**Language Fundamentals:**

To design java applications, java has provided the following building blocks.

1. **Java Tokens**
2. **Java data types and type casting**
3. **Java Statements**
4. **Java Arrays**

**Java Tokens**

**Lexeme:**

The smallest logical unit in Java program is called **Lexeme**.

**Token:**

A collection of lexemes come under a group is called as **Token**.

**Example:**

for-Lexeme

Keywords🡪Token

if-Lexeme

int-Lexeme

**+** - lexeme

Operators🡪Token

**-** - lexeme

**/** - lexeme

**To design a Java application, Java technology has provided the following tokens..**

1. **Identifiers**
2. **Literals**
3. **Keywords/Reserved words**
4. **Operators**

**\*Identifiers\***

A “name” in Java program is called an “**Identifier**”, which can be used for identification purpose.

***What is Identifier?***

An **identifie**r identifies something in a programming language. It may be, in Java,

**a) A variable  
b) A method  
c) A class  
d) A package  
e) An interface**

So finally, an identifier, in Java, can be a variable or method or a class or interface or a package.

**Example**

|  |  |
| --- | --- |
| **int marks = 50;** | Here, **marks** is a variable identifier that identifies marks of a student |
| **public void calculate() { }** | Here, the method is identified as **calculate** that calculates something |
| **public class Test { }** | Here, the class is identified as **Test** which tests something in its code |

It can be class name/ label name/ variable name/ method name/ interfaces/ packages etc..,

**Example**:

Name of class

**class** **test**{

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

**int** **x**=10;

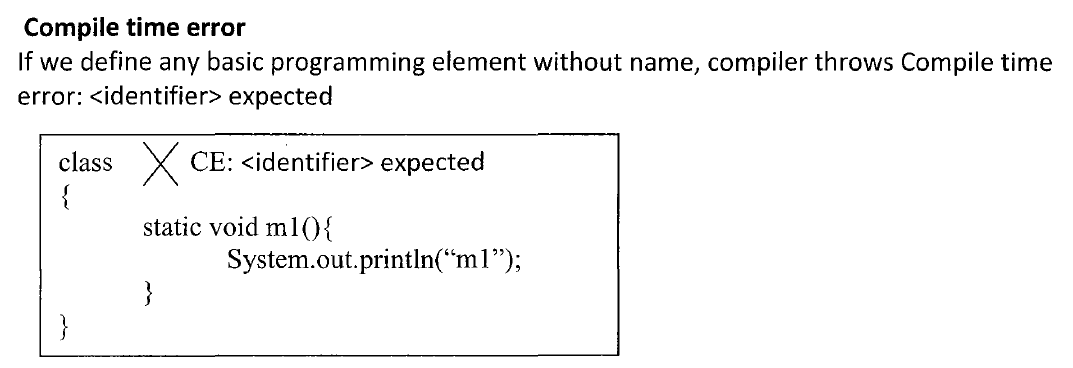
Name of Array

}  
}

Predefined class name

Name of Variable

Name of method



**Rules for defining Java Identifiers:**

**Rule 1 :**

**The only allowed characters in Java identifiers are:**

1. *a to z*
2. *A to Z*
3. *0 to 9*
4. *\_ -> Underscore*
5. *$ -> Dollar*

**Rule 2 :**

**If we are using any other character, we will get compile time error.**

**Example:**

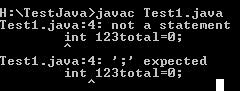
a)Totalnumber🡪 **Valid**

b) Total#--> **Invalid** - **Illegal Character**

**Rule 3 :**

**Identifiers can’t start with digit. It can start with an alphabet, \_ or $.**

**Example:**

1. total123🡪 **Valid**
   1. 123total 🡪 **Invalid** - **Illegal Character  
        
      Compilation Error:**
2. $bar🡪 **Valid**
3. \_emplAdd🡪 **Valid**
4. 9\_esal🡪 **Invalid** - **Illegal Character**

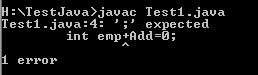
**Rule 4 :**

**Java identifiers will not allow all the special symbols like #, @, , etc.. and all operators like +,-,% etc..**

**Example:**

1. Total#123🡪 **Invalid - Illegal Character**
2. total@123 🡪 **Invalid - Illegal Character**
3. emp-sal🡪 **Invalid - Illegal Character**
4. emp+Add🡪 **Invalid - Illegal Character**

**Compilation Error:**



**Rule 5 :**

**Java identifiers are case sensitive, of-course, Java language itself is treated as case-sensitive programming language.**

**Example:**

***class*** *test{*

***int******number****=10;*

***int******NUMBER****=20;*

We can differentiate with case

***int******Number****=30;*

***int******NumBer****=40;*

*}*

**Rule 6 :**

**There is no length limit for java identifiers, but it is not recommended to take more than 15 lengths.**

**Example:**

String permanentEmployeeAddress=”Hyd”;

**🡪 Valid, but not recommended.**

String permEmpAdr=”Hyd”;**🡪 Valid and recommended.**

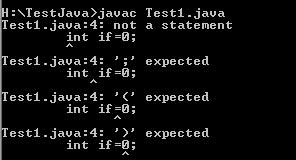
**Rule 7 :**

**We can’t use reserved words as identifiers.**

**Example :**

int if=10; 🡪 **Invalid**

**Compilation Error:**



**Rule 8:**

**All predefined Java class names and interfaces names, we can use as identifiers.**

**Example:**

***class*** *test{*

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

***int******String****=888;*

*System.out.println(****String****);*

***int******Runnable****=888;*

All are valid

*System.out.println(****Runnable****);*

***int******Exception****=888;*

*System.out.println(****Exception****);*

*}*

*}*

**Note:**

***class*** *test{*

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

***int******System****=888;*

***System.out****.println(****System****);*

*}*

*}*

**Compilation Error:**

***Once if we declare System as an integer variable then compiler will treat that system as an integer.***

***i.e,***

***int cannot be dereferenced***

***System.out.println(System);***

**Note:**

If in the above content, if we want to use System as original class then, we have to use fully qualified name.

**Example:  *class*** *test{*

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

***int*** *System=888;*

*java.lang.System.out.println(System);*

*}*

*}*

**Note:**

**Even though, it is legal to use class names and interface names as identifiers, but it is not good programming practice.**

**Note:**

**Identifiers should not be duplicated within the same scope.**

**Example 1 :**

***class*** *test{*

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

***int******i****=888;*

***float******i****=888.9f;*

*}*

*}*

**Compilation Error:   
*Both ‘i’ variables are declared at “class scope”.***

***i.e, variable i is already defined in method main(String()).***

**Example 2 :**

***class*** *test{*

***void add()*** *{*

***int******i****=888;*

***float******i****=888.9f;*

*}*

*}*

***Compilation Error:   
Both ‘i’ variables are declared at “method or local scope”.***

***i.e, variable i is already defined in method add.***

**Example 3:**

***class*** *test{*

***int******i****=888;*

***void add()*** *{*

***float******i****=888.9f;*

*}*

*}*

***No Compilation Error:   
Both ‘i’ variables are declared at different scope.***

**Note:**

In Java applications, all the identifiers must be meaningful. They should reflect a particular meaning.

**Example:**

***int xxx=”abc123”; 🡪 Valid but not suggestable.***

***int AccNo=”abc123”;🡪 Valid and suggestable.***

**Note 3 :**

If we have multiple words within a single identifier, then it is suggestable to separate multiple words with different notations like ‘\_’ symbol.

**Example:**

***String permEmpAddr=”Hyd”;🡪 Valid, but not suggestable***

***String perm\_Emp\_Addr=”Hyd”;🡪 Valid and Suggestable.***

***Q) Check which is valid identifier?***

*public class Test {*

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

*int \_a;-->* ***Valid***

*int $c;-->* ***Valid***

*int \_\_\_\_\_\_2\_w;-->* ***Valid***

*int \_$;-->* ***Valid***

*int this\_is\_a\_very\_long\_identifier;-->* ***Valid***

*int myIdentifier;-->* ***Valid***

*int $my\_identifier;-->* ***Valid***

*int $123;-->* ***Valid***

*}*

*}*

***Q) Which of the following are valid Java Identifiers ?***

*a) \_$\_ 🡪* ***Valid***

*b) Ca$h 🡪* ***Valid***

*c) Java2Share 🡪* ***Valid***

*d) all@hands 🡪* ***Invalid***

*e) 123abc🡪* ***Invalid***

*f) Total# 🡪* ***Invalid***

*g) Int 🡪* ***Valid***

*h) int 🡪* ***Invalid🡪 Reserved word****.*

*i) Integer 🡪* ***Valid (All predefined identifiers are valid)***

*j) total\_number 🡪* ***Valid***

*k) \_$\_$\_$\_$ 🡪* ***Valid***

*l) $\_$\_$\_$\_$ 🡪* ***Valid***

***Q) Choose the valid identifiers***

***A. aString🡪valid***

***B. $byte🡪 valid***

***C. ints🡪valid***

***D. $234🡪valid***

***E. finalized🡪valid***

***Q) How to find if the character is valid to be present as identifier or not?***

##### A. boolean Java.lang.[Character](eclipse-javadoc:%E2%98%82=test12/D:%5C/Oracle%5C/Middlewar10.3.5%5C/jrockit_160_24_D1.1.2-4%5C/jre%5C/lib%5C/rt.jar%3Cjava.lang(Character.class%E2%98%83Character).isJavaIdentifierPart(char ch)

##### This determines if the specified character may be part of a Java identifier as other than the first character.

##### *Example*

***class*** *Test{*

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

***System.out.println(Character.isJavaIdentifierPart(':'));🡪False***

***System.out.println(Character.isJavaIdentifierPart('#'));🡪 False***

***System.out.println(Character.isJavaIdentifierPart('$'));🡪True***

*}*

##### *}*

##### *Following are the fields related to JavaIdentifier for Java.lang.Character class :*

1. **Java.lang.Character.isIdentifierIgnorable(char ch)**

This determines if the specified character should be regarded as an ignorable character in a Java identifier or a Unicode identifier.

**Example:**

*public class Test {*

*public static void main(String[] args) {*  ***System.out.println(Character.isIdentifierIgnorable('\u0000'));//true***

***System.out.println(Character.isIdentifierIgnorable(':'));//false***

***System.out.println(Character.isIdentifierIgnorable('$'));//false***

*}  
 }*

**Note:**

**The following Unicode characters are ignorable in a Java identifier or a Unicode identifier:**

***ISO control characters that are not whitespace***

***'\u0000' through '\u0008'***

***'\u000E' through '\u001B'***

***'\u007F' through '\u009F'***

1. **java.lang.Character.isIdentifierIgnorable(int codePoint)**

**codePoint** - the character (Unicode code point) to be tested

This determines if the specified character (Unicode code point) should be regarded as an ignorable character in a Java identifier or a Unicode identifier.

**Example:**

***public******class*** *Test {*

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

***int*** *cp1 = 0x008f, cp2 = 0x0abc;*

***boolean*** *b1 = Character.isIdentifierIgnorable(cp1);*

***boolean*** *b2 = Character.isIdentifierIgnorable(cp2);*

*System.out.println( b1 );//true*

*System.out.println( b2 );//false*

*}*

*}*

1. **java.lang.Character.isJavaIdentifierPart(char ch)**

This determines if the specified character may be part of a Java identifier as other than the first character.

**Example:**

***public class Test {***

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

***char ch1 = '3', ch2 = '\_', ch3 = '#';***

***boolean b1 = Character.isJavaIdentifierPart(ch1);***

***boolean b2 = Character.isJavaIdentifierPart(ch2);***

***boolean b3 = Character.isJavaIdentifierPart(ch3);***

***System.out.println( b1 );//True***

***System.out.println( b2 );//True***

***System.out.println( b3 );//False***

***}***

***}***

1. **java.lang.Character.isJavaIdentifierPart(int codePoint)**

This determines if the character (Unicode code point) may be part of a Java identifier as other than the first character.

**Example:**

***public class Test {***

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

***int ch1=8, ch2=9;***

***boolean b1 = Character.isJavaIdentifierPart(ch1);***

***boolean b2 = Character.isJavaIdentifierPart(ch2);***

***System.out.println( b1 );//true***

***System.out.println( b2 );//false***

***}***

***}***

1. **java.lang.Character.isJavaIdentifierStart(char ch)**

This determines if the specified character is permissible as the first character in a Java identifier.

**Example:**

*public class Test{*

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

*System.out.println(Character.isJavaIdentifierStart('a'));//true System.out.println(Character.isJavaIdentifierStart('$'));//true*

*System.out.println(Character.isJavaIdentifierStart('#'));//false*

*}*

*}*

1. **java.lang.Character.isJavaIdentifierStart(int codePoint)**

This determines if the character (Unicode code point) is permissible as the first character in a Java identifier.

**Example:**

*public class Test{*

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

*System.out.println(Character.isJavaIdentifierStart(0));//false*

*System.out.println(Character.isJavaIdentifierStart(1));//false*

*System.out.println(Character.isJavaIdentifierStart(2));//false*

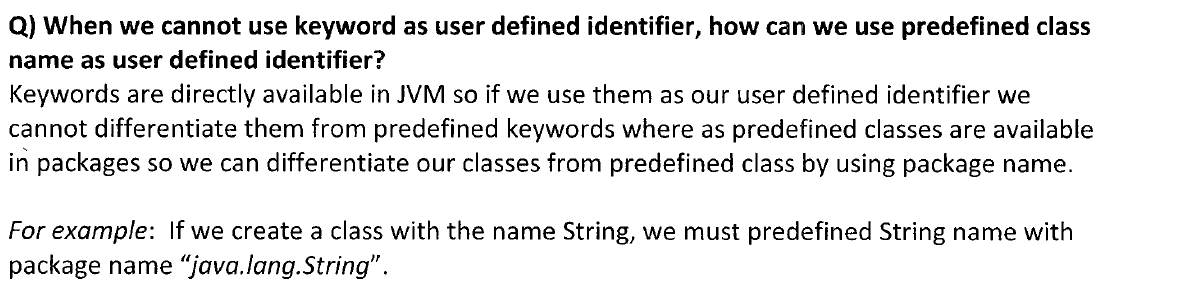
*}*

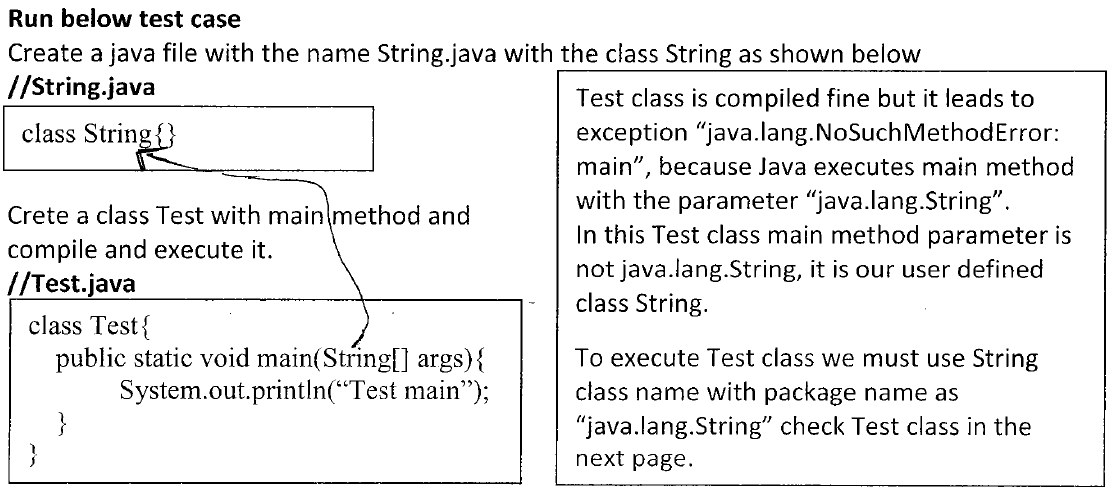
*}*

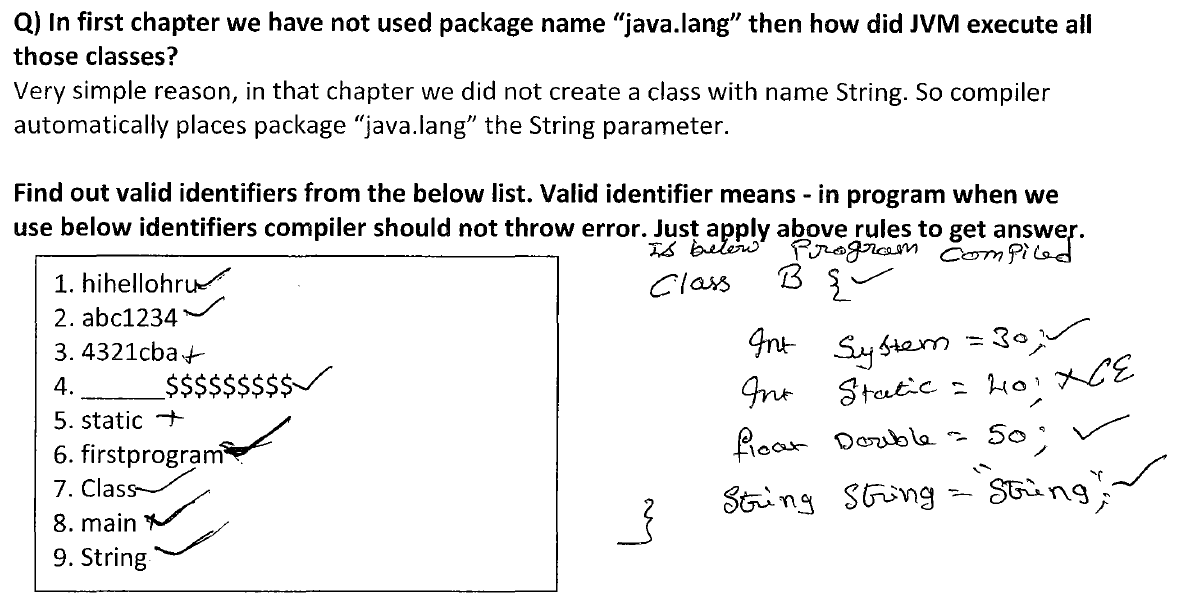
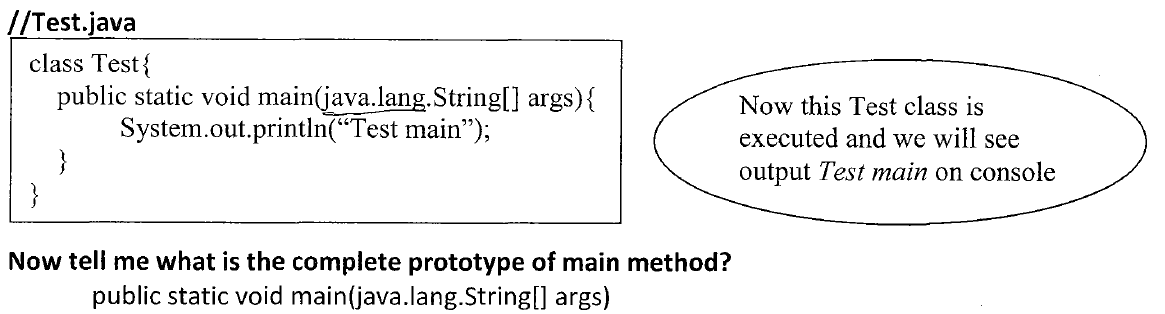
***Summary:***

1. *Character.isJavaIdentifierStart(int codePoint)*
2. *Character.isJavaIdentifierStart(char ch)*
3. *Character.isJavaIdentifierPart(int codePoint)*
4. *Character.isJavaIdentifierPart(char ch)*
5. *Character.isIdentifierIgnorable(int codePoint)*
6. *Character.isIdentifierIgnorable(char ch)*

**Note: Identifiers are the names of variables and variables are storage locations of data. Variables point to the memory location where data is read and modified.**





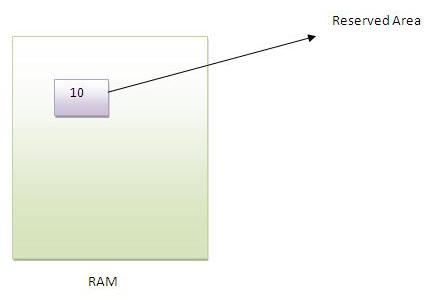


**Variable**

Variable is a container that holds the values that are used in the Java Program.

Variable is a name of reserved area allocated in memory.

To declare more than one variable of the specified type, you can use a comma-separated list.



**Syntax:**

***datatype variable [ = value][, variable [= value] ...] ;***

**Example:**

*int data=50;****//Here data is variable.***

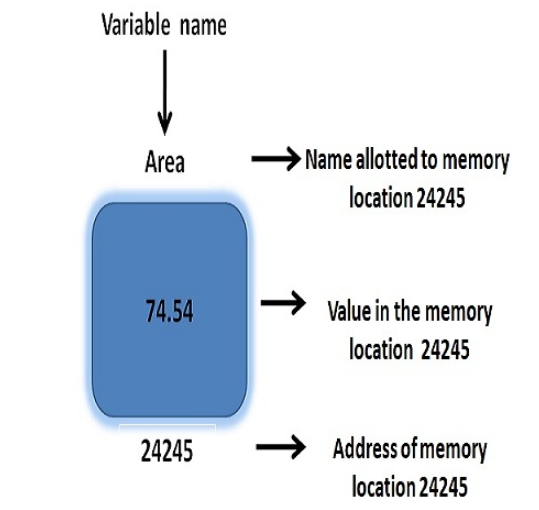
*int a, b, c;*  ***// Declares three ints, a, b, and c.***

*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’;* ***// char variable is initialized with values ‘a’.***



***Types of Variable***

**There are two types of variables, i.e.**

1. **Division 1:   
     
   Based on type of value represented by a variable:**All variables are divided into two types:

**a) Primitive Variables:**

It can be used to represent primitive values.  
 **Example:** int x=10;

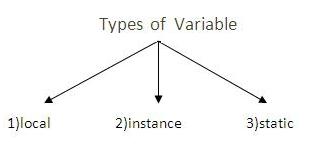
1. **Reference Variables:**

It can be used to refer objects.

