

---

Ahmadu Bello University, Zaria, Nigeria

Department of Computer Science

COSC 211- Object Oriented Programming I

# Content

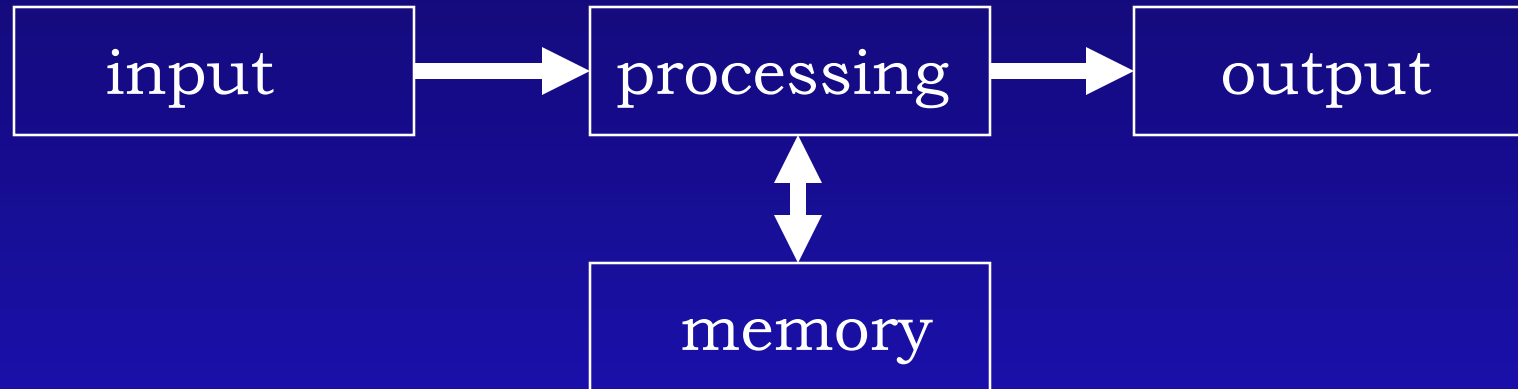
1. Computer Systems Anatomy
2. Overview of Programming Languages
3. Programming Languages Translation
4. Fundamental Data Types
5. Algorithms
6. Basic OO Concepts
7. Introduction to Strings
8. Introduction to Console I/O
9. Introduction to Classes I
10. Introduction to Classes II
11. Selection
12. Iteration
13. String Tokenization
14. More About Methods I
15. More About Methods II
16. 1-D Arrays
17. Dynamic Arrays
18. 2-D Arrays
19. Introduction to Recursion
20. Exception Handling
21. Text File I/O
22. Inheritance I
23. Inheritance II

# Introduction to Computer Systems

- Lecture Objectives:
  - The student should be able to identify and explain the major components of a computer system in terms of their functions.
- What is a Computer?
- Anatomy of a Computer System
- Computer Software
- Computer Hardware
- Fetch-Decode-Execute Cycle
- CPU Families
- Exercises

# What is a Computer System?

- A computer system is an electronic device which can input, process, and output data



- Input data of a computer may represent numbers, words, pictures etc
- Programs that control the operations of the computer are stored inside the computer

# Major Components of a Computer System

- A computer system consists of two main parts: **hardware** and **software**
- *Hardware* is the physical components and *software* is the non-physical components of a computer system.
- Computer hardware is divided into three major components:
  - 1. Input/Output (I/O) devices
  - 2. Computer memory
  - 3. The Central Processing Unit (CPU)
- Computer software is divided into two main categories:
  - 1. Systems software
  - 2. Applications software

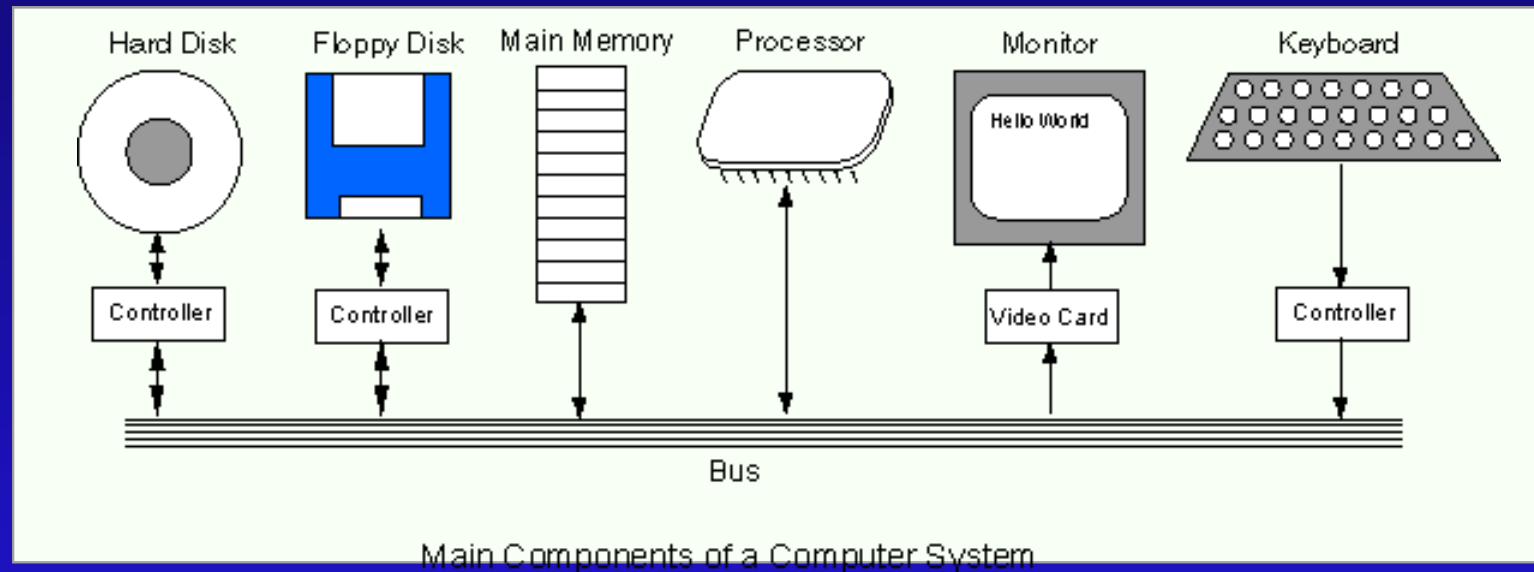
# Systems Software

- *System software* manages computer resources and makes computers easier to use
- Systems software can be divided into three categories:
  1. Operating System (OS)
    - Examples: Windows XP, UNIX and MacOS
  2. System support software
    - Examples: disk-formatting and anti-virus programs.
  3. System development software.
    - Example: *Language translators*.

# Applications Software

- An applications software enables a computer user to do a particular task
- Example applications software include:
  - Word processors
  - Game programs
  - Spreadsheets (or Excel sheets)
  - Database systems
  - Graphics programs
  - Multimedia applications

# Computer Hardware





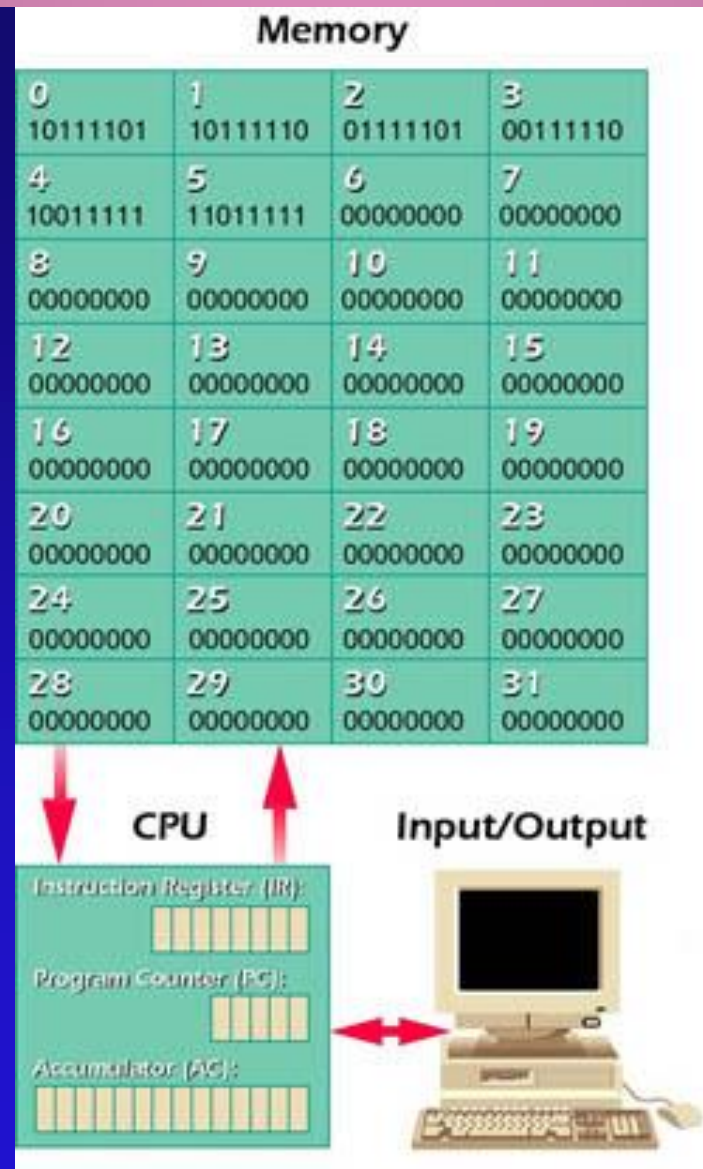
# I/O (Input/Output) Devices

- Input devices are used to enter programs and data into a computer system.
  - Examples: keyboard, mouse, microphone, and scanner.
- Output devices are where program output is shown or is sent.
  - Examples: monitor, printer, and speaker.
- I/O devices are slow compared to the speed of the processor.
  - Computer memory is faster than I/O devices: speed of input from memory to processor is acceptable.

# Computer Memory

- The main function of computer memory is to store software.
- Computer memory is divided into **primary memory** and **secondary memory**.
- Primary memory is divided into random access memory (RAM) and read-only memory (ROM):
  - The CPU can read and write to RAM but the CPU can read from ROM but cannot write to ROM
  - RAM is *volatile* while ROM is not.
- *Secondary memory*
  - Examples of secondary memory devices are: hard disks, floppy disks and CD ROMs

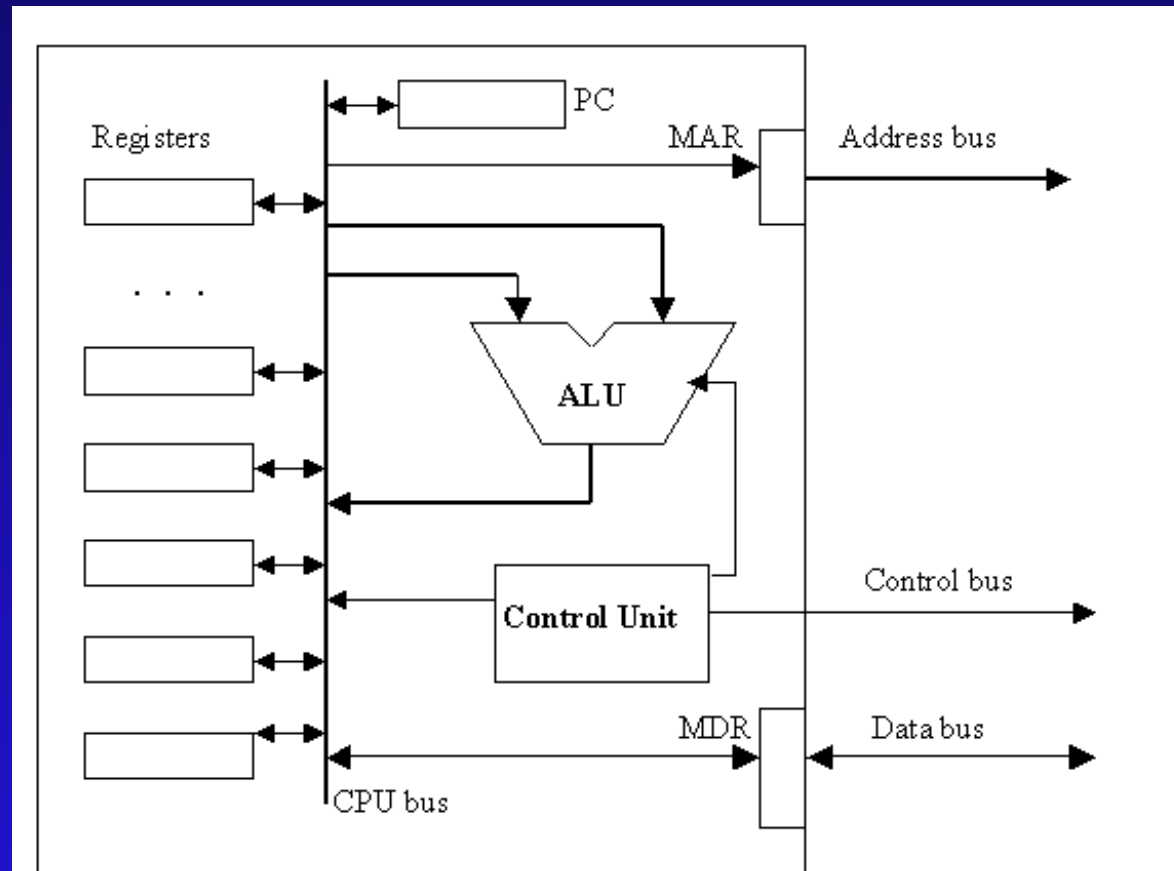
# Primary Memory



# The CPU

- The CPU is the "brain" of the computer system.
  - The CPU directly or indirectly controls all the other components.
- The CPU has a limited storage capacity.
- Thus, the CPU must rely on other components for storage.
- The CPU consists of:
  - 1. The Arithmetic and Logic Unit (ALU).
  - 2. The Control Unit (CU).
  - 3. Registers.
- The CPU components are connected by a group of electrical wires called the *CPU bus*.

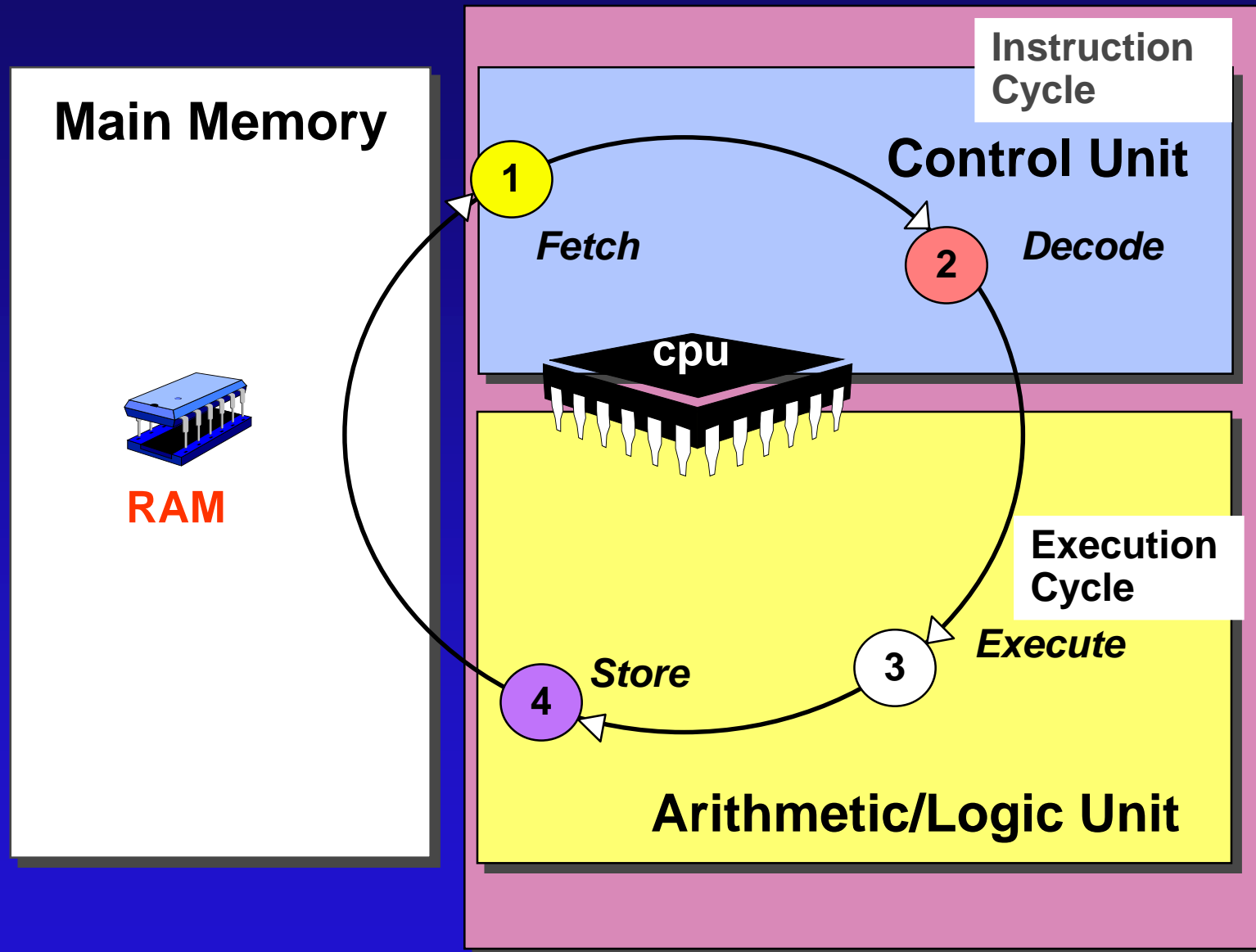
# The CPU (cont'd)



# Fetch Decode Execute Cycle

- The CPU continuously transfers data to and from memory
- Data transfer is done in units called *instructions* or *words*
- When a computer is switched on, the CPU continuously goes through a process called *fetch-decode-execute cycle*:
  - The Control Unit fetches the current instruction from memory, decodes it and instructs the ALU to execute the instruction.
  - The execution of an instruction may generate further data fetches from memory
  - The result of executing an instruction is stored in either a register or RAM

# Fetch-Decode-Execute Cycle (cont'd)



# CPU Families

- Different people understand different natural languages.
- Similarly, each processor family understands its own machine language.
- The fundamental difference between computers that are not compatible is in their processors.
- Here are some CPU families:
  - Pentium
  - Power PC
  - SPARC
- The question now is: Is it possible to write a single program that can be understood and correctly executed on machines with different processors?
  - We'll address this question in a subsequent lecture.



# Drill Questions

1. Write short notes explaining the functions of each of the following
  1. Computer memory
  2. The CPU
  3. Computer software
2. I/O devices can be used to input into and output from a computer system. Then, is computer memory necessary? Explain.
3. Since the OS controls the hardware and software in a computer system, which programs control the loading of an OS onto a computer system?
4. The system bus consists of three buses. Mention them and explain each of them briefly.
5. Since different CPUs understand different instructions, how are we able to exchange information between machines with different CPUs?

# Overview of Programming Paradigms

- Lecture Objectives:
  - Be able to explain the differences between programming languages and programming paradigms.
  - Be able to differentiate between low-level and high-level programming languages and their associated advantages and disadvantages
  - Be able to list four programming paradigms and describe their strengths and weaknesses.
- Introduction to Computer Programming
- Programming Languages
- Programming Paradigms
- Exercises

# Computer Programming

- The functions of a computer system are controlled by *computer programs*
- A computer program is a clear, step-by-step, finite set of instructions
- A computer program must be clear so that only one meaning can be derived from it,
- A computer program is written in a computer language called a *programming language*

# Programming Languages

➤ *There are three categories of programming languages:*

➤ 1. Machine languages.

➤ 2. Assembly languages.

➤ 3. High-level languages.

➤ Machine languages and assembly languages are also called low-level languages

# Programming Languages (cont'd)

- *A Machine language program* consists of a sequence of zeros and ones.
  - Each kind of CPU has its own machine language.
- Advantages
  - Fast and efficient
  - Machine friendly
  - No translation required
- Disadvantages
  - Not portable
  - Not programmer friendly

# Assembly Language

- *Assembly language programs* use mnemonics to represent machine instructions
- Each statement in assembly language corresponds to one statement in machine language.
- Assembly language programs have the same advantages and disadvantages as machine language programs.
- Compare the following machine language and assembly language programs:

8086 Machine language program for var1 = var1 + var2 ;	8086 Assembly program for var1 = var1 + var2 ;
1010 0001 0000 0000 0000 0000	MOV AX , var1
0000 0011 0000 0110 0000 0000 0000 0010	ADD AX , var2
1010 0011 0000 0000 0000 0000	MOV var1 , AX

# High-Level Programming Languages

- A high-level language (HLL) has two primary components
  - (1) a set of built-in *language primitives and grammatical rules*
  - (2) a translator
- *A HLL language program* consists of English-like statements that are governed by a strict *syntax*.
- Advantages
  - Portable or *machine independent*
  - Programmer-friendly
- Disadvantages
  - Not as efficient as low-level languages
  - Need to be translated
- Examples : C, C++, Java, FORTRAN, Visual Basic, and Delphi.

# Programming Paradigms

- Why are there hundreds of programming languages in use today?
  - Some programming languages are specifically designed for use in certain applications.
  - Different programming languages follow different approaches to solving programming problems
- A *programming paradigm* is an approach to solving programming problems.
- A programming paradigm may consist of many programming languages.
- Common programming paradigms:
  - Imperative or Procedural Programming
  - Object-Oriented Programming
  - Functional Programming
  - Logic Programming



# Programming Paradigms: Imperative

- In this paradigm, a program is a series of *statements* containing *variables*.
- Program execution involves changing the memory contents of the computer continuously.
- Example of imperative languages are: C, FORTRAN, Pascal, COBOL etc
- Advantages
  - Low memory utilization
  - Relatively efficient
  - The most common form of programming in use today.
- Disadvantages
  - Difficulty of reasoning about programs
  - Difficulty of parallelization.
  - Tend to be relatively low level.

# Programming Paradigms: Object-Oriented

- A program in this paradigm consists of *objects* which communicate with each other by *sending messages*
- Example object oriented languages include: Java, C#, Smalltalk, etc
- Advantages
  - Conceptual simplicity
  - Models computation better
  - Increased productivity.
- Disadvantages
  - Can have a steep learning curve, initially
  - Doing I/O can be cumbersome

# Programming Paradigms: Functional

- A program in this paradigm consists of *functions* and uses functions in a similar way as used in mathematics
  - Program execution involves functions calling each other and returning results. There are no variables in functional languages.
- Example functional languages include: ML, Miranda<sup>TM</sup>, Haskell
- Advantages
  - Small and clean syntax
  - Better support for reasoning about programs
  - They allow functions to be treated as any other data values.
  - They support programming at a relatively higher level than the imperative languages
- Disadvantages
  - Difficulty of doing input-output
  - Functional languages use more storage space than their imperative cousins

# Programming Paradigms: Logic

- A program in the logic paradigm consists of a set of *predicates* and *rules of inference*.
  - Predicates are statements of fact like the statement that says: water is wet.
- Rules of inference are statements like: If X is human then X is mortal.
  - The predicates and the rules of inference are used to prove statements that the programmer supplies.
- Example: Prolog
- Advantages
  - Good support for reasoning about programs
  - Can lead to concise solutions to problems
- Disadvantages
  - Slow execution
  - Limited view of the world
    - That means the system does not know about facts that are not its predicates and rules of inference.
  - Difficulties in understanding and debugging large programs

# Which Programming Paradigm is Best?

- Which of these paradigms is the best?
- The most accurate answer is that there is no best paradigm.
- No single paradigm will fit all problems well.
- Human beings use a combination of the models represented by these paradigms.
- Languages with features from different paradigms are often too complex.
- So, the search of the ultimate programming language continues!

# Review Questions

1. List two advantages and two disadvantages of low-level languages.
2. Explain the similarities and differences between an assembly language and a machine language.
3. Mention the programming paradigm to which each of the following languages belongs: Visual Basic, Java, C#, Haskell, Lisp, Prolog, Pascal.
4. Which programming paradigms give better support for reasoning about programs?
5. Which programming paradigms give better support for doing I/O?

# Programming Languages Translation

- Lecture Objectives:
  - Be able to list and explain five features of the Java programming language.
  - Be able to explain the three standard language translation techniques.
  - Be able to describe the process of translating high-level languages.
  - Understand the concept of virtual machines and how Java uses this concept to achieve platform independence.
  - Understand the structure of simple Java programs
- Java Programming Language
- Translating High-level Languages
- The Java Virtual Machine
- Java Program Structure
- Exercises

# The Java Programming Language

- The object-oriented paradigm is becoming increasingly popular compared to other paradigms.
- The Java programming language is perhaps the most popular object-oriented language today.
- Here are some reasons why Java is popular:
  - *1. Simple.*
    - Compared to many modern languages, the core of the Java language is simple to master.
  - *2. Object-oriented.*
    - Java is an object-oriented language and therefore it has all the benefits of OO languages described earlier.
  - *3. Secure.*
    - The Java system controls what parts of your computer a program access.
  - *4. Architecture neutral.*
    - A Java program will run identically on every platform. We will explain how Java achieves this portability later in this session.
  - *5. Java is for Internet applications*
    - Java was designed so that it can download programs over the Internet and execute them.



# High Level Language Translators

- As mentioned earlier, one of the disadvantages of a high-level language is that it must be translated to machine language.
- High-level languages are translated using language translators.
- A *language translator* is that translates a high-level language program or an assembly language program into a machine language program.
- There are three types of translators:
  - 1. Assemblers.
  - 2. Compilers.
  - 3. Interpreters.

# High Level Language Translators

## ➤ *Assemblers*

- An assembler is a program that translates an assembly language program, written in a particular assembly language, into a particular machine language.

## ➤ *Compilers*

- A compiler is a program that translates a high-level language program, written in a particular high-level language, into a particular machine language.

## ➤ *Interpreters*

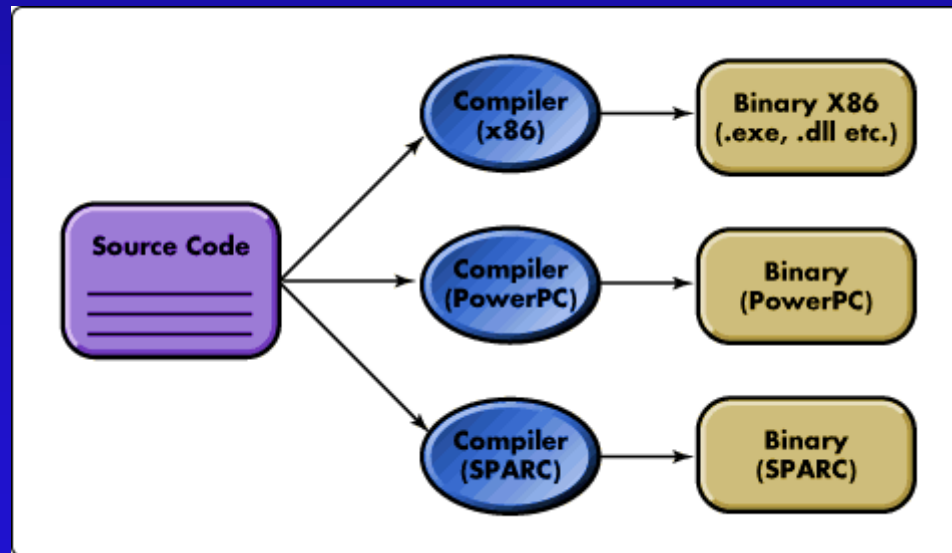
- An interpreter is a program that translates a high-level language program, one instruction at a time, into machine language.
- As each instruction is translated it is immediately executed.
- Interpreted programs are generally slower than compiled programs because compiled programs can be optimized to get faster execution.

