Basic Functions of a Computer

- perform arithmetic operations (add, subtract, multiply, divide, etc.)
- perform relational (comparison) operations (less than, greater than, equal to, etc.)
- perform logical operations (and, or, not)
- perform input/output operations (get, read, put, display, etc.)
- move data between memory and registers
- computers accept input, do some processing, and produce output input / process / output

Programming Process

1. **Understand** the problem
2. **Plan** the logic
3. **Code** a source program using a high-level language
4. **Translate** the source program
5. **Test** the program using test data (don't forget this step!)

File Processing Data Hierarchy

- file a collection of records
- record a collection of fields
- field a collection of characters or digits that make up a data item (name, number, description, etc.)

Program Translation

- **Compiler** Translate a source program to machine language OR an intermediate language in a single step
  Examples: source code → executable code, source code → intermediate language code
- **Interpreter** Step-by-step translation and execution of a program. Python is an interpreter.
  The Java Virtual Machine (JVM) is an interpreter that translates and executes bytecode.
  The JVM use a J-I-T (just-in-time) compiler to speed up the translation process.

Programming Styles

- Procedural / Object-oriented / GUI / Batch / Interactive

Fundamental Control Structures

- Sequence, Selection (aka Decision), Loop (aka repetition and iteration)
- Control structures can be stacked and nested
- Every control structure has a single entry point and a single exit point
- Control structures are processed sequentially (in sequence)
- Statements implement control structures in pseudocode and in high-level languages
- Every symbol you use in RAPTOR is a control structure
- Distinguish between a sequence structure and sequence

Flowchart Symbols

- **terminal** often described as a capsule, lozenge, or racetrack
- **process** rectangle used for assignment statement
- **predefined process** or **aka module / method / procedure / function**
- **input/output** parallelogram
- **decision** diamond used for both selection and looping
- **connector** use this to connect flow lines
- **Annotation** use this to put comments in a flowchart
- **flowlines**
**RAPTOR Flowchart Symbols**

The **Assignment** symbol is used to declare and initialize variables or assign them different values.

The **Call** symbol is used to call built-in RAPTOR functions or to call a user-defined module (flowchart in a different tab).

The **Input** symbol displays a prompt and stores user input in a variable.

The **Output** symbol allows output to include a newline (default) or this can be unchecked to keep the cursor on the same line in the Master Console or the file being written to.

The **Selection** symbol is used to code a selection control structure.

The **Loop** symbol allows you to create a pre-test, mid-test, or post-test loop. Note: Many programming languages only directly support pre- and post-test loops.

RAPTOR also uses **Terminal** symbols but they are always provided automatically, can’t be modified, and use the text **Start** and **End**.

---

**RAPTOR Coding Modes**

RAPTOR has 3 modes for coding:
- **Novice**: simplest mode – no procedures allowed, all variables are global variables
- **Intermediate**: permits creating procedures which can take parameters and declare local variables
- **Object-Oriented**: appropriate for object-oriented program design

**RAPTOR Modules**

- RAPTOR has built-in functions and procedures, user defined procedures, and subcharts (named flowcharts)
- RAPTOR uses the words **Start** / **End** in all terminal symbols for a subchart. The Start word is followed by a parameter list in a user-defined procedure
- RAPTOR procedures may have parameters that use **in**, **out**, and **in out**, parameter passing mechanisms
- Modules are declared to perform a single task. A module is invoked when the task it performs is needed. RAPTOR uses a call symbol to invoke a module (subchart or procedure).

**RAPTOR Comments (internal descriptions / documentation)**

- Commenting in a flowchart is formally called "annotation". There is an annotation symbol that is used for this when manually drawing a flowchart. The symbol is shown above with the other flowchart symbols.
- RAPTOR uses a callout symbol to display a comment. Comments may be created for a symbol in RAPTOR by right-clicking the symbol and choosing “comment”

**RAPTOR Arithmetic Operators**

- +, -, *, /, MOD, ** or ^ (add, subtract, multiply, divide [floating-point division], integer remainder, exponentiation)
- MOD is used to get the integer remainder of division 23 MOD 6 is 5 (Java uses % to do this (modulus operator))

**RAPTOR Relational Comparison Operators**

- <, <=, >, >=, =, !=

**RAPTOR Logical Operators**

Perform logical operations which return a value of true or false

Precedence for binding of operators / operands: (in order from highest to lowest)
- not Logical NOT unary operator
- and Logical AND binary operator
- or Logical OR binary operator
RAPTOR Math Functions

Each function below returns a numeric value and can be used where ever a numeric value is appropriate. The examples below show some ways functions can be used. Of course, actual Raptor programs may use different functions in different ways to achieve the programer's desired functionality. Recall the <-- symbols means "is assigned". This would be done using the Assignment symbol in RAPTOR.

