consider the following array a.
5 7 0 3 4 2 6 1.
On the left side the sorted part of the sequence is shown as underline. For each iteration, the number of positions the inserted element has moved is shown in brackets.
5 7 0 3 4 2 6 1.(0) – Initially only a[0] element is sorted

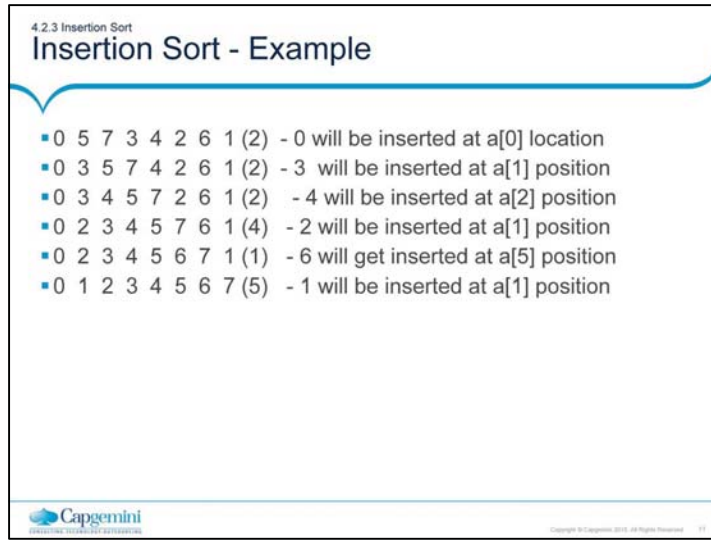
Consider element at a[1] 7 > 5 , So no change will be there and sorted part will increase from a[0] to a[1]
5 7 0 3 4 2 6 1 (0)

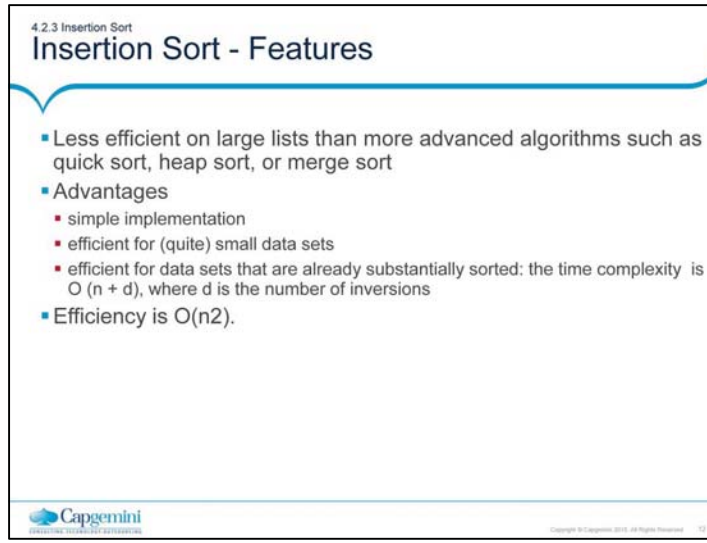
Consider element at a[2] we will copy it in index variable and check 7 >0, 5>0 hence 5 and 7 will get shifted to one location on right and put 0 at a[0] location
Now the size of sorted part is 3
0 5 7 3 4 2 6 1 (2)

Similarly we will place element 3 from a[3] to a [1] position and array will be as follows
0 3 5 7 4 2 6 1 (2)

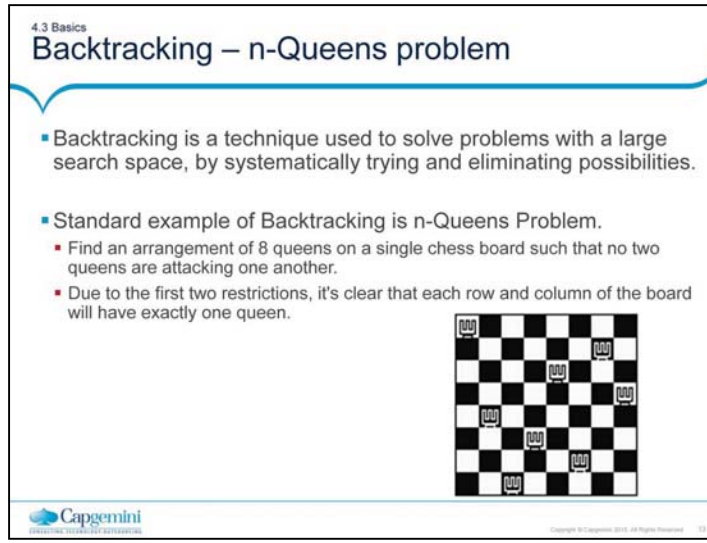
In further iterations array will get sorted as follows

0 3 4 5 7 2 6 1 (2) - 4 will be inserted at a[2] position
0 2 3 4 5 7 6 1 (4) - 2 will be inserted at a[2] position
0 2 3 4 5 6 7 1 (1) - 6 will get inserted at a[4] position
0 1 2 3 4 5 6 7 (5) - 1 will be inserted at a[1] position

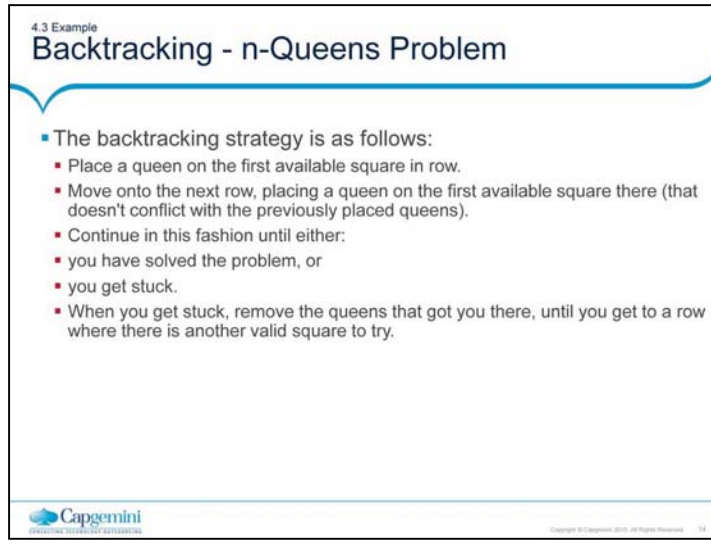

4.2.3 Insertion Sort
Insertion Sort - Example
0 5 7 3 4 2 6 1 (2) - 0 will be inserted at a[0] location
0 3 5 7 4 2 6 1 (2) - 3 will be inserted at a[1] position
0 3 4 5 7 2 6 1 (2) - 4 will be inserted at a[2] position
0 2 3 4 5 7 6 1 (4) - 2 will be inserted at a[1] position
0 2 3 4 5 6 7 1 (1) - 6 will get inserted at a[5] position
0 1 2 3 4 5 6 7 (5) - 1 will be inserted at a[1] position
Capgemini

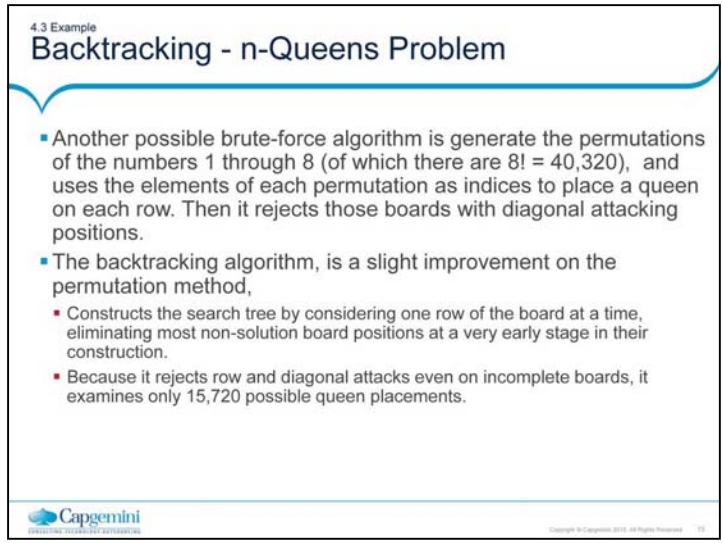


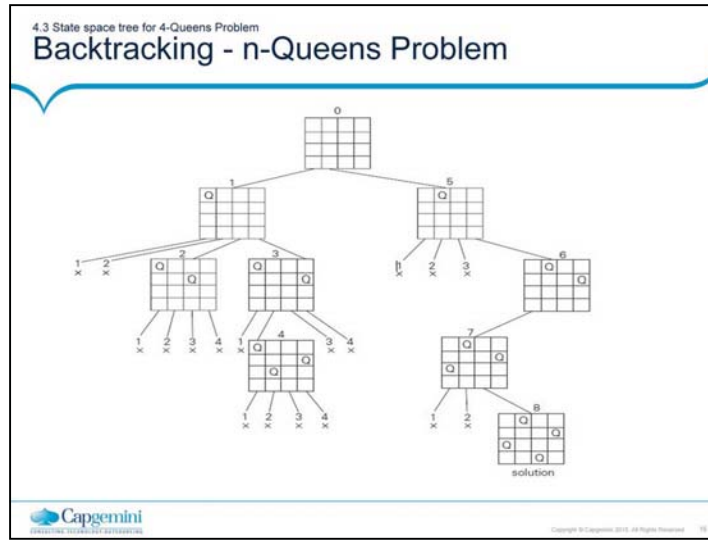
Efficiency of insertion sort is $O(n^2)$.
More efficient sorting algorithms are merge sort, heap sort and quick sort. Their efficiency falls under nlogn class but they are restricted with their own conditions.
Hence it depends on the input type to decide which algorithm must be used for a given application.



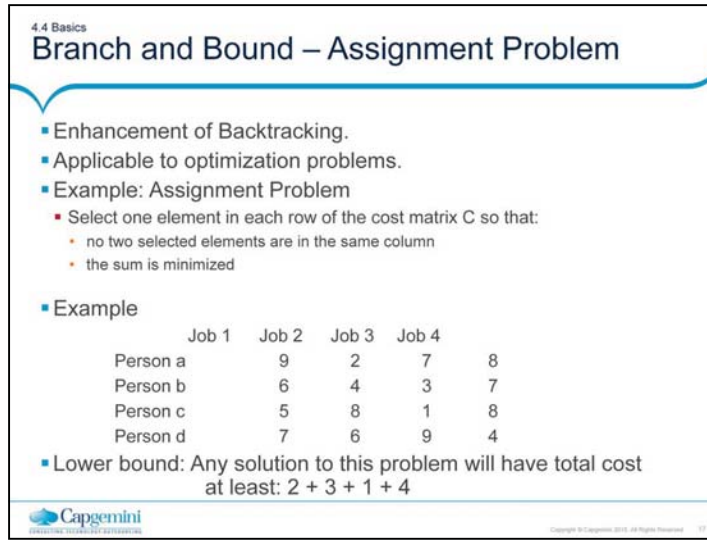
Both backtracking and branch-and-bound are based on the construction of a ***state-space tree*** whose nodes reflect specific choices made for a solution's components. Both techniques terminate a node as soon as it can be guaranteed that no solution to the problem can be obtained by considering choices that correspond to the node's descendants. Difference is that Branch and Bound is only applied to the optimization problems where as Backtracking does not have any such constraints. The other difference is that the order in which nodes of the state-space tree are generated. Backtracking uses DFS and Branch and bound uses various rules, one of them is best-first rule.


4.3 Example
Backtracking - n-Queens Problem
The backtracking strategy is as follows:
Place a queen on the first available square in row.
Move onto the next row, placing a queen on the first available square there (that doesn't conflict with the previously placed queens).
Continue in this fashion until either:
you have solved the problem, or
you get stuck.
When you get stuck, remove the queens that got you there, until you get to a row where there is another valid square to try.
Capgemini


4.3 Example
Backtracking - n-Queens Problem
Another possible brute-force algorithm is generate the permutations of the numbers 1 through 8 (of which there are 8! = 40,320), and uses the elements of each permutation as indices to place a queen on each row. Then it rejects those boards with diagonal attacking positions.
The backtracking algorithm, is a slight improvement on the permutation method,
Constructs the search tree by considering one row of the board at a time, eliminating most non-solution board positions at a very early stage in their construction.
Because it rejects row and diagonal attacks even on incomplete boards, it examines only 15,720 possible queen placements.
Capgemini



Lets place queen 1 in the first possible position of its row, which is in column 1 of row 1. Then we place queen 2, after trying unsuccessfully columns 1 and 2, in the first acceptable position for it, which is square (2, 3), the square in row 2 and column 3. This proves to be a dead end because there is no acceptable position for queen 3. So, the algorithm backtracks and puts queen 2 in the next possible position at (2, 4). Then queen 3 is placed at (3, 2), which proves to be another dead end. The algorithm then backtracks all the way to queen 1 and moves it to (1, 2). Queen 2 then goes to (2, 4), queen 3 to (3, 1), and queen 4 to (4, 3), which is a solution to the problem. This is depicted in the state space tree on the above slide.

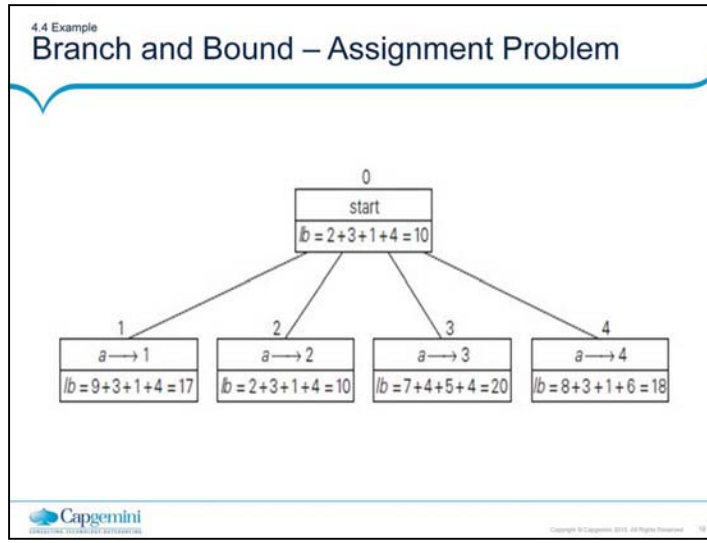


Branch and Bound is applicable for the optimization problem. ***Optimal solution*** is a feasible solution with the best value of the objective function.

we can compare a node's bound value with the value of the best solution seen so far. If the bound value is not better than the value of the best solution seen so far—i.e., not smaller for a minimization problem and not larger for a maximization problem—the node is nonpromising and can be terminated. Indeed, no solution obtained from it can yield a better solution than the one already available. This is the principal idea of the branch-and-bound technique.

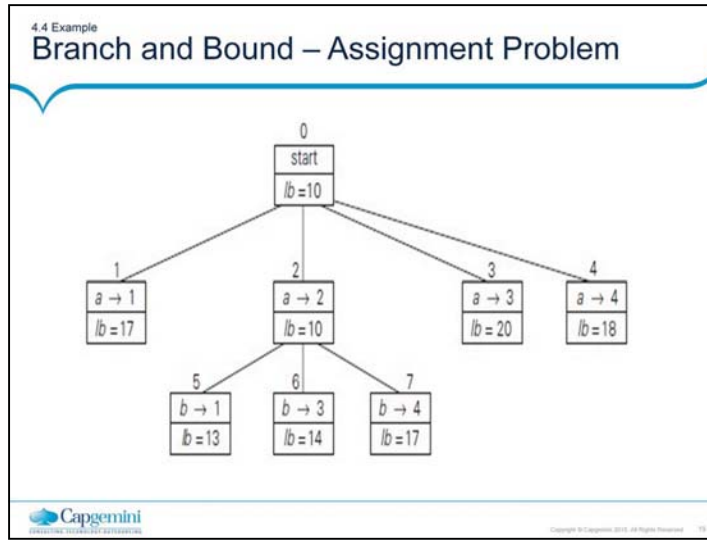
Let us illustrate the branch-and-bound approach by applying it to the problem of assigning n people to n jobs so that the total cost of the assignment is as small as possible. Here each person is assigned to exactly one job and each job is assigned to exactly one person. The cost that would accrue if the ith person is assigned to the jth job is a known quantity $C[i, j]$ for each pair $i, j = 1, 2, \ldots, n$. The problem is to find an assignment with the minimum total cost.Consider the above given instance,

The cost of any solution, including an optimal one, cannot be smaller than the sum of the smallest elements in each of the matrix's rows. For ex: 2 + 3+ 1+ 4 = 10. It is the lower bound.

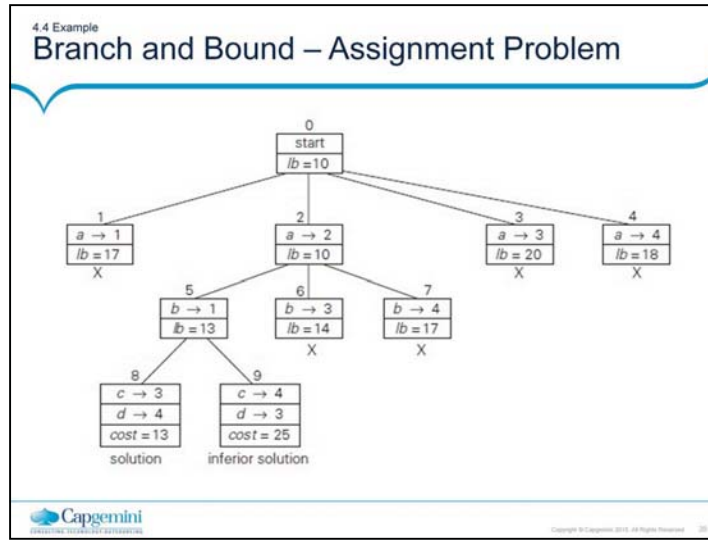


The nodes on the first level of the tree correspond to selections of an element in the first row of the matrix, i.e., a job for person *a.* So we have four live leaves—nodes 1 through 4—that may contain an optimal solution. The most promising of them is node 2 because it has the smallest lowerbound value.

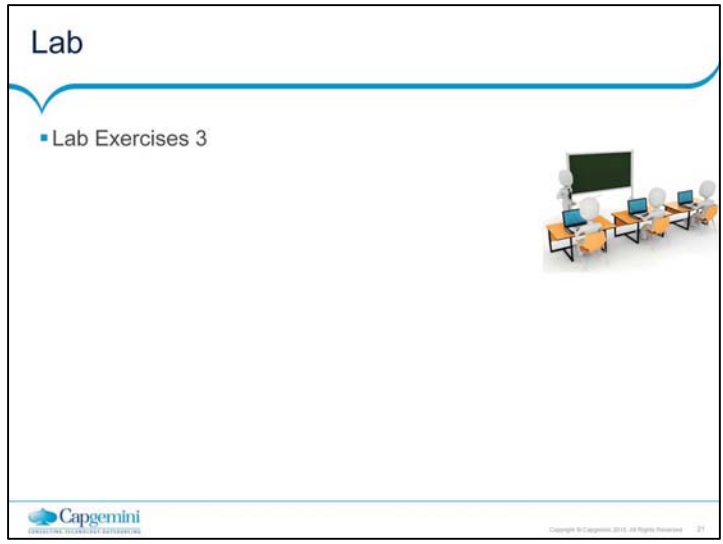
So we have four live leaves—nodes 1 through 4—that may contain an optimal solution. The most promising of them is node 2 because it has the smallest lowerbound value.



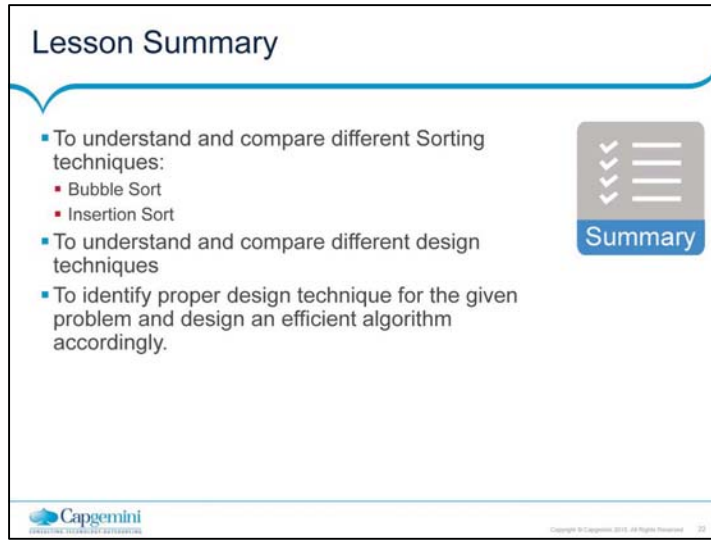
Following our best-first search strategy, we branch out from that node first by considering the three different ways of selecting an element from the
second row and not in the second column—the three different jobs that can be assigned to person *b*



From the six live leaves—nodes 1, 3, 4, 5, 6, and 7 (which may contain an optimal solution) , again node with the smallest lower bound is chosen, node 5. First, we consider selecting the third column's element from *c*'s row (i.e., assigning person *c* to job 3); this leaves us with no choice but to select the element from the fourth column of *d*'s row (assigning person *d* to job 4). This yields leaf 8 which corresponds to the feasible solution {*a*→2, *b*→1, *c*→3, *d* →4} with the total cost of 13. Its sibling, node 9, corresponds to the feasible solution {*a*→2, *b*→1, *c*→4, *d* →3} with the total cost of 25. Since its cost is larger than the cost of the solution represented by leaf 8, node 9 is simply terminated. (Of course, if its cost were smaller than 13, we would have to replace the information about the best solution seen so far with the data provided by this node.) Now, as we inspect each of the live leaves of the last state-space tree—nodes 1, 3, 4, 6, and 7 we discover that their lower-bound values are not smaller than 13, the value of the best selection seen so far (leaf 8). Hence, we terminate all of them and recognize the solution represented by leaf 8 as the optimal solution to the problem.



Write a pseudocode to test the concept of all sorting and searching problems.


Lesson Summary
To understand and compare different Sorting techniques:
Bubble Sort
Insertion Sort
To understand and compare different design techniques
To identify proper design technique for the given problem and design an efficient algorithm accordingly.
Summary
Capgemini

Review Questions

- Question 1: Which of the following sorting techniques uses swapping of two elements to sort the array:
  - A. Bubble sort
  - B. Quick sort
  - C. Insertion sort
- Question 2: What is the efficiency of bubble sort algorithm?
  - A. O(n)
  - B. O(n2)
  - C. O(nlogn)
  - D. O(logn)
- Question 3: Arranging elements in an ascending or descending order is called as _____.

Capgemini

Copyright © Capgemini 2015. All Rights Reserved

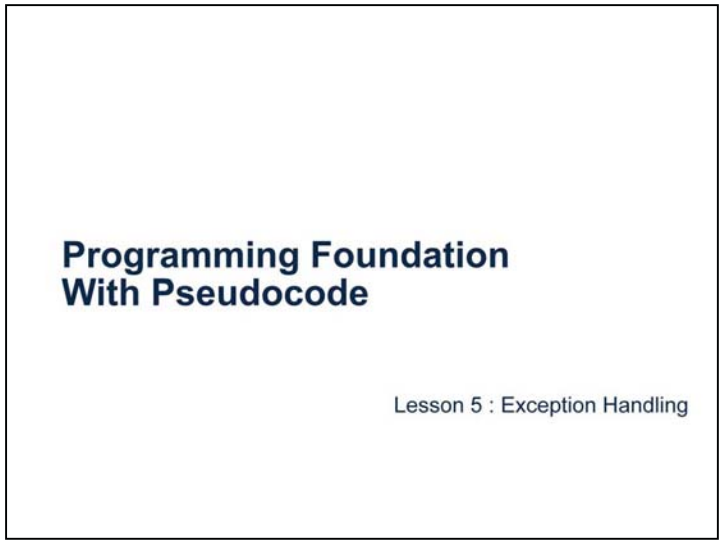
Review Questions: Match the Following

- Question 4:

1. Bubble sort	a. Best case is finding element at the fist position
2. Sequential search	b. Require to use nested loops
3. Binary search	c. Find position before inserting element
4. Insertion sort	d. Best case is finding the element at the middle
	e. Collision

Capgemini

Copyright © Capgemini 2015. All Rights Reserved

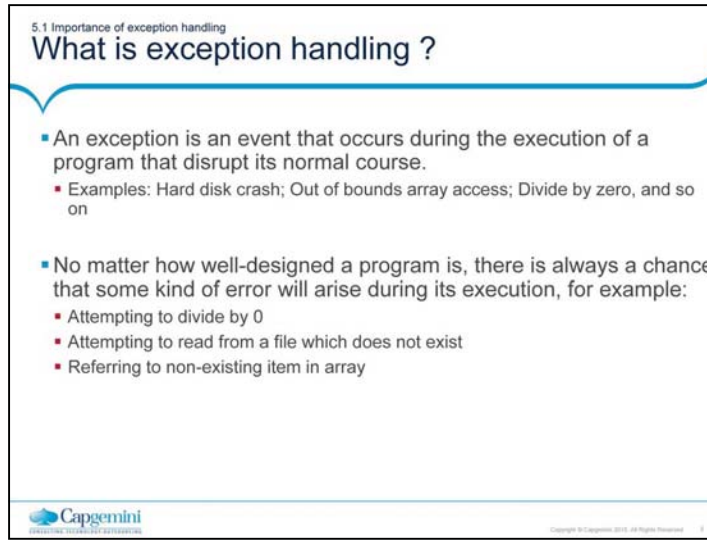

Programming Foundation
With Pseudocode
Lesson 5 : Exception Handling

Lesson Objectives

- After completing this lesson, you will be able to understand the following topics:
  - Importance of exception handling.
  - Guidelines for creating exceptions
  - Exceptional scenarios in ATM system – case study
  - Exception handling case study : Product management system
  - What is Defensive programming
  - Purpose of defensive programming
  - Techniques of defensive programming

Capgemini

Copyright © Capgemini 2015. All Rights Reserved

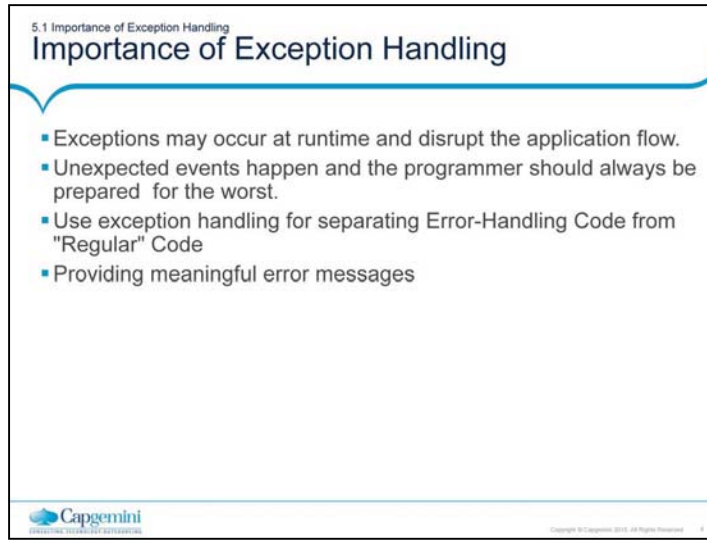


Exception Handling:
An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions. Exceptions are used as a way to report occurrence of some exceptional condition. Exception provides a means of communicating information about errors up through a chain of methods until one of them handles it.