## ➤ Note that:

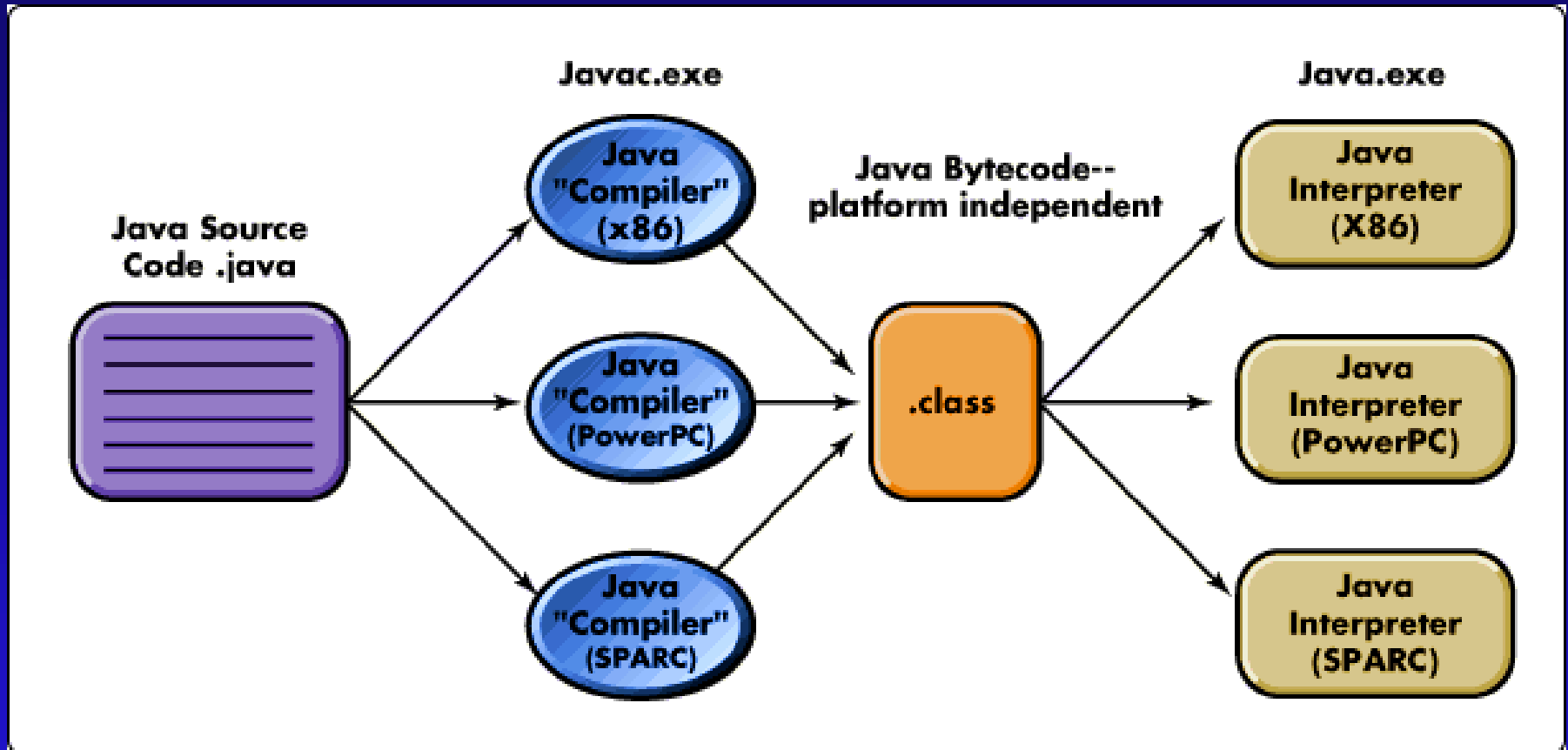
- Some high-level languages are compiled while others are interpreted.
- There are also languages, like Java, which are first compiled and then interpreted

# Compilation Process: Traditional Compilers

- In the traditional compilation process, the compiler produces machine code for a specific family of processors
- For example, given a source program, a compiler for the x86 family of processors will produce binary files for this family of processors.
- A disadvantage of this compilation method is that the code produced in each case is not portable.
- To make the resulting code portable, we need the concept of a virtual machine as we discuss in the following page.



# Compilation Process: Java Compilers



# Java Virtual Machine

- Instead of producing a processor-specific code, Java compilers produce an intermediate code called *bytecode*.
- *The bytecode is also a binary code but is not specific to a particular CPU.*
- A Java compiler will produce exactly the same bytecode no matter what computer system is used.
- The Java bytecode is then interpreted by the Java Virtual Machine (JVM) interpreter.
- Notice that each type of computer system has its own Java interpreter that can run on that system.
- This is how Java achieves compatibility.
  - It does not matter on what computer system a Java program is compiled, provided the target computer has a Java Virtual machine.

# Structure of Simple Java Programs

- The following figure shows a simplified structure of Java programs. We will consider a more detailed structure of Java programs later in this course.

```
public class ClassName {  
    public static void main(String[] args ) {  
        statement 1  
        statement 2  
        * * *  
        statement N  
    }  
}
```

- A Java program consists of essential elements called classes. The classes in a program are used to create specific things called objects.

# Source Program: Example

- This is an example of a Java source program

```
public class Greeting {  
    public static void main(String[] args  
    ) {System.out.println("Good Morning.");  
    }  
}
```

- You must type this program and save it in a file named **Greeting.java**
- Java is case sensitive and has a free-form layout
- *The words public, class, static, void, main and etc are called reserved or keyword words*
- The meaning of the words is fixed by the language. For now, they must appear in the places shown.

# Structure of Simple Java Programs (cont'd)

- By contrast, the word **Greeting**, varies from program to program.
  - What exactly goes there is chosen by the programmer.
- The first line, **public class Greeting**, starts a new class called **Greeting**.
- Classes are a fundamental concept in java. Their role is as factories for objects.
- But here we are using the **Greeting** class as a container for our program's instructions.
- Java requires that all program instructions be placed inside **methods** and that every method must be placed inside a class.
- Thus we must define methods that would contain our instructions and a class that holds the methods.
- In our program, **main()** is our only method while **Greeting** is our class.



# Structure of Simple Java Programs (cont'd)

- At this point, simply regard

```
public class ClassName{  
    ...  
}
```

as a necessary part of the plumbing that is required to write a java program.

- The construction

```
public static void main (String[] args){  
    ...  
}
```

is where we define the method **main ( )**

# Structure of Simple Java Programs (cont'd)

- The parameter `String[] args` is a required part of the `main` method.
- At this time, simply consider

```
public class ClassName{  
    public static void main (String[] args){  
        ...  
    }  
}
```

as yet another part of the plumbing for the time being, simply put all instructions between the curly braces `{ }` of the `main()` method.

- There is no limit to the number of instructions that can be placed inside the body of the `main` method.
- Our program contains only one instruction, that is

```
System.out.println ("Good Morning");
```

# Structure of Simple Java Programs: Printing Output

- The instruction,  
`System.out.println("Good Morning");`  
prints a line of text namely "Good Morning".
- A program can send the string : to a window, to a file, to a networked computer.
  - However, our program prints the string to the terminal window. That is the monitor.
- Terminal window is represented in java by an object called **out** and out object is contained in the **System** class.
- The **System** class contains useful objects and methods to access System resources. To use the out object in the System class, you must to refer to it as **System.out**
- To use **System.out** object, specify what you want to do to it. In this case, you want to print a line of text. The **println()** method carries out this.

# Structure of Simple Java Programs: Printing Output

- The `println()` method prints a string or a number and then start a new line. For example

- The sequence of statements:

```
System.out.println ("Good");  
System.out.println ("Morning.");
```

prints two lines of text

```
Good  
Morning.
```

- The statement:

```
System.out.println (3 + 4);
```

prints the number

```
7
```

- There is a second method called `print()`, which print an item without starting a new line. For example

```
System.out.print ("Good");  
System.out.println (" Morning.");
```

prints a single line

```
Good Morning.
```

# Escape Sequences

- An escape sequence is a special two-character sequence representing another character. Suppose you want to display a string containing quotation marks, such as

```
Hello, "World"!
```

- you can't use

```
System.out.println("Hello, "World"!");
```

- To display quotation marks in the above example, you should write:

```
System.out.println("Hello, \"World\"!");
```

- Therefore, `\` is an escape sequence representing quotation mark `"`.

- Similarly, `\n` an escape sequence representing a new line or line feed character. Printing a new line starts of a new line on the display. For example the statement:

```
System.out.println("*\n ** \n *** \n);
```

# Escape Sequences

- An escape sequence is a special two-character sequence representing another character. Suppose you want to display a string containing quotation marks, such as

`Hello, "World"!`

- You shouldn't use

`System.out.println("Hello, "World"!");`

- To display quotation marks in the above example, you should write:

`System.out.println("Hello, \"World\"!");`

- Therefore, `\` is an escape sequence representing quotation mark `"`.
- Similarly, `\n` an escape sequence representing a new line or line feed character. Printing a new line starts of a new line on the display.
- For example the statement:

`System.out.println("*\n ** \n *** \n);`

prints the following

`*`

`**`

`***`

## Example 2

An agent sold a property worth N50000. If the buyer pays the agent 7% of the sale amount, find the agent's commission and the total amount that the buyer must pay. The following program computes the agent's commission and the total amount the buyer must pay. Study the program and try to relate it with the above java fundamentals

you learned so far. What would be the output of the program?

```
public class Interest{  
    public static void main (String[] args) {  
        System.out.print ("The agent's commission is: " );  
        System.out.println (50000 * 0.07 );  
        System.out.print ("Total amount the buyer pays is: ");  
        System.out.println (50000 + 50000*0.07);  
    }  
}
```

# Fundamental Data Types

- Primitive Data Types
- Variable declaration
- Numbers and Constants
- Arithmetic Operators
- Arithmetic Operator Precedence
- The Math Class
- Assignment statement
- Increment and Decrement operators
- Writing Algebraic Expressions in Java
- Math Functions: Examples
- Casting



# Primitive Data Types

- Java has eight **primitive** data types as described below.

Type	Size	Range
byte	1 byte	-128 to 127
short	2 bytes	-32,768 to 32,767
int	4 bytes	about -2 billion to 2billion
long	8 bytes	about -10E18 to +10E18
float	4 bytes	-3.4E38 to +3.4E38
double	8 bytes	1.7E308 to 1.7E308
char	2 bytes	A single character
boolean	1 byte	true or false

- Other information is represented in Java as Objects.

# Variable Declaration

- A variable can be declared to hold a data value of any of the primitive types.
- A variable is a **named memory location** in which a value is stored.
- A variable name is a sequence of letters and digits starting with a letter.

```
int        counter;  
int        numStudents = 583;  
long       longValue;  
long       numberOfAtoms = 1237890L;  
float      gpa;  
float      batchAverage = 0.406F;  
double     e;  
double     pi = 0.314;  
char       gender;  
char       grade = 'B';  
boolean    safe;  
boolean    isEmpty = true;
```

# Numbers and Constants

- By default, whole numbers are **int** and real numbers are **double**.
- However, we can append a letter at the end of a number to indicate its type.
- Upper and lower case letters can be used for 'float' (F or f), 'double' (D or d), and 'long' (l or L):
- Float and double numbers may be expressed in scientific notation: **number \* 10<sup>exponent</sup>** as:

*number E integerExponent* or *number e integerExponent*

```
float maxGrade = 100f;
double temp = 583d;
float temp = 5.5; // Error as 5.5 is double
float temp = 5.5f;
long y = 583L;
double x = 2.25e-6;
```

- One use of the modifier **final** is to indicate symbolic constants.
- By convention, symbolic constants are written in uppercase letters. Underscores separate words:

```
final double SPEED_OF_LIGHT = 3.0E+10;
final double CM_PER_INCH = 2.54;
final int MONTH_IN_YEAR = 12;
```

# Arithmetic Operators

➤ A simple arithmetic expression has the form:

op1 Operator op2

where:

Operator	Description
+	Adds op1 and op2
-	Subtracts op2 from op1
*	Multiplies op1 by op2
/	Divides op1 by op2
%	Remainder of dividing op1 by op2

# Arithmetic Operators (Cont'd)

- The operators give results *depending on the type of the operands*.
- If operand1 and operand2 are integer, then the result is also integer. But if either operand1 and/or operand2 is double, then the result is double.
- Examples:

Arithmetic expression	Value
1 / 2	0
86 / 10	8
86 / 10.0	8.6
86.0 / 10	8.6
86.0 / 10.0	8.6
86 % 10	6

# Arithmetic Operator Priority

- An expression is a sequence of variables, constants, operators, and method calls that evaluates to a single value.
- Arithmetic expressions are evaluated according to **priority rules**.
- All binary operators are evaluated in left to right order.
- In the presence of parenthesis, evaluation starts from the innermost parenthesis.

Operators	Priority (Precedence)
+ - (unary)	1
* / %	2
+ - (binary)	3

Expression	Value
$3 + 7 \% 2$	4
$(2 - 5) * 5 / 2$	-7
$2 - 5 + 3$	0

# The Math class

- Many mathematical functions and constants are included in the Math class of the Java library. Some are:

Function /constant	Meaning
<code>sqrt(x)</code>	Returns the square root of x.
<code>abs(x)</code>	Returns the absolute value of x, x can be double, float, int or long.
<code>cos(a), sin(a), tan(a)</code>	Returns the trigonometric cosine/sine/tangent of an angle given in radians
<code>exp(x)</code>	Returns the exponential number e raised to the power of x
<code>log(x)</code>	Returns the natural logarithm (base e) of x
<code>max(x, y) , min(x, y)</code>	Returns the greater/smaller of two values, x and y can be double, float, int or long
<code>pow(x, y)</code>	Returns $x^y$
<code>PI</code>	The approximate value of PI

- Syntax to call a function in the Math class: `Math.functionName(ExpressionList)`
- Syntax to access a constant in the Math class: `Math.ConstantName`
- Example: `Math.PI * Math.max(4 * y, Math.abs(x - y))`

# Assignment Statement

## ➤ Syntax:

```
variable = expression;
```

- The **expression** on the right –hand side is evaluated and the result is assigned to the **variable** on the left-hand side.
- The left-hand side must be a variable.
- Examples:

```
a = 5;
```

```
b = a;
```

```
b = b + 12;    // valid: assignment operator, =, is not an equality operator
```

```
c = a + b;
```

```
a + b = c;    // invalid: left side not a variable
```



# Assignment Statement (cont'd)

- To exchange (or to swap) the contents of two variables, a third variable must be used.
- Example:

```
double x = 20.5, y = -16.7, temp;  
temp = x;  
x = y;  
y = temp;
```

# Short Hand Assignment Operators

- Java provides a number of short hand assignment operators:

Short-Form	Equivalent to
<code>op1 += op2</code>	<code>op1 = op1 + op2</code>
<code>op1 -= op2</code>	<code>op1 = op1 - op2</code>
<code>op1 *= op2</code>	<code>op1 = op1 * op2</code>
<code>op1 /= op2</code>	<code>op1 = op1 / op2</code>
<code>op1 %= op2</code>	<code>op1 = op1 % op2</code>

- Example:

```
a += 5;    // equivalent to a = a + 5;
```

# Increment and Decrement Operators

- Increment/decrement operations are very common in programming. Java provides operators that make these operations shorter.

Operator	Use	Description
++	op++	Increments op by 1;
++	++op	Increments op by 1;
--	op--	Decrements op by 1;
--	--op	Decrements op by 1;

- Example:

```
int  y = 20; x = 10, z;  
y++ ;  
z = x + y;
```

# Writing Algebraic Expressions in Java

- All operators must be explicit especially multiplications.
- For a fraction, you must use parenthesis for the numerator or denominator if it has addition or subtraction.

Algebraic expression	Java expression
$Z = \frac{4X + Y}{X_2} - 2Y$	<code>z = (4 * x + y) / x2 - 2 * y</code>
$Z = \sqrt{X + Y^2}$	<code>z = Math.sqrt(x + Math.pow(y, 2))</code>

# Example1

- The following example computes the roots of a quadratic equation using the formula:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Algorithm:

- »  $a = 1$
- »  $b = -5$
- »  $c = 6$
- »  $\text{root1} = (-b + \sqrt{b * b - 4 * a * c}) / (2 * a)$
- »  $\text{root2} = (-b - \sqrt{b * b - 4 * a * c}) / (2 * a)$
- » `print root1, root2`

- Java code:

```
public class QuadraticEquation {  
    public static void main(String[] args) {  
        double a = 1, b = -5, c = 6;  
        double root1 = (-b + Math.sqrt(b*b - 4*a*c)) / (2*a);  
        double root2 = (-b - Math.sqrt(b*b - 4*a*c)) / (2*a);  
        System.out.println("The roots are: " + root1 + " , " + root2);  
    }  
}
```

# Example2

- The following example calculates the area and circumference of circle.
- Algorithm:
  - »  $\text{radius} = 3$
  - »  $\text{area} = \pi * \text{radius}^2$
  - »  $\text{circumference} = 2 * \pi * \text{radius}$
  - » print area, circumference

```
public class Circle {  
    public static void main(String[] args) {  
        double area, circumference;  
        int radius = 3;  
        area = Math.PI * Math.pow(radius, 2);  
        circumference = 2 * Math.PI * radius;  
        System.out.println("Area = " + area + " square cm");  
        System.out.println("Circumference = " + circumference + " cm");  
    }  
}
```

# Casting

- A cast is an explicit conversion of a value from its current type to another type.
- The syntax for a cast is: `(type) expression`
- Two of the cases in which casting is required are:

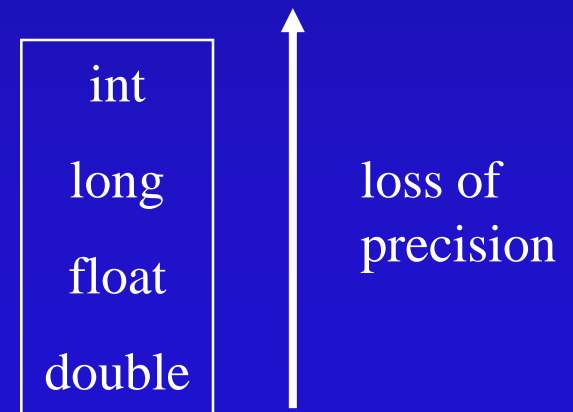
## 1. To retain the fractional part in integer divisions:

```
int sumOfGrades;  
int numberOfStudents;  
double average;  
// . . .  
average = (double) sumOfGrades / numberOfStudents;
```

Note: The cast operator has higher priority than all arithmetic operators.

## 2. When a type change will result in loss of precision

```
int sum = 100;  
  
float temp = sum; //temp now holds 100.0  
float total = 100F;  
  
int temp = total; // ERROR  
  
int start = (int) total;
```



# Algorithms and Problem Solving

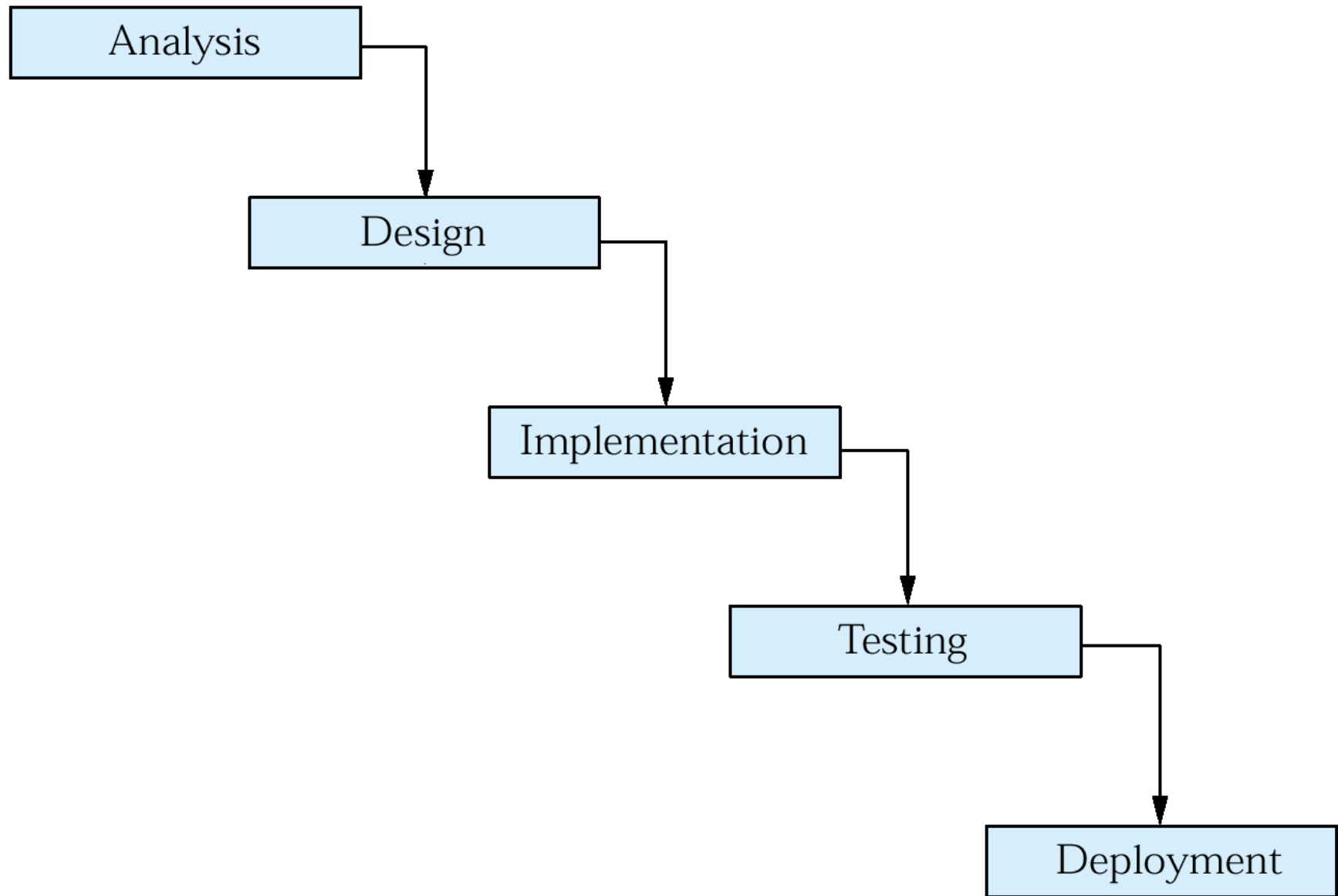
- Problem Solving
- Problem Solving Strategy
- Algorithms
- Sequential Statements
- Examples



# Problem Solving

- Solving a problem means that we know the way or the method to follow manually from the start till the end.
- Having the method known, the same method is used by the computer to solve the problem but faster and with higher precision.
- If we do not know how to solve a problem ourselves, the computer will not be of any help in this regard.
- The strategy for solving a problem goes through the following stages:
  - **Analysis**: in this stage, we should find what the problem should do.
  - **Design** : the way or method of how your problem is solved is produced
  - **Implementation**: the method found in design is then coded here in a given programming language.
  - **Testing**: here we verify that the program written is working correctly
  - **Deployment** : finally the program is ready to use

# Problem Solving Strategy



# Algorithms

- An algorithm is a sequence of instructions that solve a problem.
- An algorithm has the following properties:
  - No ambiguity in any instruction
  - No ambiguity which instruction is next
  - Finite number of instructions
  - Execution must halt
- The description of the instructions can be given in English like statements called pseudo-code
- The flow control of instructions has three types:
  - Sequential
  - Selection
  - Iteration

# Sequential Statements

- Instructions in this type of flow control are executed one after the other in sequence
- These statements include:
  - Assignment statement
  - Method calls

# Example1

- Write a program that assigns the Cartesian coordinates of two points (x1, y1) and (x2, y2) and displays the distance between them using the following formula.

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

- Algorithm:
  - » Get a value for **x1** (e.g., x1 = 1)
  - » Get a value for **y1** (e.g., y1 = 1)
  - » Get a value for **x2** (e.g., x2 = 4)
  - » Get a value for **y2** (e.g., y2 = 6)
  - » Calculate the **distance** using the formula:
    - distance = sqrt( (x2 – x1) ^ 2 + (y2 – y1) ^ 2 )
  - » print **distance**

# Example1 (cont'd)

## ➤ Java code:

```
public class Distance {  
    public static void main(String[] args) {  
        double x1, y1, x2, y2, dist;  
        x1 = 1.0;  
        y1 = 1.0;  
        x2 = 4.0;  
        y2 = 6.0;  
        dist = Math.sqrt(Math.pow(x2-x1,2)+ Math.pow(y2-y1,2));  
        System.out.println("The distance is " + dist);  
    }  
}
```

## Example2

- Write a program that finds the area of a triangle given the length of its sides: a, b, c. Use a = 3, b = 4, c = 5 to test your solution.

$$area = \sqrt{s \cdot (s - a) \cdot (s - b) \cdot (s - c)}$$
$$s = \frac{a + b + c}{2}$$

- Algorithm:
  - » Get the value of **a** (e.g., a = 3)
  - » Get the value of **b** (e.g., b = 4)
  - » Get the value of **c** (e.g., c = 5)
  - » Calculate **s** using the formula  $s = (a + b + c) / 2$
  - » Calculate **area** using the formula  $area = \text{sqrt}(s * (s - a) * (s - b) * (s - c))$
  - » print **area**

## Example2 (cont'd)

### ➤ Java code:

```
public class Distance {  
    public static void main(String[] args) {  
        double a, b, c, area;  
        a = 3.0;  
        b = 4.0;  
        c = 5.0;  
        s = (a + b + c) / 2.0;  
        area = Math.sqrt(s * (s - a) * (s - b) * (s - c));  
        System.out.println("The area is " + area);  
    }  
}
```



# Basic Object-Oriented Concepts

- Object-Oriented Paradigm
- What is an Object?
- What is a Class?
- Constructing Objects from a class
- Problem Solving in OO languages
- More OO Concepts
- Strength of Object-Oriented Paradigm

# Object-Oriented Paradigm

- To truly understand Java, we need to understand the paradigm on which it is built on: the Object-Oriented Paradigm (OOP).
- OOP is a model whereby a problem domain is modeled into objects, so that problem solving is by interaction among objects.
- OOP is a more natural model compared to other models since OOP's approach is exactly the way humans view problem solving.
- We take a closer look at the main ingredients of the model: **Objects** and **Classes**.

# What is an Object?

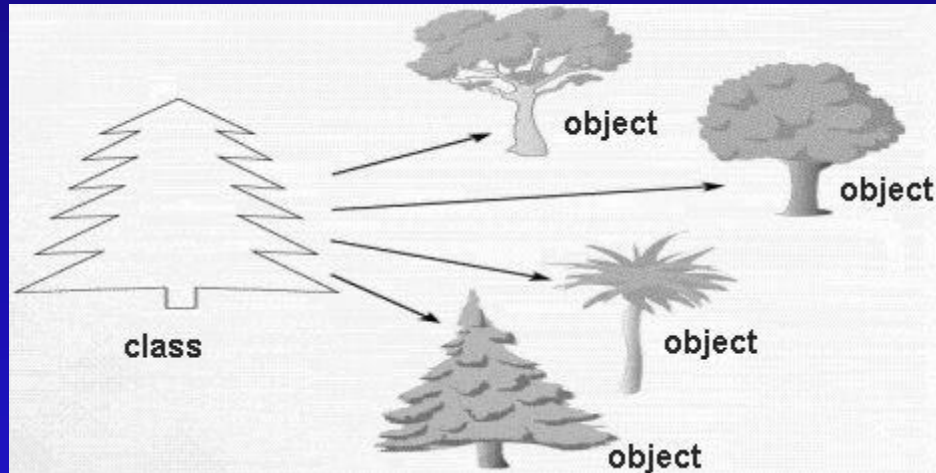
- An object is an individual, identifiable entity, either real or abstract, that has a well-defined boundary.
- An object has two main properties, namely:
  - **State**: each object has *attributes*, whose values represent its state.
  - **Behavior**, each object has a set of behavior.
- Example of an object: You, the student following this Lecture, are an object.
  - Your attributes include your name, your GPA, your major, etc.
  - You also have a set of behavior, including attending lectures, solving home works, telling someone your GPA, sleeping, etc.

# ... What is an Object?

- Other examples of objects are:
  - The instructor delivering this lecture.
  - This room.
  - This lecture.
  - This University.
  - CSC211.
  - Your Bank Account
  
- Try to list some of the attributes and set of behavior for each of the above objects.
  
- Also look around you and identify the objects you can see or imagine.

# What is a Class?

- A class is a general specification of attributes and behavior for a set of objects.
- Each object is an instance of a class. It shares the behavior of the class, but has specific values for the attributes.
- Thus, a class can be viewed as an abstract specification, while an object is a concrete instance of that specification.



- An example of a class is Student, an abstract entity that has attributes and a set of behavior.
- However, unless we have an actual student, we cannot say what the ID number is or what the major is.

## What is a Class? (cont'd)

- The following table shows further examples of classes and their instances:

Classes	Instances of (or Objects)
Instructor	Sahalu Junaidu
University	ABU
Course	CSC211
Bank Account	Ahmad's Bank Account

- Identify the classes for the objects you identified in the exercise on Slide#68.

