METRO STATE UNIVERSITY

What is a Data Structure? & Why Do we care?

Complete: 9/5

METRO STATE UNIVERSITY

What is a Data Structure?

What is Data Structure?

METRO STATE UNIVERSITY

da·ta

noun

facts and statistics collected together for reference or analysis.
"there is very little data available"

Similar: facts, figures, statistics, details, particulars, specifics, features, information, evidence, intelligence, material, background, input, proof, fuel, ammunition, statement, report, return, dossier, file, documentation, archive(s), info, dope, lowdown, deets, gen

- the quantities, characters, or symbols on which operations are performed by a computer, being stored and transmitted in the form of electrical signals and recorded on magnetic, optical, or mechanical recording media.

struc·ture

noun

the arrangement of and relations between the parts or elements of something complex.
"flint is extremely hard, like diamond, which has a similar structure"

Similar: construction, form, formation, shape, composition, fabric, anatomy, makeup, constitution, organization, system, arrangement, layout, design, frame, framework, configuration, conformation, pattern, plan, mold, setup

- A ***data structure*** is a ***collection of data***, ***organized*** so that ***items can be stored and retrieved by some fixed techniques***

data structure — internal to the computer, not a database

What is Data Structure?

METRO STATE UNIVERSITY

da·ta

noun

facts and statistics collected together for reference or analysis.
"there is very little data available"

Similar: facts, figures, statistics, details, particulars, specifics, features, information, evidence, intelligence, material, background, input, proof, fuel, ammunition, statement, report, return, dossier, file, documentation, archive(s), info, dope, lowdown, deets, gen

- the quantities, characters, or symbols on which operations are performed by a computer, being stored and transmitted in the form of electrical signals and recorded on magnetic, optical, or mechanical recording media.

struc·ture

noun

the arrangement of and relations between the parts or elements of something complex.
"flint is extremely hard, like diamond, which has a similar structure"

Similar: construction, form, formation, shape, composition, fabric, anatomy, makeup, constitution, organization, system, arrangement, layout, design, frame, framework, configuration, conformation, pattern, plan, mold, setup

- A ***data structure*** is a ***collection of data***, ***organized*** so that ***items can be stored and retrieved by some fixed techniques***

- Examples: a line of customers, a stack of books, a to-do list, a dictionary

What is Data Structure?

METRO STATE UNIVERSITY

da·ta

noun

facts and statistics collected together for reference or analysis.
"there is very little data available"

Similar: facts, figures, statistics, details, particulars, specifics, features, information, evidence, intelligence, material, background, input, proof, fuel, ammunition, statement, report, return, dossier, file, documentation, archive(s), info, dope, lowdown, deets, gen

- the quantities, characters, or symbols on which operations are performed by a computer, being stored and transmitted in the form of electrical signals and recorded on magnetic, optical, or mechanical recording media.

struc·ture

noun

the arrangement of and relations between the parts or elements of something complex.
"flint is extremely hard, like diamond, which has a similar structure"

Similar: construction, form, formation, shape, composition, fabric, anatomy, makeup, constitution, organization, system, arrangement, layout, design, frame, framework, configuration, conformation, pattern, plan, mold, setup

- A ***data structure*** is a ***collection of data***, ***organized*** so that ***items can be stored and retrieved by some fixed techniques***
- Examples: a line of customers, a stack of books, a to-do list, a dictionary
- Example of data structures inside a computer: ***array***, ***stack***, ***queue***, ***tree***, ***dictionary***, ***graph***, ***lists***, etc....

Data Structure – Why do we care?

- ***Data structures*** are often the ***main building blocks of programs***.
  - Used regularly when implementing ***algorithms***.
  - ***Algorithm*** = procedure or sequence of steps for solving a problem

- How is the data ***organized*** inside the memory?
  - Example: Array vs Linked List
  - Contiguous vs non-contiguous memory allocation

- What are the ***implications***?
  - Random access of data
  - Data Insertion, Removal, Searching, Sorting

- How ***efficient*** is an ***algorithm*** when using a particular data structure
  - Space and Time Complexity Analysis

Data Structure – Why do we care?

- ***Data structures*** are often the ***main building blocks of programs***.
  - Used regularly when implementing ***algorithms***.
  - ***Algorithm*** = procedure or sequence of steps for solving a problem

- How is the data ***organized*** inside the memory?
  - Example: Array vs Linked List
  - Contiguous vs non-contiguous memory allocation

- What are the ***implications***?
  - Random access of data
  - Data Insertion, Removal, Searching, Sorting

- How ***efficient*** is an ***algorithm*** when using a particular data structure
  - Space and Time Complexity Analysis

Basic Operations on Data Structures

- There is a set of basic operations that need to be defined on each data structure:
  - ***create:*** initially creating an empty data structure.
  - ***insert:*** inserting a new item in the data structure without affecting the data structure properties.
    - For some data structures, you can insert only at a specific position in the structure:
      - In a stack, you can insert only at the top
      - In a queue, you can inset only at then end.
  - ***delete:*** deleting an item from the data structure without affecting the data structure properties.
    - For some data structures, you can delete only a specific item:
      - For stack, you delete only the last inserted item
      - For a queue, you can delete only the item that inserted first.
  - ***search:*** searching for an item with a specific value in order to retrieve more details about this item.
  - ***isEmpty:*** returns a Boolean answer in order to say whether there are any data items in the data structure.
  - ***size:*** returns how many data items are currently there in the data structure

Why So Many Data Structures?

- Applications differ in their requirements.

- An ideal data structure for a given application is the data structure that efficiently support the most frequent operation for that application.

- Factors to be considered when choosing a data structure:
  - one operation’s performance vs. another’s
  - time vs. space
  - generality vs. simplicity

Data Structures Tradeoffs

- The study of data structures is the study of tradeoffs

- Data structures differ in:
  - **The way the data is organized:**
    - Linear vs. non-linear
    - Arrays vs. linked lists
  - **Operations supported:**
    - An array is a random-access data structures
    - Stack supports Last-in-First-out (LIFO) operations
    - Queue supports First-in-First-out (FIFO) operations
  - **Efficiency of the various operations:**
    - Arrays are characterized by fast insert() and slow search()
    - Trees are characterized by slow insert() and fast search()

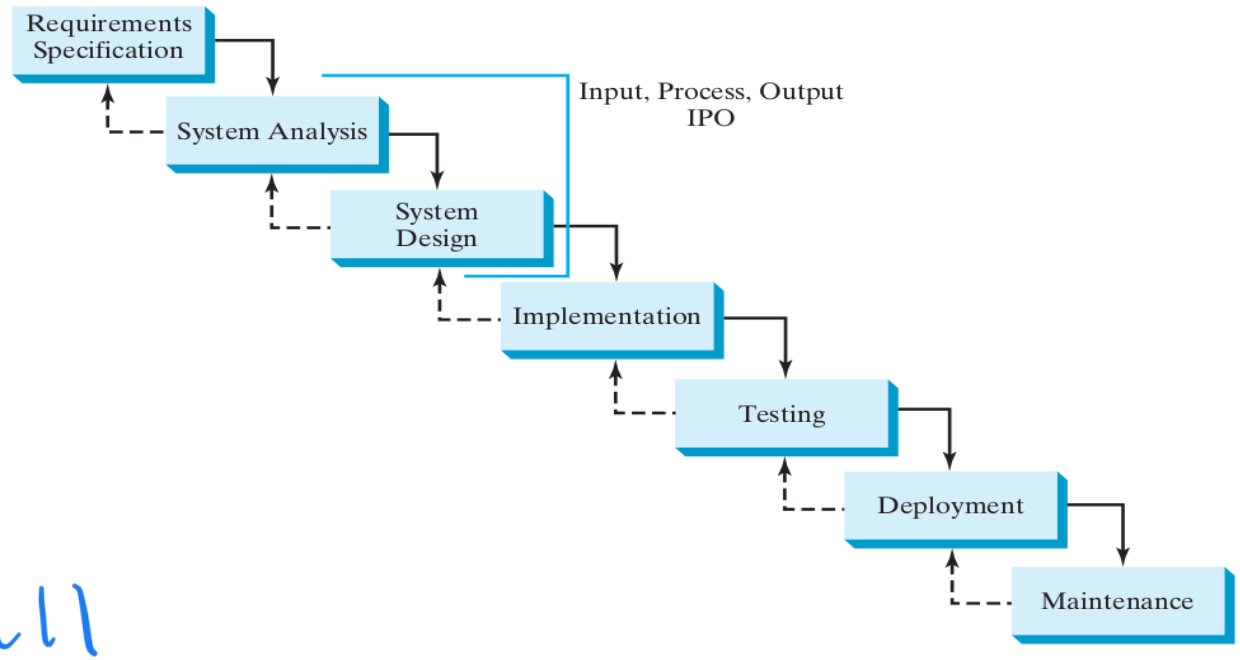
METRO STATE UNIVERSITY

Phases of Software Development

Warning: Phases of Software Development

- Some textbooks say:
  - Specification
  - Design
  - Implementation
  - Analysis
  - Testing
  - Maintenance
  - Obsolescence