**Example:** Student s=new Student();  
  
  
S

1. **Division 2:   
     
   Based on position of declaration and behavior:**

|  |
| --- |
| All the variables are divided into three types in Java:   * ***Local variable.*** * ***Instance variable.*** * ***Static variable.*** |



**Local Variable:**

Sometimes to meet temporary requirements of the programmer, we can declare variables inside a method or block or constructor, such type of variables are called **local variable/ temporary variable/ stack variable/ automatic variables**.

**Example :***m1(){  
 int x=10;  
}*

**Example :***static(){  
 int x=10;  
}*

**Example :***Test(){  
 int x=10;  
}*

**Example :***for(int i=0;i<10;i++){  
…  
}*

* Local variables will be stored inside **stack memory**.
* Local variables will be created, while executing the block, in which we declared it.
* Once block execution is completed, automatically local variable will be destroyed. Hence, the **scope of local variable is the “block”** in which we declared it.

**Example:  
  
*class*** *Test{*

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

***int*** *i=0;*

***for****(****int*** *j=0;j<3;j++){*

*i=i+j;*

*}*

***System.out.println(i+"..."+j);***

*}*

*}*

**CE: Cannot find the symbol: variable j, location: class Test.**

**Example:**

***class*** *Test{*

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

*{*

***try****{*

***int*** *j=Integer.parseInt("ten");*

*}****catch****(NumberFormatException){*

***j=10; //Invalid***

*}*

***System.out.println(j);//Invalid***

*}*

*}*

*}*

**CE: Cannot find the symbol: variable j, location: class Test.**

|  |
| --- |
|  |
| **“Access modifiers cannot be used for local variables.”**   * Local variables are visible only within the declared method, constructor or block. * Local variables are implemented at stack level internally. * There is **no default value for local variables**, so local variables should be declared and an initial value should be assigned before the first use.   **Example**:  ***class*** *Test{*  ***public******static******void*** *main(String[] args) {*  ***int*** *x;*  *System.out.println("Hello");*  *}*  *}* **O/p: Hello**  **No Error here, as x local variable is not used at-all.**  **Example**:  ***class*** *Test{*  ***public******static******void*** *main(String[] args) {*  ***int*** *x;*  *System.out.println(x);*  *}*  *}*  **CE: Test.Java:4:variable number might not have been initialized**  **Example:  *class*** *Test{*  ***public******static******void*** *main(String[] args) {*  ***int*** *x;*  ***if****(args.length>0){*  *x=10;*  *}*  *System.out.println(x);*  *}*  *}*  **CE: Test.Java:4:variable number might not have been initialized**  **Example:**  ***class*** *Test{*  ***public******static******void*** *main(String[] args) {*  ***int*** *x;*  ***if****(args.length>0){*  *x=10;*  *}****else****{*  *x=20;*  *}*  *System.out.println(x);*  *}*  *}*  **Output:**  **Java Test A B o/p: 10  Java Test o/p: 20**  **Note:**  It is not recommended to perform initialization for local variables inside logical blocks, because there is no gaurantee for the execution of these blocks always at run time.  **Note:**  It is highly recommended to perform initialization for local variables at the time of declaration, atleast with default values.  **Note:**  The only applicable modifier for local variables is “**Final**”. By mistake if we are trying to apply any other modifier, then we will get compile time eror.  **Example:**  **class** Test{  **int** x=10;  **static** **int** *y*=10;  **public** **static** **void** main(String[] args) {  **int** z=10;  }  }   |  | | --- | | Public | | Private | | <default> | | Protected |   Final  **Example:**  For local variable only “**final**” is applicable, if we try any other modifier, we will get compile time error.  *class Test{*  *public static void main(String[] args) {*  *public int a=10;//Invalid*  *private int b=10;//Invalid*  *protected int c=10;//Invalid*  *static int d=10;//Invalid*  *transient int e=10;//Invalid*  *volatile int f=10;//Invalid*  *final int g=10;//Valid*  *}*  *}*  **O/p:**  Illegal modifier for parameter a; only final is permitted  **Note**:  **If we are not declaring with any modifier, then by default it is “<default>” modifier. But, this rule is applicable only for instance and static variables, but not for local variables.**  **Example:**  ***class*** *Test{*  ***int*** *x=10;//<default> modifier*  ***static******int*** *y=20;//<default> modifier*  ***public******static******void*** *main(String[] args) {*  ***int*** *x=10****;//No <default> modifier***  *}*  *}*  ***Why we can't use access specifiers for variables declared inside method in a Java Class?***   1. Because it doesn't make sense. Variables declared in a method are local to the method; i.e. they can't be accessed outside the method. What would modifying the variable's declaration achieve?   **class Test {**  **public static void main(String[] args) {**  **private int x = 10;**  **public int y = 20;**  **protected int z = 30;**  **static int w = 40;**  **final int i = 50;**  **}**  **}**  **The only applicable modifier is final here; for other modifiers, the program gives compiler errors. Why is that? Please explain in detail.**   1. In short - none of the other modifiers make sense in that context. Saying a variable is public, private, protected, or static simply doesn't make sense in the context of a local variable that will go out of scope (and be garbage collected) once the method exits. Those modifiers are intended for class *fields* (and methods), to define their visibility (or in the case of static, their scope).   final is the only one that makes sense in the context of a local variable because all it means is that the variable cannot be modified after its initial declaration, it has nothing to do with access control. |
|  |

**Instance Variable:**

If the values of a variable are varied from Object to Object, such type of variables are called “**Instance Variable**”.

For every object a separate copy of instance variables will be created.

**Example:**

Class Student{  
 string name;  
 int rollno;  
 …  
}

|  |
| --- |
| A variable that is declared inside the class, but outside of any method, constructor or any block is called “**Instance variable**”.  **Example:**  **Class Test{**  **int x=10;🡨 Instance Variable**  **m1(){//method**  **}**  **Test(){//Constructor**  **}**  **Static{//Block**  **}**  **}**  **It is not declared as static.**  When a space is allocated for an object in the heap, a slot for each instance variable value is created. **Hence, instance variable will be stored in the heap memory as a part of object.**  Instance variables are created when an object is created with the use of the keyword **'new'** and destroyed when the object is destroyed.  Hence, the scope is exactly same as the **scope of Object.**  **Access:**  We cannot access instance variables directly from static area, but we can access by using object reference.  **Example:**  *class Main{*  *int x=10;*  *public static void main(String[] args) {*  *System.out.println(x);* ***//Invalid***  ***CE: Non-static variable x cannot be referenced from a static content.***  *Main m=new Main();*  *System.out.println(m.x);* ***//Valid🡨 Fully Qualified Name***  *}*  *}*  **But we can access instance variable directly from instance area.**  Instance variables can be accessed directly by calling the variable name inside the class. However within static methods and different class (when instance variables are given accessibility) should be called using the fully qualified name. **ObjectReference.VariableName.**  **Example:  *class*** *Main{*  ***int*** *x=10;*  ***public static void main(String[] args) {***  Static Area  ***Main m=new Main();***  ***System.out.println(x);//Invalid***  ***}***  ***public void m1(){***  Dynamic Area  ***System.out.println(x);//Valid***  ***}***  *}*  Instance variables hold values that must be referenced by more than one method, constructor or block, or essential parts of an object's state that must be present throughout the class.  Instance variables can be declared in class level before or after use.  Access modifiers can be given for instance variables.  The instance variables are visible for all methods, constructors and block in the class. Normally, it is recommended to make these variables private (access level). However visibility for subclasses can be given for these variables with the use of access modifiers.  **Instance variables have default values.** For Instance variables, JVM will provide default access values and we are not required to perform initialization.  **For numbers the default value is 0, for Booleans it is false and for object references it is null.**  **Example:**  *class Main{*  *int x;*  *double d;*  *boolean b;*  *String s;*  *public static void main(String[] args) {*  *Main m=new Main();*  *System.out.println("For Integer : "+m.x);*  *System.out.println("For Double : "+m.d);*  *System.out.println("For Boolean : "+m.b);*  *System.out.println("For String : "+m.s);*  *}*  *}*  **Output:** For Integer : 0  For Double : 0.0  For Boolean : false  For String : null  **Instance variables are also known as Object level variables or attributes.** Values can be assigned during the declaration or within the constructor.  **Example**:  *import Java.io.\*;*  *public class Employee{*  *// this instance variable is visible for any child class.*  *public String name;*  *// salary variable is visible in Employee class only.*  *private double salary;*  *// The name variable is assigned in the constructor.*  *public Employee (String empName){*  *name = empName;*  *}*  *// The salary variable is assigned a value.*  *public void setSalary(double empSal){*  *salary = empSal;*  *}*  *// This method prints the employee details.*  *public void printEmp(){*  *System.out.println("name : " + name );*  *System.out.println("salary :" + salary);*  *}*  *public static void main(String args[]){*  *Employee empOne = new Employee("Srinivas");*  *empOne.setSalary(1000);*  *empOne.printEmp();*  *}*  *}*  **This would produce the following result:**  name : Srinivas  salary :1000.0 |
|  |

**Static/Class variable:**

If the values of a variable is not varied from object to object, then it is not recommended to declare variable as instance variables, we have to declare such type of variables at class level by using **Static Modifier**.

**Example:**

***class Student{  
 String name;  
 int rollNo;  
 Static String college;  
}***

**Situation:**

**S400:  
name:**Ravi  
rollNo:500  
**college: AITS**

**S2:  
name:**Shiva  
rollNo:102  
**college: AITS**

**S1:  
name:**Srinivas  
rollNo:101  
**college: AITS**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| i.e.  **Solution:**  **college: AITS**  In this case of instance variable, for every object a separate copy is created. But, in the case of static variables, a single copy will be created at class level, and shared by every object of the class.  A variable that is declared as static is called **static variable**. It cannot be local.  **Declaration:**    Class variables also known as static variables are declared with the static keyword in a class, but outside a method, constructor or a block. i.e, static variable should be declared within the class directly, but outside of any method/block/constructor.  ***Example*:**  ***class Main{***  ***static int x=10;***  ***m1(){***  ***//Method***  ***}***  ***Test(){***  ***//Constructor***  ***}***  ***Static(){***  ***//Static Block***  ***}***  ***}***  There would only be one copy of each class variable per class, regardless of how many objects are created from it.  Static variables are rarely used other than being declared as constants. Constants are variables that are declared as public/private, final and static. Constant variables never change from their initial value.  Static variables are stored in static memory. It is rare to use static variables other than declared final and used as either public or private constants.  **Scope:**  Static variables are created when the program starts and destroyed when the program stops i.e. its created at the time of class loading and destroyed at the time of class unloading.  Hence, the scope of static variable is exactly same as scope of .class file.  **When .class file is created and destroyed:**  When you give “Java Test” on command prompt, the following things will happen:  **Step 1:**  Start JVM.  **Step 2:**  Create and start main method.  **Step 3:**  Locate Test.class file.    Here, main thread will check if we don’t have Test.class, then we will get:  **Exception in thread main, NoClassDefFound Error. Step 4:**  Main thread will load Test.class file.-  **Step 5:**  Static variable creation and destruction.  Main method will be executed.  **Step 6:**  Unload Test.class file.-  **Step 7:**  Terminate main method.  **Step 8:**  Shut down JVM.    **Memory:**  **Static variables will be stored in “Method Area”.**  **Access:**  We can access static variables:   1. **By Object Reference.** 2. **By Class Name. (But, recommended to use class name).** 3. **Within the same class, it is not required to use class name and we can access it directly.**   **Example:**  *class Test{*  *static int x=10;*  *public static void main(String[] args) {*  *Test t=new Test();*  *System.out.println(t.x);//Object Reference*  *System.out.println(Test.x);//Class Reference*  *System.out.println(x);//Within the same class*  *}*  *}*  We can access static variables directly from both instance and static areas.  **Example:**  *class Test{*  *static int x=10;*  *public static void main(String[] args) {*  *System.out.println(x);//Static Area*  *}*  *public void m1(){*  *System.out.println(x);//instance Area*  *}*  *}*  Visibility is similar to instance variables. However, most static variables are declared public since they must be available for users of the class.  **Default values are same as instance variables.**  **For numbers, the default value is 0; for Booleans, it is false; and for object references, it is null.**  **Example:** *class Test{*  *static int x=10;*  *static double d;*  *static String s;*  *public static void main(String[] args) {*  *System.out.println(x);*  *System.out.println(d);*  *System.out.println(s);*  *}*  *}*  For static variables, JVM will provide default values, we are not required to perform initialization explicitly.  **Note**  Static variables are also known as class level variables or fields.  *class Test{*  *static int x=10;*  *int y=20;*  *public static void main(String[] args) {*  *Test t1=new Test();*  Whenever we create an object a separate instance variable is created.  *t1.x=888;*  *t1.y=999;*  *Test t2=new Test();*  *System.out.println(t2.x+"..."+t2.y);*  *}*  *}*  O/p: 888…20  **Explanation:**  Static variable has only one copy. By using any object reference, if we change the value of x, then automatically that change will be reflected by all.  But, for instance variable, every object a separate copy is created, if I change for any object instance variable, it will not change for the remaining.  Values can be assigned during the declaration or within the constructor. Additionally values can be assigned in special static initializer blocks.  Static variables can be accessed by calling with the class name. **ClassName.VariableName**.  When declaring class variables as public static final, then variables names (constants) are all in upper case. If the static variables are not public and final the naming syntax is the same as instance and local variables.    **Example**:  *import Java.io.\*;*  *public class Employee{*  *// salary variable is a private static variable*  *private static double salary;*  *// DEPARTMENT is a constant*  *public static final String DEPARTMENT = "Development ";*  *public static void main(String args[]){*  *salary = 1000;*  *System.out.println(DEPARTMENT+"average salary:"+salary);*  *}*  *}*  **This would produce the following result:**  Development average salary:1000  **Note**:  If the variables are access from an outside class the constant should be accessed as Employee. DEPARTMENT.  **Variable Dynamic Initialization**  *public class MainClass {*  *public static void main(String args[]) {*  *double a = 3.0, b = 4.0;*  *// c is dynamically initialized*  *double c = Math.sqrt(a \* a + b \* b);*  *System.out.println("Hypotenuse is " + c);*  *}*  *}*  **Proper way to declare a class variable named helloMessage**  ***public class MainClass{***  ***static String helloMessage;***  ***public static void main(String[] args){***  ***helloMessage = "Hello, World!";***  ***System.out.println(helloMessage);***  ***}***  ***}***  **You don't have to place class variable declarations at the beginning of a class:**  *public class MainClass{*  *public static void main(String[] args){*  *helloMessage = "Hello, World!";*  *System.out.println(helloMessage);*  *}*  *static String helloMessage;*  *}*  **helloMessage variable is declared as a local variable:**  *public class MainClass{*  *public static void main(String[] args) {*  *String helloMessage;*  *helloMessage = "Hello, World!";*  *System.out.println(helloMessage);*  *}*  *}*  **Variable Scope:**   |  | | --- | | The scope of a variable is the part of the program over which the variable name  Can be referenced. | | You cannot refer to a variable before its declaration. | | **You can declare variables in several different places:** | | * In a class body as class fields. Variables declared here are referred to as class-level  variables. * As parameters of a method or constructor. * In a method's body or a constructor's body. * Within a statement block, such as inside a while or for block. | | **Variable scope refers to the accessibility of a variable.** | | **Rule 1:**  The variables defined in a block are only accessible from within the block. The  scope of the variable is the block in which it is defined.  **Example:** | | |  | | --- | | *public class MainClass {*  *public static void main(String[] args) {*  *for (int x = 0; x < 5; x++) {*  *System.out.println(x);*  *}*  *}*  *}* | | | **Rule 2:**  A nested block can access variables declared in the outer block.  **Consider this code.** | | |  | | --- | | *public class MainClass {*  *public static void main(String[] args) {*  *for (int x = 0; x < 5; x++) {*  *for (int y = 0; y < 3; y++) {*  *System.out.println(x);*  *System.out.println(y);*  *}*  *}*  *}*  *}* | | | **Rule 3:**  Variables declared as method parameters can be accessed from within method  body.  **Rule 4:**  Class-level variables are accessible from anywhere in the class. | | If a method declares a local variable that has the same name as a class-level  variable, the former will 'shadow' the latter.  To access the class-level variable from Inside the method body, use the “this” keyword. | |
|  |

**Summary of variables:**For instance and static variables, JVM will provide default values and we are not required to perform initialization explicitly; but for local variables, JVM won’t provide default values; compulsory we should perform initialization explicitly, before using that variable.

Instance and static variables can be accessed by multiple threads simultaneously and hence these are not thread safe. But, in the case of local variables for every thread a separate copy will be created and hence, **local variables are** **thread safe**.

|  |  |
| --- | --- |
| **Type of variable** | **Is thread safe?** |
| **Instance Variable** | No |
| **Static Variable** | No |
| **Local Variable** | Yes |

Every variable in Java should be either instance/static/local variable, and every variable in Java should be either primitive/reference.,

Hence, the possible combinations of variables in Java are:

**Instance-primitive,  
instance-reference,  
static-primitive,  
static-reference,  
local-primitive and  
local-reference.**

**Example:**

***class*** *Test{*

***int*** *x=10;//instance-primitive*

***static*** *String s="Srini";//Static-Reference*

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

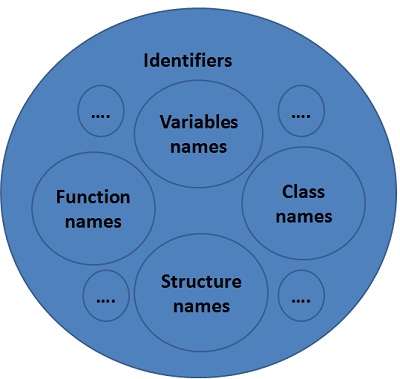
***int****[] y=****new******int****[3];//local-reference*

*}*

*}*

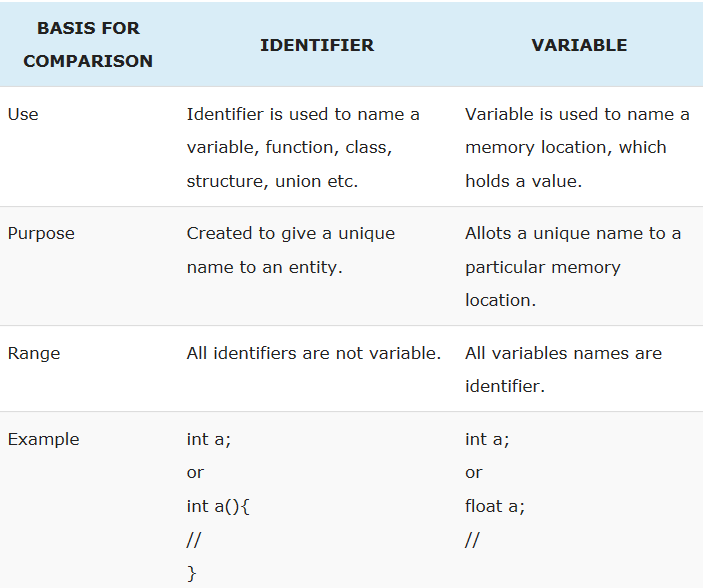
**Difference Between Identifier and Variable**

The word ‘identifier’ clearly defines itself, an identifier is a name given to an entity, which distinctly identifies an entity in a program at the time of its execution. Variable is also an identifier, its name uniquely identifies itself in a program. Here, the fundamental difference between an identifier and variable is that ***an identifier is a “name given to entity” in a program whereas, a variable is a “name given to memory location”, that is used to hold value, which may get modified during program execution.***



## Key Differences between Identifier and Variable

1. Both an identifier and a variable are the names allotted by users to a particular entity in a program. The identifier is only used to identify an entity uniquely in a program at the time of execution whereas, a variable is a name given to a memory location, that is used to hold a value.
2. Variable is only a kind of identifier, other kinds of identifiers are function names, class names, structure names, etc. So it can be said that all variables are identifiers whereas, vice versa is not true.



***Literals***

Literal is a constant value assigned to the variables.

A Literal represents a value that is stored into a variable directly in the program.

**Examples:**

boolean result=**false**;   
char gender=**’M’**;   
short s=**10000**;   
int i=-**1245**;

**Constant(Literal)**

**Example:**

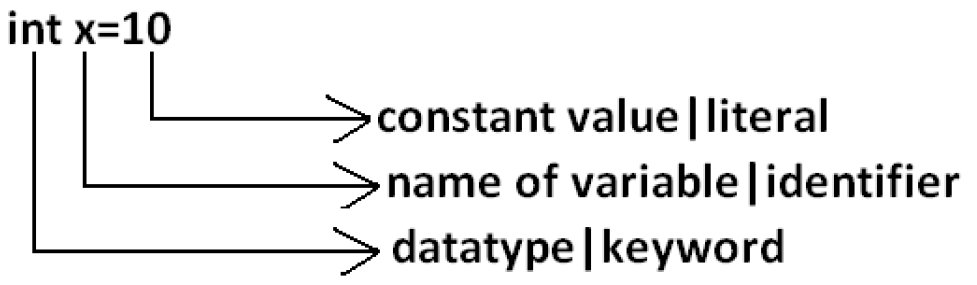
int a=10;

**operator**

**Data type**

**Variable(Identifier)**

**Example**



**To design java application, Java technology has provided the following list of literals:**

***Integer/ integer group literals****:*

Integer data types consist of the following primitive data types:

**int,**

**long,**

**byte, and**

**short**

byte, int, long, and short can be expressed in decimal (base 10), hexadecimal(base 16) or octal(base 8) number systems as well.

For integer literals, we can specify literal values in the following ways..

**a) Decimal literal(base - 10) :**

Allowed digits are 0 to 9.

**Example:** int x=10;

**b) Octal form literal(base - 8) :**

Allowed digits are 0 to 7. Literal values should be prefixed with ‘0’ [Zero].

**Example:** int x=010;

**c) Hexa decimal Literal(base - 16) :**

Allowed digits are 0 to 9, a to f.

For extra digits (a to f), we can use both lower case and upper case characters. This is one of very few areas where java is not case sensitive.