# Constructing Objects from a Class

- Our main task is to design and implement classes to be used in creating objects.
- This involves defining the variables, methods and constructors.
- To create an object from a class, we use the **new** operator, a constructor, and supply construction parameters (if any).
- Example, we create an object from the Student class as follows:

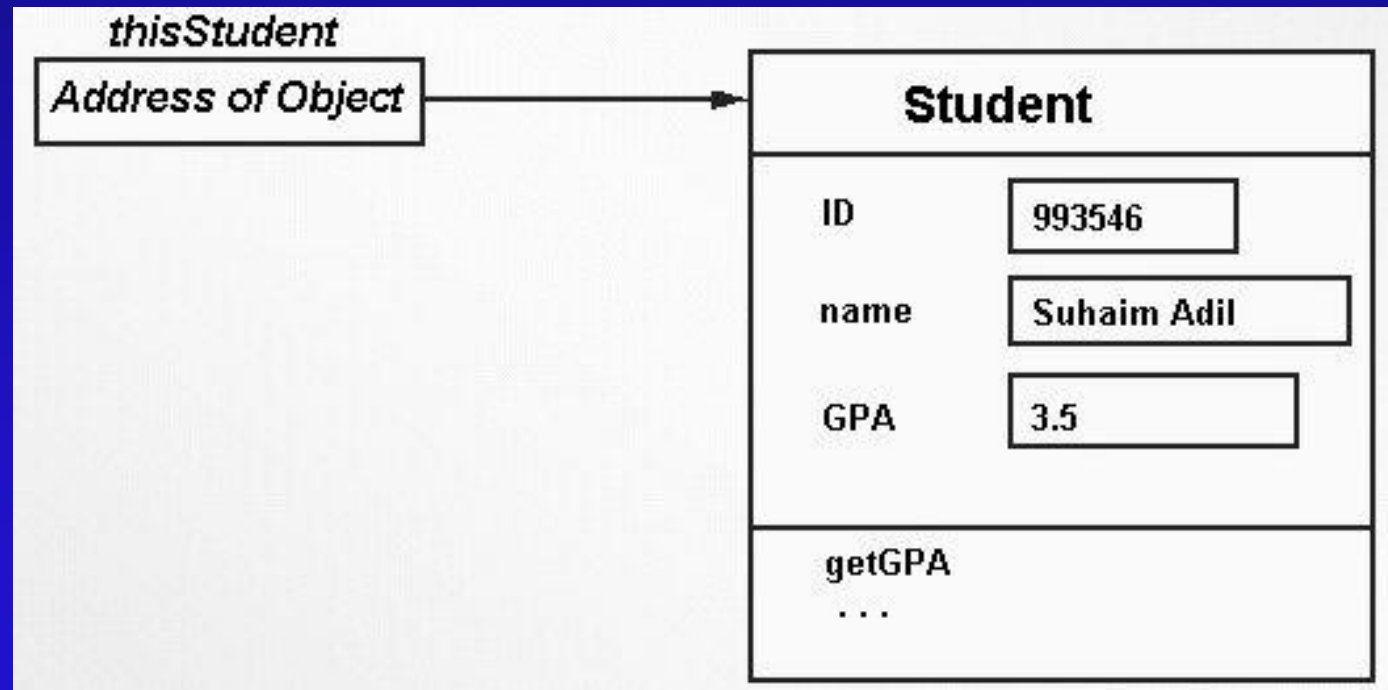
```
Student thisStudent =  
    new Student(993546, "Suhaim Adil", 3.5);
```

## ... Constructing Objects from a Class

```
Student thisStudent =
```

```
    new Student(993546, "Suhaim Adil", 3.5);
```

- The relationship between the reference variable, *thisStudent*, and the object created is shown by the following figure:





# Problem Solving in OO languages

- In the real world, problems are solved by interaction among objects.
- If you have a problem with your car, you take it to a mechanic for repair.
- Similarly, in OO programming languages, problems are solved by interactions among objects.
- Objects interact by sending messages among themselves.



# Problem Solving in OO languages

```
Student thisStudent =
```

```
    new Student(993546, "Suhaim Adil", 3.5);
```

- Messages are sent using the following general syntax:

```
referenceVariable.methodName
```

- To know the GPA of the student object we created, a message is sent as follows:

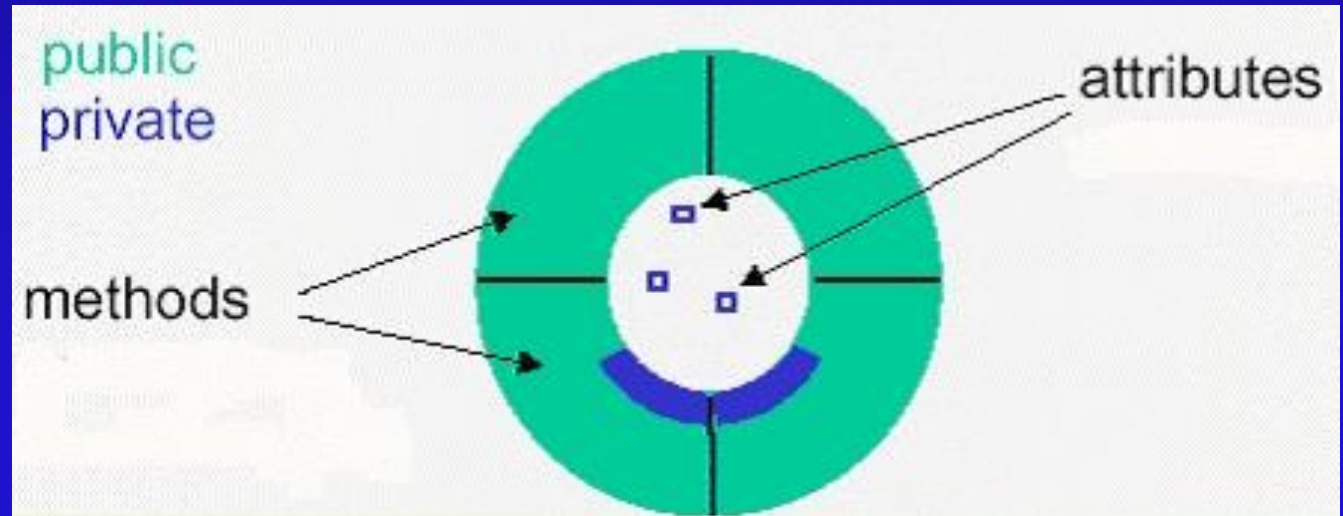
```
double gpa = thisStudent.getGPA();
```

- We have been sending messages to the `System.out` object in previous examples.

```
System.out.println("Salaaam Shabaab");
```

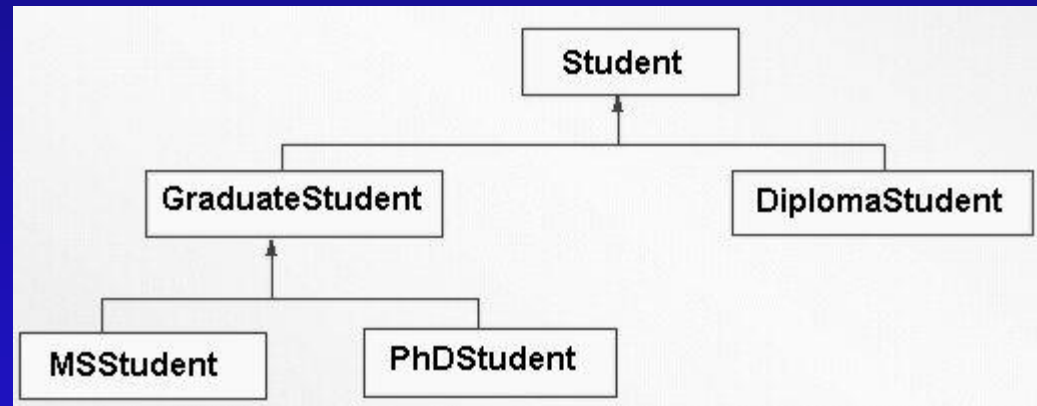
# Encapsulation

- Having understood the main ingredients of OOP, we can now take a closer look at its main features.
- One of the key features is the bringing together of data and operations – **Encapsulation**.
- Encapsulation, also called **information hiding**, allows an object to have full control of its attributes and methods.
- The following shows how an object encapsulates its attributes and methods.



# Inheritance

- Another powerful feature of OOP is inheritance.
- Classes are organized in hierarchical structure.
- For example, the following shows a **Student** class and its related sub-classes:



- The advantage is **code-reusability**.
- After implementing the **Student** class, to implement the **GraduateStudent**, we inherit the code of **Student**.

# Strength of the OO Paradigm

- We conclude this introduction to OOP by summarizing its main advantages:
  - It allows the production of software which is easier to understand and maintain.
  - It provides a clear mapping between objects of the problem domain (real world objects) and objects of the model.
  - It supports code reuse and reduces redundancy.
  - It allows "off the shelf" code libraries to be reused.
  - It allows the production of reliable (secure) software.

# Strings and String Operations

- What is a String?
- Internal Representation of Strings
- Getting Substrings from a String
- Concatenating Strings
- Comparing Strings
- Finding the index of a character or Substring
- Case Conversion and Trimming of Strings
- Strings are Immutable!
- Program Example
- More String Methods?

# What is a String?

- A String is a sequence of characters enclosed in double quotes. E.g. “Salaam Shabaab”
- “A” is a string but ‘A’ is a character
- String processing is a very frequent application.
- Thus, Java provides special support for strings.
- A string is an instance of Java’s built in String class. Thus, strings are objects.
- Like any object, a string object can be created using the *new* operator as in:  
`String greeting = new String(“Salaam Shabaab”);`
- Java allows a String object to be created without the use of *new*, as in:  
`String greeting = “Salaam Shabaab”;`

# Internal Representation of Strings

- Internally, String objects are represented as a sequence of characters indexed from 0.
- For example, the following string is represented as shown below:

**String greeting = "Salaam Shabaab";**

<i>greeting</i>		<i>the string object</i>													
address of object	➔	S	a	l	a	a	m		S	h	a	b	a	a	b
		0	1	2	3	4	5	6	7	8	9	10	11	12	13

- Many string methods return results based on this indexing:

```
char charAt(int index)
```

Returns the character at position index from this string.

- For example, the statement: **char letter = greeting.charAt(5);** stores the character **'m'** in the character variable **letter**.



# Internal Representation of Strings (cont'd)

- We can also ask a string object its length by calling its *length()* method:

<code>int length()</code>	Returns the length of this string.
---------------------------	------------------------------------

- For example, the statement:

```
int charCount = greeting.length();
```

stores **14** in the integer variable `charCount`.

# Getting Substring from a String

- A common operation on Strings is extracting a substring from a given string.

<code>String <b>substring</b>(int start)</code>	Returns the substring from <i>start</i> to the end of the string.
<code>String <b>substring</b>(int start, int end)</code>	Returns the substring from <i>start</i> to <i>end</i> but not including the character at end.

- For example, the statement:

```
String sub2 = greeting.substring(7)
```

creates the substring "Shabaab" that is referred to by `sub2`.

- For example, the statement:

```
String sub1 = greeting.substring(0, 6);
```

creates the substring "Salaam" that is referred to by `sub1`.

- What is the effect of the following statement?

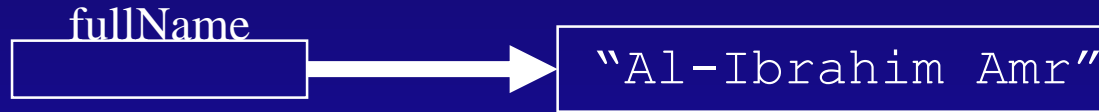
```
String sub3 = greeting.substring(8, 12);
```

# Concatenating Strings

- Concatenation means joining two or more strings together.
- Java allows two strings to be concatenated using the '+' operator.

Example:

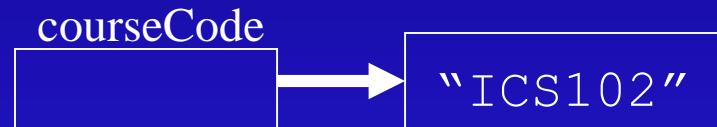
```
String firstName = "Amr";  
String lastName = "Al-Ibrahim";  
String fullName = lastName+" "+firstName;
```



- If one of the operands in an expression is a string, Java automatically converts the other to a string and concatenates them.

Example:

```
String course = "ICS";  
int code = 102;  
String courseCode = course+code;
```



- We frequently use the concatenation operator in *println* statements:

```
System.out.println("The area =" + area);
```

- You need to be careful with concatenation operator. For example, what is the output of the following statement?:

```
System.out.println("Sum =" + 5 + 6);
```

# Comparing Strings

- Strings are compared by comparing their characters left to right. Unicode codes are used in the comparison.
- Note that lowercase letters are different from uppercase letters.
- The String class has the following methods for checking whether two strings are equal:

<code>boolean equals (String another)</code>	Returns true if another is the same as this string.
<code>boolean equalsIgnoreCase (String another)</code>	Returns true if <i>another</i> is the same as this string, treating lower and upper case letters as the same.

- The following table shows some examples of applying these methods. Assuming the following declarations:

```
String s1 = "Salaam";  
String s2 = "Shabaab";  
String s3 = "SALAAM";
```

<code>s1.equals(s2)</code>	false
<code>s1.equals("Salaam")</code>	true
<code>s1.equals(s3)</code>	false

- What is the result of `s1.equalsIgnoreCase(s3)` ?

# Comparing Strings (cont'd)

- Sometimes we need to know if a string is less than another.
- Accordingly, the String class has the following additional comparison methods:

<code>int compareTo(String another)</code>	Returns a negative number if this string is less than <i>another</i> , 0 if they are equal and a positive number if this string is greater than <i>another</i> .
<code>int compareToIgnoreCase(String another)</code>	Same as above but treating lower and upper case letters as the same. .

- Assuming the following declarations:

```
String s1 = "Salaam";  
String s2 = "Shabaab";  
String s3 = "SALAAM";
```

we have:

<code>s1.compareTo(s2)</code>	a negative number
<code>s2.compareTo(s1)</code>	a positive number

- What is the result of `s1.compareToIgnoreCase(s3)`?

# Finding the index of a Character or Substring

- The following methods return an index given a character or substring:

<code>int <b>indexOf</b>(int code)</code>	Returns the index of the first occurrence of a character whose Unicode is equal to code.
<code>int <b>indexOf</b>(String substring)</code>	Same as above but locates a substring instead.
<code>int <b>lastIndexOf</b>(int code)</code>	Returns the index of the last occurrence of a character whose Unicode is equal to code.
<code>int <b>lastIndexOf</b>(String substring)</code>	Same as above but locates a substring instead.

- The table below shows some examples, assuming the declaration:

```
String greeting = "Salaam Shabaab";
```

<code>int index = greeting.indexOf('a')</code>	1
<code>int index = greeting.lastIndexOf('a')</code>	12
<code>int index = greeting.indexOf(98)</code>	10
<code>int index = greeting.indexOf("haba")</code>	8

# Case Conversion and Trimming of Strings

- It can be useful to convert a string to upper or lower case.

<code>String toLowerCase()</code>	Returns the lower case equivalent of this string.
<code>String toUpperCase()</code>	Returns the lower case equivalent of this string.

- For example, the statements:

```
String greeting = "Salaam Shabaab";  
String greeting2 = greeting.toUpperCase();
```

create two string objects. The object referenced by `greeting2` stores "SALAAM SHABAAB"

- Another useful method of String is `trim()`:

<code>String trim()</code>	Removes leading and trailing white spaces.
----------------------------	--

- For example, the statement:

```
String s = "    Salaam    ".trim();
```

stores "Salaam" in the string referenced by `s`.

- Note that return `'\r'`, tab `'\t'`, new line `'\n'` and space `' '` are all white space characters.

- All the methods of the String class can also be applied to anonymous string objects

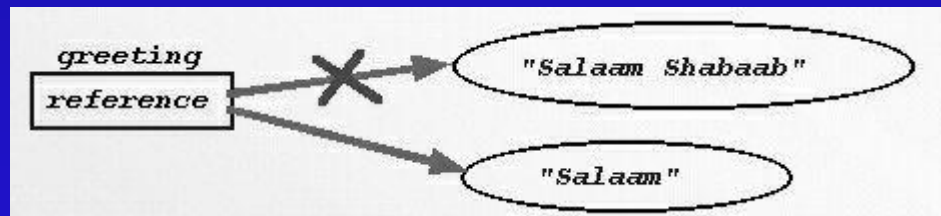
(also called string literals) as shown in the above example.

# Strings are Immutable!

- Another special feature of Strings is that they are immutable. That is, once a string object is created, its content cannot be changed.
- Thus, all methods that appear to be modifying string objects are actually creating and returning new string objects.
- For example, consider the following:

```
String greeting = "Salaam Shabaab";  
greeting = greeting.substring(0,6);
```

Instead of changing the *greeting* object, another object is created. The former is garbage collected.



- The fact that Strings are immutable makes string processing very efficient in Java.



# Program Example

- The following shows a program that uses some String methods.
- It breaks a full path for a file into drive letter, path, file name and extension and prints the result in upper case.

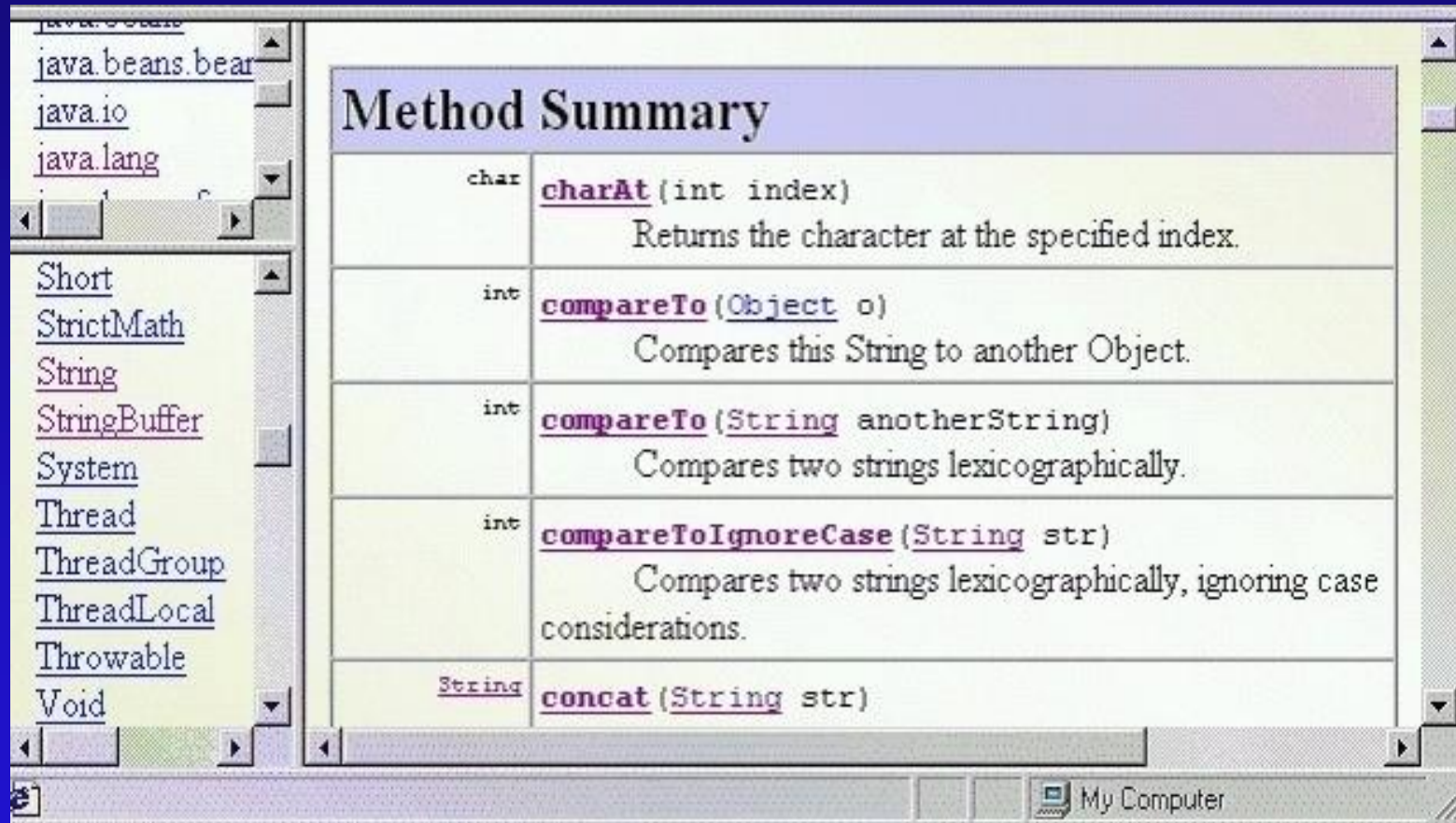
```
public class BreakPath {  
    public static void main(String[] args) {  
        String fullPath = "c:/ics102/lectures/Example1.java";  
        fullPath = fullPath.toUpperCase();  
        char driveLetter = fullPath.charAt(0);  
        int lastSlashIndex = fullPath.lastIndexOf('/');  
        String path = fullPath.substring(0, lastSlashIndex+1);  
        int dotIndex = fullPath.indexOf('.');  
        String file = fullPath.substring(lastSlashIndex+1, dotIndex);  
        String extension = fullPath.substring(dotIndex+1);  
        System.out.println("Drive letter = "+driveLetter);  
        System.out.println("Path = "+path);  
        System.out.println("File name = "+file);  
        System.out.println("File extension = "+extension);  
    }  
}
```

Output:

```
Drive letter = C  
Path = C:/ICS102/LECTURES/  
File name = EXAMPLE1  
File extension = JAVA
```

# More String Methods?

- We have discussed some of the most important methods of the String class.
- For a complete list, check the Java SDK documentation.



# Introduction to Console Input

- Primitive Type Wrapper Classes
- Converting Strings to Numbers
- `System.in` Stream
- Scanner
- Reading Strings
- Numeric Input

# Primitive Type Wrapper Classes

- Java uses primitive types, such as `int` and `char`, for performance reasons.
- However, there are times when a programmer needs to create an object representation for one of these primitive types.
- Java provides a Wrapper class for each primitive type. All these classes are in the *java.lang* package:

Primitive type	Wrapper class
char	Character
double	Double
float	Float
int	Integer
long	Long
byte	Byte
boolean	Boolean
void	Void

# String to Number Conversion

- Wrapper classes are used to provide constants and general methods for the primitive data types.
- Each of the Wrapper classes Double, Float, Integer, and Long has a method to convert the string representation of a number of the corresponding primitive type into its numeric format:

Wrapper class	parse method
Double	parseDouble(string)
Float	parseFloat(string)
Integer	parseInt(string)
Long	parseLong(string)

# String to Number Conversion

## ➤ Examples:

```
int numStudents = Integer.parseInt("500");  
String inputLine = "3.5";  
double studentGPA = Double.parseDouble(inputLine);  
val = Integer.parseInt(str);
```

# System.in stream

- In Java I/O is handled by streams.
- An input stream is an object that takes data from an input source and delivers that data to a program.
- An output stream is an object that takes data from a program and delivers it to an output destination. [*e.g., System.out that corresponds to the monitor*]
- In Java, console input is usually accomplished by reading from the input stream **System.in** of the class **java.lang.System**
- **System.in** represents the standard input stream (i.e., it corresponds to the keyboard).

# Scanner Class

- To be able to read characters, strings, or numbers, *System.in* must be wrapped in other objects.
- `java.util.Scanner` class is used to read strings and primitive types, and must be imported into a program to be used.
- A Scanner object can be set up to read input from various sources, including the user typing values on the keyboard
- The following line creates a Scanner object that reads data typed by the user from the keyboard:

```
Scanner input = new Scanner (System.in);
```

- The `new` operator creates the Scanner object



# Example: Reading String

- Once created, the Scanner object can be used to invoke various input methods, such as:

```
name = input.nextLine();
```

- The nextLine method reads all of the input until the end of the line is found

```
import java.util.Scanner;
public class ReadString{
    public static void main(String[] args){
        Scanner input = new Scanner(System.in);
        System.out.println("Enter a line of text:");
        String message = input.nextLine();
        System.out.println("You entered: " + message);
    }
}
```

# Example: Numeric Input

```
import java.util.Scanner;
public class ReadIntegers{
    public static void main(String[] args){
        Scanner input = new Scanner(System.in);

        System.out.println("Enter two integers on separate lines:");
        int num1 = input.nextInt();
        int num2 = input.nextInt();
        System.out.println("Sum = " + (num1 + num2));
    }
}
```

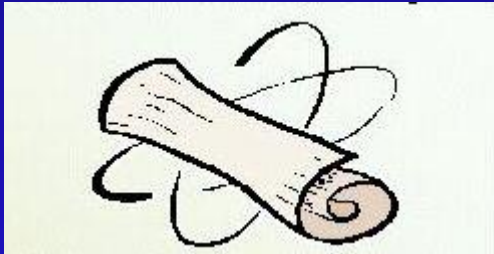
# Introduction to Classes

- Motivation
- Class Components
- Instance Variables
- Constructors
- The Student Class
- Exercises

# Motivation: The Problem

- Primitive Data types and Strings are not enough to write useful real-life programs.
- Example of some real life problems:

Student Information System



Bank Transaction System



Airline Reservation System



Hospital Management System



# Motivation: The Solution

- Solution : Interaction among Objects.



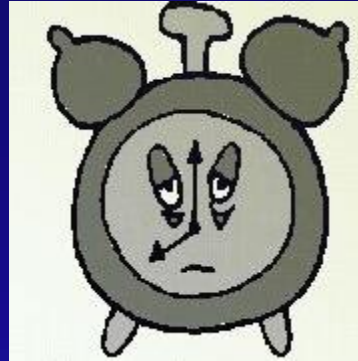
- Our main concern: how to design classes as templates for objects.



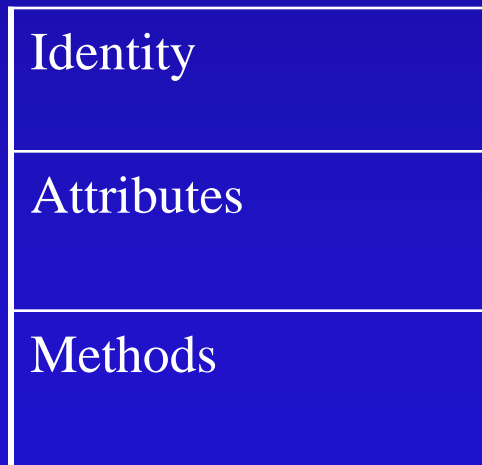
# Class Components

➤ A class has following elements:

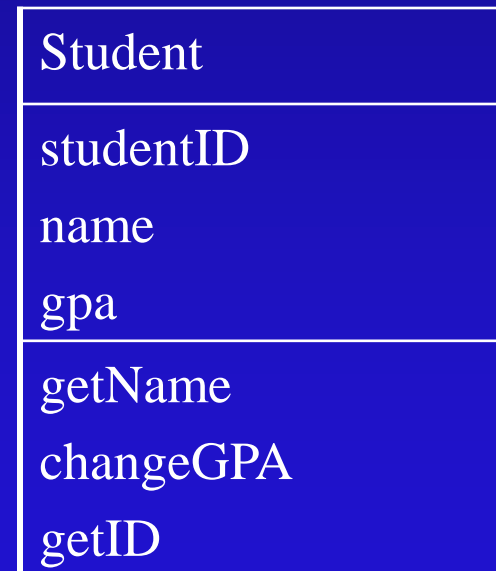
- Identity
- Attributes
- Methods



➤ Graphical Representation of a class (UML diagram):



Example:



# Class Identity

- Class naming convention:
  - Noun or Noun phrase
  - First letter of each word capitalized. No underscores.
- Examples: Tree, DatePalm, Student, GraduateStudent, BankAccount, InputStreamReader
- Syntax:

```
modifier class ClassName{
    attributes
    methods
}
```
- Example:

```
public class Student{
    . . .
}
```



classes are  
unique

# Class Attributes

- Attributes are the distinctive characteristic, quality, or feature that contribute to make objects of a class unique.
- Each attribute has a value.
- An attribute could be of simple type or it could be another class type.
- Some attribute values change over time.
- Objects store their attribute values in **instance variables**.
- It is good practice to make attributes private.