### ABS

variable <- abs(math_expression)

abs returns the absolute value of a mathematical expression. For example, abs(-3.7) is 3.7, and abs(23) is 23.

### CEILING

variable <- ceiling(math_expression)

ceiling returns the lowest integer value greater than or equal to the provided argument. For example, ceiling(15.9) is 16, ceiling(3.1) is 4, ceiling(-4.1) is -4, and ceiling(23) is 23.

### FLOOR

variable <- floor(math_expression)

floor returns the highest integer value less than or equal to the provided argument. For example, floor(15.9) is 15, floor(3.1) is 3, floor(-4.1) is -5, and floor(23) is 23.

### MAX

variable <- max(math_expression, max_expression)

max returns the maximum of the two provided arguments. For example, max(5,7) returns 7.

### MIN

variable <- min(math_expression, max_expression)

min returns the minimum of the two provided arguments. For example, min(5,7) returns 5.

### PI

radians <- degrees * (pi / 180)

pi returns the ratio of a circle's circumference to its diameter (approximately 3.14159). This is a constant function that takes no arguments and always returns the same value.

### RANDOM

variable <- random

random takes no arguments. The function returns a random number in the range [0.0,1.0). That is, it may sometimes return 0.0 but will never return 1.0. To generate a random integer from 1 to n, use floor((random*n) + 1). For example, you can simulate the roll of a die (random number from 1 to 6) with floor((random * 6) + 1).

### SQRT

variable <- sqrt(math_expression)

sqrt returns the square root of the provided argument. A negative argument results in a run-time error in the program.

### RAPTOR Built-in Procedures

RAPTOR has several built-in procedures that come in handy for working with formatted output and files:

- **End_Of_Input** - Use as the test in a Selection or Loop symbol. RAPTOR does not usually require a priming read.
- **Clear_Console** - clears the output master console
- **Redirect_Input**(yes/no or filename)  Redirect_Input(yes)  Redirect_Input("ClassData.txt")
- **Redirect_Output**(yes/no or filename)  Redirect_Output(no)  Redirect_Output("GradeReport.txt")
- **Set_Precision**(number of digits or -1)  Set_Precision(2)  Set_Precision(-1) //reset to default
RAPTOR Functions for formatting output

I have created some functions you can use to format your output. These require the file plugin_format.dll and also require the version of RAPTOR to be July 2016 or higher.

- **Get_Currency( value, size )** – value is the variable or literal (type floating-point) you want to display, size is the width of the field you want to display it in. The value is displayed right-justified with a leading $. If you do not specify a large enough field size for the value to be displayed it will use as many positions as it needs ignoring the size you entered.

- **Get_Number( value, size )** – value is the variable or literal (type integer) you want to display, size is the width of the field you want to display it in. The value is displayed right-justified. If you do not specify a large enough field size for the value to be displayed it will use as many positions as it needs ignoring the size you entered. If you try to use a floating-point value for the parameter value it will be truncated to an integer.

- **Pad_String( value, size )** – value is the variable or literal (type string) you want to display, size is the width of the field you want to display it in. The value is displayed left-justified. If you do not specify a large enough field size for the value to be displayed it will use as many positions as it needs ignoring the size you entered. The value must be of type string!

RAPTOR Operator Precedence (from highest to lowest)

- unary (most common unary operator is a minus sign (for example -5))
- arithmetic (aka mathematical)
- relational (aka comparison or boolean)
- logical
- assignment

RAPTOR Identifiers An identifier is a user-defined name for something

- Rules for constructing a valid identifier:
  - Note: RAPTOR is not case sensitive
  1. May be composed of upper/lower case letters, digits 0 – 9, and an underscore ”_”
  2. Cannot begin with a digit or contain an embedded spaces
  3. Two naming conventions: camel casing or underscores (_) may be used. Examples: boxes_sold, boxesSold

RAPTOR Data Types

- RAPTOR supports 3 data types:
  - **string** variable will store alphanumeric text or numbers that will not be used in a computation. strings are enclosed in double quotes:
    - Example String values (literals):
      - AB 125", "Cerritos College", "11110 Alondra Blvd."
    - Example string variable names:
      - name, address, description
  - **number** variable needs to store a number. RAPTOR treats all numbers as real numbers
  - **boolean** variable needs to store either True or False

- RAPTOR does not have any keywords for data types. The data type of a variable is determined by it's initialization (type of value [string or number] assigned) Examples: SET totalBoxes TO 0, SET name TO ""

RAPTOR Scope of variables (global/local)

- Default mode in RAPTOR is Novice. In Novice mode, all variables are global variables, regardless of which subchart (flowchart) they were declared in. In Intermediate mode in RAPTOR, local variables may be declared in procedures (in the parameter list and/or in the body of the procedure). Global variables in RAPTOR are available to subcharts but NOT to procedures.