- When an exception occurs, the executing method creates an Exception object and hands it to the runtime system —"throwing an exception"
- The runtime system searches the runtime call stack for a method with an appropriate handler, to catch the exception.

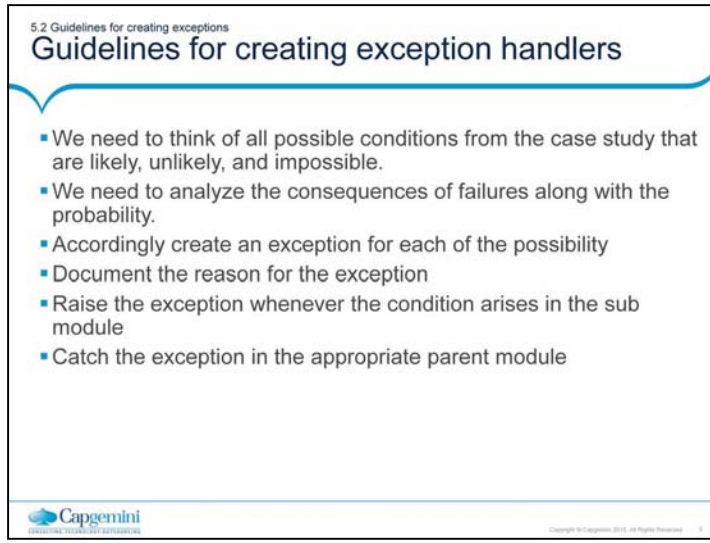
For an example, while booking a ticket in railway reservation system suddenly if the database is down and if an abrupt error page with numerous lines of exception messages displayed on the screen, then user(non-technical person) would be embarrassing and will get annoyed.

Instead of displaying an abrupt error page, if your code catches this database down scenario and displays a graceful message on screen saying "We are currently experiencing some difficulties in our system. Kindly try after some time. Thank you" , then it would be a better message for the end users.



Importance of Exception Handling

- Once an exception occurs at runtime, the program execution will be stopped immediately and an exception message gets displayed which may not be understandable by all the end users.
- Whenever an unexpected input is passed to an application by the end users, program may get crashed. So in order to create an application which should be sustainable at any scenarios, exception handling is mandatory to be taken care.
- Exceptions provide the means to separate the details of what to do when something out of the ordinary happens from the main logic of a program. Enclose the error handling code separately in try block from regular code for making program execution to happen without any disruption even though an exception occurs.



- Pseudocode example without taking care of exception's

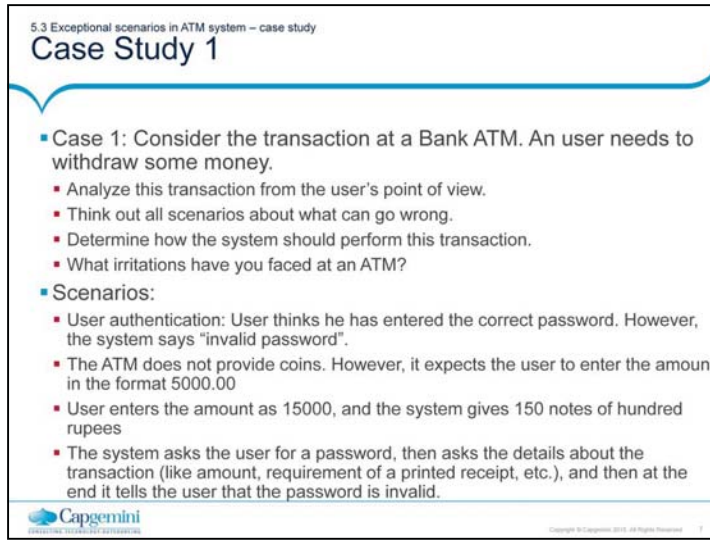
```
SUB readFile
        open the file;
        determine its size;
        read data from the file;
        close the file;
END SUB
```

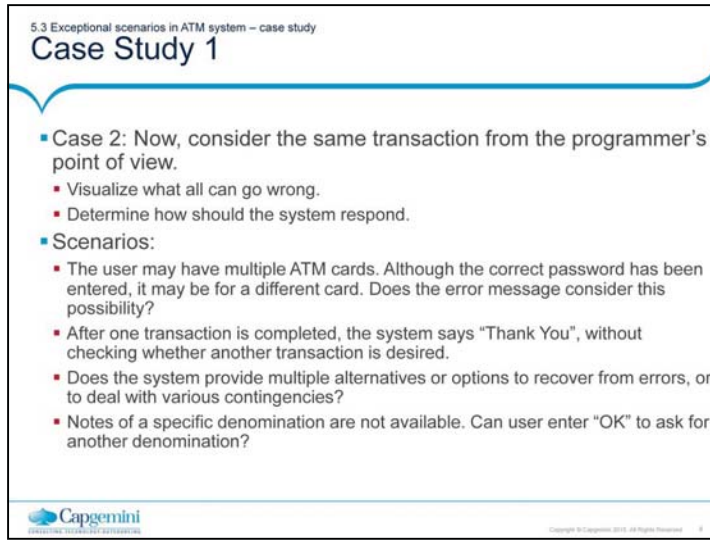
At first glance, this function seems simple enough, but it ignores all the following potential errors.

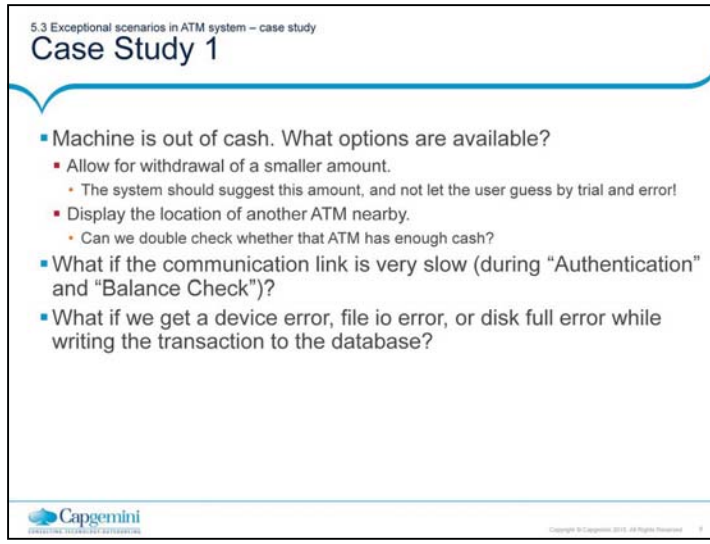
- What happens if the file can't be opened?
- What happens if the length of the file can't be determined?
- What happens if the read fails?
- What happens if the file can't be closed?

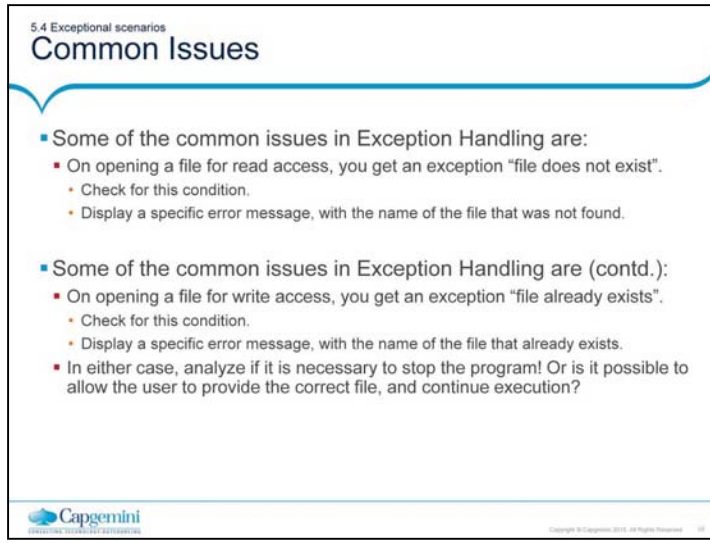
See the below Pseudocode with exception handling for taking care of the possible potential errors.

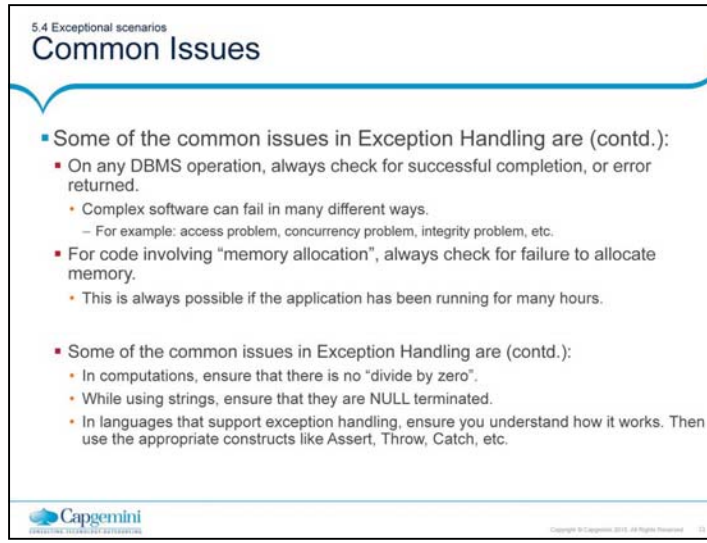
```
SUB readFile
        open the file;
        determine its size;
        read data from the file;
        close the file;
EXCEPTION
        WHEN fileOpenFailed THEN
                doSomething;
        WHEN sizeDeterminationFailed
THEN
                doSomething
        WHEN readFailed THEN
                doSomething;
        WHEN fileCloseFailed THEN
                doSomething
END SUB
```

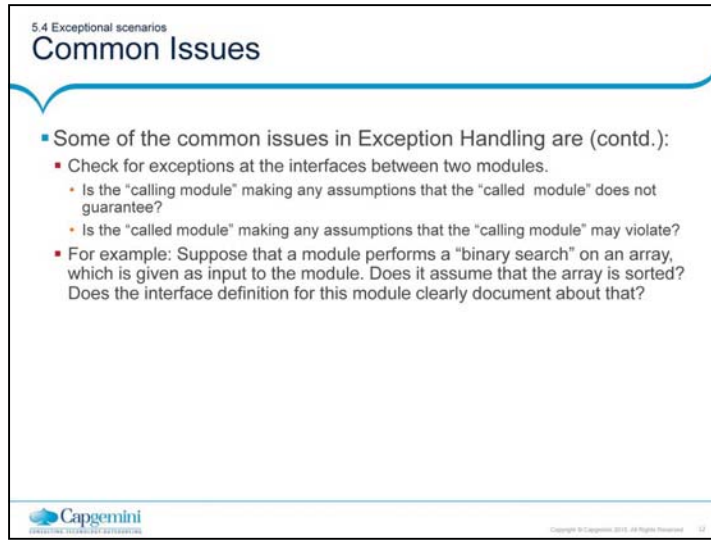

5.3 Exceptional scenarios in ATM system – case study
Case Study 1
Case 1: Consider the transaction at a Bank ATM. An user needs to withdraw some money.
Analyze this transaction from the user's point of view.
Think out all scenarios about what can go wrong.
Determine how the system should perform this transaction.
What irritations have you faced at an ATM?
Scenarios:
User authentication: User thinks he has entered the correct password. However, the system says "invalid password".
The ATM does not provide coins. However, it expects the user to enter the amount in the format 5000.00
User enters the amount as 15000, and the system gives 150 notes of hundred rupees
The system asks the user for a password, then asks the details about the transaction (like amount, requirement of a printed receipt, etc.), and then at the end it tells the user that the password is invalid.
Capgemini

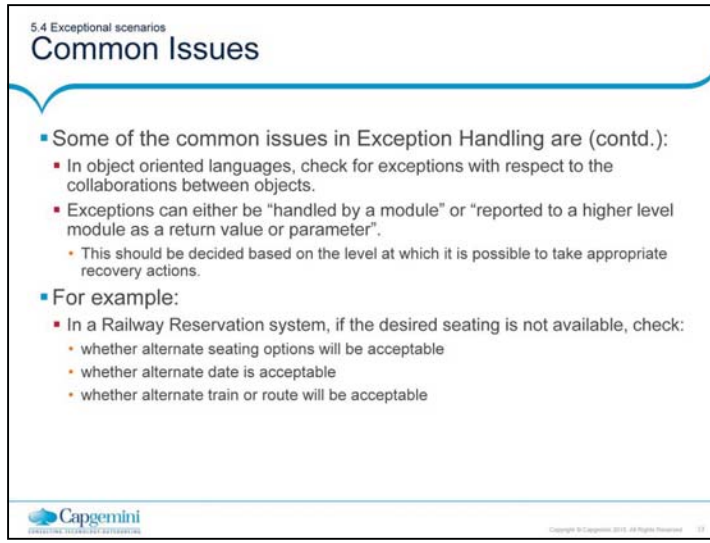

5.3 Exceptional scenarios in ATM system – case study
Case Study 1
Case 2: Now, consider the same transaction from the programmer's point of view.
Visualize what all can go wrong.
Determine how should the system respond.
Scenarios:
The user may have multiple ATM cards. Although the correct password has been entered, it may be for a different card. Does the error message consider this possibility?
After one transaction is completed, the system says "Thank You", without checking whether another transaction is desired.
Does the system provide multiple alternatives or options to recover from errors, or to deal with various contingencies?
Notes of a specific denomination are not available. Can user enter "OK" to ask for another denomination?
Capgemini

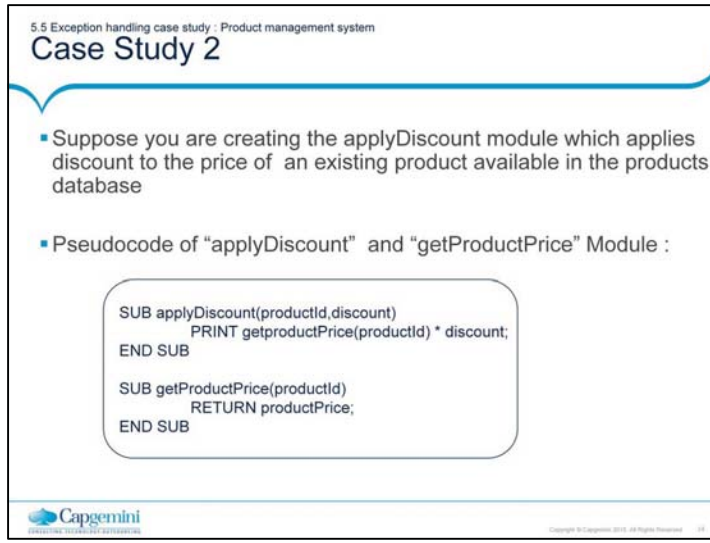

5.3 Exceptional scenarios in ATM system – case study
Case Study 1
Machine is out of cash. What options are available?
Allow for withdrawal of a smaller amount.
The system should suggest this amount, and not let the user guess by trial and error!
Display the location of another ATM nearby.
Can we double check whether that ATM has enough cash?
What if the communication link is very slow (during "Authentication" and "Balance Check")?
What if we get a device error, file io error, or disk full error while writing the transaction to the database?
Capgemini


5.4 Exceptional scenarios
Common Issues
Some of the common issues in Exception Handling are:
On opening a file for read access, you get an exception "file does not exist".
Check for this condition.
Display a specific error message, with the name of the file that was not found.
Some of the common issues in Exception Handling are (contd.):
On opening a file for write access, you get an exception "file already exists".
Check for this condition.
Display a specific error message, with the name of the file that already exists.
In either case, analyze if it is necessary to stop the program! Or is it possible to allow the user to provide the correct file, and continue execution?
Capgemini


5.4 Exceptional scenarios
Common Issues
Some of the common issues in Exception Handling are (contd.):
On any DBMS operation, always check for successful completion, or error returned.
Complex software can fail in many different ways.
For example: access problem, concurrency problem, integrity problem, etc.
For code involving "memory allocation", always check for failure to allocate memory.
This is always possible if the application has been running for many hours.
Some of the common issues in Exception Handling are (contd.):
In computations, ensure that there is no "divide by zero".
While using strings, ensure that they are NULL terminated.
In languages that support exception handling, ensure you understand how it works. Then use the appropriate constructs like Assert, Throw, Catch, etc.
Capgemini

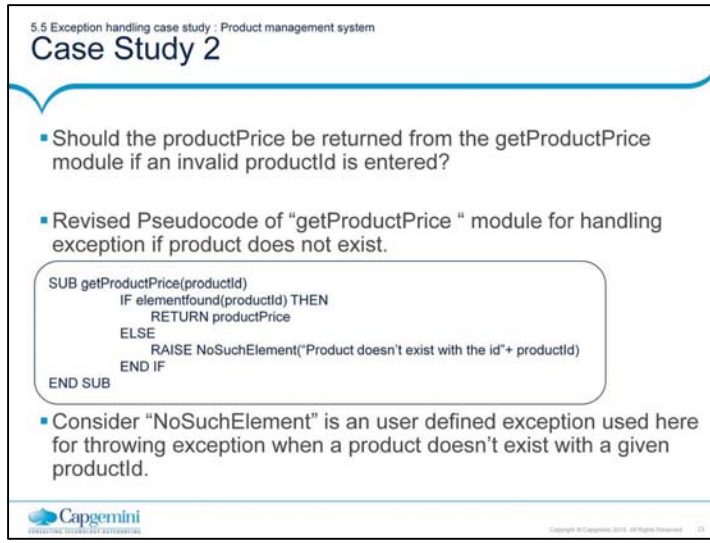

5.4 Exceptional scenarios
Common Issues
Some of the common issues in Exception Handling are (contd.):
Check for exceptions at the interfaces between two modules.
Is the "calling module" making any assumptions that the "called module" does not guarantee?
Is the "called module" making any assumptions that the "calling module" may violate?
For example: Suppose that a module performs a "binary search" on an array, which is given as input to the module. Does it assume that the array is sorted? Does the interface definition for this module clearly document about that?
Capgemini


5.4 Exceptional scenarios
Common Issues
Some of the common issues in Exception Handling are (contd.):
In object oriented languages, check for exceptions with respect to the collaborations between objects.
Exceptions can either be "handled by a module" or "reported to a higher level module as a return value or parameter".
This should be decided based on the level at which it is possible to take appropriate recovery actions.
For example:
In a Railway Reservation system, if the desired seating is not available, check:
whether alternate seating options will be acceptable
whether alternate date is acceptable
whether alternate train or route will be acceptable
Capgemini



applyDiscount module is used to apply discount on a productprice for a particular product.

The applyDiscount module includes a call to the getproductPrice module which searches for the required product based on the productId and return the price of a product



Description of the pseudocode given in slide

If the product id supplied to the productPrice module does not exist in the database then an exception, "Product does not exist along with productId", should be raised from the getproductPrice module

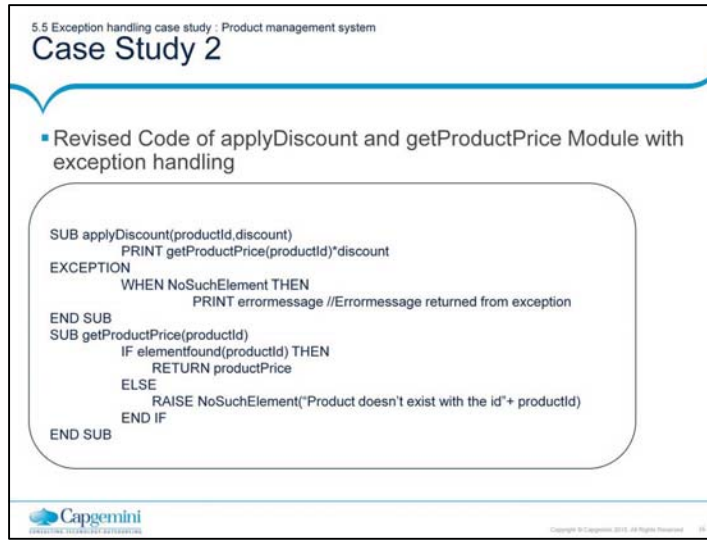
The applyDiscount module should catch and handle the exception appropriately

Rasie is a keyword which can be used to throw an exception.

Syntax: RAISE exceptionname(message);

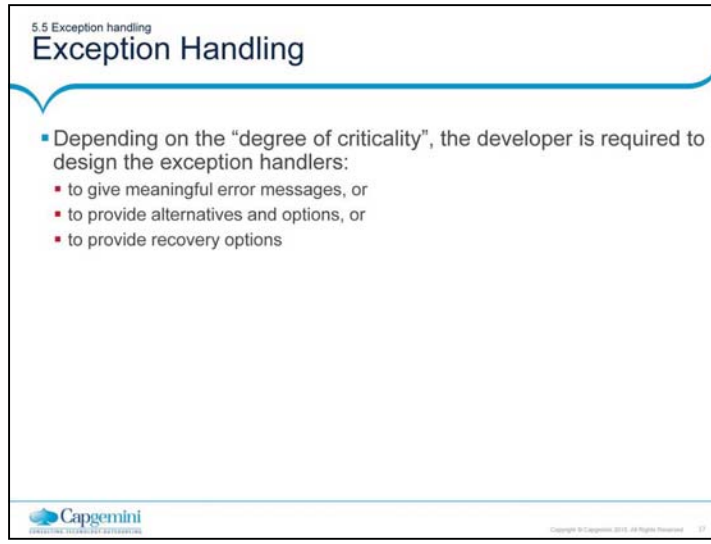
Once if an exception raised from a module, then catch the exception and display the exception message while invoking the module from which the exception is thrown.

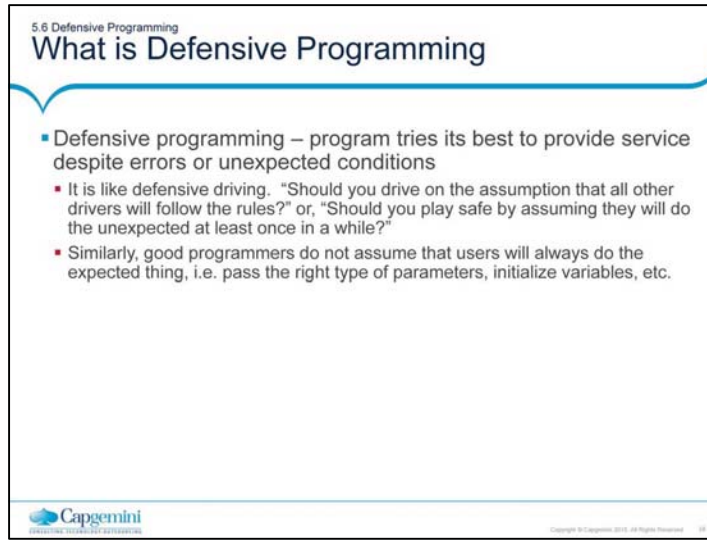
Exception messages should be more meaningful along with the entered productId.



EXCEPTION section will be executed whenever exception occurs. Using WHEN section exception can be handled.

If the given productId exists in database, then updated price will be returned else "Product doesn't exist with the given productId " message will be printed.


5.5 Exception handling
Exception Handling
Depending on the “degree of criticality”, the developer is required to design the exception handlers:
to give meaningful error messages, or
to provide alternatives and options, or
to provide recovery options
Capgemini



Defensive programming enables us to detect minor problems early on, rather than get bitten by them later when they've escalated into major disasters.