# Attributes: Syntax & Naming

- Attribute naming convention:
  - Noun, noun phrase or adjective
  - Starts with small letter and each
  - phrase's first letter capitalized

Good attribute names	Bad attribute names
studentName	readBook
color	Color
yearlySalary	yearlysalary



- Syntax:

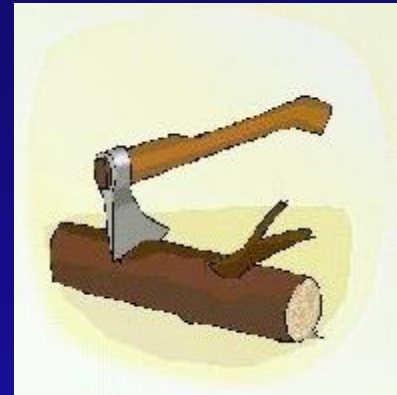
```
accessModifier type attributeName;  
accessModifier type attributeName = initialValue;
```

- Example:

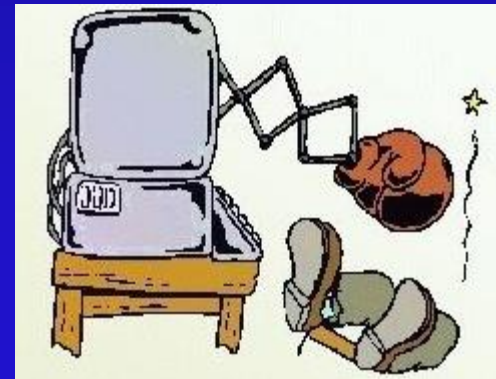
```
private int id;  
private double gpa;
```

# Methods

- Methods are the actions that an object performs. Through methods objects interact and pass messages to other objects.



- The behavior of an object depends on its attribute values and the operations performed upon it.



# Methods: Syntax & Naming

- Method naming convention:
  - Verb or verb phrase
  - Starts with small letter and each
  - phrase's first letter capitalized

Good method names	Bad method names
getName	GetName
changeGPA	studentInformation
registerCourse	playfootball

- Syntax:

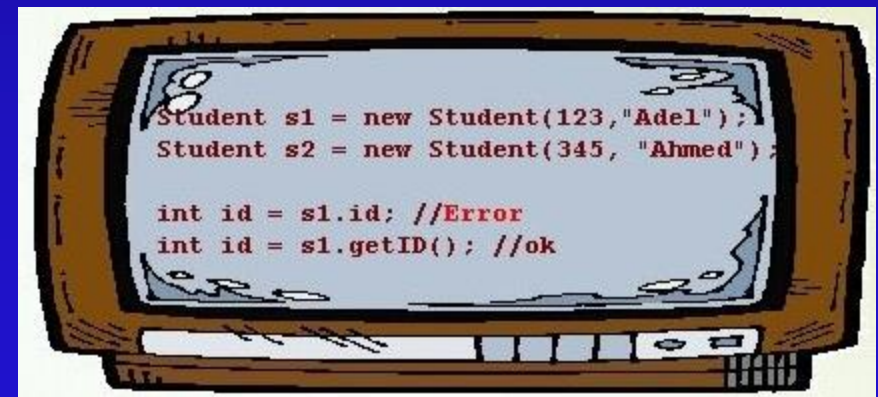
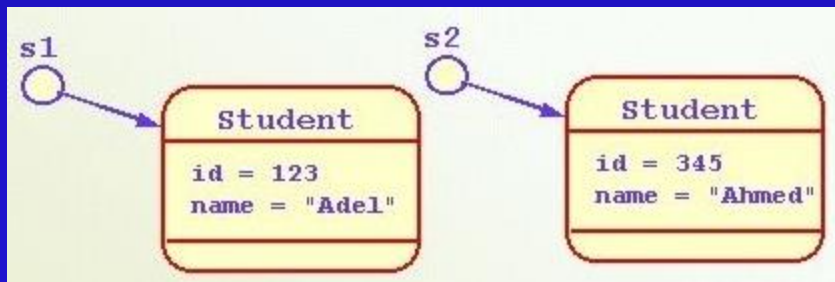
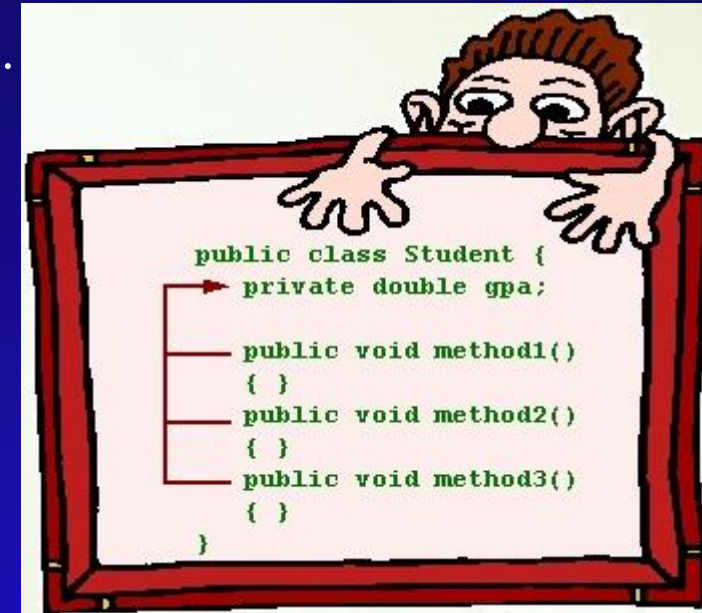
```
modifier returnType methodName(parameterType parameter,...){  
    statements  
    return expression;  
}
```

- Example:

```
public double calculateAverage(int value1, int value2){  
    double average = (value1 + value2) / 2.0;  
    return average;  
}
```

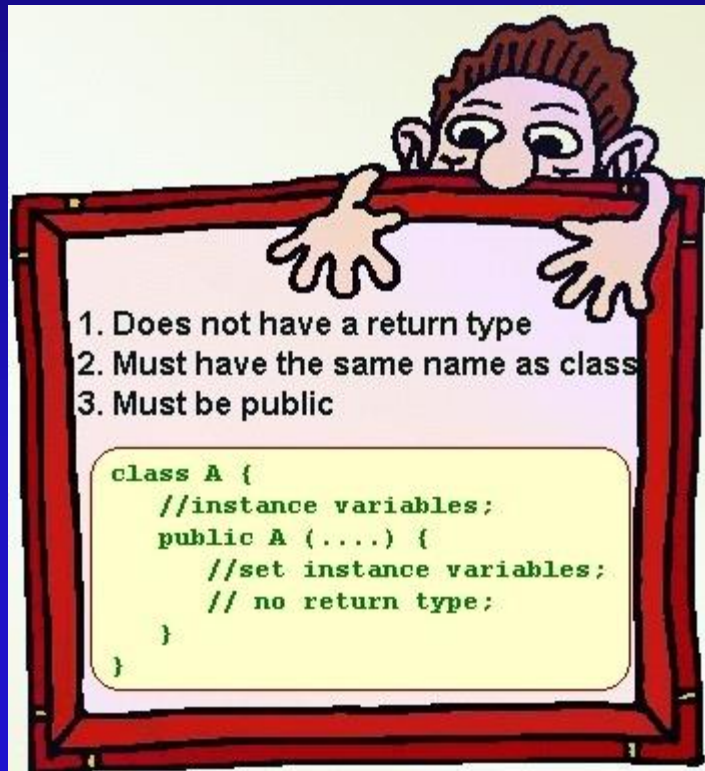
# Instance Variables

- **State** of an object: the set of values that describe the object.
- Each object stores its state in one or more **instance variables**.  
`private double gpa ;`
- **Scope** of an Instance variable: the part of the program in which you can access the variable.
- In Java, the scope of a variable is the block in which that variable is declared.
- Private instance variables can be accessed directly only in the methods of the same class.
- We can gain access to private instance variables through public methods in that class.
- Each object of a class has its own copy of an instance variable.



# Constructors

- In addition to instance variables and methods, a class may have one or more constructors.
- Constructors are used to initialize the instance variables of an object at the time of creating the object.
- What is the difference between a constructor and a method?
- A constructor:



4. Is invoked using the **new** operator.

# The Student Class

```
public class Student{
    private int id;
    private String name;
    private double gpa;
    public Student(int theID, String theName, double theGPA){
        id = theID;
        name = theName;
        gpa = theGPA;
    }
    public String getName(){
        return name;
    }
    public int getID(){
        return id;
    }
    public double getGPA(){
        return gpa;
    }
}
```

Must be stored in a file  
called: **Student.java**

# The TestStudent Class

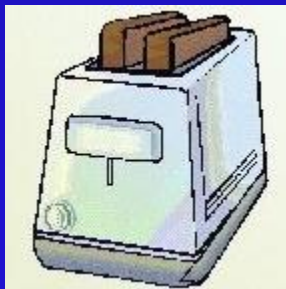
```
public class TestStudent{
    public static void main(String[] args){
        Student student =
            new Student(999999, "Ahmad Muhammad", 3.2);
        System.out.println("Name: " + student.getName());
        System.out.println("ID#: " + student.getID());
        System.out.println("GPA: " + student.getGPA());
    }
}
```

- Note: Two or more classes may be placed in a single Java file. In that case:
  - Only one class can be public; namely the class containing the main method.
  - The name of the Java file must be that of the public class.



# Exercises

- What is the difference between a constructor and a method?
- What are constructors used for? How are they defined?
- Think about representing an alarm clock as a software object. Then list some characteristics of this object in terms of states and behavior. Draw the class in UML diagram.
- Repeat the previous question for a Toaster Object.





# Introduction to Classes II

---

- Default Constructor
- **this** Keyword
- Car Example
- Exercises

# Default Constructor

➤ **Question:** What happens if a constructor is not defined in a class?

➤ **Answer:** A default constructor is created automatically.

➤ A default constructor:

- Has no parameter
- Initializes instance variables with default values
- Is not provided if a constructor is already defined

```
class Student{  
    private int id;  
    private String name;  
    private double gpa;  
  
    // no constructor defined  
  
    public String getName(){  
        return name;  
    }  
}
```

Instance variable of type:	default value
boolean	false
byte	(byte) 0
short	(short) 0
int	0
long	0L
float	0F
double	0D
char	\u0000
object reference	null

# Overloading Constructors

- Java allows more than one Constructor
- Problem: All constructors will have the same name!
- Solution: Overloading
- Each constructor will have different parameters



```
class Student{  
    private int id;  
    private String name;  
    private double gpa;  
    public Student(int theID, String theName, double theGPA){  
        id = theID;  
        name = theName;  
        gpa = theGPA;  
    }  
    public Student(int theID, String theName){  
        id = theID;  
        name = theName;  
        // gpa is initialized to 0.0  
    }  
    // . . .  
}
```

# this Keyword

- Reference to the current object
- Two usages of **this** keyword:

1. To refer to the fields and methods of **this** object  
Purpose: resolve conflict in naming.
2. From a constructor of a class call another constructor of the same class.

```
class Student{  
    private int id;  
    private String name;  
    private double gpa;  
    public Student(int id, String name, double gpa){  
        this.id = id;  
        this.name = name;  
        this.gpa = gpa;  
    }  
    public Student(int id, String name){  
        this(id, name, 0.0);  
    }  
    // ...  
}
```

# Example: Car Class

- Suppose we want to design a class that calculates the fuel consumption of a car in kilometers per liter.
- This calculation is possible if we know three things:
  1. The initial reading of the odometer
  2. The final reading of the odometer
  3. The number of liters used.



Knowing these information items,

the consumption (in km per liters) =

$$(\text{finalOdometerReading} - \text{initialOdometerReading}) / \text{litersUsed}$$

# Car class (cont'd)

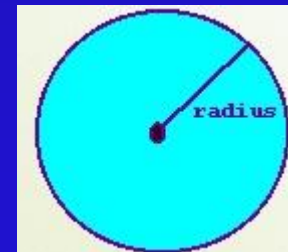
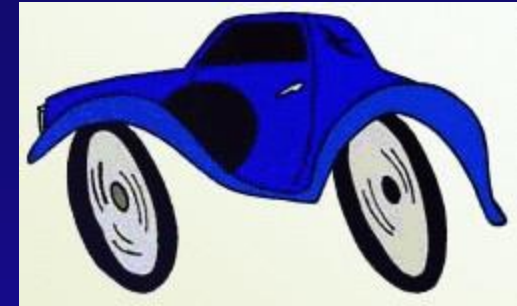
```
class Car{
    int initialOdometerReading, finalOdometerReading;
    double litersUsed;
    public Car(int initReading, int finalReading, double liters){
        initialOdometerReading = initReading;
        finalOdometerReading = finalReading;
        litersUsed = liters;
    }
    public Car(int finalReading, double liters){ //used to create new car objects
        finalOdometerReading = finalReading;
        litersUsed = liters;
    }
    public double getKilometersPerLiter(){
        return (finalOdometerReading – initialOdometerReading) / litersUsed; // assumes litersUsed is not
zero
    }
}
```

# Car class: Test program

```
public class TestCar{  
    public static void main( String[ ] args){  
        Car car1 = new Car( 32456, 32776, 40.0);  
        System.out.println("Fuel consumption for car1 is " + car1. getKilometersPerLiter( ) + " km/liter");  
        Car car2 = new Car( 365, 30.0);  
        System.out.println("Fuel consumption for car2 is " + car2. getKilometersPerLiter( ) + " km/liter");  
    }  
}
```

# Exercises

1. Add a boolean method called `isEconomyCar` to the Car class. This method will return true if the fuel consumption is less than 5 kilometers per liter.
2. Add a `boolean` method called `isFuelGuttler` to the Car class.
3. This method will return true if the fuel consumption is more than 15 KM/Liter
4. Implement a class **Product**. A product has a name and a price. Supply methods `printProduct()`, `getPrice()`, and `setPrice()`.
5. Write a test program that makes two products, prints them, reduces their prices by 5 Naira, and then prints them again.
6. Implement a class **Circle** that has methods `getArea()` and `getCircumference()`. In the constructor, supply the radius of the circle.





# Selection

# Outline

- Relational Operators
- Boolean Operators
- Truth Tables
- Precedence Table
- Selection and Algorithms
- The **if -else** statement
- Variations of **if-else** statement
- The **switch** statement

# Relational Operators

- In addition to arithmetic operators which produce numeric results, relational operators are used in comparisons.

Operator	Meaning
==	equal
!=	not equal
>	greater than
>=	greater than or equal
<	less than
<=	less than or equal

- The result of a comparison is Boolean (i.e., true or false)

➤ Example: `x == Math.sqrt(y);`

# Boolean Operators

- Connecting relational expressions requires another set of operators called boolean operators:

Operator	Meaning
&&	logical and
	logical or
!	negation

- Examples:

- `grade >= 0 && grade <= 100.0`

- `ch == 'q' || ch == 'Q'`

# Truth Tables

- Let **p** and **q** be Boolean or relational expressions

- Truth table for negation:

p	! p
true	false
false	true

- Truth table for logical and:

p	q	p && q
true	true	true
true	false	false
false	true	false
false	false	false

- Truth table for logical or:

p	q	p    q
true	true	true
true	false	true
false	true	true
false	false	false

# Precedence Table

Operator Type	Code
postfix	<i>expr</i> ++ <i>expr</i> --
unary	++ <i>expr</i> -- <i>expr</i> + <i>expr</i> - <i>expr</i> !
creation or cast	new      ( <i>type</i> ) <i>expr</i>
Multiplicative	*   /   %
Additive	+   -
relational	<   >   <=   >=
equality	==   !=
Logical AND	&&
Logical OR	
assignment	=   +=   -=   *=   /=   %=

Highest precedence



Lowest precedence

# Statements

- Statements can be simple or compound.
- A number of statements can be grouped together to form one compound statement by enclosing these statements in curly brackets { }

# Selection and Algorithms

- We have seen that flow control is divided into three types:
  - » Sequential
  - » Selection
  - » Iteration
- In Selection, we select one alternative based on a criterion (condition)



# Selection and Algorithms (cont'd)

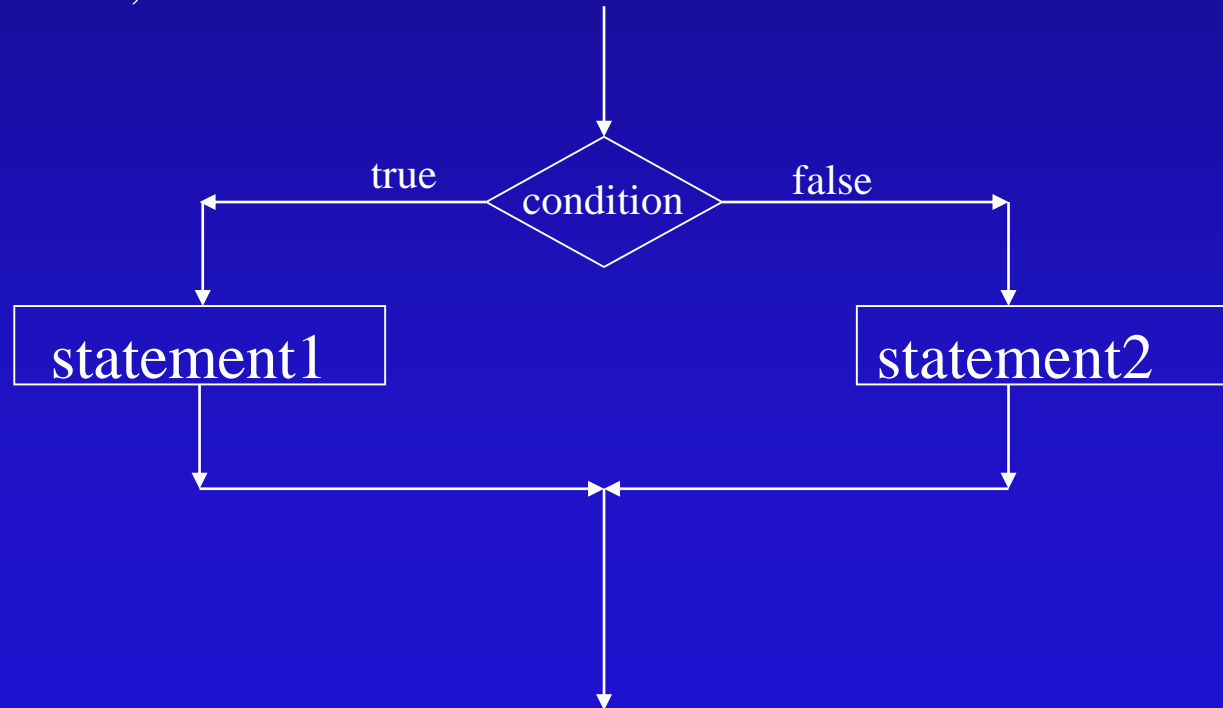
- Assume that we want to find the status of a student given his GPA.
- Algorithm:
  - » Get GPA
  - » if GPA  $\geq 2$ 
    - status = “good standing”
  - » else
    - status=“under probation”

# The if-else statement

- The general structure is:

```
if(condition)
    statement1
else
    statement2
```

- The **condition** is evaluated first; **statement1** is executed if result is true otherwise **statement2** is executed.



# Example

```
public String getStatus( )
{
    String status;
    if(gpa >= 2)
        status = "good standing";
    else
        status = "under probation";
    return status;
}
```

# Variation of if-else statement: if-statement

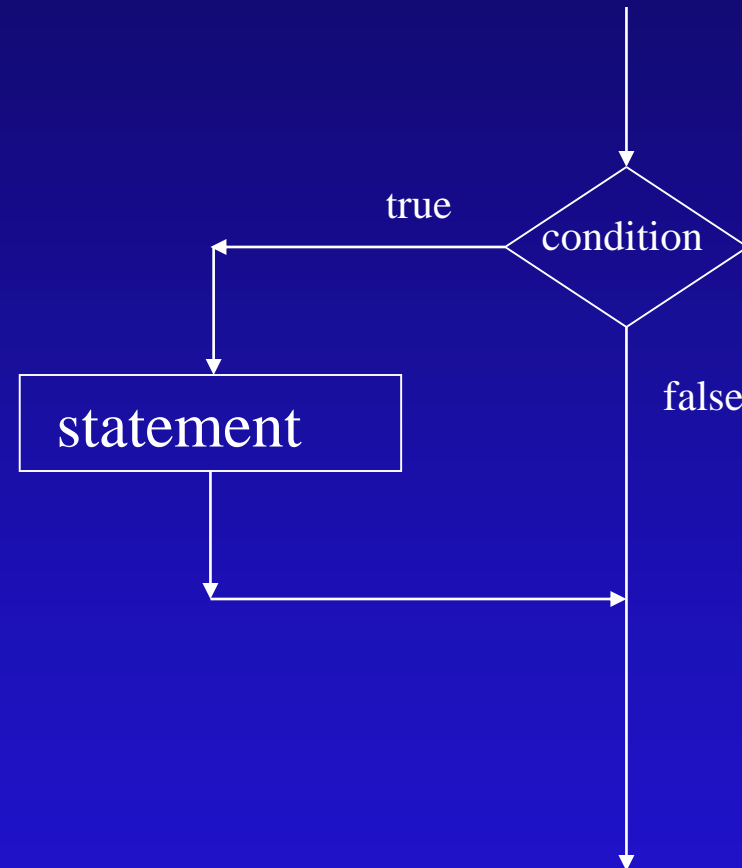
- The **else** part in the **if** may be omitted if nothing needs to be done when the condition is false.
- In such cases, the general structure is:

```
if(condition)
    statement
```

- Example: Finding the maximum of three numbers:

```
max = num1;
if(num2 > max)
    max = num2;

if(num3 > max)
    max = num3;
```

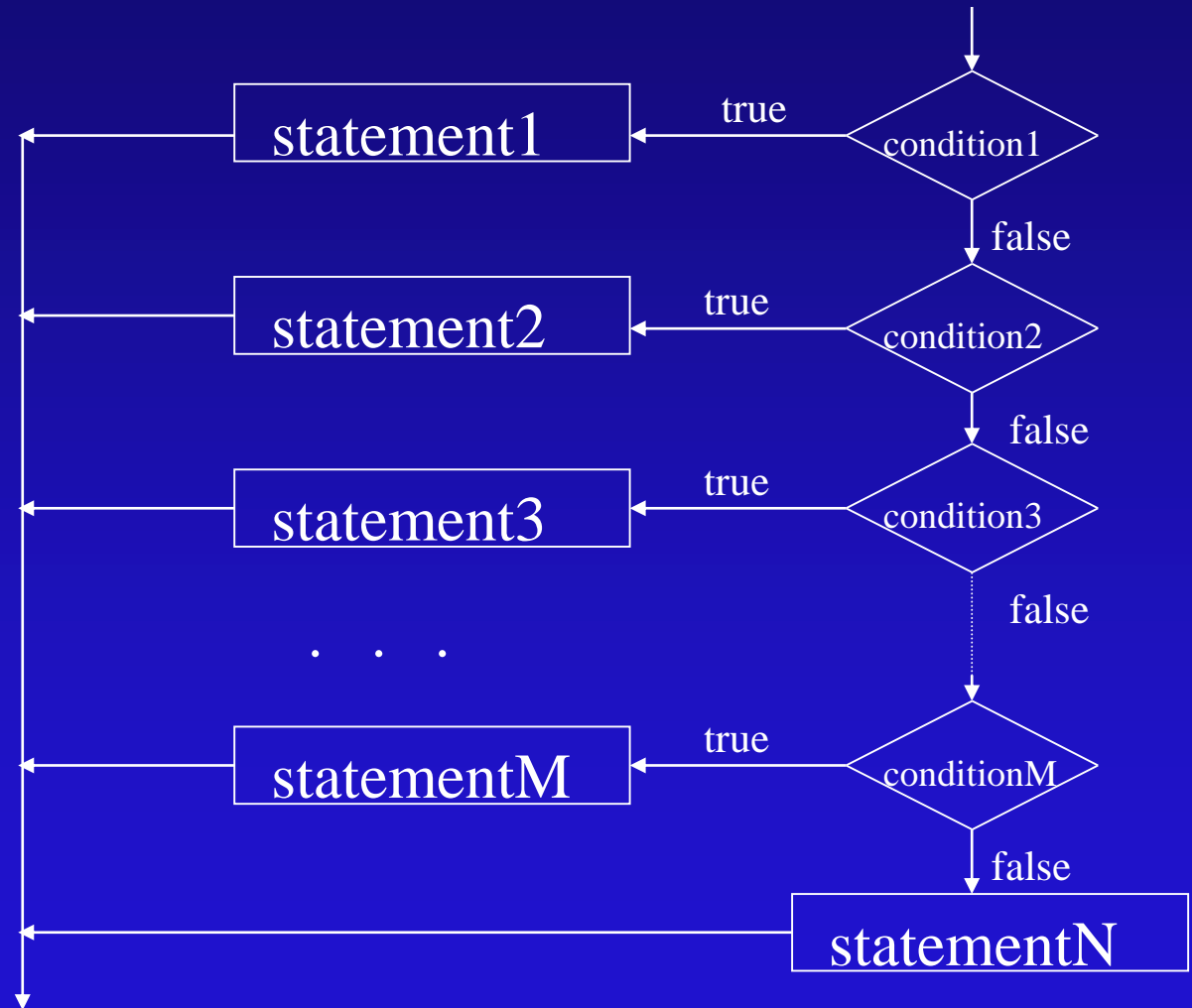


# Variation of if-else Statement: Nested if statement

- The **else** part of an **if-else** statement may contain another **if**-statement to provide multiple selection
- The general structure is:

```
if(condition1)
    statement1
else if(condition2)
    statement2
...
else
    statementN
```

- Note: The **else** part may be missing.



# Example: Convert a Day Code to a Day Name

```
import java.io.*;
class DayOfWeek1 {
    public static void main(String[] args) throws IOException {
        BufferedReader stdin = new BufferedReader( new InputStreamReader(System.in));
        System.out.print("Enter a number [1 .. 7]: ");
        String input = stdin.readLine();
        int dayNumber = Integer.parseInt(input);
        if(dayNumber == 1)
            System.out.println("Saturday");
        else if(dayNumber == 2)
            System.out.println("Sunday");
        else if(dayNumber == 3)
            System.out.println("Monday");
        else if(dayNumber == 4)
            System.out.println("Tuesday");
        else if(dayNumber == 5)
            System.out.println("Wednesday");
        else if(dayNumber == 6)
            System.out.println("Thursday");
        else if(dayNumber == 7)
            System.out.println("Friday");
        else
            System.out.println("Wrong input");
    }
}
```

# The switch statement

- When selecting among many alternatives, the **switch** statement may be used.

- The general structure is:

```
switch(controllingExpression)
{
    case value1 :    Statements;
                    break;
    case value2 :    Statements;
                    break;
    ...
    default :       Statements;
}
```

- The **controllingExpression** must have a **byte**, **char**, **short**, or **int** value.
- A case label value must be a unique constant value or a constant expression of type byte, char, short, or int.
- The statements of the case label that equals the value of the **controllingExpression** are executed; otherwise if there is no matching case label, the statements of the **default** label, if present, are executed.