The literal value should be prefixed with 0x or 0X.[zero x/X].

**Example:** int x=0x10; int y=0X10;

These are only possible ways to specify literal value for integral datatype.

Q**) Check the valid declaration values among following:**

a) int x=10; 🡪 Valid

b) int x=0786;🡪Invalid : **Compile Time Error**: Integer number too large.

c) int x=0777; 🡪 Valid

d) int x=0xFace; 🡪 Valid

e) int x=0xBeef; 🡪 Valid

f) int x=0xBeer;🡪 Invalid : **Compile Time Error**: error: ‘;’ expected..

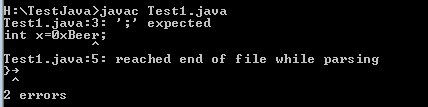
**Example:**

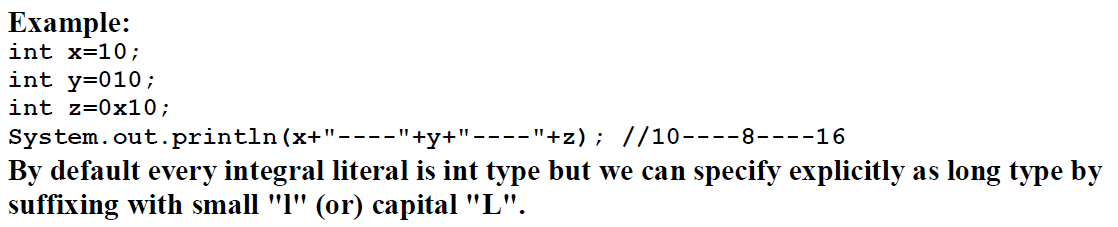
*class Test{*

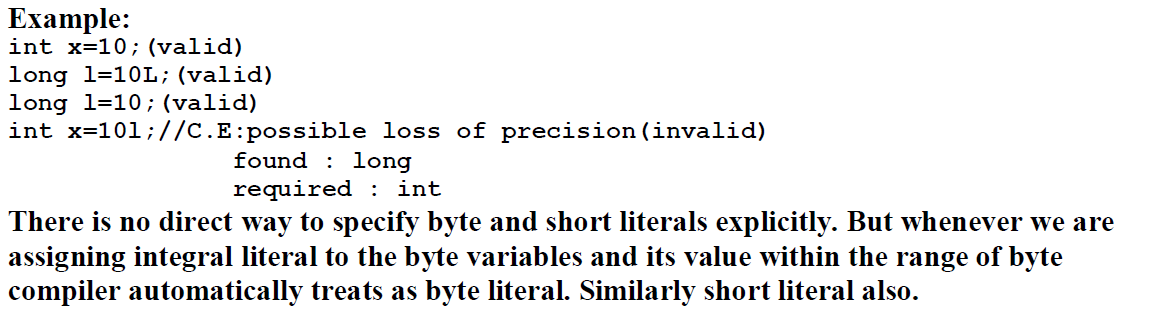
*public static void main(String[] args){*

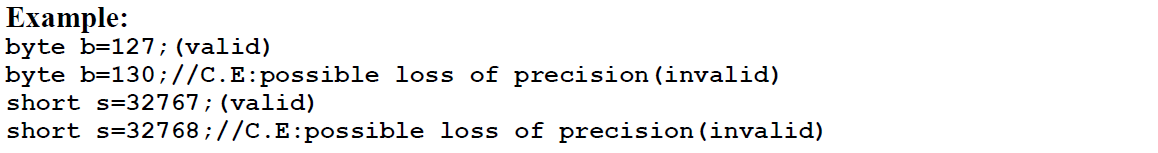
*int x=0xBeer;*

*}*

***Output:***







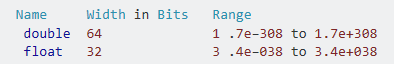
***Floating point literals:***

Floating-point numbers, also known as **real numbers**, are used when evaluating expressions that require fractional precision.

For example, calculations such as square root, or transcendental such as sine and cosine, result in a value whose precision requires a floating-point type.

Java implements the standard (IEEE–754) set of floating point types and operators. There are two kinds of floating-point types, float and double, which represent single- and double-precision numbers, respectively.

**Their width and ranges are shown here:**



**Java has two kinds of floating-point numbers:**

1. float and
2. double

**The default type when you write a floating-point literal is double**, but you can designate it explicitly by appending the D (or d) suffix.

The suffix F (or f) is appended to designate the data type of a floating-point literal as float. We can also specify a floating-point literal in scientific notation using Exponent (short E or e), for instance.

**float**

The type float specifies a single-precision value that uses 32 bits of storage. Single precision is faster on some processors and takes half as much space as double precision, but will become imprecise when the values are either very large or very small. Variables of type float are useful when you need a fractional component, but don't require a large degree of precision.

For example, float can be useful when representing dollars and cents.

**Here are some example float variable declarations:**

float hightemp, lowtemp;

**double**

Double precision, as denoted by the double keyword, uses 64 bits to store a value.

**Double precision is actually faster than single precision on some modern processors that have been optimized for high-speed mathematical calculations**.

All transcendental math functions, such as sin( ), cos( ), and sqrt( ), return double values. When you need to maintain accuracy over many iterative calculations, or are manipulating large-valued numbers, double is the best choice.

**Example:**

a) double literal 0.0314E2 is interpreted as 0.0314 \*10² (i.e 3.14).  
b) 6.5E+32 (or 6.5E32) Double-precision floating-point literal  
c) 7D Double-precision floating-point literal  
d) .01f Floating-point literal  
  
**Question:**  
**The float data type is a single-precision 32-bit IEEE 754 floating point and The double data type is a double-precision 64-bit IEEE 754 floating point.  
What does it mean? And when should I use float instead of double or vice-versa.**

**Answer:**

**To sum up:**

* **float** is represented in 32 bits, with 1 sign bit, 8 bits of exponent, and 23 bits of the mantissa (or what follows from a scientific-notation number: 2.33728\*1012; 33728 is the mantissa).
* **double** is represented in 64 bits, with 1 sign bit, 11 bits of exponent, and 52 bits of mantissa.

By default, Java uses **double** to represent its floating-point numerals (so a literal 3.14 is typed **double**). It's also the data type that will give you a much larger number range, so I would strongly encourage its use over **float**.

There may be certain libraries that actually force your usage of **float**, but in general - unless you can guarantee that your result will be small enough to fit in **float's** [prescribed range](http://docs.oracle.com/javase/7/docs/api/java/lang/Float.html), and then it's best to opt with **double**.

If you require accuracy - for instance, you can't have a decimal value that is inaccurate (like 1/10 + 2/10), or you're doing **anything** with currency (for example, representing $10.33 in the system), then use a **Big** **Decimal**, which can support an arbitrary amount of precision and handle situations like that elegantly.

***A float gives you approx. 6-7 decimal digits precision while a double gives you approx. 15-16. Also the range of numbers is larger for double.***

A double need 8 bytes of storage space while a float needs just 4 bytes.

**Example:**

*import java.util.\*;*

*class Test{*

*private static Scanner Z = new Scanner(System.in);*

*public static final double pi = 3.142;*

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

*Test A = new Test();*

*System.out.println("Enter base and height of triangle ");*

*int base = Z.nextInt();*

*int height = Z.nextInt();*

*System.out.println("Enter the radius of the circle");*

*int radius = Z.nextInt();*

*System.out.println("Enter the length of the square");*

*long length = Z.nextInt();*

*double tArea= A.calculateArea(base,height);*

*double cArea= A.calculateArea(radius);*

*long sqArea= A.calculateArea(length);*

*System.out.println("The area of the triangle is\t"+tArea);*

*System.out.println("The area of the circle is\t"+cArea);*

*System.out.println("The area of the square is\t"+sqArea);*

*}*

*double calculateArea(int base, int height){*

*double triArea=0.5\*base\*height;*

*return triArea;*

*}*

*double calculateArea(int radius){*

*double circArea=pi\*radius\*radius;*

*return circArea;*

*}*

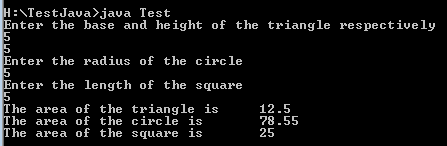
*long calculateArea(long length){*

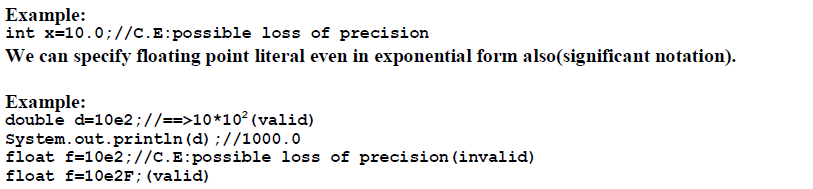
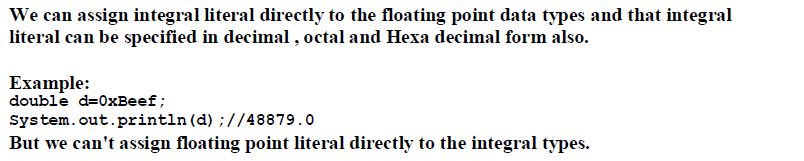
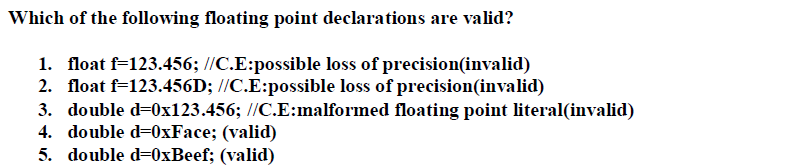
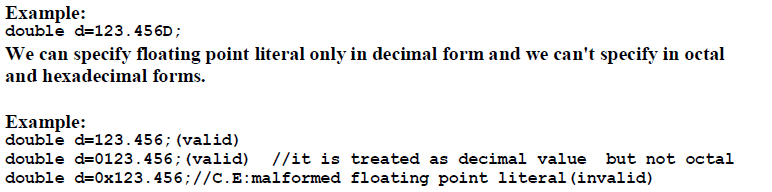
*long squaArea=length\*length;*

*return squaArea;*

*}*

*}*





***Boolean literals:***The values true and false are treated as literals in Java programming.   
Unlike C, we can't presume that the value of 1 is equivalent to true and 0 is equivalent to false in Java.

**Example**boolean chosen = true;  
boolean test = 1; //**Invalid**

# 

# Literal boolean values should not be used in condition expressions

Remove literal Boolean values from conditional expressions to improve readability. Anything that can be tested for equality with a boolean value must itself be a boolean value, and boolean values can be tested atomically.

## Noncompliant Code Example

*if (booleanVariable == true) { /\* ... \*/ }*

*if (booleanVariable != true) { /\* ... \*/ }*

*if (booleanVariable || false) { /\* ... \*/ }*

*doSomething(!false);*

*booleanVariable = condition ? true : exp;*

*booleanVariable = condition ? false : exp;*

*booleanVariable = condition ? exp : true;*

*booleanVariable = condition ? exp : false;*

*booleanVariable = condition ? true : false;*

## Compliant Solution

*if (booleanVariable) { /\* ... \*/ }*

*if (!booleanVariable) { /\* ... \*/ }*

*if (booleanVariable) { /\* ... \*/ }*

*doSomething(true);*

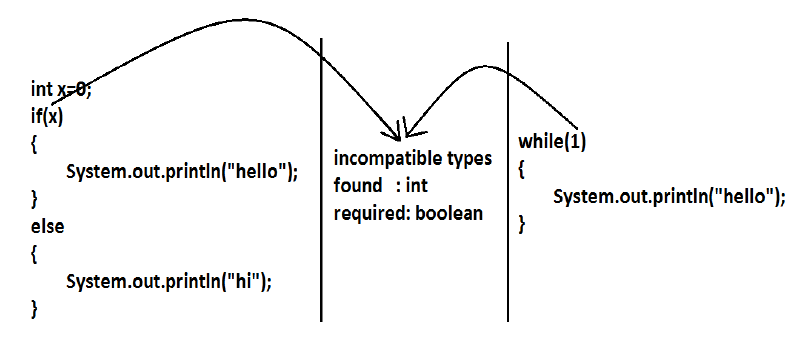
*booleanVariable = condition || exp;*

*booleanVariable = !condition && exp;*

*booleanVariable = !condition || exp;*

*booleanVariable = condition && exp;*

*booleanVariable = condition;*



***String literals***

The set of characters in represented as String literals in Java.

Always use "double quotes" for String literals.

|  |  |
| --- | --- |
| **""** | The empty string |
| **"\""** | A string containing “ |
| **"This is a string"** | A string containing 16 characters |
| **"This is a " + "two-line string"** | actually a string-valued constant expression, formed from two string literals |

#### Question:

#### [Difference between string object and string literal](http://stackoverflow.com/questions/3297867/difference-between-string-object-and-string-literal)

**Or**

**What is the difference between**

**String str = new String("abc");**

**and**

**String str = "abc";**

**Answer:**

A *String literal* is a Java language concept. This is a String literal:

"a String literal"

A *String object* is an individual instance of the java.lang.String class.

String s1 = "abcde";

String s2 = new String("abcde");

String s3 = "abcde";

All are valid, but have a slight difference. **s1 will refer to an *interned* String object**. This means, that the character sequence "abcde" will be stored at a central place, and whenever the same literal "abcde" is used again, the JVM will not create a new String object but use the reference of the *cached* String.

s2 is guranteed to be a *new String object*, so in this case we have:

s1 == s2 // is false

s1 == s3 // is true

s1.equals(s2) // is true

**Similarly,**

String is a class in Java different from other programming languages. So as for every class the object declaration and initialization is

String st1 = new String();

or

String st2 = new String("Hello");

String st3 = new String("Hello");

Here, st1, st2 and st3 are different objects.

That is:

st1 == st2 // false

st1 == st3 // false

st2 == st3 // false

Because st1, st2, st3 are referencing 3 different objects, and **== checks for the equality in memory location,** hence the result.

But:

st1.equals(st2) // false

st2.equals(st3) // true

Here .equals() method checks for the content, and the content of st1 = "", st2 = "hello" and st3 = "hello". Hence the result.

And in the case of the String declaration

String st = "hello";

**Here, intern() method of String class is called**, and checks if "hello" is in intern pool, and if not, it is added to intern pool, and if "hello" exist in intern pool, then st will point to the memory of the existing "hello".

So in case of:

String st3 = "hello";

String st4 = "hello";

Here:

st3 == st4 // true

Because st3 and st4 pointing to same memory address.

Also:

st3.equals(st4); // true as usual

**String Literal Pool**

String allocation, like all object allocation, proves costly in both time and memory. The JVM performs some trickery while instantiating ***string literals*** to increase performance and decrease memory overhead. To cut down the number of String objects created in the JVM, the String class keeps a pool of strings. Each time your code create a string literal, the JVM checks the string literal pool first. If the string already exists in the pool, a reference to the pooled instance returns. If the string does not exist in the pool, a new String object instantiates, then is placed in the pool.

Java can make this optimization since strings are immutable and can be shared without fear of data corruption.

**For Example**

*public class Program{  
 public static void main(String[] args) {  
 String str1 = "Hello";   
 String str2 = "Hello";   
 System.out.print(str1 == str2);  
 }  
}*

The result is **true**

**Unfortunately, when you use**

**String a=new String("Hello");**

a *String* object is created out of the String literal pool, even if an equal string already exists in the pool. Considering all that, avoid new String unless you specifically know that you need it! For example

*public class Program{  
 public static void main(String[] args) {  
 String str1 = "Hello";   
 String str2 = new String("Hello");  
 System.out.print(str1 == str2 + " ");  
 System.out.print(str1.equals(str2));  
 }  
}*

The result is

**false true**

A JVM has a string pool where it keeps at most one object of any String. String literals always refer to an object in the string pool. String objects created with the new operator do not refer to objects in the string pool but can be made to using String's intern() method. The *java.lang.String.intern()* returns an interned String, that is, one that has an entry in the global String pool. If the String is not already in the global String pool, then it will be added. For example

*public class Program{  
 public static void main(String[] args) {* ***// Create three strings in three different ways.*** *String s1 = "Hello";  
 String s2 = new StringBuffer("He").append("llo").toString();  
 String s3 = s2.intern();* ***// Determine which strings are equivalent using the ==  
 // operator*** *System.out.println("s1 == s2? " + (s1 == s2));  
 System.out.println("s1 == s3? " + (s1 == s3));  
 }  
}*

The output is

***s1 == s2? false  
s1 == s3? true***

There is a table always maintaining **a single reference** to each unique String object in the global string literal pool ever created by an instance of the runtime in order to optimize space. That means that they always have a reference to String objects in string literal pool, therefore, the string objects in the string literal pool not eligible for garbage collection.  
  
**When to use string literal and String New Object?**If you create a string using String literals, the string stays in the string-pool forever. It is never garbage collected.   
The down side of this is that the string-pool has a fixed size and it will get full at some time.   
  
With 'new' operator new strings objects are created out of the string-pool, and they are eligible for garbage-collection. So if you don't need them anymore you can make the eligible for garbage collection, but the down side of this is that with 'new' operator JVM will always have to create a new object and it is an overload for JVM.   
  
So if i know that my application is going to use some fixed number of string values , I will use string literals for better performance. But if my application will need different string at different times i will use 'new' operator to create them.

A String objects stores its data in a character array. When you take a substring of a string, by calling the substring() method, then the new String object that is created does not copy part of the data of the original string. Instead, it stores a reference to the underlying data of the original string, along with an offset and length to indicate what part of the old string the new String object represents.   
  
When you have a very long string (for example you read the contents of a file into a String object) and you take a substring out of it, then the JVM will retain all the data of the original string in memory - even if you discard the original String object, because the String object created with substring() still holds a reference to the whole character array with all the data.   
  
To prevent this memory inefficiency, you can explicitly create a new String object using the substring object. That second new String object will copy the data from the substring object, but just the part that you need.

|  |  |
| --- | --- |
|  | **// Suppose this contains 100K characters read from a file**  String largeString = ...;    **// This will refer to the 100K char array from largeString, keeping the whole buffer in memory**  **// even though sub represents only 20 characters**  String sub = largeString.substring(80, 100);    **// This will copy the 20 characters from sub into a new buffer, so that the whole 100K buffer doesn't need to be kept**  String sub2 = new String(sub); |
|  |  |

#### Null Literals

#### To reduce the number of references to an object, use null literal.

The type of the null literal is always null.

**We typically assign null literals to object reference variables. For instance  
s = null**

## What is Null in Java ?

It was originally invented to denote absence of something e.g. absence of user, a resource or anything, but over the year it has troubled Java programmer a lot with nasty null pointer exception.

1) First thing, first, null is a keyword in Java, much like public, static or final. It's case sensitive, you cannot write null as Null or NULL, compiler will not recognize them and give error.

**Object** obj **=** **NULL**; // Not Ok

**Object** obj1 **=** **null** //Ok

2) Just like every primitive has default value e.g. int has 0, boolean has false, null is the default value of any reference type, loosely spoken to all object as well. Just like if you create a boolean variable, it got default value as false, any reference variable in Java has default value **null**. This is true for all kind of variables e.g. [member variable or local variable](http://javarevisited.blogspot.sg/2012/02/difference-between-instance-class-and.html), instance variable or static variable, except that compiler will warn you if you use a local variable without initializing them.

In order to verify this fact, you can see value of reference variable by creating a variable and them printing it's value, as shown in following code snippet :  
  
***private******static*** *Object myObj;*

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

*System.out.println("What is value of myObjc : " + myObj);*

*}*

***Output:*** *What is value of myObjc :* ***null***

This is true for both static and non-static object, as you can see here that I made myObj a static reference so that I can use it directly inside main method, which is static method and doesn't allow non-static variable inside.  
  
3) Unlike common misconception, null is not Object or neither a type. It's just a special value, which can be ***assigned to any reference type*** and [you can type cast null to any type](http://javarevisited.blogspot.sg/2012/12/what-is-type-casting-in-java-class-interface-example.html), as shown below :

***String*** *str* ***=******null****; // null can be assigned to String****Integer*** *itr* ***=******null****; // you can assign null to Integer also****Double*** *dbl* ***=******null****; // null can also be assigned to Double****String*** *myStr* ***=*** *(****String****)* ***null****; // null can be type cast to String****Integer*** *myItr* ***=*** *(****Integer****)* ***null****; // it can also be type casted to Integer****Double*** *myDbl* ***=*** *(****Double****)* ***null****; // yes it's possible, no error*

You can see type casting null to any reference type is fine at both compile time and runtime, unlike many of you might have thought, it will also not throw NullPointerException at runtime.  
  
4) null can only be assigned to reference type, you cannot assign null to primitive variables e.g. int, double, float or boolean. Compiler will complain if you do so, as shown below.

***int*** *i =* ***null****; //* ***type mismatch : cannot convert from null to int  
short*** *s =* ***null****; //* ***type mismatch : cannot convert from null to short  
byte*** *b =* ***null****: //* ***type mismatch : cannot convert from null to byte  
double*** *d =* ***null****; //****type mismatch : cannot convert from null to double***

*Integer itr =* ***null****; // this is ok****int*** *j = itr; //* ***this is also ok, but NullPointerException at runtime***

As you can see, when you directly assign null to primitive error it's compile time error, but if you assign null to a wrapper class object and then assign that object to respective primitive type, compiler doesn't complain, but you would be greeted by null pointer exception at runtime. This happens because of autoboxing in Java, and we will see it in next point.  
  
5) Any wrapper class with value null will throw java.lang.NullPointerException when Java unbox them into primitive values. Some programmer makes wrong assumption that, [auto boxing will take care of converting null into default values](http://javarevisited.blogspot.sg/2012/07/auto-boxing-and-unboxing-in-java-be.html) for respective primitive type e.g. 0 for int, false for boolean etc, but that's not true, as seen below.

**Integer** iAmNull **=** **null**;  
int i **=** iAmNull; // Remember - No Compilation Error

but when you run above code snippet you will see Exception in thread "main" java.lang.NullPointerException  in your console.

This happens a lot while working with HashMap and Integer key values. Code like shown below will break as soon as you run.