RAPTOR Named Constants

- Create a variable using all caps for the name. Separate words/phrases with underscores
- RAPTOR and the examples in the textbook do not have any way of declaring named constants. You can only express your intent to use a variable as a named constant by using all capital letters in the variable name
Raptor Arrays

- An array is a structure that exists in memory to aggregate data.
- Use an array to store a collection of numbers or strings with a shared name instead of declaring a bunch of variables. For example, use `numbers[3] <- 0` instead of `number1 <- 0, number2 <- 0, number3 <- 0`.
- When you declare an array you must specify the number of elements in the array and an initialization value.
- To declare an array of 10 numbers called `values` you would use `values[10] <- 0`.
- To declare an array of 3 strings called `stooges` you would use `stooges[3] <- ""`.
- Array indexes are in the range of 1 to array size (for examples above 1 to 10 and 1 to 3).
- RAPTOR has a `Length_Of` function which returns the number of elements in an array. `Length_Of(array_name)`
- To access a specific element in an array, use an index inside square brackets `values[1], stooges[2], etc.`
- Array elements can be used anywhere a variable or constant can be used.
- To give names to the stooges you would do something like this:
  o `Length_Of(stooges)` is 3

Logic Planning Tools

- Hierarchy Chart shows high-level processing and the relationship of modules in a program.
- Modules represents a named task to be performed. A Hierarchy chart only requires module names.
  o provide abstraction when designing program logic (allow you to defer details until later)
  o should perform a specific task (be cohesive - "Functional Cohesion")
- Flowchart used to develop program mainline logic and the logic of modules [algorithm]
- Pseudocode used to develop program mainline logic and the logic of modules [algorithm]
- I-P-O Chart used to identify input, outputs, and processing
- Spacing Chart used to show the layout of a report produced by a program PSC, SSC
- Record Layout Form used to show the layout of the data in a data file RLF

Types of Program Errors

- syntax Invalid statement – syntax errors are also referred to as compile-time errors. Program can't be translated until all syntax errors have been corrected
- logic Incorrect output form a program (such as an error in a calculation) detected at run-time
- run-time Division by zero, file unavailable or corrupted, memory protection violation. Program ends abnormally.

Program Structure Pseudocode / Flowchart  RAPTOR / Java

- A program is a collection of one or more modules (these are called methods in Java. They are called built-in procedures and/or functions and user-defined procedures / subcharts in RAPTOR). Every flowchart in RAPTOR uses Start/End in the terminal symbols.
- Older programming languages did not have a module name for the “main program” logic. Examples of these languages include COBOL, Fortran, RPG, BASIC. Examples of languages that must have a named module include: C, C++, C#, Java, and Visual Basic. RAPTOR also has a main flowchart.

Simple Java top-level flowchart

```plaintext
Start
Declar global variables here
main()
End
```

This flowchart demonstrates where Start and End terminal symbols (used in the optional textbook for every program) would conceptually fit into a Java program.

Note: RAPTOR treats all variables as global variables no matter where they are declared (in Novice mode) and the declarations must be inside a flowchart (module). Our convention is to declare global variables in the main module or in an initialization module (houseKeeping(), startUp(), initialize(), etc.).

RAPTOR allows procedures to declare local variables in a parameter list and in the body of the module.

Simple Java top-level pseudocode

```plaintext
Start
  // global variable declarations (these are NOT inside of a method).
  main()  // Java’s method main
End
```
Execution always begins with method main (Java) or flowchart main (RAPTOR). In Java, a program normally ends when method main ends, but may be terminated in ANY method by calling the `System.exit()` method:

```java
System.exit(0);
```

### Java Methods

- Modules are called **methods** in Java. Other languages use words like procedure, function, subroutine, etc.
- Java methods may have parameters and may return a single value using a return statement.

#### Java method headers take the form:

- `<return-type> <method-name> <parameter-list> ( [ type parameter1 ] [ , type parameter2,… ] )`
- `[]` denote optional items. i.e. - No parameters are required for a method declaration.
- If a method returns a value, it can be: (1) assigned to a variable or (2) printed in an output statement.

Example method header:
```java
int sum( int x, int y )
//return type is int, method name is sum, parameter list has two integer parameters
```

- The return data type for a Java method must be one of the following three types:
  - `void` indicates the method does not return a value.
  - `value-type` one of the primitive types: char, boolean, byte, short, int, long, float, double.
  - `reference-type` data type is a class name (like `String`).

- You must specify a return type for every method you create! More about categories of data and data types follows later.
- The body of a method is **always** enclosed in curly braces `{ ... }`

#### Sample Method Headers:

- `double sum(double num1, double num2)`
- `void printHeadings( )`
- `void printInfo( String name, int ID )`

#### File Processing

- Usually uses at least 3 modules in addition to module `main()`.
- Your choice of names may be different than the examples that follow.