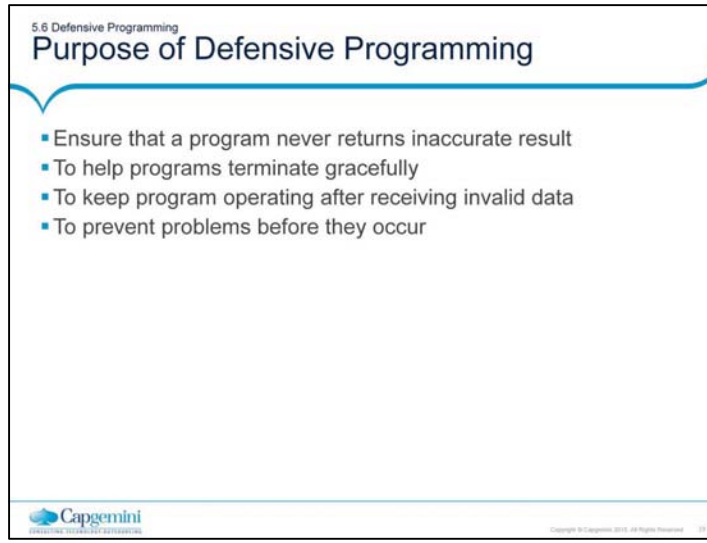
Defensive programming is a method of prevention, rather than a form of cure. Compare this to debugging—the act of removing bugs after they've bitten. Debugging is all about finding a cure.

Defensive Programming is not

Error checking

Testing

Debugging



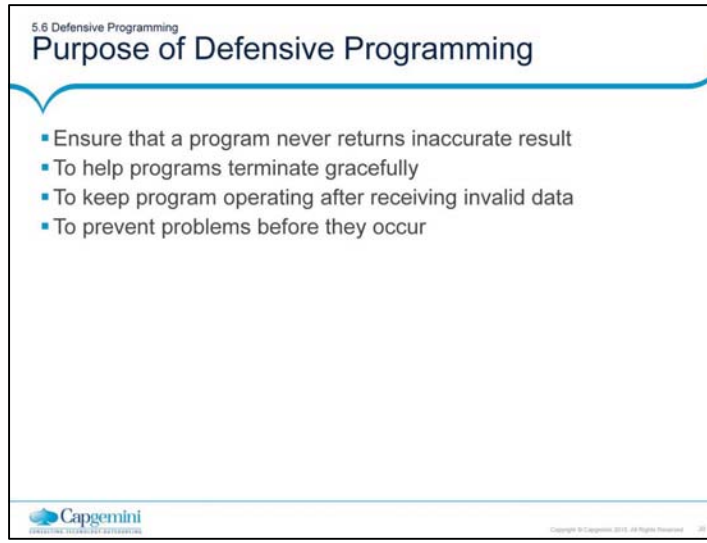
Purpose of Defensive Programming

Ensure that a program never returns inaccurate result even though valid data is passed.

Abnormal termination of the program will be avoided

Even though, invalid data is passed instead of abnormal termination of program execution, meaningful error messages has to be displayed.

Thinking all the possible and impossible scenarios and take care of exception to prevent problem occurrence.



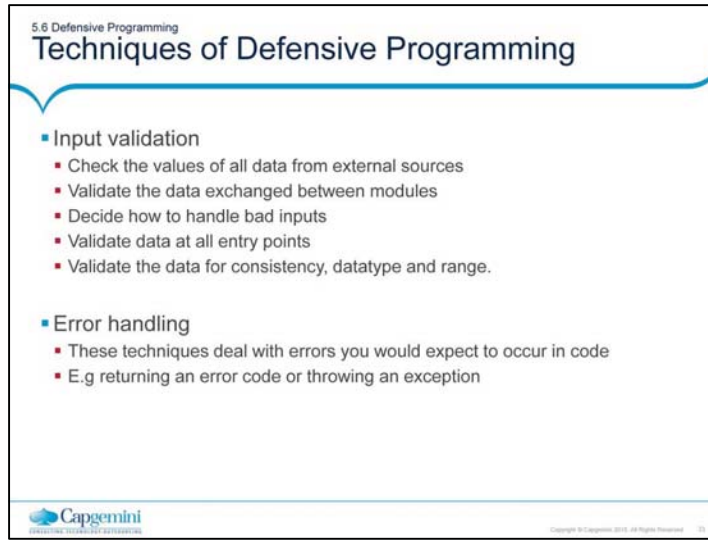
Purpose of Defensive Programming

Ensure that a program never returns inaccurate result even though valid data is passed.

Abnormal termination of the program will be avoided

Even though, invalid data is passed instead of abnormal termination of program execution, meaningful error messages has to be displayed.

Thinking all the possible and impossible scenarios and take care of exception to prevent problem occurrence.



Techniques of Defensive Programming

1. Input validation

Check the values of all data from external sources : When getting data either from a file or from a user, check to be sure that the data falls within the allowable range and correct type of value is accepted.

Validate the data exchanged between modules: Checking the values of routine input parameters is essentially the same as checking data that comes from an external source

Decide how to handle bad inputs: Once you have detected an invalid input, take care of necessary technique to display an error message or return error code.

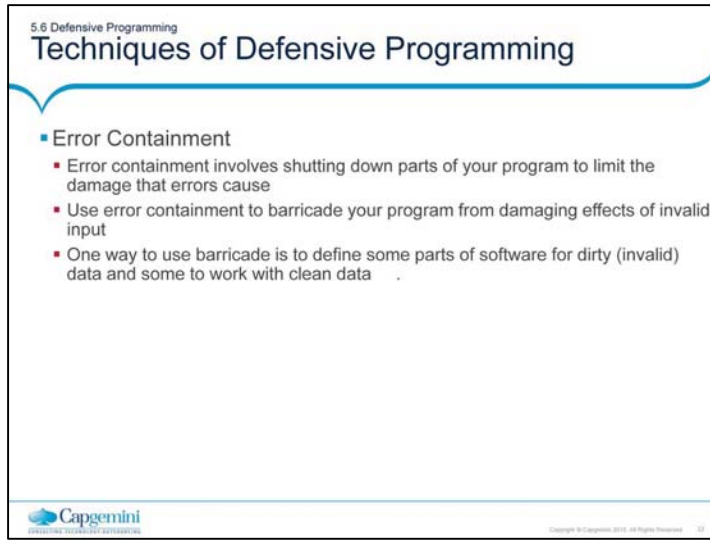
Validate the data for

consistency : The consistency check compares new data with previous data. For example, a current meter reading against past meter readings.

data type : The data type check ensures input data is of the correct data type. For example, numeric or alphabetic.

range : check ensures that input data is within a specified range.

2. Error handling : these techniques deal with errors you would expect to occur in code



Techniques of Defensive Programming(Contd..)

Example for Error Handling:

```
IF (fileExists)  THEN
        Read the data from the file
ELSE
        errorCode = -1; //File doesn't exists
        return errorCode
END IF
```

3. Error Containment

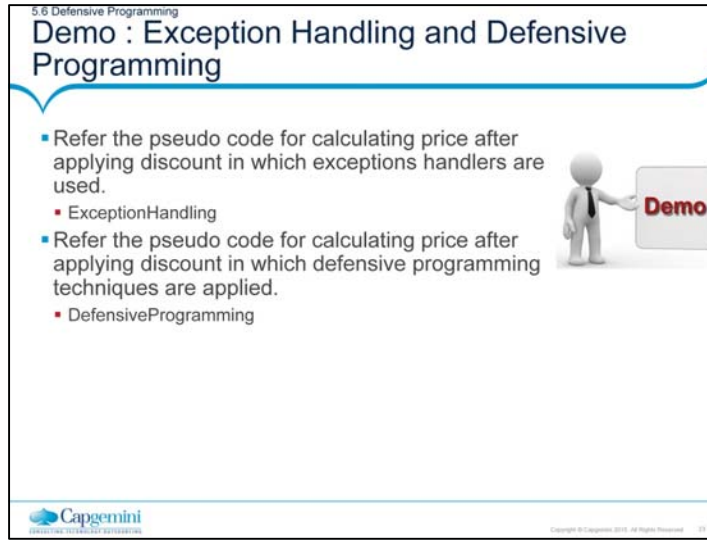
Protect your code from an invalid data coming from "outside"(Data from an external system or the user or a file)

Establish "barricades"(module where validation logic exists) for leaving dangerous data outside of the boundary(like before invoking module), everything inside of the boundary(within module) is safe. For an example, send input parameters to a module only if it is valid(Inside of the boundary is safe)

In the barricade code(isValid Module), validate all input data (check all input parameters) for the correct type, length, and range of values. Double check for limits and bounds.

For an Example:

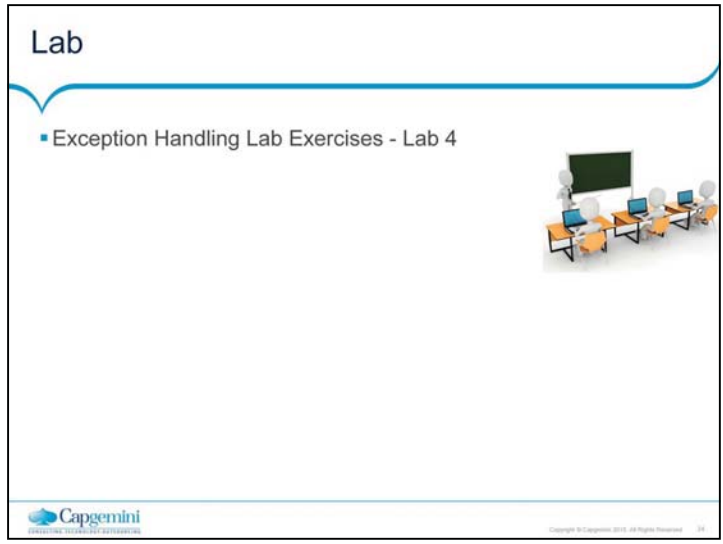
```
IF(data is invalid) THEN
        PRINT error message
ELSE
        Invoke a module to execute logic
END IF
```



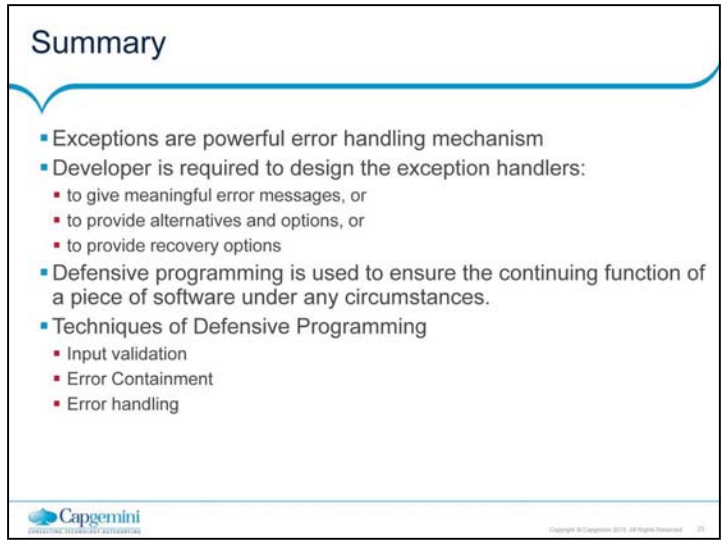
Pseudocode to calculate price after applying discount .

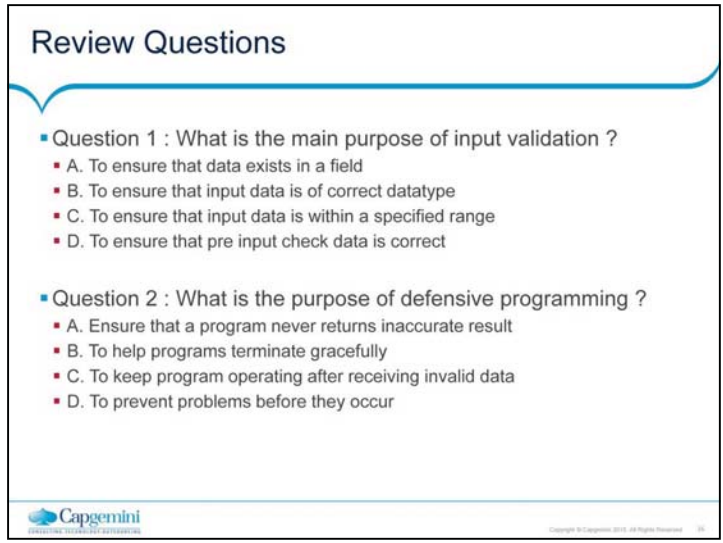
The below defensive programming techniques are applied.

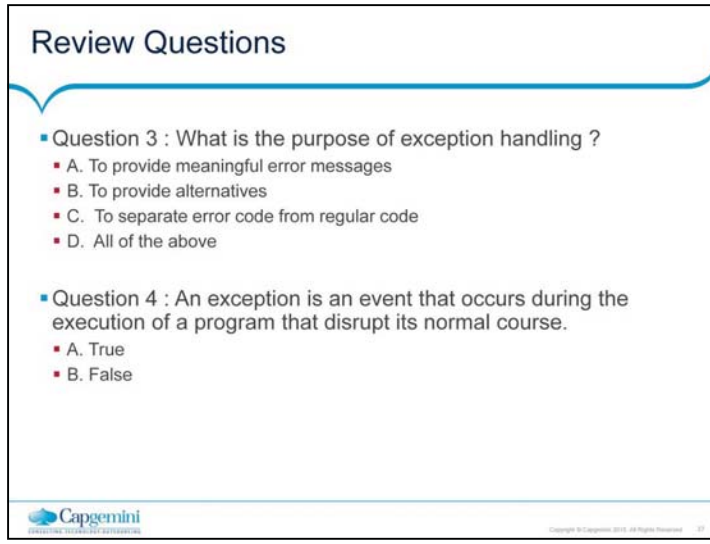
- Input Validation
  - ProductId and discount value should accept only digits
  - ProductId should already exists
- Error Handling
  - Error code(-1) will be returned if productId doesn't exists
- Error Containment
  - applyDiscount module will be invoked only if the given inputs are valid else error message will be printed

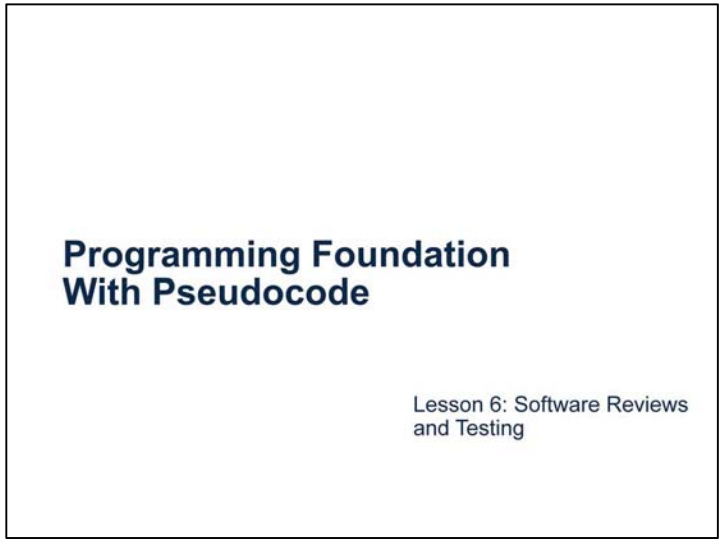


Write pseudocode to test the concept of handling exceptional scenarios.


Summary
Exceptions are powerful error handling mechanism
Developer is required to design the exception handlers:
to give meaningful error messages, or
to provide alternatives and options, or
to provide recovery options
Defensive programming is used to ensure the continuing function of a piece of software under any circumstances.
Techniques of Defensive Programming
Input validation
Error Containment
Error handling
Capgemini


Review Questions
Question 1 : What is the main purpose of input validation ?
A. To ensure that data exists in a field
B. To ensure that input data is of correct datatype
C. To ensure that input data is within a specified range
D. To ensure that pre input check data is correct
Question 2 : What is the purpose of defensive programming ?
A. Ensure that a program never returns inaccurate result
B. To help programs terminate gracefully
C. To keep program operating after receiving invalid data
D. To prevent problems before they occur
Capgemini


Review Questions
Question 3 : What is the purpose of exception handling ?
A. To provide meaningful error messages
B. To provide alternatives
C. To separate error code from regular code
D. All of the above
Question 4 : An exception is an event that occurs during the execution of a program that disrupt its normal course.
A. True
B. False
Capgemini

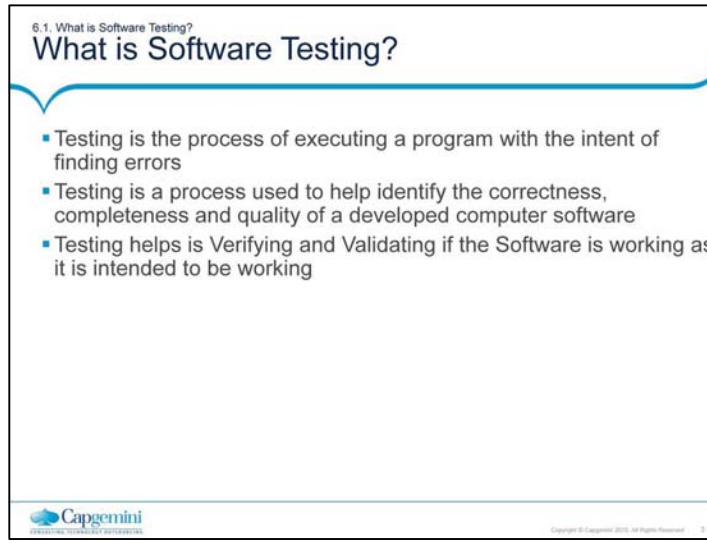

Programming Foundation
With Pseudocode
Lesson 6: Software Reviews
and Testing

Lesson Objectives

- To Understand the following concepts
  - What is software Testing?
  - What is Debugging?
  - Software Testing Principles
  - TestCase
  - Exhaustive Testing
  - Testing Techniques
  - Static Testing
  - Dynamic Testing
  - Testing Approaches
  - Unit Testing
  - Integration Testing
  - System Testing
    - Verification and Validation testing
    - Acceptance Testing
    - Regression testing

Capgemini

Copyright © Capgemini 2015. All Rights Reserved



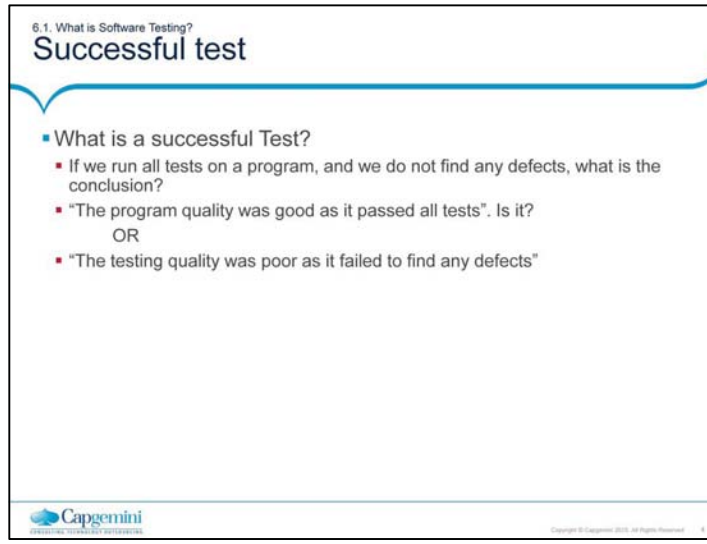
What is software testing?

Exercising (analyzing) a system or component with

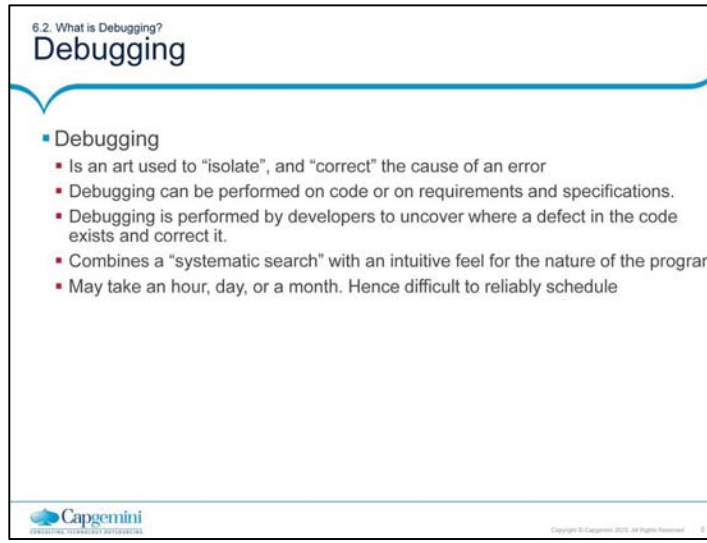
- defined inputs
- capturing monitored outputs
- comparing outputs with specified or intended requirements

To maximize the number of errors found by a finite no of test cases.

Testing is successful if you can prove that the product does what it should not do and does not do what it should do.



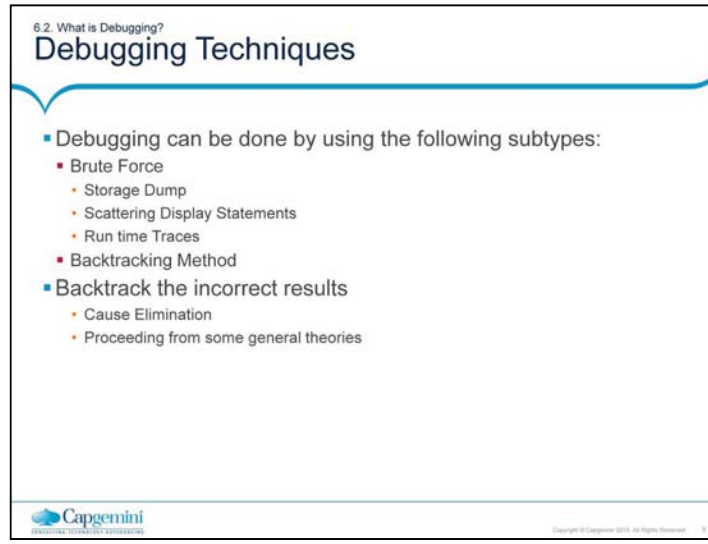
Testing is in a way a destructive process and a successful test case is one that brings out an error in the program . Detection of an error/failure is a success



Objective of debugging is to find and correct the cause of the software error. Debugging is not testing.
The symptom may appear in one part of a program, while the cause may actually be located at a site that is far removed.
The symptom may be caused by round off in accuracies.
Intermittent problems in embedded systems because hardware is tightly coupled with software
As the consequences of an error increase, the amount of pressure to find the cause also increases. Often, pressure forces a software developer to fix one error while at the same time introducing two more.

The debugging process has one of the two outcomes:
The cause will be found, corrected and removed.
The cause will not be found.

In the second outcome, the person performing debugging may suspect a cause, design a test case to help validate his or her suspicion and work toward error correction in iterative manner



Brute Force: It is the most common and least efficient method for isolating the cause of a software error. We apply brute force debugging methods when all else fails.

Example of debugging by Brute Force are

1. By studying Storage Dumps I.e. usually a crude display of storage location
2. by invoking run-time traces
3. by scattering print statements
4. by use of automated debugging tools

Backtracking is a common debugging approach. It is basically used in small programs. The program code is manually tracked beginning at the place where the symptom is uncovered, the source code is traced backward until the site of the cause is found.