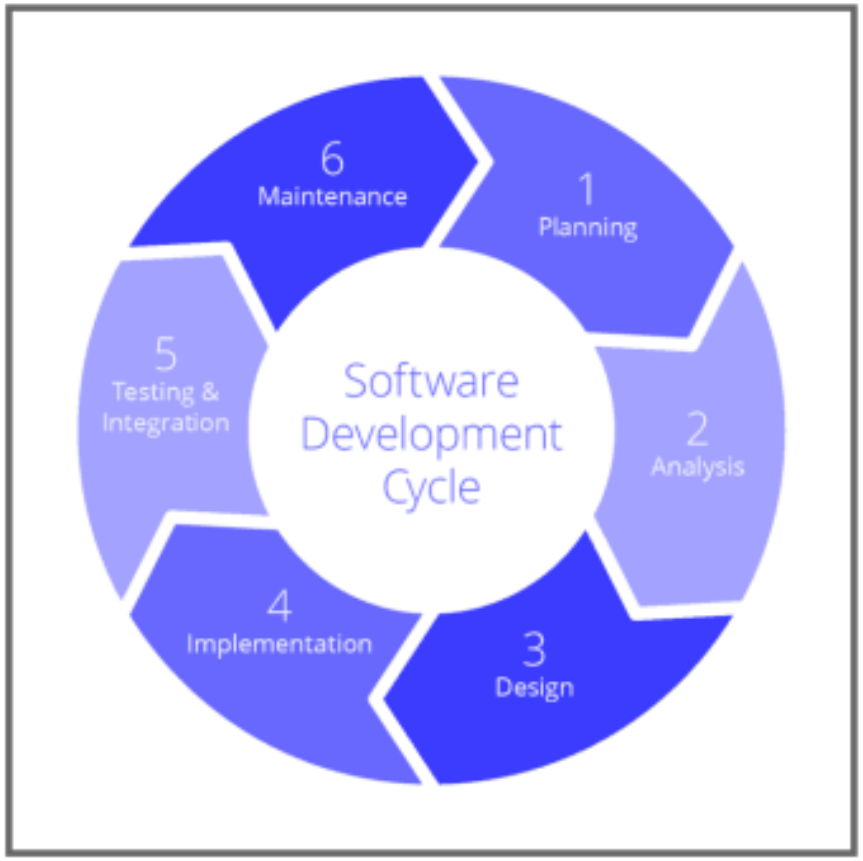
OR -->



Waterfall

Warning: Phases of Software Development

- This is called "***waterfall***" development.
- Not often used because specs change and evolve too often and too fast
- Led to "***agile***" development ---> practices



A typical SDLC representation

Some Good Points on Software Development

- One should ***understand*** a problem ***before*** attempting to ***design*** a solution
- Divide and Conquer:
  - It helps to ***break down problems*** into ***manageable subtasks***
    - For ***complex problems***, each subtask could be handled by ***one individual*** or one ***team***
- Aim to ***reuse*** existing solutions
  - Avoid rewriting code when possible

Some Good Points on Software Development

- One should ***understand*** a problem ***before*** attempting to ***design*** a solution
- Divide and Conquer:
  - It helps to ***break down problems*** into ***manageable subtasks***
    - For ***complex problems***, each subtask could be handled by ***one individual*** or one ***team***
- Aim to ***reuse*** existing solutions
  - Avoid rewriting code when possible
- Testing and Debugging tips
  - Test ***boundary values*** of inputs
  - Fully exercise code – testing ***every*** line of the code - (ex: all options of if/else statements)
  - ***Regression*** Testing – Ensuring that one modification does not break something else in the code

Some Good Points on Software Development

- One should ***understand*** a problem ***before*** attempting to ***design*** a solution
- Divide and Conquer:
  - It helps to ***break down problems*** into ***manageable subtasks***
    - For ***complex problems***, each subtask could be handled by ***one individual*** or one ***team***
- Aim to ***reuse*** existing solutions
  - Avoid rewriting code when possible

What are the implications for having multiple teams / individuals working on the same project?

- Testing and Debugging tips
  - Test ***boundary values*** of inputs
  - Fully exercise code – testing ***every*** line of the code - (ex: all options of if/else statements)
  - ***Regression*** Testing – Ensuring that one modification does not break something else in the code

Some Good Points on Software Development

- One should ***understand*** a problem ***before*** attempting to ***design*** a solution
- Divide and Conquer:
  - It helps to ***break down problems*** into ***manageable subtasks***
    - For ***complex problems***, each subtask could be handled by ***one individual*** or one ***team***
- Aim to ***reuse*** existing solutions
  - Avoid rewriting code when possible

Communication, Data sharing, etc...

- Testing and Debugging tips
  - Test ***boundary values*** of inputs
  - Fully exercise code – testing ***every*** line of the code - (ex: all options of if/else statements)
  - ***Regression*** Testing – Ensuring that one modification does not break something else in the code

METRO STATE UNIVERSITY

Brief Overview & Components of Java Programming

Flow of Execution of a Java Program

- ***Every*** Java program starts execution with ***main()*** method

Flow of Execution of a Java Program

- ***Every*** Java program starts execution with ***main()*** method
- Statements in the program are executed ***sequentially***

Flow of Execution of a Java Program

- ***Every*** Java program starts execution with ***main()*** method
- Statements in the program are executed *sequentially*
- When a ***method*** is called:
  - ***control is transferred*** to the first statement in the body of the ***called method***
  - After the last statement of the called method is executed, ***control is passed back*** to the point ***immediately following*** the method call

```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  System.out.println(
    "The maximum between " + i +
    " and " + j + " is " + k);
}
```

```
public static int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

What do we use in a Java program?

- ***Data types*** - example: integer, double, boolean, character
  - How do we make use of data types?
    - Define a ***variable*** using an ***identifier***
    - Assign ***permitted value*** or apply ***permitted operations*** to the variables

- ***Expressions***
- ***Statements***: Write expressions that define computations to be performed
- ***Control flow execution***: Conditionals, Loops, Function Calls, Returns
- ***Methods***

What do we use in a Java program?

- ***Data types*** - example: integer, double, boolean, character
  - How do we make use of data types?
    - Define a ***variable*** using an ***identifier***
    - Assign ***permitted value*** or apply ***permitted operations*** to the variables

- ***Expressions***
- ***Statements***: Write expressions that define computations to be performed
- ***Control flow execution***: Conditionals, Loops, Function Calls, Returns
- ***Methods***

Give examples of data types and permissible operations on each type

Boolean Expressions

Boolean Expressions

- ***Boolean Expressions*** = conditional statements that evaluates to ***true*** or ***false***.

- Use ***relational*** and ***logical operators***
  - Relational operators to express relation between entities ➔

Relational Operators	
Operator	**Meaning**
A == B	is A equal to B ?
A < B	is A less than B ?
A <= B	is A less than or equal to B ?
A > B	is A Greater than B ?
A >= B	is A Greater than or equal to B ?
A != B	is A not equal to B ?

  - Logical operators - used to form ***compound expressions***
    - Logical ***AND*** (***&&***): expression evaluates to ***true*** only if ***all*** its components evaluate to ***true independently***
    - Logical ***OR*** (***||***): expression evaluates to ***true*** if ***any*** of its components evaluate to ***true***
    - Logical ***NOT*** (***!***) returns the ***negation*** of the expression it is applied to.

Example on using AND (&&)

- Assume a car rental agency wants a program to determine who can rent a car. The rules are:
  - A renter must be 21 years old or older.
  - A renter must have a credit card with $10,000 or more of credit.

Example on using AND (&&)

- Assume a car rental agency wants a program to determine who can rent a car. The rules are:
  - A renter must be 21 years old or older.
  - A renter must have a credit card with $10,000 or more of credit.

