KING KHALID UNIVERSITY

Applied College

Web and Mobile Application Development Program

1331-CSA

Introduction to Computer Programming

Prepared by Ahmed Asiri

Chapter 1

Introduction

Programming Languages

Programming languages are broadly classified into three levels: machine languages, assembly languages, and high-level languages.

Machine language is the only programming language the CPU understands. Each type of CPU has its own machine language. Machine-language instructions are binary-coded and very low level, binary codes (0 and 1)—one machine instruction may transfer the contents of one memory location into a CPU register or add numbers in two registers. Thus, we must provide many machine-language instructions to accomplish a simple task.

One level above machine language is ***assembly language***, which allows "higher-level" symbolic programming. Instead of writing programs as a sequence of bits, assembly language allows programmers to write programs by using symbolic operation codes. For example, instead of 10110011, we use MV to move the contents of a memory cell into a register.

High-level languages were developed to enable programmers to write programs faster than when using assembly languages. For example, **JAVA**; a programming language intended for mathematical computation, allows programmers to express numerical equations directly as

```
X = (Y + Z) / 2
```

Definitions

- **program**: A set of instructions that are to be carried out by a computer.

- **program execution**: The act of carrying out the instructions contained in a program.
  - this is done by feeding the instructions to the CPU

- **programming language**: A systematic set of rules used to describe computations, generally in a format that is editable by humans.

Java

- There are hundreds of high level computer languages. Java, C++, C, Basic, Fortran, Cobol, Lisp, Perl, Prolog, Eiffel, Python
- The capabilities of the languages vary widely, but they all need a way to do
  - declarative statements
  - conditional statements
  - iterative or repetitive statements
- A compiler is a program that converts commands in high level languages to machine language instructions

Java is a new object-oriented language that is receiving wide attention from both industry and academia. The language was based on C and C++ and was originally intended for writing programs that control consumer appliances such as toasters, microwave ovens, and others. Java is often described as a *Web programming language* because of its use in writing programs.

A Simple Java Program

```
public class Hello
{   public static void main(String[] args)
   {   System.out.println("Hello World!");
   }
}
```

More Definitions

- **code or source code**: The sequence of instructions in a particular program.
  - The code in this program instructs the computer to print a message of **Hello, world!** on the screen.
- **output**: The messages printed to the computer user by a program.
- **console**: The text box or window onto which output is printed.

Compiling and Running

- **Compiler**: a program that converts a program in one language to another language
  - compile from C++ to machine code
  - compile Java to *bytecode*
- **Bytecode**: a language for an imaginary cpu
- **Interpreter**: A converts one instruction or line of code from one language to another and then executes that instruction
  - When java programs are run the bytecode produced by the compiler is fed to an interpreter that converts it to machine code for a particular CPU
  - on my machine it converts it to instructions for a Pentium cpu

Structure of Java programs

```
public class <name> {
    public static void main(String[] args) {
        <statement(s)>;
    }
}
```

- Every executable Java program consists of a **class**...
  - that contains a **method** named `main`...
    - that contains the **statements** to be executed
- The previous program is a class named `Hello`, whose `main` method executes one statement named `System.out.println`

Static methods

- **static method**: A group of statements that is given a name so that it can be executed in our program.
  - Breaking down a problem into static methods is also called "procedural decomposition."
- Using a static method requires two steps:
  - **declare** it (write down the recipe)
    - When we declare a static method, we write a group of statements and give it a name.
  - **call** it (cook using the recipe)
    - When we call a static method, we tell our main method to execute the statements in that static method.

- Static methods are useful for:
  - denoting the structure of a larger program in smaller, more understandable pieces
  - eliminating redundancy through reuse

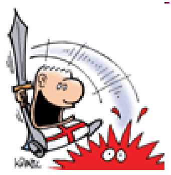
Static method syntax

- The structure of a static method:

```
public class <Class Name> {

    public static void <Method name> () {
        <statements>;
    }

}
```

- Example:

```
public static void printCheer() {
    System.out.println("Three cheers for Pirates!");
    System.out.println("Huzzah!");
    System.out.println("Huzzah!");
    System.out.println("Huzzah!");
}
```



Static methods example

```
public class TwoMessages {
    public static void main(String[] args) {
        printCheer();
        System.out.println();
        printCheer();
    }

    public static void printCheer() {
        System.out.println("Three cheers for Pirates!");
        System.out.println("Huzzah!");
        System.out.println("Huzzah!");
        System.out.println("Huzzah!");
    }
}
```

Program's output:

```
Three cheers for Pirates!
Huzzah!
Huzzah!
Huzzah!

Three cheers for Pirates!
Huzzah!
Huzzah!
Huzzah!
```

Methods calling each other

- One static method may call another:

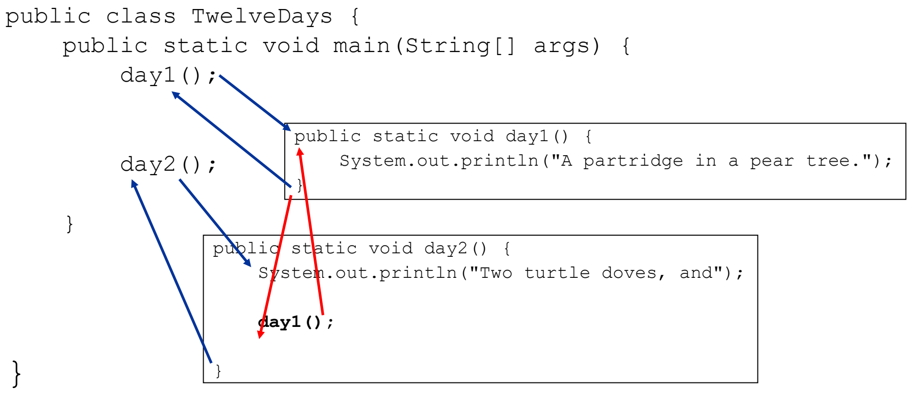
```
public class TwelveDays {
    public static void main(String[] args) {
        day1();
        day2();
    }

    public static void day1() {
         System.out.println("A partridge in a pear tree.");
    }

    public static void day2() {
         System.out.println("Two turtle doves, and");
         day1();
    }
}
```

Program's output:

```
A partridge in a pear tree.
Two turtle doves, and
A partridge in a pear tree.
```

Control flow of methods

- When a method is called, a Java program 'jumps' into that method, executes all of its statements, and then 'jumps' back to where it started.



```
public class TwelveDays {
    public static void main(String[] args) {
        day1();

        day2();

    }

}
```

```
public static void day1() {
    System.out.println("A partridge in a pear tree.");
}
```

```
public static void day2() {
    System.out.println("Two turtle doves, and");

    day1();

}
```

Identifiers

- **identifier**: A name that we give to a piece of data or part of a program.
  - Identifiers are useful because they allow us to refer to that data or code later in the program.
  - Identifiers give names to:
    - classes
    - methods
    - variables (named pieces of data; seen later)

- The name you give to a static method is an example of an identifier.
  - What are some other example identifier we've seen?

Details about identifiers

- Java identifier names:
  - first character must a letter or _ or `$`
  - following characters can be any of those characters or a number
  - identifiers are case-sensitive; `name` is different from `Name`

- Example Java identifiers:
  - legal:

```
olivia          second_place     _myName
      TheCure         ANSWER_IS_42     $variable
```

  - illegal:

```
      me+u            :-)              question?
      side-swipe      hi there         ph.d
      belles's        2%milk
kelly@yahoo.com
```

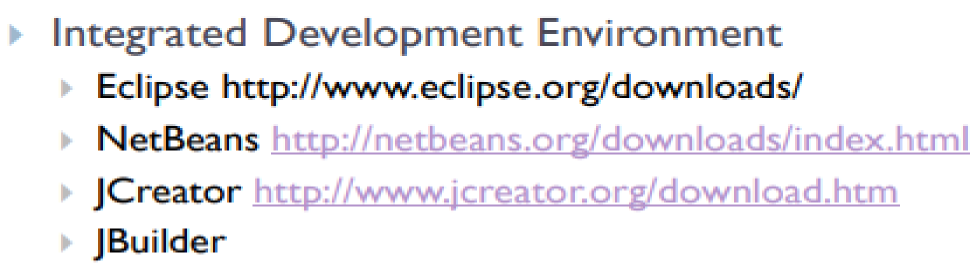
Variable Example

```
class Example2 {

public static void main(String args[]) {

int var1; // this declares a variable
int var2; // this declares another variable

var1 = 1024; // this assigns 1024 to var1

System.out.println("var1 contains " + var1);

var2 = var1 / 2;

System.out.print("var2 contains var1 / 2: ");

System.out.println(var2);

}

}
```

Predicted Output:

```
var2 contains var1 / 2: 512
```

The above program uses two variables, var1 and var2. var1 is assigned a value directly while var2 is filled up with the result of dividing var1 by 2, i.e. var2 = var1/2. The words int refer to a particular data type, i.e. integer (whole numbers).



- Integrated Development Environment
  - Eclipse http://www.eclipse.org/downloads/
  - NetBeans http://netbeans.org/downloads/index.html
  - JCreator http://www.jcreator.org/download.htm
  - JBuilder

In order to make a running program, the developed class must contain a particular method:

the "**main**" method is the entry point into the program: the microprocessor knows it will start executing instructions from this place.

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

Comments

In addition to the instructions for computers to follow, programs contain ***comments*** in which we state the purpose of the program, explain the meaning of code, and provide.

Any other descriptions to help programmers understand the program. Here's the comment in the sample the following program:

```
/*
    Chapter 2 Sample Program: Displaying a Window

    File: Ch2Sample1.java
*/

import javax.swing.*;

class Ch2Sample1 {

public static void main(String[] args)

{ JFrame myWindow;

myWindow = new JFrame();

myWindow.setSize(300, 200);
myWindow.setTitle("My First Java Program"); myWindow.setVisible(true);
}
}

/*
This is a comment with
three lines of
text.
```

```
*/

// This is a comment
// This is another comment
// This is a third comment

/* This is a comment on one line */

/*
 Comment number 1
*/
/*
 Comment number 2
*/
```

