**Learn Java with Shamik**

##### 

**About Me:**

**I have 9 years of professional experience in Java. Currently I work for IBM as a Technical leader. During my service life I was awarded “Eminence &Excellence”,” Spark award” “Deep Skill” in IBM. Won the “Jury Award” in Techathon Coding competition which was held over all location in IBM India. I love to share my knowledge so I am open to teach one-to-one or group classes. I prefer Skype or Google hangout as online tutoring tool.**

**Email:** [**mitrashamik@gmail.com**](mailto:mitrashamik@gmail.com)

**skype: mitrashamik**

**Core Java Understanding**

**What is Java?**

Java is a high-level programming language originally developed by Sun Microsystems and released in 1995. Java runs on a variety of platforms, such as Windows, Mac OS, and the various versions of UNIX. So java is a platform independent language. This tutorial gives a complete understanding of Java.

**Java Latest Release and Founder/History**

The latest release of the Java Standard Edition is Java SE 8. With the advancement of Java and its widespread popularity, multiple configurations were built to suite various types of platforms. Ex: J2EE for Enterprise Applications, J2ME for Mobile Applications.

The new J2 versions were renamed as Java SE, Java EE and Java ME respectively. Java is guaranteed to be **Write Once, Run Anywhere.**

**James Gosling** initiated the Java language project in June 1991 for use in one of his many set-top box projects. The language, initially called **Oak** after an oak tree that stood outside Gosling's office, also went by the name **Green** and ended up later being renamed as **Java**, from a list of random words.

**Sun released the first public implementation as Java 1.0 in 1995.**

**What is Object Oriented Language/OOPS:**

Object means a real word entity such as pen, chair, table, Bottle etc. Object-Oriented Programming is a methodology or paradigm to design a program using classes and objects. It simplifies the software development and maintenance by providing some concepts:

* Object
* Class
* Inheritance
* Polymorphism
* Abstraction
* Encapsulation

Java satisfies all above concept so java is an Object Oriented Programming Language.

#### Object

Any entity that has state /properties and behaviour/methods is known as an object. For example: chair, pen, table, keyboard, bike etc. It can be physical and logical. Object is real-time entity in **java world so called,** **JVM(Java Virtual Machine)**. In Physics object is which take some places which has properties and which can do some actions Same in Java, Object takes memory space i.e. space in RAM it has properties and it can do some actions like, by **pen we can write so write is an action /in java term we call it method.**

If we consider a dog, then its state/properties is - name, breed, colour, and the behaviour /methods is - barking, wagging, running

**Class**

Class is Logical entity or we can call it blueprint. Like for each house there is a plan for that house. So plan is Class and the House itself is an Object

#### Inheritance

**When one object acquires all the properties and behaviours of parent object** i.e. known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.

For Examples, All Vehicle must have wheels and BMW has 4 wheels so If Vehicles is **parent** then **BMW** is child and it inherits wheels property.

#### Polymorphism

When **one task is performed by different ways** i.e. known as polymorphism.

In java, we use method **overloading** and method **overriding** to achieve polymorphism.

example can be to speak behaviour e.g. cat speaks meow, dog barks woof etc.

#### Abstraction

**Hiding internal details and showing functionality** is known as abstraction. For example: driving a car, we know how to drive but don't know how the **internal mechanism** actually moves the car.

**In java, we use abstract class and interface to achieve abstraction.**

#### Encapsulation

**Binding (or wrapping) code and data together into a single unit is known as encapsulation**. For example: capsule, it is wrapped with different medicines.

**In Java Class wrap properties and methods.**

**Installing Java:**

**Download Java from here**

[**http://www.oracle.com/technetwork/java/javase/downloads/index.html**](http://www.oracle.com/technetwork/java/javase/downloads/index.html)

**Download Eclipse from here**

[**http://www.eclipse.org/downloads/packages/eclipse-ide-java-developers/marsr**](http://www.eclipse.org/downloads/packages/eclipse-ide-java-developers/marsr)

**Install java and extract eclipse folder on your local folder**

**Now open eclipse**

**Go to File->JavaProject->create project**

**Create project and happy coading.**

**Java Basic Syntax of a Class**

package <dot separated name> optional (Address of the class)

import <fqdn of class want to use >(address of class which are used in following class)

<Access Specifier/modifiers><non access modifirs> class <Name of Class>

{

<Access Specifier/modifiers> <type of property><propertyName>=<intialization value>;

<Access Specifier/modifiers> <return Type/void><method Name>(<type of property> <propertyName> can be 1...n)

{

//business logic goes here

return <return Type>;

}//end of method

}//end of class

**Now First Java Program:**

HelloWorld.java

package com.example

public class HelloWorld

{

private String msg=”Hello Shamik !!!”

public String getMessage()

{

return msg;

}

public static void main(String args[])//entry point , To run the program it is requred.

{

HelloWorld world = new HelloWorld(); //create a HelloWorld object and link it with world refrence

String msg = world.getMessage();//call method by help of refrence . Operator use in java to call methods.

System.out.println(msg);

}

}

**Now Right click on the class and go to Run As and click on run as a java application**

**You will see the output**

**Hello Shamik!!!**

**Discussing About Syntax**

A class is consisting of properties and methods.