```
if ((age >= 21) && (credit >= 10000)){
   System.out.println("You can rent! ");
}else{
   System.out.println("You cannot rent! ");
}
```

Could a 30 year old with $10000 credit rent a car?

Example on using OR (||)

- You would like to buy a $25,000 sports car. To pay for the car you need either enough cash or enough credit.

Example on using OR (||)

- You would like to buy a $25,000 sports car. To pay for the car you need either enough cash or enough credit.

```
if ((cash >= 25000) || (credit >= 25000)){
   System.out.println("Enough to buy the car! ");
}else{
   System.out.println("Sorry! You cannot buy the car");
}
```

Example on using Not (!)

- You are shopping for new shoes. You are only interested in shoes that cost less than $50. Here is how you program your decision.

Example on using Not (!)

- You are shopping for new shoes. You are only interested in shoes that cost less than $50. Here is how you program your decision.

```
if ( !(cost < 50) ){
        System.out.println("Reject these shoes");
}else{
        System.out.println("Acceptable shoes");
}
```

Example on using Not (!)

- You are shopping for new shoes. You are only interested in shoes that cost less than $50. Here is how you program your decision.

```
if ( !(cost < 50) ){
        System.out.println("Reject these shoes");
}else{
        System.out.println("Acceptable shoes");
}
```

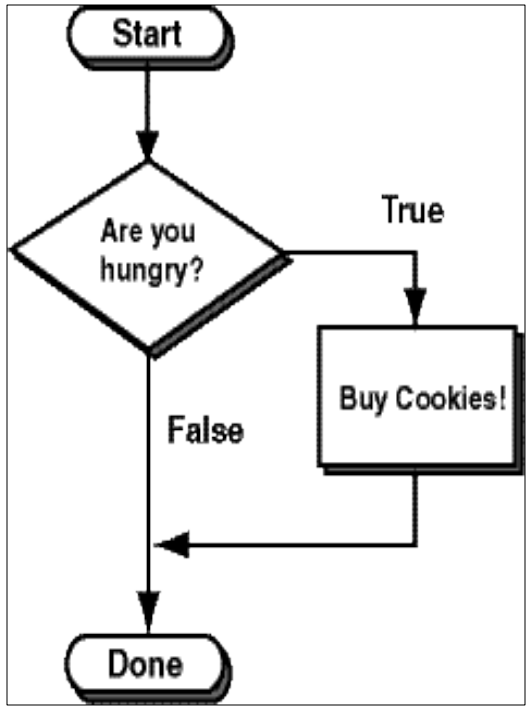
Note: The following code is not correct because ! is only applies to `cost`

```
if ( !cost < 50 ) {
        System.out.println("Reject these shoes");
}else{
        System.out.println("Acceptable shoes");
}
```

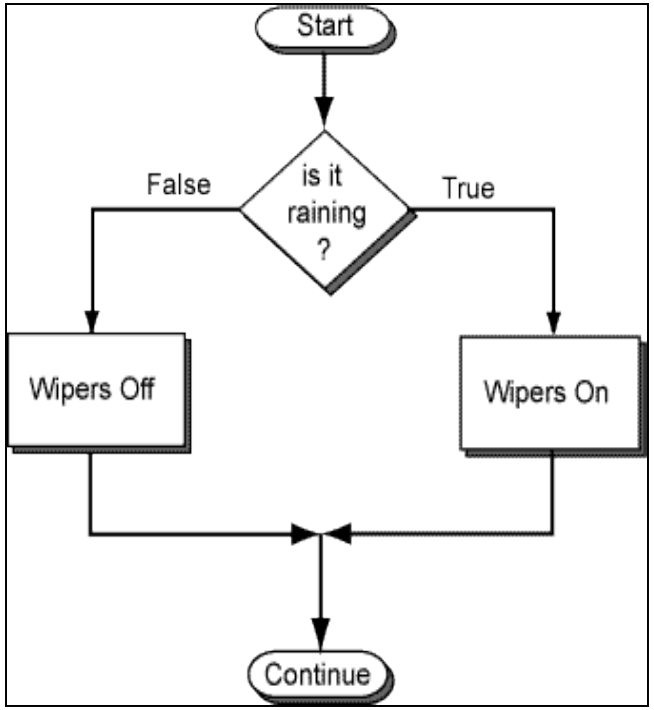
If

Types of decisions

Single-branch



Two-way



`if` statement Syntax

Syntax

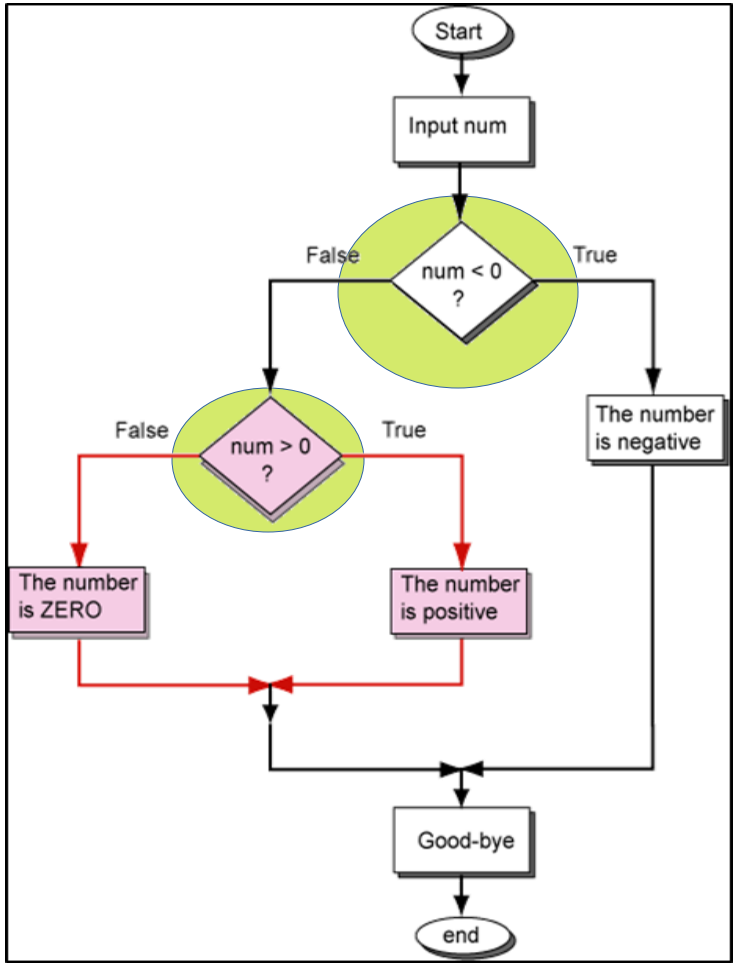
```
if (condition){
    statement block 1;
}
else{
    statement block 2;
}
```

Example

```
if (age < 13){
    System.out.println("Child");
}

else{
    System.out.println(" Not a child");
}
```

Multiple Branch Decisions



Multi-branch `if` statement

```
if (condition-1){
  statement block-1
}else if (condition-2) {
  statement block-2
}else if (condition-3) {
  statement block-3
}else if (condition-4) {
  statement block-4
}else {
  statement block-5
}
```

- The conditions are checked in order
- Only one statement will be executed which corresponds to the to the first condition that is evaluated to true
- **Order matters!**

Note

```
if i > 0 {
  System.out.println("i is positive");
}
```

(a) Wrong

```
if (i > 0) {
  System.out.println("i is positive");
}
```

(b) Correct

```
if (i > 0) {
  System.out.println("i is positive");
}
```

(a)

Equivalent

```
if (i > 0)
  System.out.println("i is positive");
```

(b)

CAUTION

A `return` statement is required for a value-returning method. The method shown below in (a) is logically correct, but it has a compilation error because the Java compiler thinks it possible that this method does not return any value.

```
public static int sign(int n) {
  if (n > 0)
    return 1;
  else if (n == 0)
    return 0;
  else if (n < 0)
    return -1;
}
```

(a)

Should be

```
public static int sign(int n) {
  if (n > 0)
    return 1;
  else if (n == 0)
    return 0;
  else
    return -1;
}
```

(b)

To fix this problem, delete *if (n < 0)* in (a), so that the compiler will see a return statement to be reached regardless of how the `if` statement is evaluated.

Switch

.Alternative to if/else structure if all conditions are using a comparison for one variable.

Revisit if/else if

Write a program that reads a one-digit number from the user then prints on the screen the digit in letters.