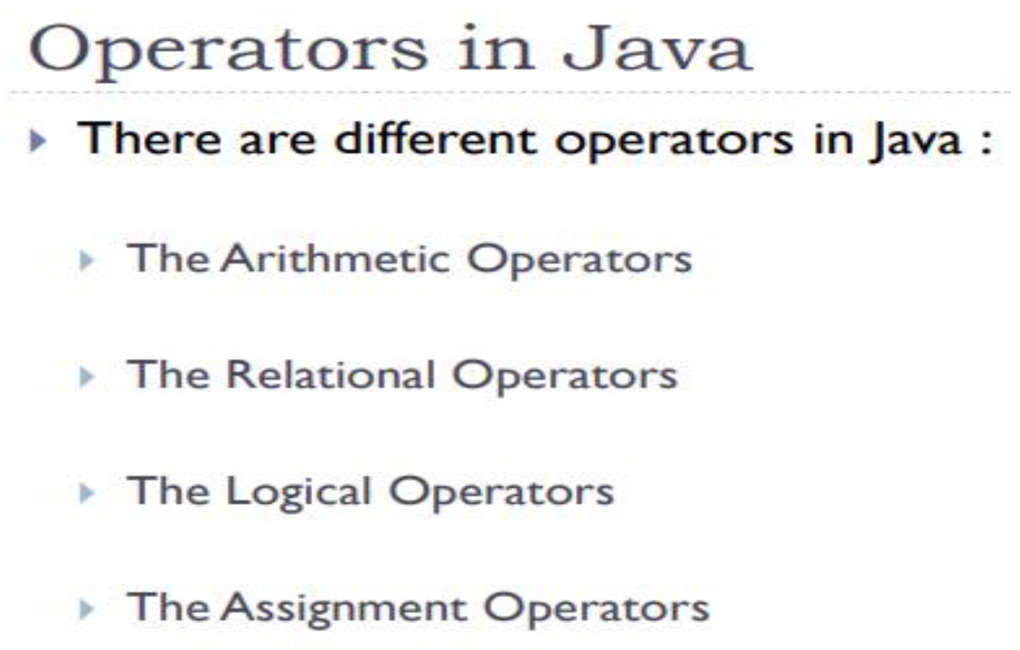
Java - Variable Types

A variable provides us with named storage that our programs can manipulate. Each variable in Java has a specific type.
Following is the basic form of a variable declaration:

```
data type variable [ = value][, variable [ = value] ...] ;
```

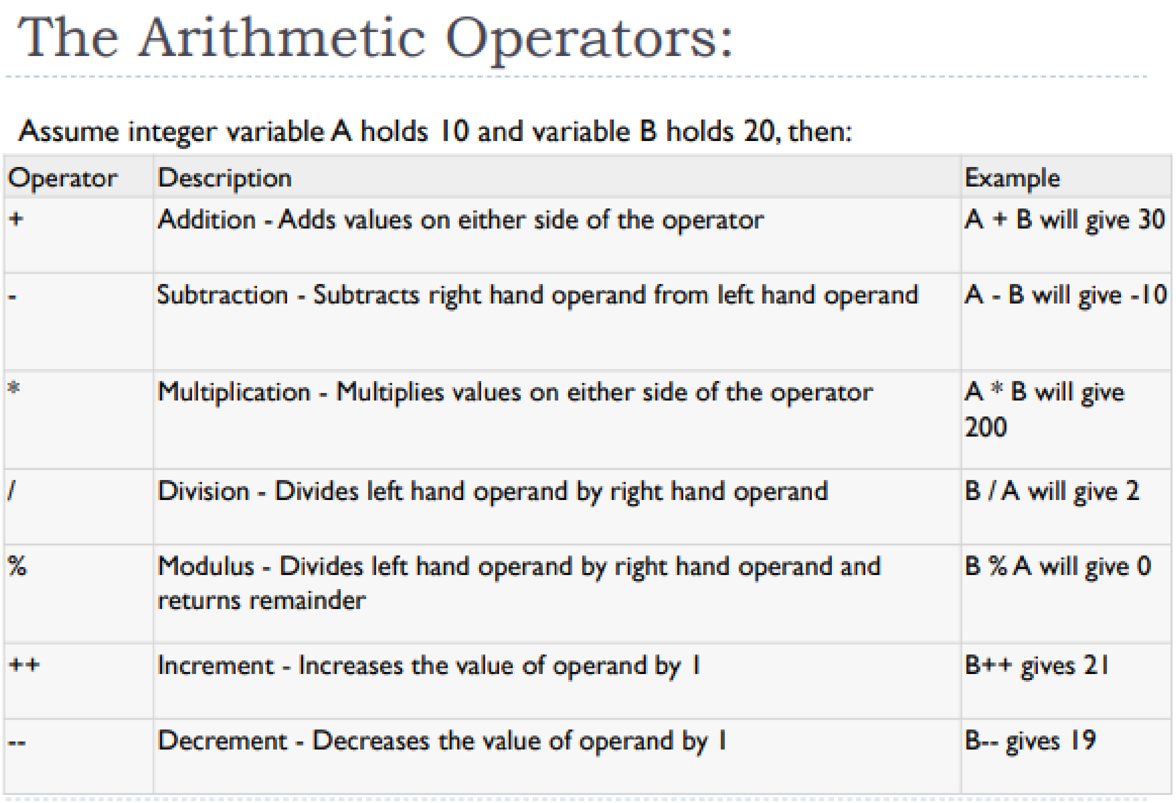
Example:

```
int a, b, c;            // Declares three integer, a, b, and c.
int a = 10, b = 10;     // Example of initialization
byte B = 22;            // initializes a byte type variable B.
double pi = 3.14159;    // declares and assigns a value of PI.
char a = 'a';       // the char variable a is initialized with value 'a'
```



Operators in Java

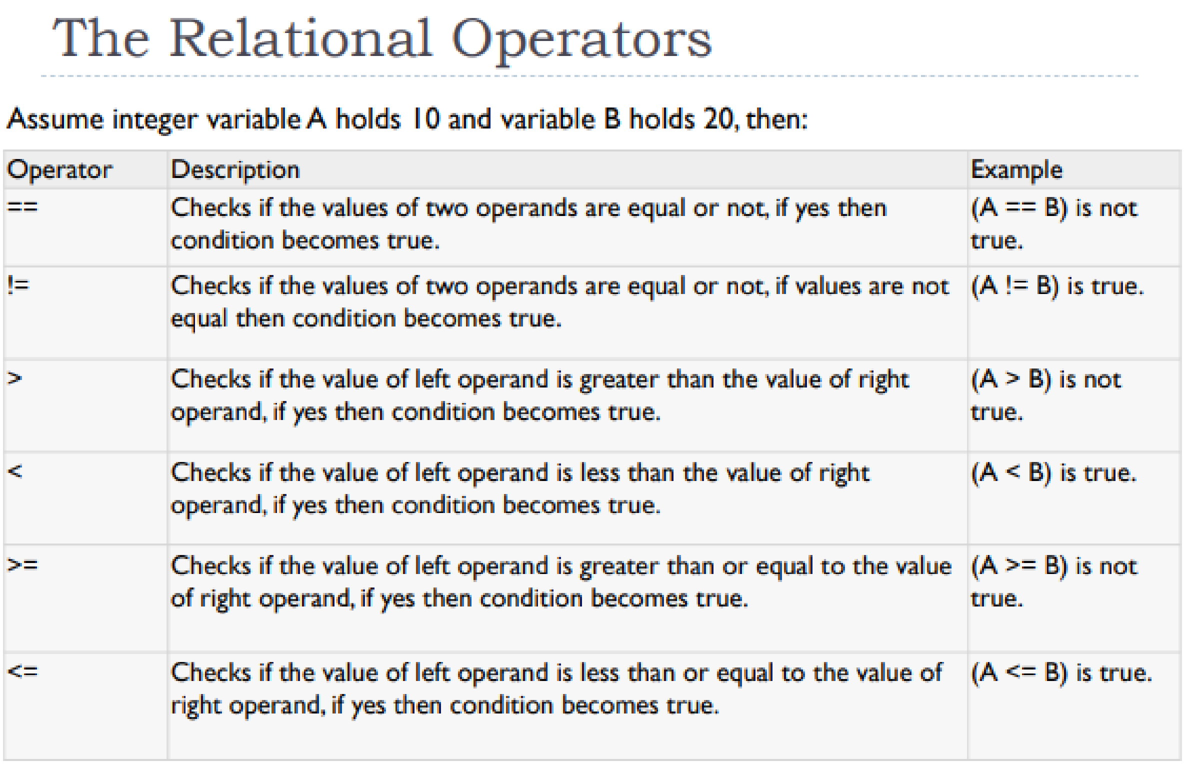
- There are different operators in Java :
  - The Arithmetic Operators
  - The Relational Operators
  - The Logical Operators
  - The Assignment Operators



The Arithmetic Operators:

Assume integer variable A holds 10 and variable B holds 20, then:

Operator	Description	Example
+	Addition - Adds values on either side of the operator	A + B will give 30
-	Subtraction - Subtracts right hand operand from left hand operand	A - B will give -10
*	Multiplication - Multiplies values on either side of the operator	A * B will give 200
/	Division - Divides left hand operand by right hand operand	B / A will give 2
%	Modulus - Divides left hand operand by right hand operand and returns remainder	B % A will give 0
++	Increment - Increases the value of operand by 1	B++ gives 21
--	Decrement - Decreases the value of operand by 1	B-- gives 19



The Relational Operators

Assume integer variable A holds 10 and variable B holds 20, then:

Operator	Description	Example
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

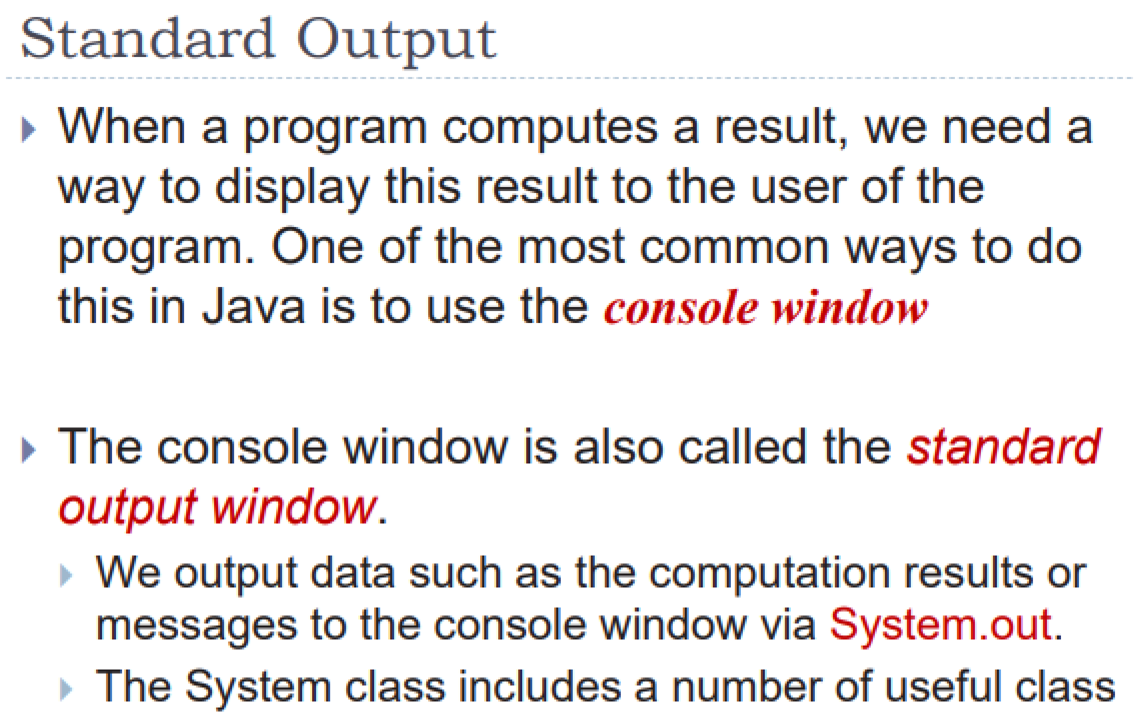
The Logical Operators

Assume Boolean variables A holds true and variable B holds false, then:

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands are non-zero, then the condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true.