**Methods -** A method is basically a behaviour or we can think it as a verb. A class can contain many methods. It is in methods where the business logics are written, data is manipulated and all the actions are executed.

**Instance Variables /properties/member variables-** Each object has its unique set of instance variables/member variables. An object's state is created by the values assigned to these instance variables/properties. In above example msg is member cariable.

Java Best practises and Key things to remember

**Case Sensitivity -** Java is case sensitive, which means identifier **MSG** and **msg** would have different meaning in Java.

**Class Names -**class name’s first letter should be in Upper Case.   
  
If several words are used to form a name of the class, each inner word's first letter should be in Upper Case. Like HelloWorld

**Method Names -** All method names should start with a Lower Case letter.   
  
If several words are used to form the name of the method, then each inner word's first letter should be in Upper Case. We called it camelCase.

Like getMessage()

**Keywords**- Java has reserved keywords so you can't use them as method or variable name. Like class is reserved keyword so you can't use it as a variable name/method name.

**Program File Name -** Name of the program file should exactly match the class name.

If File name is **HelloWorld** then public class name should be **HelloWorld.** If file does not contain any public class, then name is immaterial.

**Java Identifiers** - We call it as properties. Rule as follows

All identifiers should begin with a letter (A to Z or a to z), currency character ($) or an underscore

(\_).

After the first character identifiers can have any combination of characters.

A key word cannot be used as an identifier.

Most importantly identifiers are case sensitive.

Examples of legal identifiers: age, $salary, \_value, \_\_1\_value

Examples of illegal identifiers: 123abc, -salary

**Java Modifiers -**

There are two categories of modifiers:

* **Access Modifiers:** default, public, protected, private
* **Non-access Modifiers:** final, abstract, strictfp

**Java Variables** -

We would see following type of variables in Java:

* Local Variables
* Class Variables (Static Variables/Global variables/static)
* Instance Variables (Non-static variables)

**Java Arrays** :

Arrays are objects that store multiple variables of the same type. However, an array itself is an object on the heap.

Enums were introduced in java 5.0. Enums restrict a variable to have one of only a few predefined values. The values in this enumerated list are called enums.

For example, if we consider an application for a Shoe shop, it would be possible to restrict the size to small, medium and large. This would make sure that it would not allow anyone to order any size other than the small, medium or large.

Example

Class ShoeShop {

enum ShoeSize{ SMALL, MEDIUM, LARGE }

public ShoeSize size;

}

public class ShoeShopTest {

public static void main(String args[]){

ShoeShop shop = new ShoeShop();

shop.size = ShoeShop.ShoeSize.LARGE ;

System.out.println("Shoe Size is: " + shop.size.toString());

}

}

**Java Keywords**

Java has following reserved keywords

|  |  |  |  |
| --- | --- | --- | --- |
| **abstract** | **assert** | **boolean** | **break** |
| **byte** | **case** | **catch** | **char** |
| **class** | **const** | **continue** | **default** |
| **do** | **double** | **else** | **enum** |
| **extends** | **final** | **finally** | **float** |
| **for** | **goto** | **if** | **implements** |
| **import** | **instanceof** | **int** | **interface** |
| **long** | **native** | **new** | **package** |
| **private** | **protected** | **public** | **return** |
| **short** | **static** | **strictfp** | **super** |
| **switch** | **synchronized** | **this** | **throw** |
| **throws** | **transient** | **try** | **void** |
| **volatile** | **while** |  |  |

**How to Define a class in Java**

**A class is a blue print from which individual objects are created.**

A sample of a class is given below:

public class TechnoStudent{

public static String collegeName=”Techno India”;//Class variable/global

private String name;//member variable/instance variable

private int age

public void getName(){

return name;

}

public void getAge(){

return age;

}

public String getStream()

{

String stream = “CSE”;//local variable

return stream

}

}

**Local variables:** Variables defined inside methods, constructors or blocks are called local variables. The variable will be declared and initialized within the method and the variable will be destroyed when the method has completed. Ex : stream in above example

* **Instance variables:** Instance variables are variables within a class but outside any method. These variables are initialized when the class is instantiated. Instance variables can be accessed from inside any method, constructor or blocks of that particular class. Ex : age
* **Class variables:** Class variables are variables declared with in a class, outside any method, with the static keyword. It will be share by all instances of this class Ex: collegeName

**Constructor**

Every class has a constructor. If we do not explicitly write a constructor for a class the Java compiler builds a default constructor for that class.

Each time a new object is created, at least one constructor will be invoked. The main rule of **constructors is that they should have the same name as the class. A class can have more than one constructor. It looks like a method but without return type .** If you define constructor explicitly then compile does not injected constructor

Example

public class Teacher{

public Teacher(){

}

public Teacher(String name){

// This constructor has one parameter, *name*.

}

}

**Object Creation**

As mentioned previously, a class provides the blueprints for objects. So basically an object is created from a class. In Java, the new key word is used to create new objects.

There are three steps when creating an object from a class:

* **Declaration:** A variable declaration with a variable name with an object type. We call it refrence so we can link the create object. It is like pointer by which we can access the created objects. Like the phone number is a pointer of a person to access.
* **Instantiation:** The 'new' key word is used to create the object. So we can think of it as labour who will actually build the object in jvm from the blue print/class.
* **Initialization:** The 'new' keyword is followed by a call to a constructor. This call initializes the new object. Actually here objects acquire memory in RAM.

