**Assessment**

**Case-1 :**

Parse the give command line and display in the below format

|  |  |  |
| --- | --- | --- |
| **Input: parser.exe 1 a sss 4.5643** | |  |
| **Output:** |  |  |
| **Type** | **Value** | **Size of** |
| **Char** | **A** | **X** |
| **Int** | **1** | **X** |
| **String** | **Sss** | **X** |
| **Float/double** | **4.5643** | **X** |

Note: X means are actual sizeof() value depends on the OS.

**Program:-**

**public class Pro1 {**

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

**// TODO Auto-generated method stub**

**//Ingeneral as Java has fixed datatypes, sizeof command is not worked in Java**

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

**{**

**try**

**{**

**int a=Integer.parseInt(args[i]);**

**System.out.println(a+" integer "+Integer.SIZE/8);**

**}**

**catch(Exception e1)**

**{**

**try**

**{**

**System.out.println(Float.parseFloat(args[i])+" float/double "+Float.SIZE/8);**

**}**

**catch(Exception e2)**

**{**

**if(args[i].length()==1)**

**{**

**System.out.println(args[i]+" char "+Character.SIZE/8);**

**}**

**else**

**{**

**System.out.println(args[i]+" string "+args[i].length());**

**//String is non primitive datatype so no default size is not specified.**

**}**

**}**

**}**

**}**

**}**

**}**

**Note:** Java does not provide anything like C's sizeof(). However, let's consider why a Java programmer might occasionally want it.

Java object allocation and construction are tied together (it is impossible to use an allocated but uninitialized object instance). If a Java class defines fields that are references to further objects, it is also common to set them at construction time. Allocating a Java object therefore frequently allocates numerous interconnected object instances: an object graph. Coupled with automatic garbage collection, this is all too convenient and can make you feel like you never have to worry about Java memory allocation details.

**Case 2:**

**Find the formula for give expression and write the program for the given input. Expression: 1+4+9+16+….+100**

**Input: 4**

**Output: 30**

**Program:**

**import** java.io.BufferedReader;

**import** java.io.IOException;

**import** java.io.InputStreamReader;

**public** **class** MaExpre {

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

**int** num,temp = 0;

// **TODO** Auto-generated method stub

System.***out***.println("Please enter the itteration number ");

//Scanner sc =new Scanner(System.in);

BufferedReader brr=**new** BufferedReader(**new** InputStreamReader(System.***in***));

num = Integer.*parseInt*(brr.readLine());

**for**(**int** itteration=1;itteration<=num;itteration++) {

temp += (itteration\*itteration);

}

System.***out***.println("Output: "+temp);

}

}

Output:

Please enter the itteration number

4

Output: 30

**Case 3:**

**Write your own program that explains the below.**

**Variable Scope Modifier Types Storage Classes**

**Variable Scope:-** Scope of a variable is the part of the program where the variable is accessible. Like C/C++, in Java, all identifiers are lexically (or statically) scoped, i.e.scope of a variable can determined at compile time and independent of function call stack.Java programs are organized in the form of classes. Every class is part of some package.

**Java scope rules can be covered under following categories.**

**Member Variables (Class Level Scope)**

These variables must be declared inside class (outside any function). They can be directly accessed anywhere in class. Let’s take a look at an example:

public class Test

{

// All variables defined directly inside a class

// are member variables

int a;

private String b

void method1() {....}

int method2() {....}

char c;

}

 We can declare class variables anywhere in class, but outside methods.

 Access specified of member variables doesn’t effect scope of them within a class.

 Member variables can be accessed outside a class with following rules

Modifier Package Subclass World

public Yes Yes Yes

protected Yes Yes No

Default (no

modifier) Yes No No

private No No No

**Local Variables (Method Level Scope)**

Variables declared inside a method have method level scope and can’t be accessed outside the method.

public class Test

{

void method1()

{

// Local variable (Method level scope)

int x;

}

}

**Note :** Local variables don’t exist after method’s execution is over.  
Here’s another example of method scope, except this time the variable got passed in as a parameter to the method:

class Test

{

private int x;

public void setX(int x)

{

this.x = x;

}

}

The above code uses this keyword to differentiate between the local and class variables

**Modifier Types: There are two types of access modifiers in java**

Access Modifiers in Java

* Private access modifier
* Role of private constructor
* Default access modifier
* Protected access modifier
* Public access modifier

**Access Modifier with Method Overriding**

There are two types of modifiers in Java: access modifiers and non-access modifiers.

The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

**There are four types of Java access modifiers**:

Private: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.

Default: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.

Protected: The access level of a protected modifier is within the package and outside the package through child class. If you do noNt make the child class, it cannot be accessed from outside the package.

Public: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

There are many non-access modifiers, such as static, abstract, synchronized, native, volatile, transient, etc. Here, we are going to learn the access modifiers only.

**Understanding Java Access Modifiers**

Let's understand the access modifiers in Java by a simple table.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Access Modifiers** | **within class** | **Withinpackage** | **outside package by subclass only** | **Outside package** |
| **Private** | **Y** | **N** | **N** | **N** |
| **Default** | **Y** | **Y** | **N** | **N** |
| **Protected** | **Y** | **Y** | **Y** | **N** |
| **Public** | **Y** | **Y** | **Y** | **Y** |

**Non-Access Modifiers:**

Java provides a number of non-access modifiers to achieve many other functionality.

• The static modifier for creating class methods and variables.

• The final modifier for finalizing the implementations of classes, methods, and variables.

• The abstract modifier for creating abstract classes and methods.

• The synchronized and volatile modifiers, which are used for threads.

**Case 4:**

**What is the purpose and difference in public, private and protected access specifiers. Illustrate a design example with this.**

**Public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

**Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.

**Protected**: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.

**Simple example of private access modifier**

In this example, we have created two classes A and Simple. A class contains private data member and private method. We are accessing these private members from outside the class, so there is a compile-time error.

class A{

private int data=40;

private void msg(){System.out.println("Hello java");}

}

public class Simple{

 public static void main(String args[]){

   A obj=new A();

   System.out.println(obj.data);//Compile Time Error

   obj.msg();//Compile Time Error

   }

}

Role of Private Constructor

If you make any class constructor private, you cannot create the instance of that class from outside the class. For example:

**class A{**

**private A(){}//private constructor**

**void msg(){System.out.println("Hello java");}**

**}**

**public class Simple{**

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

**A obj=new A();//Compile Time Error   }  }**

**Simple Example of protected access modifier**

In this example, we have created the two packages pack and mypack. The A class of pack package is public, so can be accessed from outside the package. But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

//save by A.java

package pack;

public class A{

protected void msg(){System.out.println("Hello");}

}

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

//save by B.java

package mypack;

import pack.\*;

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

  class B extends A{

  public static void main(String args[]){

   B obj = new B();

   obj.msg();

  }

}

**Simple Example of public access modifier**

//save by A.java

package pack;

public class A{

public void msg(){System.out.println("Hello");}

}

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

//save by B.java

package mypack;

import pack.\*;

class B{

  public static void main(String args[]){

   A obj = new A();

   obj.msg();

  }

}