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	C = A + B will assign value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator	C ^= 2 is same as C = C ^ 2
|=	bitwise inclusive OR and assignment operator	C |= 2 is same as C = C | 2

❖ **Standard input and output in Java**



Standard Output

- When a program computes a result, we need a way to display this result to the user of the program. One of the most common ways to do this in Java is to use the ***console window***
- The console window is also called the *standard output window*.
  - We output data such as the computation results or messages to the console window via System.out.
  - The System class includes a number of useful class

The print method is used to output a value.

Example:

```
System.out.print("King Khalid University");
```

Output: King Khalid University

Print Vs Println

- The ***println*** method will skip to the next line after printing out its argument.

Code

```
System.out.print("How do you do? ");
System.out.print("My name is ");
System.out.print("Seattle Slew.");
```

Output

```
How do you do? My name is Seattle Slew.
```

Code

```
System.out.println("How do you do? ");
System.out.println("My name is ");
System.out.println("Seattle Slew.");
```

Output

```
How do you do?
My name is
Seattle Slew.
```

Example

- Example1 :

```
int num = 15;

System.out.print(num);  //print a variable
System.out.print(" ");  //print a blank space
System.out.print(10);   //print a constant
```

Executing the code will result in the following console window:

Console Window

```
15 10
```

- Example2

```
int num = 15;
System.out.println(num);
System.out.println(10);
```

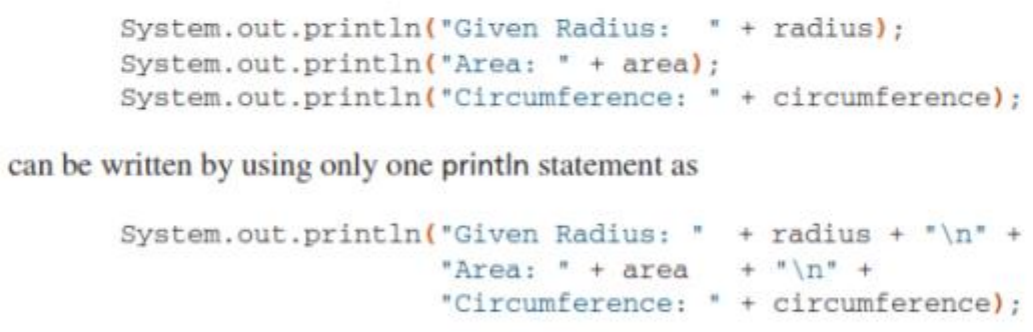
will result in

Console Window

```
15
10
```

Example

- Example 4 : new-line control character



```
System.out.println("Given Radius:  " + radius);
System.out.println("Area: " + area);
System.out.println("Circumference: " + circumference);
```

can be written by using only one println statement as

```
System.out.println("Given Radius: "  + radius + "\n" +
                   "Area: " + area   + "\n" +
                   "Circumference: " + circumference);
```

Standard Input

- Analogous to System.out for output, we have System.in for input. We call the technique to input data using System.in standard input.

- System.in is an instance of the InputStream class.

- The Scanner class from the java.util package provides a necessary input facility to accommodate various input routines.

Example

```java
Scanner scanner = new Scanner(System.in);

String firstName, lastName;

System.out.print("Enter your first name: ");
firstName = scanner.next( );

System.out.print("Enter your last name: ");
lastName = scanner.next( );

System.out.println("Your name is " + firstName +
                   "" + lastName + ".");
```

```
Enter your first name: George ENTER
Enter your last name: Washington ENTER
Your name is George Washington.
```

Example: int

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

int num1, num2;

System.out.print("Enter two integers: ");

num1 = scanner.nextInt( );
num2 = scanner.nextInt( );

System.out.print("num1 = " + num1 + " num2 = " + num2);
```

And here's a sample interaction:

Space separates the two input values.

```
Enter two integers: 12 8 ENTER
num1 = 12 and num2 = 87
```

Since the new-line character (when we press the Enter key, this new-line character is entered into the system) is also a white space, we can enter the two integers by pressing the Enter key after each number. Here's a sample:

```
Enter two integers:  12 ENTER
87 ENTER
num1 = 12 and num2 = 87
```

```
import java.util.*;

class Ch5Sample1 {

    public static void main(String[] args) {

        double    radius, circumference, area;

        Ch5Circle circle;

        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter radius: ");
        radius = scanner.nextDouble();

        circle = new Ch5Circle(radius);

        circumference = circle.getCircumference();

        area          = circle.getArea();

        System.out.println("Input radius:  " + radius);
        System.out.println("Circumference: " + circumference);
        System.out.println("Area:          " + area);
    }
}
```

Chapter 2

Control Structure (Conditional/selection and looping Statements)

Following is the syntax of the if statement.

if (condition)

{

// code

}

And here is a small code snippet which displays the message "Hi" only if the value of the boolean variable “**wish**” is true.

if (wish == true)

{

System.out.println("Hi");

}

Following is the syntax of if else structure.

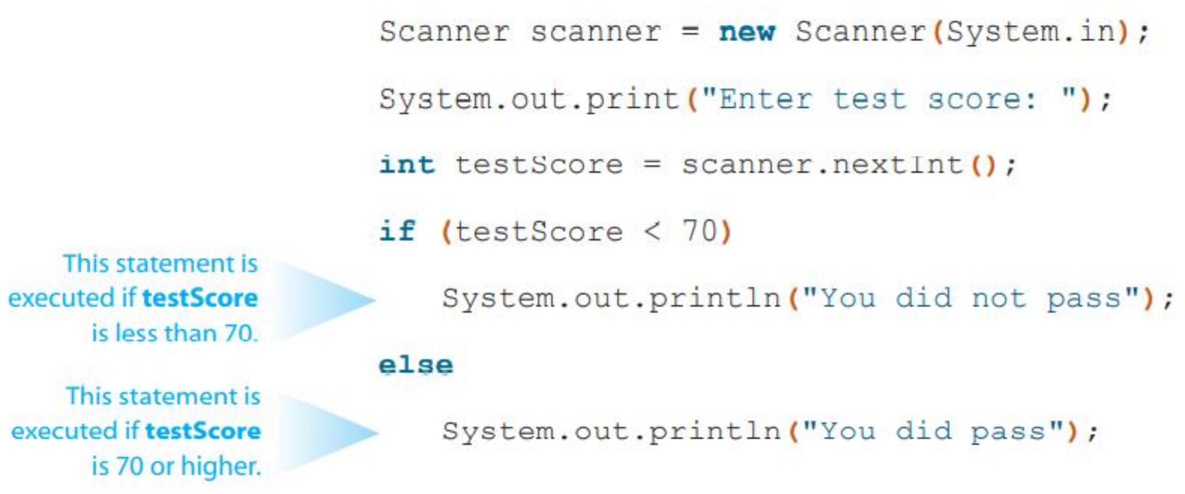
```
if ( condition )

{

// code

}

else

{

// code

 }
```

And here is an example usage which states whether a number is an **even number** or an **odd number**. Even numbers are divisible by zero and hence leave a remainder of zero. The remainder on dividing the number by two can be obtained using the modulo (%) mathematical operator.

```
if ( num%2==0)

{

System.out.println("Number is even");

}

Else

{

System.out.println("Number is odd");

}
```

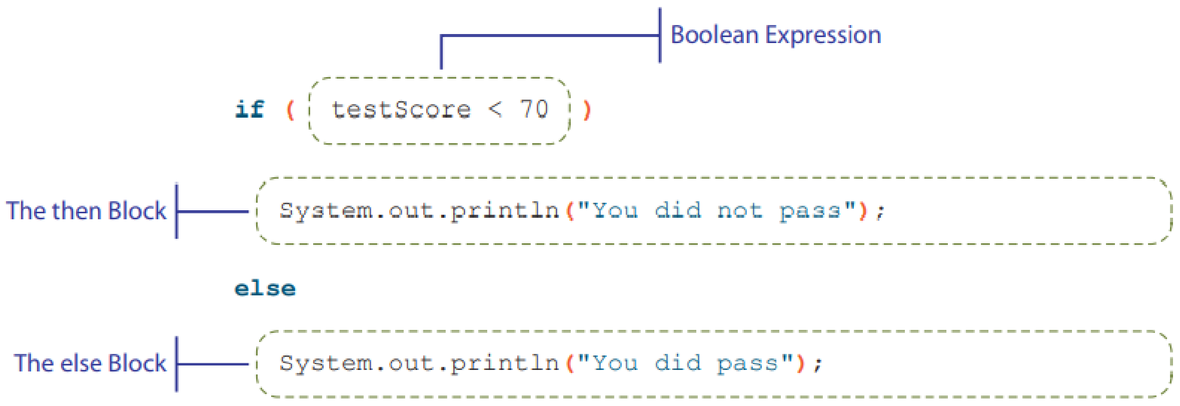
If statement:

There are two versions of the if statement, called if–then–else and if–then. We begin with the first version. Suppose we wish to enter a student's test score and print out the message You did not pass if the score is less than 70 and You did pass if the score is 70 or higher. Here's how we express this logic in Java:



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

System.out.print("Enter test score: ");

int testScore = scanner.nextInt();

if (testScore < 70)

    System.out.println("You did not pass");

else

    System.out.println("You did pass");
```

This statement is executed if **testScore** is less than 70.
This statement is executed if **testScore** is 70 or higher.



```
if ( testScore < 70 )

    System.out.println("You did not pass");

else

    System.out.println("You did pass");
```

Mapping of the sample if–then–else statement to the general format:

boolean expression

result of evaluating a test condition, called a ***boolean expression***. The if–then–else statement follows this general format:

if-then-else syntax

```
if ( <boolean expression> )
   <then block>

else
   <else block>
```

The <boolean expression> is a conditional expression that is evaluated to either true or false. For example, the following three expressions are all conditional:

```
testScore < 80
testScore * 2 > 350
30 < w / (h * h)
```

relational operators

The six ***relational operators*** we can use in conditional expressions are:

```
<       // less than
<=      // less than or equal to
==      // equal to
!=      // not equal to
>       // greater than
>=      // greater than or equal to
```

Here are some more examples:

```
a * a <= c      //true if a * a is less than or equal to c
x + y != z      //true if x + y is not equal to z
a == b          //true if a is equal to b
```

If multiple statements are needed in the <then block> or the <else block>, they must be surrounded by braces { and }. For example, suppose we want to print out additional messages for each case. Let's say we also want to print Keep up the good work when the student passes and print Try harder next time when the student does not pass. Here's how:

```
if (testScore < 70)
{
   System.out.println("You did not pass");
   System.out.println("Try harder next time");
}
else
{
   System.out.println("You did pass");
   System.out.println("Keep up the good work");
}
```

Compound Statements

The braces are necessary to delineate the statements inside the block. Without the braces, the compiler will not be able to tell whether a statement is part of the block or part of the statement that follows the if statement.