```
  int digit;
  System.out.println("Enter a single digit number:");
  digit = input.nextInt();


  if (digit == 0){
        System.out.println(" ZERO ");
}else if (digit==1){
        System.out.println(" ONE ");
}else if (digit==2){
        System.out.println(" TWO ");
        …and so on….
}else{
        System.out.println(" Not a single-digit number");
    }
```

`switch` Statement Rules

The switch-expression must yield a value of char, byte, short, int, or String type and must always be enclosed in parentheses.

The value1, ..., and valueN must have the same data type as the value of the switch-expression and must be a constant variable or a literal value.

The resulting statements in the case statement are executed when the value in the case statement matches the value of the switch-expression.

```
switch (switch-expression) {
   case value1:
        statement(s)1;
         break;
   case value2:
        statement(s)2;
         break;
   …
   case valueN:
        statement(s)N;
         break;
   default:
        statement(s)-for default;
}
```

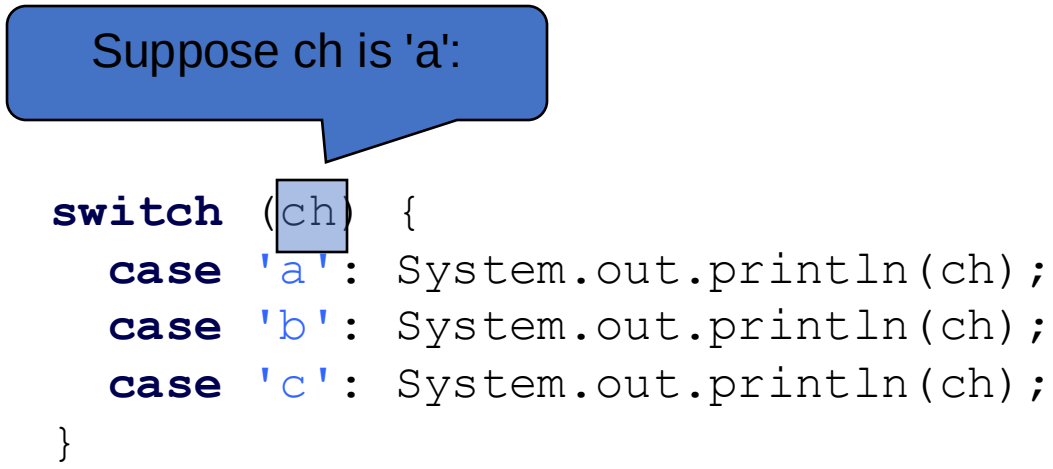
`switch` Statement Rules (continued)

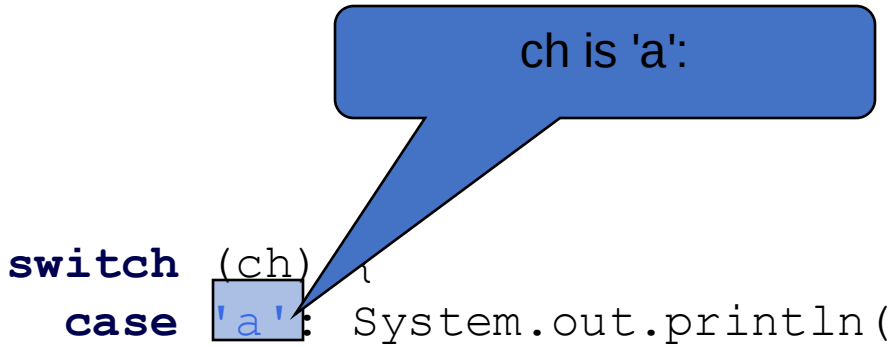
The keyword `break` is optional, but it should be used at the end of each case in order to terminate the remainder of the `switch` statement. If the `break` statement is not present, the next `case` statement will be executed.

The `default` case, which is optional, can be used to perform actions when none of the specified cases matches the switch-expression.

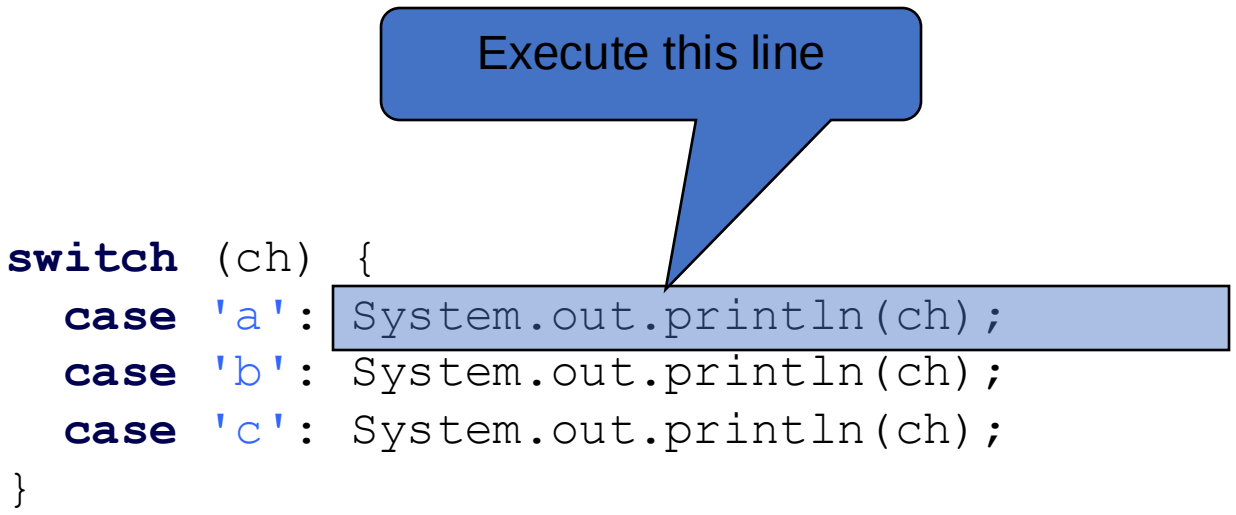
```
switch (switch-expression) {
  case value1:          statement(s)1;
        break;
  case value2:          statement(s)2;
        break;
  …
  case valueN:          statement(s)N;
        break;
  default:
        statement(s)-for-default;
}
```

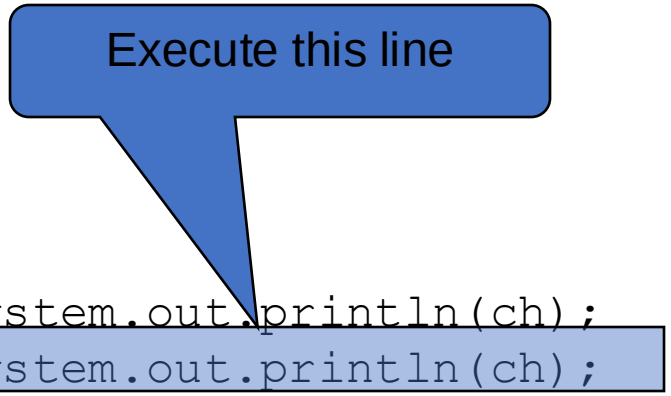
The `case` statements are executed in sequential order, but the order of the cases (including the `default` case) does not matter. However, it is good programming style to follow the logical sequence of the cases and place the default case at the end.


Suppose ch is 'a':
switch (ch) {
case 'a': System.out.println(ch);
case 'b': System.out.println(ch);
case 'c': System.out.println(ch);
}

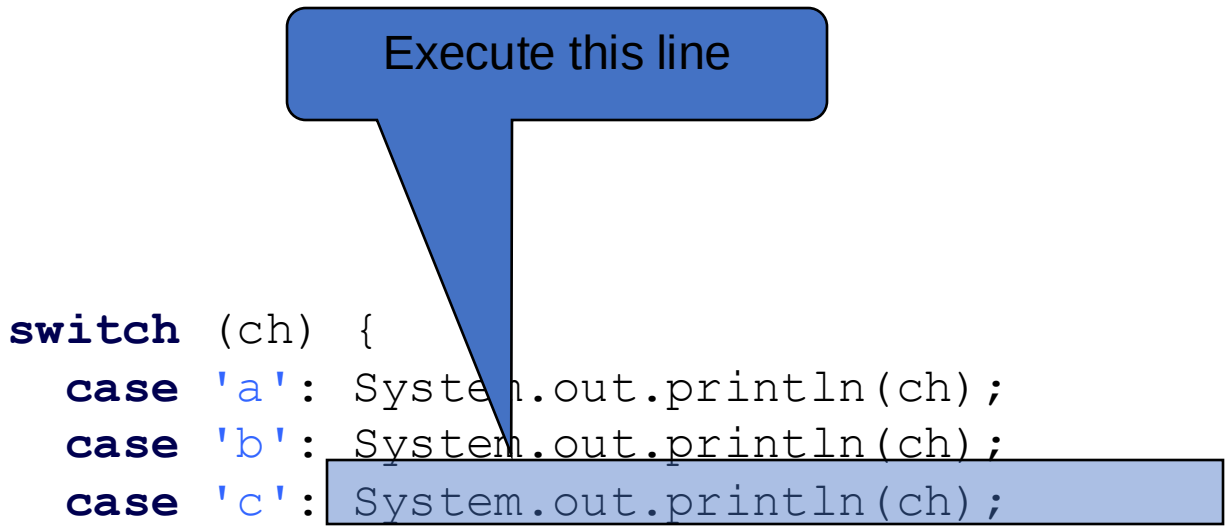


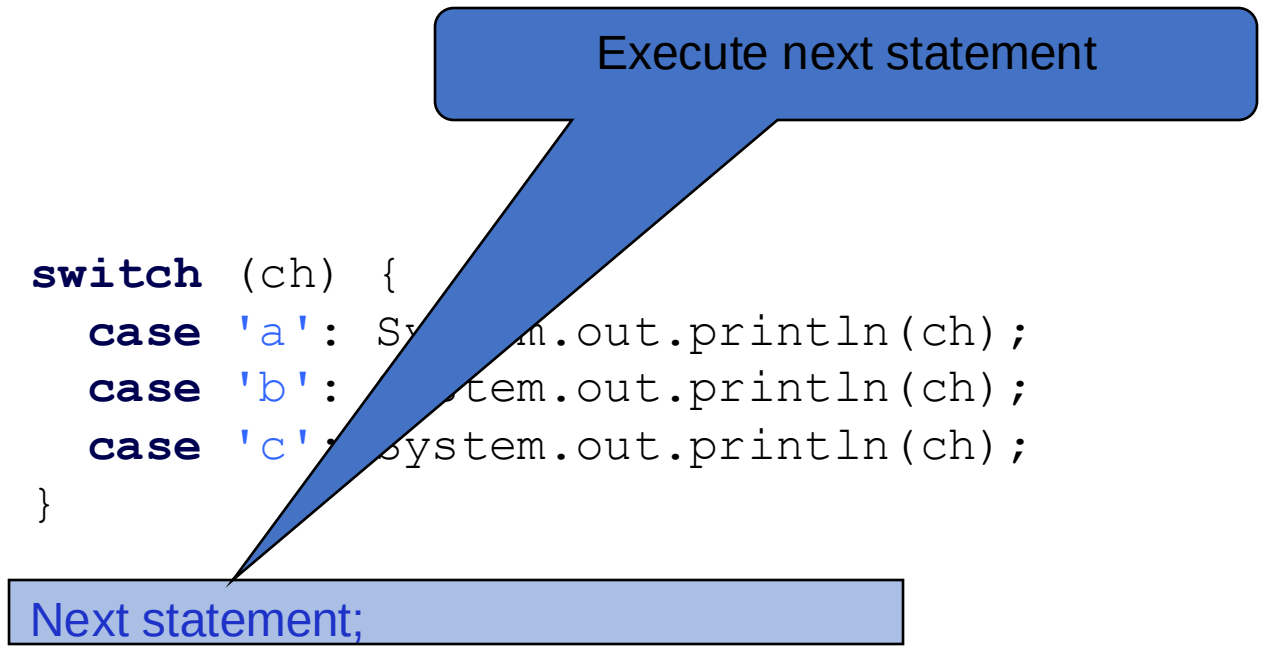