# Example

```
import java.io.*;
class DayOfWeek2 {
    public static void main(String[] args) throws IOException {
        BufferedReader stdin = new BufferedReader(new
            InputStreamReader(System.in));
        System.out.print("Enter a number [1 .. 7]: ");
        String input=stdin.readLine();
        int dayNumber=Integer.parseInt(input);
        switch(dayNumber){
            case 1:  System.out.println("Saturday"); break;
            case 2:  System.out.println("Sunday");      break;
            case 3:  System.out.println("Monday");      break;
            case 4:  System.out.println("Tuesday");     break;
            case 5:  System.out.println("Wednesday");  break;
            case 6:  System.out.println("Thursday");   break;
            case 7:  System.out.println("Friday");     break;
            default: System.out.println("Wrong input");
        }
    }
}
```



# Example: Classifying a character

```
➤ // . . .
System.out.println("Enter a character: ");
char ch = (stdin.readLine()).charAt(0);
if(! Character.isLetter(ch))
    System.out.println(ch + " IS NOT A LETTER");
else{
    switch(ch){
        case 'a': case 'A':
        case 'e': case 'E':
        case 'i': case 'I':
        case 'o': case 'O':
        case 'u': case 'U': System.out.println(ch + " IS A VOWEL");
                           break;
        default: System.out.println(ch + " IS A CONSONANT");
    }
}
// . . .
```

- Note: **Character** is a wrapper class for the primitive type **char**. It contains several methods to process characters. It is defined in **java.lang** package

# Outline

---

- Iteration and Algorithms
- The while loop
- The for loop
- The do while loop

# Iteration and Algorithms

- We have seen that flow control has three types:
  - Sequential
  - Selection
  - Iteration
- In the third control, a statement/statements will be executed repeatedly, a number of times, based on a criterion (condition)

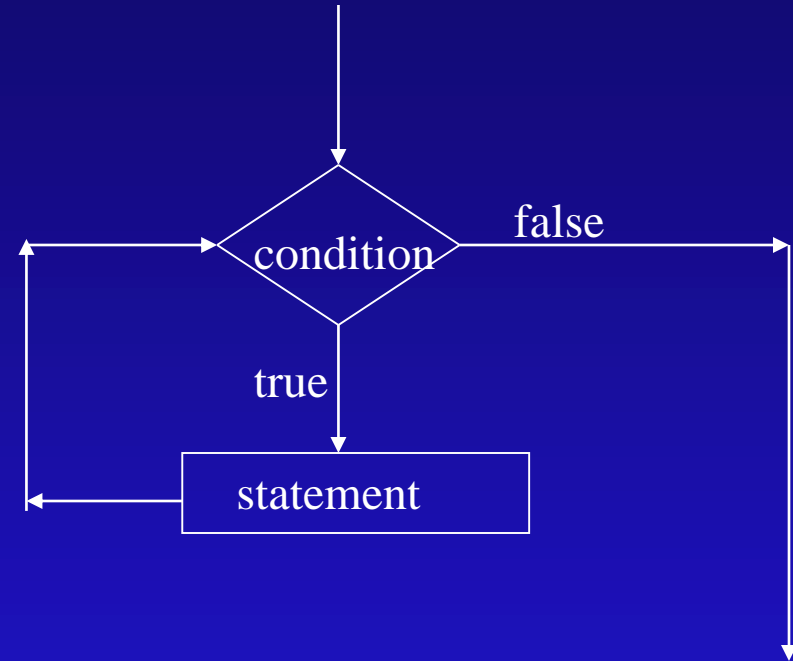
# Iteration and Algorithms

- Assume that we want to find the sum of numbers from 1 to n
- Algorithm
  - » `sum = 0`
  - » `index = 1`
  - » `while index <= n`
    - `sum = sum + index`
    - `index = index + 1`

# The while loop

- This loop is a precondition loop or 0-trip loop.
- The general structure is

```
while (condition)
    statement
```
- The **condition** is evaluated first.
- **Statement** is executed if condition is true.
- The **condition** is evaluated again and again until it evaluates to false.



# Example

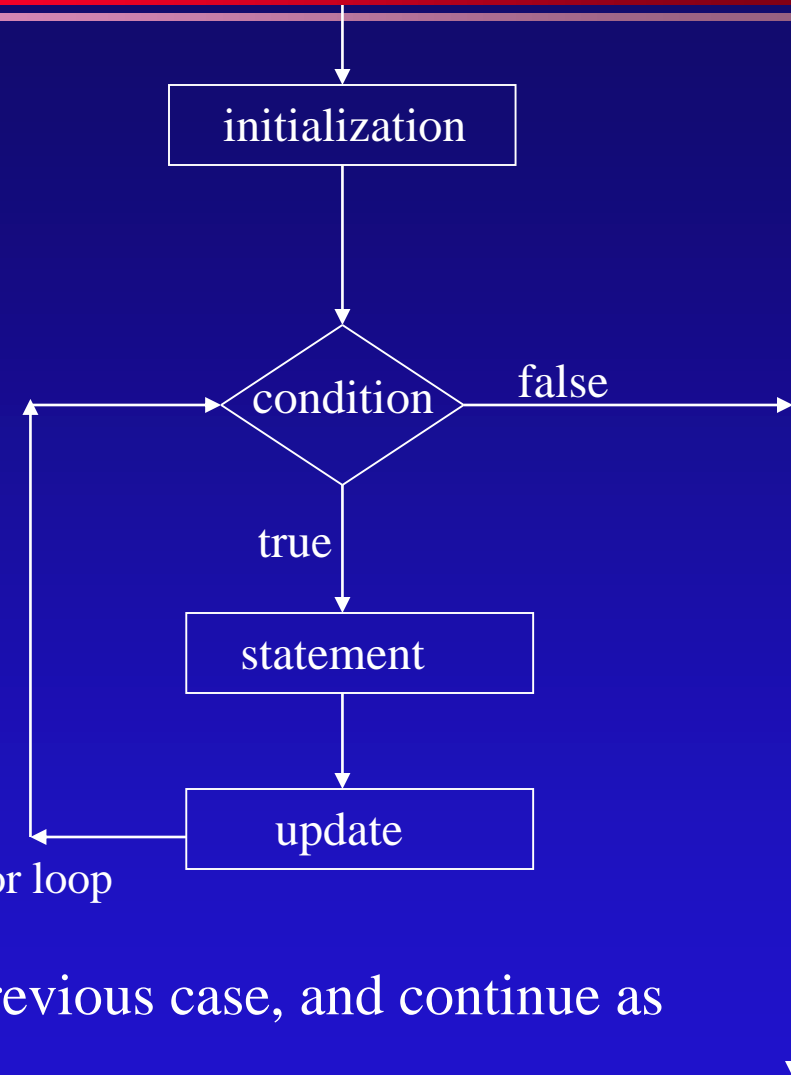
```
public static int sum1toN(int n){  
    int sum = 0, index = 1;  
    while ( index <= n )  
    {  
        sum += index;  
        index++;  
    }  
    return sum;  
}
```

- Notes: when using the while loop notice the following:
  - use braces if more than one statement to be repeated
  - initialize the variable used in the condition before the loop
  - update the variable inside the loop

# The for loop

- This loop is precondition loop or 0-trip loop.
- The general structure is

```
for (initialization; condition; update)
    statement
```
- First the **initialization** is executed once.
- The **condition** is evaluated next.
  - if the **condition** is true then
    - execute **statement**
    - execute **update**
  - if **condition** is false then
    - continue execution at the statement following the for loop
- Evaluate **condition** again, if it was true in the previous case, and continue as in the step above



# Example

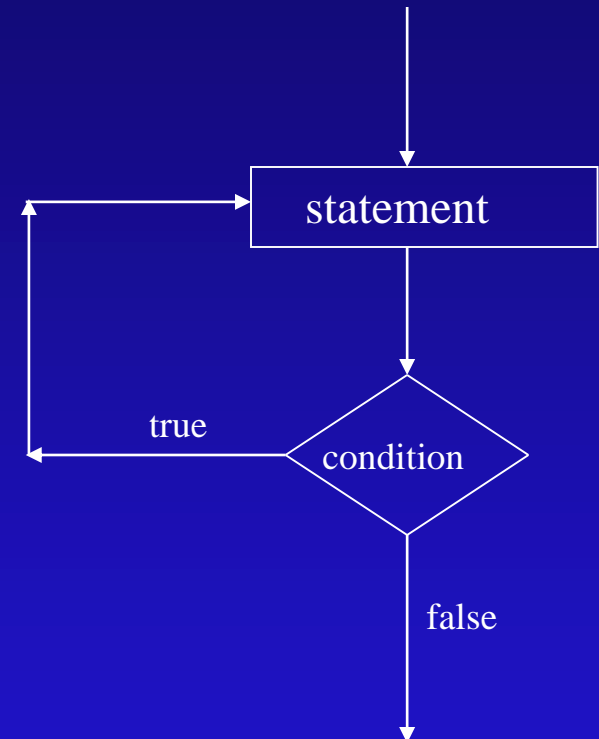
```
public static int sum1toN(int n){  
    int sum = 0, index = 1;  
    for ( index = 1; index <= n; index++)  
        sum += index;  
    return sum;  
}
```



# The do while loop

- This loop is post condition loop or 1-trip loop.
- The general structure is

```
do
    statement
while (condition);
```
- The **statement** is executed first.
- The **condition** is evaluated next.
  - If the **condition** is true, the **statement** is executed again and again until it evaluates to false.



# Example

```
public static int sum1toN(int n){  
    int sum = 0, index = 1;  
    do  
    {  
        sum += index;  
        index++;  
    }while ( index <= n );  
    return sum;  
}
```

# Nested Loops

---

- Loops can be nested and will be executed such that for every iteration of the outer loop, all the iterations of the inner loop.
- The inner loop must be completely inside the outer loop.
- Loop indexes must be different.

# Example

```
public static void multiplicationTable(int n) {  
    int row, col;  
    for (row = 1; row <= n; row++)  
    {  
        for (col = 1; col <= n; col++)  
            System.out.print(row*col + "\t");  
        System.out.println( );  
    }  
}
```

# String Tokenization

- What is String Tokenization?
- The StringTokenizer class
- Examples

# What is String Tokenization?

- So far we have been reading our input one value at a time.
- Sometimes it is more natural to read a group of input at a time.
- For example, when reading records of students from a text file, it is natural to read a whole record at a time.  

```
"995432 Al-Suhaim Adil 3.5"
```
- The `nextLine()` method of the `Scanner` class can read a group of input as a single `String` object.
- The problem is, how do we break this string object into individual words known as *tokens*?

```
"995432"  
"Al-Suhaim Adil"  
"3.5"
```

- This process is what String **tokenization** is about.

# The StringTokenizer Class

- The **StringTokenizer** class, of the **java.util** package, is used to break a **String** object into individual tokens.
- It has the following constructors:

Constructor	function
<code>StringTokenizer(String str)</code>	Creates a <code>StringTokenizer</code> object that uses white space characters as delimiters.
<code>StringTokenizer(String str, String delimiters)</code>	Creates a <code>StringTokenizer</code> object that uses the characters in <i>delimiters</i> as separators.
<code>StringTokenizer(String str, String delimiters, boolean returnTokens)</code>	Creates a <code>StringTokenizer</code> object that uses characters in <i>delimiters</i> as separators and treats separators as tokens.

# StringTokenizer Methods

- The following are the main methods of the **StringTokenizer** class:

Method	function
<code>String nextToken()</code> throws <code>NoSuchElementException</code>	Returns the next token as a string from this <code>StringTokenizer</code> object. Throws an exception if there are no more tokens.
<code>int countTokens()</code>	Returns the count of tokens in this <code>StringTokenizer</code> object that are not yet processed by <i>nextToken()</i> -- initially all.
<code>boolean hasMoreTokens()</code>	Returns true if there are more tokens not yet processed by <i>nextToken()</i> .



# How to apply the methods

- To break a string into tokens, first, a **StringTokenizer** object is created.

```
String myString = "I like Java very much";  
StringTokenizer tokenizer = new StringTokenizer(myString);
```

- Then any of the following loops can be used to process the tokens:

```
while(tokenizer.hasMoreTokens()) {  
    String token = tokenizer.nextToken();  
    // process token  
}
```

or

```
int tokenCount = tokenizer.countTokens();  
for(int k = 1; k <= tokenCount; k++){  
    String token = tokenizer.nextToken();  
    // process token  
}
```

# Example 1

- The following program reads grades from the keyboard and finds the average. The grades are read in one line.

```
import java.util.Scanner;
import java.util.StringTokenizer;
public class TokenizerExample1{
    public static void main(String[] args)throws IOException{
        Scanner input = new Scanner(System.in);
        System.out.print("Enter grades in one line:");
        String inputLine = input.nextLine();
        StringTokenizer tokenizer = new StringTokenizer(inputLine);
        int count = tokenizer.countTokens();
        double sum = 0;
        while(tokenizer.hasMoreTokens())
            sum += Double.parseDouble(tokenizer.nextToken());
        System.out.println("\nThe average = "+ sum / count);
    }
}
```

## Example 2

- This example shows how to use the second constructor of **StringTokenizer** class.
- It tokenizes the words in a string, such that the punctuation characters following the words are **not appended** to the resulting tokens.

```
import java.util.StringTokenizer;
public class TokenizerExample2{
    public static void main(String[] args){
        String inputLine =
            "Hi there, do you like Java? I do;very much.";
        StringTokenizer tokenizer =
            new StringTokenizer (inputLine, ",.?:;! \t\r\n");
        while(tokenizer.hasMoreTokens())
            System.out.println(tokenizer.nextToken());
    }
}
```

Output:

Hi  
there  
do  
you  
like  
Java  
I  
do  
very  
much

## Example 3

- This example shows how to use the third constructor of **StringTokenizer** class.
- It tokenizes an arithmetic expression based on the operators and returns both the operands and the operators as tokens.

```
import java.util.StringTokenizer;
public class TokenizerExample3{
    public static void main(String[] args){
        String inputLine = "(2+5)/(10-1)";
        StringTokenizer tokenizer = new
            StringTokenizer(inputLine,"+-*/()",true);
        while(tokenizer.hasMoreTokens())
            System.out.println(tokenizer.nextToken());
    }
}
```

Output:

(  
2  
+  
5  
)  
/  
10  
-  
1  
)

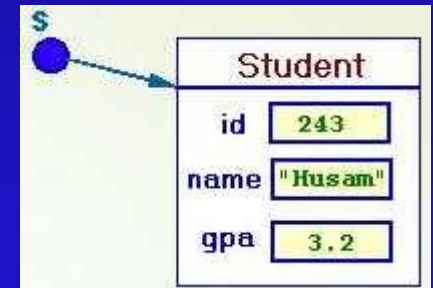
# Introduction to Methods

- Method Calls
- Parameters
- Return Type
- Method Overloading
- Accessor & Mutator Methods
- Student Class: Revisited

# Method Calls

- The set of methods defined for an object determines the behavior of that object.
- For example, for an object of **Student** class; methods: **getID**, **setGPA**, and **getName** will make the behavior of this object known to us.
- After creating an object from a class, a series of methods are called to accomplish some tasks.
- The following code creates a **Student** object, then it increments the GPA of this student by 0.1:

```
Student s = new Student(243, "Husam" , 3.1);  
double gpa = s.getGPA();  
s.setGPA(gpa + 0.1) ;
```



# Methods: Syntax & Naming

## ➤ Syntax

```
modifier returnType methodName (paraType p1,...) {  
    statement1;  
    statement2;  
    ...  
    statementn;  
    return expression;  
}
```

## Example

```
public double calculateAverage(int n1, int n2){  
    double average = (n1+ n2) / 2.0;  
    return average;  
}  
  
}
```

# Parameters

```
//main method  
Student adel = new Student(987623," Adel", 1.9);  
double increment = 0.1;  
adel.setGPA(adel.getGPA()+ increment);
```

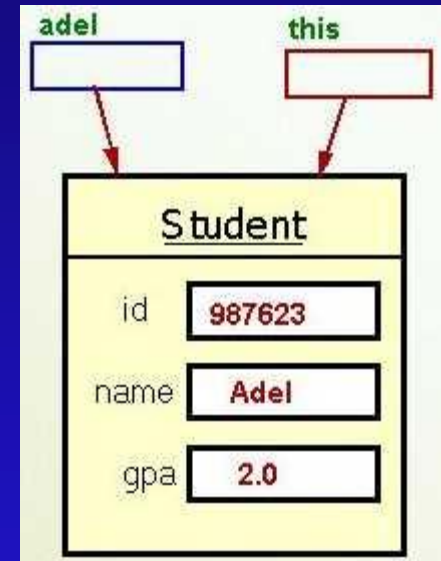


- Actual parameter: Constant, variable or expression in method call.
- Example: 2.0 and **adel** (implicit actual parameter).

```
//method in Student Class  
public void setGPA(double gpa){  
    this.gpa = gpa;  
}
```



- Formal parameter: Variable in the method definition
- Example: grade and **this** (implicit formal parameter).





# Self-check Exercise

- What is the difference between actual parameters and formal parameters?
- What are the problems in the following two program segments of method headers and method calls?



```
// method call
double a = 1.5;
int b = 4;
objectReference.methodA(a, b);
// method header
public void methodA(int n, double x) {.....}
```

```
// method call
double a = 1.5;
int b = 4, c = 2;
objectReference.methodB(a, b, c);
// method header
public void methodB(double x, int n) {.....}
```

- Can we pass an Object reference as an actual parameter?

# Return Type

- The return type indicates the type of value the method returns – a primitive type or object reference.
- If no value or reference is returned by a method, the return type is specified as **void**.

```
public double getGPA() {  
    return gpa;  
}  
public void setGPA(double newGPA) {  
    gpa = newGPA;  
}
```

- A Java method cannot return more than one value.



# Method Overloading

- Methods can be overloaded.
- Overloaded methods:
  - Two or more methods of the same class with the same name but different signatures.
  - The return type of a method is not counted as part of its signature .
- Formal parameters of overloaded constructors and methods must NOT have the same type, order, and count.
- Example: Valid overloading:

```
public double compute(int num, int num2) { . . . }  
public double compute(int num, double num2) { . . . }  
public double compute(int num) { . . . }
```

- Example: Invalid overloading:

```
public double compute(int num) { . . . }  
public int compute(int num) { . . . }
```

# Accessor Methods

- Encapsulation: Data + Methods to operate on the data.
- Only methods of the same class can directly access its private instance variables.
- A public method that returns the private value of an instance variable of an object is called an accessor method.



```
public class Student{
    private int id;
    private String name;
    private double;
    public String getName(){
        return name;
    }
    public int getID(){
        return id;
    }
    public double getGPA(){
        return gpa;
    }
}
```

# Mutator Methods

- A method that changes the value of some instance variable is called a mutator method.

```
public class Student{  
    private int id;  
    private String name;  
    private double gpa;  
    // . . .  
    public void setGPA(double newGPA) {  
        gpa = newGPA;  
    }  
    // . . .  
}
```



- Classes that do not have any mutator methods are called immutable classes. (Example: String class).

# Student class

```
public class Student{
    private int id;
    private String name;
    private double gpa;
    public Student(int id String
        name, double gpa){
        this.id = id;
        this.name = name;
        this.gpa = gpa;
    }
    public Student(int id, String
        name){
        this(id, name, 0.0);
    }
}
```

```
    public String getName(){
        return name;
    }

    public int getID(){
        return id;
    }
    public double getGPA(){
        return gpa;
    }
    public void setGPA(double
        newGPA){
        gpa = newGPA;
    }
}
```

# Exercises

- Write a test program for the Student class. Create two students. Test all accessor and mutator methods.
- Add another constructor to the Student class such that it will take only the name as argument. The ID will be set to 000 and GPA to 0.0
- Add a method called `evaluate(gpa)`. This method will return strings "honor", "good standing" or "under probation" according to the value of gpa.



# Introduction to Methods

- Type of Variables
- Static variables
- Static & Instance Methods
- The toString
- equals methods
- Memory Model
- Parameter Passing



# Types of Variables

➤ There are four types of variables in Java:

variable	declaration	scope	life-time	comment
instance variable	Outside all methods, not proceeded by keyword <b>static</b>	Throughout the class definition	Throughout the object life	Property of each object
static or class variable	Outside all methods, proceeded by keyword <b>static</b> . Example: <b>public static int numberOfCircles;</b>	Throughout the class definition	Throughout the program life	Property of the class. All objects of this class share one copy of the variable.
local variable	Within a method body	Throughout its method	Whenever its method is invoked	Property of each object
formal parameter	In a method or constructor header	Throughout its constructor or method	Whenever its method is invoked	Property of each object

Note: Unnecessary use of static variables should be avoided.

# Type of Variables (cont'd)

static variable

instance variable

formal parameter

formal parameter

local variable

```
public class Circle{  
    private static int numberOfCircles;  
    private double radius;  
  
    public Circle(double theRadius){  
        radius = theRadius;  
        numberOfCircles++;  
    }  
  
    public void setRadius(double newRadius){  
        radius = newRadius;  
    }  
    public double getArea(){  
        double area = Math.PI * radius * radius;  
        return area;  
    }  
}
```

```
graph LR  
    SV[static variable] --> NSI[private static int numberOfCircles;]  
    IV[instance variable] --> ND[private double radius;]  
    FP1[formal parameter] --> TR[double theRadius]  
    FP2[formal parameter] --> NR[double newRadius]  
    LV[local variable] --> DA[double area]
```

# Static vs. Instance Methods

	static or class method	instance method
Declaration	Proceeded by <b>static</b> keyword. Example: <b>public static double sqrt(double x)</b>	Not proceed by <b>static</b> keyword. Example: <b>public double getGPA()</b>
How to call?	<b>ClassName.methodName(parameters)</b> or <b>objectName.methodName(parameters)</b> Example: <b>Math.sqrt(x)</b>	<b>objectName.methodName(parameters)</b> Example: <b>student1.getGPA()</b>
What can be accessed?	Can access constants and static variables; but cannot access instance variables.	Can access constants, static variables, and instance variables.
What can be referenced?	Cannot refer to the <b>this</b> reference.	Can refer to the <b>this</b> reference
When to use?	Implement a task that is not specifically related to an object or that does not need an object.	Implement a particular behavior of an object.

# The finalize Method

- A class whose objects need to perform some task when they are no longer referred to and are about to be garbage collected should redefine the **finalize** method of the **Object** class:

```
void finalize()
```

# Example

```
class Circle{
    private static int numberOfCircles;

    private double radius;

    public Circle(double theRadius){
        radius = theRadius;
        numberOfCircles++;
    }

    public void finalize(){
        numberOfCircles--;
    }

    public static int getCount(){
        return numberOfCircles;
    }
}
```

```
public class StaticTest{
    public static void main(String[] args){
        System.out.println(Circle.getCount());
        Circle circle1 = new Circle(4.0);
        Circle circle2 = new Circle(2.5);
        System.out.println(circle2.getCount());
        circle1 = null;
        /* Request the garbage collector to
           perform a garbage-collection pass */
        System.gc();
        System.out.println(Circle.getCount());
    }
}
```

**Output:**

0

2

1

# toString Method

- Prints the content of an object.
- It is recommended to add `toString()` method when designing a new class.
- When `toString()` is provided it is automatically invoked when object name is used where string is expected (print, concatenation)
- When `toString()` is NOT provided, `toString()` of standard **Object** class is used, with the following format:

`ClassName@memoryLocation`

- Example:

```
class Student{
    private int id;
    private String name;
    private double gpa;
    // . . .
    public String toString(){
        return "Name: " + name + ", ID: " + id + ", GPA: " + gpa;
    }
}
```

```
Student student = new Student(123, "Ahmad", 3.5);
System.out.println(student);

Output:
```

Name: Ahmad, ID: 123, GPA: 3.5

# equals Method

- Compares between the contents of two objects.
- Operator == compares *only references* of two objects.
- It is important to provide equals method when designing a class.
- When this method is not provided the **equals** method of standard **Object** class is used (compares only the references)
- Example:

```
class Student{
    private int id;
    private String name;
    private double gpa;
    public Student(int id, String name,
    double gpa){
        this.id = id;
        this.name = name;
        this.gpa = gpa;
    }

    public boolean equals(Student student){
        return this.id == student.id ;
    }
}
```

```
public class TestStudent{
    public static void main(String[] args){
        Student s = new Student(123,"Ahmed",3.2);
        Student x = new Student(123,"Ahmed",3.2);
        Student y = new Student(456, "Yusuf", 2.0);
        System.out.println(s.equals(x));
        System.out.println(s == x);
        System.out.println(x.equals(y));
    }
}
```

**Output:**

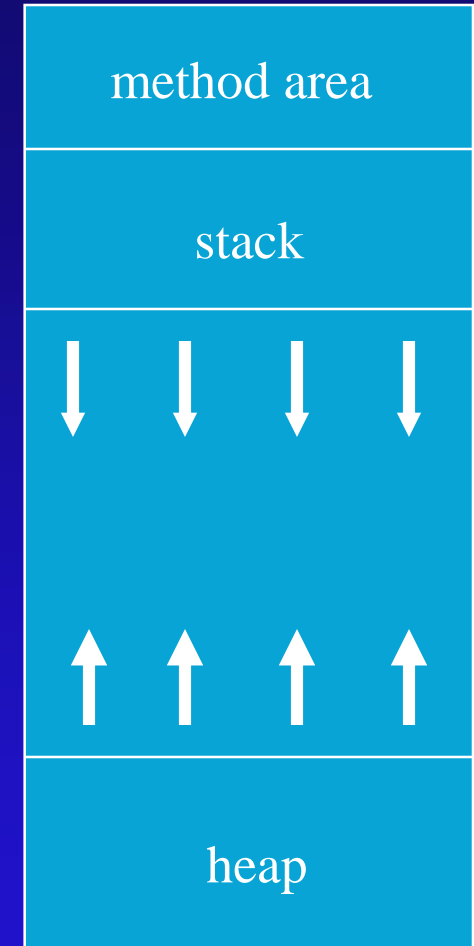
**true**

**false**

**false**

# Memory Model of JVM

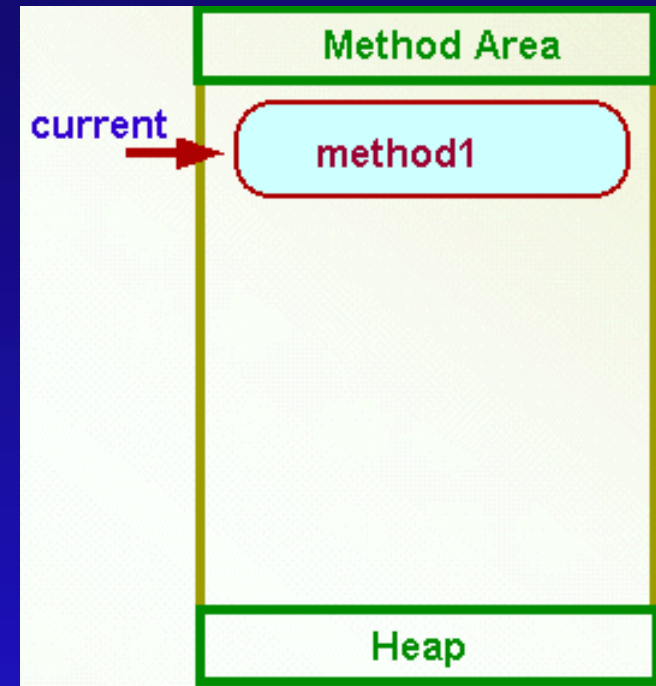
- To execute a Java program:
  - first compile it into bytecode,
  - then invoke the JVM which loads and executes it.
- JVM divides the memory into three main areas:
  - **Method Area:**
    - The bytecode for each method in the class
    - The class variables (static variables)
    - Class information - modifies, etc
    - Methods' information - modifies, return type, etc
  - **Heap:**
    - Memory allocation for object instance variables
    - Reference to Method Area containing class data
  - **Stack:**
    - Keeps track of the order from one method call to another





# Stack

```
class A{  
    void method1(){  
        method2();  
    }  
    void method2(){  
        method3();  
        method4();  
    }  
    void method3(){  
        statements;  
    }  
    void method4(){  
        statements;  
    }  
}
```



## ➤ Activation Record:

- local variables and parameters
- reference to objects created by the method
- return address and return value of the method

# Parameter Passing

- Actual parameters of a method call may contain:
  - Simple or primitive data types (int, double, etc.)
  - Object References.
- In case of simple data type:
  - The value is copied in corresponding formal parameters.
  - Thus, changing the value in the method will not affect the original value.
- In case of object reference:
  - Another reference is created which points to the same object.
  - This other reference may be used to change the values of the object.

# Parameter Passing Example

```
public class StudentTest{
    static int i = 10;
    public static void main(String[] args) {
        String str = "First Message";
        Student s1 =new Student(123, "Khalid" ,1.3);
        Student s2=new Student(456, "Amr", 3.1);
        System.out.println("i = " + i);
        System.out.println("str = " + str);
        System.out.println("Student1: " + s1);
        System.out.println("Student2: " + s2);
        mixUp(i, str, s1, s2);
        System.out.println("i = " + i);
        System.out.println("str = " + str);
        System.out.println("Student1: " + s1);
        System.out.println("Student2: " + s2);
    }
    static void mixUp(int i, String str, Student s1, Student s2){
        i++;
        str = "Second Message";
        one = two;
        one.setGPA(3.4);  one.setName("Ali");
    }
}
```

## Output:

**i = 10**

**str = First Message**

**Student1: Name: Khalid, ID: 123, GPA: 1.3**

**Student2: Name: Amr, ID: 456, GPA: 3.1**

**i = 10**

**str = First Message**

**Student1: Name: Khalid, ID: 123, GPA: 1.3**

**Student2: Name: Ali, ID: 456, GPA: 3.4**

# One-Dimensional Arrays

- What are and Why 1-D arrays?
- 1-D Array Declaration
- Accessing elements of a 1-D Array
- Initializer List
- Passing Array as a parameter
- What if size is unknown?
- Array as return type to methods
- Array of Objects

# What are and Why 1D-arrays?

- Some applications require several values in the memory at the same time.
- For example: counting the number of students whose grades are above the *average* in a class of 30
- This involves scanning through the grades two times:
  - First to compute the *average* and second, to count those above average
- How can we scan through 30 grades two times?
  1. Declare 30 variables to store the grades – inconvenient

```
double grade1 = Double.parseDouble(stdin.readLine());
double grade2 = Double.parseDouble(stdin.readLine());
. . .
double grade30 = Double.parseDouble(stdin.readLine());
double average = (grade1 + grade2 + . . . + grade30) / 30;
int count = 0;
if(grade1 > average) count++;
if(grade2 > average) count++;
. . .
if(grade30 > average) count++;
```

# What are and Why 1D-arrays? (cont'd)

2. Use a loop to find the average and ask the user to re-type the values or re-read them from a file for the second scan --inconvenient

```
double grade, average, sum = 0;
for(int i = 1; i <= 30; i++){
    grade = Double.parseDouble(stdin.readLine());
    sum += grade;
}
average = sum / 30;
int count = 0;
for(int i = 1; i <= 30; i++){
    grade = Double.parseDouble(stdin.readLine());
    if(grade > average) count++;
}
```

- Is there a better approach? - Yes, this is what 1-D arrays are for.
- An array is a contiguous collection of variables of the same type, referenced using a single variable. The type is called the "base type" of the array.