Example:

public static void main(String []args){

// Following statement would create an object Teacher

Teacher teacher = new Teacher( "Shamik" );

}

here teacher is reference of newly created object whose name is Shamik.

**Java Class and File's Contract**

***There can be only one public class per source file.***

***(TIP : Best practice...use one class for each file. )***

* A source file can have multiple **non-public** classes.
* The **public class name should be the name of the source file** as well which should be appended by **.java** at the end. For example: the class name is *public class Teacher{}* then the source file should be as Teacher.java.
* If the class is defined inside a package, then the package statement should be the first statement in the source file. **Package is used for avoid namespce collition**
* If i**mport statements are present, then they must be written between the package statement and the class declaration**. If there are no package statements, then the **import statement should be the first line in the source file**. Import says about which classes we are going to use. Remember when a class is outside of the package of **declaration class** for that class **import** statement required.
* Import and package statements will imply to all the **classes** present in the source file. It is not possible to declare different **import and/or package** statements to different classes in the source file.

**DataTypes**

**There are two data types available in Java:**

* Primitive Data Types, in**t,char,float,double,boolean,short,long-- default value is 0**
* Reference/Object Data Types

Reference variables are created using defined **constructors of the classes**. They are used to access objects. These variables are declared to be of a specific type that cannot be changed. For example, Student, Teacher etc.

* Class objects, and various type of array variables come under reference data type.
* Default value of any reference variable is **null**.
* A reference variable can be used to refer to any object of the declared type or any compatible type.
* Example: Animal animal = new Animal("Tiger");

A literal is a source code representation of a fixed value. They are represented directly in the code without any computation.

Literals can be assigned to any primitive type variable.

int a=10;

char a='A' // here A is literal

String literal

String name=”Shamik” // here Shamik is literal.

**Java Modifiers:**

**Java has two kind of modifiers**

1. Java Access Modifiers.

2. Non access Modifiers.

**Java Access Modifiers:**

**Java has 3 Access modifiers but 4 access level**.

3 modifiers are **private,protected,public**

4 access levels are **private,protected,public,default**

If we not mention any access modifiers then it is known as default in java.

Ex : **String name**; or **String getName()**

**Private :** Most restricted modifiers only accessible in the class which creates it.

**Default**: Accessible by own class and by the other classes of same package of its owner class.

**Protected**: Accessible by own class and by the other classes of same package of its owner class. All subclasses of its owner class in different package **via Inheritance**.

**Public**: Accessible by all.

**Visibility Matrix**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Own class | Own package | Subclass in Defferent Package | Defferent Package |
| Private | Yes | No | No | No |
| Default | Yes | Yes | No | No |
| Protected | Yes | Yes | Yes(**Inheritence**) | No |
| Public | Yes | Yes | Yes | Yes |

**Non access Modifiers**

mainly in java we have following non access modifiers

**1. Static**

**2. Final**

**3. Volatile**

**4. Synchronize**

**Static :** Static is a java keyword we use it for class level variables or methods. When we need an property or method which we want to invoke without object/instance we make them static remember static method and property is not associated with object /instance .it

associated with class and **only one copy of this static property/method is share among all objects of this class.**

Let’s take an example**, We want to count number of objects created from a class.**

We know when a object is created constructor is called so If we use a Static counter and for each object creation we increase it we can number of object created.

Here is the program

**package** com.example.staticTest;

**public** **class** ObjectTracking {

**public** **static** **int** *COUNTER*=0;

**public** ObjectTracking()

{

*COUNTER*++;

}

**public** **static** **void** displayCount()

{

System.***out***.println("Number of Objects in JVM " + *COUNTER*);

}

**public** **static** **void** main(String[] args) {

**for**(**int** i=0;i<10;i++)

{

**new** ObjectTracking();

}

ObjectTracking.*displayCount*();//call *displayCount as static*

}

}

Output: Number of Objects in JVM 10

Please pay attention to the line ObjectTracking.*displayCount*()

we call *displayCount*() method without create a object but with Class name

it is possible because static is attach to class not object.

Please please remember

to call static we need classname no need object refrence

**<Class Name>.<Static Method/Property>**

**We also call it through object reference but it is a bad practice and compiler will change reference to class name don't get confused it is not object's property.**

Class StaticTest

{

public static String CLASS\_NAME=”StaticTest”

public static void main(String[] args) {

StaticTest test = new StaticTest();

System.out.println(test. CLASS\_NAME);// test will replace by Static Test

}

}

**Here test. CLASS\_NAME replaced by StaticTest.CLASS\_NAME**

**Good practice is always creating Static variable name in Capital Letters**

**if multi word then put \_ beetween them**

**so CLASS\_NAME is good practice but className or CLASSNAME is bad practice.**

**Final: Final is a keyword in java. It can be used in three level**

**a. Variable and Reference**

**b. Method**

**c. Class**

**Variable and Reference:**

**A final variable can be explicitly initialized only once. A reference variable declared final can never be reassigned to refer to an different object. However, you can change the state of the object. So don't fool with this.**

***TIP: You can't reassign reference variable to another object but you can change the object State.***

With variables, the *final* modifier often is used with *static* to make the constant a class variable.