```
switch (ch) {
  case 'a': System.out.println(ch);
  case 'b': System.out.println(ch);
  case 'c': System.out.println(ch);
}
```

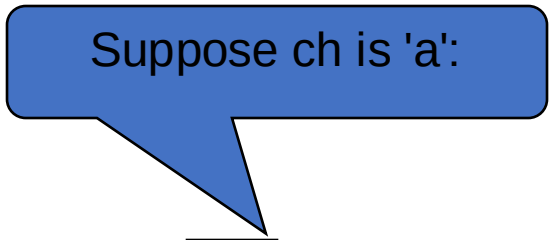

Execute this line
switch (ch) {
case 'a': System.out.println(ch);
case 'b': System.out.println(ch);
case 'c': System.out.println(ch);
}



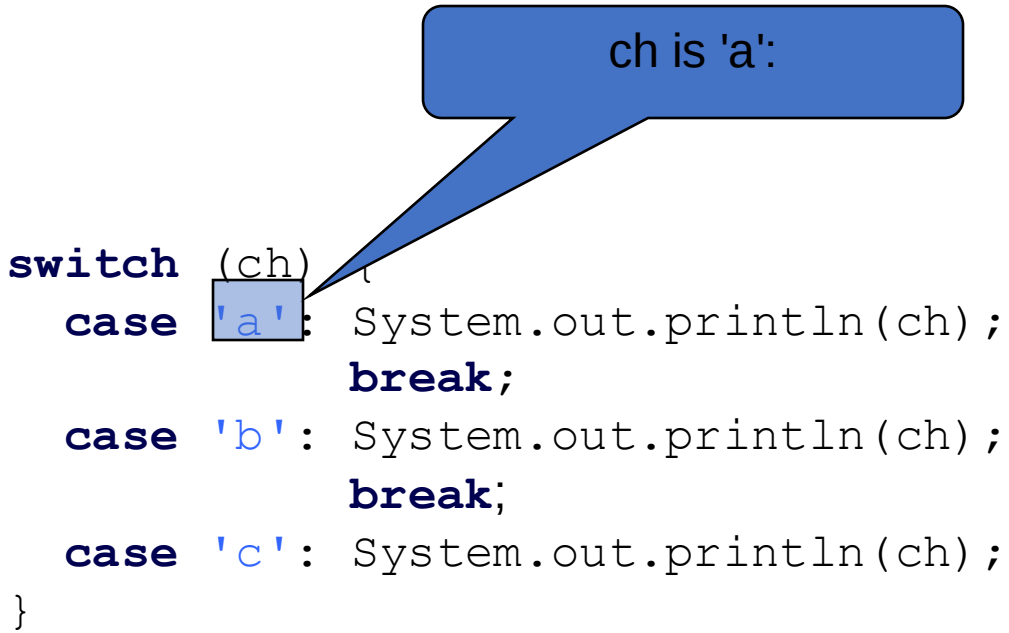