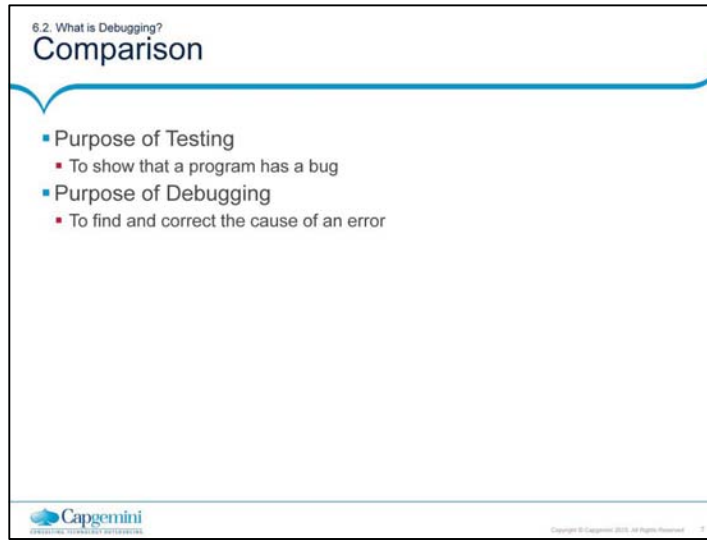
Cause elimination

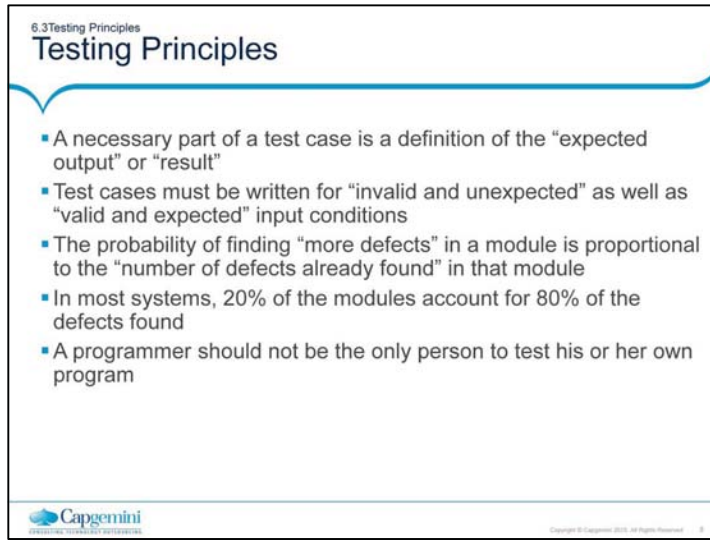
Debugging by Induction

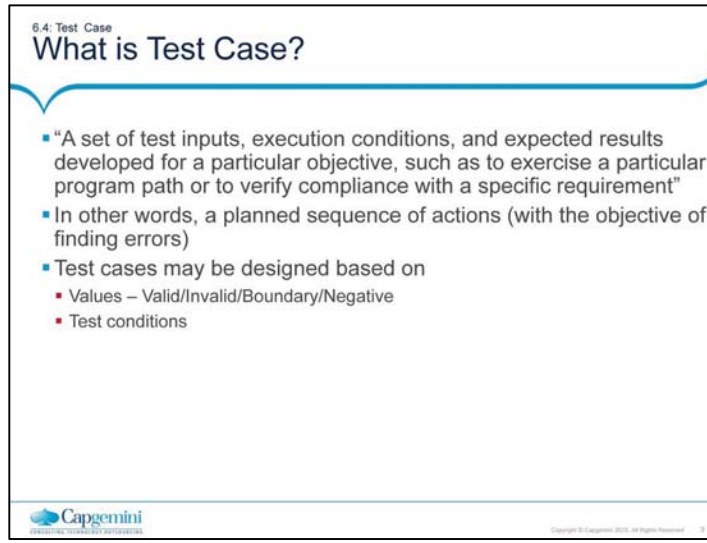
1. Locate data about what program did correctly/incorrectly
2. Organize data
3. Device a hypothesis about the cause of the error
4. Prove the hypothesis

Debugging by deduction

1. Enumerate the causes of error
2. Eliminate each cause of error

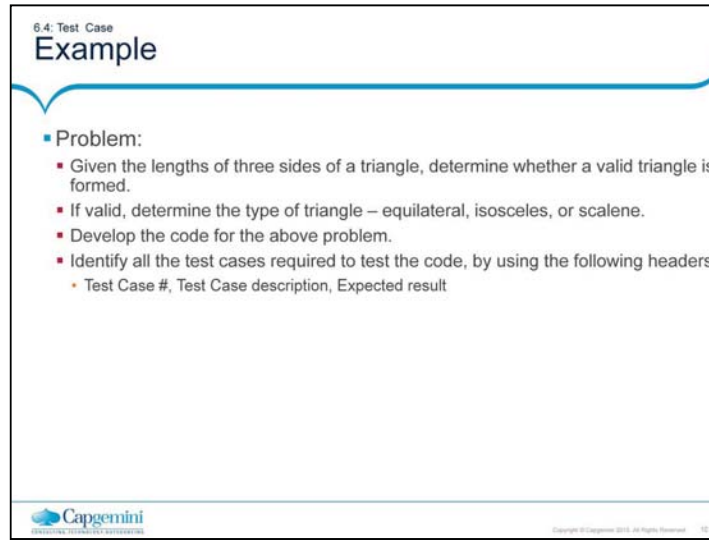

6.2. What is Debugging?
Comparison
Purpose of Testing
To show that a program has a bug
Purpose of Debugging
To find and correct the cause of an error
Capgemini


6.3Testing Principles
Testing Principles
A necessary part of a test case is a definition of the “expected output” or “result”
Test cases must be written for “invalid and unexpected” as well as “valid and expected” input conditions
The probability of finding “more defects” in a module is proportional to the “number of defects already found” in that module
In most systems, 20% of the modules account for 80% of the defects found
A programmer should not be the only person to test his or her own program
Capgemini

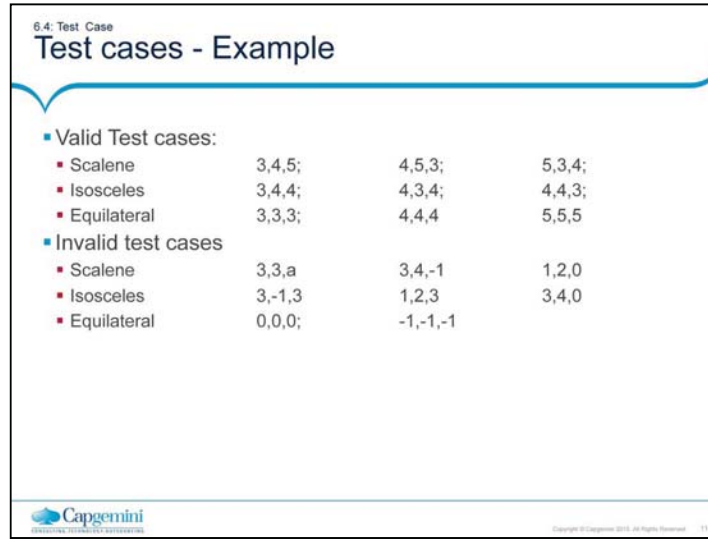


A Test case is a planned sequence of actions.

Characteristics of a Good Test:
They are: likely to catch bugs
not redundant
not too simple or too complex.

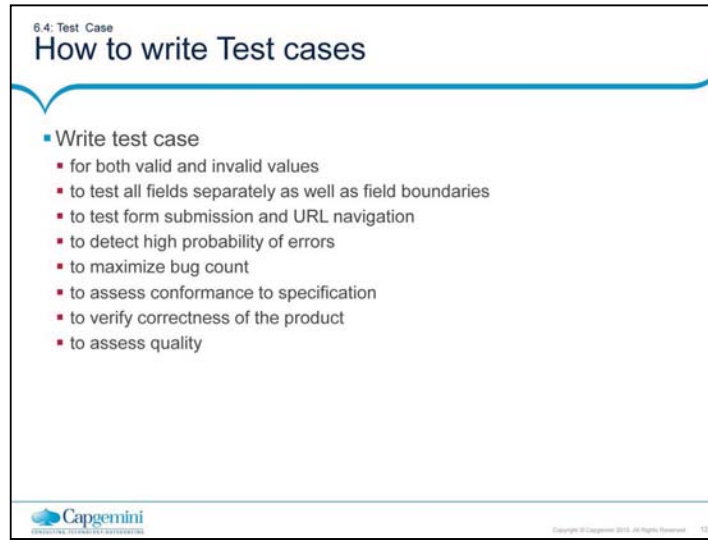


While testing the code consider valid expected and invalid unexpected scenarios.



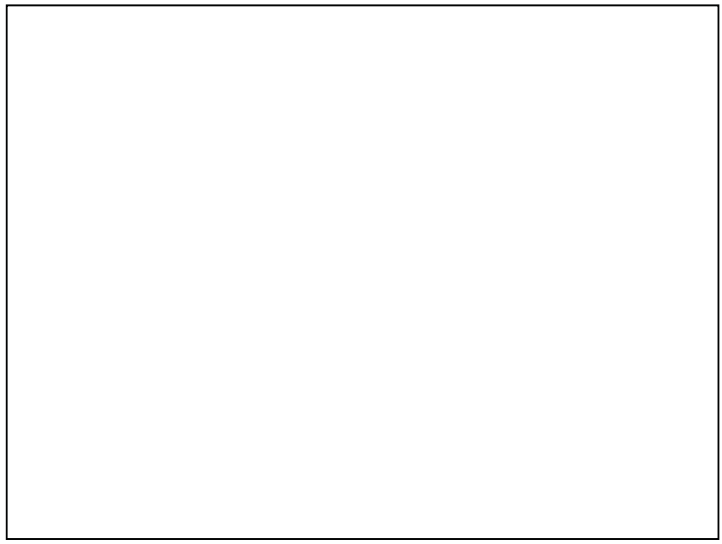
If you observe the examples given on the slide, Some of the input values are checking for valid values for scalene, isosceles, equilateral triangle. and some of the input values are checking invalid values like

- Length of a triangle cannot be negative
- Length of triangle cannot be zero
- Length of a triangle cannot be alphabet etc

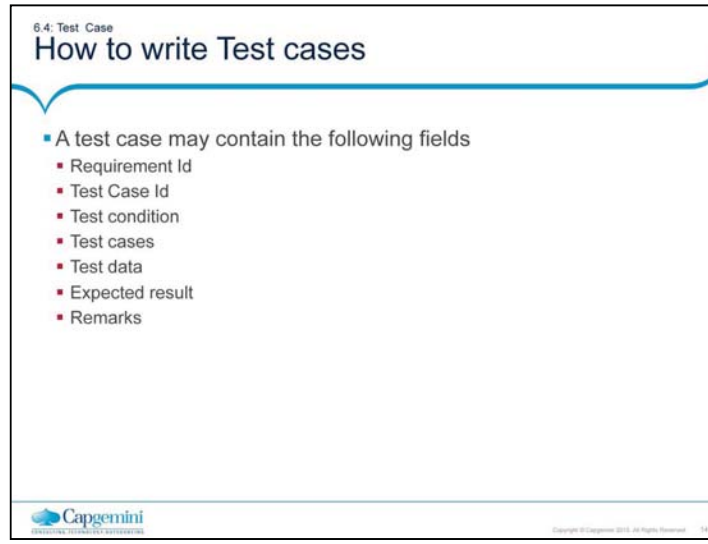


How to write test case?

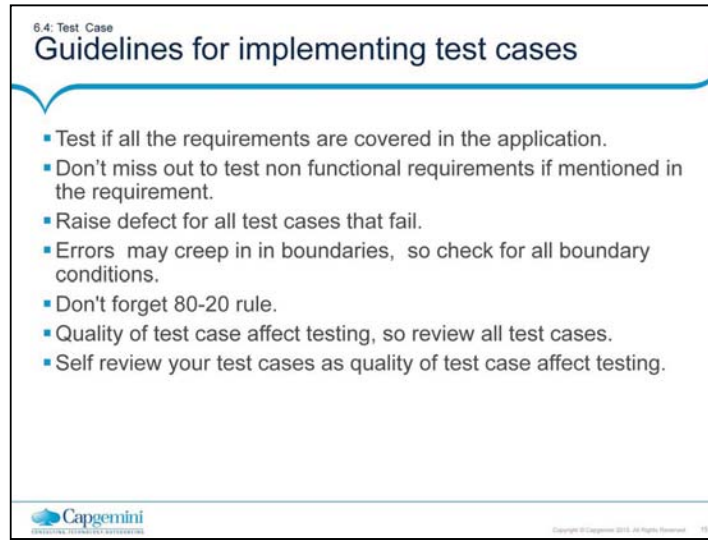
- Write test case for both valid and invalid values. For an Example, if you want to validate the salary, then consider the below mentioned test cases.
    - write test case with valid input data as "10000".
    - write test case with invalid input data as "Test".
- Write test case to test all fields separately as well as field boundaries. For an Example, consider a login form contains two fields like username and password. Write both valid and invalid test cases for all the fields(i.e, Username and Password) available in the page.
- Write test case to test form submission and URL navigation. For an Example, consider a login form contains two fields like username and password. If you want to navigate to the next page, while clicking on submit button after entering valid username and password, then prefer writing test case for form submission.
- Defect high probability of errors: This is the classic objective of testing. A test is run in order to trigger failures that expose defects. Generally, we look for defects in all interesting parts of the product.
- Maximize bug count: The distinction between this and “find defects” is that total number of bugs is more important than coverage. We might focus narrowly, on only a few high-risk features, if this is the way to find the most bugs in the time available.



- Assess conformance to specification. Any claim made in the specification is checked. Program characteristics not addressed in the specification are not (as part of this objective) checked.
- Verify correctness of the product. It is impossible to do this by testing. You can prove that the product is not correct or you can demonstrate that you didn't find any errors in a given period of time using a given testing strategy. However, you can't test exhaustively, and the product might fail under conditions that you did not test. The best you can do (if you have a solid, credible model) is assessment--test-based estimation of the probability of errors. (See the discussion of reliability, above).
- Assure quality. Despite the common title, quality assurance, you can't assure quality by testing. You can't assure quality by gathering metrics. You can't assure quality by setting standards. Quality assurance involves building a high quality product and for that, you need skilled people throughout development who have time and motivation and an appropriate balance of direction and creative freedom. This is out of scope for a test organization. It is within scope for the project manager and associated executives. The test organization can certainly help in this process by performing a wide range of technical investigations, but those investigations are not quality assurance. Given a testing objective, the good test series provides information directly relevant to that objective. Different types of tests are more effective for different classes of information.

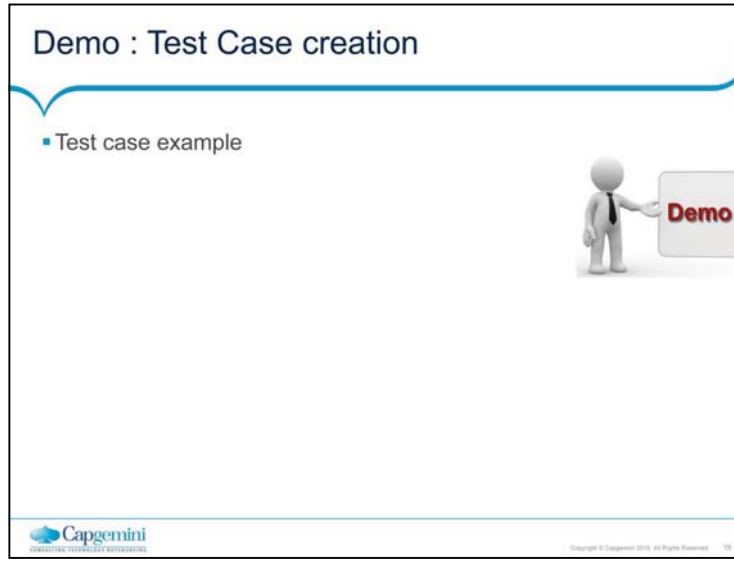


Test case has to be written to validate the testing coverage of the application.
Requirement Id: The ID of the requirement this test case relates/traces to.
Test Case Id: Unique ID for each test case. Follow some convention to indicate types of test. E.g. 'TC_UI_1' indicating 'user interface test case #1'.
Test condition: Describes what to test for i.e. the condition which is being tested for example, Validate name
Test cases: Step-by-step procedure to execute the test.
Test data: Use of test data as an input for this test case. You can provide different data sets with exact values to be used as an input
Expected result: What should be the system output after test execution? Describe the expected result in detail including message/error that should be displayed on screen
Remarks: Any comments on the test case or test execution.

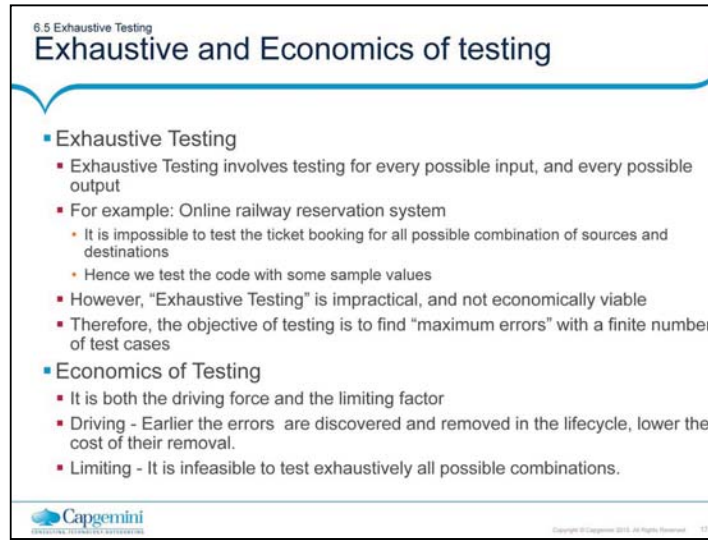


Guidelines for implementing test cases:

- Write test case for all the requirements specified in the application
- Take care of writing test case for non functional requirements like security, performance, etc..
- If any test case fails, log the failed test cases as defect in defect tracking sheet.
- Check for all boundary conditions.
- **80-20 Rule:** In most systems, 20% of the modules account for 80% of the defects found. The probability of finding defect in a module is directly proportional to the number of defects already found in the module.
- Do self review and peer review for all test cases as quality of test case affects testing.

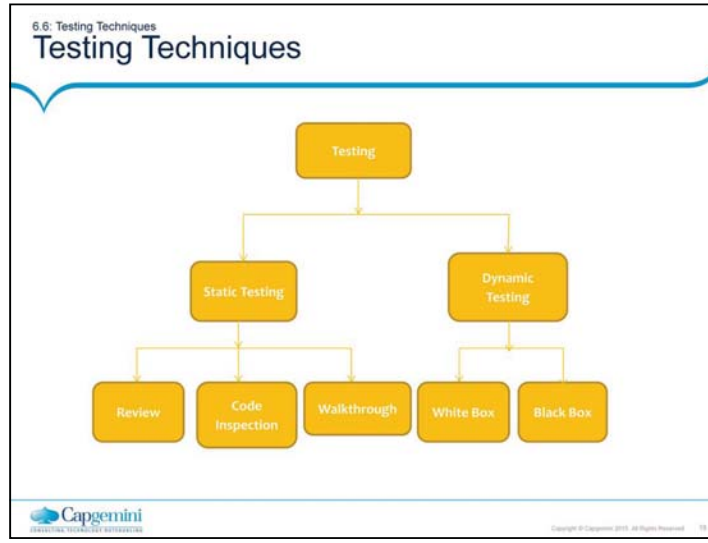


Refer Hotel Bookings Management System.doc and Test_Case-HotelBookingsSystem.xls file to understand test case example.

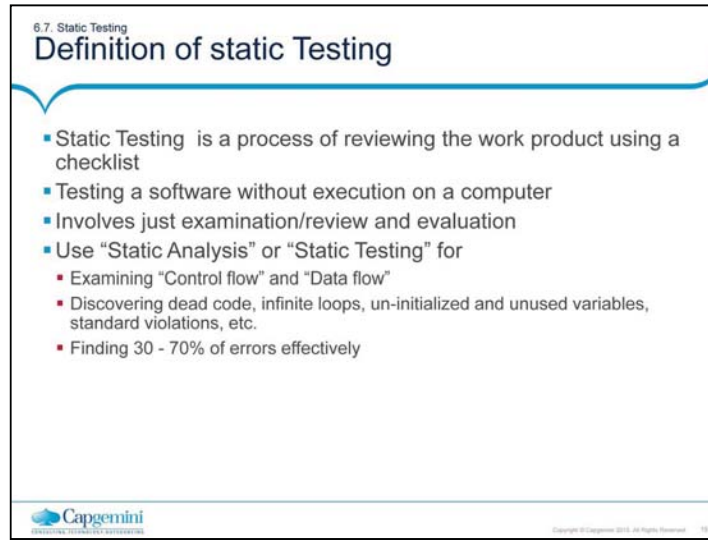


Exhaustive testing is impossible.
For E.g. COBOL Compiler
•Impossible to create test cases to represent all valid cases.
• Impossible to create test cases for all invalid cases
The compiler has to be tested to see that it does not do what it is not supposed to do E.g. to successfully compile a syntactically incorrect program

But writing Cobol programs to include all possible syntactical errors that compiler check is impractical. It is time consuming and hence not suggestible.

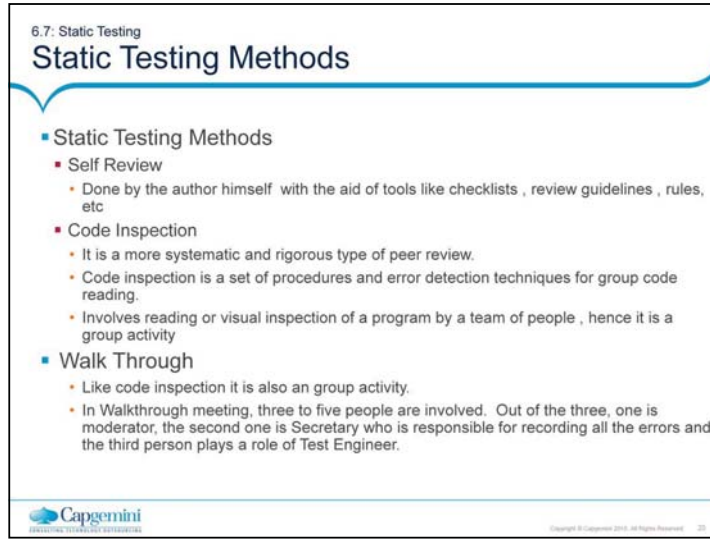


- Static Testing: Testing a software without execution on a computer
  - Review : Review the created artifacts using checklist
  - Code Inspection : Code inspection is a set of procedures and error detection techniques for group code reading.
  - Walkthrough : Like code inspection it is also an group activity.
- Dynamic Testing techniques exercise the software by using sample input values
  - WhiteBox testing : Used to test the internal structure of the code
  - BlackBox Testing : Test the functionality of application by providing input and getting expected output



Static Testing:

- Static Testing is a process of reviewing the work product using a checklist.
- No need to execute a program for performing static Testing.
- Static testing may be conducted manually or through the use of various software review tools like PMD, Checkstyle. Specific types of static software testing include code analysis, inspection, code reviews and walkthroughs.
- Through static testing, errors in the control flow/data flow can be identified at the earlier stage.



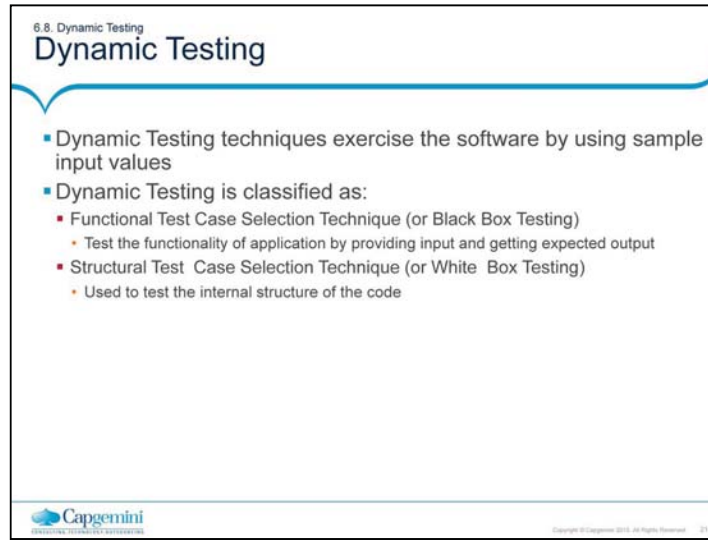
Review Process

Input : Work Product , Specifications, Checklists, Guidelines, Historical Data

Process

- Prepare for Review
- Conduct Reviews
- Analyze Deviations
- Correct Defects

Output : Review Form, reviewed work product



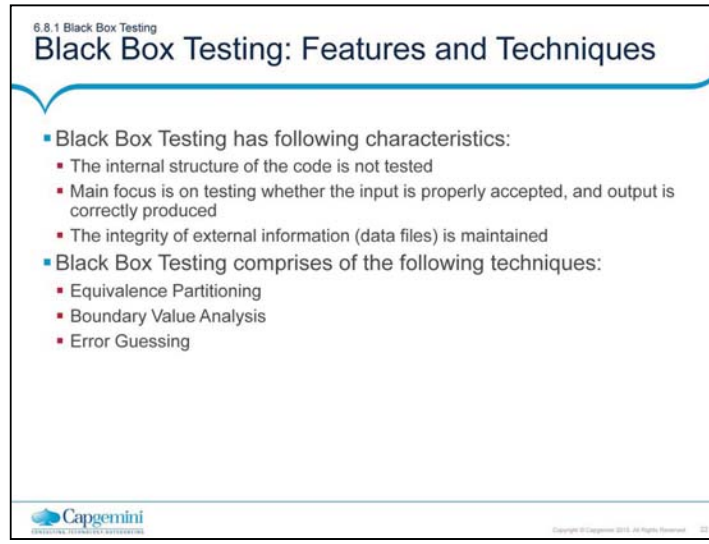
Dynamic Testing

- Dynamic Testing involves the testing of a software by executing the system.
- Using Dynamic Testing, internal structure and functionality of an application will be tested.
- Perform white box testing for testing the internal structure of the code
- Test the functionality of an application using Black box testing.

For an Example, if you want to validate all the fields in the login page to accept valid data, then perform black box testing.