## Method:

A final method cannot be overridden by any subclasses. final modifier prevents a method from being modified in a subclass.

**The main intention of making a method final would be that the content of the method should not be changed by any outside class**.

## Class: A final class cannot be extended. If a class is marked as final, then no class can inherit any feature from the final class.

Example :

**package** com.example.finalTest;

**import** java.util.ArrayList;

**import** java.util.List;

**public** **final** **class** FinalTest {

**public** **final** **int** score =10;

**public** **final** List<String> greet = **new** ArrayList<String>();

**public** **final** **void** reassign()

{

// score=80; it can't be reassign as it is final

// greet = new ArrayList<String>(); new ArrayList<String>();

greet.add("able to change the arraylist state");

}

}

Here **FinalTest** class marked as final so no one can extend it and allow to change the content of the class.

Method **reassign** is final so it can't be overridden in subclass (if FinalTest is not final.)

Score is final and assign a value 10. So it can't be reassigning again if you do that compiler will complain.

Volatile and Synchronize will be discussed later. When we will discuss about Thread.

**Java Basic Operators**

Java provides a rich set of operators to manipulate variables. We can divide all the Java operators into the following groups:

* Arithmetic Operators
* Relational Operators
* Bitwise Operators
* Logical Operators
* Assignment Operators
* Misc Operators

**Arithmetic Operators:**

**If A=20 qnd B=10 then**

|  |  |
| --- | --- |
| 1 | **+ ( Addition )**  Adds values on either side of the operator  **Example:** A + B will give 30 |
| 2 | **- ( Subtraction )**  Subtracts right hand operand from left hand operand  **Example:** A - B will give 10 |
| 3 | **\* ( Multiplication )**  Multiplies values on either side of the operator  **Example:** A \* B will give 200 |
|  |  |
| 4 | **/ (Division)**  Divides left hand operand by right hand operand  **Example:** A/B will give 2 |
| 5 | **% (Modulus)**  Divides left hand operand by right hand operand and returns remainder  **Example:** A % B will give 0 |
| 6 | **++ (Increment)**  Increases the value of operand by 1  **Example:** B++ gives 11 |
| 7 | **-- ( Decrement )**  Decreases the value of operand by 1  **Example:** B-- gives 9 |

* **Relational Operators:**

**If A=20 qnd B=10 then**

|  |  |
| --- | --- |
| 1 | **== (equal to)**  Checks if the values of two operands are equal or not, if yes then condition becomes true.  **Example:** (A == B) is not true. |
| 2 | **!= (not equal to)**  Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.  **Example:** (A != B) is true. |
| 3 | **> (greater than)**  Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.  **Example:** (A > B) is true. |
| 4 | **< (less than)**  Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.  **Example:** (A < B) is not true. |
| 5 | **>= (greater than or equal to)**  Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.  **Example** (A >= B) is true. |
| 6 | **<= (less than or equal to)**  Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.  **example**(A <= B) is not true. |

## Bitwise Operators:

Java defines several bitwise operators, which can be applied to the integer types, long, int, short, char, and byte.

Bitwise operator works on bits and performs bit-by-bit operation. Assume if a = 60; and b = 13; now in binary format they will be as follows:

a = 0011 1100

b = 0000 1101

-----------------

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a  = 1100 0011

The following table lists the bitwise operators:

Assume integer variable A holds 60 and variable B holds 13 then:

|  |  |
| --- | --- |
| SR.NO | Operator and Description |
| 1 | **& (bitwise and)**  Binary AND Operator copies a bit to the result if it exists in both operands.  **Example:** (A & B) will give 12 which is 0000 1100 |
| 2 | **| (bitwise or)**  Binary OR Operator copies a bit if it exists in either operand.  **Example:** (A | B) will give 61 which is 0011 1101 |
| 3 | **^ (bitwise XOR)**  Binary XOR Operator copies the bit if it is set in one operand but not both.  **Example:** (A ^ B) will give 49 which is 0011 0001 |
| 4 | **~ (bitwise compliment)**  Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.  **Example:** (~A ) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number. |
| 5 | **<< (left shift)**  Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand  **Example:** A << 2 will give 240 which is 1111 0000 |
| 6 | **>> (right shift)**  Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.  **Example:** A >> 2 will give 15 which is 1111 |
| 7 | **>>> (zero fill right shift)**  Shift right zero fill operator. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros.  **Example:** A >>>2 will give 15 which is 0000 1111 |

## Logical Operators

**If A=20 and B=10**

|  |  |
| --- | --- |
| Operator | Description |
| 1 | **&& (logical and)**  Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.  **Example** (A && B) is false. |
| 2 | **|| (logical or)**  Called Logical OR Operator. If any of the two operands are non-zero, then the condition becomes true.  **Example** (A || B) is true. |
| 3 | **! (logical not)**  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.  **Example** !(A && B) is true. |