```
switch (ch) {
  case 'a': System.out.println(ch);
  case 'b': System.out.println(ch);
  case 'c': System.out.println(ch);
}
```

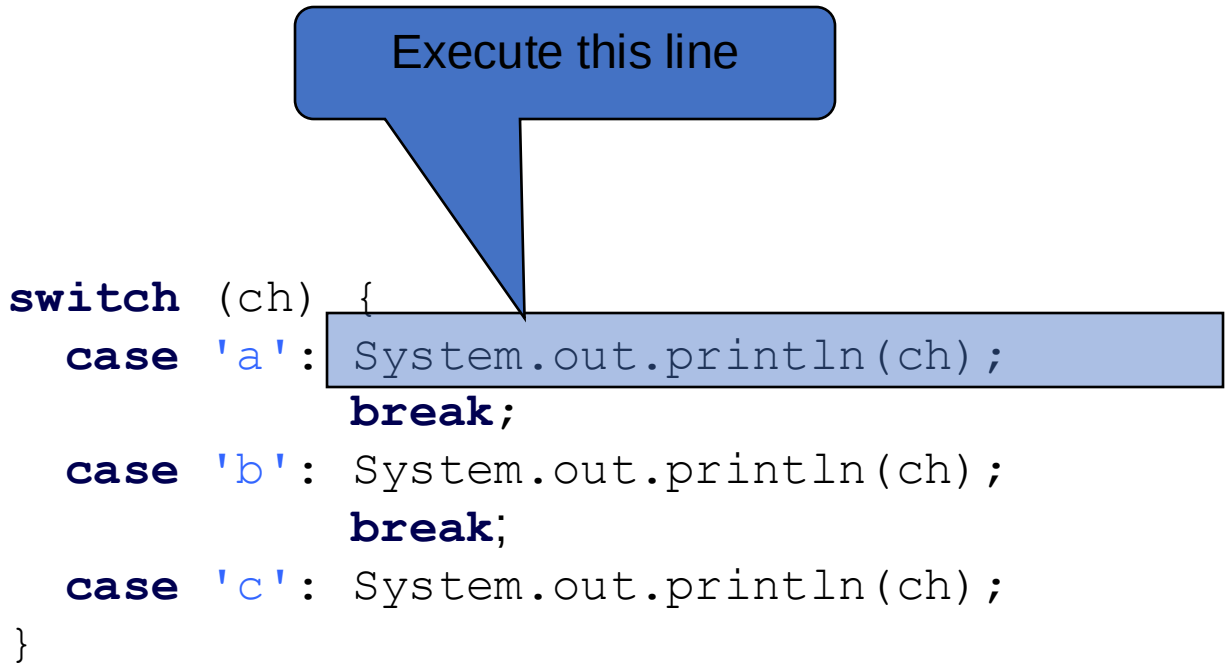

Execute this line
switch (ch) {
case 'a': System.out.println(ch);
case 'b': System.out.println(ch);
case 'c': System.out.println(ch);
}

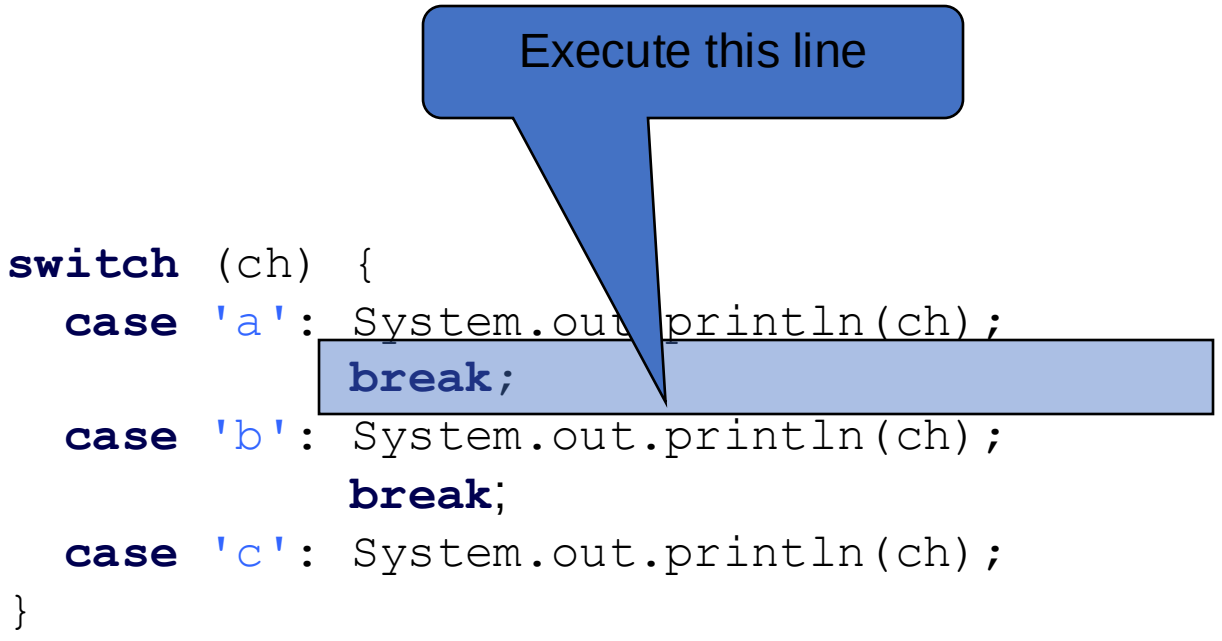

Execute next statement
switch (ch) {
case 'a': System.out.println(ch);
case 'b': System.out.println(ch);
case 'c': System.out.println(ch);
}
Next statement;

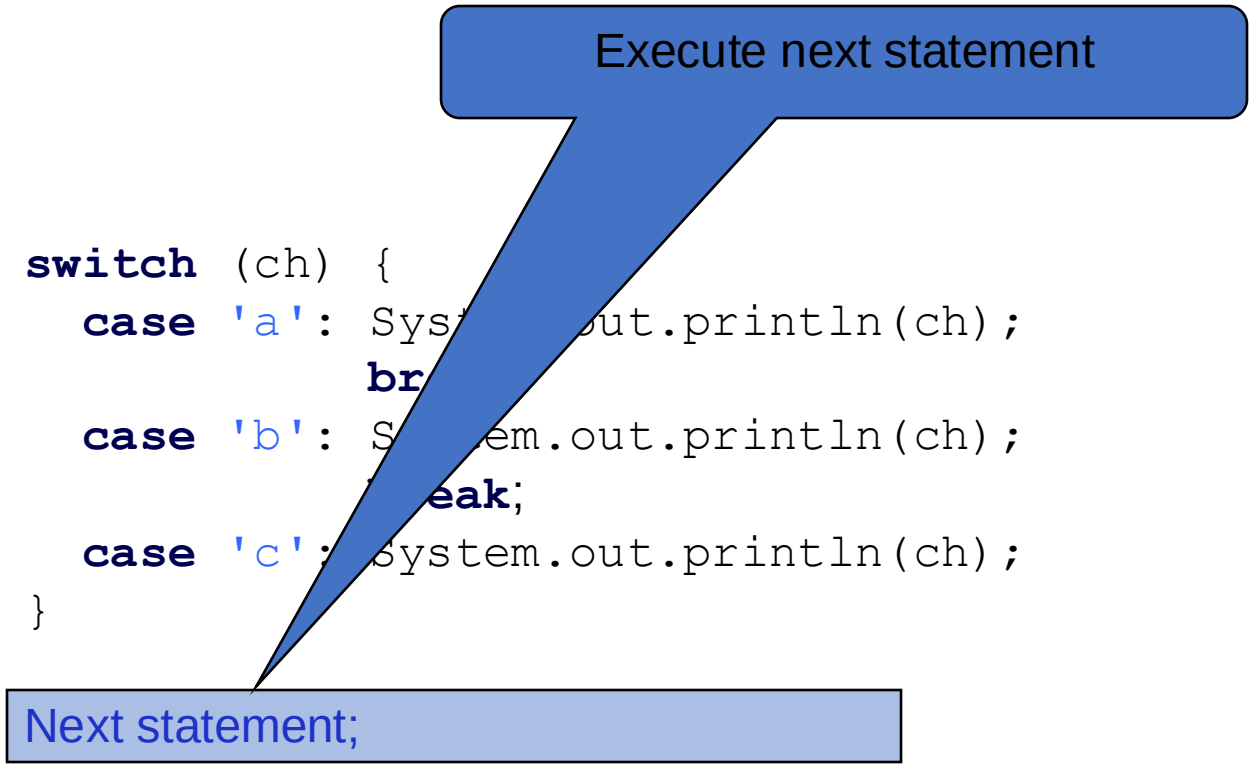



```
switch (ch) {
  case 'a': System.out.println(ch);
            break;
  case 'b': System.out.println(ch);
            break;
  case 'c': System.out.println(ch);
}
```


ch is 'a':
switch (ch) {
case 'a': System.out.println(ch);
break;
case 'b': System.out.println(ch);
break;
case 'c': System.out.println(ch);
}


Execute this line
switch (ch) {
case 'a': System.out.println(ch);
break;
case 'b': System.out.println(ch);
break;
case 'c': System.out.println(ch);
}


Execute this line
switch (ch) {
case 'a': System.out.println(ch);
break;
case 'b': System.out.println(ch);
break;
case 'c': System.out.println(ch);
}


Execute next statement
switch (ch) {
case 'a': Syst out.println(ch);
br
case 'b': S em.out.println(ch);
eak;
case 'c': System.out.println(ch);
}
Next statement;

Recall:

Write a program that reads a one-digit number from the user then prints on the screen the digit in letters.