*import java.util.HashMap;*

*import java.util.Map;*

*/\*\**

*\* An example of Autoboxing and NullPointerExcpetion*

*\*/*

*public class Test {*

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

*Map numberAndCount = new HashMap<>();*

*int[] numbers = {3, 5, 7,9, 11, 13, 17, 19, 2, 3, 5, 33, 12, 5};*

*for(int i : numbers){*

*int count = numberAndCount.get(i);*

*numberAndCount.put(i, count++); // NullPointerException here*

*}*

*}*

*}*

**Output:**

*Exception in thread "main" java.lang.NullPointerException at Test.main(Test.java:****25****)*

This code looks very simple and innocuous. All you are doing is finding how many times a number has appeared in a array, classic technique to find duplicates in Java array. Developer is getting the previous count, increasing it by one and putting it back into Map. He might have thought that auto-boxing will take care of converting Integer to int , as it doing while calling put method, but he forget that when there is no count exist for a number, [get() method of HashMap](http://java67.blogspot.sg/2013/06/how-get-method-of-hashmap-or-hashtable-works-internally.html) will return null, not zero because default value of Integer is null not 0, and auto boxing will throw null pointer exception while trying to convert it into an int variable. Imagine if this code is inside an if loop and doesn't run in QA environment but as soon as you put into production, BOOM :-)

6) instanceof operator will return false if used against any reference variable with null value or null literal itself, e.g.

*Integer iAmNull = null;*

*if(iAmNull instanceof Integer){*

*System.out.println("iAmNull is instance of Integer");*

*}else{*

*System.out.println("iAmNull is NOT an instance of Integer");*

*}*

**Output** **:*****iAmNull******is******NOT******an******instance******of******Integer***

This is an important property of instanceof operation which makes it useful for type casting checks.  
  
7) You may know that you cannot call a non-static method on a reference variable with null value, it will throw NullPointerException, but you might not know that, you *can* call static method with reference variables with null values. Since [static methods are bonded using static binding](http://javarevisited.blogspot.sg/2011/11/static-keyword-method-variable-java.html), they won't throw NPE.

**Here is an example :**

*public class Testing {*

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

*Testing myObject = null;*

*myObject.iAmStaticMethod();*

*myObject.iAmNonStaticMethod();*

*}*

*private static void iAmStaticMethod(){*

*System.out.println("I am static method, can be called by null reference");*

*}*

*private void iAmNonStaticMethod(){*

*System.out.println("I am NON static method, don't date to call me by null");*

*}*

*}*

**Output:**

*I am* ***static*** *method, can be called by* ***null*** *reference*

*Exception in thread "main" java.lang.NullPointerException*

*at Testing.main(Testing.java:****11****)*

8) You can pass null to methods, which accepts any reference type e.g. public void print(Object obj) can be called as print(null). This is Ok from compiler's point of view, but behavior is entirely depends upon method. Null safe method, doesn't throw NullPointerException in such case, they just exit gracefully. It is recommended to write null safe method if business logic allows.  
  
9) You can compare null value using ==  (equal to ) operator and !=  (not equal to) operator, but cannot use it with other arithmetic or logical operator e.g. less than or greater than. Unlike in SQL, in Java null == null will return true, as shown below :

*public class Test {*

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

*String abc = null;*

*String cde = null;*

*if(abc == cde){*

*System.out.println("null == null is true in Java");*

*}*

*if(null != null){*

*System.out.println("null != null is false in Java");*

*}*

*// classical null check*

*if(abc == null){*

*// do something*

*}*

*// not ok, compile time error*

*if(abc > null){*

*}*

*}*

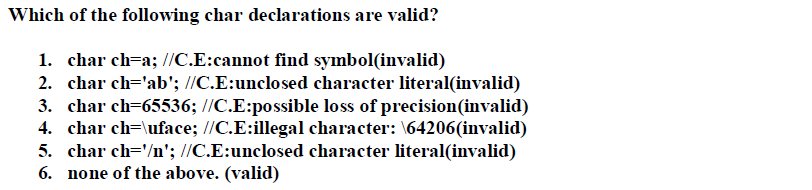
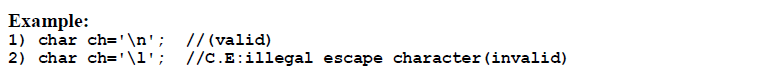
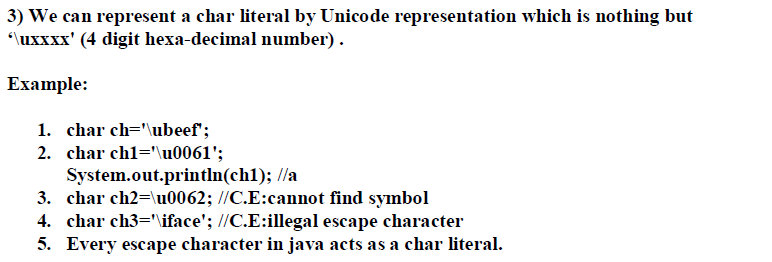
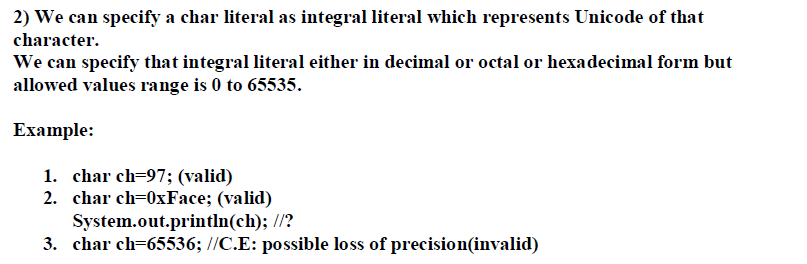
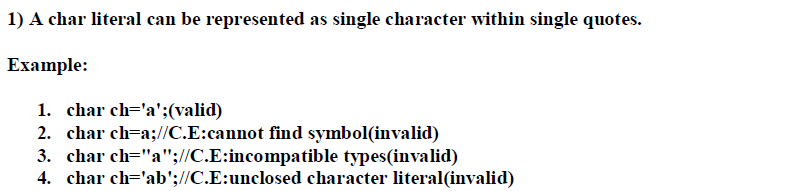
*}*

**Output:**

**null** == **null** is **true** in Java  
  
That's all about null in Java. By some experience in Java coding and by using [simple tricks to avoid NullPointerExcpetion](http://javarevisited.blogspot.sg/2013/05/ava-tips-and-best-practices-to-avoid-nullpointerexception-program-application.html), you can make your code null safe. Since null can be treated as empty or uninitialized value it's often source of confusion, that's why it's more important to document behavior of a method for null input. Always remember, null is default value of any reference variable and you cannot call any instance method, or access an instance variable using null reference in Java.

***Character literals***

char data type is a single 16-bit Unicode character.  
We can specify a character literal as a single printable character in a pair of single quote characters such as 'a', '#', and '3'.

We must know about the ASCII character set. The ASCII character set includes 128 characters including letters, numerals, punctuation etc. 

To write a character in Java, write a single quote, the character, and another single quote. For example, **'A'** is a **char** literal. So is **'a'** (although it's different because it's lower case). So is **'3'**. This is *not* the same as **3**, which is an **int**. Java stores **'3'** in the box in a different way than **3** even though both look about the same.

**' '** is also a **char**. It is a blank space. So is **'?'**. Nearly every character you see on the keyboard is a character, and even a few you don't.

## Not Char Literals

**"3"** is *not* a **char** literal. It uses double quotes, instead of single quotes. (Double quotes makes it a **String** which we'll discuss later).

**''** is *not* a **char** literal. There isn't a character between the two single quotes. You need one character between the double quotes.

**'ab'** is *not* a **char** literal. There are two characters in between the single quotes. **char** literals only have one character in between.

**Question:**

To include a character literal in a Java program, simply place it between single quotes" i.e.

char c = 'A';

What exactly does this do^? I thought char only took in values 0 - 65,535. I don't understand how you can assign 'A' to it?

You can also assign 'B' to an int?

int a = 'B'

The output for 'a' is 66. Where/why would you use the above^ operation?

**Answer:**

char is actually an integer type. It stores the 16-bit Unicode integer value of the character in question.

In Java char literals represent UTF-16 (character encoding schema) code units. What you got from UTF-16 is mapping between integer values (and the way they are saved in memory) with corresponding character (graphical representation of unit code).

You can enclose characters in single quotes - this way you don't need to remember UTF-16 values for characters you use. You can still get the integer value from character type and put if for example in int type (but generally not in short, they both use 16 bits but short values are from -32768 to 32767 and char values are from 0 to 65535 or so).

Some characters, such as the backspace, cannot be written out like this, so these characters are represented by escape sequences. Escape sequences, like all character literals, are enclosed within single quotes. They consist of a backslash followed by one of the following:

* A single character (b, t, n, f, r, ", ', or \)
* An octal number between 000 and 377
* A u followed by four hexadecimal digits specifying a Unicode character

The escape sequences built from single characters are shown in Table

|  |  |
| --- | --- |
| **Escape** | **Meaning** |
| \n | New line |
| \t | Tab |
| \b | Backspace |
| \r | Carriage return |
| \f | Form feed |
| \\ | Backslash |
| \' | Single quotation mark |
| \" | Double quotation mark |
| \d | Octal |
| \xd | Hexadecimal |
| \ud | Unicode character |

If we want to specify a single quote, a backslash, or a non-printable character as a character literal use an escape sequence.

An escape sequence uses a special syntax to represents a character.

The syntax begins with a single backslash character.

**You can see the below table to view the character literals use Unicode escape sequence to represent printable and non-printable characters:**

|  |  |
| --- | --- |
| 'u0041' | Capital letter A |
| '\u0030' | Digit 0 |
| '\u0022' | Double quote " |
| '\u003b' | Punctuation ; |
| '\u0020' | Space |
| '\u0009' | Horizontal Tab |

**NOTE:** C and C++ programmers should note that Java does not include character codes for \a (bell) or \v (vertical tab).

**Example:**

String example = "Socrates asked, \"Hemlock is poison?\"";

System.out.println("Sincerely,\nMillard Fillmore\n");

String title = "Sams Teach Yourself Rebol While You Sleep\u2122"

Here, Unicode code sequence \u2122 produces a !

You can use Unicode sequences anywhere in your Java code, not just as character literals. As indicated earlier, identifiers can be composed of any Unicode character. In fact, comments, identifiers, and the contents of character and string literals can all be expressed using Unicode. You must use caution, however, because they are interpreted early by the compiler. For example, if you were to use the Unicode representation for a linefeed ('\u000a') as part of a print statement, it would cause a compiler error. This is because the compiler would see this as an actual linefeed in your source code that occurs before the closing single quote of a character literal. This is the reason for the earlier caution to always use '\n' and '\r' for line termination literals.

For an example of using Unicode, look at the following statements that declare and reference a variable using an identifier specified with a Unicode sequence:

*int \u0074\u0065\u0073\u0074 = 3;*

*System.out.println( test );*

*System.out.println( \u0074\u0065\u0073\u0074 );*

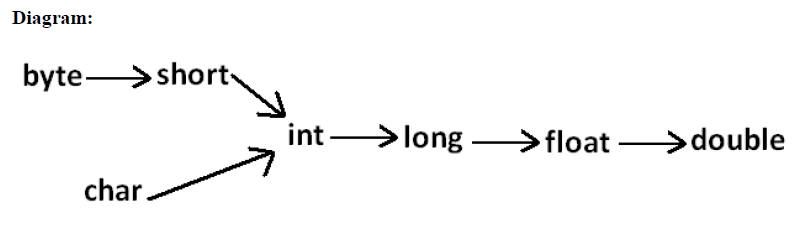
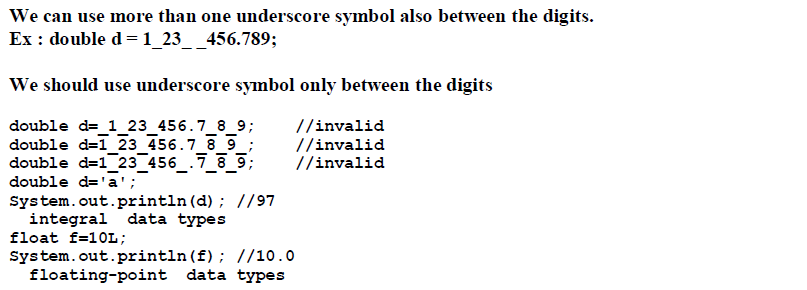
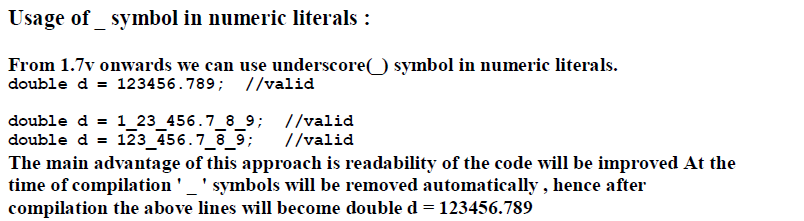
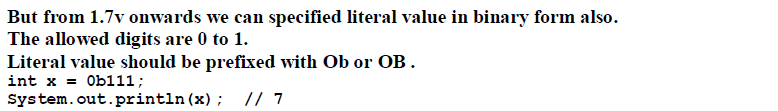
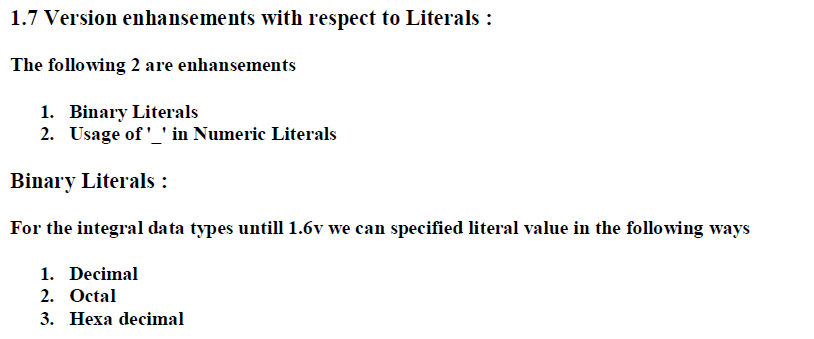
This code probably looks strange to you, but the first statement in this example declares and initializes an integer variable named test ('\u0074' equates to 't', '\u0065' equates to 'e', and so on). Although quite different in appearance, both println statements are equivalent; they each display the value assigned to test when executed.

Now look at two attempts to output a linefeed using different representations:

*System.out.print( "\n" ); // OK*

*System.out.print( '\u000a' ); // a compiler error*

The first statement is valid and is the equivalent of calling System.out.println(). The second statement, however, causes a compiler error. As mentioned previously, the Unicode sequence is interpreted early, and it appears to the compiler that the argument to print is a character literal that is prematurely terminated by a linefeed.



***Number system in java***

In java, all the four number systems are allowed, but, default number system is decimal number system.

**Decimal number system :**

The **Decimal** number system, base 10.  In our number system we utilize the digits 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9.  All numbers are formed by combining these digits.

**Binary number system :**

The **Binary** number system, base 2, uses only the digits 0 and 1.

To represent number must be **prefixed with ‘ob’ or ‘OB’.**

**Example:**

int a=ob10;🡪 **Valid**

int b=0B10;🡪**Valid**

int c=0b102;🡪 **Invalid**

**Note:** Up to Java 6 version, java technology has not allowed binary number system. Java has **introduced binary number system in Java 7** version as a new feature.

**Octal number system :**

The **Octal** number system, base 8, uses the digits 0, 1, 2, 3, 4, 5, 6, and 7.

If we want to represent an octal number, then, we have to prepare number by using the symbols like 0,1,2,…7 and the number must be **prefixed with ‘0’**.

**Example:**

**int** a=0123;🡪 **Valid**

**int** b=03456;🡪 **Valid**

**int** c=078;🡪 **Invalid, because symbol 8 is not allowed in octal.**

**Hexadecimal number System:**

The Hexadecimal number System, base 16, uses the digits and letters 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F.

If we want to represent hexadecimal number in java applications then we have to prepare number by using the symbols like **0 to 9 and a to f** and respective number must be **prefixed with ‘0x’ or ‘OX’.**

**Example:**

**int** a=0x23;🡪 **Valid**

**int** b=0x45e6;🡪 **Valid**

**int** c=0Xefg;🡪 **Invalid, as g is not a number in hexadecimal format.**

**Examples:**

**public** **class** Test1 {

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

**int** n=12;

System.*out*.println("int n : " + n);

n=012;

System.*out*.println("Oct n : " + n);

n=0x12;

System.*out*.println("Hex n : " + n);

**byte** n1=0b11;

System.*out*.println("byte n : " + n1);

}

}

**Output**:

**int** n : 12

Oct n : 10

Hex n : 18

**byte** n : 3

**Explanation**:

012🡪 (12)8 🡪 2\*8^0+1\*8^1=2+8=10

0x12🡪 (12)16 🡪 2\*16^0+2\*16^1=18

0b12🡪 (11)2🡪 1\*2^0 + 1\*2^1 =3

## *Java integer literal and Underscore:*

In general, in number representations, we will not include ‘\_’ symbol, but, “**JAVA 7**” **allows ‘\_’** symbols in the number representations to implement readability.

1. In JDK 7, we can embed one or more underscores in an integer literal.
2. It makes easier to read large integer literals.
3. When the literal is compiled, the underscores are discarded.

**int num = 19\_90;**

1. Java compiler will discard ‘**\_**‘ from the above number and **will assign 1990** to variable “num”. Thus it is as good as writing –

**int num = 1990;**

|  |  |  |
| --- | --- | --- |
| **Literal** | **Using Underscore** | **Actual Value** |
| **Integer Literal** | 45\_89 | 4589 |
| **Octal Literal** | 045\_23 | Equivalent Octal : 04523 |
| **Hexadecimal Literal** | 0x56\_23 | Equivalent Hex : 0x5623 |
| **Binary Literal** | 0b1000\_1001 | Equivalent Binary : 10001001 |

**Note : Using Underscore in Integer**

1. Don’t Use Underscore as first and last character.
2. It is used to read long number easily.

### Illegal ways of using underscore:

Below are some places where we cannot put the underscore while using the Java integer literal –

1. We cannot put underscore at the beginning or end of a number
2. Underscore should not be placed adjacent to a decimal point in a floating point literal
3. Use of underscore prior to an F or L suffix is illegal
4. Underscore should not be used in positions where a string of digits is expected

The following examples demonstrate valid and invalid underscore placements (which are highlighted) in numeric literals:

// **Invalid: cannot put underscores** **adjacent to a decimal point**

float pi1 = 3\_.1415F;

// **Invalid: cannot put underscores** **adjacent to a decimal point**

float pi2 = 3.\_1415F;

// **Invalid: cannot put underscores** **prior to an L suffix**

long socialSecurityNumber1 = 999\_99\_9999\_L;

// OK (decimal literal)

int x1 = 5\_2;

// **Invalid: cannot put underscores** a**t the end of a literal**

int x2 = 52\_;

// OK (decimal literal)

int x3 = 5\_\_\_\_\_\_\_2;

// **Invalid: cannot put underscores** **in the 0x radix prefix**

int x4 = 0\_x52;

// **Invalid: cannot put underscores** **at the beginning of a number**

int x5 = 0x\_52;

// OK (hexadecimal literal)

int x6 = 0x5\_2;

// **Invalid: cannot put underscores** **at the end of a number**

int x7 = 0x52\_;

**Data Type Vs Literals:**

**Data types :**

Primitive types are special data types built into the language; they are not objects created from a class

**Literal :**

A Literal is the source code representation of a fixed value; literals are represented directly in your code without requiring computation

**Example:**

boolean result = true;

Here, boolean - is data type and true - is literal

**Note:** Binary literals and Underscores in numeric literals are introduced in Java 7.

**Java program to convert decimal to hexadecimal:**

**Method 1:**

**Decimal to hexadecimal Using toHexString() method :**

***import*** *java.util.Scanner;*

***class*** *Test{*

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

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

*System.out.print("Enter a decimal number : ");*

***int*** *num =input.nextInt();*

*String str = Integer.toHexString(num);*

*System.out.println("Method 1: Decimal to hexadecimal: "+str);*

*}*

*}*

**Output:**Enter a decimal number : 123

Method 1: Decimal to hexadecimal: 7b

**Method 2:**

**Decimal to hexadecimal without using predefined method:**

***import*** *java.util.Scanner;*

***class*** *Test{*

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

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

*System.out.print("Enter a decimal number : ");*

***int*** *num =input.nextInt();*

*// For storing remainder*

***int*** *rem;*

*// For storing result*

*String str2="";*

*// Digits in hexadecimal number system*

***char*** *hex[]={'0','1','2','3','4','5','6','7','8','9','A','B','C','D','E','F'};*

***while****(num>0) {*

*rem=num%16;*

*str2=hex[rem]+str2;*

*num=num/16;*

*}*

*System.out.println("Method 2: Decimal to hexadecimal: "+str2);*

*}*

*}*

**Output:**

*Enter a decimal number : 123*

*Method 2: Decimal to hexadecimal: 7B*

**Java program for decimal to octal conversion:**

**Method 1:**

**Using predefined method** [**Integer.toOctalString(int num)**](http://docs.oracle.com/javase/7/docs/api/java/lang/Integer.html#toOctalString(int))**:**

***import*** *java.util.Scanner;*

***class*** *Test{*

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

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

*System.out.print("Enter a decimal number : ");*

***int*** *num =input.nextInt();*

*/\* Method 1:*

*\* Using predefined method toOctalString(int)*

*\* Pass the decimal number to this method and*

*\* it would return the equivalent octal number*

*\*/*

*String octalString = Integer.toOctalString(num);*

*System.out.println("Method 1: Decimal to octal: " + octalString);*

*}*

*}*

**Output:**

*Enter a decimal number : 123*

*Method 1: Decimal to octal: 173*

**Method 2:**

**Without using predefined method :**

***import*** *java.util.Scanner;*

***class*** *Test{*

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

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

*System.out.print("Enter a decimal number : ");*

***int*** *num =input.nextInt();*

*/\* Method 2:*

*\* Writing your own logic: Here we will write*

*\* our own logic for decimal to octal conversion*

*\*/*

*// For storing remainder*

***int*** *rem;*

*// For storing result*

*String str="";*

*// Digits in Octal number system*

***char*** *dig[]={'0','1','2','3','4','5','6','7'};*

***while****(num>0){*

*rem=num%8;*

*str=dig[rem]+str;*

*num=num/8;*

*}*

*System.out.println("Method 2: Decimal to octal: "+str);*

*}*

*}*

**Output:**

Enter a decimal number : 123

Method 2: Decimal to octal: 173

**Java program to convert decimal to binary :**

**Method 1:**

**Using toBinaryString() method:**

***class*** *Test{*

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

*System.out.println("Binary representation of 124: ");*

*System.out.println(Integer.toBinaryString(124));*

*System.out.println("\nBinary representation of 45: ");*

*System.out.println(Integer.toBinaryString(45));*

*System.out.println("\nBinary representation of 999: ");*

*System.out.println(Integer.toBinaryString(999));*

*}*

*}*

**Output:**

Binary representation of 124:

1111100

Binary representation of 45:

101101

Binary representation of 999:

1111100111

**Method 2:**

**Without using predefined method:**

***class*** *Test{*

***public******void*** *convertBinary(****int*** *num){*

***int*** *binary[] =* ***new******int****[40];*

***int*** *index = 0;*

***while****(num > 0){*

*binary[index++] = num%2;*

*num = num/2;*

*}*

***for****(****int*** *i = index-1;i >= 0;i--){*

*System.out.print(binary[i]);*

*}*

*}*

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

*Test obj =* ***new*** *Test();*

*System.out.println("Binary representation of 124: ");*

*obj.convertBinary(124);*

*System.out.println("\nBinary representation of 45: ");*

*obj.convertBinary(45);*

*System.out.println("\nBinary representation of 999: ");*

*obj.convertBinary(999);*

*}*

*}*

**Output:**

Binary representation of 124:

1111100

Binary representation of 45:

101101

Binary representation of 999:

1111100111

**Method 3:**

**Decimal to Binary using Stack:**

***import*** *java.util.\*;*

***class*** *Test{*

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

*Scanner in =* ***new*** *Scanner(System.in);*

*// Create Stack object*

*Stack<Integer> stack =* ***new*** *Stack<Integer>();*

*// User input*

*System.out.println("Enter decimal number: ");*

***int*** *num = in.nextInt();*

***while*** *(num != 0) {*

***int*** *d = num % 2;*

*stack.push(d);*

*num /= 2;*

*}*

*System.out.print("\nBinary representation is:");*

***while*** *(!(stack.isEmpty() )) {*

*System.out.print(stack.pop());*

*}*

*System.out.println();*

*}*

*}*

**Output:**

Enter decimal number: 999

Binary representation is:1111100111

**Data Types**

A data type

* Is a scheme for using bits to represent values.
* Values are not just numbers, but any kind of data that a computer can process.
* All values in a computer are represented using one data type or another.