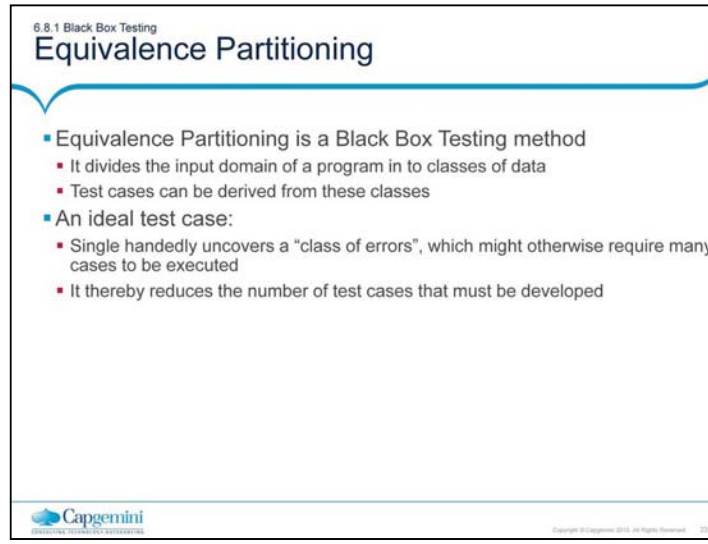
Steps to be performed for black box testing are:
1. Execute an application
2. Type the input
3. Validate the actual result against the expected result mentioned in the test plan.
4. If actual result and expected result matches, then the test case result is pass else the result is fail.

For an Example, if you want to check whether the logic(code) of login functionality is working fine, then perform white box testing.

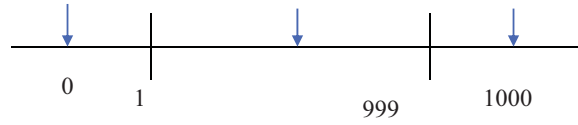


E.G
While testing washing machine, you need not be aware of internal structure of washing machine.
We need to know how to use interface and give instructions to washing machine.

Hence in black box testing we need to know what is input and what is output.



Examples
If an input condition specifies that a variable, say ***count,*** can take range of values(1 - 999),
Identify - one valid equivalence class (1 < ***count*** < 999)
- two invalid equivalence classes (***count*** < 1) & (***count*** >999)



Equivalence classes may be defined according to the following guidelines.
1. If an input condition specifies a range, one valid and two invalid equivalence classes are defined.
2. If an input condition requires a specific value, one valid and two invalid EC are defined.
3. If an input condition specifies a member of a set, one valid and one invalid EC are defined.
4. In an input condition is Boolean, one valid and one invalid class are defined.

6.8.1 Black Box Testing

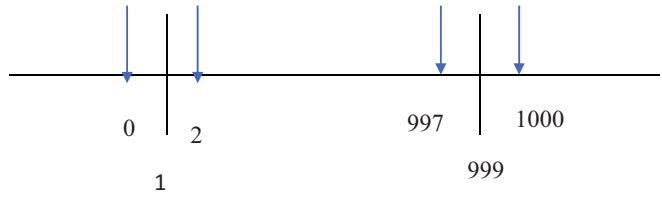
Boundary Value Analysis

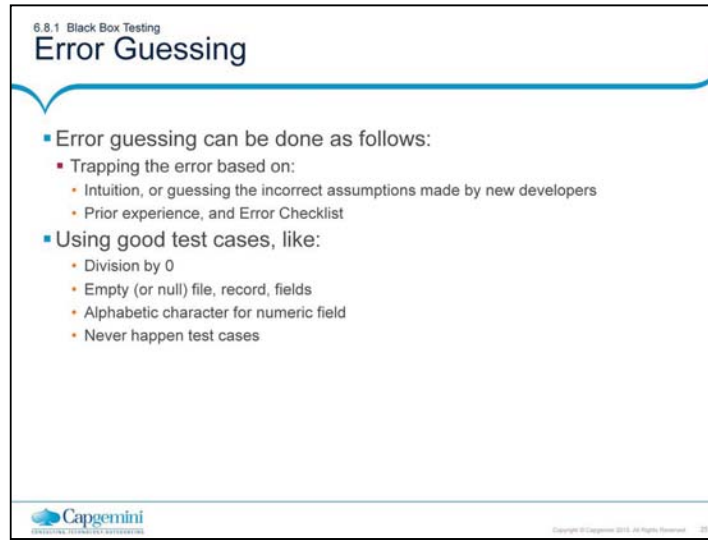
- Boundary Value Analysis (BVA) complements Equivalence Partitioning
  - Rather than selecting any element of an Equivalence Class, BVA leads to select test cases at the "edges of the class"
- Guidelines:
  - If an "input condition" specifies a range of values "a" and "b", test cases should be designed with values "a" and "b", just above and just below, "a" and "b" respectively

Capgemini

Copyright © Capgemini 2015. All Rights Reserved

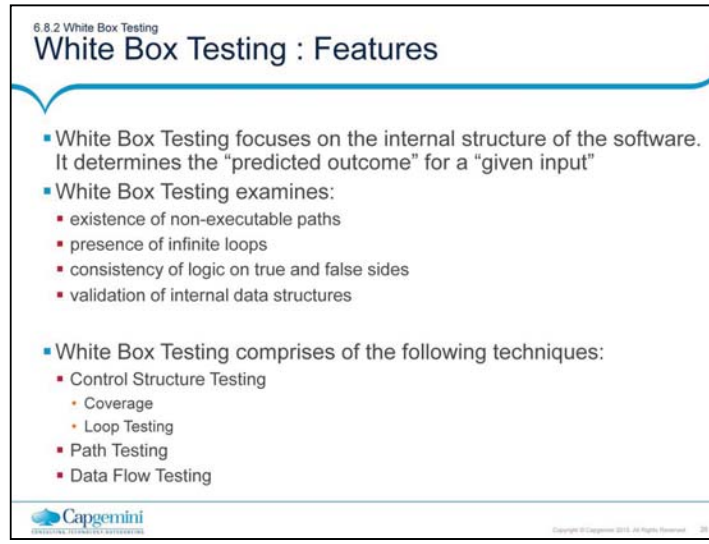
From **previous example,** we have the valid equivalence class as (1 < ***count*** < 999).
Now, according to boundary value analysis, we need to write test cases for ***count=0, count=1, count=2, count=997, count=999 and count=1000 respectively***

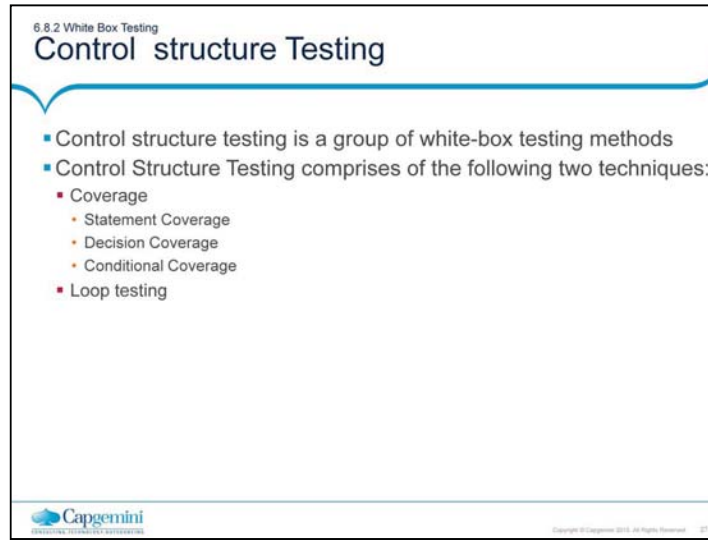




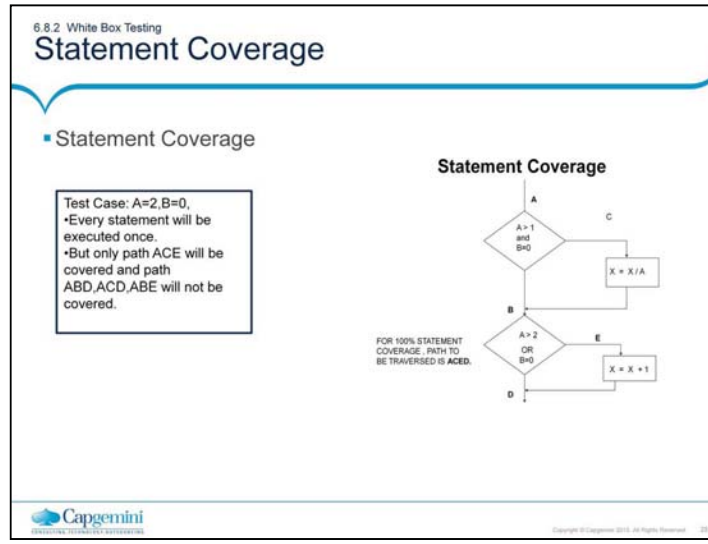
Examples
Suppose we have to test the login screen of an application. An experienced test engineer may immediately see if the password typed in the password field can be copied to a text field which may cause a breach in the security of the application.

Error guessing testing for sorting subroutine situation
- The input list empty
- The input list contains only one entry
- All entries in the list have the same value
- Already sorted input list


6.8.2 White Box Testing
White Box Testing : Features
White Box Testing focuses on the internal structure of the software. It determines the "predicted outcome" for a "given input"
White Box Testing examines:
existence of non-executable paths
presence of infinite loops
consistency of logic on true and false sides
validation of internal data structures
White Box Testing comprises of the following techniques:
Control Structure Testing
Coverage
Loop Testing
Path Testing
Data Flow Testing
Capgemini

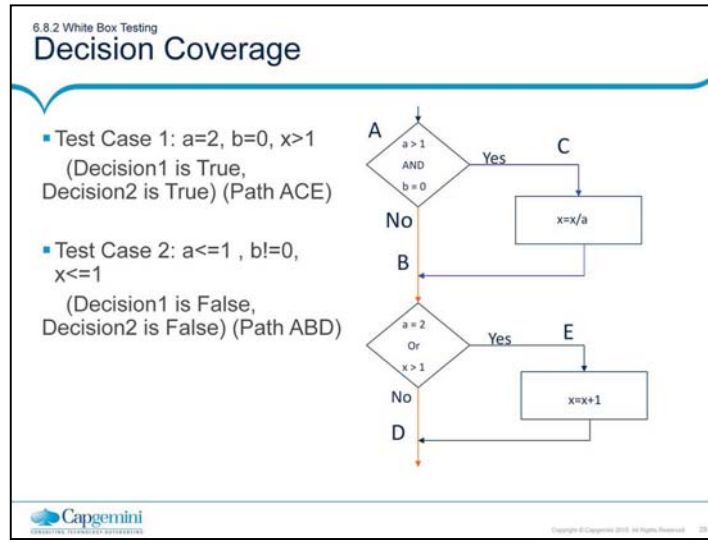


"Coverage" is a measure of how thoroughly a program is exercised



Statement Coverage:
Every statement can be executed by writing a single test case. This case covers only ACE path.
This criteria is weak one. Since it is not considering other paths to traverse. So the path ABD, ACD, ABE would go undetected.

```
BEGIN
        PRINT "Enter 2 numbers"
        READ a and b
        If (a>1) && (b=0) THEN
                x=x/a;
        ELSE IF (a=2 || x>1) THEN
                x=x+1;
        END IF
        PRINT x
END
```



Decision Coverage:

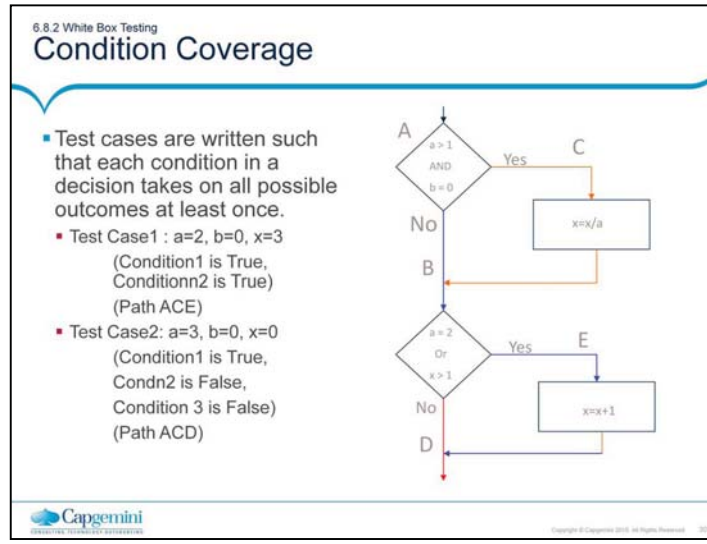
- Decision means any predicate. i.e the statement which returns true or false.
- In decision coverage test cases should be designed in such a way that each decision will be tested for true and false value.

```
BEGIN
        PRINT "Enter 2 numbers"
        READ a and b
        If (a>1) && (b=0) THEN
                x=x/a;
        ELSE IF (a=2 || x>1) THEN
                x=x+1;
        END IF
        PRINT x
END
```

Example: In the above example there are 2 decisions

1. a>1 and b=0
2. 2. a=2 or x>1

- So decision coverage can cover two test cases covering paths ACE and ABD. Even if the above test cases satisfy decision coverage it still does not cover the path ACD and path ABE. Hence decision coverage though stronger criteria than statement it is still weak. There is only 50 percent chance that we would explore the path.



Condition testing is a test case design method that exercises the logical conditions contained in a program module. A simple condition is a Boolean variable or a relational expression. Relational operator is one of the following <, <=, =, not =, > , =>.
A compound condition is composed of two or more simple conditions, Boolean operators, and parentheses.

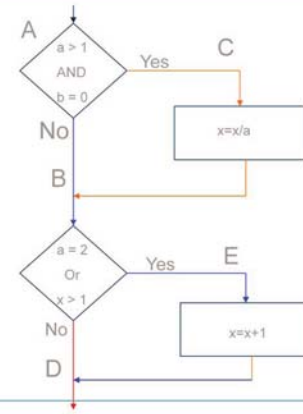
Condition coverage focuses on testing each condition in a program. The purpose of the condition testing is to detect not only errors in the conditions of a program but also other errors in the program.

```
BEGIN
        PRINT "Enter 2 numbers"
        READ a and b
        If (a>1) && (b=0) THEN
                x=x/a;
        ELSE IF (a=2 || x>1) THEN
                x=x+1;
        END IF
        PRINT x
END
```

6.8.2 White Box Testing

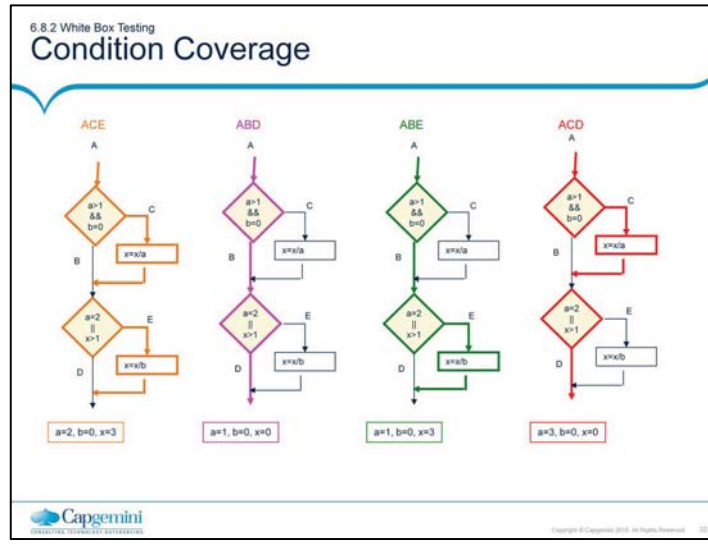
Condition Coverage

- Test Case3 : a=1, b=0, x=3
  (Condition1 is False, Condition2 is True)
  (Path ABE)
- Test Case4: a=1, b=1, x=1
  (Condition1 is False, Condition2 is False)
  (Path ABD)



Capgemini

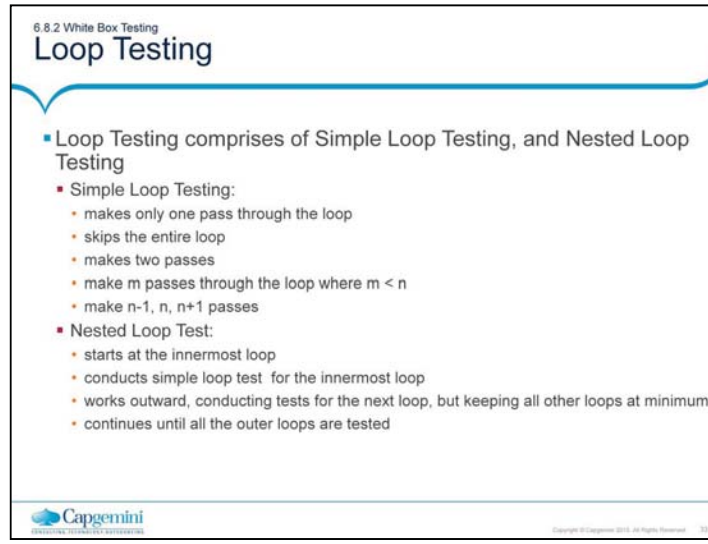
Copyright © Capgemini 2015. All Rights Reserved 31



What does “coverage” mean?

- NOT all possible combinations of data values or paths can be tested
- Coverage is a way of defining how many of the paths were actually exercised by the tests
- Coverage goals can vary by risk, trust, and level of test

In the above diagrams, each condition in decision takes all possible outcomes at least once.



Guidelines for identifying test cases for white box testing

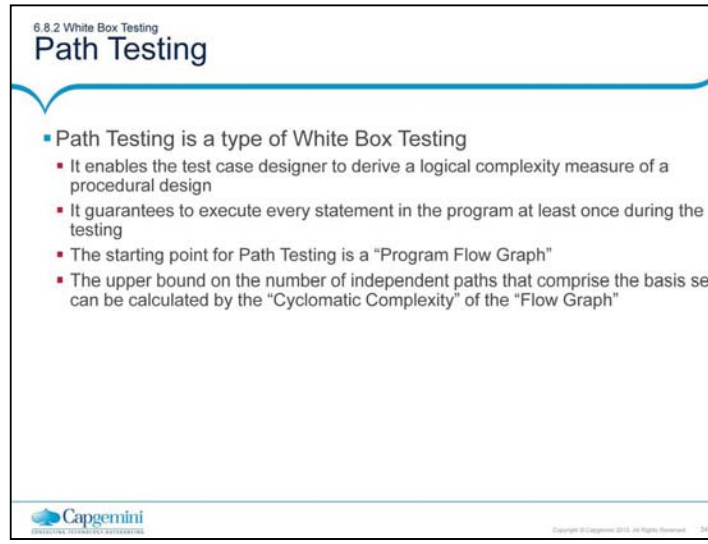
Look at each line of code, and check if a test is needed.

Look at computations – check if positive, negative, 0 test is needed.

Look at IF conditions – ensure that both sides of the conditions are tested.

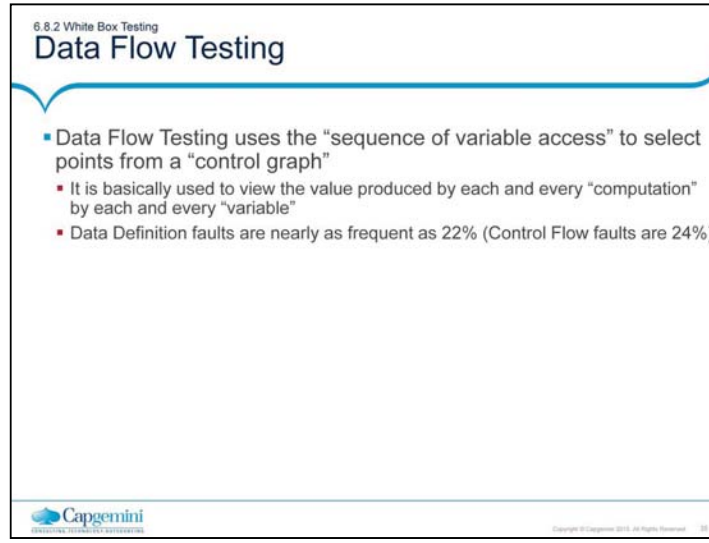
Look at Loops – test for 0, 1, n, and (n+1) iterations (remember Induction!)

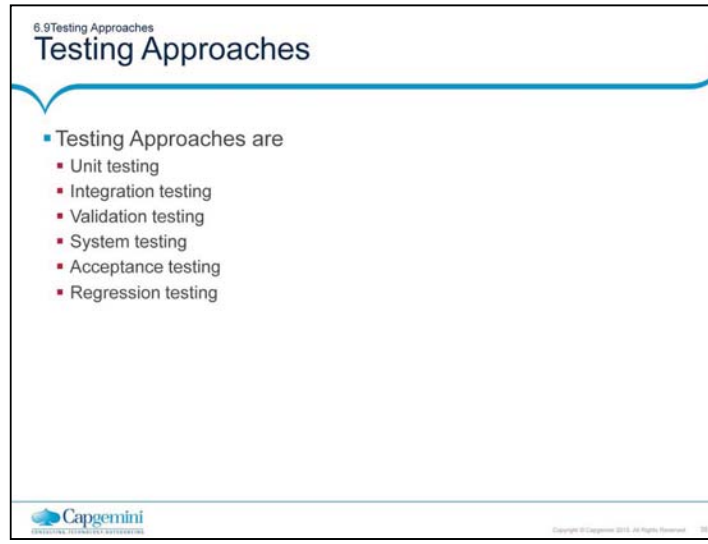
Look for special operations like file IO, memory / string manipulations.



Flow graph:

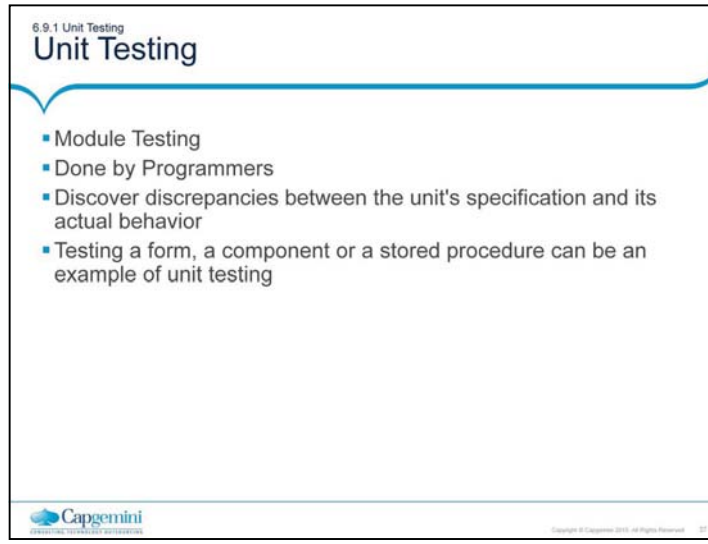
- The "Flow Graph" is the "main tool" for test case identification.
- A "Flow Graph analysis" is concerned with statically determining the number of different paths by which the flow of control can pass through an algorithm.
- It shows the relationship between "program segments".
  - A program segment is a sequence of statements. The program segment has a property that when the first member of the sequence is executed, all the other statements in that sequence get executed, as well.
- Nodes represent one program segment.
  - Nodes bounded by edges and nodes are called "regions".
  - Areas bounded by edges and nodes are called "regions".
- An independent path is any path through the program that introduces at least one new set of processing statements or a new condition.
- An independent path must move along at least one edge that has not been traversed before the path is defined.


6.8.2 White Box Testing
Data Flow Testing
Data Flow Testing uses the "sequence of variable access" to select points from a "control graph"
It is basically used to view the value produced by each and every "computation" by each and every "variable"
Data Definition faults are nearly as frequent as 22% (Control Flow faults are 24%)
Capgemini



Testing Approaches:

- Unit testing is code-based and performed primarily by developers to demonstrate that their smallest pieces of code execution works properly.
- Integration testing demonstrates that two or more units or other integrations work together properly.
- Validation Testing can be used for performing validation of software typically includes evidence that all software requirements have been implemented correctly and completely and are traceable to system requirements.
- System testing demonstrates that the system works end-to-end in a production-like environment to provide the business functions specified in the high-level design(both functional and not functional requirement)
- Acceptance testing is conducted by business owners and users to confirm that the system does, in fact, meet their business requirements.
- Regression Testing is the testing of software after a modification has been made to ensure the reliability of each software release.



What is a Unit?

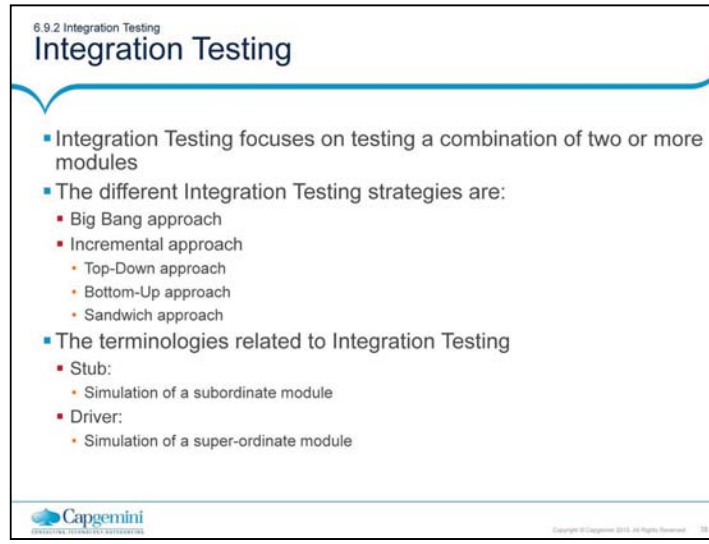
Synonyms are “component” and “module.”