```
  int digit;
  System.out.println("Enter a single digit number:");
  digit = input.nextInt();

  if (digit == 0){
        System.out.println(" ZERO ");
}else if (digit==1){
        System.out.println(" ONE ");
}else if (digit==2){
        System.out.println(" TWO ");
        …and so on….
}else{
        System.out.println(" Not a single-digit number");
     }
```

Rewriting Example using `switch-case`

```
int digit;
System.out.println("Enter a single digit number");
digit = input.nextInt();
switch(digit)
{
        case 0:
                System.out.println("ZERO");
                break;
        case 1:
                System.out.println("ONE");
                break;
        case 2:
                System.out.println("TWO");
                break;
        …and so on …..
        default: System.out.println("Not a single digit number");
  }
```

Switch program example

- Assume a clothing store might offer a discount that depends on the quality of the goods:
  - Class 'A' goods are not discounted at all.
  - Class 'B' goods are discounted 10%.
  - Class 'C' goods are discounted 20%.
  - anything else is discounted 30%.

```
double discount;
char code = 'B' ;
switch ( code ) {
        case 'A':
                discount = 0.0;
                break;
        case 'B':
                discount = 0.1;
                break;
        case 'C':
                discount = 0.2;
                break;
        default:
                discount = 0.3;
}
```

Loops

- Used for ***repetition***
- ***while*** loops or ***do while*** loops
  - When number of iterations is ***unknown ahead of time***
  - Condition is updated using a statement inside the loop ***body***

Loops

- Used for ***repetition***
- ***while*** loops or ***do while*** loops
  - When number of iterations is ***unknown ahead of time***
  - Condition is updated using a statement inside the loop ***body***

```
while (condition){
    Statements
    Condition_Update_statement
}
```

```
int x = 100;
while (x > 0){
    System.out.println("ICS 240");
    x = x - 1;}
```

Loops

- Used for ***repetition***

- ***while*** loops or ***do while*** loops
  - When number of iterations is ***unknown ahead of time***
  - Condition is updated using a statement inside the loop ***body***

```
while (condition){
    Statements
    Condition_Update_statement
}
```

```
int x = 100;
while (x > 0){
    System.out.println("ICS 240");
    x = x - 1;}
```

- ***for*** loops
  - When number of iterations is ***known***

Loops

- Used for ***repetition***

- ***while*** loops or ***do while*** loops
  - When number of iterations is ***unknown ahead of time***
  - Condition is updated using a statement inside the loop ***body***

```
while (condition){
    Statements
    Condition_Update_statement
}
```

```
int x = 100;
while (x > 0){
    System.out.println("ICS 240");
    x = x - 1;}
```

- ***for*** loops
  - When number of iterations is ***known***

```
for (initial_value; condition; step){
    statements
}
```

```
for (int i=0 ; i < 6; i++ ){
System.out.println(i);
}
```

Q1

In the main method, declare and integer variable x. Using a loop, print a countdown from x to 0.

Methods

Why Write Methods?

- Methods are commonly used to break a problem down into small manageable pieces. This is called *divide and conquer*.
- Methods simplify programs.
- If a specific task is performed in several places in the program, a method can be written once to perform that task, and then be executed anytime it is needed. This is known as *code reuse*.

Choosing Method Name

- Each method should be limited to performing a single, well-defined task
- A method name should be concise name that expresses what the function does
- If you cannot choose a concise name then most probably your method is attempting to perform too many diverse tasks. It is usually best to break such a method into several smaller methods.
- Choosing meaningful method names and meaningful parameter names makes programs more readable and helps avoid excessive use of comments

`void` Methods and Value-Returning Methods

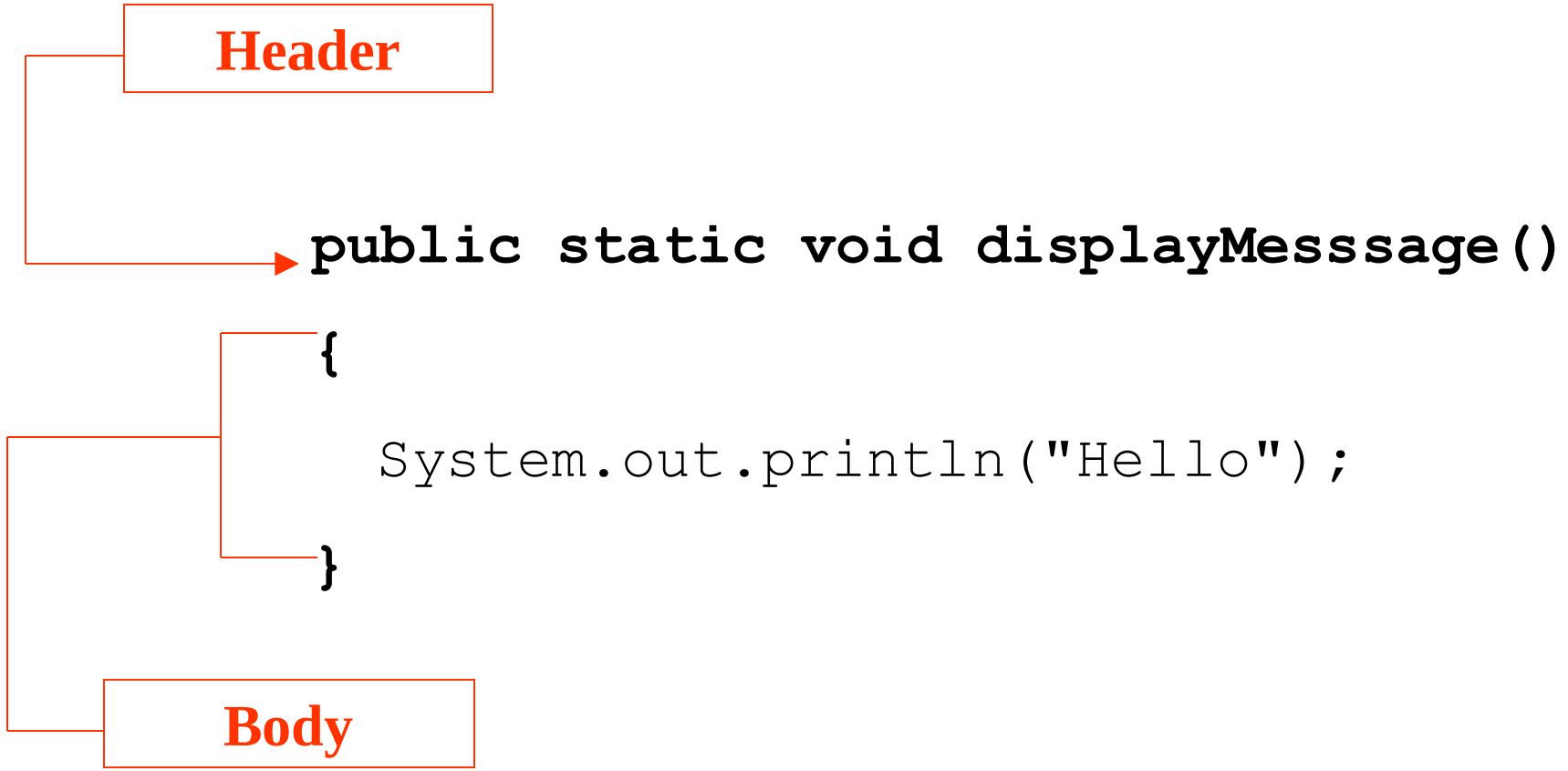
- A `void` method is one that simply performs a task and then terminates.

```
System.out.println("Hi!");
```

- A value-returning method not only performs a task, but also sends a value back to the code that called it.