## Assignment Operators:

There are following assignment operators supported by Java language:

|  |  |
| --- | --- |
| SR.NO | Operator and Description |
| 1 | **=**  Simple assignment operator, Assigns values from right side operands to left side operand.  **Example:** C = A + B will assign value of A + B into C |
| 2 | **+=**  Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand.  **Example:** C += A is equivalent to C = C + A |
| 3 | **-=**  Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand.  **Example:**C -= A is equivalent to C = C - A |
| 4 | **\*=**  Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand.  **Example:** C \*= A is equivalent to C = C \* A |
| 5 | **/=**  Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand  **Example**C /= A is equivalent to C = C / A |
| 6 | **%=**  Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand.  **Example:** C %= A is equivalent to C = C % A |
| 7 | **<<=**  Left shift AND assignment operator.  **Example**C <<= 2 is same as C = C << 2 |
| 8 | **>>=**  Right shift AND assignment operator  **Example** C >>= 2 is same as C = C >> 2 |
| 9 | **&=**  Bitwise AND assignment operator.  **Example:** C &= 2 is same as C = C & 2 |
| 10 | **^=**  bitwise exclusive OR and assignment operator.  **Example:** C ^= 2 is same as C = C ^ 2 |
| 11 | **|=**  bitwise inclusive OR and assignment operator.  **Example:** C |= 2 is same as C = C | 2 |

## Conditional Operator ( ? : )

Conditional operator is also known as the ternary operator. This operator consists of three operands and is used to evaluate Boolean expressions. The goal of the operator is to decide which value should be assigned to the variable. The operator is written as:

***Tip: variable x = (expression) ? value if true : value if false***

**Example**

**package com.example.operator;**

**public** **class** OperatorExample {

**int** a=20;

**int** b=10;

**public** **void** airithMeticOperation()

{

**int** result = a+b;

System.***out***.println("airithMeticOperation is"+ result);

}

**public** **void** relationalOperation()

{

**boolean** result = a>b;

System.***out***.println("relationalOperation is"+ result);

}

**public** **void** bitwiseOperation()

{

**int** result = a|b;

System.***out***.println("bitwiseOperation is"+ result);

}

**public** **void** logicalOperation()

{

**boolean** result =(a>10 && b<=10);

System.***out***.println("logicalOperation is"+ result);

}

**public** **void** assignmentOperation()

{

a += 10;

System.***out***.println("assignmentOperation is"+ a);

}

**public** **void** conditionalOperation()

{

**int** result = a>20?a:b;

System.***out***.println("conditionalOperation is"+ result);

}

**public** **static** **void** main(String[] args) {

OperatorExample example = **new** OperatorExample();

example.airithMeticOperation();

example.relationalOperation();

example.bitwiseOperation();

example.logicalOperation();

example.assignmentOperation();

example.conditionalOperation();

}

}

**Output :**

airithMeticOperation is30

relationalOperation istrue

bitwiseOperation is30

logicalOperation istrue

assignmentOperation is30

conditionalOperation is30

**Loop & Array**

There may be a situation when you need to execute a block of code several number of times. Suppose you have to print 1 to 10000.

Programming languages provide loop for that.

A **loop** statement allows us to execute a statement or group of statements multiple times and following is the general form of a loop statement in most of the programming languages:

There is three type of Loop in java

1. **for Loop**

2. **while Loop**

3. **do while loop**

**For Loop:**

Syntax of for loop

for(initialization; boolean expression or condition ;increment or decrement)

{

//block of code

}

Here is the sequnce flow of for loop

The **initialization** step is executed first, and **only once.** This step allows you to declare and initialize any loop control variables. a semi colon (;) use for separator

* Next, the **Boolean expression** is evaluated. **If it is true**, the body of the loop is executed. If it is **false**, the body of the loop will not be executed and control jumps to the next statement after the for loop.
* After the **body** of the for loop gets executed, the control jumps back up to the increment/decrement statement. This statement allows you to update any loop control variables. **This statement can be left blank with a semicolon at the end**.
* The Boolean expression is now evaluated again. If it is true, the loop executes and the process repeats (body of loop, then update step, then Boolean expression). After the Boolean expression is false, the for loop terminates.

Example

Print 1 to 100 using for loop

public class LoopTest {

public static void main(String args[]) {

for(int x = 10; x < 100; x++) {

System.out.println("value of x : " + x );

}

}

}

**Each style For Loop**

From java 5 onwards we can use each style for loop. Here we don't need to initialize value or increment or decrement variables. We just have to create a temporary variable of the type of Collection or Array(discussed later) that variable holds the value for current Iteration refrence.

Syntax

for(variable type of collection element : collection)

{

//Statement

}

Example :Print Integer array using each type for loop

**package** com.example.loop;

**public** **class** EachStyleLoop {

**public** Integer[] arr=**new** Integer[]{10,20,30,40,50};

**public** **static** **void** main(String args[]) {

EachStyleLoop obj =**new** EachStyleLoop();

**for**(Integer i : obj.arr)

System.***out***.println("value of Integer is : " + i);

}

}

Here **obj.arr** is type of Integer array so we create a temporary variable **i** of type Integer , Now for each element i will point that element and print its value.

**While Loop:**

A while loop statement in java programming language repeatedly executes a target statement as long as a given condition is true.

The syntax of a while loop is:

while(boolean\_expression==true)

{

//block of code

}

Here, **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression, and true is any non-zero value.

When executing, if the *boolean\_expression* result is true, then the actions inside the loop will be executed. This will continue as long as the expression result is true.