The IEEE glossary says (for module):

- A program unit that is discrete and identifiable with respect to compiling, combining with other units, and loading.
- A logically separable part of a program.

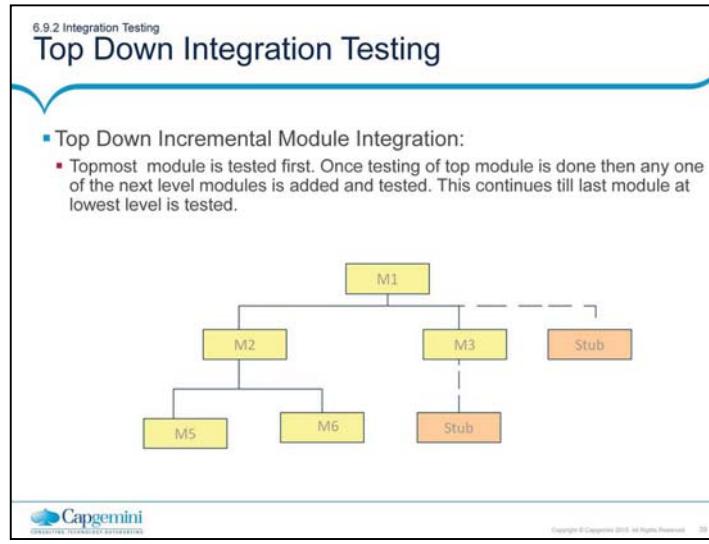
Unit Testing:

- The most 'micro' scale of testing to test particular functions, procedures or code modules. Also called as Module testing.
- Typically done by the programmer and not by Test Engineers, as it requires detailed knowledge of the internal program design and code.
- Purpose is to discover discrepancies between the unit's specification and its actual behavior.
- Testing a form, a class or a stored procedure can be an example of unit testing



Non-incremental Testing (Big Bang Testing)

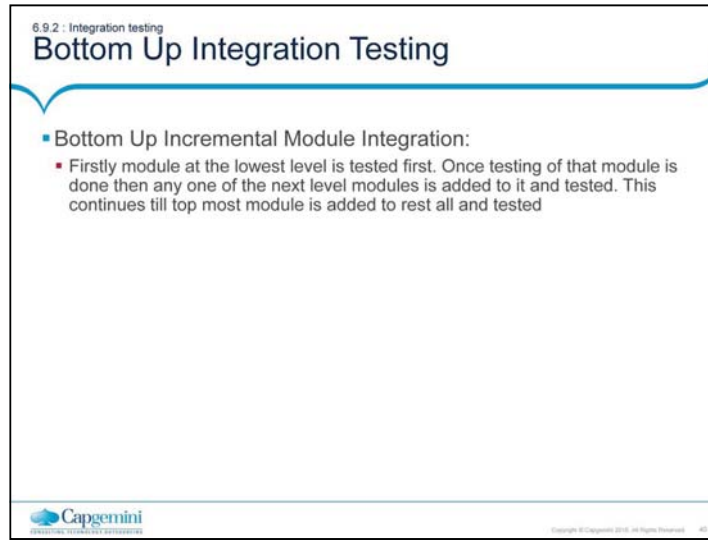
Each Module is tested independently and at the end, all modules are combined to form a application.



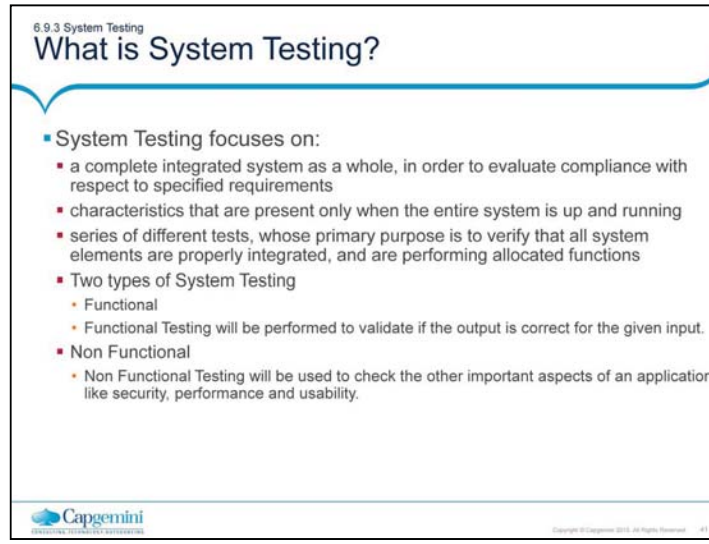
The main control module is used as a test driver. Stubs are substituted for all components directly subordinate to the main control module. Depending on the approach subordinate stubs are replaced by actual components.

Disadvantages:
Many tests are delayed until stubs are replaced by actual modules.
Time taken to develop stubs to perform the functions of the actual modules.

Advantage :
Fast

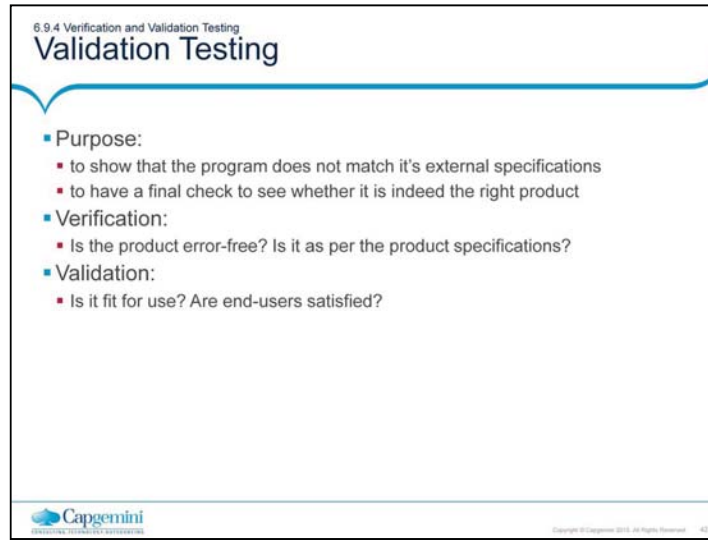


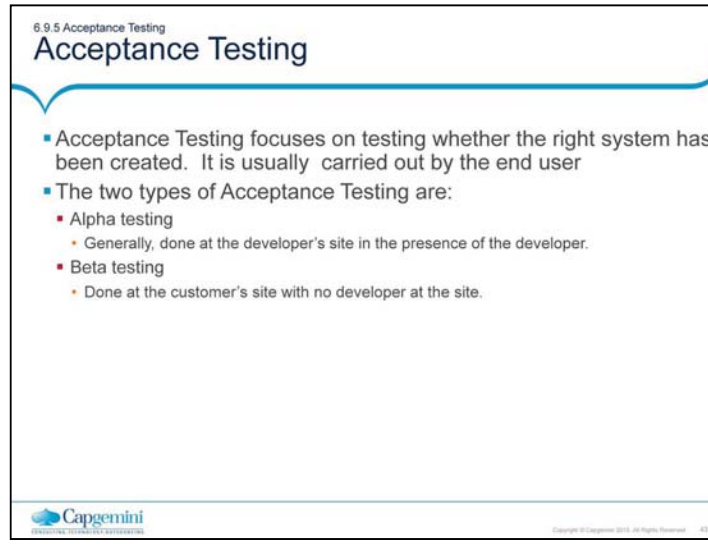
Low-level components are combined into clusters (builds) that perform a specific sub function. A driver is written to coordinate test case input and output. Drivers are removed and clusters are combined moving upward in the program structure.



Types of System Testing:

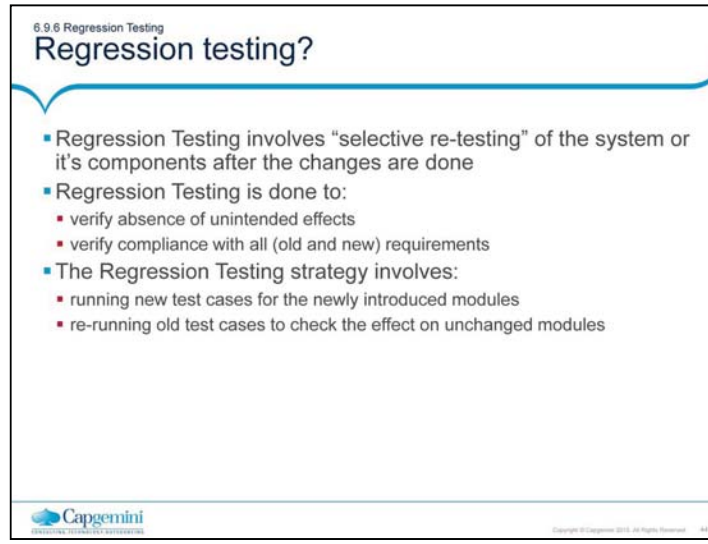
- Functional Testing will be performed to validate if the output is correct for the given input.
- Non Functional Testing will be used to check the other important aspects of an application like security, performance and usability.


6.9.4 Verification and Validation Testing
Validation Testing
Purpose:
to show that the program does not match it's external specifications
to have a final check to see whether it is indeed the right product
Verification:
Is the product error-free? Is it as per the product specifications?
Validation:
Is it fit for use? Are end-users satisfied?
Capgemini



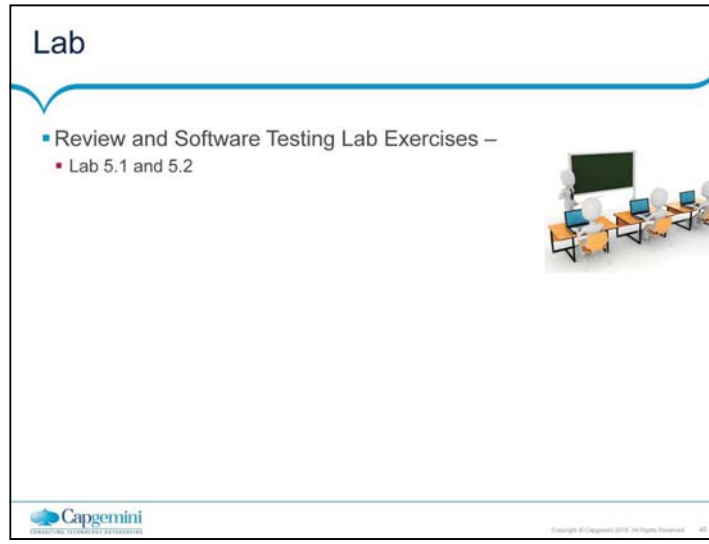
Acceptance Testing:

A test executed by the end user(s) in an environment simulating the operational environment to the greatest possible extent, that should demonstrate that the developed system meets the functional and quality requirements.

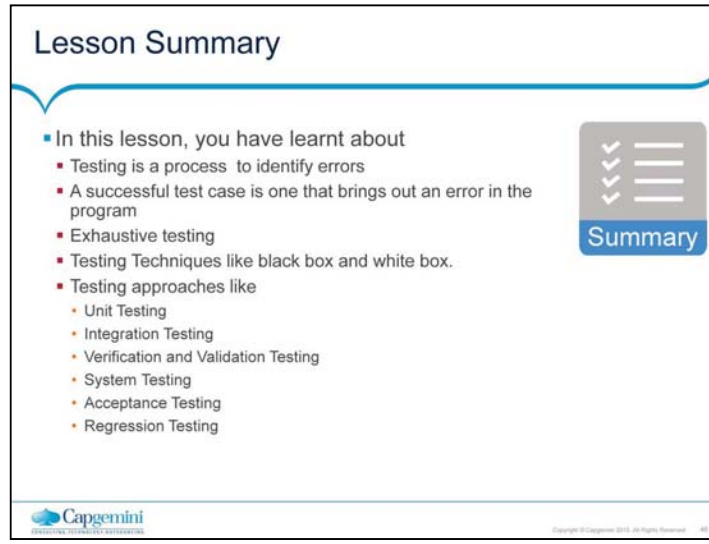


Regression Testing:

- Regression Testing is the testing of software after a modification has been made to ensure the reliability of each software release.
- Testing after changes have been made to ensure that changes did not introduce any new errors into the system.
- It applies to systems in production undergoing change as well as to systems under development
- Re-execution of some subset of test that have already been conducted is required



Do review of existing pseudocode and write test case to perform unit testing of an program which you have created.


Lesson Summary
In this lesson, you have learnt about
Testing is a process to identify errors
A successful test case is one that brings out an error in the program
Exhaustive testing
Testing Techniques like black box and white box.
Testing approaches like
Unit Testing
Integration Testing
Verification and Validation Testing
System Testing
Acceptance Testing
Regression Testing
Summary
Capgemini

Review - Questions

- Question 1: Alpha Testing is ----------
  - A : done at the developer's site in the presence of the end user.
  - B: done at the developer's site in the presence of the developer.
  - C: done at the end user's site in the presence of the developer.
  - D: done at the end user's site in the presence of the end user.
- Question 2 : Which of the following are true for testing
  - A: Test cases should be designed for valid and expected values only
  - B: While testing you should check for each and every possible value
  - C: White Box Testing focuses on the internal structure of the software
  - D: The purpose of testing is to find and correct the cause of the error

Capgemini

Copyright © Capgemini 2015. All Rights Reserved

Review - Questions

- Question 3: In simple loop testing minimum -------- test cases need to be designed
  - A: 5
  - B:6
  - C:7
  - D: None of the above
- Question 4 : In top down approach of integration testing we replace modules with -------
  - A: driver
  - B: stub
  - C: control module
  - D: All of the above

Capgemini

Copyright © Capgemini 2015. All Rights Reserved

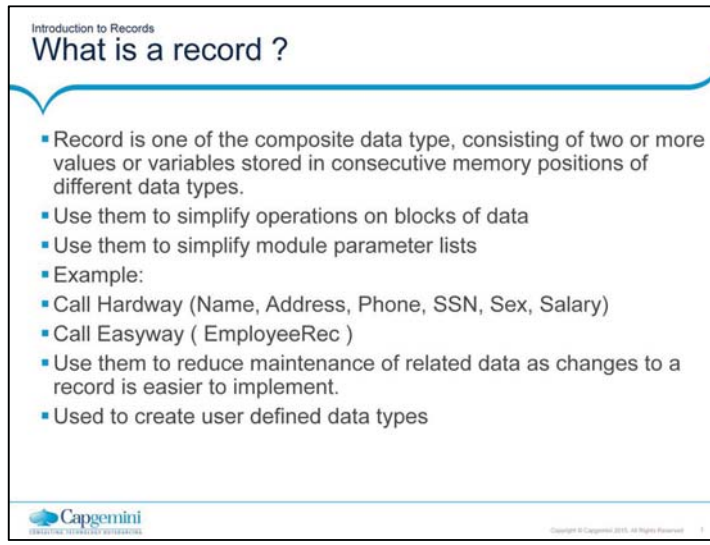
Review – Match the Following

- Question 5 :

1. Acceptance testing	A. Big Bang approach
2. Regression testing	B. Exhaustive testing
3. Integration testing	C. Beta testing
4. Use every possible input condition as a test case	D. Selective retesting
5. Debugging	E. Maximize bug count
	F. Storage dump

Capgemini

Copyright © Capgemini 2015. All Rights Reserved

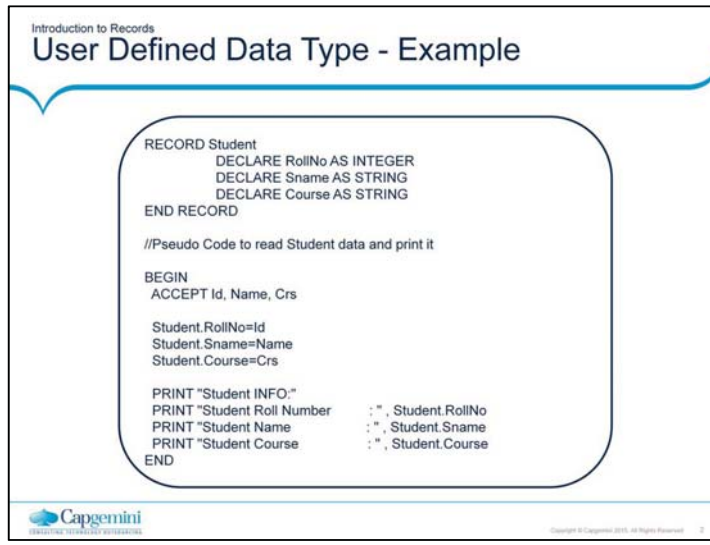


Record: Record is one of the composite data type, consisting of two or more values or variables stored in consecutive memory positions .

Example for record:

```
RECORD Employee
        DECLARE ecode AS INTEGER
        DECLARE ename AS STRING
        DECLARE esal AS INTEGER
        DECLARE edept AS STRING
END RECORD
```

The above record is used to hold details about an employee such as employee code, employee name, employee salary and department in which employee is working.

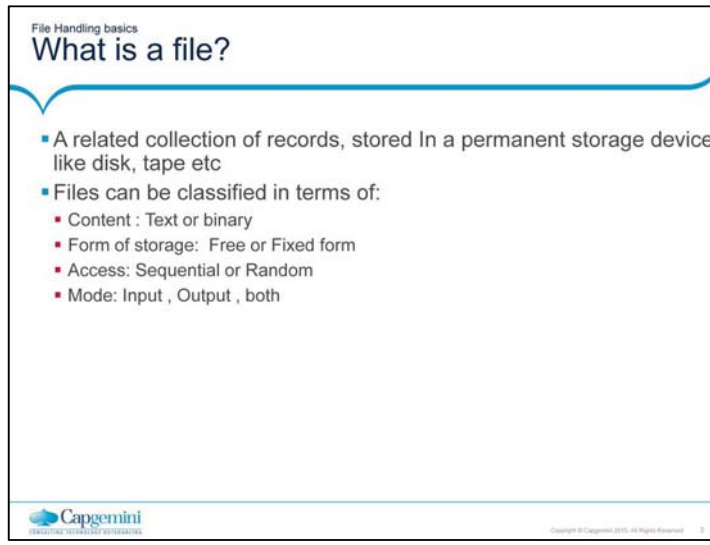


The code given in slide is used to maintain information about a student like rollno, sname and course details.

Functionalities implemented in the above code are

Accepting student details like id, name and course

Printing the same.

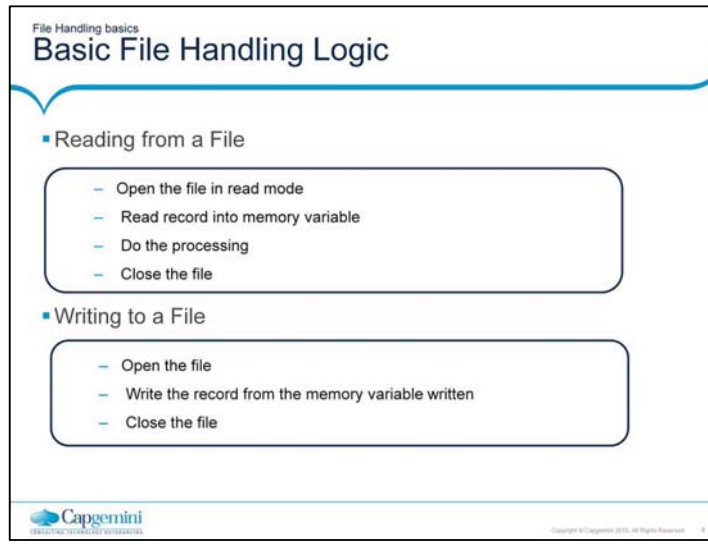


Files:
So far the input data was read from standard input and the output was displayed on the standard output. These programs are adequate if the volume of data involved is not large. However many business-related applications require that a large amount of data be read, processed, and saved for later use. In such a case, the data is stored on storage device, usually a disk in the form of file.
There are two types of files:

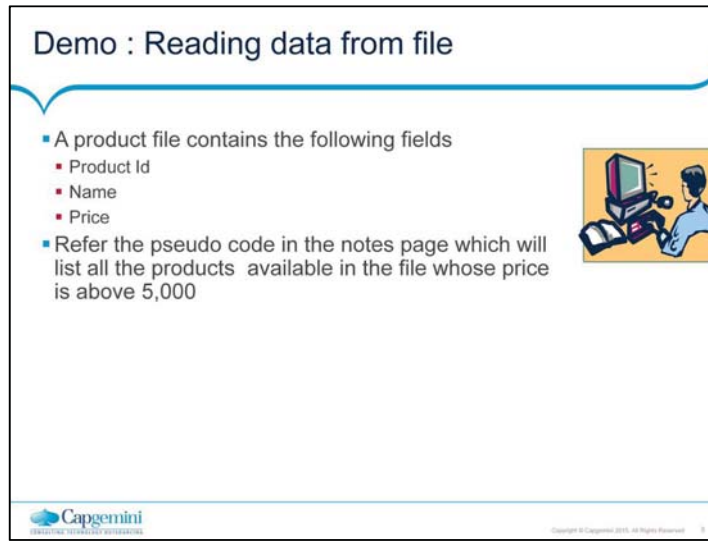
Binary file:
It is efficient for large amount of numeric data.

Text file:
It stores text and numbers as one character per byte.
It consumes large amount of storage for numbers.
It is useful for printing reports and text documents.

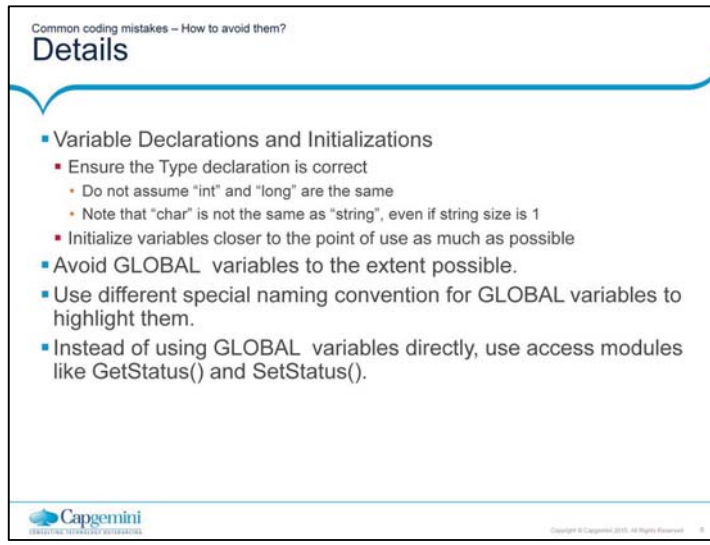
Access: Data from the file is accessible either sequentially or randomly.
Mode: Data from the file can be readable(Output) or data can be writable to the file(Input).



File I/O requires four functions:
open: It allows access to file.
close: It ends access to a file.
read: It gets data from file.
write: It adds data into file.




```
RECORD ProductRec
        DECLARE productId AS INTEGER
        DECLARE name AS STRING
        DECLARE price AS INTEGER
END RECORD
BEGIN
        DECLARE file AS FILE
        DECLARE product AS ProductRec
        PROMPT "Enter the filename" AND STORE IN file
        IF(fileExists(file)) THEN
              OPEN file
              READ data from the file AND STORE IN product
              IF(product.price>5000) THEN
                          DISPLAY "Product Id" + product.productId
                           DISPLAY "Product Name" +
product.name
                           DISPLAY "Product Price" + product.price
              END IF
         END IF
END
```



Variable Declarations and Initializations:

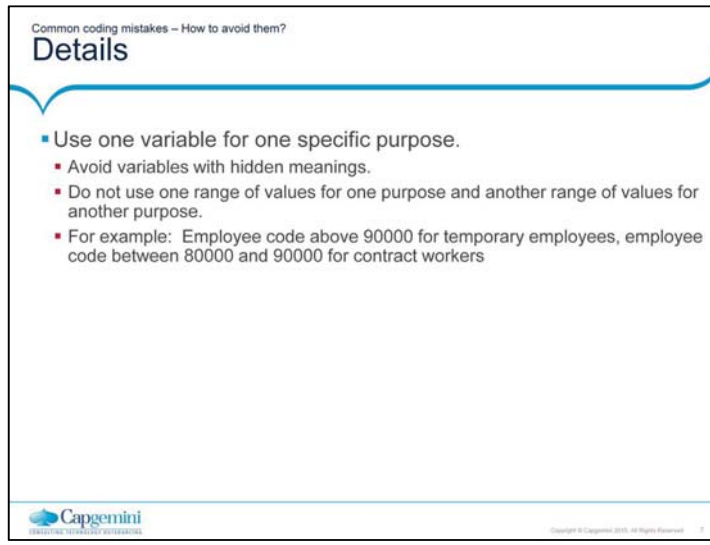
Code Snippet - Example:

```
Boolean more-records = TRUE;
…
while ( more-records ) do …
```

Better version of the above code snippet:

```
Boolean more-records;
…
more-records = TRUE;
while ( more-records ) do …
```

Avoid GLOBAL variables to the extent possible. : Note that it is difficult to debug, because a GLOBAL variable can be changed from any module. It may create problems with a recursive module.
Use different special naming convention for GLOBAL variables to highlight them. For example: MAX_USERS_G for global
Instead of using GLOBAL variables directly, use access modules like GetStatus() and SetStatus(). Note that this ensures that the variable is used only at once place inside those modules. This becomes easier to change or debug.