1. `houseKeeping( )` //initialization activities
2. `processRecord()` //processing of all the records in the file.
3. `finishUp( )` //final activities

#### Examples of Java File Processing method headers:

- Additional header examples: `void initialize( )` `void startUp( )`
- Additional header example: `void detailLoop( )`
- Additional header example: `terminate( )` `finishUp( )`

### Java method main (file processing application) [used exclusively for Computer Programming Logic course]

```java
public static void main( String args[ ] )
{
    <class-name> obj = new <class-name>(); //Example: ScoutReport obj = new ScoutReport();
    obj.main();
}
```

### Java method main (in a Java file processing application - similar to subchart main in RAPTOR)

```java
void main()
{
    initialize( ); //initialization activities
    while ( inFile.hasNext( ) ) //process every record in the file
        processRecord( );
    terminate( ); //termination activities
}
```
Arithmetic Operators in Java

+ , - , *, /, % (add, subtract, multiply, divide [integer or floating-point], modulus [integer remainder])

/ is used for both floating-point and integer division 23 / 3 is 7 (integer quotient because of integer operands)

% is used to get the integer remainder of a division 23 % 3 is 2 (integer remainder)

Relational Operators in Java

<, <=, >, >=, == (Note: = means assign, == means compare for equality)

Logical Operators in Java (associate operators with operands and perform boolean operations)

Perform logical operations which return a value of true or false

Precedence for binding of operators / operands: (in order from highest to lowest)

• ! Logical NOT unary operator
• && Logical AND binary operator
• || Logical OR binary operator

Note: Once binding has been determined, evaluation of the boolean expression is from left to right

Short-circuit boolean evaluation occurs when condition-1 is TRUE for an OR operation, or condition-1 is FALSE for an AND operation.

Logical expressions should take advantage of this for efficiency. For an OR (||), first ask the question most likely to evaluate to true. For an AND (&&), first ask the question most likely to evaluate to false.

Assignment Operators in Java [The assignment operator in RAPTOR is ←]

• = assignment operator
• +=, -=, *=, /=, %= arithmetic operation and assignment (aka compound assignment)

String comparison methods in Java

For Java Strings, use the equals() method or the compareTo() method for comparisons. Do NOT use the equality operator (==) in Java to compare strings!

Examples if (name.equals("Jack")) or if (name1.compareTo(name2) == 0)

equals method returns a boolean value (true or false). The result of calling the compareTo method is an integer:

• 0 equal comparison
• negative number name1 < name2 : name1 comes before name2 in sequence
• positive number name1 > name2 : name1 comes after name2 in sequence

Identifiers in Java

• Rules for constructing a valid identifier: Note: Java is case sensitive!
  1. May be composed of upper/lower case letters, digits 0 – 9, and special characters "$" and "_" (underscore)
  2. Cannot begin with a digit or contain an embedded space
  3. By convention camel casing is used.

• variables int boxesSold, totalBoxesSold; [variable names should use camel casing]
• named constants final int MAX_SIZE = 100; [all upper case letters by convention]
• class names class ScoutReport [class names should use Title Case]
  method names static void startUp() [class names should use Title Case]

Commenting in Java

//this is a single-line comment Note: may follow statements on the same line
/* this is a multi-line comment */ /* to start <statement(s)> */ to end

Comments may be created for a symbol in RAPTOR by right-clicking the symbol and choosing “comment”

Fundamental Classification of Data Types in Java

• Most newer languages two fundamental categories of data types – reference and value
  o reference types are used to store the address where the actual data can be found
  o value types store the actual data within the variable
Data Types in Java
- Java has two fundamental categories of data types – reference and value
  o reference types are used to store the address of an object
  o value types are used to store one of the 8 primitive data types in Java

- Primitive Data Types (8) in Java (all are value types)
  - Integral negative and positive whole numbers and 0
    - Examples: -5, 1000, 0
  - Floating-point real numbers (numbers with decimal digits)
    - Examples: 12.5, 6.0, -7.222
  - Alphanumeric letters, digits, special characters
    - Examples: "Cerritos", 'c', "12.5%"
  - Logical aka flag or switch or boolean
    - boolean The Java boolean type only accepts the predefined literals true and false

Variables Used to store data while a program is running
- variables must normally be declared before they can be used. They can be initialized in the declaration.
- variables need to be assigned a value before you try to output them or use them in an expression.
- loop control variable j = j + 1
- subscript/index for an array totalSales[year]
- counter count = count + 1;
- accumulator totalSales = totalSales + sales;
- calculated field maturity = principle * pow(1 + intRate, payments);
- flag or switch badRecord = true; //badRecord is a boolean variable
- hold field used when doing control break processing holdEmpNumber = empNumber;
- format string "%-15s %3d %6.2f" //holds the formatting for a printf() method call

Variable Scope in Java (global/local)
The scope of a variable in Java is where it can be accessed in a class
- global Available for use by all methods. Defined OUTSIDE of all methods in Java
- local Available for use only within a method. Defined INSIDE of a method in Java
- global variables are always automatically initialized to the default for the data type
- global variables typically are declared at the beginning of the class before the first method
- global variables (class-level) are automatically initialized to the default for the data type
- local variables (method-level) are NOT automatically initialized.
- You must initialize a local variable or assign some value to it (perhaps by getting input from the user and storing it in the variable) BEFORE you try to use the variable in any other type of statement. If you forget to do this, you will get a compile time error. Java will complain that "the variable might not have been initialized".