**In Java, every variable and every expression has some type.**

Each and every data type is clearly defined. Every assignment should be checked by compiler for the compatibility. Because of the above reasons, we can conclude Java language as **“*STRONGLY TYPED*”** programming language.

Data types are used for storing input of our program to the main memory of the computer by allocating sufficient amount of memory space.

Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory.

**Q) Java is pure object oriented programming or not ?**

Java is not considered as pure object oriented programming language because several oops feature like ***multiple inheritance, operator overloading*** are not supported by java. Moreover, we are depending on primitive data types which are non-objects.

**In any programming language, there are three types of data types. They are:**

* 1. *Fundamental Data type.*
  2. *Derived Data type and*
  3. *User/programmer/secondary/custom defined data type.*

**Fundamental Data Type:**

Fundamental data types are those whose variables are **allowed to store only one** value but **they never allow us multiple values of same type**.

**Example:**

int a;

a=10; //**Valid**

a=10,20,30; //**Invalid**

**Derived Data Type:**

Derived data types are those which are developed by programmers and whose variables allow us to ***store multiple values either same type*** or different type or both.

In every programming language, the concept of arrays is treated as derived data types.

**Example:**

int a[]={10,20,30};//**Valid**

int b[]={10,10.75,’A’};//**Invalid**.

**User Defined Data Type:**

User defined data types are those which are developed by programmers and whose variables allows us to ***store multiple values either same type or different type or both***.

In C programming language, we have a concept called Structures, Unions, enum, typedef for developing user defined data types.

Similarly, in Java programming, to develop user defined data types, we have a concept called **classes and interfaces.**

**Example:**

Student s=new Student();

|  |
| --- |
| **1** |
| **Sathya** |
| **99.99** |
| **A** |

Here, S is a variable which is holding multiple values of same type or different type or both and whose type is Student. Student is one of the user defined data type.

Broadly, Data Types in Java Programming Language are classified into two main groups –

* 1. **Primitive Data Types and**
  2. **Non primitive/Reference Data Types.**

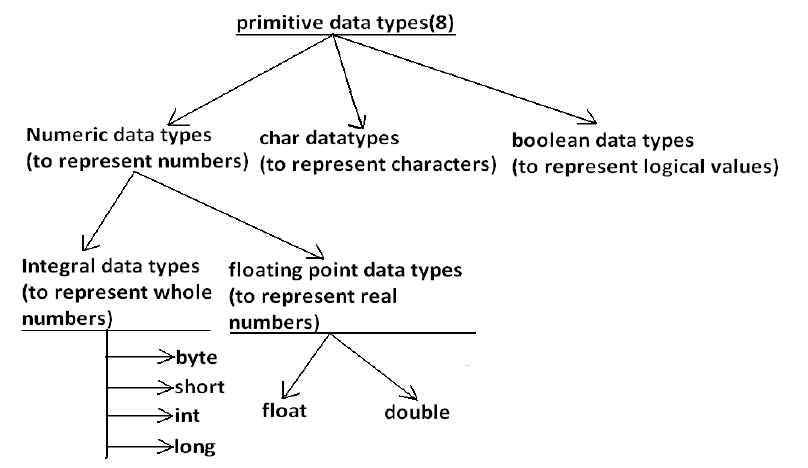
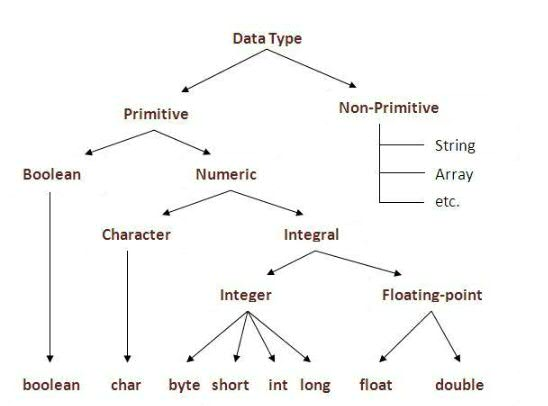
## A primitive Data Types are:

1. Data types are predefined by the Java language.
2. Predefined data types are reserved keyword. So, we cannot use them as variable name inside program/application
3. Primitive values do not share state with other primitive values.
4. Total Number of Primitive Data Types in Java Programming is 8
5. All Primitive Data Types have respective Wrapper Classes i.e **Integer** is wrapper class for primitive type **int.**

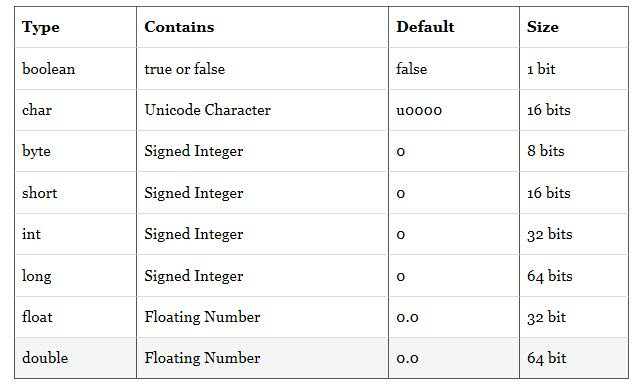
**Note:**

The range of values is calculated as −(2n−1) to (2n−1)−1; where n is the number of bits required. For example, the byte data type requires 1 byte = 8 bits. Therefore, the range of values that can be stored in the byte data type is −(28−1) to (28−1)−1  
= −27 to (27) -1  
= −128 to 127

## Classification of Data Types in Java Programming Language



**Note: Except Boolean and char all remaining data types are considered as signed data types because we can represent both “+ve” and “-ve” numbers.**



**Integer Data Types :**

**These data types represent integer numbers i.e. numbers without any factorial parts or decimal parts.**

**Examples: 125, -223455, 0, 1022, etc.**

**Integer data types are again divided into byte, short, int and long types.**

**Byte:**

**Size: 1 byte (8 bits)**

**Max Value: +127**

**Min Value: -128**

**Range: -128 to 127 [-2^7 to 2^7-1]**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **x** | **1** | **1** | **1** | **1** | **1** | **1** | **1** |

2^6\*1 + 2^5\*1 + 2^4\*1 + 2^3\*1 + 2^2\*1 + 2^1\*1 + 2^0+1=+127

**Sign Bit**  
0— --(+ve)  
1— --(-ve)

The most significant bit as sign bit.

“0” means “+ve” number and “1” means “-ve” number.

“+ve” numbers will be represented directly in the memory whereas “-ve” numbers will be represented in 2’s complement form.

**Example:**

1. byte b=10; 🡪 Valid
2. byte b=127;🡪 **Valid**
3. byte b=128;🡪 **Invalid**

* **Compile Time Error:** Possible loss of precision, found: int required byte.

1. byte b=10.5; 🡪 **Invalid**

* **Compile Time Error:** Possible loss of precision, found: double required byte.

1. byte b=true; 🡪 **Invalid**

* **Compile Time Error:** Incompatible types, found: Boolean required byte.

1. byte b=”durga”; 🡪 **Invalid**

* **Compile Time Error:** Possible loss of precision, found: java.lang.String required byte.

**Note:**

Byte is the best choice, if we want to handle data in terms of streams, either from the file or from the network.

File supported form or network supported form is the byte.

**Short:**

This is the **most rarely** used datatype in Java.

**Size**: 2 bytes

**Max Value: +**32767

**Min Value: -**32768

**Range**: -2^15 to 2^15-1

**i**.e, -32768 to +32767

**Example:**

1. short s=32767; 🡪 **Valid**
2. short s=32768;🡪 **Invalid**

* **Compile Time Error:** Possible loss of precision, found: int required short.

1. short s=10.5; 🡪 **Invalid**

* **Compile Time Error:** Possible loss of precision, found: double required short.

1. short s=true; 🡪 **Invalid**

* **Compile Time Error:** Incompatible types, found: Boolean required short.

1. short s=”durga”; 🡪 **Invalid**

* **Compile Time Error:** Possible loss of precision, found: java.lang.String required short.

**Note:**

Short datatype is best suitable for 16 bit processors like 8085, but these processors are completely outdated and hence, corresponding short datatype is also outdated datatype.

========================================================

**int:**

The most **commonly used** data type in Java in **int**.

**Size**: 4 bytes(32 bits)

**Range:** -2^31 to 2^31-1 i.e, -2147483648 to 2147483647

**Example:**

1) int i=2147483647; 🡪 **Valid**

2) int i=2147483648;🡪 **Invalid**

**Compile Time Error:** integer number too large.

3) int i=2147483648**L**; 🡪 **Invalid**

**Compile Time Error:** Possible loss of precision, found: long required int.

4) int i=true; 🡪 **Invalid**

**Compile Time Error:** Incompatible types, found: Boolean required int.

5) int i=”durga”; 🡪 **Invalid**

**Compile Time Error:** Possible loss of precision, found: java.lang.String required int.

**Long:**

Sometimes, int may not be enough to hold big values, and then we should go for long type.

**Example 1:**

The amount of distance travelled by light in thousand days, to hold this value int may not be enough, we should go for long data type.

i.e, long l=1,26,000\*60\*60\*24\*1000 miles;

**Example 2:**

The number of characters present in a big file may exceed int range, hence, the return type of length method in long but not int.

**Example 3:**

long l=f.length(); //length / Number of characters of a file.

**Size**: 8 bytes (64 bits)

**Range**: -2^63 to 2^63-1  
i.e. 9,223,372,036,854,775,808 to 9,223,372,036,854,755,807

**Note:**

1) All the above data types (byte, short, int, long), meant for representing integral values. If we want to represent floating point values, then we should go for floating point datatypes.

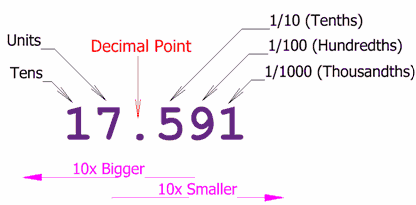
2) Wrapper classes are built-in classes of Java, which is used to represent the primitive data types of java.

|  |  |
| --- | --- |
| **Data type** | **Wrapper Classes** |
| byte | java.lang.Byte |
| short | java.lang.Short |
| integer | java.lang.Integer |
| long | java.lang.Long |

**Floating point data types:**

Decimal values with a fractional component is called floating point.

They can be expressed in either **standard or scientific notation**.



### Standard Notation

1. Standard notation consists of a whole number component followed by a decimal point followed by a fractional component.
2. **For Example**:  78.0, 3.14159 represent valid **standard-notation floating-point numbers**.

### Scientific Notation

1. Scientific notation uses a standard-notation, floating-point number plus a suffix that specifies a power of 10 by which the number is to be multiplied.
2. The exponent is indicated by an E or e followed by a decimal number, **which can be positive or negative**.   
   e or E represents X10 to the power. Hence, 1.98e8 means 1.98\*10^8. This is also called scientific notation of representing numbers.
3. **Valid Examples are :**
   * 6.02E21
   * 314159E–05
   * 2e+100.
4. Floating-point literals in Java default to **double precision**.

|  |  |
| --- | --- |
| **Float** | **Double** |
| If we want 5 to 6, decimal places of accuracy, then we should go for float. | If we want 14 to 15 decimal places of accuracy, then we should go for double. |
| Float follows single precision (less accuracy) | Double follows double precision(More accuracy) |
| **Size:** 4 Bytes | **Size:** 8 Bytes |
| **Range:** -3.4e38 to 3.4e38 | **Range:** -1.7e308 to 1.7e308 |

## float:

* Float data type is a single-precision 32-bit IEEE 754 floating point.
* Float is mainly used to save memory in large arrays of floating point numbers.
* Default value is 0.0f.
* Float data type is never used for precise values such as currency.

**Example**: float f1 = 234.5f

* Wrapper class is java.lang.Float.

## double:

* double data type is a double-precision 64-bit IEEE 754 floating point.
* This data type is generally used as the default data type for decimal values, generally the default choice.
* Double data type should never be used for precise values such as currency.
* Default value is 0.0d.

**Example**: double d1 = 123.4

* Wrapper class is java.lang.Double.

**Note:**

If we use a real constant value directly to the Java program, then that real constant value is by **default treated** as highest data type in float category i.e, **Double datatype.**

i.e, 10.65 is double and 10.65f or 10.65F is float.

**Boolean Data Type:**

* boolean data type represents one bit of information.
* There are only two possible values: true and false.
* This data type is used for simple flags that track true/false conditions.
* **Default value is false**.

**Example**: boolean one = true

**a) Size:** NA [Virtual machine Dependent]

**b) Range:** NA [But, allowed values are true/false]

**Example:**

* 1. boolean b=true;
  2. Boolean b=0;
  + **Compile Time Error:** Incompatible types found: int required: Boolean.
  1. boolean b=True;
  + **Compile Time Error:** Cannot find symbol, symbol: Variable True location: class Test.
  1. Boolean b=”true”;
  + **Compile Time Error:** Incompatible types found: java.lang.String required Boolean.

**Example :**

**int** x=0;

**if**(x){

System.*out*.println("Hello");

}**else**{

System.*out*.println("Hi");

}

**Compile Time Error:** Incompatible types found: int required: Boolean

**Example**

**while**(1){

System.*out*.println("Hello");

}

**Compile Time Error:** Incompatible types found: int required: Boolean

**char data type:**

A character is an identifier which is enclosed within single quotes.

**Example:**

‘A’, ‘$’, ‘9’, ‘#’, etc..,

**Size :** 2 Bytes

**Range:** 0 to 65535

Collection of characters enclosed within double quotes is known as **String**.

**Example:**

* “ABC”

|  |
| --- |
| Before Unicode, there were many language standards: |
| * **ASCII** (American Standard Code for Information Interchange) for the United States. * **ISO 8859-1** for Western European Language. * **KOI-8** for Russian. * **GB18030 and BIG-5** for chinese, and so on. |

|  |
| --- |
| **This caused two problems:**   1. A particular code value corresponds to different letters in the various language standards. 2. The encodings for languages with large character sets have variable length. Some common characters are encoded as single bytes, other require two or more byte. |
| To solve these problems, a new language standard was developed i.e. Unicode System. |

**Why char is taking 2 bytes in Java, and for other old programming languages it is 1 Byte ?**

Old languages (like ***C/C++) are ASCII code*** based, and the number of different allowed ASCII code characters are <=256.

To represent 256 characters, 8 bits are enough. Hence, the size of char in old languages is 1 byte.

But, Java is ‘***unicode’*** based, and the number of different Unicode characters are >256 and <=65,536.

To represent these many characters 8 bits maynot be enough. So, compulsory we should go for 16 bits. Hence, the size of char in Java is 2 Bytes.

**UNICODE - Universal Code Character.**

**C/C++ [ASCII]**a-z  
A-Z  
0-9  
$,#,+,-,\*,…

**Java [Unicode]**256 char  
+  
Telugu letters  
+  
Hindi letters  
+  
…

For these 256, we will have same Unicode values and ASCII values.

***Example:*** ASCII value and Unicode value of ‘a’ is 97.

* Minimum value is '\u0000' (or 0).
* Maximum value is '\uffff' (or 65,535 inclusive).

**Defination of UNICODE:**

A UNICODE character set is one which contains all the characters which are available in 18 international languages.

ASCII character set is one which contains all the characters which are available in only international languages called English.

The languages like C/C++ are available in English and the language Java is available in 18 international languages(English, Greek, latin, etc..).

**Note:**

As per as English java programmer is concerned, to represent a character, we need one byte and the rest of the one byte is simply wasted. This is one of the negligible drawbacks of Java language.

**Summary of Java primitive Data Type:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Data type** | **Size** | **Range** | **Wrapper Class** | **Default Value** |
| Byte | 1 byte | -2^7 to +2^7-1  (-128 to +127) | Byte | 0 |
| Short | 2 bytes | -2^15 to 2^15-1  (-32768 to 32767) | Short | 0 |
| Int | 4 bytes | -2^31 to +2^31-1 (-2147483648 to +2147483647) | Integer | 0 |
| Long | 8 bytes | -2^63 to +2^63-1 | Long | 0 |
| Float | 4 bytes | -3.4e38 to +3.4e38 | Float | 0.0 |
| Double | 8 bytes | -1.7e308 to 1.7e308 | Double | 0.0 |
| Boolean | NA | NA(allowed values are true/false) | Boolean | false |
| Char | 2 bytes | 0 to 65535 | Character | 0[represents space character] |

**Note:**

‘null’ for primitives are not applicable. If we try to apply it, we will get **“Compile Time Error”.**

‘null’ is the default value for object reference, and we can’t apply for primitives.

**Example:**

Char ch = null;

**Complie Time Error:**

Incompatible types found: <null type> required: char.

**Example for default values in Primitive Datatypes:**

**public** **class** ClassInitializer1 {

**static** **boolean** *bool*;

**static** **byte** *by*;

**static** **char** *ch*;

**static** **double** *d*;

**static** **float** *f*;

**static** **int** *i*;

**static** **long** *l*;

**static** **short** *sh*;

**static** String *str*;

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

System.*out*.println("bool = " + *bool*);

System.*out*.println("by = " + *by*);

System.*out*.println("ch = " + *ch*);

System.*out*.println("d = " + *d*);

System.*out*.println("f = " + *f*);

System.*out*.println("i = " + *i*);

System.*out*.println("l = " + *l*);

System.*out*.println("sh = " + *sh*);

System.*out*.println("str = " + *str*);

}

}

**Output**:

*bool = false*

*by = 0*

*ch = \_*

*d = 0.0*

*f = 0.0*

*i = 0*

*l = 0*

*sh = 0*

*str = null*

**Java lets you overflow**

***public******class*** *MainClass {*

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

***int*** *big = 0x7fffffff; // max int value*

*System.out.println("big = " + big);*

***int*** *bigger = big \* 4;*

*System.out.println("bigger = " + bigger);*

*}*

*}*

**Output**:

*big = 2147483647*

*bigger = -4*

**Print the limits of primitive types (e.g. byte, short, int ...) in Java**

***public******class*** *Main {*

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

*System.out.println("Byte.MIN = " + Byte.MIN\_VALUE);*

*System.out.println("Byte.MAX = " + Byte.MAX\_VALUE);*

*System.out.println("Short.MIN = " + Short.MIN\_VALUE);*

*System.out.println("Short.MAX = " + Short.MAX\_VALUE);*

*System.out.println("Integer.MIN = " + Integer.MIN\_VALUE);*

*System.out.println("Integer.MAX = " + Integer.MAX\_VALUE);*

*System.out.println("Long.MIN = " + Long.MIN\_VALUE);*

*System.out.println("Long.MAX = " + Long.MAX\_VALUE);*

*System.out.println("Float.MIN = " + Float.MIN\_VALUE);*

*System.out.println("Float.MAX = " + Float.MAX\_VALUE);*

*System.out.println("Double.MIN = " + Double.MIN\_VALUE);*

*System.out.println("Double.MAX = " + Double.MAX\_VALUE);*

*}*

*}*

**Output:**

Byte.MIN = -128

Byte.MAX = 127

Short.MIN = -32768

Short.MAX = 32767

Integer.MIN = -2147483648

Integer.MAX = 2147483647

Long.MIN = -9223372036854775808

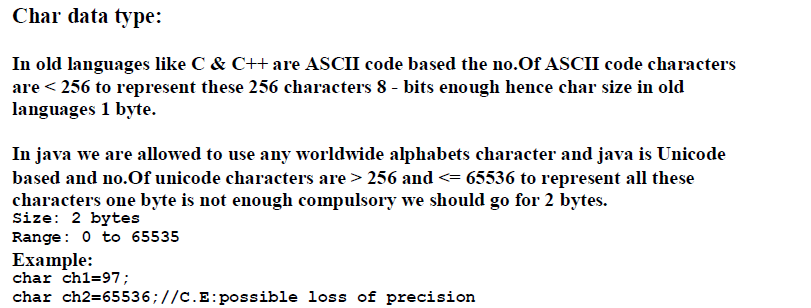
Long.MAX = 9223372036854775807

Float.MIN = 1.4E-45

Float.MAX = 3.4028235E38

Double.MIN = 4.9E-324

Double.MAX = 1.7976931348623157E308



**Comments**

**Comments are used to explain about the program and make the program more readable and understandable**

We can write any number of comments and anywhere inside the program

Comments are non-executable statements;

**In java we have following 3 types of Comments**

1. **single line comment**

If we want to write comments in 1 line then we go for this single line comment.