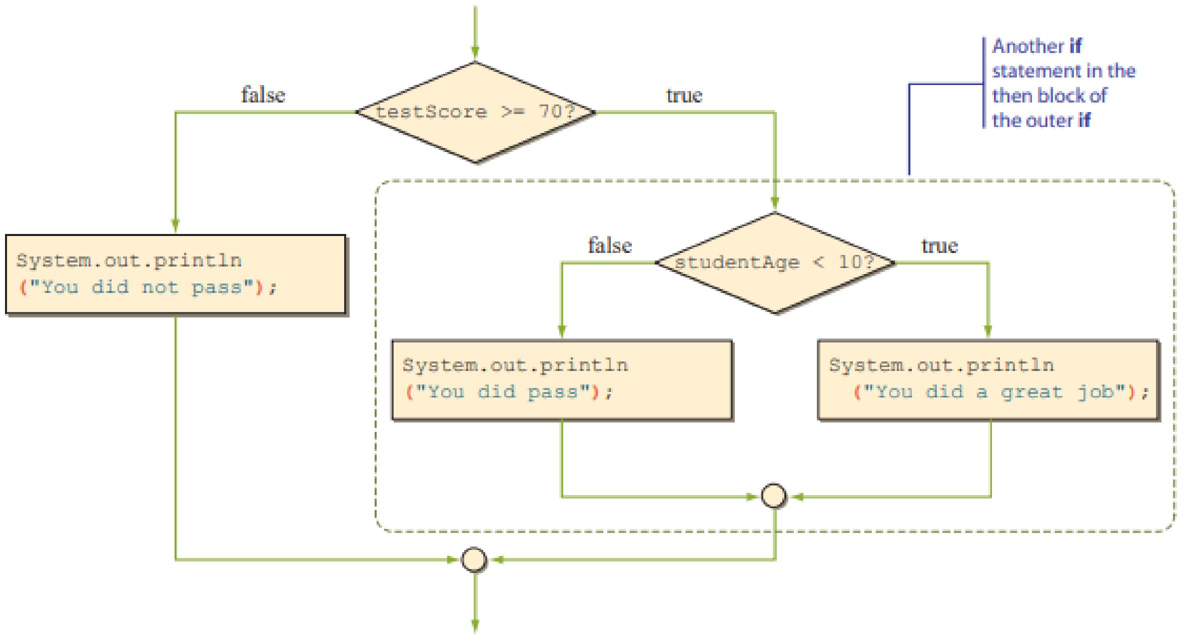
Notice the absence of semicolons after the right braces. A semicolon is never necessary immediately after a right brace. A compound statement may contain zero

Nested if statements

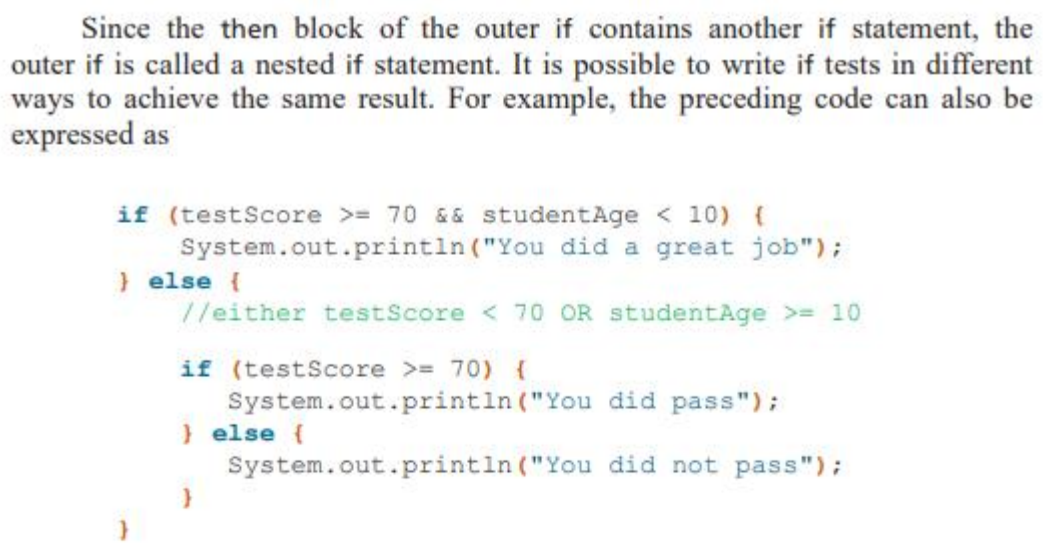
nested **if** statement

The then and else blocks of an if statement can contain any statement including another if statement. An if statement that contains another if statement in either its then or else block is called a *nested if* statement. Let's look at an example. In the earlier example, we printed out the messages You did pass or You did not pass depending on the test score. Let's modify the code to print out three possible messages. If the test score is lower than 70, then we print You did not pass, as before. If the test score is 70 or higher, then we will check the student's age. If the age is less than 10, we will print You did a great job. Otherwise, we will print You did pass, as before. Figure 5.4 is a diagram showing the logic of this nested test. The code is written as follows:

```
if (testScore >= 70) {
   if (studentAge < 10) {
      System.out.println("You did a great job");
   } else {

      System.out.println("You did pass");//test score >= 70
   }                                     //and age >= 10
} else { //test score < 70

   System.out.println("You did not pass");
}
```


false
testScore >= 70?
true
Another if statement in the then block of the outer if
System.out.println ("You did not pass");
false
studentAge < 10?
true
System.out.println ("You did pass");
System.out.println ("You did a great job");

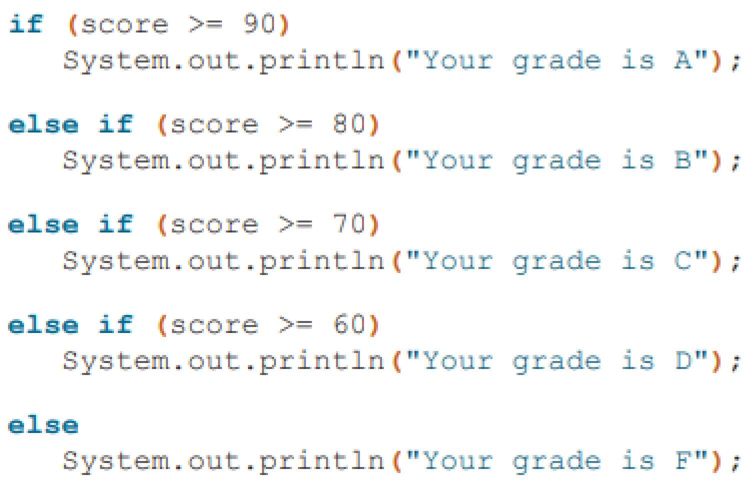
A diagram showing the control flow of the example nested **if** statement.



Since the then block of the outer if contains another if statement, the outer if is called a nested if statement. It is possible to write if tests in different ways to achieve the same result. For example, the preceding code can also be expressed as

```
if (testScore >= 70 && studentAge < 10) {
    System.out.println("You did a great job");
} else {
    //either testScore < 70 OR studentAge >= 10

    if (testScore >= 70) {
       System.out.println("You did pass");
    } else {
       System.out.println("You did not pass");
    }
}
```

More example (student grade using nested if statement):



```
if (score >= 90)
   System.out.println("Your grade is A");

else if (score >= 80)
   System.out.println("Your grade is B");

else if (score >= 70)
   System.out.println("Your grade is C");

else if (score >= 60)
   System.out.println("Your grade is D");

else
   System.out.println("Your grade is F");
```

Switch statement

Given below is the syntax of the switch structure in Java.

```
switch ( expression )
{
case value1:
  // code
break;
case value2:
  // code
 break;
   ...
   ...
case valueN:
 // code
break;
    ...
```

Given below is an example which prints the day of the week based on the value stored in the int variable num.

1 stands for Sunday, 2 for Monday and so on:

```
int day = s.nextInt; // s is a Scanner object connected to System.in
String str; // stores the day in words


switch(day)
{
case 1:
   str="Sunday";
   break;
case 2:
   str="Monday";
   break;
case 3:
   str="Tuesday";
   break;
case 4:
   str="Wednesday";
   break;
case 5:
   str="Thursday";
   break;
case 6:
   str="Friday";
   break;
case 7:
   str="Saturday";
   break;
default:
   str=" Invalid day";
}
System.out.println(str);
...
default:
   // code
   break;
}
```

Looping statements:

The for loop syntax

```
for ( initialization; condition; increment/ decrement )
{
// body
}
```

The following code prints the numbers from 1 to 10 using a for loop.

```
for ( int i=1; i<=10; i++)
{
System.out.println(i);
}
```

The while loop syntax

Following is the syntax of the while loop.

The statements in the while block are executed as long as the condition evaluates to true.

```
while ( condition )
{
```

```
// code
}
```

Consider the following code snippet which is used to print the statement "KKU" thrice on the screen.

```
int ctr = 0;
while (ctr < 3 )
{
System.out.println("KKU");
ctr++;
}
```

The following code is used to add the numbers from 1 to 100.

```
int ctr = 1;
sum = 0;
while (ctr<=100)
{
  sum = sum + ctr;
  ctr++;
}
```

Syntax of do while loop

```
do {
   // code
}
while ( condition );
```

Following code uses the do while loop to print "KKU" thrice on the screen.

```
int ctr = 0;
do {
   System.out.println("KKU");
   ctr++;
}  while (ctr < 3);
```

Chapter 3

ARRAYS & COLLECTIONS

Java provides a data structure, the **array**, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables.

This tutorial introduces how to declare array variables, create arrays, and process arrays using indexed variables.

DECLARING ARRAY VARIABLES

To use an array in a program, you must declare a variable to reference the array, and you must specify the type of array the variable can reference. Here is the syntax for declaring an array variable −

SYNTAX

```
dataType[] arrayRefVar;   // preferred way.
or
dataType arrayRefVar[];  // works but not preferred way.
```

Note − The style **dataType[] arrayRefVar** is preferred. The style **dataType arrayRefVar[]** comes from the C/C++ language and was adopted in Java to accommodate C/C++ programmers.

EXAMPLE

The following code snippets are examples of this syntax −

```
double[] myList;   // preferred way.
or
double myList[];   // works but not preferred way.
```

CREATING ARRAYS

You can create an array by using the new operator with the following syntax −

SYNTAX

arrayRefVar = new dataType[arraySize];

The above statement does two things –

- It creates an array using new dataType[arraySize].
- It assigns the reference of the newly created array to the variable arrayRefVar.

Declaring an array variable, creating an array, and assigning the reference of the array to the variable can be combined in one statement, as shown below –

dataType[] arrayRefVar = new dataType[arraySize];

Alternatively you can create arrays as follows –

dataType[] arrayRefVar = {value0, value1, ..., valuek};

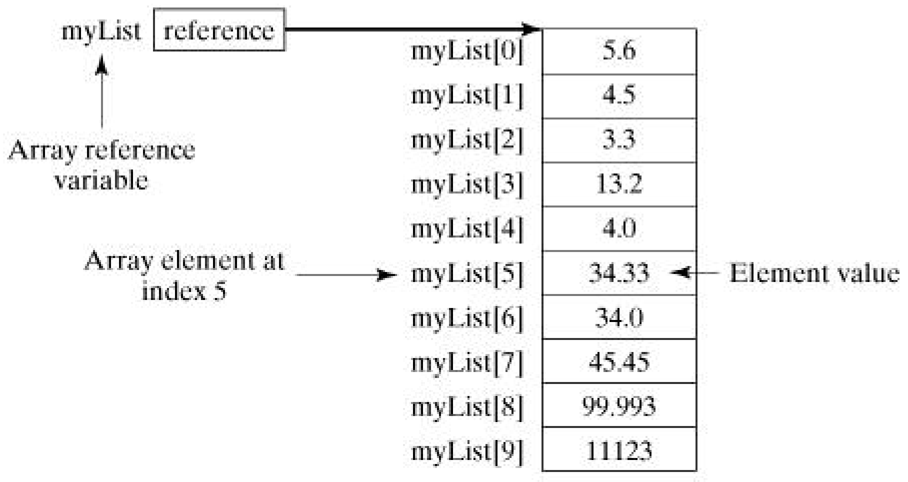
The array elements are accessed through the **index**. Array indices are 0-based; that is, they start from 0 to **arrayRefVar.length-1**.