Literals in Java All literals have a data type based on the characters used in the literal
- integer type int in Java 5 -3 0
- floating-point type double in Java 12.5 -3.89, 123.4567
- character type char in Java 'A' 'b' "" "$" - single quotes
- string type String in Java "Hello World", "A" "123" - double quotes
- boolean true / false (lowercase, no double quotes)

Named Constants in Java
- use the keyword final in Java in a variable declaration to create a named constant
- convention is to use all uppercase letters with words separated by underscore (_) characters
- named constants must be initialized (usually in the declaration) before they can be used
- final int MAX_STUDENTS = 32;  final double PI = 3.1415;
Control Structures are implemented as statements in Java

**sequence**  process of executing statements one after another until the program ends.

Note: A **sequence structure** performs a **single action**. Distinguish between a sequence structure and sequence.

**Examples of sequence structures in Java**

- input operation  
  ```java
  boxesSold = input.nextInt();
  ```
- output operation  
  ```java
  System.out.println("Total Sales: "+ totalSales);
  ```
- declaration of variables  
  ```java
  int x, y, z;  String scoutName;
  ```
- performing a calculation  
  ```java
  totalSales = boxesSold * 5.00;
  ```
- incrementing a counter  
  ```java
  scoutCounter = scoutCounter + 1;
  ```
- accumulating into an accumulator  
  ```java
  grandTotalSales += totalSales;
  ```
- calling a module  
  ```java
  finishUp();
  ```

**selection**  performing a test to determine which of possibly 2 paths of execution should be followed. Order affects efficiency!

**Examples of selection structures in Java:**

- Remember, to perform more than one statement, create a compound statement (block) --->  
  ```java
  {  }
  ```
- if-then:  
  ```java
  if ( condition )
  
  statement;
  ```
  ```java
  if ( boxesSold != -1 )
  
  totalBoxesSold = totalBoxesSold + boxesSold;
  ```
- if-then-else:  
  ```java
  if ( condition )
  
  statement_1;
  ```
  ```java
  if ( x > 0 )
  
  System.out.println("X is a positive number");
  ```
  ```java
  else
  
  else
  
  statement_2;
  ```
  ```java
  System.out.println("X is 0 or a negative number");
  ```
- A special version of a selection structure where branching occurs on the ELSE branches based on the value of an integral expression (usually a variable of type int or char) is called a **CASE structure**.

**looping**  a group of statements are executed repeatedly while a condition is true or until a condition becomes true. Also known as repetition and iteration.

**Types of loop structures / Statements in Java/RAPTOR**

- **pre-test**  the test occurs before the body of the loop.
  - **while** / **for in Java**
- **post-test**  the test occurs after the body of the loop has been executed
  - **do..while in Java**
- **mid-test**  the test can occur anywhere in the loop (not covered in our textbook)
  - **Many languages do NOT support this directly. It is possible to draw a flowchart with a mid-test loop in RAPTOR.**
- **definite**  [AKA counter-controlled]  
  - Uses a loop control variable (lcv) and initialize, test, and update expressions  
  - Loop control variable is incremented or decremented [update]  
  - Loop control variable is tested against a maximum or minimum value [test]
- **indefinite**  [AKA sentinel-controlled]  
  - A **sentinel value** is a special value used to signal end of input. Uses the test expression to compare against a sentinel value and updates a variable.

**Loop Expressions**  the following are used with a definite loop. Indefinite loops only require a test expression

- **initialize**  declare and/or initialize the loop control variable
- **test**  perform the test expression to see if the body of the loop should be entered (repeated)
• update  increment/decrement the loop control variable using a step value or alter the variable being compared to the sentinel value in an indefinite loop (usually by doing an input operation).