```
int number = Integer.parseInt("700");
```

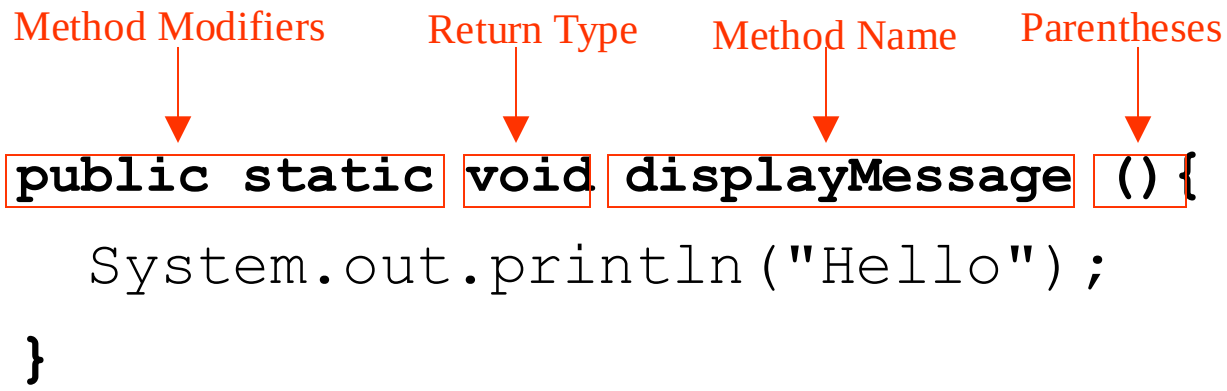
Two Parts of Method Declaration (void Method)



- To create a method, you must write a definition, which consists of a *header* and a *body*.
- The method header, which appears at the beginning of a method definition, lists several important things about the method, including the method's name.
- The method body is a collection of statements that are performed when the method is executed.

Parts of a Method Header

- Method modifiers
  - `public`—method is publicly available to code outside the class
  - `static`—method belongs to a class, not a specific object.
- Return type—`void` or the data type from a value-returning method
- Method name—name that is descriptive of what the method does
- Parentheses—contain nothing or a list of one or more variable declarations if the method is capable of receiving arguments.



Calling a Method

- A method executes when it is called.
- The `main` method is automatically called when a program starts, but other methods are executed by method call statements.

```
displayMessage();
```

- Notice that the method modifiers and the `void` return type are not written in the method call statement. Those are only written in the method header.

Documenting Methods

- A method should always be documented by writing comments that appear just before the method's definition.
- The comments should provide a brief explanation of the method's purpose.
- The documentation comments begin with **`/**` and end with `*/`**.

Passing Arguments to a Method

- Values that are sent into a method are called arguments.

```
System.out.println("Hello");
number = Integer.parseInt(str);
```

- The data type of an argument in a method call must correspond to the variable declaration in the parentheses of the method declaration. The parameter is the variable that holds the value being passed into a method.

- By using parameter variables in your method declarations, you can design your own methods that accept data this way.

Passing 5 to the `displayValue` Method

```
displayValue(5);
```

The argument 5 is copied into the parameter variable **num**.

```
public static void displayValue (int num)
{
 System.out.println("The value is " + num);
}
```

The method will display **`The value is 5`**

Argument and Parameter Data Type Compatibility

- When you pass an argument to a method, be sure that the argument's data type is compatible with the parameter variable's data type.
- Java will automatically perform widening conversions, but narrowing conversions will cause a compiler error.

```
double d = 1.0;

displayValue(d);
```

Passing Multiple Arguments

The argument 5 is copied into the **num1** parameter.

The argument 10 is copied into the **num2** parameter.

NOTE: Order matters!

```
showSum(5, 10);
```

```
public static void showSum(double num1, double num2)
{
   double sum;    //to hold the sum
   sum = num1 + num2;
   System.out.println("The sum is " + sum);
}
```

Arguments are Passed by Value

- In Java, all arguments of the primitive data types are *passed by value*, which means that only a copy of an argument's value is passed into a parameter variable.
- A method's parameter variables are separate and distinct from the arguments that are listed inside the parentheses of a method call.
- If a parameter variable is changed inside a method, it has no affect on the original argument.

More About Local Variables

- A local variable is declared inside a method and is not accessible to statements outside the method.
- Different methods can have local variables with the same names because the methods cannot see each other's local variables.
- A method's local variables exist only while the method is executing. When the method ends, the local variables and parameter variables are destroyed and any values stored are lost.
- Local variables are not automatically initialized with a default value and must be given a value before they can be used.

Returning a Value from a Method

- Data can be passed into a method by way of the parameter variables. Data may also be returned from a method, back to the statement that called it.

```
int num = Integer.parseInt("700");
```

- The string “700” is passed into the `parseInt` method.
- The `int` value 700 is returned from the method and assigned to the `num` variable.

Defining a Value-Returning Method

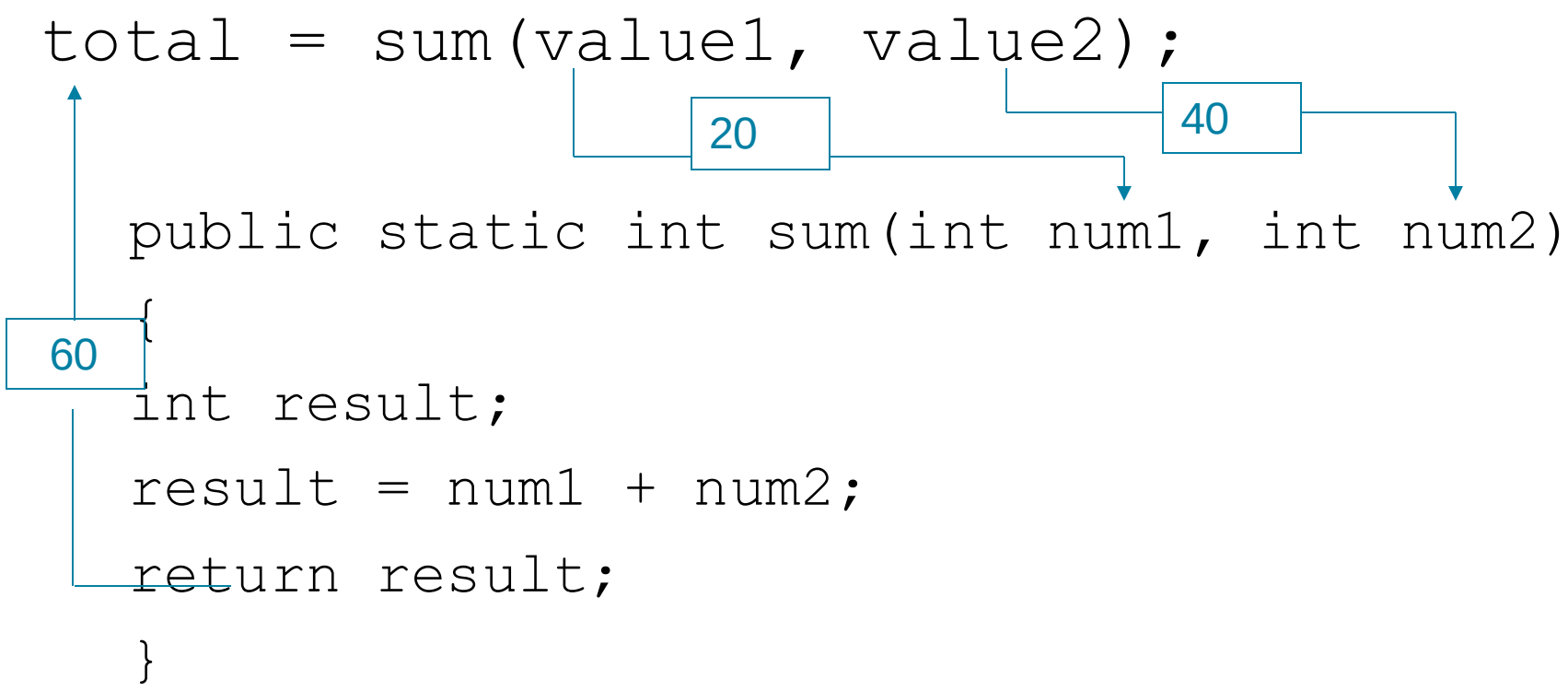
```
public static int sum(int num1, int num2)
{
 int result;
 result = num1 + num2;
 return result;
}
```

Return type

This expression must be of the same data type as the return type

The `return` statement causes the method to end execution and it returns a value back to the statement that called the method.

Calling a Value-Returning Method



Returning a `boolean` Value

- Sometimes we need to write methods to test arguments for validity and return true or false

```
 public static boolean isValid(int
number)
 {
    boolean status;
    if(number >= 1 && number <= 100)
       status = true;
    else
       status = false;
    return status;
 }
```

- Calling Code:

```
int value = 20;
if(isValid(value))
    System.out.println("The value is within range");
else
    System.out.println("The value is out of range");
```

Q2

a) Write a method to compute the average of two numbers.

- Will this method require any inputs?
- Will it have a return value?

b)Write a method that prints the current age of a student when given the year of birth.

- Will this method require any inputs?
- Will it have a return value?