EXAMPLE

Following statement declares an array variable, myList, creates an array of 10 elements of double type and assigns its reference to myList –

double[] myList = new double[10];

Following picture represents array myList. Here, myList holds ten double values and the indices are from 0 to 9.



ARRAY OF OBJECTS

Following is the definition of Student class.

```
class Student {
  int marks;
}
```

An array of objects is created just like an array of primitive type data items in the following way.

```
Student[] studentArray = new Student[7];
```

The above statement creates the array which can hold references to seven Student objects.

MULTI DIMENSIONAL ARRAYS

We have represented an array type using a pair of brackets. Two dimensional arrays are in a similar way represented by two such pairs of brackets and an N dimensional array is represented by using N such pairs of brackets. The following statement creates a two dimensional array of integers, which contains 3 arrays containing 4 integers each.

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

Elements of this array are accessed by specifying the index numbers, here two of them. The first representing the array number and the second representing the index element in that particular array.

```
a[0][2] = 34;
```

A two dimensional airy can also be initialised using an array initialiser in the following way. This paints a better picture of a 2D array as an array of arrays.

```
int[][] d = { { 1,5,74,2}, {4,68,45,65},{5,0,34,54}}:
```

THE FOREACH LOOPS

JDK 1.5 introduced a new for loop known as foreach loop or enhanced for loop, which enables you to traverse the complete array sequentially without using an index variable.

EXAMPLE

PASSING ARRAYS TO METHODS

Just as you can pass primitive type values to methods, you can also pass arrays to methods. For example, the following method displays the elements in an **int** array –

EXAMPLE

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

You can invoke it by passing an array. For example, the following statement invokes the printArray method to display 3, 1, 2, 6, 4, and 2 –

EXAMPLE

```
printArray(new int[]{3, 1, 2, 6, 4, 2});
```

LISTS & MAPS

LIST INTERFACE

A List is a ordered collection that can contain duplicate values. It provides three general purpose implementations.

1) ArrayList
2) LinkedList
3) Vector

1) ARRAYLIST

ArrayList is said to be the best performing list implementation in normal conditions. In simple words we can say that ArrayList is a expendable array of values or objects.

```
package com.beingjavaguys.core;

import java.util.ArrayList;

public class ArrayListImplementation {

  public static void main(String args[]){
        ArrayList<String> arrayList = new ArrayList<String>();

        arrayList.add("element1");
        arrayList.add("element2");
        arrayList.add("element3");
        System.out.println(arrayList);

        arrayList.remove("element3");
        System.out.println(arrayList);
    }
}
```

2) LINKEDLIST

Linked list is a bit slower than ArrayList but it performs better in certain conditions.

3) VECTOR

Vector is also a growable array of objects, but unlike ArrayList Vector is thread safe in nature.

Map Interface

A Map interface provides key-value pairs, map objects contains keys associated with an value. Maps can not contain duplicate keys and one key can be mapped to atmost one element. In Java Map interface provides three general purpose implementations.

1) HashMap- A HashMap is a HashTable implementation of Map interface, unlike HashTable it contains null keys and values. HashMap does not guarantees that the order of the objects will remain same over the time.
2) TreeMap- A TreeMap provides red-black tree based implementation of Map interface.
3) LinkedHashMap- It is HashTable and LinkedList implementation of Map interface. LinkedHashMap has a double-linkedList running through all its elements.

Chapter 4: Classes and Objects

What is Object Oriented Programming?

- OBJECT-ORIENTATION is a set of tools and methods that enable software engineers to build reliable, user friendly, maintainable, well documented, reusable software that fulfills the requirements of its users.
- A software system is seen as a community of objects that cooperate with each other by passing messages in solving a problem.

Benefits of OO programming

– Easier to understand (closer to how we view the world)

– Easier to maintain (localized changes)

– Good level of code reuse (inheritance)

OOP consists of the following features (Concepts):

Inheritance:

Inheritance is the process of forming a new class from an existing class or base class.

Inheritance helps in reducing the overall code size of the program, which is an important concept in object-oriented programming.

Data Abstraction:

Data Abstraction represents the needed information in the program without presenting the details.

Data Encapsulation:

Data Encapsulation combines data and functions into a single unit called Class. When using Data Encapsulation, data is not accessed directly; it is only accessible through the functions present inside the class.

Data Encapsulation enables the important concept of **data hiding** possible.

Polymorphism:

Polymorphism is the capability of a method to do different things based on the object that it is acting upon.

What Is an Object?

- An object represents an entity (thing) in the real world that can be distinctly identified.

- **For example:**

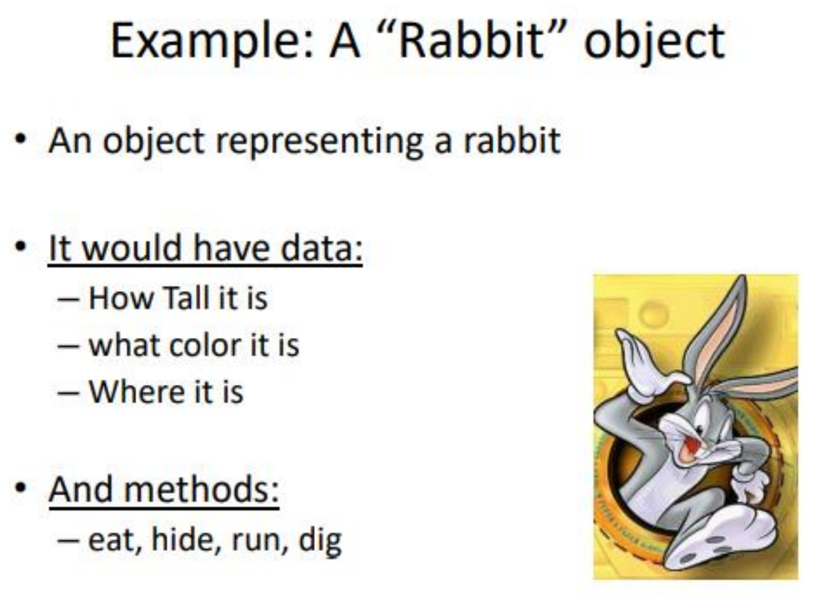
  a student, a desk, a circle, a button… can all be viewed as objects.

- An object has a unique identity state, and behaviors.

- The state of an object consists of a set of data fields (also known as properties) with their current values.

- The behavior of an object is defined by a set of methods.

- An object is called an instance of a class.



Example: A "Rabbit" object

- An object representing a rabbit
- It would have data:
  - How Tall it is
  - what color it is
  - Where it is
- And methods:
  - eat, hide, run, dig

3

Classes in Java

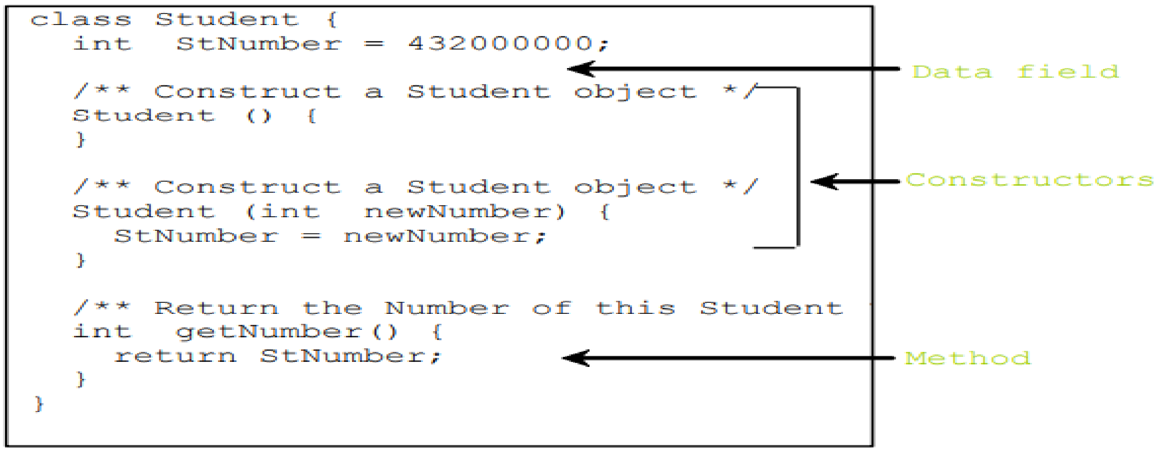
- A class is a template that describes the data and behavior associated with instances of that class.
- A Java class uses variables to define data fields and methods to define behaviors.
  - A class provides a special type of methods, known as **Constructor** which are invoked to construct objects from the class.

Syntax of creating Class:

```
  class classname
{
  Data fields (Variables)
 Constructors
 Methods
}
```

- A class definition starts with the keyword **class** followed by the class name.
- The class body contains the (Data fields) , (Constructors), and (methods) enclosed by curly braces.

Classes



```
class Student {
   int  StNumber = 432000000;

   /** Construct a Student object */
   Student () {
   }

   /** Construct a Student object */
   Student (int  newNumber) {
      StNumber = newNumber;
   }

   /** Return the Number of this Student
   int  getNumber() {
      return StNumber;
   }
}
```

Declaring an object

Syntax

ClassName objectName;

e.g., Student myStudent;

Creating an object

objectName = new className();

e.g., myStudent = new Student();

Declaring/Creating Objects in a Single Step using java

ClassName objectName = new ClassName();

e.g., Student myStudent = new Student();

Accessing Objects

- Members are accessed using the **dot(.**) operator.

- you must write its object’s name followed by dot sign(.) and the names of its members (data or Method).

• **Referencing the object’s data:**

```
Objectname.data
```

e.g., myStudent.stNumber

• **Invoking the object’s method:**

```
ObjectName.methodName(arguments)
```

e.g., myStudent.getNumber()

<u>Example1:</u>

Write Java program as described below:

Create a **Student** class which has two data members: **id** and **name**.