Examples of repetition statements (loops) in Java

• while  //pre-test loop
  
  initialize-expression  int count = 10;
  
  while ( test-expression )
  
  {  statement;
  
      update-expression;  count = count – 1;
  
  }

• do..while  //post-test loop
  
  initialize-expression  int count = 10;
  do
  
  {  statement;
  
      update-expression;  count = count – 1;
  
  } while ( test-expression );

• for  //pre-test loop
  
  for (initialize-expression(s) ; test-expression ; update-expression(s) )
  
  {  <loop body>}  [ {   } are optional if the body of the loop is a single statement… ]

  for ( int j = 1; j <= 10; j++)
  System.out.println( j );

Arrays in Java

• An array is a structure that exists in memory to aggregate data
• An array type (int [], double [], etc.) is a reference type (like String). Arrays in Java are objects.
• Java supports both single and multiple dimension arrays

How to define arrays:

1. create an array reference variable:  int scores[];
2. create an array object and assign the reference to the reference variable:
   
   scores = new int[100];  //here 100 is called the array size declarator
   
   Note: Steps 1 and 2 can be done in a single statement:  int scores[] = new int[ 100 ];
3. Alternatively, you can create an array object using an initializer list:
   
   int scores[] = { 5, 5, 4, 5, 3, 5, 4, 4, 5, 3, 5, 3, 4, 5 };

Note: You do not specify the size of the array or use the new operator when you are using an initializer list.

Keep in mind that arrays are indexed beginning at 0. For a 5 element array, the valid subscripts (indexes) would be 0, 1, 2, 3, 4. In general valid subscripts are: 0 to arraySize - 1.

Sometimes it is more convenient (or more natural) to use subscripts for an array that begin with 1 instead of 0. This is easily accomplished by declaring the array size to be 1 larger than the number of data values you want to store.

For example, to keep track of the total sales for 5 years, we could use an array of size 6.

double totalSales[] = new double[ 6 ];  Valid valid subscripts for this array would be 0 – 5. The total number of elements in the array would be 6. You would not use the first element in the array to store anything. Instead, store data in elements 1, 2, 3, 4, and 5. If we processed the data in a for loop, the header would look like this:

for ( int index = 1; index < totalSales.length; index++ ) { totalSales[ index ] ... }

If you are using an initializer list to create the array you can provide a "bogus" initializer for the first element. For example:

String WeekDay[] = { "", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday" };
weekDay[0] is an empty string. weekDay[1] is "Monday", weekDay[5] is "Friday".

If the array is constructed using numbers, use a value of 0 for the first element.
For example:  int score[ ] = { 0, 100, 95, 82, 73, 64, 86, … };

- Types of arrays
  - "pre" run-time data is read from an external file and loaded into an array
  - compile-time data for the array is in the program
  - runtime values are assigned dynamically as the program runs

- Uses of arrays
  - when applicable, you may replace nested if or case statements with the use of an array
  - use as a collection of counters or accumulators
  - use as a collection of data items for searching/sorting operations

Examples: an array of: customer numbers, book titles, scores on an exam, etc.
- use parallel arrays when multiple pieces of information need to be synchronized.

- Loading of arrays
  - use a loop to load the array with data from a file ("pre" run-time array)
  - create an array using an initializer list

- Searching of arrays
  - use a search algorithm to determine if a value is stored in an array
    - linear search
      - array does not need to be sorted
    - binary search
      - array must be sorted in ascending or descending sequence

- Sequencing of data in arrays
  - unsorted
  - sorted in ascending or descending order

- Accessing an array element
  - scores[ 1 ], scores[ item ], scores[ item + 5 ]
  - any valid integer expression

- Valid subscripts are in the range 0 to array size – 1 (array.length – 1).
- the size of the array can be determined by used the length field for the array object.

For example if you have an array called scores, the expression scores.length will give you the size of the array. This can be used in a test expression for a loop as the upper limit.

For example: for ( int j = 0; j < scores.length; j++ ) //body of for loop...

- Valid subscripts are in the range 0 to array size – 1 (array.length – 1)

- Using array elements in calculations
  - An array element of an appropriate type can be used in any expression where a variable of that type could be used.

- Using loops with arrays
  - use loop control variable as a subscript

  int scores[ ] = { 5, 5, 3, 4, 4, 5, 5, 2, 5, 4, 4, 3, 5, 2, 4, 4, 5, 5, 5, 5 };
int sumOfScores = 0;  //initialize accumulator

for ( int j=0; j < scores.length; j++ )
    sumOfScores += scores[j]; //accumulate scores

---

**Java Import Statements**

An import statement makes a single class or all classes in a package available for use in a Java program. An import statement is required when the class to be used is not in the same package as the class that needs to use it. Following are examples of import statements:

- import javax.swing.JOptionPane;  // necessary to use input/output dialog boxes
- import java.util.Scanner;  // necessary to use the Scanner class for input from console or from a file
- import java.io.*;  // required when any file input/output is done

**Java throws Clause**

A throws clause is usually required when you are doing input or output using files:
- typically you need at a minimum throws FileNotFoundException as part of a method header
- you may also use (or may have to use instead…) “throws IOException”
- the compiler can tell you which methods must have these clauses and what the exception clause is