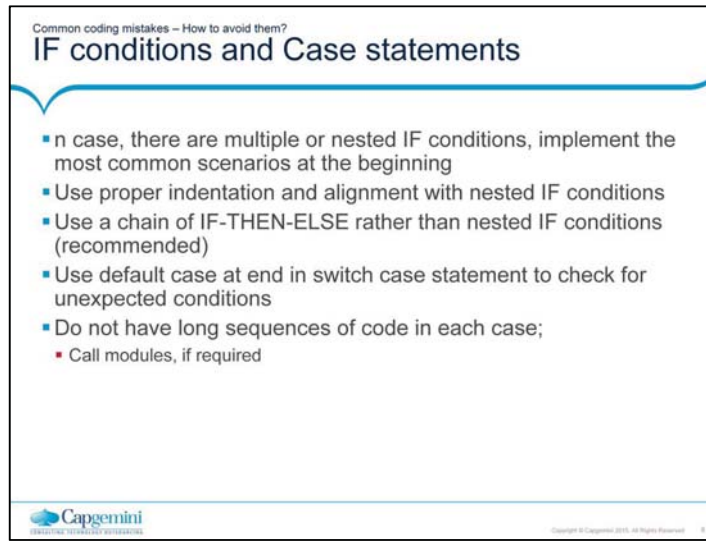


Use one variable for only one purpose.

For an example, if a variable is used for multiple different what can we provide? Consider a variable that is used to count the number of students, and count their grades. How would you name this variable: count, studentCount/gradeCount?

So create one variable for one purpose, otherwise data management related to the requirement will be complex.

Don't use variable with hidden meanings because it might not be understandable by others.

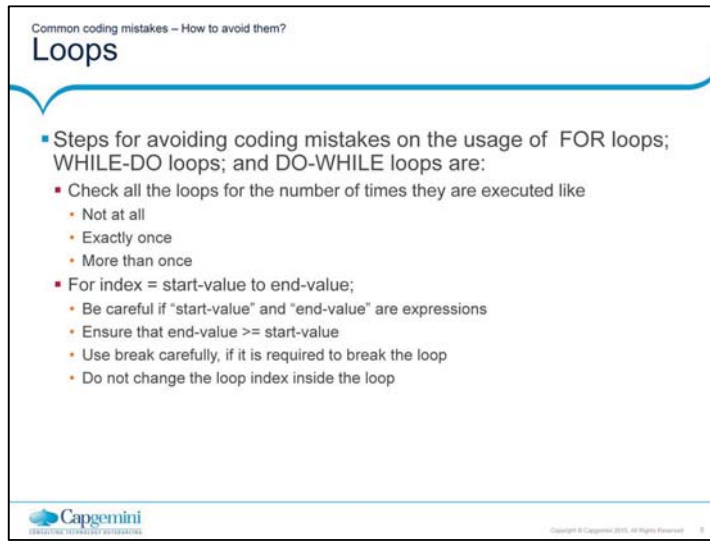


How to avoid Common Coding Mistakes while using IF and case statements:

If there are multiple nested IF conditions, then find out the condition which has high priority and specify the same in first if condition.
Adopt a standard indentation style for your code, and stick with it throughout the program so that program will be easily readable by other programmers.
For an Example:

```
BEGIN
    DECLARE file AS FILE
    PROMPT "Enter the filename" AND STORE IN file
    IF (theFileExits) THEN
            determine the length of the file
             IF(FileLength>o) THEN
                 readFile(file); // Invoke readFile module
             ELSE
                 PRINT "File doesn't contain data"
              END IF
    ELSE
                 PRINT "File doesn't exist"
    END IF
END
```

In the above code, indentation is followed, if-else is used and fileexists condition is checked first before finding length of a file for avoiding common mistakes.



Steps for avoiding Common Coding Mistakes while using Loops are:

Check for the number of times the loop executed: For an example, execute the below loop thrice, by considering num values as 5, 4 and 2.

```
WHILE (num<5)
DO
 ....
END WHILE
```

The above loop will be executed once for the value of 4, and the loop will not be executed at all for the num value of 5 and the same loop will be executed more than once if num value is 2.

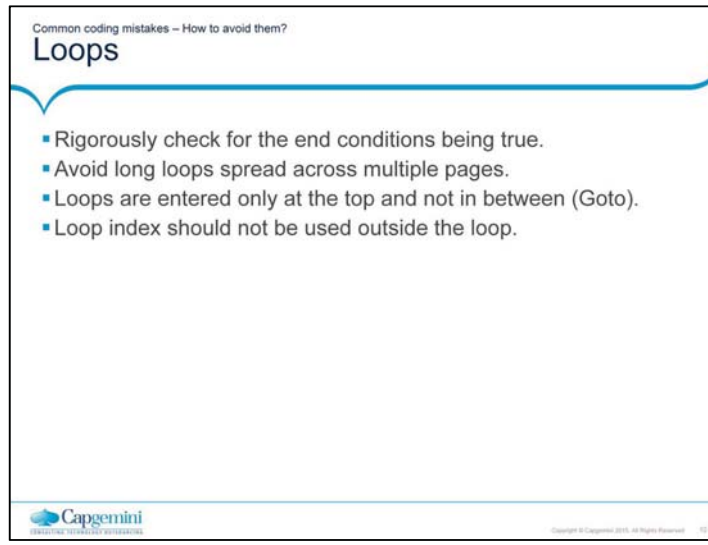
For loop:

Ensure that endvalue>=startvalue.

Better version	Not recommended to be used
FOR index= 0 to 10	FOR index=10 to 0
..	...
END LOOP	END LOOP

Do not change the loop index within the loop as shown in the below example. In the below example, loop execution will be stopped after the re-initialization of loop index inside the loop.

```
FOR index = 0 to 10
      index=13
END LOOP
```



Steps for avoiding Common Coding Mistakes while using Loops are:

Rigorously check for the end conditions being true : Rigorously avoid coding infinite loops.

Avoid long loops spread across multiple pages : The loop should be short enough to view all loop code at once.

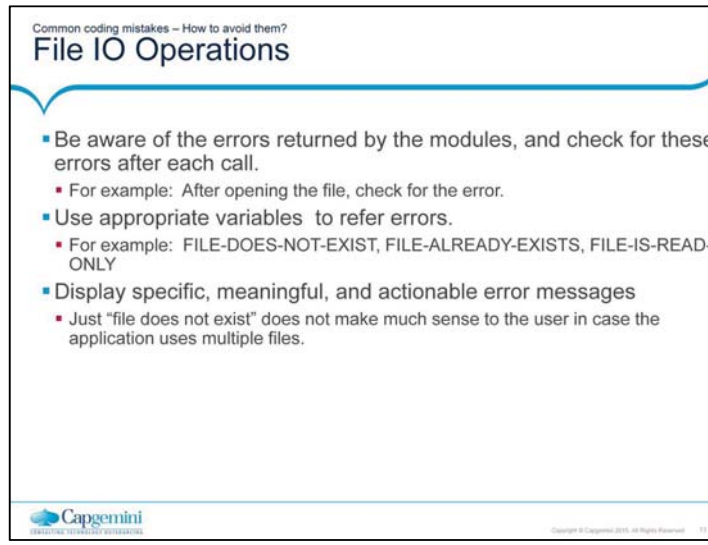
Loops are entered only at the top and not in between (Goto) : “goto” statement is always not recommended to be used inside loop as the loop execution always re start from the lower limit.

Loop index should not be used outside the loop. Consider the below code

```
FOR index=1 to 5
END LOOP
index=3;
```

Don’t use index outside of the loop. Use another variable, if required as shown below:

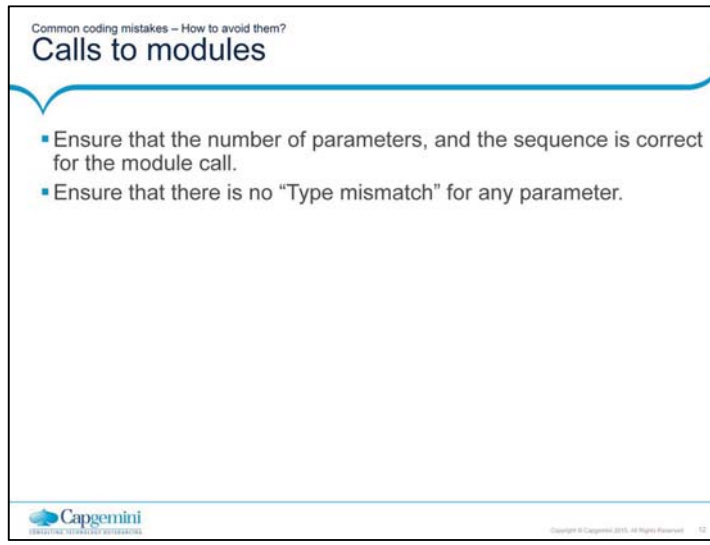
```
FOR index=1 to 5
END LOOP
num=3;
```



Points to be considered for avoiding Common Coding Mistakes while performing file IO operations are:
Be aware of the errors returned by the modules, and check for these errors after each call like

Check for the error, if file doesn't exists
After opening the file, check for the error.
Check for the error, if file size is zero.
Check for the error, if file is not readable/writable.

Use appropriate variables to refer errors. For example: FILE-DOES-NOT-EXIST, FILE-ALREADY-EXISTS, FILE-IS-READ-ONLY
Display specific, meaningful, and actionable error messages. For an Example, Just "file does not exist" does not make much sense to the user in case the application uses multiple files. Instead use the error message as "Employee.txt file does not exist", considering "Employee.txt" is a filename.



Consider the below signature of a module to calculate total price.

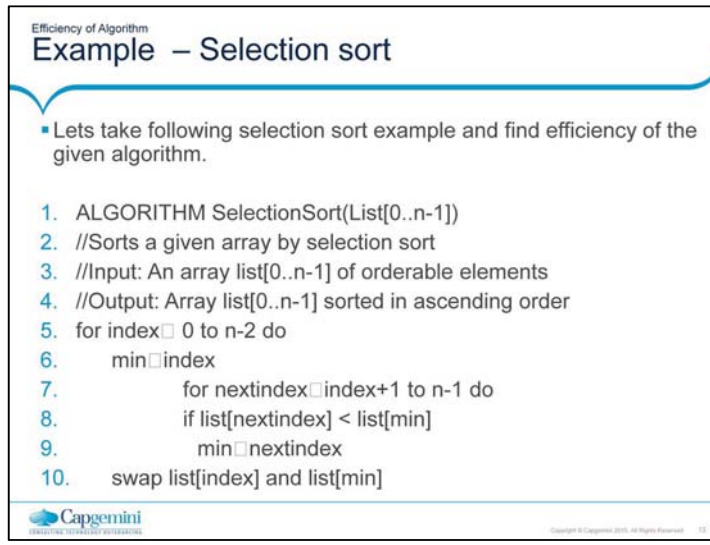
calculateTotal(Integer price, Integer quantity)

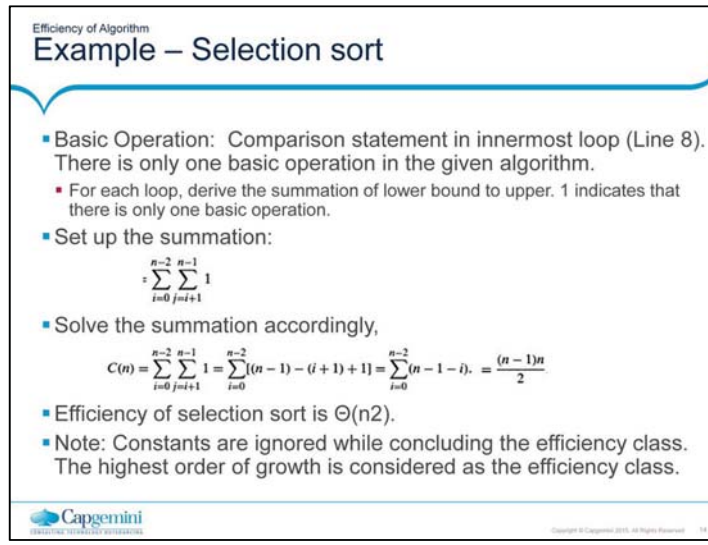
Refer the valid and invalid statements to invoke a module

calculateTotal(3,5); //Valid

calculateTotal(4,3,4); //Invalid

calculateTotal('Test',3); //Invalid

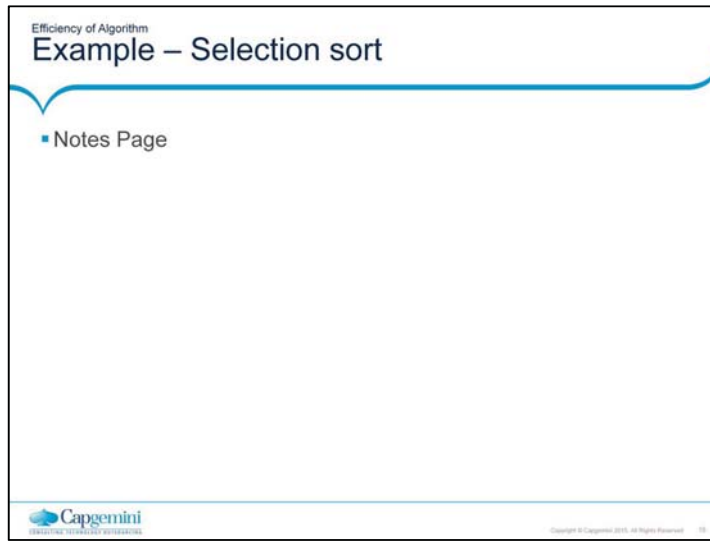

Efficiency of Algorithm
Example – Selection sort
Lets take following selection sort example and find efficiency of the given algorithm.
1. ALGORITHM SelectionSort(List[0..n-1])
2. //Sorts a given array by selection sort
3. //Input: An array list[0..n-1] of orderable elements
4. //Output: Array list[0..n-1] sorted in ascending order
5. for index 0 to n-2 do
6. min index
7. for nextindex index+1 to n-1 do
8. if list[nextindex] < list[min]
9. min nextindex
10. swap list[index] and list[min]
Capgemini



How it works?

Basic Operation: Comparison statement in the innermost loop. Summation of lower bound to upper bound. For each for loop derive the summation. Hence as given algorithm has 2 loops, we end up with 2 summations. 1 indicates that there is only one basic operation. By this we get the expression.

Now solving the above expression with proper formula's, we end up with (n-1)n/2, which results in (n2-n)/2. Highest order of growth from the expression will be considered as the efficiency class. Hence we say, efficiency is O(n2). If we have a series of statements within a loop, it would not matter because we are considering the basic operation which will take care of finding the highest order of growth. And we are only interested in highest order of growth, neither lesser order nor constants.



[For easy understanding lets use index as i and nextindex as j in solving the efficiency]
Summation formula used:

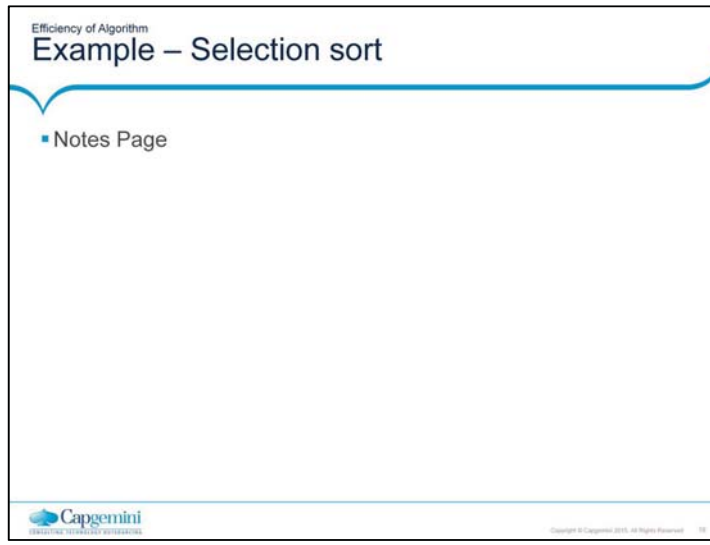
$$\sum_{i=l}^{u} 1 = u-l+1 \quad \text{where } l<=u \text{ are lower and upper integer limits.}$$

$$\sum_{i=0}^{n} i = \sum_{i=1}^{n} i = 1+2+3+\ldots..+n=n(n+1)/2 \approx (1/2)n2 \in O(n2)$$

Solution in detail:

$$=\sum_{i=0}^{n-2} (n-1-i)$$

$$=\sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i$$



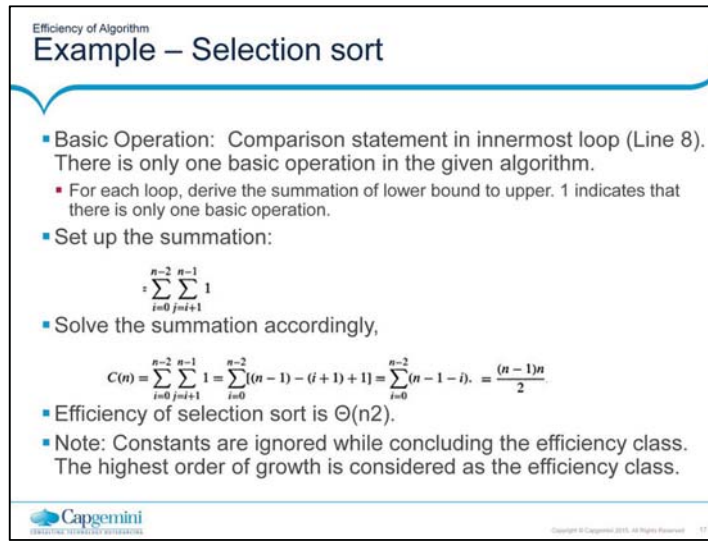
$$=(n-1)\sum_{i=0}^{n-2} 1 - \sum_{i=0}^{n-2} i$$

$$=(n-1)\sum_{i=0}^{n-2} 1 - (n-2)(n-1)/2$$

$$=(n-1)(n-1) – (n-2)(n-1)/2$$

$$=(n-1)((n-1) – (n-2)/2)$$

$$=(n-1)(2n-2-n+2)/2$$
$$=(n-1)(n)/2$$
$$=(n2-n)/2$$
$$=n2/2-n/2$$
$$\in O(n2)$$



How it works?

Basic Operation: Comparison statement in the innermost loop. Summation of lower bound to upper bound. For each for loop derive the summation. Hence as given algorithm has 2 loops, we end up with 2 summations. 1 indicates that there is only one basic operation. By this we get the expression.

Now solving the above expression with proper formula's, we end up with (n-1)n/2, which results in (n2-n)/2. Highest order of growth from the expression will be considered as the efficiency class. Hence we say, efficiency is O(n2). If we have a series of statements within a loop, it would not matter because we are considering the basic operation which will take care of finding the highest order of growth. And we are only interested in highest order of growth, neither lesser order nor constants.

[For easy understanding lets use index as i and nextindex as j in solving the efficiency]

Summation formula used:

$\sum_{i=l}^{u} 1 = u-l+1$ where l<=u are lower and upper integer limits.

$\sum_{i=0}^{n} i = \sum_{i=1}^{n} i = 1+2+3+\ldots..+n=n(n+1)/2 \approx (1/2)n^2 \in O(n^2)$

Solution in detail:

$$= \sum_{i=0}^{n-2} (n-1-i)$$

$$= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i$$

$$= (n-1) \sum_{i=0}^{n-2} 1 - \sum_{i=0}^{n-2} i$$

$$= (n-1) \sum_{i=0}^{n-2} 1 - (n-2)(n-1)/2$$

$$= (n-1)(n-1) - (n-2)(n-1)/2$$

$$= (n-1)((n-1) - (n-2)/2)$$

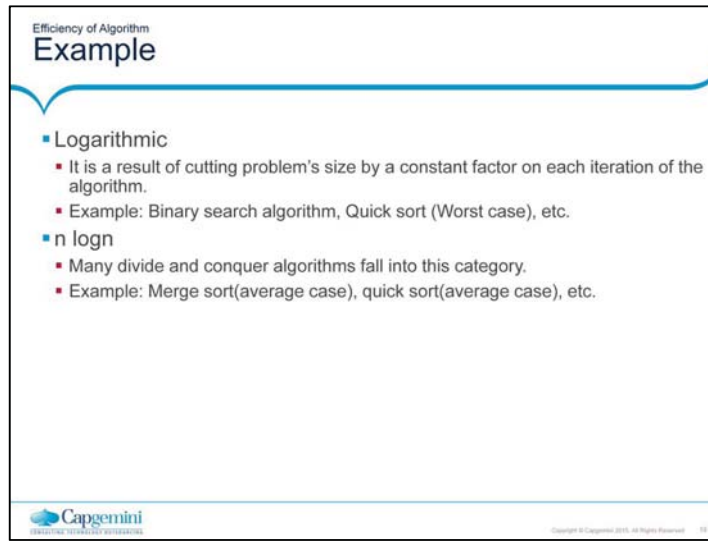
$$= (n-1)(2n-2-n+2)/2$$

$$= (n-1)(n)/2$$

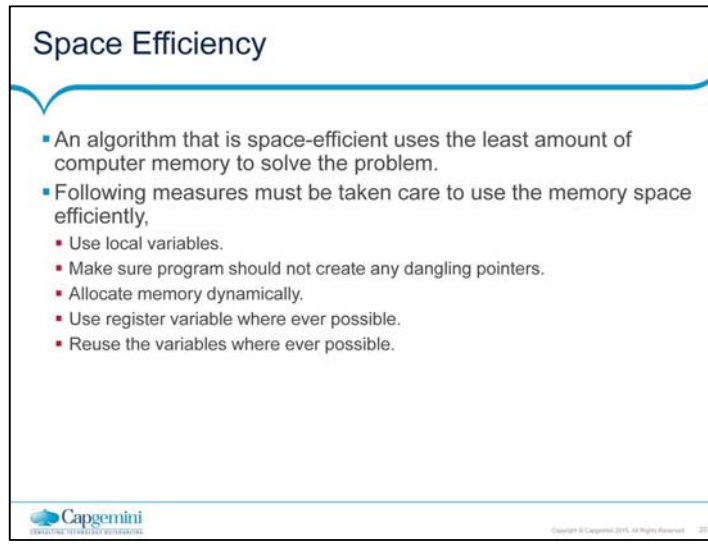
$$= (n^2-n)/2$$

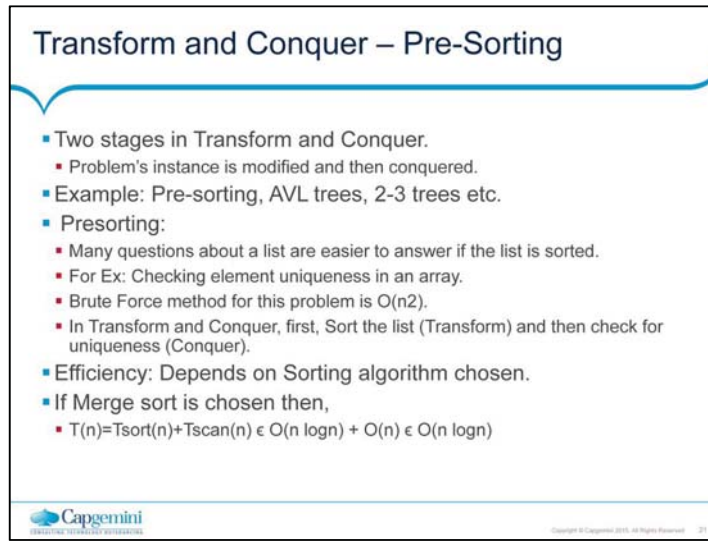
$=n^2/2-n/2$

$\in O(n^2)$



After constant, log n is the next best efficiency class. Logarithmic algorithm will not take all into account. Any algorithm that does so will at least have Linear running time. Examples of algorithms which fall under this category are Binary search, Quick sort etc.

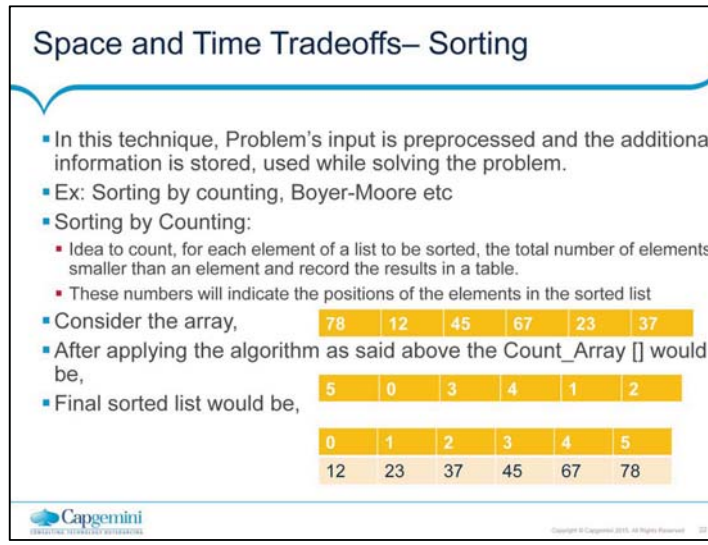

Space Efficiency
An algorithm that is space-efficient uses the least amount of computer memory to solve the problem.
Following measures must be taken care to use the memory space efficiently,
Use local variables.
Make sure program should not create any dangling pointers.
Allocate memory dynamically.
Use register variable where ever possible.
Reuse the variables where ever possible.
Capgemini



This technique is based on the idea of transformation. These methods work as two-stage procedures. First, in the transformation stage, the problem's instance is modified to be, for one reason or another, more amenable to solution. Then, in the second or conquering stage, it is solved.
Pre-sorting as an example is discussed in detail in the above given slides. Many questions about a list are easier to answer if the list is already sorted. One such example is the problem to find element uniqueness. This can be easily solved if the list/array is already sorted. Hence sorting the array is transformation to the simpler instance and then solving the problem.