# 1-D Array Declaration

- For any type `T`, `T[ ]` is a class, whose instances are arrays of type `T`.
- Thus, the following statement declares a reference variable, *b*, of type `T` array:  

```
T[] b;
```
- For any positive integer *n*, the following expression creates a new `T[ ]` object of size *n* and stores its reference in *b*:  

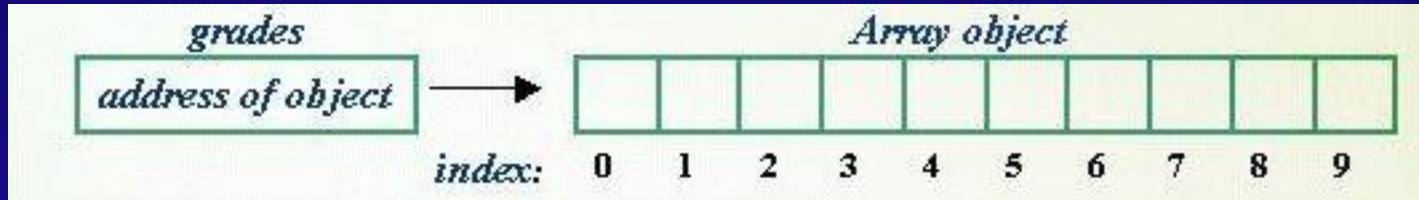
```
b = new T[n] ;
```
- As usual, the two expressions can be combined together as:  

```
T[] b = new T[n] ;
```
- For example, the following declares an `int[ ]`, *grades*, of size 10:  

```
int[] grades = new int[10];
```

# 1-D Array Declaration (cont'd)

```
int[] grades = new int[10];
```



- The declaration of an array of size *n* creates *n* variables of base type.
- These variables are indexed starting from 0 to *n*-1.
- Each array object has a public instance variable, *length*, that stores the size of the array.
- Thus, the following statement prints 10, the size of *grades*:  

```
System.out.println(grades.length);
```

- Other examples of 1D-array declaration are:

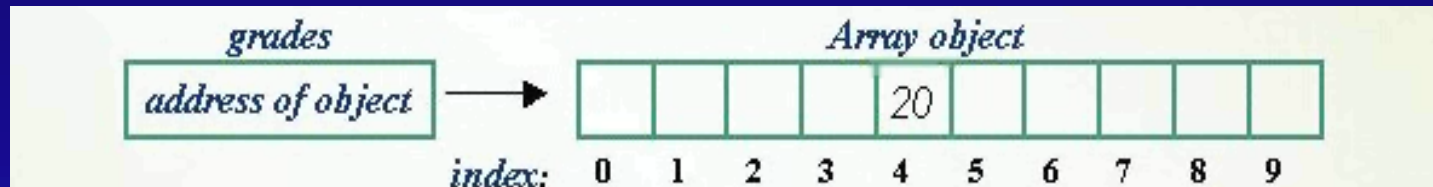
```
double[] price = new double[500];  
boolean[] flag = new boolean[20];
```



# Accessing Elements of a 1-D Array

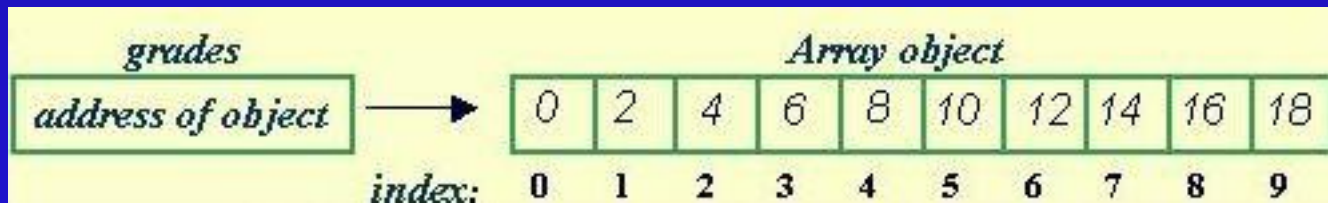
- A particular variable is accessed by indexing the array reference with the index of the variable in bracket:

```
grades[4] = 20;
```



- The following example, initializes each variable with twice its index:

```
int[] grades = new int[10];  
for(int i = 0; i < grades.length; i++)  
    grades[i] = 2*i;
```



- The use of *grades.length* makes the code more general.

# Accessing Elements of a 1-D Array (Cont'd )

- The following prints the values of the array initialized by the example in the previous slide.

```
for(int i = 0; i < grades.length; i++)  
    System.out.print(grades[i] + " ");
```

- Output:

```
0 2 4 6 8 10 14 16 18
```

- Note: Trying to access an element with an invalid index causes a run-time error:

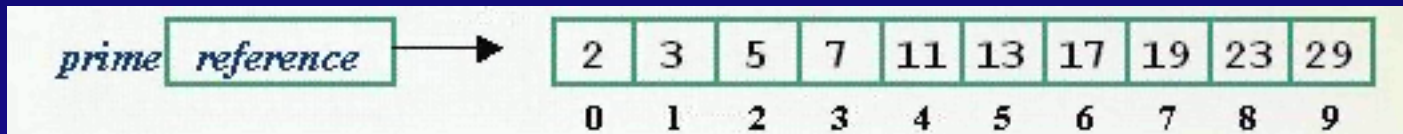
ArrayIndexOutOfBoundsException:

```
int x = grades[10]; // run-time error
```

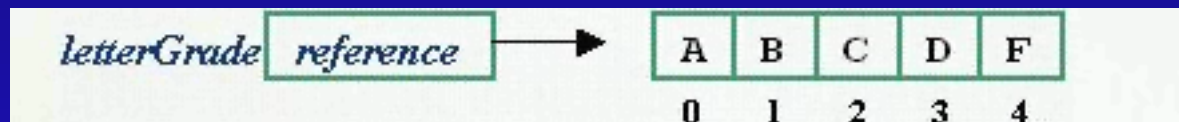
# Initializer List

- Initializer list can be used to instantiate and initialize an array in one step:

```
int[] prime = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29};
```



```
char[] letterGrade = {'A', 'B', 'C', 'D', 'F'};
```



- It is actually the compiler that fills the gap. Thus, in the first example, the compiler would add the following:

```
int[] prime = new int[10];  
prime[0] = 2; prime[1] = 3; ... prime[9] = 29;
```

- Observe that when an initializer list is used:
  - The `new` operator is not required.
  - The size is not required; it is computed by the compiler.

# Passing Array as a Parameter

- To make a method accept an array as argument we must specify its type in the parameter list.
- The following method prints the content of an **int** array passed to it as parameter:

```
public static void printArray(int[] a){  
    for(int i = 0; i < a.length; i++)  
        System.out.print(a[i]+ "  ");  
    System.out.println();  
}
```

- A method can change the values of an array passed to it as parameter:

```
public static void doubleArray(int[] a){  
    for(int i = 0; i < a.length; i++)  
        a[i] = a[i] * 2;  
}
```

# Passing Array as a Parameter (cont'd)

- It is only the reference to the actual array that is passed.
- Thus, any changes done by a method affect the actual array.
- The following uses the two methods of the last slide:

```
public static void main(String[] args){  
    int[] grades = {5, 7, 6, 8, 10};  
    System.out.println("Grades before doubling:");  
    printArray(grades);  
    doubleArray(grades);  
    System.out.println("Grades after doubling:");  
    printArray(grades);  
}
```

- The output is:

```
Grades before doubling:  
5    7    6    8    10  
Grades after doubling:  
10   14   12   16   20
```

# What if size is Unknown?

- The size of an array must be specified before it can be created..
- If the actual size is not known, a reasonably large size is specified.
- An extra variable is then used to keep count of the values stored..

```
public static void main(String[] args) throws IOException{
    BufferedReader stdin = new BufferedReader(
        new InputStreamReader(System.in));
    double[] grade = new double[100];
    int gradeCount = 0;
    System.out.print("EnterNextGrade:");
    double value = Double.parseDouble(stdin.readLine());
    while(value >= 0 && gradeCount < 100){
        grade[gradeCount] = value;
        gradeCount++;
        System.out.print("EnterNextGrade (negative number to terminate):");
        value = Double.parseDouble(stdin.readLine());
    }
    double average = getAverage(grade, gradeCount);
    System.out.println("The average grade is: " + average);
    System.out.println("Grades above average are:");
    printAboveAverage(grade, gradeCount, average);
}
```

# What if size is unknown? (cont'd)

```
public static double getAverage(double[] a, int count){
    if(count == 0) throw new IllegalArgumentException("zero count");
    double sum = 0;
    for(int i = 0 ; i < count; i++)
        sum = sum + a[i];
    return sum/count;
}

public static void printAboveAverage(double [ ] a, int count,
double average){
    for(int i =0 ; i < count; i++)
        if(a[i] > average)
            System.out.println(a[i]);
}
```

# Array as return Type to Methods

- Methods can also have arrays as their return type.
- The following creates two arrays a and b of same size n and print their dot product:

$$a_0b_0 + a_1b_1 + a_2b_2 + \dots + a_{n-1}b_{n-1}$$

```
public static double[] createArray(int size) throws IOException{
    double[] array = new double[size];
    for(int i = 0; i <size; i++){
        System.out.print("Enter element #" + (i+1) + ": ");
        array [i] = Double.parseDouble(stdin.readLine());
    }
    return array;
}

public static void main( String [ ] args) throws IOException {
    int size;
    System.out.print("Enter array size: ");
    size = Integer.parseInt(stdin.readLine());
    double[] a = createArray(size);
    double[] b = createArray(size);
    System.out.println ( "The dot product = "+dotProduct(a, b));
}
```

- The implementation of dotProduct is left as an exercise.



# Array of Objects

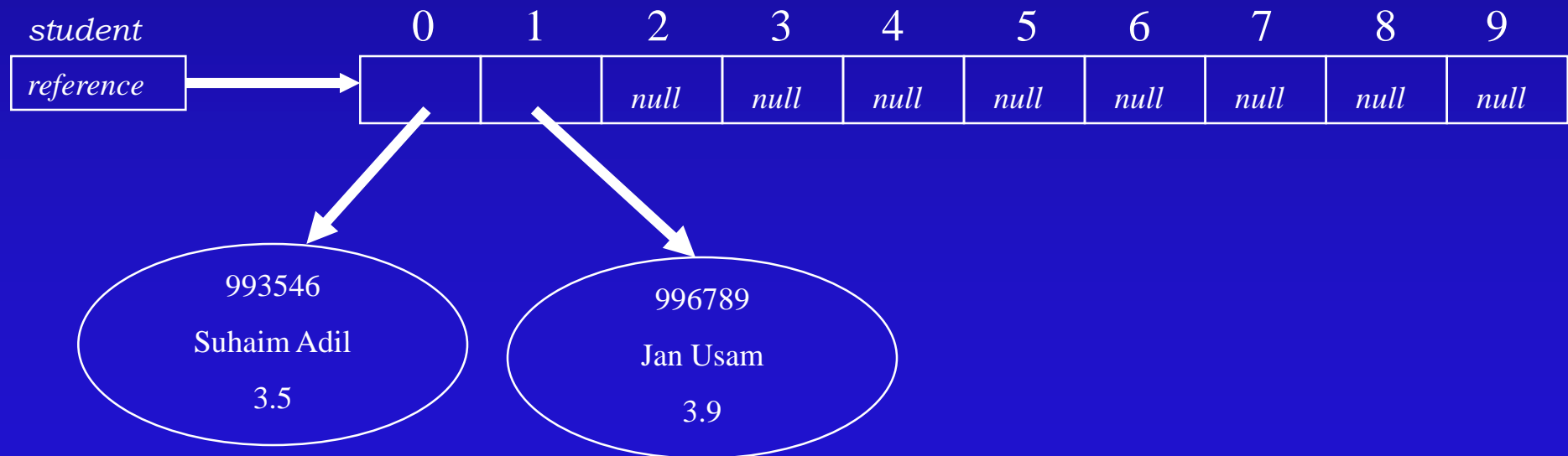
- So far, our examples have been on arrays of primitive types.
- We can equally create arrays whose elements are objects.
- We can create an array to store 10 **Student** objects as follows:

```
Student[] student = new Student[10];
```

- However, only the references to the **Student** objects are stored.
- The figure shows the array after adding two **Student** objects.

```
student[0] = new Student(993546, "Suhaim Adil", 3.5);
```

```
student[1] = new Student(996789, "Jan Usam", 3.9);
```



# Array of Objects (cont'd)

- The following method takes **size**, and returns an array initialized with **size Student** objects.

```
public static Student[] createArray(int size) throws IOException{
    Student[] array = new Student[size];
    String name;
    int id;
    double gpa;
    for(int i = 0; i < size; i++){
        System.out.print("ID Number : ");
        id = Integer.parseInt(stdin.readLine());
        System.out.print("Name : ");
        name = stdin.readLine();
        System.out.print("GPA : ");
        gpa = Double.parseDouble(stdin.readLine());
        array[i] = new Student(id, name, gpa);
    }
    return array;
}
```

# Array of Objects (cont'd)

- Each array element is treated exactly as a reference to an object.
- For example, to call the `getName()` method of the object at cell 0:  
`student[0].getName();`
- The following takes an array of Students and prints those with GPA $\geq$ 2.0

```
public static void printGoodStanding(Student[] student) {  
    for (int i=0; i<student.length; i++)  
        if(student[i].getGPA() >= 2.0)  
            System.out.println(student[i]);  
}
```

- The two methods are called as follows:

```
public static void main(String[] args) throws IOException {  
    int size;  
    System.out.print("Enter number of students: ");  
    size = Integer.parseInt(stdin.readLine());  
    Student[] student = createArray(size);  
    printGoodStanding(student);  
}
```

# Dynamic Arrays

- Why Dynamic Arrays?
- A Dynamic Array Implementation
- The Vector Class
- Program Example
- Array Versus Vector

# Why Dynamic Arrays?

- A problem with arrays is that their size must be fixed at creation.
- Thus, once an array of size  $n$  is declared, it cannot be extended to hold more than  $n$  elements.
- But the programmer may not know the size required.
- Is there a way out?
- Yes, Java provides the *Vector* class in the *java.util* package that can grow dynamically as needed.
- To understand how it works, we shall implement a similar class.

# A Dynamic Array Implementation

- The following defines a class that works like array but whose size can grow dynamically:

```
public class DynamicArray{
    private int[] b;
    private int numberOfElements;
    //Constructor: Creates an array with default size of 10
    public DynamicArray(){
        b = new int[10];
    }
    //Constructor: Creates an array with specified size
    public DynamicArray(int size){
        b = new int[size];
    }
    public int size(){
        return numberOfElements;
    }
    public int capacity(){ // returns total number of cells
        return b.length;  // including unused ones
    }
}
```

# A Dynamic Array Implementation

```
public int getElement(int i){
    if(i < 0 || i > numberOfElements - 1)
        throw new IllegalArgumentException("index out of Bounds");
    return b[i];
}

public void set(int i, int value){
    if(i < 0 || i > numberOfElements)
        throw new IllegalArgumentException("index out of Bounds");
    if(i == numberOfElements && i == b.length){
        // For efficiency purposes, double the array capacity
        int[] newb = new int[2*b.length];
        for(int k = 0; k < numberOfElements; k++)
            newb[k] = b[k];
        b = newb;
    }
    b[i] = value;
    if(i == numberOfElements)
        numberOfElements++;
}}
```

# Our Class Versus Array

- We can create an instance of our `DynamicArray` class as follows:

```
DynamicArray c = new DynamicArray(20);
```

- `c` can be viewed as an *int* array of size 20. However, `c` can store more than 20 integers.
- How we access and modify content of `c` is also slightly different:
  1. Corresponding to: `b[i]` , we use a function call: `c.getElement(i)`
  2. Corresponding to: `b[i]=value;` , we use a function call: `c.set(i,value);`
  3. Corresponding to: `b.length` , we use a function call: `c.capacity()`



# The Vector Class

- *Vector* is similar to our *DynamicArray*, but it has much more.
- It has the following Constructors:

<code>Vector()</code>	Creates a vector of size 10, It doubles the capacity when exhausted.
<code>Vector(int initialCapacity)</code>	Creates a vector of size <code>initialCapacity</code> , It doubles the capacity when exhausted.
<code>Vector(int initialCapacity, int increment)</code>	Creates a vector of size <code>initialCapacity</code> , increases by <i>increment</i> when the capacity is exhausted.

- To Create a vector with an initial capacity of 20 elements:

```
Vector v = new Vector(20);
```

# Adding an Element

- The base type of our DynamicArray is int What about Vector?
- The base type of Vector is *Object*.
- Thus, primitive types must be wrapped using wrapper classes.
- Elements can be added using the following add method:

<code>add(Object element)</code>	Adds element to the next empty cell, increases the capacity if necessary.
----------------------------------	--

- The following adds 10 objects into a vector of initial capacity 4:

```
Vector v = new Vector(4);  
for(int i = 0; i < 10; i++)  
    v.add(new Integer(i*2));
```

- The capacity is automatically increased to take the 10 objects

# Adding an Element (cont'd)

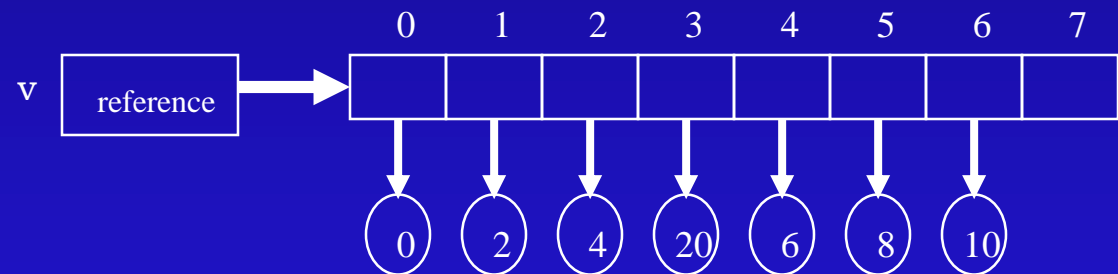
- The following add method can also be used to add an element:

```
add(int index, Object  
element)
```

Adds element at index, shifts element at index and beyond, if any, by 1. Increases capacity if necessary. If `index < 0` or `index > size`, it throws `IndexOutOfBoundsException`.

- The following adds 6 objects into a vector, it then inserts an Integer object with value of 20 at index 3:

```
Vector v = new Vector(4);  
for(int i = 0; i < 6; i++)  
    v.add(new Integer(i*2));  
v.add(3, new Integer(20));
```



- Note that this method does not allow an empty cell in-between:

```
v.add(8, new Integer(100)); // Run-time error, cell 7 will be empty
```

# Accessing and Changing an Element

- The following method can be used to access an element:

<code>Object get(int index)</code>	Returns the element at index. throws <code>IndexOutOfBoundsException</code> if <code>index &lt; 0</code> or <code>index &gt; size</code> .
------------------------------------	--

- However since the return type is object, we have to cast-down to get the original object.

- The following prints the result of dividing the element at index 3 with 2:

```
Integer element = (Integer) v.get(3);  
System.out.println(element.intValue() / 2);
```

- To modify an element we use the `set` method:

<code>Object set(int index, Object element)</code>	Replaces the object at index with element and returns the replaced object. Throws <code>IndexOutOfBoundsException</code> if <code>index &lt; 0</code> or <code>index &gt; size</code> .
--	---

- The following replaces the element at index 3 with 100:

```
v.set(3, new Integer(100));
```

# Searching for an Element

- To check if an element is in a vector, use the `contains` method:

<code>boolean contains(Object element)</code>	Returns true if element is contained in the vector and false otherwise .
---	--

- The `contains` method uses the equals method of the vector element in its search.
- The following checks if the vector `v` contains 100:

```
if(v.contains(new Integer(100))
    System.out.println("100 found");
else
    System.out.println("100 not found");
```

- If you also need to know the index of the object when found, use:

<code>int indexOf(Object element)</code>	Returns the index of element if found; -1 otherwise.
--	--

```
int index = v.indexOf(new Integer(100));
if(index != -1)
    System.out.println("100 found at index " + index);
else
    System.out.println("100 not found");
```

# Size versus Capacity

➤ Two related accessor methods for the Vector class are:

<code>int capacity()</code>	Returns the current capacity of the vector.
<code>int size()</code>	Returns the actual number of elements stored in the vector.

➤ **size()** is more useful. It is usually used in a loop to process all the vector elements

```
public static void main(String[] args){  
    Vector v = new Vector(4);  
    System.out.println("SIZE\tCAPACITY");  
    for(int i = 0; i < 10; i++){  
        v.add(new Integer(i*2));  
        System.out.println(v.size()+"\t"  
            + v.capacity());  
    }  
    for(int i = 0; i < v.size(); i++)  
        System.out.println(v.get(i) + " ");  
}
```

Output:

SIZE	CAPACITY
1	4
2	4
3	4
4	4
5	8
6	8
7	8
8	8
9	16
10	16
0	2
2	4
4	6
6	8
8	10
10	12
12	14
14	16
16	18

# Removing an Element

- An element can be removed from a vector using any of the following methods:

<code>boolean remove(Object element)</code>	Removes element from the vector and returns true if successful; returns false if element is not found. The elements after the removed element are shifted to the left
<code>Object remove(int index)</code>	Removes element at index and returns it; throws <code>IndexOutOfBoundsException</code> if <code>index &lt; 0</code> or <code>index &gt; size</code> . The elements after the removed element are shifted to the left

- The following removes 100 from a vector `v` and prints a message if successful:

```
if(v.remove(new Integer(100))
    System.out.println("100 removed");
else
    System.out.println("100 not found");
```

- Does the capacity of a vector shrink automatically after a deletion?:
- No. However, we can use the following method to shrink it:

<code>void trimToSize()</code>	Trims the capacity of this vector to be the vector's current size.
--------------------------------	--

# Program Example

- The following example reads an unknown number of grades, it then prints the average and the grades above the average:

```
public static void main(String[] args) throws IOException{
    Vector grade = new Vector();
    readGrades(grade);
    double average = getAverage(grade);
    System.out.print("The average grade is: " + average);
    System.out.print("Grades above average are: ");
    printAboveAverage(grade, average);
}

public static void readGrades(Vector v) throws IOException{
    BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));
    System.out.print("Enter grade: ");
    double value = Double.parseDouble(stdin.readLine());
    while(value >= 0){
        v.add(new Double(value));
        System.out.print("Enter next grade(negative value to terminate): ");
        value = Double.parseDouble(stdin.readLine());
    }
}
```



# Program Example (cont'd)

```
public static double getAverage(Vector v) {
    if(v.size() == 0)
        throw new IllegalArgumentException("vector size is zero");
    double sum = 0;
    for(int i = 0; i < v.size(); i++){
        Double element = (Double) v.get(i);
        sum = sum + element.doubleValue();
    }
    return sum/v.size();
}

public static void printAboveAverage(Vector v, double average){
    for(int i = 0; i < v.size(); i++){
        Double element = (Double) v.get(i);
        if(element.doubleValue() > average)
            System.out.println(element);
    }
}
```

# Array versus Vector

- Vectors can grow and shrink, arrays cannot.
- Vector elements must be object references. Array elements can be object references or a primitive type.

# Multi-Dimensional Arrays

- Why do we Need 2-D arrays?
- 2-D Array Declaration
- Referencing Elements of a 2-D Array
- What is the Meaning of length in a 2-D array?
- 2-D Array Initializers
- Processing 2-D arrays
- Closer Look at 2-D Arrays in Java
- Ragged Arrays

# Why do we Need 2-D arrays?

- The same reason that necessitated the use of 1-D arrays can be extended to 2-D and other multi-D Arrays.
- For example, to store the grades of 30 students, in 5 courses require multiple 1-D arrays.
- A 2-D array allows all these grades to be handled using a single variable.
- This idea can be easily extended to other higher dimensions.
- Thus, we shall focus on 2-D arrays.
- A 2-D array is a contiguous collection of variables of the same type, that may be viewed as a table consisting of rows and columns.

# 2-D Array Declaration

- For any type **T**, **T[ ][ ]** is a class of 2-D arrays of base type **T**.
- Thus, the following statement declares a reference variable, **b**, of type 2-D array of base type **T**.

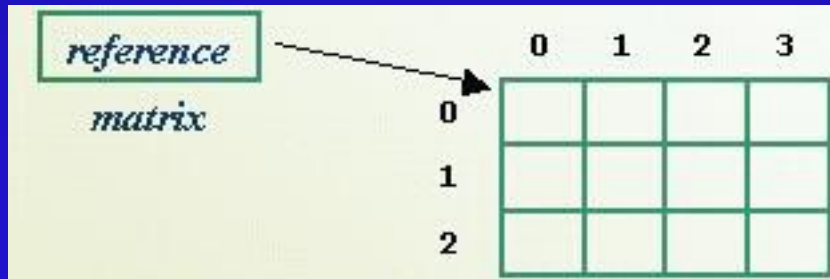
```
T[ ][ ] b;
```

- For any two positive integers **r** and **c**, the following expression creates a 2-D array of type **T** with **r** rows, **c** columns and stores its reference in **b**.

```
b = new T[r][c];
```

- The following creates a 2-D array, **matrix**, of **int** type with 3 rows and 4 columns:

```
int[ ][ ] matrix = new int[3][4];
```



- Both rows and columns are indexed from zero.

# Referencing Elements of a 2-D Array

- A particular element of a 2-D array, *b*, is referenced as:

`b[RowIndex][columnIndex]`

- For example, given the declaration:

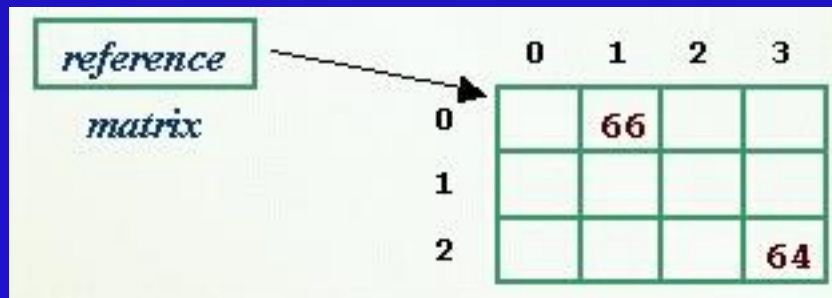
```
int[][] matrix = new int[3][4];
```

- The following stores 64 in the cell with row index 2, column index 3.

```
matrix[2][3] = 64;
```

- We use the same format to refer to an element in an expression:

```
matrix[0][1] = matrix[2][3] + 2;
```



# What is the Meaning of length in a 2D-array?

- Suppose a matrix is declared as a 2D-array as:

```
int[][] matrix = new int[3][4];
```

- Then `matrix.length` returns the number of rows in matrix; 3 in this case.