Example: Print Even numbers between 1 to 100

**package** com.example.loop;

**public** **class** WhileLoop {

**public** **void** displayEven()

{

**int** i=0;//initialization

**while**(i<=100)

{

**if**(i%2==0)

{

System.***out***.println("Even Number is " + i);

}

i++;//increment

}

}

**public** **static** **void** main(String[] args) {

WhileLoop obj =**new** WhileLoop();

obj.displayEven();

}

}

**do While Loop:**

A do...while loop is similar to a while loop, except that a do while loop is guaranteed to execute at least one time as condition is tested after body finishes,

Syntax:

do

{

//Statements

}while(boolean\_expression);

**Please note that the Boolean expression appears at the end of the loop, so the statements in the loop execute once before the Boolean is tested.**

If the Boolean expression is true, the control jumps back up to do statement, and the statements in the loop execute again. This process repeats until the Boolean expression is false.

Example: Demo Account act as a real account for first time publicity purpose

**package** com.example.loop;

**public** **class** DoWhileLoop {

**private** **boolean** isDemoAccount=**true**;

**public** **void** executAccount()

{

**do**

{

System.***out***.println("Although a Demo account For publicity it will act as real account for first time");

}

**while**(!isDemoAccount);

}

**public** **static** **void** main(String[] args) {

DoWhileLoop obj = **new** DoWhileLoop();

obj.executAccount();

}

}

**Arrays:**

As of now we just explore single valued property. If we have to deal with multivalued properties? then an **array is answer it is a data structure,** 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**.

**Why array is so handy?**

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

**Array declaration**

**You can declare an array in two ways**

* 1. <datatype>[] arrayRef; // best practice . as you can tell by seeing datatype it is an array
  2. <datatype> arrayRef[];// not wrong but avoid to use

**Array Creation**

There is two ways to create array

* + 1. Array declaration with post initialization.
    2. Array declaration with initialization.

**Array declaration with post initialization:**

<datatype>[] arr = new <dataType>[size];

Integer[] arr = new Integer [10];// It will create a Integer array with size 10.

The above statement does two things:

* It creates an array with size 10 using new Integer [10];
* It assigns the reference of the newly created array to the variable arr.

***TIP: Please remember when you declare array in above way you mist have to provide size, which is always in right side. Don’t ever put it to left side or left it blank it is wrong way to declare array.***

***So Integer[] arr = new Integer[10];// is right***

***But Integer[10] arr = new Integer[];// is wrong***

***Integer[] arr = new Integer[];// is wrong***

**Array declaration with initialization:**

<datatype>[] arr = {value0, value1, ..., valuek};//initialization value

String[] arr = {“Shamik”,”Swastika”,”Samir”} //etc

Integer[] arr = {10,20,30,40} //etc

**Multi Dimension Array:**

An Array can be multi dimension so it can be 2D or 3D or ND array where D refer the dimension.

A 2D Integer array with size 5X5, we can explain in following term

1. An Integer 2D array which has 5 elements where each element is an Integer array
2. Each element array contains 5 integer value

By java we can declare it with

Integer [][] arr = new Integer[5][5];

So Integer[][] arr = new Integer[2][2];

**Pictorial view is**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | arr[0][0] | arr[0][1] | | |  |  | | --- | --- | | arr[1][0] | arr[1][1] | |

Integer 2D array of Size 2 with name arr

One thing to remember array is Zero-based index i.e. it starts from zero so if an array with size 10 then it has 10 cells with number 0 to 9.

We can access a cell through it’s index position

So arr[0][0] point to First cell of 2D array and then first cell of element array under the 2D array

Similarly So arr[1][0] point to Second cell of 2D array and then first cell of element array under the 2D array.

***TIP: when you initialize an array like Integer[] arr = new Integer[5]; It creates a arrar with length 5 and its each cell will populate to default value of data type so in this case it is 0. If it is a String array or any other Object array then it’s each cell value will be null;***

***Here arr[0] gives 0 if String arr[0] gives null.***

***One more thing to populate or retrieve value for each cell you have to traverse through each cell. So For loop is required for that general rule is Number of dimension increase number of inner for loop is increased to populate value***

***Examaple : A program populate and retrieve value from 2D array***

**package** com.example.arrytest;

**public** **class** TwoDArrayTest {

Integer[][] arr = **new** Integer[2][2];

**public** **void** insertion()

{

**for**(**int** i=0;i<2;i++)

{

**for**(**int** j=0;j<2;j++)

{

**int** res =i\*j+1;

arr[i][j]=res;

System.*out*.println("value Inserted in cell arr["+i+"]["+j+"] is " + res);

}

}

}

**public** **void** traversal()

{

**for**(**int** i=0;i<2;i++)

{

**for**(**int** j=0;j<2;j++)

{

**int** res =arr[i][j];

System.*out*.println("value retrive from cell arr["+i+"]["+j+"] is " + res);

}

}

}

**public** **static** **void** main(String[] args) {

TwoDArrayTest obj = **new** TwoDArrayTest();

obj.insertion();

System.*out*.println("GOING TO RETRIVE");

obj.traversal();

}

}

Output :

value Inserted in cell arr[0][0] is 1

value Inserted in cell arr[0][1] is 1

value Inserted in cell arr[1][0] is 1

value Inserted in cell arr[1][1] is 2

GOING TO RETRIVE

value retrive from cell arr[0][0] is 1

value retrive from cell arr[0][1] is 1

value retrive from cell arr[1][0] is 1

value retrive from cell arr[1][1] is 2

**Pass an Array to Method**

Just as you can pass primitive type values to methods, you can also pass arrays to methods.