Print the data member values by creating an object **s1** from **Student** class.

```
class Student
{
 int id =5;
 String name= "Fatima";      /
}

public class program1
   {
 public static void main(String[ ] args)
{
 Student s1=new Student( );                    //creating an object of Student
 System.out.println(s1.id);
 System.out.println(s1.name);
 }
}
```

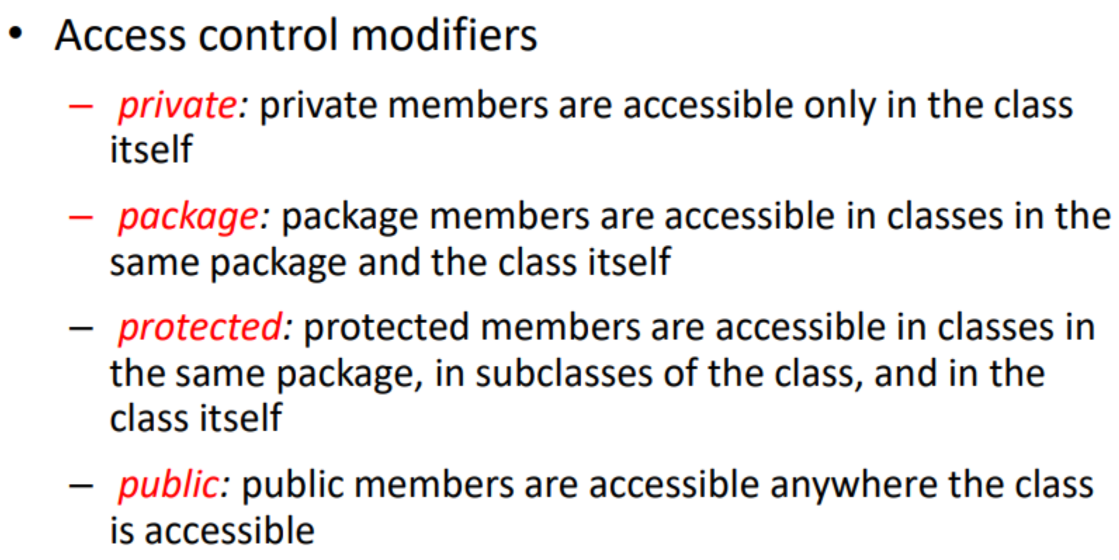
Output

```
5
Fatima
```

More about Data Fields

***Data fields*: data variables which determine the status of the class or an object.**

- **Field Declaration**
  - **As shown in the previous example, data afield is declared by adding Datatype name followed by the field name, and optionally an initialization clause.**
  - **Example:** int EmpID =1234;
- **field declarations can be preceded by different modifiers**
  - **access control modifiers**
  - **static**
  - **final**



- Access control modifiers
  - *private:* private members are accessible only in the class itself
  - *package:* package members are accessible in classes in the same package and the class itself
  - *protected:* protected members are accessible in classes in the same package, in subclasses of the class, and in the class itself
  - *public:* public members are accessible anywhere the class is accessible

Pencil.java

```
public class Pencil {
    public String color = "red";
    public int length;
    public float diameter;
    private float price;

    public static long nextID = 0;

    public void setPrice (float newPrice) {
        price = newPrice;
    }
}
```

CreatePencil.java

```
public class CreatePencil {
    public static void main (String args[]){
        Pencil p1 = new Pencil();
        p1.~~price~~ = 0.5f;
    }
}
```

```
CreatePencil.java:4: price has private access in Pencil
        p1.price = 0.5f;
          ^
```

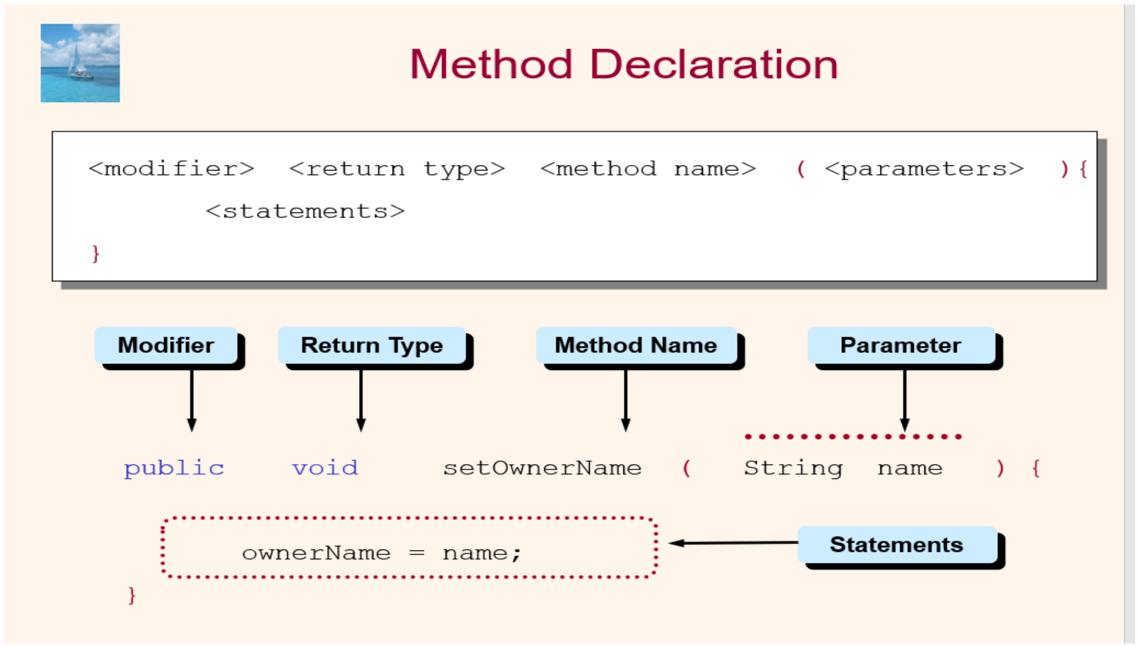
- static
  - can be accessed directly in the class itself
  - access from outside the class must be preceded by the class name as follows

```
System.out.println(Pencil.nextID);
```

    or via an object belonging to the class
  - from outside the class, non-static fields must be accessed through an object reference
- final
  - once initialized, the value cannot be changed
  - often be used to define named constants

Methods

A Java method is a collection of statements that are grouped together to perform an operation.



Method Declaration

```
<modifier>  <return type>  <method name>  ( <parameters>  ){
       <statements>
}
```

```
public    void    setOwnerName  (  String  name   )  {
      ownerName = name;
}
```

Method Calling

- To execute a method, you simply call the method by typing object name followed by dot operator (.) followed by method name and then follow the name with a set of parentheses.
- You can call the same method multiple times.
- calling methods usually occur inside other methods like **main** method.
- When a method runs, the compiler jumps to where the method is defined, executes the code inside of it, then goes back and proceeds to the next line.

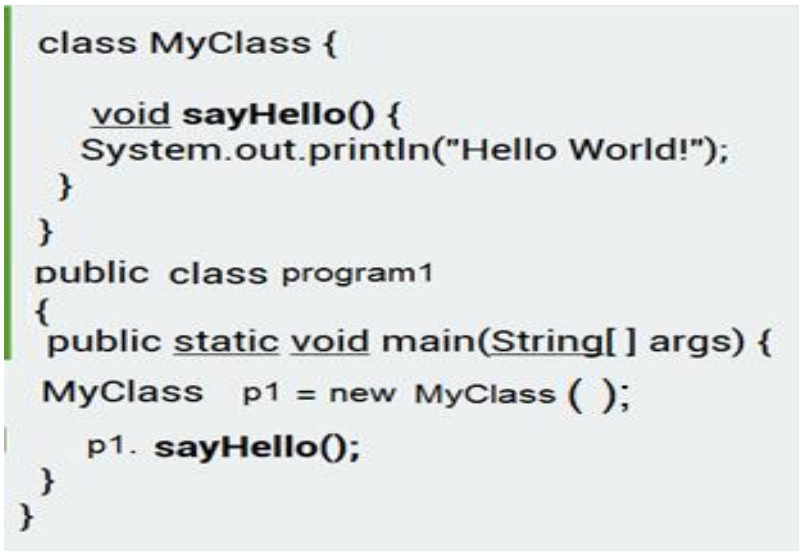
The following example demonstrates how the method is declared and then called:

Example 1

Write JAVA program as described below:

1. Create a class named **MyClass**, with **sayHello** method to display (Hello world!) on the screen.

2. Invoke(call) the **sayHello**() method by creating an object **p1** from **MyClass** class.



```
class MyClass {

    void sayHello() {
   System.out.println("Hello World!");
  }
}
public  class program1
{
 public static void main(String[ ] args) {
 MyClass   p1 = new MyClass ( );
    p1. sayHello();
 }
}
```

Output:

```
Hello World!
```

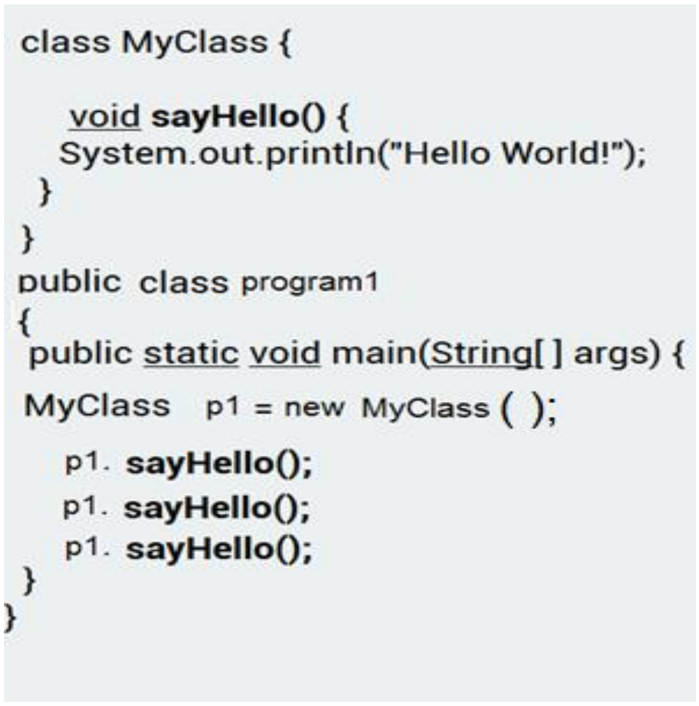
The following example demonstrate how the method can be called many times as necessary:

Example 2:

Write JAVA program as described below:

1. Create a class named **MyClass**, with **sayHello** method to display (Hello world!) on the screen.

2. Invoke the sayHello() method **3 times** by creating an object **p1** from **MyClass** class.



```
class MyClass {

    void sayHello() {
   System.out.println("Hello World!");
  }
 }
public  class program1
{
 public static void main(String[ ] args) {
 MyClass   p1 = new MyClass ( );
    p1. sayHello();
    p1. sayHello();
    p1. sayHello();
 }
}
```

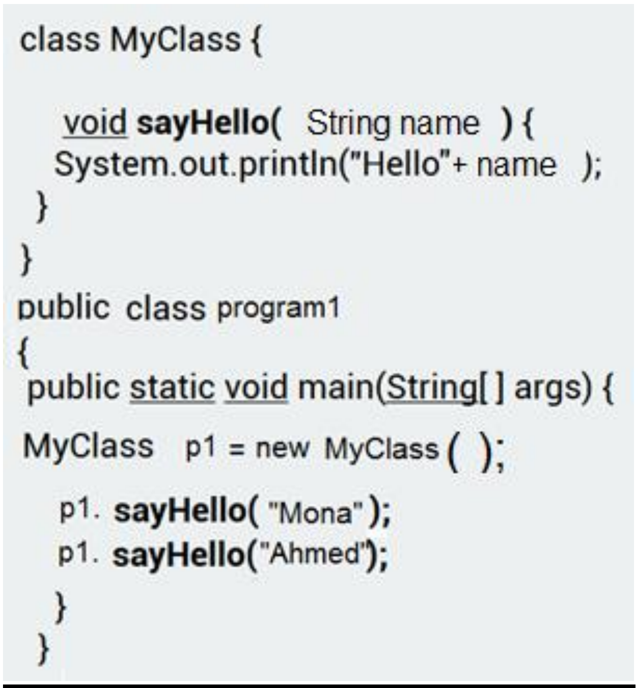
Output:

```
Hello World!
Hello World!
Hello World!
```

<u>Method Parameters</u>

- You can also create a method that takes some data, called **parameters**
- Write parameters within the method's parentheses.