**Eg**:

// -------------------------

2. **Multi line comment**

If we want to write comments in more than 1 line then we go for this multi-line comment.

**Eg**:

/\* -------------------------

------------------------- \*/

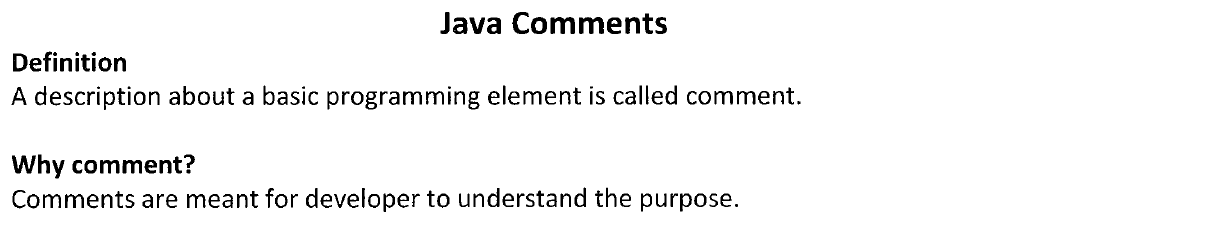
3. **Document comments**

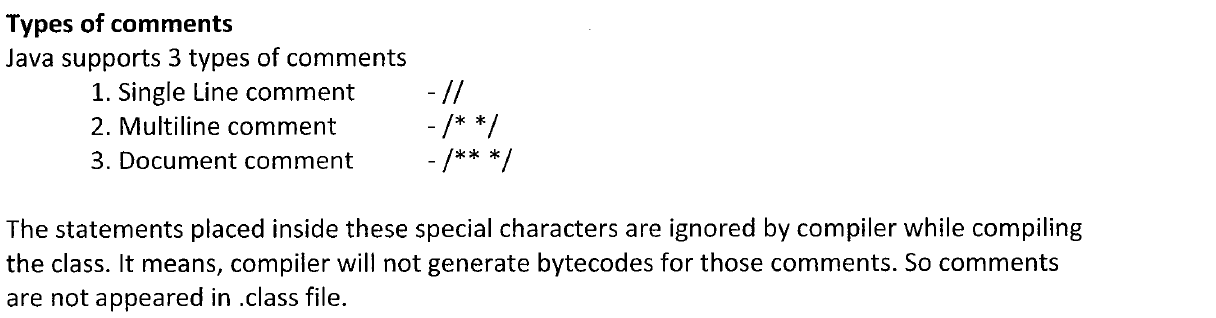
This comments are used to create a separate document which contains information about the program in detail.

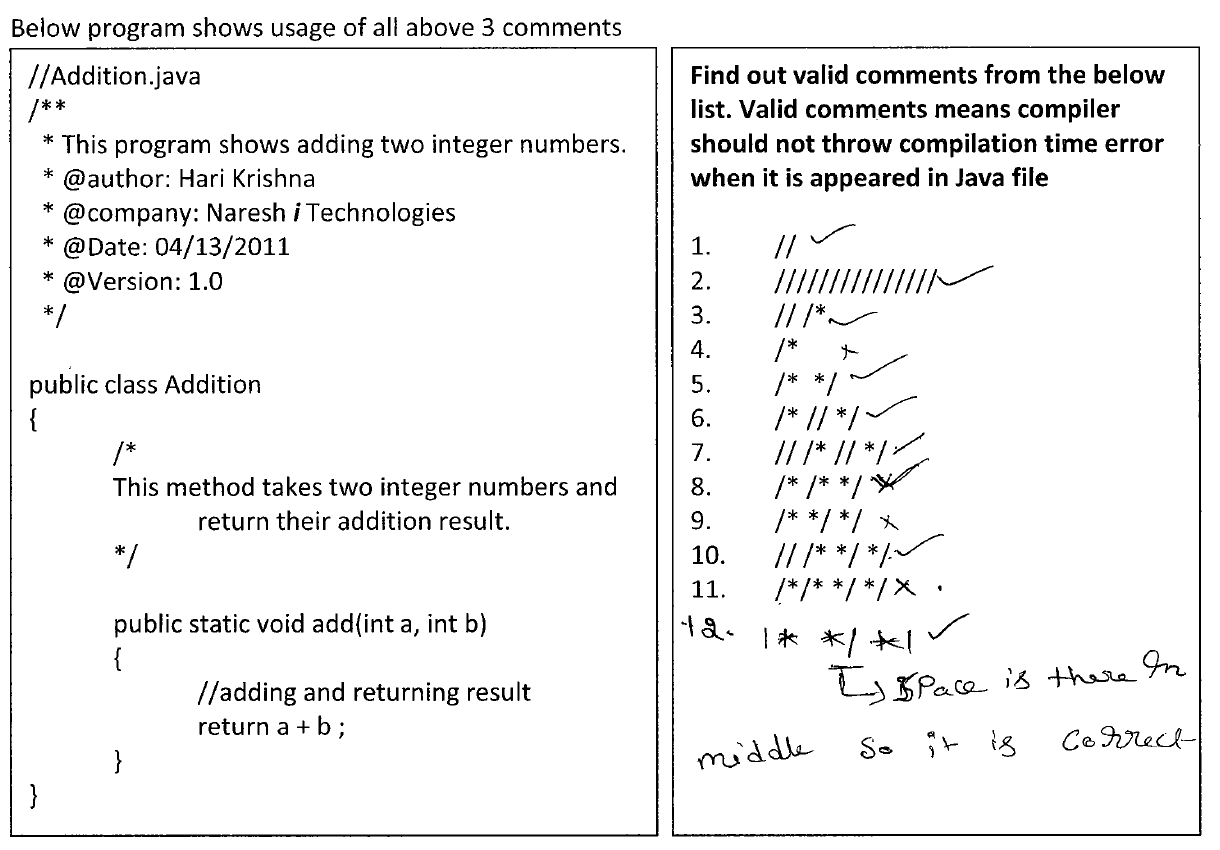
**Eg**:

/\*\*-------------------------

------------------------- \*/







**Keywords/Reserved Words**

In Java, some words are reserved to represent some meaning or functionality, such type of words are called “Reserved words”.

**Keywords:**

Every programming language comes with keywords.

**A keyword is just a word which has got a special meaning and purpose to the compiler**.

For this reason, we cannot use them in our program to identify our own (to give a name) a class, variable or method.

**Keywords Java:**

Some people interchangeably use the two words, **keyword** and **reserved word**. They think both are same; may be correct in many programming languages. But in Java, they are different.

While Java is being developed, **the designers placed the two words, goto and const in reserved word list and not in keyword list**. Designers placed in reserved list for the reason thinking that **goto** and **cosnt** may be required in future versions of Java; any time they may be brought into keyword list. But as on today, Java proved that a programming language can exist without these two words.

**In Java 53 reserved words are there.**

Unused Keywords(2)

Used Keywords(48)

Reserved Literals(3)

Keywords(50)

Reserved words Reserved words Reserved words

true

falsee

Null

If

gotoif

Elseif

Ifif

**Keywords:**

If the reserved words are associated with functionality, it is called as Keywords.

**Reserved literals:**

If the reserved word is associated with only value, is called Reserved literals.

**Example:** true, false, null

**Keywords for datatype: (8)**

byte

short

int

long

double

float

Boolean

char

**Keywords for flow control: (11)**

if

else

switch

case

default

do

while

for

break

continue

return

**Keywords for modifiers: (11)**

public

private

protected

default🡪 already used

static

final

abstract

synchronized

native

strictfp(1.2 version)

transient

volatile

**Keywords for Exception Handling: (6)**

try

catch

finally

throw

throws

assert(1.4 version)

**Class related keywords: (6)**

class

interface

extends

implements

package

import

**Object related keywords: (4)**

new

instanceof

super

this

**Return type keyword: (1)**

void

In java, return type is mandatory. If a method won’t return anything, then we have to declare that method type with void return.

But, in C-language, return type is optional and default return type is int.

**Unused Keywords:**

If by mistake we are using goto or const keywords in our program, we will get compile time error.

**goto:**

* Usage of goto created several problems in old languages, and hence ‘sun’ people banned this keyword in java.

**const:**

* Use final instead of const.

**Reserved Literals:**

1. true

Values for Boolean data type

1. false
2. null 🡪 Default value for object reference.

**enum keyword: (1.5 version)**

* This keyword is introduced in 1.5 version
* We can use enum to define a group of named const.

**Example 1 :**

enum month{

JAN,FEB,MAR;

}

**Example 2 :**

enum Beer{

KF,RC,KO;

}

**Conclusions:**

1. All 53 reserved words, in Java contains **only lower case alphabet symbols**.
2. In java, we have only ‘new’ keyword, and there is **no delete keyword**, because destruction of useless objects is the responsibility of garbage collector.
3. **The following are new keywords in Java.**
4. strictfop – 1.2 Version
5. assert – 1.4 version
6. enum – 1.5 version
7. **Small variations to be noted are:**
8. strict**fo**p but not strictFp
9. instanceof but not instanceOf
10. synchronized but not synchronize
11. extends but not extend
12. implements but not implement
13. import but not imports
14. const but not constant.

**Q) How to find whether a given word is a reserved keyword or not in Java program?**

A. **import** java.util.Arrays;

**public** **class** Test {

**static** **final** String *keywords*[] = { "abstract", "assert", "boolean",

"break", "byte", "case", "catch", "char", "class", "const",

"continue", "default", "do", "double", "else", "extends", "false","final", "finally", "float", "for", "goto", "if",   
"implements","import", "instanceof", "int", "interface", "long", "native","new", "null", "package", "private", "protected", "public",”return", "short", "static", "strictfp", "super", "switch","synchronized", "this", "throw", "throws", "transient", "true","try", "void", "volatile", "while" };

**public** **static** **boolean** isJavaKeyword(String keyword) {

**return** (Arrays.*binarySearch*(*keywords*, keyword) >= 0);

}

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

System.*out*.println(" Is void a java Keyword : "+*isJavaKeyword*("void"));

System.*out*.println(" Is test a java Keyword : "+*isJavaKeyword*("test"));

}

}

**Output:**Is void a java Keyword : true

Is test a java Keyword : false

**Q) which of the following are not reserved words in Java?**

A. transient🡪**Valid**

B. include🡪 **Invalid**

C. goto🡪 **Valid**

D. union🡪**Invalid**

**Q Which of the following list contains only Java reserved words?**

1. new, delete🡪 **Invalid**, delete is not a reserved keyword.
2. goto, constant🡪 **Invalid**, constant is not a reserved keyword.
3. break, continue, return, exit🡪 **Invalid**, exit is a method.
4. final, finally, finalize🡪 **Invalid**, finalize is a method.
5. throw, throws, thrown🡪 **Invalid**, thrown is not a reserved keyword.
6. notify, notifyAll🡪 **Invalid**
7. implements,extends,imports🡪 **Invalid**, it must be import but not imports.
8. sizeof, instanceof🡪**Invalid**, sizeof is not a reserved word.
9. instanceOf, strictFp🡪 **Invalid**.
10. byte,short, Int🡪 **Invalid**.
11. None of the above 🡪 **Valid**.

**Q) Which of the following are java reserved words?**

a) public🡪 **valid**

b) static🡪 **valid**

c) void🡪 **valid**

d) main🡪 **Invalid**, name of the method.

e) string 🡪 **Invalid**, name of predefined class.

f) args 🡪 **Invalid**, name of variable.

**Summary:**

The following table lists and annotates all of the Java reserved identifiers. Reserved identifiers cannot be used in any other way within Java programs. In particular, you cannot use any of these reserved identifiers as your own identifiers in Java programs.

|  |  |  |
| --- | --- | --- |
| **Identifier** | **Context** | **Description** |
| **Abstract** | class modifier | **abstract class** MyAbstractClass **{** *members;* **}**  For classes: The class cannot be instantiated directly; that is, you may not "new MyAbstractClass()". Subclasses of an abstract class must provide bodies in overridden methods to be capable of being instantiated.  Abstract classes generally have at least one abstract method. However, this is not a requirement. Abstract classes may have a mix of both abstract and concrete methods, or they may have all concrete methods. |
| method modifier | **abstract** voidaProcedure**();**  For methods: There is no body associated with the method, only a semicolon marks the end of the method declaration.  Classes that contain abstract methods must themselves be abstract. However, abstract classes need not have any abstract methods. |
| **Assert** | run-time debugging check |  |
| **Boolean** | primitive type | **boolean** trueOrFalse**;**  Boolean variables can only be assigned the values **true** or **false**. |
| **Break** | control statement (found in iteration or switch statements) | * **break;**   Always ends the innermost control structure in which the **break** statement is embedded. Iteration statements do not test to see if they should execute another cycle. |
| * **break** *label***;**   Labeled **break** statements end all nested control structures out to the iteration or switch statement with the same label. |
| **Byte** | primitive type | **byte** tensOfThings**;**  Byte variables can be assigned 8-bit integer values between –128 and 127. |
| **Case** | switch statement clause | **case** CASEVALUE**:** *statements;* **break;**  Within a **switch** statement, execution is suspended until a case value that matches the switch expression is found. |
| **Catch** | try statement clause (handles exceptions) | **try {** *statements;* **}**  **catch (**MyException e**) {** *statements;* **}**  The **catch** statement can only be used in conjunction with a **try** statement. When an exception is thrown in the try block of a **try** statement, program execution continues in a **catch** block which matches the exception thrown. Catch blocks handle any exceptions thrown by code within a **try** block. |
| **Char** | primitive type | **char** aVowel**;**  Char variables can be assigned 16-bit Unicode characters. |
| **Class** | class declaration | * **class** MyClass **{** *members;* **}** * **class** MyClass **extends** MySuperClass **implements** FirstInterface, SecondInterface **{** *members;* **}**   The class is the basic building module of Java. It creates a generic description (or template, or blueprint) of one are more *objects,* the i*nstances* of the class. The class body lists the *members* of the class: the fields (or variables, or attributes), the methods (or functions), or local class/interface declarations. |
| **Const** | unused | Use **final** to indicate fields that are not changed after initialization. |
| **continue** | statement (found in iteration statements) | * **continue;** * **continue** looplabel**;**   The **continue** statement returns to the beginning of an embedded loop, at which time the loop tests itself whether to execute the body again or exit. The unlabeled **continue** goes to the beginning of the innermost nested loop. The labeled **continue** goes to the beginning of the iteration statement with that label. |
| **default** | switch statement clause | **default:** *statements;*  If none of the case clauses in a switch statement match the switch expression, execution of statements within the switch block begins at the "default case" or "default label". |
| **do** | iteration statement | **do** *statements;***while (***test***);**  Executes a sequence of statements while the while condition is true. The iterated statements are executed at least once. |
| **double** | primitive type | **double** astronomicalQuantity**;**  Double variables can be assigned 64-bit double-precision floating-point values. The largest possible double value is 1.79769313486231570e+308. |
| **else** | if statement clause | **if (***test***)** *statement1***;**  **else** *statement2***;**  The **else** keyword is used only in conjunction with an **if** statement. The statement or block that follows the **else** is executed when the **if** test evaluates to **false**. |
| **enum** | programmer-defined type |  |
| **extends** | class/interface declaration clause | * **class** MySubClass **extends** MySuperClass **{** *members;* **}** * **interface** SecondInterface **extends** FirstInterface **{** *members;* **}**   If a class does not explicitly extend a superclass, the class is taken to be a subclass of the class Object (as if you had written, "**extends** Object"). |
| **false** | boolean literal | boolean allAnimalsAreDogs **= false;** |
| **final** | class modifier | * **final class** NeverSuper **{** *members;* **}**   declares a class that can never be extended. |
| method modifier | * **final** voidaProcedure() **{** *statements;* **}**   declares a method that can never be overridden in a subclass. |
| field modifier | * **final** int LIMIT = 100;   declares a constant. |
| **finally** | try statement clause | **try {** *statements;* **}**  **finally {** *statements;* **}**  The **finally** block is executed whether or not an exception was thrown in a **try** block. It is not possible to determine within the **finally** block whether or not an exception was thrown within the **try** block. |
| **float** | primitive type | **float** aFloat**;**  Float variables can be assigned 32-bit single-precision IEEE floating point values. The largest possible **float** value is 3.40282347e+38f (note the "f"). |
| **for** | iteration statement | **for (** *initialize; test; update* **)** *statement;*  Executes a statement (or block) repeatedly while a test condition is **true**. |
| **goto** | unused | Only **break**, **continue**, **return** and **throw** are permitted to disrupt structured flow. |
| **if** | selection statement | **if (***test***)** *statement****;***  Creates an optional execution path. The *statement* or block is executed only if *test* evaluates to **true**. |
| **if (***test***)** *statement1****;***  **else** *statement2****;***  Creates an alternative execution  path. *Statement1* is executed if *test* evaluates to true.  Otherwise, *statement2* is executed. |
| **implements** | class declaration clause | * **class** MyClass **implements** MyInterface **{** *members;* **}**   If a class *implements* an *interface*, it guarantees to define all the methods specified in the interface. This is analogous to the class extending the interface and *overriding* its methods. |
| * **class** MyClass **implements** FirstInterface, SecondInterface **{** members; **}**   A class may implement more than one interface, in which case the interfaces are separated by commas in the class declaration: |
| **import** | statement | **import** *package*.*class***; import**  *package*.\***;**  Lets you access classes and interfaces that have already been created, especially the Java standard classes. |
| **instanceof** | operator | if **(**anObject **instanceof** MyClass**)** statement**;**  Determines whether the left-hand argument is an instance of a named class or interface or an instance of a *subclass* of the named class/interface. Evaluates to **true** or **false**. |
| **int** | primitive type | **int** hundredsOfMillionsOfThings;  Int variables may be assigned 32-bit integer values. The largest possible **int** value is 2147483647, about two billion. |
| **interface** | interface declaration | **interface** MyInterface **{** members; **}**  A specification of the methods which a class must in order to implement the interface. |
| **long** | primitive type | **long** billionsOfBillionsOfThings**;**  Long variables may be assigned 64-bit integer values. The largest possible long value is 9223372036854775807L (note the "L" suffix), about nine billion, billion. |
| **native** | method modifier | **native** void aProcedure()**;**  Identifies methods that have bodies implemented outside of a Java class. |
| **new** | operator |  MyClass variable=**new** MyClass()**;**  Creates a new instance of a class. |
| * MyInterface variable=**new** MyInterface(){ members; }**;**   Creates an instance of an anonymous class that is assignment compatible with the named class/interface reference variables. |
| * int[] anArray= **new** int[5]**;**   Allocates memory for an array. |
| There is no free, delete nor destructor mechanism within the Java language. Objects become subject to garbage collection when they are no longer referenced anywhere. All instances of a class do have a predefined (but not reserved) method, *finalizer*(), that is called prior to garbage collection, but there is no mechanism for forcing destruction. |
| **null** | reference literal | MyClass variable=**null;**  Null is the reference value that does not indicate an object. The null value is assignment compatible with all reference variables. |
| **package** | statement | **package** mypackage**;**  Declares the classes defined within a Java source file as parts of a named package. Packages are used by other source files through the **import** statement. |
| **private** | member scope modifier | **private** int aField;  Private members may only be referenced within the class in which they are declared. |
| **protected** | member scope modifier | **protected** void aProcedure()**;**  Protected members may be referenced within the class in which they are declared, classes in the same package, or subclasses. |
| **public** | class/interface scope modifier | * **public** class MyClass **{** members; **}** * **public** interface MyInterface **{** members; **}**   Public classes and interfaces may be referenced outside the package in which they are declared. |
| member scope modifier | * **public** int getField()**{** statements; **}**   Public members of public classes may be referenced outside the package in which the class is declared. |
| **return** | control statement | * **return;**   Exits from a void method. This statement is optional in a void method. If not present, the void method still exits when it reaches the end of the method body. |
| * **return** expression**;**   Exits a method that returns a value. |
| **short** | primitive type | **short thousandsOfThings;**  Short variables may be assigned 16 bit integer values. |
| **static** | member modifier | Associates members with the class and not with the individual instances.   * **static** int numberOfInstances;   Static variables are 'shared' between all the instances of a class (as opposed to 'instance variables', for which each instance has an independant value). |
| * **static** void aStaticProcedure()**{** members; **}**   Static methods may not refer to instance variables (only to static variables), and may be invoked using the class name, rather than on an instance of the class. ("MyClass.aStaticProcedure()") |
| **strictfp** | class/interface/member modifier (added in Java 2) | **strictfp** double accountBalance;  Ensures that floating point values are normalized during calculations rather than just when assigned to floating point variables. |
| **super** | class reference | * **super();**   Invokes the constructor of the parent class from the constructor of a subclass. Must be the first executable statement within a constructor. |
| * **super.method()**   Invokes a method of the parent class that may have been overridden in a subclass. |
| **switch** | selection statement | **switch(** expression**)**  **{**  cases; // see **case**  optional\_default\_case; // see **default**  **}**  Provides multiple entries into a sequence of statements. The **switch** *expression* (char or integer) is evaluated, and the statement following the **case** label that matches the value is executed. Subsequent statements are executed, even if they follow other **case** labels. The sequence of statements continues to be executed until **break**, **continue** or **return** is executed, or if an unhandled exception is thrown.  If no **case** matches the expression value, statements following the **default** label are executed. |
| **synchronized** | method modifier | * **synchronized** void aProcedure()   **{** statements; **}**  The **synchronized** methods of a class can only be executed by a single thread. Threads invoking a synchronized method in a class when another thread is executing a synchronized method will be suspended until the thread that acquired the "lock" on the class releases the lock. |
| statement | * **synchronized** (reference)   statement**;**  A thread will wait until a lock is acquired on the reference before executing the statement. A lock can only be acquired for any given object by a single thread at a time. |
| **this** | reference variable | aComponent.addListener(**this**);  A predefined reference variable defined for all non-static methods in a class. It provides a reference to the object whose method was called. |
| **throw** | statement (exceptions) | **throw** new Exception()**;**  Interrupts the current sequence of statements being executed, and all parent structures and methods out to the innermost **try** statement. |
| **throws** | method header clause (exceptions) | voidaProcedure**() throws** Exception  **{** statements; **}**  Identifies a procedure that either throws an exception directly, or does not handle the exception when it calls another method that throws the exception. |
| **transient** | field modifier | **transient** int temporary;  Ostensibly, **transient** identifies fields that are not "persistent" when the object is saved or "serialized". However, there is nothing in Java that takes advantage of this feature. |
| **true** | boolean literal | boolean allDogsAreAnimals = **true;** |
| **try** | statement (exceptions) | **try {** statements; **}**  **catch (**IOException e**) {** statements; **}**  **catch (**Exception e**) {** statements; **}**  **finally {** statements; **}**  Defines a section that could throw exceptions.  The statements in the **try** block are executed in sequence. If an exception occurs, and the exception class matches the type of the catch variable, then the statements in the **catch** block are executed, and the exception is deemed "handled". Regardless of whether or not an exception occurs, and whether it is handled or not, the statements in the finally **block** are executed. |
| **void** | method return type substitute | **void** aProcedure**() {** statements; **}**  Identifies a method that does not return a value. |
| **volatile** | field modifier (threads) | **volatile** int threadFlag;  Identifies fields that must be reevaluated at every reference. Most programmers prefer just to use **synchronized**. |
| **while** | iteration statement | **while (**test**)** statement**;**  Executes a *statement* or block repeatedly until a *test* condition is **true**. If the *test* condition evaluates to **false** initially, the *statement* is not executed. |