- We can use this number to process a particular column:

```
int sumColumn0 = 0;
for(int rowIndex = 0; rowIndex < matrix.length; rowIndex++)
    sumColumn0 = somColumn0 + matrix[rowIndex][0];
```

- To manipulate a particular row n, We use the expression: `matrix[n].length` to obtain its length:

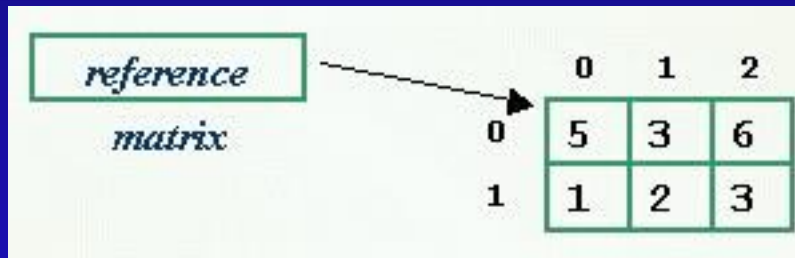
```
int sumRow2 = 0;
for(int columnIndex = 0; columnIndex < matrix[2].length; columnIndex++)
    sumRow2 = sumRow2 + matrix[2][columnIndex];
```

# 2D-Array Initializers

- We can declare and initialize a 2D-array in one statement as follows:

```
int[][] matrix = {{5, 3, 6}, {1, 2, 3}};
```

- Thus, each row is represented by a 1D-array initializer.



- To have three rows, we just add another 1D-array initializer:

```
int[][] matrix = {{5, 3, 6}, {1, 2, 3}, {8, 4, 3}};
```

- Again, this notation can be extended to higher dimensions.



# General Algorithm - Processing 2-D Arrays

- • We can extend algorithms for processing 1-D arrays to 2-D arrays by using nested loops.

- To process an array row-wise, we use:

```
for(int rowIndex = 0; rowIndex < array.length; rowIndex++){  
    process row# rowIndex;  
}
```

- But processing a row involves processing each element in that row.

```
for(int rowIndex = 0; rowIndex < array.length; rowIndex++){  
    // process row# rowIndex  
    for(int columnIndex = 0; columnIndex < array[rowIndex].length;  
columnIndex++){  
        process element array[rowIndex][columnIndex]  
    }  
}
```

- To process an array, whose columns have equal length, column-wise, we use:

```
for(int columnIndex = 0; columnIndex < array[0].length;  
columnIndex++){  
    // process column# columnIndex  
    for(rowIndex = 0; rowIndex < array.length; rowIndex++){  
        process element array[rowIndex][columnIndex]  
    }  
}
```

# Examples on Processing 2-D arrays

- The following method prints the elements of a 2-D array one row per line.

```
public void print2Darray(int[][] matrix){  
    for(int rowIndex = 0; rowIndex < matrix.length; rowIndex++){  
        // process row# rowIndex  
        for(int columnIndex = 0; columnIndex < matrix[rowIndex].length; columnIndex++){  
            System.out.print(array[rowIndex][columnIndex] + "\t");  
        }  
        System.out.println();  
    }  
}
```

- The following method takes two matrices of equal dimensions and return their sum.

```
public int[][] sum2Darray(int[][] a, int[][] b){  
    int[][] matrix = new int[a.length][a[0].length];  
    for(int rowIndex = 0; rowIndex < matrix.length; rowIndex++){  
        for(int columnIndex = 0; columnIndex < matrix[rowIndex].length; columnIndex++){  
            matrix[rowIndex][columnIndex] = a[rowIndex][columnIndex]+  
                                              b[rowIndex][columnIndex] ;  
        }  
    }  
    return matrix;  
}
```

# Processing 2D-Arrays of Objects

- Assume the **Student** class has the following additional members:

```
private double grade;

public double getGrade(){
    return grade;
}
```

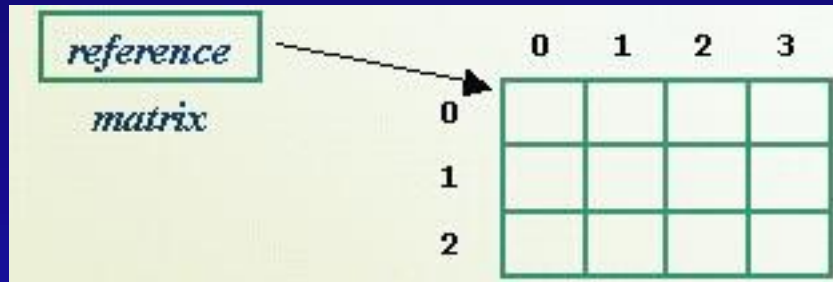
- Then the following method takes a 2D **Student** array representing a multi-section course and returns the overall average:

```
public double overallAverage(Student[][] student){
    double sum = 0; int numberOfStudents = 0;
    for(int section = 0; section < student.length; section++){
        for(int count = 0; count < student[section].length; count++){
            sum = sum + student[section][count].getGrade() ;
            numberOfStudents++;
        }
    }
    return sum / numberOfStudents;
}
```

# A closer look at 2D-arrays in Java

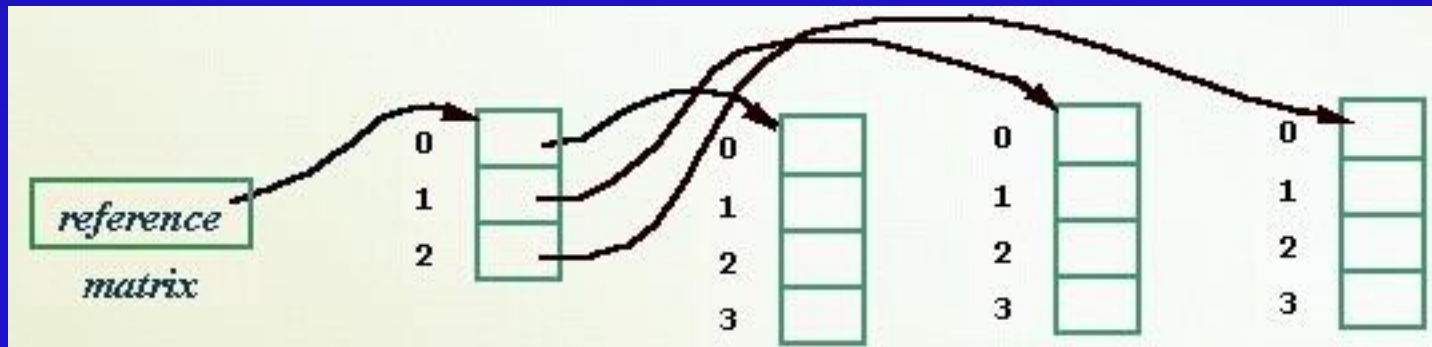
- So far, we have described a 2D-array as a table, consisting of rows and columns:

```
int[][] matrix = new int[3][4];
```



- Although this is natural, in Java a 2D-array is actually a 1D-array whose elements are references to 1D-array objects.

- Here is the actual representation:



# A Closer Look at 2D-Arrays in Java (cont'd)

➤ The creation of the 2D-array involves three steps:

1. The creation of the reference **matrix**:

```
int[][] matrix;
```



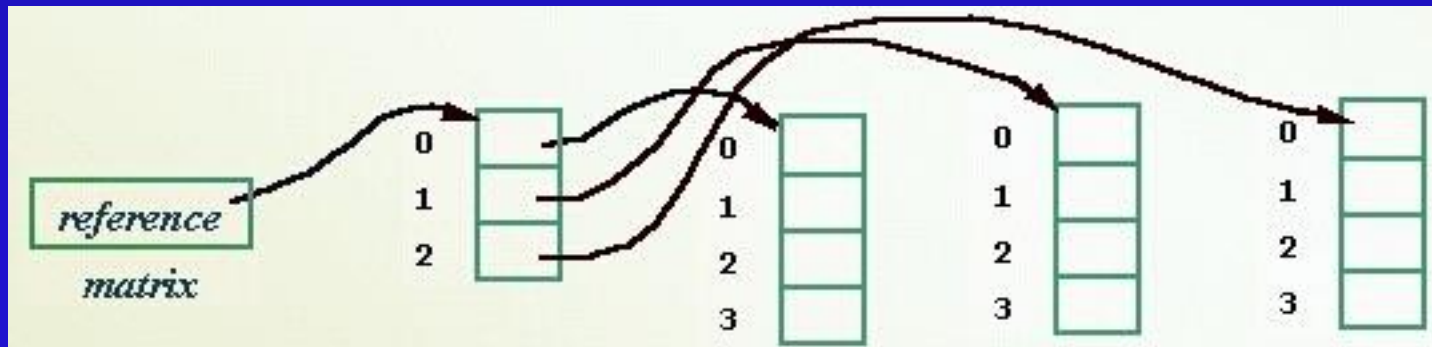
2. The creation of a 1D-array whose elements are references to 1D-arrays:

```
matrix = new int[3][];
```



3. The creation of the individual 1D-arrays:

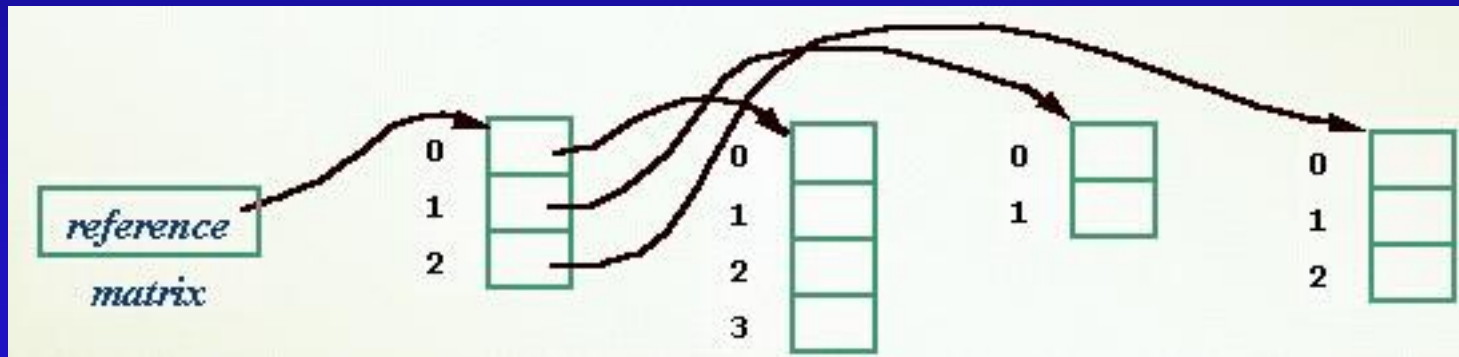
```
for(int i = 0; i < matrix.length; i++)  
    matrix[i] = new int[4];
```



# Ragged Arrays

- Since each column of a 2-D array is actually a separate 1D-array, each column may have a different length.
- For example, we could have the following:

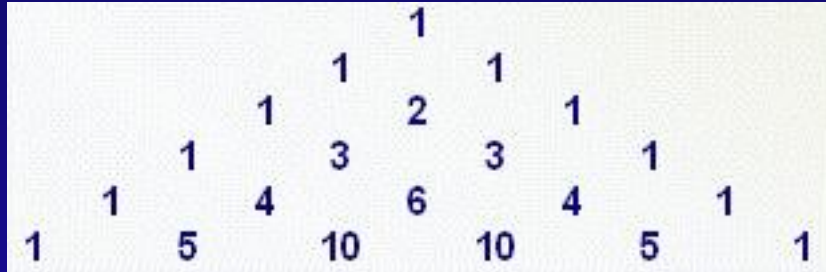
```
int[][] matrix = new int[3][];  
matrix[0] = new int[4];  
matrix[1] = new int[2];  
matrix[2] = new int[3];
```



- This type of 2D-array where the columns are not of the same size is called a **ragged** array.
- Ragged arrays can be used to save memory space in some programming situations.

# Example: Using a Ragged Array

- Pascal's triangle has many applications in Mathematics:



- The following method returns a ragged array representing Pascal's triangle of n rows:

```
public static int[][] pascalTriangle(int n){
    int[][] b = new int[n][];
    for(int i = 0; i < n; i++){
        // Create row I of Pascal's triangle
        b[i] = new int[i+1];
        b[i][0] = 1;
        for(int k = 1; k < i; k++){
            b[i][k] = b[i - 1][k - 1] + b[i - 1][k];
        }
        b[i][i] = 1;
    }
    return b;
}
```

# Introduction to Recursion

- Introduction to Recursion
- Example 1: Factorial
- Example 2: Reversing Strings
- Example 3: Fibonacci
- Infinite Recursion
- Review Exercises



# Introduction to Recursion

- We saw earlier that a method can call another method leading to the creation of activation records on the runtime stack.
- Recursion is one of the powerful techniques of solving problems.
- A *recursive method* is a method that calls itself *directly or indirectly*
- A *well-defined* recursive method has:
  - A base case that *determines the stopping condition in the method*
  - A recursive step *which must always get “closer” to the base case from one invocation to another.*
- The code of a recursive method must *be structured to* handle both the base case and the recursive case.
- Each call to the method sets up a new execution environment, *with new parameters and local variables.*

# Example 1: The Factorial Function

$$\text{factorial}(n) = \begin{cases} 1, & \text{if } n = 0 \\ n * \text{factorial}(n-1), & \text{if } n > 0 \end{cases}$$

```
public class Factorial{
    public static void main(String[]
args){
        long answer = factorial(5);
        System.out.println(answer);
    }
}
```

```
public long factorial(int n)
{
    if(n == 0)
        return 1L;
    else
        return n*factorial(n-
1);
}
```

# The Factorial: Sample Execution Trace

```
factorial (6) =  
    = (6*factorial(5))  
    = (6*(5*factorial(4)))  
    = (6*(5*(4*factorial(3))))  
    = (6*(5*(4*(3*factorial(2)))))  
    = (6*(5*(4*(3*(2*factorial(1))))))  
    = (6*(5*(4*(3*(2*1))))))  
    = (6*(5*(4*(3*2))))  
    = (6*(5*(4*6)))  
    = (6*(5*24))  
    = (6*120)  
    = 720
```

## Example 2: Reversing Strings

```
public void reverseString(String str, int i) {  
    if(i < str.length()){  
        reverseString(str, i+1);  
        System.out.print(str.charAt(i));  
    }  
}
```

# Example 3: The Fibonacci Function

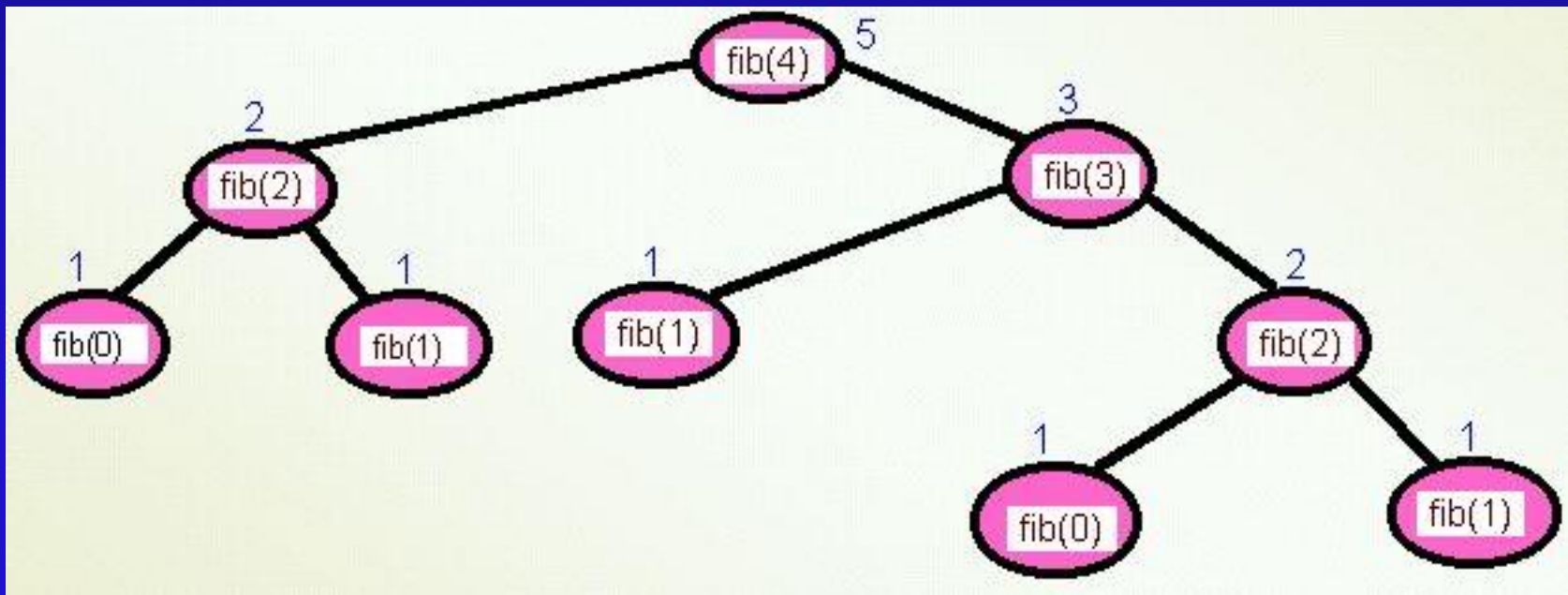
$$\text{fibonacci}(n) = \begin{cases} 1, & \text{if } n < 2 \\ \text{fibonacci}(n-1) + \text{fibonacci}(n-2), & \text{if } n \geq 2 \end{cases}$$

```
public class Fibonacci{  
    public static void  
        main(String[] args){  
        long answer = fibonacci(4);  
        System.out.println(answer);  
    }  
}
```

```
public static long fibonacci(int n) {  
    if (n < 2)  
        return 1L;  
    else  
        return fibonacci(n-1) + fibonacci(n-2);  
}
```

# Fibonacci Call Tree

```
public static long fibonacci(int n) {  
    if (n < 2)  
        return 1L;  
    else  
        return fibonacci(n-1) + fibonacci(n-2);  
}
```



# Infinite Recursion

- A recursive method must always call itself with a smaller argument
- Infinite recursion results when:
  - The base case is omitted.
  - Recursive calls are not getting closer to the base case.
- In theory, infinite recursive methods will execute “forever”
- In practice, the system reports a stack overflow error.

# Drill Questions

1. Write a recursive method `public int sumUpTo(int n)` which adds up all integers from 1 to n.
2. Write a recursive method `public int multiply(x, y)` that multiplies two integers `x` and `y` using repeated additions and without using multiplication.
3. Write a recursive method  
`public int decimalToBinary(int n)` that takes an integer parameter and prints its binary equivalent.
4. Write a recursive method  
`public boolean isPalindrome(String str)`  
that returns true if `str` is a palindrome and returns false otherwise.

*Note: a palindrome is a string that has the same characters when read from left to right or from right to left (Examples: eye, radar, madam, dad, mom, 202).*



# Exception Handling

- Introduction to Exceptions
- How exceptions are generated
- A partial hierarchy of Java exceptions
- Checked and Unchecked Exceptions
- What exceptions should be handled?
- How exceptions are handled
- Unreachable catch-blocks
- The semantics of the try statement
- Recovering from Exceptions

# Introduction to Exceptions

- A program may have one or more of three types of errors:
  - Syntax errors or Compile-time errors.
  - Run-time or Execution-time errors.
  - Logic errors.
- A Java exception is an **object** that describes a **run-time error** condition that has occurred in a piece of Java code or in the Java run-time System.
- All Java exception classes are subclasses of the **Throwable** class.
- Most exception classes are defined in the **java.io** and **java.lang** packages.
- Other packages like: **java.util**, **java.awt**, **java.net**, **java.text** also define exception classes.
- A piece of Java code containing statements to handle an exception is said to **catch** the exception; otherwise it is said to **throw** that exception.

# How Exceptions are Generated

- Exceptions can be generated in two ways:
  - By the Java run-time system.
  - By the programmer using the *throw* statement.

- Common forms of the throw statement are:

- `throw new ThrowableClass();`
- `throw new ThrowableClass(string);`

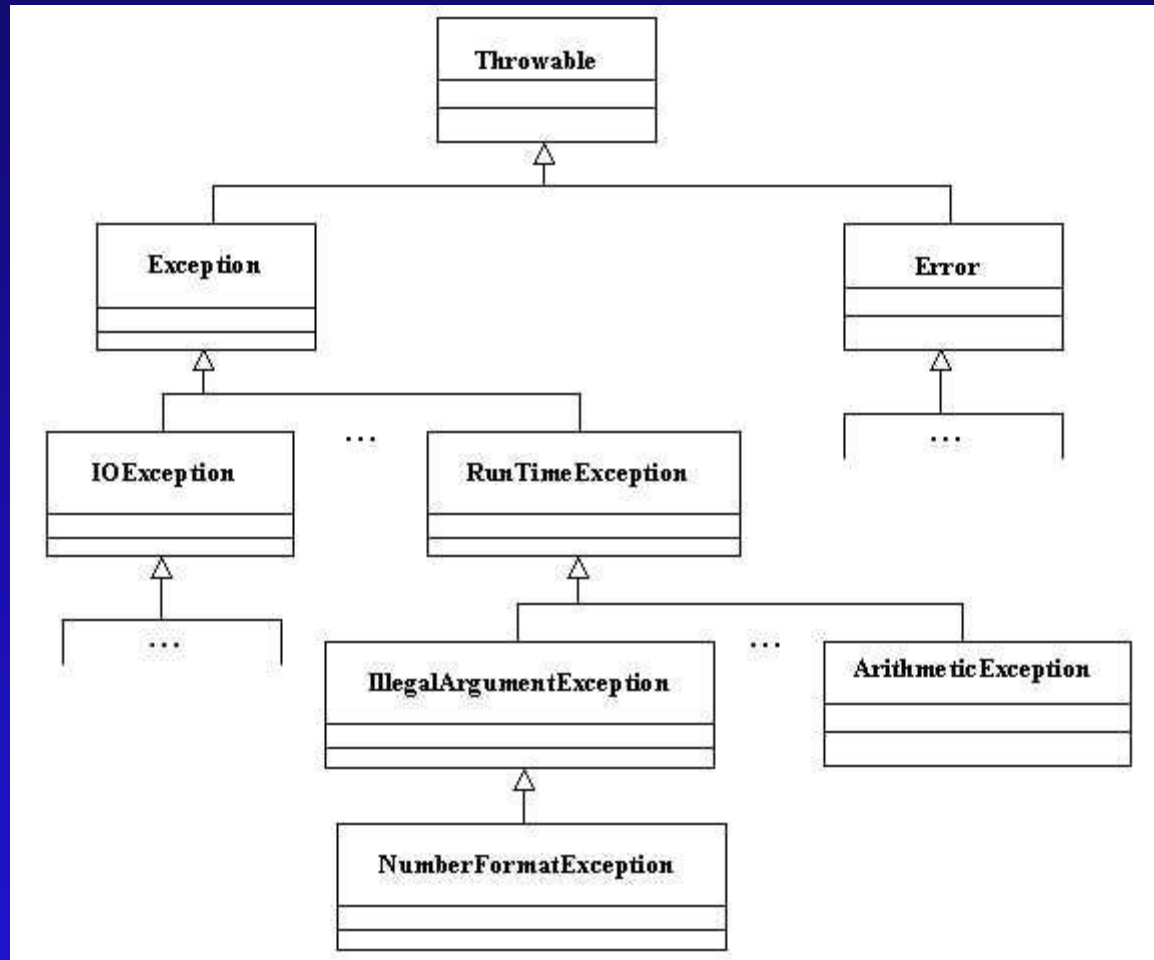
where ThrowableClass is either Throwable or a subclass of Throwable.

- Examples:

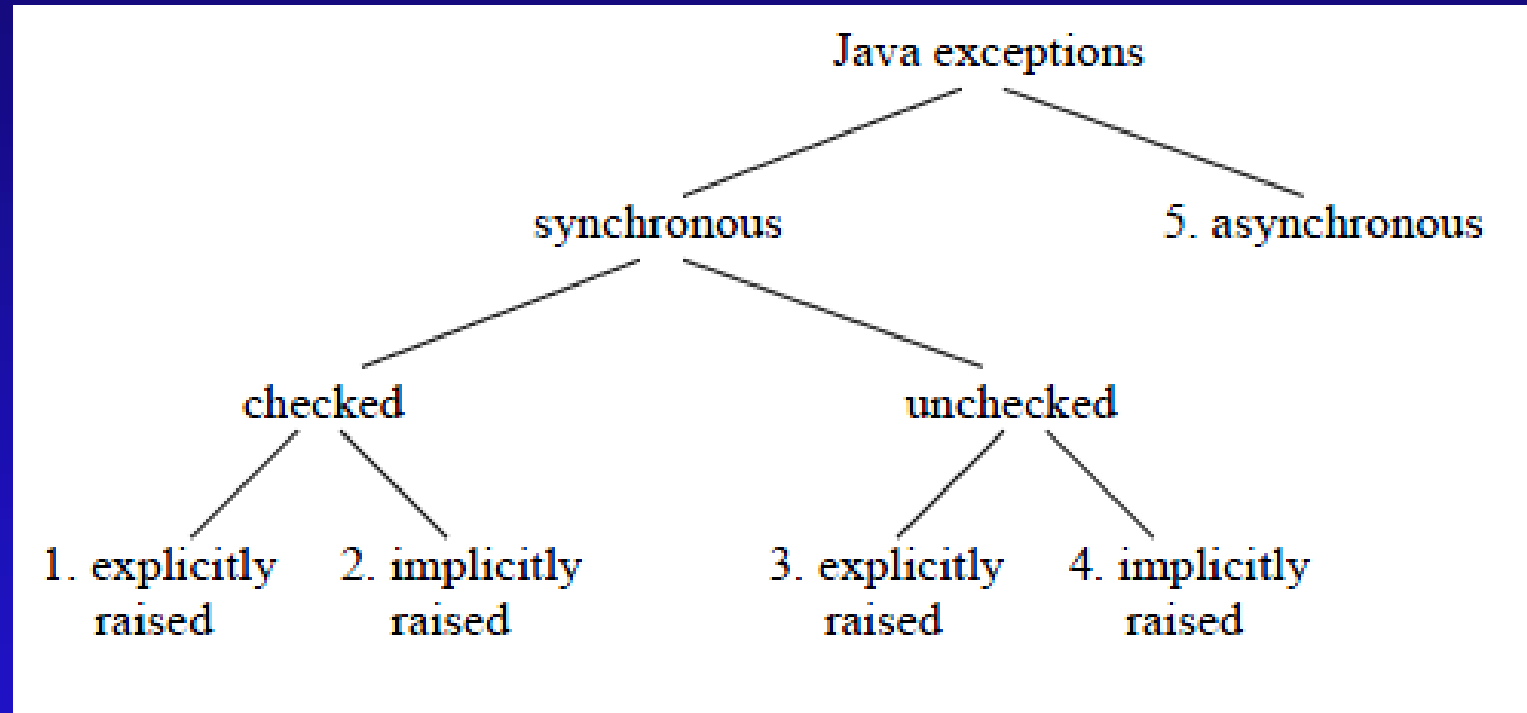
```
if(divisor == 0)
    throw new ArithmeticException("Error - Division by zero");
```

```
if(withdrawAmount < 0)
    throw new InvalidArgumentException("Error - Negative withdraw amount");
else
    balance -= withdrawAmount;
```

# A partial hierarchy of Java exceptions



# A partial hierarchy of Java exceptions



# Checked and Unchecked Exceptions

- Java exceptions are classified into two categories: checked exceptions and unchecked exceptions:
  - **Checked exception:** An object of Exception class or any of its subclasses except the subclass RuntimeException.
  - **Unchecked exception:**
    - An object of the Error class or any of its subclasses.
    - An object of RuntimeException or any of its subclasses.
- A method that does not handle checked exceptions it may generate, must declare those exceptions in a **throws** clause; otherwise a compile-time error occurs.
- A method may or may not declare, in a throws clause, unchecked exceptions it may generate but does not handle.
- Syntax of a throws clause:

```
accessSpecifier returnType methodName(parameters) throws  
    ExceptionType1, ExceptionType2, ..., ExceptionTypeN {  
    // method body  
}
```
- Note: When a method declares that it throws an exception, then it can throw an exception of that class or any of its subclasses.

# Checked and Unchecked Exceptions (cont'd)

## ➤ Examples:

```
public static void main(String[] args) throws IOException{  
    // . . .  
    String name = stdin.readLine();  
    // . . .  
}
```

If `IOException` is not handled, the **throws `IOException`** is required

```
public static void main(String[] args){  
    // . . .  
    int num1 = num2 / num3;  
    // . . .  
}
```

Statement will generate `ArithmeticException` if `num3` is zero.