For example, the following method print the elements in an int array:

package com.example.arrytest;

public class PassArrayToMethod {

public void display(int[] array) {

for (int i = 0; i < array.length; i++) {

System.*out*.print(array[i] + " ");

}

}

public static void main(String[] args) {

int[] arr = {1,5,8,11,14};

PassArrayToMethod obj = new PassArrayToMethod();

obj.display(arr);

}

}

**Return Array from Method**

A method may also return an array.

Example : the method shown below returns an array that is the reversal of another array:

**package** com.example.arrytest;

**public** **class** ReturnArrayFromMethod {

**public** **int**[] reverse(**int**[] arr) {

**int**[] reverseArr = **new** **int**[arr.length];

**for** (**int** i = 0, j = reverseArr.length - 1; i < arr.length; i++, j--) {

reverseArr[j] = arr[i];

}

**return** reverseArr;

}

**public** **void** print(**int**[] arr)

{

**for** (**int** i = 0;i <arr.length;i++) {

System.*out*.print(arr[i]+ " ");

}

System.*out*.println();

}

**public** **static** **void** main(String[] args) {

ReturnArrayFromMethod obj =**new** ReturnArrayFromMethod();

**int**[] arr={1,2,3,4};

obj.print(arr);

**int**[] res = obj.reverse(arr);

obj.print(res);

}

}

***TIP: Always remember Array itself is Object. An array of primitives mean an array holds primitive elements but array itself is an Object***

**Arrays Class:**

Arrays is an helper class which java provides to do some operations on Array easily .The **java.util.Arrays** class contains various static methods for sorting and searching arrays, comparing arrays, and filling array elements. These methods are overloaded for all primitive types.

|  |  |
| --- | --- |
|  |  |
| 1 | **public static int binarySearch(Object[] a, Object key)**  Searches the specified array of Object ( Byte, Int , double, etc.) for the specified value using the binary search algorithm. The array must be sorted prior to making this call. This returns index of the search key, if it is contained in the list; otherwise, (-(insertion point + 1). |
| 2 | **public static boolean equals(long[] a, long[] a2)**  Returns true if the two specified arrays of longs are equal to one another. Two arrays are considered equal if both arrays contain the same number of elements, and all corresponding pairs of elements in the two arrays are equal. This returns true if the two arrays are equal. Same method could be used by all other primitive data types (Byte, short, Int, etc.) |
| 3 | **public static void fill(int[] a, int val)**  Assigns the specified int value to each element of the specified array of ints. Same method could be used by all other primitive data types (Byte, short, Int etc.) |
| 4 | **public static void sort(Object[] a)**  Sorts the specified array of objects into ascending order, according to the natural ordering of its elements. Same method could be used by all other primitive data types ( Byte, short, Int, etc.) |

**Java Decision making**

Decision making structures have one or more conditions to be evaluated or tested by the program, along with a statement or statements that are to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false

Java programming language provides following types of decision making statements.

* 1. **If statement**
  2. **If else statement**
  3. **If else if else**
  4. **Switch case**

**If Statement:**

An **if** statement consists of a Boolean expression followed by one or more statements.

Syntax :

if(Boolean expression==true)

{

//Statements

}

If the Boolean expression evaluates to true, then the block of code inside the if statement will be executed. If not the first set of code after the end of the if statement (after the closing curly brace) will be executed.

Example : If number grater that 9 then it is a two digit Number

**package** com.example.decision;

**public** **class** IfTest {

**public** **void** checkIf(**int** number)

{

**if**(number >9)

{

System.*out*.println("This is 2 digit Number");

}

}

**public** **static** **void** main(String[] args) {

IfTest obj =**new** IfTest();

obj.checkIf(10);

}

}

**If Else statement**

An **if** statement can be followed by an optional **else** statement, which executes when the Boolean expression is false. Like in above example if it is an one digit number then we can [print the same.

Syntax:

if(Boolean\_expression==true){

//Executes when the Boolean expression is true

}else{

//Executes when the Boolean expression is false

}

Example:

**package** com.example.decision;

**public** **class** IfElse {

**public** **void** checkIf(**int** number)

{

**if**(number >9)

{

System.*out*.println("This is 2 digit Number");

}

**else**

{

System.*out*.println("This is 1 digit Number");

}

}

**public** **static** **void** main(String[] args) {

IfElse obj =**new** IfElse();

obj.checkIf(8);

}

}

**If else if else**

An if statement can be followed by an optional *else if...else* statement, which is very useful to test various conditions using single if...else if statement.