**Input/Output Objects in Java**

- Using the **Scanner** class to do input:
  - from the keyboard: Scanner input = new Scanner ( System.in );
  - from a file: Scanner inFile = new Scanner( new File( “input_file.txt” ) );
- To display output to the screen use the **System.out** object
- Using the **PrintWriter** class to do output to a file:
  - PrintWriter outFile = new PrintWriter ( “output_file.txt” );
- first declare the objects as global variables:
  - Scanner inFile;
  - PrintWriter outFile;
- then create the objects and assign them in your initialization module [ houseKeeping( ) for example ]
  - inFile = new Scanner( new File (“input_file.txt” ) );
  - outFile = new PrintWriter( “output_file.txt” );

- **JOptionPane Dialog Boxes**
  - Input  display a prompt / accept input / store input string in a String variable
    - String numberString = JOptionPane.showInputDialog(“Please enter a number”);
    - if you use an input dialog box to get numeric information, you must remember to accept the input into a String variable and then use a Java wrapper class conversion method: Double.parseDouble(), Integer.parseInt(), etc. to convert the string and store it in a numeric variable.
  - Output  display a message (a String) in a dialog box in the center of the screen
    - JOptionPane.showMessageDialog( null, “square of the number is “ + numberSquared );
- Remember to close input and output files!  Input.close( );   output.close( );

---

**Input in Java using Scanner class methods**

The Scanner class is used to create an input object that can be used to get input from the user interactively (console input), input from a file, even input from a String. Once the input object is created, the following methods can be used to get the input or test to see if input is available.

- next( )  //get a String delimited by whitespace
- nextLine( )  //get rest of line up to newline character as a String
- nextInt( )  //get an integer
- nextFloat( )  //get a float number
• `nextDouble( )`  //get a double number

• `hasNext( )`  //returns true if some type of input exists, false otherwise
  // The `hasNext()` method is often used to determine if you are at the end of input in a loop

• `hasNextInt( )`  //is there an integer?
• `hasNextDouble( )`  //is there a double?
• `hasNext( )`  //is there anything?

---

**Output in Java using `print()`, `printf()`, and `println()` methods**

These methods are used for the `System.out` object or for a `PrintWriter` object (mapped to a file).

Methods:

• `print( )`  //print a line (does not include a newline character)
• `printf( )`  //print a formatted string
• `println( )`  //print a line (includes a newline character)
Java printf() Method Quick Reference

System.out.printf("format-string", arg1, arg2, ...); //format string and optional argument list

Format String:
Composed of literals and format specifiers. Arguments are required only if there are format specifiers in the format string. Format specifiers include: flags, width, precision, and conversion characters in the following sequence:

% [flags] [width] [.precision] conversion-character  ( square brackets denote optional parameters )

Flags:
- : left-justify (default is to right-justify)
+ : explicitly output a plus (+) or minus (-) sign for a numerical value
0 : forces numerical values to be zero-padded (default is blank padding)
, : comma grouping separator (for numbers > 1000)
  : space will display a minus sign if the number is negative or a space if it is positive

Width:
Specifies the field width for outputting the argument and represents the minimum number of characters to be written to the output. Include space for expected commas and a decimal point in the determination of the width for numerical values. Note: the width is ignored if it is too small for the value being formatted

Precision:
Used to restrict the output depending on the conversion. It specifies the number of digits of precision when outputting floating-point values or the length of a substring to extract from a String. Numbers are rounded to the specified precision.

Conversion-Characters:
c : character (Capital C will uppercase the letter)
d : decimal integer
f : floating-point number (real number)
s : String (Capital S will uppercase all the letters in the string)

h : hashcode (A hashcode is like an address. This is useful for printing a reference)
n : newline (%n is a platform-independent way of generating a newline character in output)

Examples:
System.out.printf("Total is: $%,.2f\n", dblTotal);
System.out.printf("Total: %10.2f: ", dblTotal);
System.out.printf("%4d", intValue);
System.out.printf("%-20.10s\n", stringVal);
String s = "Hello World";
System.out.printf("The String object %s is at hash code %h\n", myString, myString);

String class format() method:
You can build a formatted String and assign it to a variable using the static format method in the String class. The use of a format string and argument list is identical to its use in the printf method. The format method returns a reference to A String. Example:

String grandTotal = String.format("Grand Total: %,10.2f: ", dblTotal);
Testing for End of File (using an input file in Java)

- use the Scanner class method `hasNext()` with your input file object (inFile for example) as the condition that controls the loop. This is the way you do the EOF test in Java.

```java
while ( inFile.hasNext( ) ) //read and process a record
    processRecord( );
```

When using the Scanner class for input from a file, generally you do not need to do a priming read. This is a departure from the way this is explained in the book! No priming read is necessary for Java to determine if there is something in the file that can be read.