## Java Keywords With Examples

1. **abstract**

abstract keyword is used to implement the abstraction in java. A method which doesn’t have method definition must be declared asabstract and the class containing it must be declared as abstract. You can’t instantiate abstract classes. Abstract methods must be implemented in the sub classes. You can’t use abstract keyword with variables and constructors.

|  |  |
| --- | --- |
|  | *abstract class AbstractClass{*  *abstract void abstractMethod();*  *}* |

1. **assert**

assert keyword is used in the assertion statements. These statements will enable you to test your assumptions about a program. Assertion statements provide the best way to detect and correct the programming errors. Assertion statements take one boolean expression as input and assumes that this will be always true. If the boolean expression returns false, AssertionError will be thrown.

|  |  |
| --- | --- |
|  | *public class MainClass{*  *public static void main(String[] args) {*  *System.out.println("Enter your marks");*  *Scanner sc = new Scanner(System.in);*  *int marks = sc.nextInt();*  *assert marks > 35 : "FAIL";*  *}*  *}* |

1. **boolean**

boolean keyword is used to define boolean type variables. boolean type variables can hold only two values – either true or false.

*boolean isActive = true;*

|  |  |
| --- | --- |
|  |  |

1. **break**

The break keyword is used to stop the execution of a loop(for, while, switch-case) based on some condition.

|  |  |
| --- | --- |
|  | *public class MainClass{*  *public static void main(String[] args)    {*  *for (int i = 0; i < 100; i++)        {*  *System.out.println(i);*  *if(i == 50)            {*  *break;*  *}*  *}*  *}*  *}* |

1. **byte**

byte keyword is used to declare byte type of variables. A byte variable can hold a numeric value in the range from -128 to 127.

|  |  |
| --- | --- |
|  | *byte b = 50;* |

1. **switch**
2. **case**

Both switch and case keywords are used in the switch-case statement.

|  |  |
| --- | --- |
|  | public class MainClass{      public static void main(String[] args)     {          Scanner sc = new Scanner(System.in);           System.out.println("Enter Day :");            int day = sc.nextInt();          switch (day){              case 1 : System.out.println("SUNDAY");              break;              case 2 : System.out.println("MONDAY");              break;              case 3 : System.out.println("TUESDAY");              break;              case 4 : System.out.println("WEDNESDAY");              break;              case 5 : System.out.println("THURSDAY");              break;              case 6 : System.out.println("FRIDAY");              break;              case 7 : System.out.println("SATURDAY");              break;                default: System.out.println("Invalid");              break;          }      }  } |

1. **try**
2. **catch**
3. **finally**

try, catch and finally keywords are used to handle the exceptions in java. The statements which are to be monitored for exceptions are kept in the try block. The exceptions thrown by the try block are caught in the catch block.

finally block is always executed.

|  |  |
| --- | --- |
|  | *public class MainClass{*  *public static void main(String[] args) {*  *try{*  *int i = Integer.parseInt("abc");*  *}*  *catch(NumberFormatException ex){*  *System.out.println(ex);*  *}*  *finally{*  *System.out.println("This will be always executed");*  *}*  *}*  *}* |

1. Char

char keyword is used to declare primitive char type variables. char represents the characters in java.

|  |  |
| --- | --- |
|  | *char a = 'A';*  *char b = 'B';*  *char c = 'C';* |

1. **class**

class keyword is used to define the classes in java.

|  |  |
| --- | --- |
|  | *class MyClass{*  *class MyInnerClass{*  *//Inner Class*  *}*  *}* |

1. **continue**

continue keyword is used to stop the execution of current iteration and start the execution of next iteration in a loop.

|  |  |
| --- | --- |
|  | *public class MainClass{*  *public static void main(String[] args){*  *for (int i = 0; i <= 100; i++){*  *if(i % 5 != 0){*  *continue;*  *}*  *System.out.println(i);*  *}*  *}*  *}* |

1. **default**

default keyword is used to define the default methods in an interface (From Java 8). default keyword is also used in the switch-casestatements.

|  |  |
| --- | --- |
|  | *interface MyInterface{*  *public default void myDefaultMethod(){*  *System.out.println("Default Method");*  *}*  *}* |

1. **do**

do keyword is used in a do–while loop. do-while loop is used to execute one or more statements repetitively until a condition returns false.

|  |  |
| --- | --- |
|  | *public class MainClass{*  *public static void main(String[] args){*  *int a = 10;*  *int b = 20;*  *do{*  *a = a + b;*  *b = b + 10;*    *System.out.println("a = "+a);*  *System.out.println("b = "+b);*    *} while (a <= 100);*  *}*  *}* |

1. **double**

double keyword is used to declare primitive double type of variables.

|  |  |
| --- | --- |
|  | *public class MainClass{*  *public static void main(String[] args){*  *double d1 = 23.56;*  *double d2 = 56.23;*  *double d3 = d1 + d2;*  *System.out.println(d3);*  *}*  *}* |

1. **if**
2. **else**

if and else keywords are used in if-else block.

|  |  |
| --- | --- |
|  | *public* *class* *MainClass{*  *public* *static* *void* *main(String[] args){*  *Scanner sc = new* *Scanner(System.in);*   *System.out.println("Enter a string :");*   *String input = sc.next();*  *if(input.equalsIgnoreCase("JAVA")) {*  *System.out.println("It's JAVA");*  *}*  *else{*  *System.out.println("It's not JAVA");*  *}*  *}*  *}* |

1. **enum**

enum keyword is used to define enum types.

|  |  |
| --- | --- |
|  | *enum* *MyEnums{*  *A, B, C, D;*  *}* |

1. **extends**

extends keyword is used in inheritance. It is used when a class extends another class.

|  |  |
| --- | --- |
|  | *class SuperClass{*  *//Super Class*  *}*  *class SubClass extends SuperClass{*  *//Sub Class*  *}* |

1. **final**

final keyword is used when a class or a method or a field doesn’t need further modifications. final class can’t be extended, final methodcan’t be overridden and the value of a final field can’t be changed. [[***See more***](http://javaconceptoftheday.com/final-keyword-in-java/)]

|  |  |
| --- | --- |
|  | *final* *class* *FinalClass{*  *final* *int* *finalVariable = 10;*  *final* *void* *finalMethod()*  *{*  *//final method*  *}*  *}* |

1. **float**

float keyword indicates primitive float type of variables.

|  |  |
| --- | --- |
|  | *public class MainClass{*  *public static void main(String[] args)     {*  *float f1 = 45.26f;*  *float f2 = 84.25f;*  *float f3 = f2 - f1;*  *System.out.println(f3);*  *}*  *}* |

1. **for**

for loop is used to execute the set of statements until a condition is true.

|  |  |
| --- | --- |
|  | *public class MainClass{*  *public static void main(String[] args) {*  *for (int i = 0; i <= 10; i++){*  *System.out.println(i);*  *}*  *}*  *}* |

1. **implements**

implements keyword is used while implementing an interface.

|  |  |
| --- | --- |
|  | *interface MyInterface{*  *void myMethod();*  *}*  *class MyClass implements MyInterface{*  *public void myMethod()    {*  *System.out.println("My Method");*  *}*  *}* |

1. **import**

import keyword is used to import the members of a particular package into current java file.

|  |  |
| --- | --- |
|  | *import java.sql.\*;*  *import java.util.Arrays;*  *import java.util.Scanner;* |

1. **instanceOf**

instanceOf is used to check whether an object is of specified type. The syntax for using instanceOf keyword is “**Object\_Reference**instanceOf**Type**“.

|  |  |
| --- | --- |
|  | *class A{*  *}*  *public class MainClass{*  *public static void main(String[] args){*  *A a = new A();*  *if(a instanceof A){*  *System.out.println("a is of type A");*  *}*  *}*  *}* |

1. **int**

int keyword is used to declare primitive integer type of variables.

|  |  |
| --- | --- |
|  | *public* *class* *MainClass{*  *public* *static* *void* *main(String[] args)   {*  *int* *i1 = 10;*  *int* *i2 = 20;*  *int* *i3 = i1 \*  i2;*  *System.out.println(i3);*  *}*  *}* |

1. **interface**

interface keyword is used to define the interfaces in java.

|  |  |
| --- | --- |
|  | *interface MyInterface{*  *void myMethod();*  *}* |

1. **long**

long is used to define the primitive long type variables.

|  |  |
| --- | --- |
|  | *public class MainClass{*  *public static void main(String[] args){*  *long l1 = 101;*  *long l2 = 202;*  *long l3 = l1 +  l2;*  *System.out.println(l3);*  *}*  *}* |

1. **native**

native keyword is used with a method to indicate that a particular method is implemented in native code using Java Native Interfaces(JNI).

|  |  |
| --- | --- |
|  | *class* *AnyClass{*  *public* *native* *void* *anyMethod(int* *i, double* *d);*  *}* |

1. **new**

new keyword is used while creating the instances of a class.

|  |  |
| --- | --- |
|  | *class* *A{*  *}*  *public* *class* *MainClass{*  *public* *static* *void* *main(String[] args)*  *{*  *A a = new* *A();*  *}*  *}* |

1. **package**

package keyword is used to specify a package to which the current file belongs to.

|  |  |
| --- | --- |
|  | package pack1;  class A{  } |

1. **private**

private keyword is used to declare a member of a class as private. private methods and fields are visible within the class in which they are defined.

|  |  |
| --- | --- |
|  | *class A{*  *private int i = 111;   //private field*  *private void method(){*  *//private method*  *}*  *}* |

1. **protected**

protected keyword is used to declare a member of a class as protected. protected members of a class are visible within the package only, but they can be inherited to any sub classes.

|  |  |
| --- | --- |
|  | *class A{*  *protected int i = 111;   //protected field*  *protected void method()    {*  *//protected method*  *}*  *}* |

1. **public**

public keyword is used to declare the members of a class or class itself as public. public members of a class are visible from anywhere and they can be inherited to any sub classes.

|  |  |
| --- | --- |
|  | *public class A{*  *public int i = 222;   //public field*  *public A() {*  *//public constructor*  *}*  *public void method(){*  *//public method*  *}*  *}* |

1. **return**

return keyword is used to return the control back to the caller from the method.

|  |  |
| --- | --- |
|  | *class A{*  *int method(int i)    {*  *return i\*i;     //method returning a value*  *}*  *}* |

1. **short**

short keyword is used to declare primitive short type variables.

|  |  |
| --- | --- |
|  | *short s1 = 11;*  *short s2 = 22;* |

1. **static**

***static*** keyword is used to define the class level members of a class. static members of a class are stored in the class memory and you can access them directly through class name. No need to instantiate a class. [[See more](http://javaconceptoftheday.com/static-members-java/)]

|  |  |
| --- | --- |
|  | *class A{*  *static int staticField = 555;    //Static Field*  *static void staticMethod()    {*  *//Static method*  *}*  *}*  *public class MainClass{*  *public static void main(String[] args) {*  *System.out.println(A.staticField);*  *//Accessing staticField via class name*  *A.staticMethod();     //Accessing staticMethod via class name*  *}*  *}* |
|  |  |

1. **strictfp**

strictfp keyword is used to implement the strict precision of floating point calculations on different platforms. strictfp can be used with classes, interfaces and methods.

|  |  |
| --- | --- |
|  | *strictfp interface I{*  *//strictfp applied on interface*  *}*  *strictfp class C{*  *//strictfp applied on class*  *}*  *class A{*  *strictfp void method()*  *{*  *//strictfp applied on method*  *}*  *}* |

1. **super**

super keyword is used to access super class members inside a sub class.

|  |  |
| --- | --- |
|  | *class A{*  *int i;*    *public A(int i) {*  *this.i = i;*  *}*  *void methodA() {*  *System.out.println(i);*  *}*  *}*  *class B extends A{*  *public B() {*  *super(10);    //Calling super class constructor*  *}*  *void methodB(){*  *System.out.println(super.i);    //accessing super class field*  *super.methodA();    //Calling super class method*  *}*  *}* |

1. **synchronized**

synchronized keyword is used to implement the synchronization in java. only one thread can enter into a method or a block which is declared as synchronized. Any thread which wants to enter synchronized method or block must acquire object lock of those methods or blocks. [[***See more***](http://javaconceptoftheday.com/synchronization-in-java/)]

|  |  |
| --- | --- |
|  | *class AnyClass{*  *synchronized void synchronizedMethod()    {*  *//Synchronized method*  *}*  *void anyMethod() {*  *synchronized (this) {*  *//Synchronized block*  *}*  *}*  *}* |

1. **this**

this keyword is used to access other members of the same class.

|  |  |
| --- | --- |
|  | *class AnyClass{*  *int i;*  *AnyClass(){*  *System.out.println("First Constructor");*  *}*  *AnyClass(int j){*  *this();    //calling statement to First Constructor*  *System.out.println("Second Constructor");*  *}*  *void methodOne(){*  *System.out.println("From method one");*  *}*  *void methodTwo(){*  *System.out.println(this.i);  //Accessing same class field*  *this.methodOne();      //Accessing same class method*  *}*  *}* |

1. **throw**

throw keyword is used to throw the exceptions manually. [[***See more***](http://javaconceptoftheday.com/throwing-rethrowing-exception-java/)]

|  |  |
| --- | --- |
|  | *public class MainClass{*  *public static void main(String[] args)     {*  *try{*  *//throwing NumberFormatException manually*  *throw new NumberFormatException();*  *} catch(Exception ex) {*  *System.out.println(ex);*  *}*  *}*  *}* |

1. **throws**

throws keyword is used to specify the exceptions which the current method may throw.

|  |  |
| --- | --- |
|  | *class A{*  *void method() throws NumberFormatException    {*  *int i = Integer.parseInt("abc");*  *}*  *}* |

1. **transient**

transient keyword is used in serialization. A variable which is declared as transient will not be eligible for serialization.

|  |  |
| --- | --- |
|  | *class MyClass implements Serializable{*  *int a;*  *transient String s;   //This will not be serialized*  *double d;*  *}* |

1. **void**

void keyword is used to indicate that method returns nothing.

|  |  |
| --- | --- |
|  | *class A{*  *void methodReturnsNothing()    {*  *//Method returns no value*  *}*  *}* |

1. **volatile**

volatile keyword is used in the concurrent programming. The value of a variable which is declared as volatile will be written into or read from the main memory.

|  |  |
| --- | --- |
|  | *class A{*  *public volatile int counter = 0;*  *}* |

1. **while**

while keyword is used in the while loop.

|  |  |
| --- | --- |
|  | *public* *class* *MainClass{*  *public* *static* *void* *main(String[] args) {*  *int* *i = 10;*  *while* *(i <= 100)   {*  *System.out.println(i);*  *i = i + 10;*  *}*  *}*  *}* |

1. **goto**
2. **const**

Both goto and const are reserved words in java but they are currently not used.

**Note** : true, false and null are not the keywords. They are literals in java.

### Interview Questions and Answers on Integer and Long

**Q1.  What are the Wrapper classes available for primitive types ?**  
  
Ans.

*boolean  - java.lang.Boolean  
byte - java.lang.Byte  
char - java.lang.Character  
double - java.lang.Double  
float - java.lang.Float****int - java.lang.Integer******long - java.lang.Long*** *short - java.lang.Short  
void - java.lang.Void*

**Q2.  Difference between long.Class and Long.TYPE ?**  
  
Ans.

They both represent the long primitive type. They are exactly the same.  
  
**Q3.  What are the default or implicitly assigned values for data types in java ?**  
  
Ans.

boolean ---> false  
byte ----> 0  
short ----> 0  
**int** -----> 0  
**long** ------> 0l  
char -----> /u0000  
float ------> 0.0f  
double ----> 0.0d  
any object reference ----> null  
  
**Q4.  Will this code give error if i try to add two heterogeneous elements in the arraylist. ? and Why ?**  
  
List list1 = new ArrayList<>();  
list1.add(5);  
list1.add("5");  
  
Ans.

If we don't declare the list to be of specific type, it treats it as list of objects.  
int 1 is auto boxed to Integer and "1" is String and hence both are objects.  
  
**Q5.  How to find whether a given integer is odd or even without use of modules operator in java?**  
  
Ans.

public static void main(String ar[]){  
     int n=5;  
     if((n/2)\*2==n) {  
          System.out.println("Even Number ");  
     } else{  
          System.out.println("Odd Number ");  
     }  
}  
  
**Q6.  Is it legal to initialize List like this ?**  
  
LinkedList<Integer> l=new LinkedList<int>();   
  
Ans.

No, Generic parameters cannot be primitives.  
  
**Q7.  What will the following code print ?**  
  
public static void main(String[] args){  
 Integer i1 = new Integer("1");  
        Integer i2 = new Integer("2");  
        Integer i3 = Integer.valueOf("3");   
        int i4 = i1 + i2 + i3;   
        System.out.println(i4);    
}  
  
Ans.

6  
  
**Q8.  Which of the following syntax are correct ?**  
  
a. LinkedList<Integer> l=new LinkedList<int>();  
b. List<Integer> l=new LinkedList<int>();  
c. LinkedList<Integer> l=new LinkedList<Integer>();  
d. List<Integer> l = new LinkedList<Integer>();  
  
Ans.

c and d are correct.  
  
**Q9.  Can we compare Integers by using equals() in Java ?**  
  
Ans.

Yes for the Wrapper class Integer but not for the primitive int.  
  
**Q10.  Is this valid in Java ?**  
  
Long x = new Long ("42");  
  
Ans.

Yes. Long wrapper class has overloaded constructor which takes String as input and then translate it to the long value and stored it as long.  
  
**Q11.  What is the size of long data type ?**  
  
 a. 16 bit  
 b. 32 bit  
 c. 64 bit  
 d. 128 bit  
  
Ans.

64 bit  
  
**Q12. Which interfaces are implemented by  Integer?**  
  
Ans.

[Comparable]  
  
**Q13. What is the package name for Long class?**  
  
Ans.

java.lang  
  
**Q14. Which is the Parent Class of Long class?**  
  
Ans.

java.lang.Number

### Interview Questions and Answers on Long

**Q1.  What are the Wrapper classes available for primitive types ?**  
  
Ans.

boolean  - java.lang.Boolean  
byte - java.lang.Byte  
char - java.lang.Character  
double - java.lang.Double  
float - java.lang.Float  
int - java.lang.Integer  
**long - java.lang.Long**  
short - java.lang.Short  
void - java.lang.Void  
  
**Q2.  Difference between long.Class and Long.TYPE ?**  
  
Ans.

They both represent the long primitive type. They are exactly the same.  
  
**Q3.  What are the default or implicitly assigned values for data types in java ?**  
  
Ans.

boolean ---> false  
byte ----> 0  
short ----> 0  
int -----> 0  
**long**------> 0l  
char -----> /u0000  
float ------> 0.0f  
double ----> 0.0d  
any object reference ----> null  
  
**Q4.  Is this valid in Java ?**  
  
Long x = new Long ("42");  
  
Ans.

Yes. Long wrapper class has overloaded constructor which takes String as input and then translate it to the long value and stored it as long.  
  
**Q5.  What is the size of long data type ?**  
  
 a. 16 bit  
 b. 32 bit  
 c. 64 bit  
 d. 128 bit  
  
Ans.

64 bit  
  
**Q6. What is the package name for Long class?**  
  
Ans.

java.lang  
  
**Q7. Which is the Parent Class of Long class?**  
  
Ans.

java.lang.Number

### Java - Interview Questions and Answers on Primitive

**Q1.  What are the Wrapper classes available for primitive types ?**

Ans.   
boolean  - java.lang.Boolean

byte - java.lang.Byte

char - java.lang.Character

double - java.lang.Double

float - java.lang.Float

int - java.lang.Integer

long - java.lang.Long

short - java.lang.Short

void - java.lang.Void

**Q2.  Difference between long.Class and Long.TYPE ?**

Ans.