For example, we can modify our **sayHello**() method to take a **String** parameter.



```
class MyClass {

   void sayHello(  String name  ) {
  System.out.println("Hello"+ name  );
  }
}
public  class program1
{
 public static void main(String[ ] args) {
 MyClass   p1 = new MyClass (  );
   p1. sayHello( "Mona" );
   p1. sayHello("Ahmed");
  }
 }
```

Output:

```
Hello Mona

Hello Ahmed
```

<u>The void keyword</u>

When you do not need to return any value from a method, (like methods in the above examples) we usually use the keyword **void**.

<u>The return keyword</u>

The **return** keyword can be used in methods to return a value.

Keyword this

- Can be used only inside method
- When call a method within the same class, don't need to use ***this***, compiler do it for you.
- When to use it?
  - method parameter or local variable in a method has the same name as one of the fields of the class
  - Used in the return statement when want to return the reference to the current object.
- Example …

Keyword *this* example I

```
class A{
            int w;
            public void setValue (int w) {
                        this.w = w;  //same name!
            }
}
```

When a method parameter or local variable in a method has the same name as one of the fields of the class, you must use this to refer to the field.

Constructors

Constructors are a special kind of methods that are invoked to perform initializing actions.

```
Student (int newNumber) {
StNumber = newNumber;
}
```

- A constructor with no parameters is referred to as: **default** constructor.
- Constructors must have the same name as the class itself.
- Constructors do not have a return type—not even void.
- Constructors play the role of initializing objects.

Constructor example

```
class Circle{
        double r;
        public Circle (double r) {
                this.r = r;  //same name!
        }
        public static void main(String[] args){
                Circle c = new Circle(2.0); //OK
                Circle c2 = new Circle(); //error!!, no more default
        }
}
```

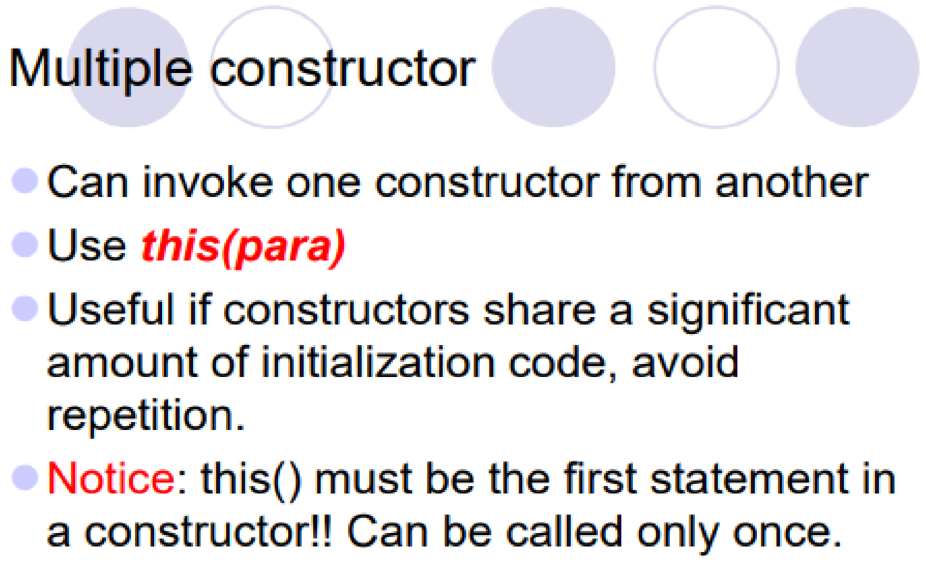
```
Circle.java:8: cannot resolve symbol
symbol  : constructor Circle ()
location: class Circle
              Circle c2 = new Circle(); //error!!
                          ^
1 error
```

Constructor example

```
class Circle{
        double r;
        public Circle(){
                r = 1.0; //default radius value;
        }
        public Circle (double r) {
                this.r = r;  //same name!
        }
        public static void main(String[] args){
                Circle c = new Circle(2.0); //OK
                Circle c2 = new Circle();   // OK now!
        }
}
```

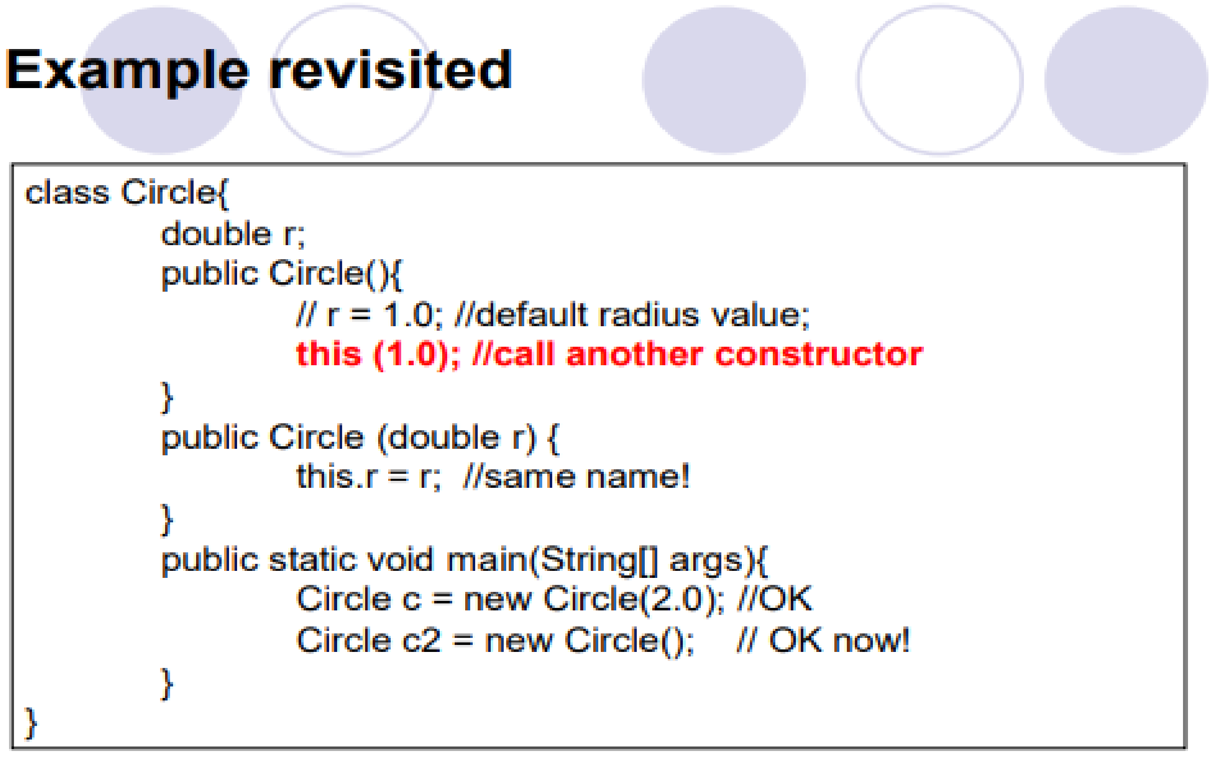
Multiple constructor now!!

Call one constructor inside another one.



Multiple constructor

- Can invoke one constructor from another
- Use ***this(para)***
- Useful if constructors share a significant amount of initialization code, avoid repetition.
- Notice: this() must be the first statement in a constructor!! Can be called only once.



Example revisited

```
class Circle{
        double r;
        public Circle(){
                // r = 1.0; //default radius value;
                this (1.0); //call another constructor
        }
        public Circle (double r) {
                this.r = r;  //same name!
        }
        public static void main(String[] args){
                Circle c = new Circle(2.0); //OK
                Circle c2 = new Circle();    // OK now!
        }
}
```

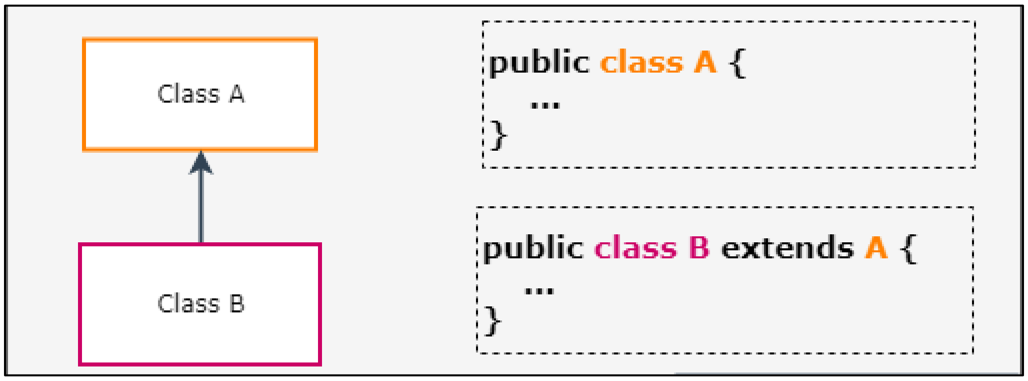
Chapter 5

Inheritance & polymorphism

Part1: Inheritance

- Inheritance is the ability to define a new class in terms of an existing class.
    - The existing class is the (**parent, base or superclass**).
    - The new class is the (**child, derived or subclass**)
- The child class inherits all the attributes and behavior of its parent class.
    - It can then add new attributes or behavior.
- Inheritance is therefore another form of code reuse
- Use keyword **extends** for inheritance.