However ..., you should use a priming read when writing a control break program! See below.

In this situation, you would use a boolean variable that would be explicitly set every time a read operation is performed to determine when the end of the file has been reached.

Control Break Programs

Control break programs permit creating reports with summary lines and/or detail and summary lines. A file processed by a control break program must be sorted on all fields which will be used as control fields.

- Control break reports detail and summary or summary only (detail lines are optional)
- Control break levels single, double, triple, etc
- Control break fields file must be sorted on these fields (can be one or many)

Initialization activities
- Initial priming read of the first record in the data file
- Assign values to control break hold fields (reason priming read must be done… )

Record processing loop
- In a control break program, you should do a priming read in your initialization module and read the rest of the records in the module which is being called repeatedly (I will call this module `processRecord` for simplicity) to process the records.

The last statement in your `processRecord` module should be to read the next record from the file.

- When writing a control break program in Java use an end-of-file boolean variable as the condition for your main loop. Example: `while ( eof == false )` or alternately `while ( ! eof )`

Use a method to read a record from the data file. The method that reads the record should check to see if data is available and read it. If it is not available, change the value of the end-of-file boolean variable.

```java
Example boolean variables:
boolean eof = false;        boolean more = true;

Example loops in method main:
while ( eof == false )       while ( more )
    processRecord();        processRecord();
```

Example readRecord method:

```java
static void readRecord( ) throws FileNotFoundException
{
    if ( inFile.hasNext( ) )
    {
        < statement(s) to read fields in a record >
    }
    else
    {
        //use one of the following statements
        eof = true;  //using eof boolean variable
        more = false; //using more boolean variable
    }
}
```

You can use a boolean variable named `eof` or `more` as a flag to do end-of-file checking for a control break program. The name you choose determines how you implement the logic.

- initialize `eof` to false and later set it to `true` OR
- initialize `more` to true and later set it to `false`

Both approaches implement the same logic! – detecting an end of file condition and setting a boolean variable to reflect the change.

`while(more)` is equivalent to `while( more == true )`
RAPTOR readRecord subchart template

```
readRecord
  if  End_OF_Input
    eof <-- true   [ or  more <-- false ]
  else
    < statement(s) to read fields from the input file and store them in variables >
  end if
end
```

- **Control break processing**
  - check highest level to lowest level in order (major to minor)
  - if a break is detected: process from **lowest** level up to level at which the break occurred
  - you must force a final control break at end of the file (after the last record has been read)

- **Control break activities**
  - Print stuff:
    - print headers/footers
    - print separator line
    - print summary line (remember to print hold fields, not the actual field) and optionally detail lines
  - accumulate into higher-level accumulators (**roll up the totals**)
  - increment counter(s)
  - zero out accumulators
  - update hold fields

- **Location of method calls for a control break method**
  - **Multiple calls from one level**
    Example (double-level control break on state / city when state changes (or end of file)):
    ```
    process lowest level first up to level at which break occurred
    processCityBreak();
    processStateBreak();
    ```
  - **embedded calls in higher level modules to lower level modules**
    Example (double-level control break on state / city when state changes (or end of file)):
    ```
    processStateBreak() method calls processCityBreak() first (to process the city control break)
    processStateBreak(); // first statement call the processCityBreak() method
    ```

- **Termination activities**
  - call the control break method one last time (must force last control break)
  - do any final calculations necessary
  - print final summary lines (grand totals, averages, counts, etc.)

- **Checking for correctness of a control break program**
  - Be sure and carefully check the first and last detail/summary lines in a control break report
  - When you print a detail/summary line you usually should be printing the value of a hold field if it exists
  - You should have a test data file and know what the program should produce to verify output
  - If producing both detail and summary lines, it is helpful to indent the summary lines for clarity
Sorting a data file in Windows using the command line utility `sort`

Windows has a utility for sorting a data file. If you are sorting on the first field in the file, it is very easy to use. Data files must be sorted for a control break program so if you have an unsorted file, this makes sorting it very easy.

The format of the command is: `sort datafile-name > sorted-datafile-name`

- `datafile-name` represents the file to be sorted.
- `sorted-datafile-name` represents the sorted data file to be created.

The `>` sign after the `datafile-name` is a redirection operator. In this case it takes the output of the sort operation and writes it to the file `sorted-datafile-name`.

Example of Use:

C:\Users\jwilson\Desktop> `type Troop_10_data.txt`
3  7
1  5
4  5
4  9
1  7
2  12
3  11
2  9
1  8

C:\Users\jwilson\Desktop>`sort Troop_10_data.txt > Sorted_Troop_10_data.txt`

C:\Users\jwilson\Desktop>`type Sorted_Troop_10_data.txt`
1  5
1  7
1  8
2  12
2  9
3  11
3  7
4  5
4  9