They both represent the long primitive type. They are exactly the same.

**Q3.  What are wrapper classes ?**

Ans.

They are wrappers to primitive data types. They allow us to access primitives as objects.

**Q4.  What is casting?**

Ans.

There are two types of casting, casting between primitive numeric types and casting between object references. Casting between numeric types is used to convert larger values, such as double values, to smaller values, such as byte values. Casting between object references is used to refer to an object by a compatible class, interface, or array type reference

**Q5.  Does Declaring an object "final" makes it immutable ?**

Ans.

Only declaring primitive types as final makes them immutable. Making objects final means that the object handler cannot be used to target some other object but the object is still mutable.

**Q6.  Difference between boolean and Boolean ?**

Ans.

boolean is a primitive type whereas Boolean is a class.

**Q7.  Explain Autoboxing ?**

Ans.

Autoboxing is the automatic conversion that the Java compiler makes between the primitive types and their corresponding object wrapper classes

**Q8.  What are Wrapper Classes ? What are Primitive Wrapper Classes ?**

Ans.

A wrapper class is any class which "wraps" or "encapsulates" the functionality of another class or component. A Wrapper Class that wraps or encapsulates the primitive data type is called Primitive Wrapper Class.

**Q9.  What are the Disadvantages of using Collection Classes over Arrays ?**

Ans.

Collections can only hold objects, It can't hold primitive data types.

Collections have performance overheads as they deal with objects and offer dynamic memory expansion. This dynamic expansion could be a bigger overhead if the collection class needs consecutive memory location like Vectors.

Collections doesn't allow modification while traversal as it may lead to concurrentModificationException.

**Q10.  Is it legal to initialize List like this ?**

LinkedList<Integer> l=new LinkedList<int>();

Ans.

No, Generic parameters cannot be primitives.

**Q11.  Can we compare Integers by using equals() in Java ?**

Ans.

Yes for the Wrapper class Integer but not for the primitive int.

### Java - Data Types - Interview Questions and Answers on Float

**Q1.  Which is the Wrapper classes available for float ?**

Ans.

java.lang.Float

**Q2.  What is the default assigned value for float ?**

Ans.

0.0f

**Q3.  What is the difference between float and double?**

Ans.

Float can represent up to 7 digits accurately after decimal point, whereas double can represent up to 15 digits accurately after decimal point.

**Q4. Which interfaces are implemented by Float?**

Ans.

[, Serializable, Comparable<Float>]

**Q5. What is the package name for Float class?**

Ans.

java.lang

**Q6. Which is the Parent Class of Float class?**

Ans.

java.lang.Number

### Interview Questions and Answers on Boolean / boolean

**Q1.  Variable of the boolean type is automatically initialized as?**

Ans.

The default value of the boolean type is false.

**Q2. What is the difference between boolean and Boolean ?**

Ans.

boolean is a promitive type whereas Boolean is a wrapper class.

**Q3.  What are the Wrapper classes available for primitive types ?**

Ans.

**boolean  - java.lang.Boolean**

byte - java.lang.Byte

char - java.lang.Character

double - java.lang.Double

float - java.lang.Float

int - java.lang.Integer

long - java.lang.Long

short - java.lang.Short

void - java.lang.Void

**Q4.  Which data types are supported by JSON ?**

Ans.

Number

String

**Boolean**

Array

Object

null

**Q5.  What are the default or implicitly assigned values for data types in java ?**

Ans.

boolean ---> false

byte ----> 0

short ----> 0

int -----> 0

long ------> 0l

char -----> /u0000

float ------> 0.0f

double ----> 0.0d

any object reference ----> null

**Q6. What is the use of Boolean method valueOf(String s) ?**

Ans.

This method returns the boolean value for the specified string. The returned value represents a true value if the string argument is not Null and is equal, ignoring case, to the string "true".

### Interview Questions and Answers on Double

**Q1.  What are the Wrapper classes available for double primitive type ?**

Ans. java.lang.Double

**Q2.  What is the default assigned values for double in java ?**

Ans. 0.0d

**Q3.  What is the difference between float and double?**

Ans. Float can represent up to 7 digits accurately after decimal point, whereas double can represent up to 15 digits accurately after decimal point.

**Q4.  What is the size of double type ?**

 a. 16 bit

 b. 32 bit

 c. 64 bit

 d. 128 bit

Ans. 64 bit

### Interview Questions and Answers on Data Types

**Q1.  What are the Wrapper classes available for primitive types ?**

Ans.

boolean  - java.lang.Boolean

byte - java.lang.Byte

char - java.lang.Character

double - java.lang.Double

float - java.lang.Float

int - java.lang.Integer

long - java.lang.Long

short - java.lang.Short

void - java.lang.Void

**Q2.  Difference between long.Class and Long.TYPE ?**

Ans. They both represent the long primitive type. They are exactly the same.

**Q3.  What are wrapper classes ?**

Ans. They are wrappers to primitive data types. They allow us to access primitives as objects.

**Q4.  What is casting?**

Ans. There are two types of casting, casting between primitive numeric types and casting between object references. Casting between numeric types is used to convert larger values, such as double values, to smaller values, such as byte values. Casting between object references is used to refer to an object by a compatible class, interface, or array type reference

**Q5.  What are the default or implicitly assigned values for data types in java ?**

Ans.

boolean ---> false

byte ----> 0

short ----> 0

int -----> 0

long ------> 0l

char -----> /u0000

float ------> 0.0f

double ----> 0.0d

any object reference ----> null.  
  
**Q6.  What is the advantage of using arrays over variables ?**  
Ans.

Arrays provide a structure wherein multiple values can be accessed using single reference and index. This helps in iterating over the values using loops.  
  
**Q7.  What are the disadvantages of using arrays ?**  
Ans.

Arrays are of fixed size and have to reserve memory prior to use. Hence if we don't know size in advance arrays are not recommended to use.  
  
Arrays can store only homogeneous elements.  
  
Arrays store its values in contentious memory location. Not suitable if the content is too large and needs to be distributed in memory.   
  
There is no underlying data structure for arrays and no ready made method support for arrays, for every requriment we need to code explicitly  
  
**Q8.  Advantage of Collection classes over Arrays ?**  
Ans.

Collections are re-sizable in nature. We can increase or decrease the size as per recruitment.  
Collections can hold both homogeneous and heterogeneous data's.  
Every collection follows some standard data structures.  
Collection provides many useful built in methods for traversing,sorting and search.   
  
**Q9.  What are the Disadvantages of using Collection Classes over Arrays ?**  
Ans.

Collections can only hold objects, It can't hold primitive data types.  
  
Collections have performance overheads as they deal with objects and offer dynamic memory expansion. This dynamic expansion could be a bigger overhead if the collection class needs consecutive memory location like Vectors.  
  
Collections doesn't allow modification while traversal as it may lead to concurrentModificationException.  
  
**Q10. What is the difference between Data Type and Data Structure ?**  
Ans.   
Data type: a set of values together with operations on that type   
Data structure: a physical implementation of a data type  
  
**Q11. Is it correct to say that Interfaces are abstract data types ?**  
Ans.

No.  
Data Type holds data whereas Interface doesn't hold anything. Interface is a contract about how to communicate with the underlying Class.

**Q12. Do you see Class as a Data Type or Data Structure ?**

Ans.

Class can be better seen as Data Type. This could be implemented as a Data Structure too in some cases.

One thing worth understanding here is that Data type and Data structure are conceptual things. Class could be implementation of either of these.

**Question 1:**

class Main {

   public static void main(String args[]) {

         int t;

         System.out.println(t);

    }

}

**Question 1 Explanation**:

Unlike class members, local variables of methods must be assigned a value to before they are accessed, or it is a compile error.

|  |
| --- |
| **Question 2** |

Predict the output of following Java program.

|  |
| --- |
| class Test {      public static void main(String[] args) {        for(int i = 0; 0; i++)        {            System.out.println("Hello");            break;        }      }  } |

**Question 2 Explanation:**

The error is in for loop where 0 is used in place of boolean value. Unlike C++, use of non boolean variables in place of bool is not allowed

|  |
| --- |
| **Question 3** |

Predict the output of the following program.

|  |
| --- |
| class Test  {      public static void main(String[] args)      {          Double object = new Double("2.4");          int a = object.intValue();          byte b = object.byteValue();          float d = object.floatValue();          double c = object.doubleValue();            System.out.println(a + b + c + d );        }  } |

**Question 3 Explanation:**

Arithmetic conversions are implicitly performed to cast the values to a common type. The compiler first performs integer promotion. If the operands still have different types, then they are converted to the type that appears highest in the hierarchy.

1. which of the following is the smallest integer data type ?

[A] int

[B] long

[C] **byte**

[D] short

**2.** which of the following is not a primitive data type.

**[A]** byte

**[B]** short

**[C] enum**

**[D]** int

**3.** which value cannot store character data type.

**[A]** Digit

**[B]** Special character

**[C]** String

**[D] Letter**

Answer: C. byte

**Answer:** C. byte

**4.** which of the following data types comes under floating data types ?

**[A]** byte

**[B]** int

**[C] float**

**[D]** long

**5.** Range of byte data type is ?

**[A]** -128 to 255

**[B]** -128 to 256

**[C] -128 to 127**

**[D]** -127 to 128

**6.** In order to fetch stream of data from network or file, following data type used ?

**[A]** double

**[B]** char

**[C]** int

**[D] byte**

**7.** what is size of integer in java programming.

**[A] 4 bytes**

**[B]** 5 bytes

**[C]** 2 bytes

**[D]** 8 bytes

**8.** How many byte will byte data type take in java ?

**[A] 1**

**[B]** 2

**[C]** 3

**[D]** 4

**9.** which data type can store  64 bit value ?

**[A] double**

**[B]** int

**[C]** long

**[D]** float

**10.** Short data type has a minimum value of……….. ?

**[A]** -32767

**[B]** -2,147,483,648

**[C]** -**32768**

**[D]** -127

An expression involving byte, int, and literal numbers is promoted to which of these?

**[A]** int

**[B]** long

**[C]** byte

**[D]** float

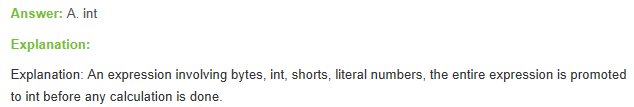
**Answer:** A. int

**Explanation:**

Explanation: An expression involving bytes, int, shorts, literal numbers, the entire expression is promoted to int before any calculation is done

**Answer:** A. int

**Explanation:**

Explanation: An expression involving bytes, int, shorts, literal numbers, the entire expression is promoted to int before any calculation is done.

What is the output of following code snippet ?

public class NumberSystem{

public static void main(String[] args){

int val = 0b11010;

System.out.println("Value : " + val);

}

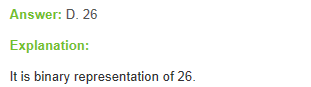
}

**[A]** 32

**[B]** 64

**[C]** 28

**[D]** 26

It is binary representation of 26.

**Data type mcq questions of java and answers**  
(1)

class Datatype {

    public static void main(String[] args) {

         byte num=(byte)130;

         System.*out*.print(num);

    }

}

What will be the output when you compile and run the above code?

(a)130

(b)3

(c)-126

(d)Compiler error

Answer: (c)

(2)

class Datatype {

    public static void main(String[] args) {

         byte num=130;

         System.out.print(num);

    }

}

What will be the output when you compile and run the above code?

(a)130

(b)3

(c)-126

(d)Compiler error

Answer: (d)

(3)

class Datatype {

    public static void main(String[] args) {

         byte number=0101;

         System.out.print(number);

    }

}

What will be the output when you compile and run the above code?

(a)101

(b)65

(c)5

(d)Compiler error

Answer: (b)

(4)

class Datatype {

    public static void main(String[] args) {

         short num=0x8fff;

         System.out.print(num);

    }

}

What will be the output when you compile and run the above code?

(a)589823

(b)Infinity

(c)NaN

(d)Compiler error

Answer: (d)

(5)

class Datatype {

    public static void main(String[] args){

         char num=65;

         System.out.println(num);

    }

}

What will be the output when you compile and run the above code?

(a)65

(b)A

(c)0

(d)Compiler error

Answer: (b)

Guess the output of the following program -

public class NumberSystem{

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

**int** hexVal = 0x1a;

System.out.println("Value : " + hexVal);

}

}

**Explanation:**

0x1a = 0001 1010

= 2^1 + 2^3 + 2^4

= 2 + 8 + 16

= 26

Guess the output of the following program -

public class NumberSystem{

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

**int** val = 0b11010;

System.out.println("Value : " + val);

}

}

**Explanation:**

Its binary representation of 26. Binary Literal is accepted in latest JDK 7.

**Note:**

We can construct Boolean Object from boolean and String.

Compiler never assigns a default value to an uninitialized local variable in Java Programming !!

"Accessing an uninitialized local variable will result in a compile-time error in Java Programming"

What is Literal in Java Programming ?

* source code representation of a fixed value

From Java SE 7 and later, binary literal can be created.

What is transient variable?

A variable is said to be ‘transient’, which is not participated in object serialization. The variable with ‘transient’ access modifier is not persisted onto the object. Frequently updating values in a Java application will not be serialized by practice. Such variables can be specified as transient.

What is transient variable?

A variable declared as "transient" in a Serializable class cannot be serialized although the class is declared as serializable.  
  
When the class is written to an ObjectStream, the value of the variable can't be written to the stream and when the class is retrieved from the ObjectStream the value of the variable becomes null.

What is the difference between static and non-static variables?

**Static Variables:**  
- A static variable is associated with the class has only one copy per class but not for each object. An instance of a class does not have static variables.  
  
- Static variables can be accessed by static or instance methods  
  
- Memory is allocated when the class is loaded in context area at run time.  
  
**Non-Static Variables:**  
- Non-static variables will have one copy each per object. Each instance of a class will have one copy of non-static variables.  
  
- Instance variables can be accessed only by the instance methods.  
  
- Instance variables are allocated at compile time.

What is the difference between static and non-static variables?

- A static variable is shared among all instances of a class.  
  
- A non-static variable is specific to a single instance of that class.  
  
**Eg**: A static variable can be shared by all users for the current running system. It’s a globally available value and can be used by all users.

**Interview Questions and Answers on Java Keywords**

**Q1.  What is a Final Variable ?**

Ans. Final variable is a variable constant that cannot be changed after initialization.

**Q2.  What is a final method ?**

Ans. Its a method which cannot be overridden. Compiler throws an error if we try to override a method which has been declared final in the parent class.

**Q3.  What is a Final Class ?**

Ans. A Class that cannot be sub classed.

**Q4.  What are the common uses of "this" keyword in java ?**

Ans. "this" keyword is a reference to the current object and can be used for following -

1. Passing itself to another method.

2. Referring to the instance variable when local variable has the same name.

3. Calling another constructor in constructor chaining.

**Q5.  What are transient variables in java?**

Ans. Transient variables are variable that cannot be serialized.

**Q6.  Explain static blocks in Java ?**

Ans. A static initialization block is a normal block of code enclosed in braces, { }, and preceded by the static keyword. Here is an example:

static {

    // whatever code is needed for initialization goes here

}

A class can have any number of static initialization blocks, and they can appear anywhere in the class body. The runtime system guarantees that static initialization blocks are called in the order that they appear in the source code.

**Q7.  What is Volatile keyword used for ?**

Ans. Volatile is a declaration that a variable can be accessed by multiple threads and hence shouldn't be cached.

**Q8.  What is the use of Transient Keyword ?**

Ans. It in Java is used to indicate that a field should not be serialized.

**Q9.  Does Declaring an object "final" makes it immutable ?**

Ans. Only declaring primitive types as final makes them immutable. Making objects final means that the object handler cannot be used to target some other object but the object is still mutable.

**Q10.  Explain the use of "Native" keyword ?**

Ans. Used in method declarations to specify that the method is not implemented in the same Java source file, but rather in another language

**Q11.  What is "super" used for ?**

Ans. Used to access members of the base class.

**Q12.  What is "this" keyword used for ?**

Ans. Used to represent an instance of the class in which it appears.

**Q13.  What is "Import" used for ?**

Ans. Enables the programmer to abbreviate the names of classes defined in a package.

**Q14.  What is a Static import ?**

Ans. By static import , we can access the static members of a class directly without prefixing it with the class name.

**Q15.  Can we use both "this" and "super" in a constructor ?**

Ans. No, because both this and super should be the first statement.

**Q16.  Is it necessary that each try block to be followed by catch block ?**

Ans. It should be followed by either catch or finally block.

**Q1) What are different types of access modifiers in Java?**

Ans) There are four different types of modifiers:

|  |  |  |
| --- | --- | --- |
| Modifer | Accessible in the same package | Accessible in different package |
| Private | No | No |
| Protected | Yes | Yes, only if the class extends the main class |
| Default | Yes | No |
| Public | Yes | Yes |

**Q2) What is the use of final keyword?**

Ans) The final keyword can be assigned to

1. Class level variable
2. method
3. class
4. Objects

If a final is assigned to a variable, the variable behaves as a constant. It means that the value of variable once set cannot be changed.

final int i = 1;

i = 5; // error

If a final is assigned to a method then it cannot be overridden in its child class.

class Parent {

final void print() {

System.out.println("Inside");

}

}

class Child extends Parent {

public final void print() {

// error cannot override final method

System.out.println("Inside");

}

}

If a class is made as final, then no other class can extend it and make it as parent class. E.g. String Class.

Final objects are instantiated only once. i.e

final Map map = new HashMap();

map.put("key";,"value");

map = new HashMap(); // error

**Q3) What is use of synchronized keyword?**

Ans) This keyword is used to prevent concurrency. Synchronized keyword can be applied to static/non-static methods or a block of code. Only one thread at a time can access synchronized methods and if there are multiple threads trying to access the same method then other threads have to wait for the execution of method by one thread. Synchronized keyword provides a lock on the object and thus prevents race condition. E.g.

public void synchronized simpleMethod() {}

public void synchronized staticMethod() {}

public void myMethod() {

synchronized (this) { // synchronized keyword on block of code

}

}

**Q4) What is volatile keyword?**

Ans) In general each thread has its own copy of variable, such that one thread is not concerned with the value of same variable in the other thread. But sometime this may not be the case. Consider a scenario in which the count variable is holding the number of times a method is called for a given class irrespective of any thread calling, in this case irrespective of thread access the count has to be increased. In this case the count variable is declared as volatile. The copy of volatile variable is stored in the main memory, so every time a thread access the variable even for reading purpose the local copy is updated each time from the main memory. The volatile variable also have performance issues.

**Q5) What is a transient variable?**

Ans) If some of the properties of a class are not required to be serialized then the varaibles are marked as transient. When an object is deserialized the transient variables retains the default value depending on the type of variable declared and hence lost its original value.

**Q6) What is a strictfp modifier?**

Ans) Strictfp is used with variable only . It is used to restrict floating point calculations ( fp ) to ensure portability ( platform Independent ). When this modifier is specified, the JVM adheres to the Java specifications ( IEEE-754 floating-point specification ) and returns the consistent value independent of the platform. That is, if you want the answers from your code (which uses floating point values) to be consistent in all platforms, then you need to specify the strictfp modifier.

**Q7) What is a static variable?**

Ans) Static keyword can be used with the variables and methods but not with the class. Anything declared as static is related to class and not objects.

**Static variable** : Multiples objects of a class shares the same instance of a static variable.Consider the example:

public class Counter {

private static int count=0;

private int nonStaticcount=0;

public void incrementCounter() {

count++;

nonStaticcount++;

}

public static int getCount() {

return count;

}

public int getNonStaticcount() {

return nonStaticcount;

}

public static void main(String args[]) {

Counter countObj1 = new Counter();

Counter countObj2 = new Counter();

countObj1.incrementCounter();

countObj2.incrementCounter();

System.out.println("Static count for Obj1: "+countObj1.getCount());

System.out.println("NonStatic count for Obj1: "+countObj1.getNonStaticcount());

System.out.println("Static count for Obj2: "+countObj2.getCount())

System.out.println("NonStatic count for Obj2: "+countObj2.getNonStaticcount())

}

Output

Static count for Obj1: 2

NonStatic count for Obj1: 1

Static count for Obj2: 2

NonStatic count for Obj2: 1

In the above program obj1 and obj2 share the same instance of static variable count hence if the value is incremented by one object , the incremented value will be reflected across the other objects.

**Q8) What is a static method?**

Ans)A method defined as static is called static method. A static method can be accessed without creating the objects. Just by using the Class name the method can be accessed.

Static method can only access static variables and not local or global non-static variables. For eg:

public class Test{

public static void printMe() {

System.out.println("Hello World");

}

}

public class MainClass {

public static void main(String args[]) {

Test.printMe()

}

}

Output:

Hello World

Also static method can call only static methods and not non static methods. But non-static methods can call static mehtods.

**Q9) Why static methods cannot access non static variables or methods?**

Ans) A static method cannot access non static variables or methods because static methods can be accessed without instantiating the class, so if the class is not instantiated the variables are not intialized and thus cannot be accessed from a static method.

**Q10) What is static class ?**

Ans) A class cannot be declared static except inner class. But a class can be said a static class if all the variables and methods of the class are static and the constructor is private. Making the constructor private will prevent the class to be instantiated. So the only possibility to access is using Class name only

**Q9) What is throw keyword?**

Ans) Throw keyword is used to throw the exception manually. It is mainly used when the program fails to satisfy the given condition and it wants to warn the application.The exception thrown should be subclass of Throwable.

public void parent() {

try {

child();

} catch(MyCustomException e) {

//oops error is here

}

}

public void child() throws MyCustomException {

if(2%2 == 0) {

throw new MyCustomException("exception using throw keyword");

}

}

**Q10) What is use of throws keyword?**

Ans) Throws clause is used to throw the exception from a method to the calling method which could decide to handle exception or throw to its calling method in a class.

public void parent(){

try {

child();

} catch(MyCustomException e) { }

}

public void child throws MyCustomException {

//put some logic so that the exception occurs.

}