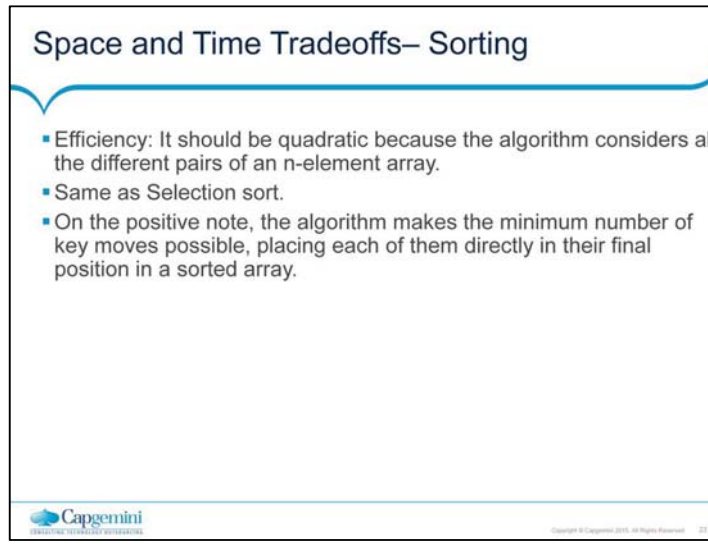
The brute-force algorithm compares pairs of the array's elements until either two equal elements were found or no more pairs were left. Its worst-case efficiency would be O(n2).
Alternatively, we can sort the array first and then check only its consecutive elements: if the array has equal elements, a pair of them must be next to each
other, and vice versa.

ALGORITHM ElementUniqueness(List[0..n − 1])
//Solves the element uniqueness problem by sorting the array first
//Input: An array List[0..n − 1] of orderable elements
//Output: Returns "true" if List has no equal elements, "false" otherwise
sort the array List
for index←0 to n − 2 do
if A[index]= A[index + 1] return false
return true



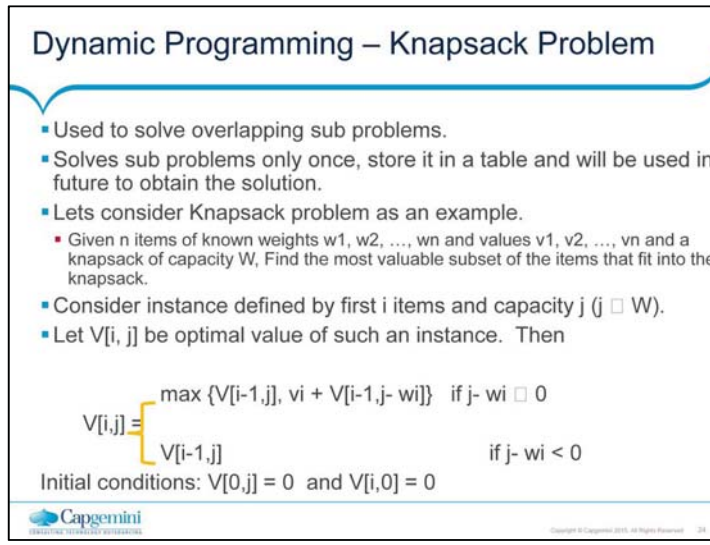
In this technique, Problem's input is preprocessed and the additional information is stored, used while solving the problem. Idea to count, for each element of a list to be sorted, the total number of elements smaller than an element and record the results in a table. These numbers will indicate the positions of the elements in the sorted list
e.g., if the count is 10 for some element, it should be in the 11th position (with index 10, if we start counting with 0) in the sorted array. Thus, we will be able to sort the list by simply copying its elements to their appropriate positions in a new, sorted list.
Algorithm for your reference is given below.

```
ALGORITHM ComparisonCountingSort(List[0..n − 1])
//Sorts an array by comparison counting
//Input: An array List[0..n − 1] of orderable elements
//Output: Array Sort[0..n − 1] of A's elements sorted in nondecreasing order
for index←0 to n − 1 do Count[index]←0
for index ←0 to n − 2 do
for nextindex←index + 1 to n − 1 do
if List[index]<List[jnextindex]
Count[nextindex ]←Count[nextindex]+ 1
else Count[index]←Count[index]+ 1
for index ←0 to n − 1 do Sort[Count[index]]←List[index]
return Sort
```



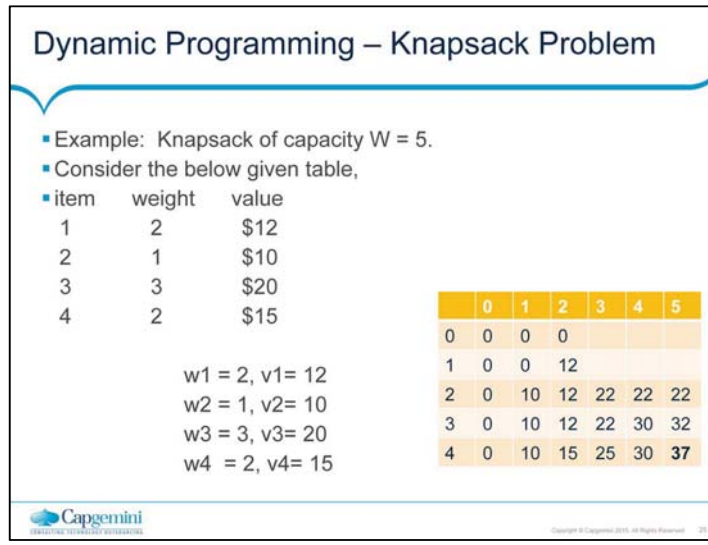
e.g., if the count is 10 for some element, it should be in the 11th position (with index 10, if we start counting with 0) in the sorted array. Thus, we will be able to sort the list by simply copying its elements to their appropriate positions in a new, sorted list.

```
ALGORITHM ComparisonCountingSort(List[0..n − 1])
//Sorts an array by comparison counting
//Input: An array List[0..n − 1] of orderable elements
//Output: Array Sort[0..n − 1] of A's elements sorted in nondecreasing order
for index←0 to n − 1 do Count[index]←0
for index ←0 to n − 2 do
for nextindex←index + 1 to n − 1 do
if List[index]<List[jnextindex]
Count[nextindex ]←Count[nextindex]+ 1
else Count[index]←Count[index]+ 1
for index ←0 to n − 1 do Sort[Count[index]]←List[index]
return Sort
```



Dynamic programming is a technique for solving problems with overlapping subproblems. Rather than solving overlapping subproblems again and again, dynamic programming suggests solving each of the smaller subproblems only once and recording the results in a table from which a solution to the original problem can then be obtained.

Knapsack Problem: Given n items of known weights w1, w2, . . . , wn and values v1, v2, . . . , vn and a knapsack of capacity W, find the most valuable subset of the items that fit into the knapsack. Brute force approach for this problem leads to O(2n).

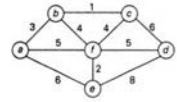


Using the formula given, compute and fill the table. After computing all values, the maximal value is F(4, 5) = $37.
Optimal solution is {1, 2, 4}. We can find the composition of an optimal subset by backtracing the computations of this entry in the table.

Since F(4, 5) > F(3, 5), item 4 has to be included in an optimal solution along with an optimal subset for filling 5 − 2 = 3 remaining units of the knapsack capacity. The value of the latter is F(3, 3). Since F(3, 3) = F(2, 3), item 3 need not be in an optimal subset. Since F(2, 3) > F(1, 3), item 2 is a part of an optimal selection, which leaves element F(1, 3 − 1) to specify its remaining composition. Similarly, since F(1, 2) > F(0, 2), item 1 is the final part of the optimal solution {item 1,item 2, item 4}.
The time efficiency and space efficiency of this algorithm are both in O(nW). The time needed to find the composition of an optimal solution is in O(n).

Greedy Technique – Kruskal's Algorithm

- It constructs the solution through a sequence of steps.
- Example - Kruskal's Algorithm.
  - Algorithm begins by sorting the graph's edges in non decreasing order of their weights.
  - Start with empty sub graph.
  - Scan the sorted list and add next edge on the list to the current sub graph. If the edge is creating a cycle then simply skip the edge.
- Consider the following graph,



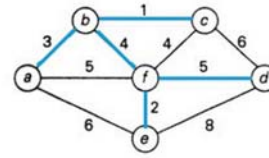
- Sorted Edge List: (bc,1) (ef,2) (ab,3) (bf,4) (cf,4) (af,5) (df,5) (ae,6) (cd,6) (de, 8)

Capgemini

The greedy approach suggests constructing a solution through a sequence of steps, each expanding a partially constructed solution obtained so far, until a complete solution to the problem is reached. Kruskal's algorithm looks at a minimum spanning tree of a weighted connected graph G = V, E as an acyclic subgraph with |V| − 1 edges for which the sum of the edge weights is the smallest.
Aspanning tree of an undirected connected graph is its connected acyclic subgraph (i.e., a tree) that contains all the vertices of the graph. If such a
graph has weights assigned to its edges, a minimum spanning tree is its spanning tree of the smallest weight, where the weight of a tree is defined as the sum of the weights on all its edges. The minimum spanning tree problem is the problem of finding a minimum spanning tree for a given weighted connected graph.

Greedy Technique – Kruskal's Algorithm

- Sub tree which forms minimum spanning tree is according to Kruskal's algorithm is
  - (bc,1) (ef,2) (ab,3) (bf,4) (df,5)
- Below figure shows the subgraph which is minimum spanning tree.



Capgemini

Algorithm begins by sorting the graph's edges in non decreasing order of their weights.
Start with empty sub graph.
Scan the sorted list and add next edge on the list to the current sub graph. If the edge is creating a cycle then simply skip the edge.

Programming Foundation with Pseudocode - Lab Book

©2016 Capgemini. All rights reserved.
The information contained in this document is proprietary and confidential. For Capgemini only.

Document Revision History

Date	Revision No.	Author	Summary of Changes
Jan 2015	1.0	Rathnajothi Perumalsamy	Initial Draft
May 2015	1.1	Shraddha P Jadhav	Added assignments related to algorithm design and techniques.
May 2016	1.2	Anjulata Tembhare	Removed some assignments as per new ToC

©2016 Capgemini. All rights reserved.
The information contained in this document is proprietary and confidential. For Capgemini only.

Table of Contents

©2016 Capgemini. All rights reserved.
The information contained in this document is proprietary and confidential. For Capgemini only.

Getting Started

Overview

This lab book is a guided tour for learning Programming Foundation with Pseudocode. It comprises 'To Do' assignments. Follow the steps provided and work out the 'To Do' assignments given.

Setup Checklist for Programming Foundation with Pseudocode for All LOTs

Here is what is expected on your machine in order for the lab to work.

Minimum System Requirements

- Intel Pentium 90 or higher (P166 recommended)
- Microsoft Windows 95, 98, or NT 4.0, 2k, XP, 7.
- Memory: 32MB of RAM (64MB or more recommended)

Please ensure that the following is done:

- An editor like Notepad/Notepad ++ is installed.

Instructions

- For all instructions on Programming Foundation lab assignment refer Appendix A. All lab assignments should refer these set of instructions.
- Create a directory by your name in drive <drive>. In this directory, create a subdirectory programming_foundation_assgn. For each lab exercise create a directory as lab <lab number>.

©2016 Capgemini. All rights reserved.
The information contained in this document is proprietary and confidential. For Capgemini only.

Lab 1. Basic program development with pseudocode

Goals	Test the basic programming concept.
Time	4 Hours

1 Write a pseudocode to accept 2 numbers and find the difference between those numbers.

Solution:
Step 1: Open notepad, write the below pseudocode and save the file as "Difference.txt"

```
BEGIN
        DECLARE num, num1 AS INTEGER
        PROMPT "Enter 2 numbers" AND STORE IN num, num1
        PRINT "The result is", num1-num2;
END
```

2 Write a pseudocode to accept 10 numbers and find maximum number among 10 numbers.

Solution:

```
BEGIN
    DECLARE numbers[10] AS INTEGER ARRAY
    DECLARE max AS INTEGER
    INITIALIZE max TO 0
        PROMPT "Enter 10 numbers"
    FOR index=1 TO 10
        ACCEPT numbers[index]
    END FOR
    max=numbers[0]
    FOR index=1 TO 10
        IF numbers[index] > max THEN
            max=numbers[index]
        END IF
    END FOR
    PRINT max
END
```

Assignments: <<TODO>>

1.1 Rima is working in State Electricity Board Project. She got the following requirement. The following information has to be accepted from the user to calculate the net electricity bill amount.

- User ID
- User Name
- Last month meter reading
- Current month meter reading

Unit consumed = (Last month meter reading) – (Current month meter reading)
Net Amount = Unit consumed * 1.15 +Fixed Charge
Assume Fixed Charge is always Rs.100.
Write pseudo code to print the electricity bill. The bill should be in the following format
User ID:
User Name:

©2016 Capgemini. All rights reserved.
The information contained in this document is proprietary and confidential. For Capgemini only.

Unit Consumed:
Net amount:

1.2 Organization employees are recognized with different tags based on their experience.

Years of Experience	Tag Color
0 - <3	Blue
3 - <5	Grey
5 - <10	Yellow
>10	Red

Write pseudo code to accept the experience and display their tag Color.

1.3 Write pseudo code to print the following mathematical series.
0 1 1 2 3 5 8 13 21 …. N. Where N is accepted from the user.

1.4 Write a pseudo code to accept a number and check whether a given number is an Armstrong number. For example, 371 is an Armstrong number since (3*3*3) + (7*7*7) + (1*1*1) = 371.

1.5 Write a pseudo code to convert a binary number to decimal. For an Example, if the given input is 00000100(binary number), then the output should be 4(Decimal number).

1.6 Write a pseudo code to accept 10 numbers in an array and do the following using a loop.

1.6.1 Display the smallest number.
1.6.2 Display all ODD and EVEN numbers separately

1.7 Modify the below Pseudocode to implement good programming practices. The below Pseudocode is used to calculate total price of a product including tax.

```
BEGIN
    Print "Enter price of your product"
    Accept p
    tc=p*.56
    Print "Total price of product is":  tc
END
```

Hint: .56 is the Tax rate.

©2016 Capgemini. All rights reserved.
The information contained in this document is proprietary and confidential. For Capgemini only.

Lab 2. Implementation of good programming practices

Goals	Test the concept of applying the characteristics of good programming practices like modularity approach.
Time	3 Hours

1 Write a pseudocode to accept 2 numbers and find the difference between two given numbers

Solution:

```
BEGIN
                    DECLARE num, num1 AS INTEGER
                    PROMPT "Enter 2 numbers" AND STORE IN num, num1
                    difference (num,num1);
 END
 SUB difference (num1, num2)
                    PRINT "The result is  " , num1-num2;
 END SUB
```

Assignments: <<TODO>>

2.1 Modify all the pseudo code written in lab 1 to ensure that the code adheres to good programming practices such as readability and maintainability.

2.2 Write a module in pseudo code to accept a number and return the sum of its digits. Invoke the user defined module in main program. For an example: if a user enters input as 12, then the output will be 3.

2.3 Write a module in pseudo code to accept an array with 10 numbers and return number of unique values in an array. For an example, if the given input array contains values such as {10,22,13,22,44,6,24,44,77,8}, then the output should be 6.

2.4 Refactor the below given code. The below pseudocode is used for calculating total leaves applicable per year for an employee. Number of Leaves added to an employee account differs based on the type of employment. For an Example, 2 days leave will be added per month for a permanent employee and 1 day leave will be added per month for a contract based employee

```
RECORD Employee
    DECLARE EmpId as INTEGER
    DECLARE employmentType AS STRING
END RECORD

BEGIN
                DECLARE emp AS Employee
          //leaves variable is used to store number of leaves per month
          DECLARE leaves, TotalLeaves AS INTEGER
          FOR index= 1 to 5
                PROMPT "Enter the EmployeeId" AND STORE IN
     emp.EmpId
                PROMPT "Enter the employmentType" AND STORE IN
  emp.employmentType
          END FOR
```

©2016 Capgemini. All rights reserved.
The information contained in this document is proprietary and confidential. For Capgemini only.

```
                    FOR index= 1 to 5
                        IF(employmentType=='PERMANENT') THEN
                            leaves=2;
                            TotalLeaves=leaves*12;
                        ELSE IF(employmentType=='CONTRACT') THEN
                            leaves=1;
                            TotalLeaves=leaves*12;
                        END IF
                            PRINT "Employee Id :" , emp.EmpId
                          PRINT "Total Available Leaves are :", TotalLeaves
                    END FOR
                    Index2=5;
END
```

©2016 Capgemini. All rights reserved.
The information contained in this document is proprietary and confidential. For Capgemini only.

Lab 3. Algorithm Analysis and Design

Goals	Algorithm analysis and Test the concept of searching and sorting algorithms.
Time	3 Hours

Assignments: <<TODO>>

3.1 **ALGORITHM Check(A[0..n-1])**

```
//Input: An array A[0..n-1] of n real numbers
x← A[0]; y←A[0]
for i←1 to n-1 do
    if A[i]<x
        x← A[i]
    if A[i]>y
        y← A[i]
return y-x
```

For the algorithm given above,

- What does this algorithm computes?
- What is its basic operation?
- Check the no of times the basic operation is executed depends only on the size of an input or on any other parameter.
- Set up a sum/recurrence, expressing the number of times the basic operation is executed.
- What is the efficiency class of the given algorithm?

3.2 Determine the output of below given algorithm by tracing it for the taking any sample input.

ALGORITHM surprise(A[0..n-1])

```
//Input: An array A[0..n-1] of real numbers
If n=1
    return A[0]
else
    t←surprise(A[0..n-2])
    if t<=A[n-1]
        return t
    else
        return A[n-1]
```

3.3 Find the time efficiency class of the following algorithm,

```
ALGORITHM Find(Matrix[0..n-1,0..n])
//Input: An n by n+1 matrix Matrix[0..n-1,0..n] of real numbers
for index← 0 to n-2 do
    for nextindex← index+1 to n-1 do
            for tempindex← index to n do

                Matrix[nextindex,tempindex]←Matrix[nextindex,tempindex]-
            Matrix[index,tempindex] * Matrix[nextindex,index] / Matrix[index,index]
```

3.4 Consider the following Algorithm,

©2016 Capgemini. All rights reserved.
The information contained in this document is proprietary and confidential. For Capgemini only.

```
ALGORITHM Sum(n)
//Input: A nonnegative integer n
sum←0
for index←1 to n do
      sum←sum+i
return sum
```

- What does the above algorithm compute?
- What is the efficiency class of the given algorithm?

3.5 Write a module in pseudo code with 2 parameters: first parameter is a word, the second is a character. The function must return the number of times the character is found in the word.

3.6 Write a Pseudocode to accept 5 numbers in an array and sort the array if array is not sorted and search for a number in the array using binary search. Implement sorting and searching logic in module.

<<Stretched Assignments>>

3.7 Write a module in pseudo code to accept 2 strings and return success message if second string is a substring of first string. For Example: If a user enters first string as “Testing” and second string as “Test”, then “Substring found in source string” message should be displayed.

©2016 Capgemini. All rights reserved.
The information contained in this document is proprietary and confidential. For Capgemini only.

Lab 4. Exception Handling

Goals	Test the concept of handling exceptional scenarios.
Time	1.5 Hours

4.1 Write a pseudo code for calculating price after applying discount in which exceptions handlers are used.

Solution:

```
/*****************************************************************************
 * File                         : DefensiveProgramming.txt
 * Author Name                          : Capgemini
 * Desc                         : Program to apply discount on productprice
 * Version                      : 1.0
 * Last Modified Date           : 8-May-2016
 * Change Description           : Description about the changes implemented
 *****************************************************************************/

BEGIN
 DECLARE productId AS INTEGER
 DECLARE discount AS REAL
 PROMPT "Enter productId" AND STORE IN productId
 IF(isValid(productId)) THEN
     PROMPT "Enter discount" AND STORE IN discount
     IF(isValid(discount)) THEN
             applyDiscount(productId,discount);
     ELSE
             PRINT"Discount value should contain numbers"
     END IF
 ELSE
     PRINT"Product Id should contain numbers"
 END IF
END

/*****************************************************************************
* Module Name                  : applyDiscount()
* Input Parameters    : productId, discount
* Return Type         : INTEGER
* Author              : Capgemini
* Creation Date       : 8-May-2016
* Description         : Applying discount on the product price
 *****************************************************************************/

SUB applyDiscount(productId,discount)
 DECLARE result AS INTEGER
 result=getProductPrice(productId)*discount
 PRINT "Product Price" + result;
EXCEPTION
 WHEN NoSuchElement THEN
    PRINT errormessage //Errormessage returned from exception
END SUB

/*****************************************************************************
* Module Name                  : getProductPrice()
* Input Parameters    : productId
```

©2016 Capgemini. All rights reserved.
The information contained in this document is proprietary and confidential. For Capgemini only.

```
* Return Type        : INTEGER
* Author             : Capgemini
* Creation Date      : 8-May-2016
* Description        : Based on productId, fetching product price if productId exists,
else exception will be raised
 *****************************************************************************/

SUB getProductPrice(productId)
 DECLARE errorcode AS INTEGER AND STORE 0
 IF(elementfound(productId)) THEN
      RETURN productPrice
 ELSE
    RAISE "Product doesn't exist with this id"+ productId
 END IF
END SUB

/*****************************************************************************
* Module Name                : isValid()
* Input Parameters   : data
* Return Type        : BOOLEAN
* Author             : Capgemini
* Creation Date      : 8-May-2016
* Description        : To validate data for accepting only digits
*****************************************************************************/

SUB isValid(data)
 IF(isDigits(data)) THEN
    RETURN true
 ELSE
    RETURN false
 END IF
END SUB
```

Assignments: <<TODO>>

4.2 Take care of necessary validations in 2.4 assignment in order to avoid exceptional scenarios

4.3 Modify the below Pseudocode to implement good programming practices and take care of necessary validations to avoid exceptional scenarios. The below Pseudocode is used to calculate total price of a product including tax.

```
BEGIN
    Print "Enter price of your product"
    Accept p
    Print "Enter quantity of your product"
    Accept q
    tc=p*q*.56
    Print "Total price of product is":  tc
END
```

Hint: .56 is the Tax rate.

©2016 Capgemini. All rights reserved.
The information contained in this document is proprietary and confidential. For Capgemini only.

Lab 5. Software Review and Testing

Goals	Test the concept of reviews and testing
Time	1.5 Hours

Assignments: <<TODO>>

5.1 Do self and peer review of the pseudo code for previous lab assignments 1.2, 2.4 and 3.4 using the review check list and record the review comments in the checklist.

5.2 Write test cases for Lab assignment 1.2, 2.4 and 3.7 using the test case template

©2016 Capgemini. All rights reserved.
The information contained in this document is proprietary and confidential. For Capgemini only.

Appendices

Appendix A: Programming Standards

Instructions for lab assignments:

Compile the list of learning's from the previous day.

a. Maintain the list of steps required to complete each type of task. For example the following steps are required to develop a new piece of code:

   i. Study the Specifications or Problem Definition (Analysis).
   ii. Develop the Program Design or Pseudo Code (Design).
   iii. Develop the Black Box Test cases in the specified format (Testing).
   iv. Develop the Documented Code (Coding).
   v. Develop the White Box Test cases in the specified format (Testing).
   vi. Perform the Code Review - Self and/or Peer (Reviews).
   vii. Note the Number and Category of defects found.
   Categories are: Standards (Program output not affected), Logic (Program output would differ)
   viii. Compile and remove compilation errors (Testing).
   ix. Test and log all defects found (Testing).
   x. Debug and fix all defects found (Debugging).
   xi. Repeat steps xi and xii, as required. However, note the number of iterations.
   xii. Note the Time / Efforts spent on each step above. The item in brackets indicates the category against which Time / Efforts should be logged.
   xiii. Compare the Actual Time against the estimated time. Note down reasons for significant variations.

b. Maintain a check-list of "Things-To-Do", or "Things-To-Check-for" (includes "Things-NOT-To-Do"). Use this check-list during Code Review and Test Case Review. Revise the check-list every day based on the learnings of the previous day.

c. For all assignments, the test cases should be created in the following format by using an EXCEL sheet:

   i. Requirement Id #, Test Case Id #, Test condition, Test cases, Test data, Expected result, Remarks

d. Defects should be logged in the following format:

   i. Bug #, Failed test case #, Bug description, Bug Status, Fixed in Iteration #, Time/Efforts spent on debugging, Type of Error

e. Time / Efforts should be logged against the following categories:
   i. Analysis, Design, Coding, Reviews, Testing, and Debugging
   Check the % of time spent in different categories.

Sequence of assignments may be adjusted by the faculty based on performance of participants.

©2016 Capgemini. All rights reserved.
The information contained in this document is proprietary and confidential. For Capgemini only.