***TIP*** : **When using if , else if , else statements there are few points to keep in mind.**

* **An if can have zero or one else's and it must come after any else if's.**
* **An if can have zero to many else if's and they must come before the else.**
* **Once an else if succeeds, none of the remaining else if's or else's will be tested.**

Example; Greet World according to time

**package** com.example.decision;

**public** **class** Greet {

**public** **void** greetWorld(**int** time)

{

**if**(time>0 && time <12)

{

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

}

**else** **if**(time>=12 && time <16)

{

System.*out*.println("Good Afternoon");

}

**else** **if**(time>=16 && time <18)

{

System.*out*.println("Good Evening");

}

**else**

{

System.*out*.println("Good Night");

}

}

**public** **static** **void** main(String[] args) {

Greet greet = **new** Greet();

greet.greetWorld(17);

}

}

**Switch Case:**

A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case.

Syntax:

switch(expression){

case value :

//Statements

break; //optional

case value :

//Statements

break; //optional

//You can have any number of case statements.

default : //Optional

//Statements

}

The following rules apply to a **switch** statement:

* The variable used in a switch statement can only be integers, convertible integers (byte, short, char), strings and enums
* You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
* The value for a case must be the same data type as the variable in the switch and it must be a constant or a literal.
* When the variable being switched on is equal to a case, the statements following that case will execute until a *break* statement is reached.
* When a *break* statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
* Not every case needs to contain a break. If no break appears, the flow of control will *fall through* to subsequent cases until a break is reached.
* A *switch* statement can have an optional default case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

Example: Greet world using Switch

**package** com.example.decision;

**public** **class** SwitchTest {

**public** **void** greetWorld(**int** time)

{

**if**(time>0 && time <12)

{

time =0;

}

**else** **if**(time>=12 && time <16)

{

time =12;

}

**else** **if**(time>=16 && time <18)

{

time =16;

}

**else**

{

time =124;

}

// set time to 0,12,16,24 as case only support constant or literal

**switch**(time)

{

**case** 0:

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

**break**;

**case** 12:

System.*out*.println("Good Afternoon");

**break**;

**case** 16:

System.*out*.println("Good Evening");

**break**;

**default**:

System.*out*.println("Good Night");

}

}

**public** **static** **void** main(String[] args) {

Greet greet = **new** Greet();

greet.greetWorld(17);

}

}

**Java Encapsulation:**

Encapsulation is one of the four fundamental OOP concepts. The other three are inheritance, polymorphism, and abstraction.

Encapsulation in Java is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit.

In encapsulation the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class, therefore it is also known as data hiding.

**Why Encapsulation?**

Why we need to hide your data from outside? It is because you don’t want your code will break by outsider intentionally. Let say you have a class Movie and Movie can only be seen if age is over 18 now so naturally you have class name **Movie** which has one property **age** and one method say **watchMovie**.

Now without Encapsulation class will look like

**package** com.example.encapsulation;

**public** **class** Movie {

**public** Integer age;

**public** **void** watchMovie()

{

**if**(age.intValue()>18)

{

System.*out*.println("Watching Movie");

}

**else**

{

System.*out*.println("You are not permitted to Watch Movie");

}

}

**public** **static** **void** main(String[] args) {

Movie movie=**new** Movie();

// intentionally I set the age to null

movie.age=**null**;

movie.watchMovie();

}

}

Pay attention to the line

movie.age=**null**;

here I set the age as null intentionally as I have direct access to property age.so an outcome I am able to break the code of Movie class by providing **null pointer exception**

No I will change the Movie class in a way so I can achieve encapsulation

To achieve encapsulation in Java

* Declare the variables of a class as private.
* Provide public setter and getter methods to modify and view the variables values.

Now class look like

**package** com.example.encapsulation;

**public** **class** Movie {

**private** Integer age;

**public** **void** watchMovie()

{

**if**(age.intValue()>18)

{

System.*out*.println("Watching Movie");

}

**else**

{

System.*out*.println("You are not permitted to Watch Movie");

}

}

**public** Integer getAge() {

**return** age;

}

**public** **void** setAge(Integer age) {

**this**.age = age;

}

**public** **static** **void** main(String[] args) {

Movie movie=**new** Movie();

// intentionally I did not set the age

movie.setAge(**null**);

movie.watchMovie();

}

}

So still I can break the code easily. So what the heck encapsulation doing here?

What benefits I got from make age property private and access the property through getter and setter.

Think??????????????

Actually by providing setter and getter you make a provision that you can provide validation when the value is set to class.

For say if we put a validation in setAge method ,if age is null then we set it to 0 ,then the program run flawlessly.

Also I can put it into watchMovie method but think if there are many methods in Movie class

And each uses age property then in each method you have to put same validation which is nothing but a hectic and repeatable work. Instead of that if we do it in setAge method it will be in single place.

So here is the beauty of Encapsulation always there is a provision for doing something on the property in future.

***TIP: Always go for encapsulation***

Updated Code

**package** com.example.encapsulation;

**public** **class** Movie {

**private** Integer age;

**public** **void** watchMovie()

{

**if**(age.intValue()>18)

{

System.*out*.println("Watching Movie");

}

**else**

{

System.*out*.println("You are not permitted to Watch Movie");

}

}

**public** Integer getAge() {

**return** age;

}

**public** **void** setAge(Integer age) {

**if**(age==**null**)

{

age=0;

}

**this**.age = age;

}

**public** **static** **void** main(String[] args) {

Movie movie=**new** Movie();

// intentionally I did not set the age

movie.setAge(**null**);

movie.watchMovie();

}

}