Syntax



```
public class A {
    ...
}

public class B extends A {
    ...
}
```

Example1: Program that illustrates inheritance in java using Animal class

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
Public class Test{
public static void main(String args[]){
Dog d=new Dog();
d.bark();
d.eat();
}
}
```

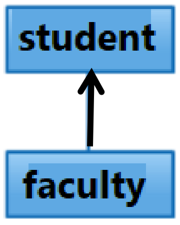
Output:

```
barking...

eating...
```

Types Of Relationships in Object Oriented Programming (OOP)

Association

- Association is a relationship between two objects.
- ***Example***: A Student and a Faculty are having an association.



Association

Aggregation

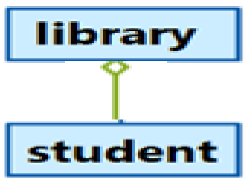
- Aggregation is a special case of association.
- When an object '**has-a**' another object, then you have got an aggregation between them.
- Aggregation is also called a "Has-a" relationship.

Composition

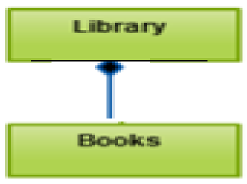
- Composition is a special case of aggregation.

Example of aggregation and composition

- A Library contains students and books.
- Relationship between library and student is aggregation.
- Relationship between library and book is composition.
- A student **can exist** without a library and therefore it is **aggregation**.
- A book **cannot exist** without a library and therefore it is a **composition**.



Aggregation

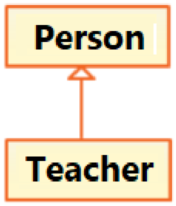


Composition

Generalization

- Generalization uses a “is-a” relationship from a specialization to the generalization.
- At a very broader level you can understand this as **inheritance**.
- Generalization is also called a “Is-a” relationship.
- ***Example:*** Consider there exists a class named Person.

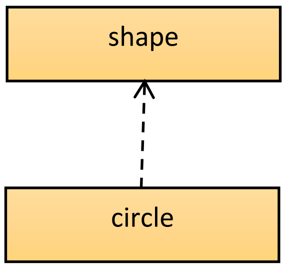
  A teacher is a person. Therefore, here the relationship between teacher and person, is **generalization**.



Generalization or Inheritance

Dependency

- Change in structure or behavior of a class affects the other related class, then there is a dependency between those two classes.
- ***Example***: Relationship between shape and circle is dependency.



Dependency

Part 2: Polymorphism

What is polymorphism in programming?

Polymorphism is the capability of a method to do different things based on the object that it is acting upon.

The polymorphism concept can be achieved using:

1) Method Overloading

2) Method Overriding

1) Method Overloading

Two or more methods in the class can have the same name but their argument lists are different.

This concept is known as **Method Overloading.**

Argument lists could differ in –

1. Number of parameters.
2. Data type of parameters.
3. Sequence (Order) of Data type of parameters

Example 1: Overloading – Number of parameters

```
class Sum
  {
   public void add(int a, int b)
       {
          System.out.println(a+b);
      }
    public void add(int a, int b, int c)
       {
           System.out.println(a+b+c);
      }
 }
public class Sample
  {
      public static void main(String args[])
        {
                   Sum obj = new Sum ();
                  obj.add(3,4);
                  obj.add(2,5,7);
        }
 }
```

Output:

```
7
14
```

In the above example – method `add()` has been overloaded based on the number of parameters. We have two definition of method `add()`, one with two parameters andanother with three parameters.

Example 2: Overloading – Data type of parameters

```
class DisplayOverloading2
{
    public void disp(char c)
      {
        System.out.println(c);
      }

    public void disp(int c)
      {
        System.out.println(c );
      }
}

public class Sample2
{
   public static void main(String args[])
     {
          DisplayOverloading2 obj = new DisplayOverloading2();
         obj.disp('a');
         obj.disp(5);
    }
}
```

Output:

```
a
5
```

In the above example – method disp() is overloaded based on the data type of parameters – Like example 1 here also, we have two definition of method disp(), onewith char parameter and another with int parameter.

Example3: Overloading – Sequence (Order) of data type of parameters

```
class DisplayOverloading3
  {
     public void disp(char c, int num)
       {
           System.out.println(c+ ”  “ + num );
       }
     public void disp(int num, char c)
     {
        System.out.println(num + “  “+ c );
     }
  }
public class Sample3
 {
      public static void main(String args[])
      {
          DisplayOverloading3 obj = new DisplayOverloading3();
           obj.disp('x', 51 );
           obj.disp(52, 'y');
      }
}
```

Output:

```
X  51

52  y
```

Here method `disp()` is overloaded based on sequence of data type of parameters –Both the method have different sequence of data type of parameters.

2) Method overriding.

Declaring a method in **subclass** which is already present in **parent class** is known as **method overriding.**

Rules for Method Overriding:

1. applies only to inherited methods
2. object type (NOT reference variable type) determines which overridden method will be used at runtime
3. Overriding methods must have the same return type
4. Overriding method must not have more restrictive access modifier
5. Abstract methods must be overridden
6. Static and final methods cannot be overridden
7. Constructors cannot be overridden
8. It is also known as Runtime polymorphism.

Advantage of method overriding

The main advantage of method overriding is that the class can give its own specific implementation to an inherited method without even modifying the parent class(base class).

Example:

One of the simplest example – Here `Boy` class extends `Human` class. Both the classes have a common method `void eat()`. Boy class is giving its own implementation to the `eat()` method or in other words it is overriding the method `eat()`.

```
class Human{
  public void eat()
  {
    System.out.println("Human is eating.");
  }
}

public class Boy extends Human {
    public void eat(){
    System.out.println("Boy is eating.");
  }
  public static void main( String rgs[]) {
        Boy obj = new  Boy();
        obj.eat();
  }
}
```

Output:

```
Boy is eating.
```

Super keyword in Overriding.

super keyword is used for calling the parent class method/constructor.

super.methodname() calling the specified method of base class while **super()** calls the constructor of base class.

Let’s see the use of super in Overriding.

```
class ABC{
    public void mymethod()
    {
        System.out.println("Class ABC: mymethod()");
    }
}
class Test extends ABC{
   public void mymethod(){
        //This will call the mymethod() of parent class
        super.mymethod();
        System.out.println("Class Test: mymethod()");
      }
     public static void main( String args[]) {
       Test obj = new Test();
       obj.mymethod();
     }
}
```

Output:

```
Class ABC: mymethod()
Class Test: mymethod()
```

Types of polymorphism in java-

There are two types of polymorphism in java:

1. **Runtime polymorphism (Dynamic polymorphism)**
   - Method Overriding is a perfect example of runtime polymorphism.

2. **Compile time polymorphism (static polymorphism).**
   - Method Overloading is a perfect example of runtime polymorphism.

Chapter 6

Abstract Classes ,Interfaces, and Encapsulation

Abstract Classes

A class which **cannot** be **instantiated** is known as **abstract class.** In other words –you are not allowed to create **object** of Abstract class.

Abstract class declaration

Specifying **abstract keyword** before the class during declaration, makes it **abstract class**.
Have a look at below code:

```
abstract class AbstractDemo{
public void myMethod(){
//Statements here
}
}
```

Abstract vs Concrete

A class which is not abstract is referred as **Concrete class**.

Abstract methods

Apart from having abstract class you can have **abstract methods** as well.

Syntax of abstract method:

```
public abstract void display();
```

Points to remember about abstract method:

1) Abstract method has no body.
2) Always end the declaration with a **semicolon(;).**
3) It must be overridden.
    - An abstract class must be extended and in a same wayabstract method must be overridden.
4) Abstract method must be in a abstract class.

Note: The class which is extending abstract class must override (or implement) all theabstract methods.

Example of Abstract class and method

```
abstract class Demo1
{
  public void disp1(){
     System.out.println("Concrete method of abstract class");
    }

 abstract public void disp2();
}

class Demo2 extends Demo1{
public void disp2()
 {
  System.out.println("I'm overriding abstract method");
 }
public static void main(String args[]){
 Demo2 obj = new Demo2();
obj.disp2();
}
}
```

Output:

```
I'm overriding abstract method
```

Interface in JAVA

Declaration

Interfaces are created by specifying a keyword “**interface**”. E.g.:

```
interface MyInterface
 {
    //All the methods are public abstract by default
     public void method1();
     public void method2();
  }

class XYZ implements MyInterface
  {
    public void method1()
      {
         System.out.println("implementation of method1");
      }
   public void method2()
     {
        System.out.println("implementation of method2");
     }
   public static void main(String arg[])
     {
        MyInterface obj = new XYZ();
        obj. method1();
    }
  }
```

Output:

```
implementation of method1
```

Key points:

1. While providing implementation of any method of an interface, it needs to bementioned as **public**.
2. Class implementing any interface must implement all the methods.
3. Interface cannot be declared as private, protected or transient.
4. All the interface methods are by default **abstract and public**.
5. Variables declared in interface are **public, static and final** by default.

```
interface Try
{
int a=10;

public int a=10;

public static final int a=10;

 final int a=10;

static int a=10;

}
```

All the above statements are identical.

6. Interface variables must be initialized at the time of declaration otherwise compiler will through an error.

```
interface Try
{
int x;//Compile-time error
}
```

Above code will throw a compile time error as the value of the variable x is notinitialized at the time of declaration.

7. Inside any implementation class, you cannot change the variables declared in interface because by default, they are public, static and final.

```
Class Sample implements Try
 {
   public static void main(String arg[])
  {
     x=20; //compile time error
   }
  }
```

8. Any interface can extend any other interface but cannot implement it. Class implements interface and interface extends interface.
9. A class can implements any number of interfaces.
10. If there are having two or more same methods in two interfaces and a class implements both interfaces, implementation of one method is enough.

```
interface A
 {
    public void aaa();
 }

interface B
 {
    public void aaa();
 }

class Central implements A,B
{

public void aaa()
 {
   //Any Code here
 }
public static void main(String arg[])
  {
   //Statements
  }
}
```

11. Variable names conflicts can be resolved by interface name e.g:

```
      interface A
       {
         int x=10;
       }

      interface B
       {
        int x=100;
       }

class Hello implement A,B
 {
   public static void Main(String arg[])
     {
      System.out.println(A.x);
      System.out.println(B.x);
     }
  }
```

Interface and Inheritance

As I discussed above that one interface can not implement another interface. It has toextend the other interface if required.

See the below example where I have two interfaces Inf1 and Inf2. Inf2 extends Inf1 so If class implements the Inf2 it has to provide implementation of all the methods of interfaces Inf1 and Inf2.

```
    public interface Inf1

       {

       public void method1();
     }
     public interface Inf2 extends Inf1
     {
       public void method2();
      }

    public class Demo implements Inf2
    {
    public void method1(){
    //Implementation of method1
    }
    public void method2(){
    //Implementation of method2
    }
    }
```

Benefits of having interfaces: Following are the benefits of interfaces:

1. achieve the security ofimplementation.
2. In java, **multiple inheritance** is not allowed, However by using interfaces you can achieve the same .

Encapsulation in Java

What is encapsulation?

- The whole idea behind encapsulation is to hide the implementation details from users.
- If a data member is **private** it means it can only be accessed within the same class.
- No outside class can access private data member (variable) of other class.
- However if we setup public getter and setter methods to update the private data fields then the outside class can access those private data fields via public methods.
- This way data can only be accessed by public methods thus making the private fields and their implementation hidden for outside classes. That’s why encapsulation is known as **data hiding.**

Lets see an example to understand this concept better.

```
public class EncapsulationDemo{
      private int ssn;
      private String empName;
      private int empAge;

      //Getter and Setter methods
      public int getEmpSSN() {

            return ssn;
        }


      public String getEmpName(){
        return empName;
       }

      public int getEmpAge(){
          return empAge;
      }

      public void setEmpAge(int newValue){
          empAge = newValue;
      }

      public void setEmpName(String newValue){
         empName = newValue;
      }

      public void setEmpSSN(int newValue){
           ssn = newValue;
       }
}

public class EncapsTest{
      public static void main(String args[]){
              EncapsulationDemo obj = new EncapsulationDemo();
              obj.setEmpName("Mario");
              obj.setEmpAge(32);
              obj.setEmpSSN(112233);
              System.out.println("Employee Name: " + obj.getEmpName());
              System.out.println("Employee SSN: " + obj.getEmpSSN());
              System.out.println("Employee Age: " + obj.getEmpAge());
              }
              }
```

Output:

```
Employee Name: Mario
Employee SSN: 112233

Employee Age: 32
```

Advantages of encapsulation:

1. It improves maintainability and flexibility and re-usability
2. The fields can be made read-only (If we don’t define setter methods in the class) or write-only (If we don’t define the getter methods in the class).
3. User would not be knowing what is going on behind the scene.