The method may or may not contain a **throws `ArithmeticException`** clause

# What Exceptions Should be Handled?

- Exceptions of class **IOException** and its subclasses should be handled.
  - They occur because of bad I/O operations (e.g., trying to read from a corrupted file).
- Exceptions of class **RuntimeException** and its subclasses and all subclasses of **Exception** (except **IOException**) are either due to programmer-error or user-error.
  - Programmer-errors should not be handled. They are avoidable by writing correct programs.
  - **User-errors should be handled.**
  - Example: Wrong input to a method may generate:  
    **ArithmeticException, IllegalArgumentException, or NumberFormatException.**
- The exceptions of class **Error** and its subclasses should not be handled.
  - They describe internal errors inside the Java run-time system. There is little a programmer can do if such errors occur.
  - Examples: **OutOfMemoryException, StackOverflowException.**



# How Exceptions are Handled

- Java uses **try-catch** blocks to handle exceptions.
- **try-catch** blocks have the form:

```
try{
    statementList
}
catch (ExceptionClass1  variable1) {
    statementList
}
catch (ExceptionClass2  variable2) {
    statementlist
}
. . .
catch (ExceptionClassN  variableN) {
    statementlist
}
```

- A statement or statements that may throw exceptions are placed in the try-block.
- A **catch**-block defines how a particular kind of exception is handled.

# Unreachable catch-blocks

- A catch-block will catch exceptions of its exception class together with any of its subclasses.
  - Example: Since `ArithmeticException` is a subclass of `Exception`, the following code will cause compile-time error because of unreachable code:

```
try{
    int a = 0;
    int b = 42 / a;
}
catch (Exception e) {
    System.out.println(e);
}
catch (ArithmeticException e) {
    System.out.println("There is an arithmetic exception");
}
```

# The Semantics of the try Statement

- A **catch** block cannot catch an exception thrown by another try block, except in the case of nested try blocks.
- When a try block is executed, there are three possible cases:
  1. **No exception is generated:**
    - All the statements in the try block are executed .
    - No catch block is executed.
    - Processing continues with the statement following all the catch blocks for the try block.
  2. **An exception is thrown and there is a matching catch block.**
    - The statements in the try block following the statement that caused the exception are NOT executed.
    - The first matching catch block is executed.
    - No other catch block is executed.
    - Processing continues with the statement following all the catch clauses for the try block.
  3. **An exception is thrown and there is no matching catch block.**
    - Control is immediately returned (propagated) to the method that called this method that caused the exception.
    - The propagation continues until the exception is caught, or until it is passed out of the Java program, where it is caught by the default handler in the Java-runtime System.
    - The default exception handler displays a string describing the exception.

# Recovering from Exceptions

- The code to recover from an exception is usually a loop that repeats as long as the condition that causes the exception has not been removed.
- Example:

```
import java.io.*;
public class TestException{
    public static void main(String[] args)throws IOException{
        BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));
        int number = 0;
        boolean done = false;
        do{
            try{ System.out.println("Enter an integer: ");
                number = Integer.parseInt(stdin.readLine().trim());
                done = true;
            }
            catch(NumberFormatException e){
                System.out.println("Error - Invalid input!");
            }
        }while(! done);
        System.out.println("The number entered is " + number);
    }
}
```

# Recovering from Exceptions (cont'd)

## ➤ Example:

```
import java.io.*;
class BankAccount{
    private int accountNumber;
    private double balance;
    public void deposit(double amount){
        if(amount < 0)
            throw new IllegalArgumentException("Negative deposit");
        else
            balance += amount;
    }
    //...
}
public class BankAccountTest{
    public static void main(String[] args)throws IOException{
        //...
        boolean done = false;
        do{
            System.out.println("Enter amount to deposit:");
            double amount = Double.parseDouble(stdin.readLine());
            try{
                account.deposit(amount);
                done = true;
            }
            catch(IllegalArgumentException e){System.out.println(e);}
        }while(! done);
    }
}
```

# Text File I/O

- I/O streams
- Opening a text file for reading
- Closing a stream
- Reading a text file
- Writing and appending to a text file

# I/O streams

- Files can be classified into two major categories: Binary files and Text files.
  - A binary file is a file whose contents must be handled as a sequence of binary digits.
  - A text file is a file whose contents are to be handled as a sequence of characters.
- Why use files for I/O?
  - Files provide permanent storage of data.
  - Files provide a convenient way to deal with large quantities of data.
- Recall that in In Java, I/O is handled by objects called streams.
- The standard streams are **System.in**, **System.out**, and **System.err**.

# Opening a Text File for Reading

- A stream of the class Scanner is created and connected to a text file for reading as follows:

```
Scanner input = new Scanner(new File(filename));
```

- Where filename is a File object or a constant string or a String variable containing the name or the full path of the file to be read.
  - Example of valid filenames: “myinput.txt”, “C:\\homework\\StudentTest.java”, “C:/homework/StudentTest.java”
- File class belong to the java.io package, it must be imported for it to be used.



# Closing a Stream

- When a program has finished writing to or reading from a file, it should close the stream connected to that file by calling the **close** method of the stream:

```
streamName.close();
```

```
Eg. input.close();
```

- The **close** method is defined as:

```
public void close()
```

- When a stream is closed, the system releases any resources used to connect the stream to a file.
- If a program does not close a stream before the program terminates, then the system will automatically close that stream.

# Reading a Text File

- After a stream has been connected to a text-file for reading, the `nextLine` or `next` methods of the stream can be used to read from the file:
  - `public String nextLine()`
  - `public int nextInt()`
- The `nextLine` method reads a line of input from the file and returns that line as a string.

# Reading a Text File: Example1

- The following program displays the contents of the file **myinput.txt** on the screen by reading one character at a time:

```
import java.io.*;

public class ShowFile{

    public static void main(String[] args)throws IOException{

        int input; Scanner  fin = null;

        try{

            fin = new Scanner(new File("myinput.txt"));

        }catch(FileNotFoundException e){

            System.out.println("Error - File myinput.txt not found");

            System.exit(1);

        }

        while(( input = fin.hasNext()) != -1)

            System.out.print((char) input);

        fin.close();

    }
}
```

# Reading a Text File: Example2

- The following program displays the ID, number of quizzes taken, and average of each student in grades.txt:

```
import java.io.*;import java.util.StringTokenizer;import java.util.Scanner;

public class QuizResults{

    public static void main(String[] args)throws IOException{

        Scanner  inputStream = new Scanner(new File("grades.txt"));

        StringTokenizer tokenizer; String inputLine,  id; int  count; double  sum;

        System.out.println("ID#    Number of Quizzes          Average\n");

        while(inputLine = inputStream.hasNext()){

            tokenizer = new StringTokenizer(inputLine);

            id = tokenizer.nextToken();

            count = tokenizer.countTokens();

            sum = 0.0;

            while(tokenizer.hasMoreTokens())

                sum += Double.parseDouble(tokenizer.nextToken( ));

            System.out.println(id + "    " + count + "    " + sum / count);

        }  }}
```

# Opening a Text File for Writing

- A stream is created and connected to a text file for writing by a statement of the form: .

```
PrintWriter streamName = new PrintWriter(new File(filename));
```

- Any preexisting file by the same name and in the same folder is destroyed. If the file does not exist it is created.
- Any preexisting file by the same name is not destroyed. If the file does not exist it is created.
- Both **PrintWriter** and **File** classes belong to **java.io** package.

# Writing to a Text File

- The `PrintWriter` class has methods `print` and `println`.
  - The `print` method prints output without generating a new line.
  - The `println` method prints output, it then generates a new line.
- Each constructor of the `FileWriter` can throw an `IOException`:
  - `public FileWriter(String filename) throws IOException`
  - `public FileWriter(String filename , boolean appendFlag) throws IOException`

# Example: Appending to a Text-file

- The following program appends a message to the file **datafile.txt**

```
import java.io.*;

public class FileAppend{

    public static void main(String[] args)throws IOException{

        String message = "Java is platform independent";

        PrintWriter outputStream =

            new PrintWriter(new FileWriter("datafile.txt", true));

        outputStream.println(message);

        outputStream.close();

    }

}
```

# Example: Writing to a Text File

- The following program copies the first 200 non-blank characters from one file to another.

```
import java.io.*;
public class FileCopy{
    public static void main(String[] args){
        int input;
        BufferedReader fin = null;
        PrintWriter fout = null;
        try{
            fin = new BufferedReader(new FileReader("myinput.txt"));
        }
        catch(FileNotFoundException e){
            System.out.println("Input File not found");
            System.exit(1);
        }
        try{
            fout = new PrintWriter(new FileWriter("myoutfile.txt"));
        }
        catch(IOException e){
            System.out.println("Error opening output file");
            System.exit(1);
        }
    }
}
```



# Example: Writing to a Text File (cont'd)

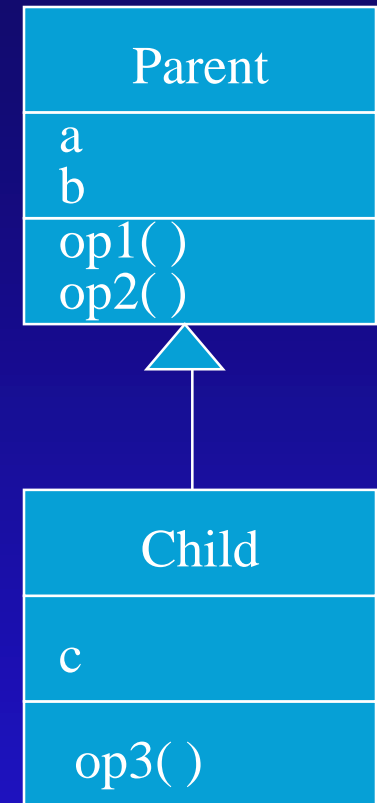
```
try{
    int count = 0;
    while((input = fin.read()) != -1 && count < 200){
        char ch = (char) input;
        if(ch != ' '){
            fout.print(ch);
            count++;
        }
    }
}
catch(IOException e){
    System.out.println("Error in reading the file myinput.txt");
}
try{
    fin.close();
    fout.close();
}
catch(IOException e){
    System.out.println("Error in closing a file");
}
System.out.println("File copied successfully");
}
}
```

# Introduction to Inheritance

- What is Inheritance?
- Superclass vs. Subclass
- Why Inheritance?
- Subclass Syntax
- Final Classes
- Class Hierarchy
- Subclass Constructors
- Example
- Exercises

# What is Inheritance?

- Inheritance is a mechanism for enhancing existing, working classes.
- A new class can inherit from a more general existing class.
- For Class Child:
  - Attributes:
    - Inherited: a , b
    - not inherited: c
  - Methods:
    - Inherited: op1( ) , op2( )
    - not inherited: op3( )
- Note: Constructors and private members of a class are not inherited by its subclasses.
- Java supports single inheritance: A subclass has one parent only.
- In multiple inheritance a subclass can have more than one parent.



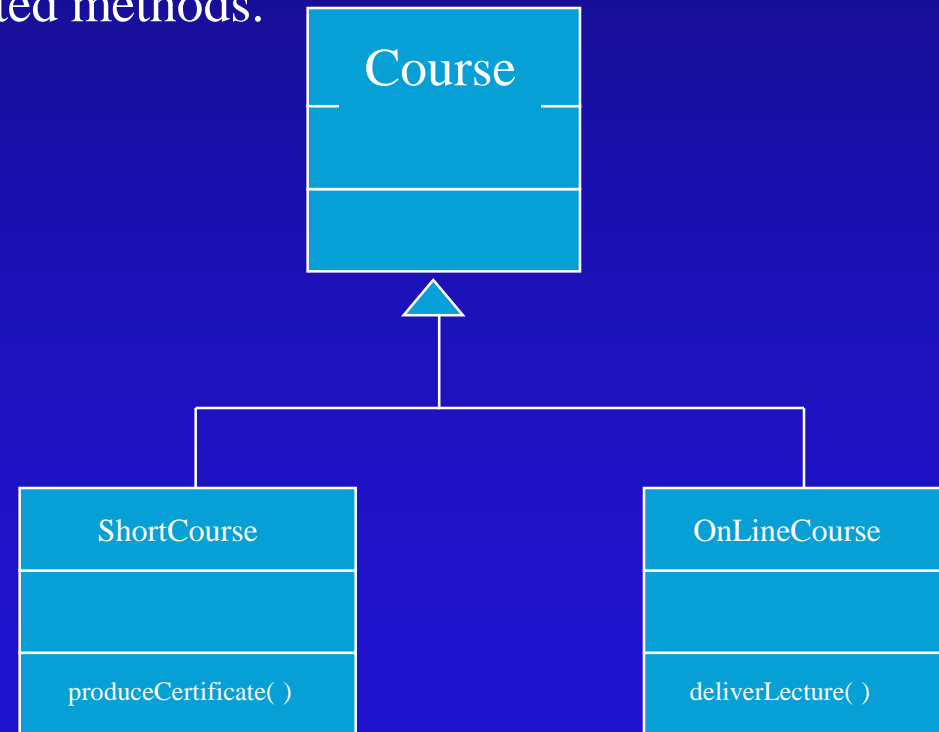
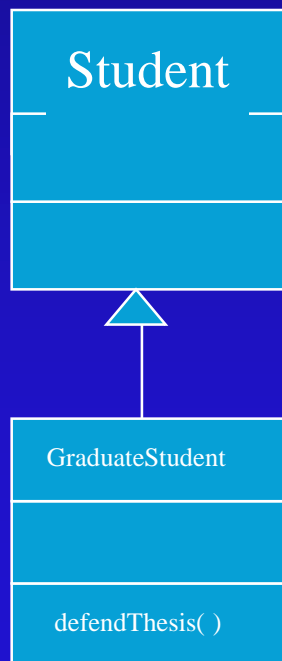
# Superclass vs. Subclass

## ➤ Superclass:

- A general class from which a more specialized class (a subclass) inherits.

## ➤ Subclass:

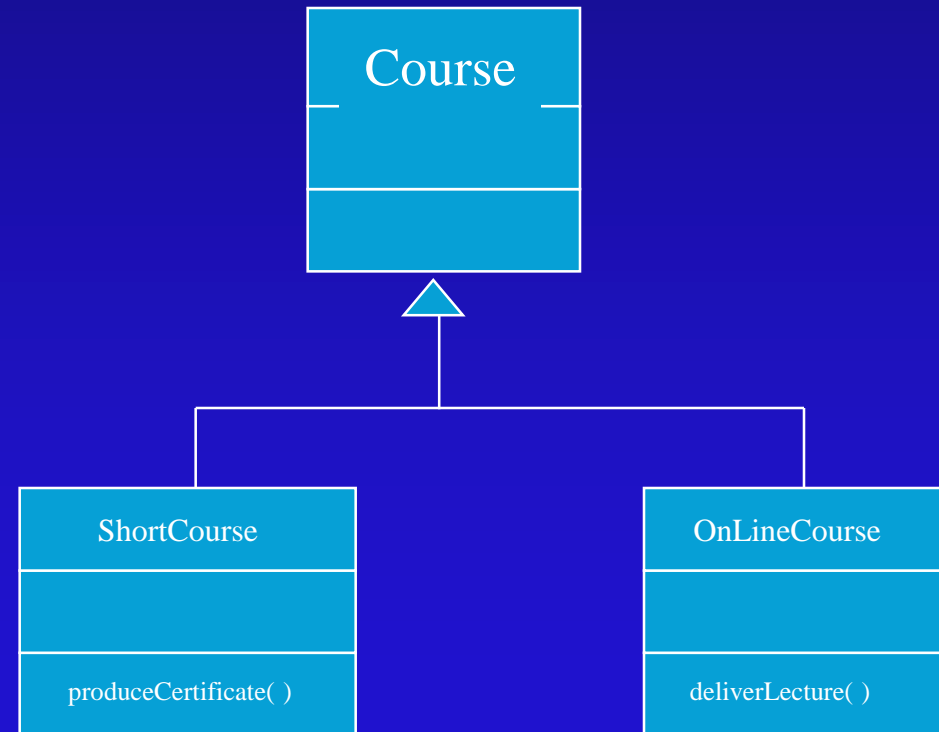
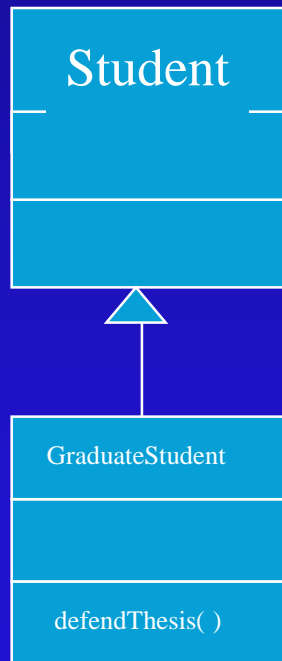
- A class that inherits variables and methods from a superclass but adds instance variables, adds methods, or redefines inherited methods.



# isA relationship

➤ A subclass-superclass pair defines “isA” relationship:

- A graduate student is a student.
- A short course is a course.
- An online course is a course.
- A graduate student is a student.



# Why Inheritance?

- Through inheritance we gain software reuse.
- Helps in writing structured programs.
- It is more natural and closer to real world.



# Subclass Syntax

```
class SubclassName extends SuperclassName
{
    variables
    methods
}
```

➤ Example:

```
class GraduateStudent extends Student
{
    String thesisTitle;
    . . .
    defendThesis() { . . . }
    . . .
}
```

# Final Classes

- A class that is declared as final cannot be extended or subclassed.

```
final class className
{
    variables
    methods
}
```

- Examples of final classes are:

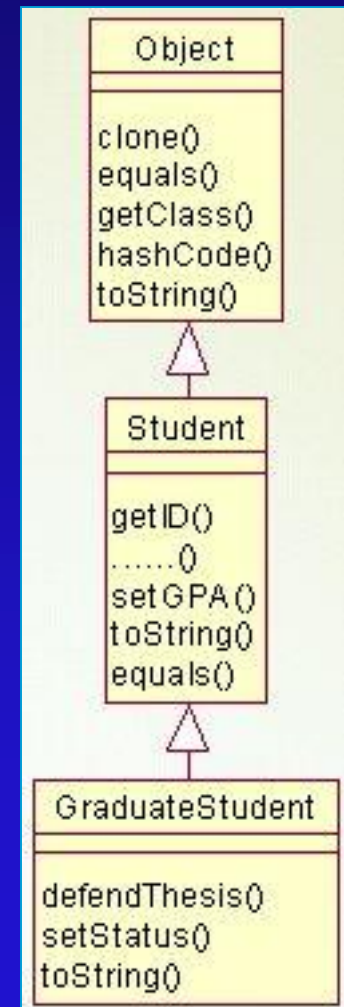
- java.lang.System.

- The primitive type wrapper classes: Byte, Character, Short, Integer, Long, Float, and Double.



# Class Hierarchy

- In Java, every class that does not extend another class is a subclass of the Object class. that is defined in the java.lang package.
- Thus, classes in Java form a hierarchy, with the Object class as the root.
- Example of a class hierarchy:



# Subclass Constructors

- To initialize instance variables of a superclass, a subclass constructor invokes a constructor of its superclass by a call of the form:

`super(parameters);`

- This statement must be the first statement within the constructor of the subclass.

- Example:

```
public GraduateStudent(int id, String name, double gpa, String thesisTitle){  
    super(id, name, gpa);  
    this.thesisTitle = thesisTitle;  
}
```

- If the first statement in a constructor does not explicitly invoke another constructor with `this` or `super`; Java implicitly inserts the call: `super()`; If the superclass does not have a no-argument constructor, this implicit invocation causes **compilation error**.
- Constructor calls are chained; any time an object is created, a sequence of constructors is invoked; from subclass to superclass on up to the Object class at the root of the class hierarchy.

# Example

```
class Vehicle{
    private String vehicleIdNumber ;
    private String vehicleChassisNumber ;
    private String model ;
    public Vehicle(String vin, String vcn, String model){
        vehicleIdNumber = vin;
        vehicleChassisNumber = vcn;
        this.model = model;
    }
    public String toString(){
        return "Vehicle ID = " + vehicleIdNumber +
            "\nVehicle Chassis Number = " +
                vehicleChassisNumber + "\nVehicle Model = " + model;
    }

    public boolean equals(Vehicle vehicle){
        return this.vehicleChassisNumber ==
            vehicle.vehicleChassisNumber;
    }
}
```

```
class Bus extends Vehicle{
    private int numberOfPassengers ;
    public Bus(int numPassengers, String vin, String vcn, String model){
        super(vin, vcn, model) ;
        numberOfPassengers = numPassengers ;
    }

    public int getNumberOfPassengers(){
        return numberOfPassengers ;
    }
}
```

# Example (cont'd)

```
class Truck extends Vehicle{
    private double cargoWeightLimit ;
    public Truck(double weightLimit, String vin, String vcn, String model){
        super(vin, vcn, model) ;
        cargoWeightLimit = weightLimit ;
    }

    public double getCargoWeightLimit(){
        return cargoWeightLimit ;
    }
}
```

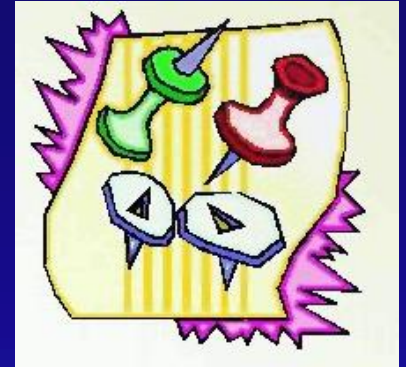
```
public class VehicleTest{
    public static void main(String[] args){
        Vehicle vehicle = new
            Vehicle ("QMY 489", "MX-0054322-KJ", "BMW 500");
        Bus bus1 = new Bus(30, "TMK 321", "AF-987654-WR",
            "MERCEDEZ BENZ");
        Bus bus2 = new Bus(30, "2348976", "AF-987654-WR",
            "MERCEDEZ BENZ");
        Truck truck = new
            Truck(10.0, "DBS 750", "RZ-70002345-PN", "ISUZU");

        System.out.println(vehicle) ;
        System.out.println(bus1) ;
        System.out.println(truck) ;
        if(bus1.equals(bus2))
            System.out.println("Bus1 and Bus2 are the same") ;
        else
            System.out.println("Bus1 and Bus2 are not the same") ;
    }
}
```

# Exercises

Question 1:

What inheritance relationships would you establish among the following classes:  
Student, Professor, Teaching Assistant, Employee, Secretary,  
DepartmentChairman, Janitor, Person, Course, Seminar, Lecture, ComputerLab



Question 2:

In the following pairs of classes, identify  
the superclass and the subclass: Manager -Employee,  
Polygon -Triangle, Person - Student, Vehicle - Car, Computer-Laptop,  
Orange - Fruit.



# Introduction to Inheritance

- Access Modifiers
- Methods in Subclasses
- Method Overriding
- Converting Class Types
- Why up-cast?
- Why Down-cast?
- Exercises

# Access Modifiers

weaker access  
privilege



stronger access  
privilege

Access Modifier	Members with the access modifier can be accessed directly by:
private	<ul style="list-style-type: none"><li>• Methods of their own class only.</li></ul>
protected	<ul style="list-style-type: none"><li>• Methods of their own class.</li><li>• Methods of subclasses and other classes in the same package.</li><li>• Methods of subclasses in different packages.</li></ul>
No access modifier (default or package access)	<ul style="list-style-type: none"><li>• Methods of their own class.</li><li>• Methods of subclasses and other classes in the same package.</li></ul>
public	<ul style="list-style-type: none"><li>• Methods of all classes in all packages.</li></ul>

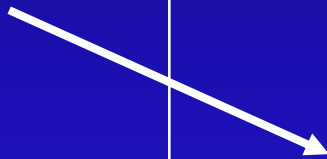
# Methods in Subclasses

- A method in a subclass can be:
  - A new method defined by the subclass.
  - A method inherited from a superclass.
  - A method that redefines (i.e., overrides) a superclass method.

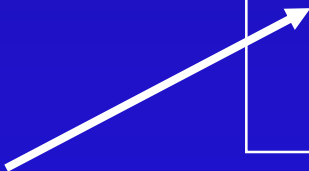


```
public class GraduateStudent extends Student{  
    private String thesisTitle ;  
    public GraduateStudent(. . .){ . . .}  
    public String getThesisTitle() {return thesisTitle; }  
    public String toString( ){ . . .}  
    . . .  
}
```

New method



Overridden  
method



Inherited methods: `getID( )`, `getName( )`, `getGPA( )`, `setGPA( )`, etc.



# Method Overriding

- A subclass overrides a method it inherits from its superclass if it defines a method with the same name, same return type, and same parameters as the superclass method.
- Each of the following causes **compilation error**:
  - The overriding method has a weaker access privilege than the method it is overriding.
  - The overriding method has the same signature but different return type to the method it is overriding.
- A method declared as final, **final returnType methodName( . . . )**, cannot be overridden.
- A subclass can access an overridden method can by a call of the form:  

```
super.methodName(parameters) ;
```
- Note: We have already come across examples of method overriding when we discussed the redefinition of the **toString** and the **equals** methods in the previous lectures.

# Method Overriding Example1

```
public class Student{  
    private int id; private String name; private double gpa;  
    . . .  
    public String toString(){return "ID: " + id + ",Name: " + name  
        + ", GPA: " + gpa; }  
}
```

```
public class GraduateStudent extends Student{  
    String thesisTitle;  
    . . .  
    public String toString(){  
        return super.toString() + ", Thesis Title:" + thesisTitle;  
    }  
}
```

# Method Overriding Example2

```
public class Student{  
    protected int id; protected String name; protected double gpa;  
    . . .  
    public String toString(){return "ID: " + id + ",Name: " + name  
        + ", GPA: " + gpa;  
    }  
}
```

```
public class GraduateStudent extends Student{  
    String thesisTitle;  
    . . .  
    public String toString(){  
        return "ID: " + id + ",Name: " + name  
            + ", GPA: " + gpa + ", Thesis Title: " + thesisTitle;  
    }  
}
```

# Converting between Class Types

- A widening conversion (up casting) can be done without a class cast:

```
GraduateStudent gradstudent = new GraduateStudent(...);  
Student student = gradstudent;
```

- However, the reference **student** cannot access members of GraduateStudent even if they are public:

```
String title = student.getThesisTitle(); // Compile time error
```

- A narrowing conversion (down casting) requires a class cast; otherwise a compile time error occurs:

```
Object object = new Object();  
String string = object ; // Compile time error
```

# Converting between Class Types (cont'd)

- A narrowing conversion in which a superclass reference does not refer to a subclass does not cause a compilation error; but it causes a runtime error:

java.lang.ClassCastException:

```
Object object = new Object();  
String string = (String) object;
```

- A narrowing conversion is valid, if the superclass reference refers to a subclass:

```
Object object = new Object();  
String string1 = "KFUPM";  
object = string1;  
String string2 = (String) object;  
int j = ((String) object).length();
```

# Why Up-cast?

- One use of up-casting is to write methods that are general.
- Example:

```
class Vehicle{
    private String vehicleIdNumber ;
    private String vehicleChassisNumber ;
    private String model ;
    // . . .
    public boolean equals(Vehicle vehicle){
        return this.vehicleChassisNumber ==
            vehicle.vehicleChassisNumber;
    }
}

class Bus extends Vehicle{
    private int numberOfPassengers ;
    // . . .
}
```

```
class Truck extends Vehicle{
    private double cargoWeightLimit ;
    // . . .
}

public class VehicleTest{
    public static void main(String[] args){
        // . . .
        boolean x = vehicle1.equals(vehicle2);
        boolean y = bus1.equals(bus2);
        boolean z = truck1.equals(truck2);
        // . . .
    }
}
```

# Why down-cast?

## ➤ Example:

```
public class Employee {  
    ...  
}  
  
public class Manager extends Employee {  
    public getDepartmentName() { . . . }  
    ...  
}
```

```
Employee e = new Manager(. . .);  
String string = e.getDepartmentName();
```

compile time error. **getDepartmentName()** is not a member of Employee.

Use down-casting to overcome the difficulty.

```
Manager m = (Manager)e;  
String string = m.getDepartmentName();
```

# Exercises

## 1. Question 1:

- a) Write a class `Employee` with a name and salary.
- b) Make a class `Manager` inherit from `Employee`. Add an instance variable, named `department`, of type `String`. Supply a method `toString` that prints the manager's name, department, and salary.
- c) Make a class `Executive` inherit from `Manager`. Supply a method `toString` that prints the string `"Executive"`, followed by the information stored in the `Manager` superclass object.
- d) Supply a test program that tests these classes and methods.

## 2. Question 2:

- a) Implement a superclass `Person`. A person has a name and a year of birth.
- b) Make two classes, `Student` and `Instructor`, inherit from `Person`. A student has a major, and an instructor has a salary. Write the class definitions, the constructors, and the method `toString` for all classes.
- c) Supply a test program that tests these classes and methods.