Arrays in Java

Unlike C++, arrays are first class objects in Java. For example, in the following program, size of array is accessed using *length* which is a member of *arr[]*object.

|  |
| --- |
| // file name: Main.java  public class Main {      public static void main(String args[]) {         int arr[] = {10, 20, 30, 40, 50};         for(int i=0; i < arr.length; i++)         {               System.out.print(" " + arr[i]);         }      }  } |

Output:  
*10 20 30 40 50*

How does default virtual behavior differ in C++ and Java ?

**Default virtual behavior of methods is opposite in C++ and Java:**

In C++, class member methods are non-virtual by default. They can be made virtual by using *virtual*keyword. For example, *Base::show()*is non-virtual in following program and program prints *“Base::show() called”*.

|  |
| --- |
| #include<iostream>    using namespace std;    class Base {  public:        // non-virtual by default      void show() {           cout<<"Base::show() called";      }  };    class Derived: public Base {  public:      void show() {           cout<<"Derived::show() called";      }  };    int main()  {    Derived d;    Base &b = d;    b.show();    getchar();    return 0;  } |

Adding *virtual*before definition of *Base::show()* makes program print *“Derived::show() called”*

In Java, methods are virtual by default and can be made non-virtual by using *final*keyword. For example, in the following java program, *show()* is by default virtual and the program prints *“Derived::show() called”*

|  |
| --- |
| class Base {        // virtual by default      public void show() {         System.out.println("Base::show() called");      }  }    class Derived extends Base {      public void show() {         System.out.println("Derived::show() called");      }  }    public class Main {      public static void main(String[] args) {          Base b = new Derived();;          b.show();      }  } |

Unlike C++ non-virtual behavior, if we add *final*before definition of show() in *Base*, then the above program fails in compilation.

>>>>>>>>>>>>

In Java, parameters are always passed by value. For example, following program prints i = 10, j = 20.

|  |
| --- |
| //  Test.java  class Test {     // swap() doesn't swap i and j     public static void swap(Integer i, Integer j) {        Integer temp = new Integer(i);        i = j;        j = temp;     }     public static void main(String[] args) {        Integer i = new Integer(10);        Integer j = new Integer(20);        swap(i, j);        System.out.println("i = " + i + ", j = " + j);     }  } |

How are Java objects stored in memory?

In Java, all objects are dynamically allocated on Heap. This is different from C++ where objects can be allocated memory either on Stack or on Heap. In C++, when we allocate abject using new(), the abject is allocated on Heap, otherwise on Stack if not global or static.

In Java, when we only declare a variable of a class type, only a reference is created (memory is not allocated for the object). To allocate memory to an object, we must use new(). So the object is always allocated memory on heap (See [this](https://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/garbage_collect.html) for more details).

For example, following program fails in compilation. Compiler gives error *“Error here because t is not initialed”.*

|  |
| --- |
| class Test {      // class contents      void show() {          System.out.println("Test::show() called");      }  }    public class Main {      public static void main(String[] args) {          Test t;          t.show(); // Error here because t is not initialed      }  } |

Allocating memory using new() makes above program work.

|  |
| --- |
| class Test {      // class contents      void show() {          System.out.println("Test::show() called");      }  }    public class Main {      public static void main(String[] args) {          Test t = new Test(); //all objects are dynamically allocated          t.show(); // No error      }  } |

Are static local variables allowed in Java?

Unlike C/C++, static local variables are not allowed in Java. For example, following Java program fails in compilation with error *“Static local variables are not allowed”*

|  |
| --- |
| class Test {     public static void main(String args[]) {       System.out.println(fun());     }       static int fun()     {       static int x= 10;  //Error: Static local variables are not allowed       return x--;     }  } |

How long do you want your 'static within a method' variable to last? If it's until the end of the method, then you can just use it without static. If it's for the lifetime of the class, then you can declare it as a static member variable. What other options are there?

C++ allows statics within a method, but they end up behaving just like a static class variable, but with reduced scope. Even in C++ they are rarely used. They also end up being stored exactly like static member variables.

The Java designers decided that the small amount of benefit wasn't worth the additional complexity of the language.

final variables in Java

In Java, when final keyword is used with a variable of primitive data types (int, float, .. etc), value of the variable cannot be changed.

For example following program gives error because i is final.

|  |
| --- |
| public class Test {      public static void main(String args[]) {          final int i = 10;          i = 30; // Error because i is final.      }  } |

When final is used with non-primitive variables (Note that non-primitive variables are always references to objects in Java), the members of the referred object can be changed. final for non-primitive variables just mean that they cannot be changed to refer to any other object

|  |
| --- |
| class Test1 {     int i = 10;  }    public class Test2 {      public static void main(String args[]) {        final Test1 t1 = new Test1();        t1.i = 30;  // Works      }  } |

Default constructor in Java

Like C++, Java [automatically creates default constructor if there is no default or parameterized constructor written by user](http://geeksforgeeks.org/?p=8316), and (like C++) the default constructor automatically calls parent default constructor. But unlike C++, default constructor in Java initializes member data variable to default values (numeric values are initialized as 0, booleans are initialized as *false*and references are initialized as *null*).

For example, output of the below program is

0  
null  
false  
0  
0.0

|  |
| --- |
| // Main.java  class Test {     int i;     Test t;     boolean b;     byte bt;     float ft;  }    public class Main {      public static void main(String args[]) {        Test t = new Test(); // default constructor is called.        System.out.println(t.i);        System.out.println(t.t);        System.out.println(t.b);        System.out.println(t.bt);        System.out.println(t.ft);      }  } |

Assigning values to static final variables in Java

**Assigning values to static final variables in Java:**  
In Java, non-static final variables can be assigned a value either in constructor or with the declaration. But, static final variables cannot be assigned value in constructor; they must be assigned a value with their declaration.

For example, following program works fine.

|  |
| --- |
| class Test {    final int i;  // i could be assigned a value here also    Tets() {      i = 10;    }      //other stuff in the class  } |

If we make *i* as *static final* then we must assign value to i with the delcaration.

|  |
| --- |
| class Test {    static final int i = 10;  // Since i is static final, it must be assigned value here only.      //other stuff in the class  } |

Such behavior is obvious as static variables are shared among all the objects of a class; creating a new object would change the same static variable which is not allowed if the static variable is final.

Shadowing of static functions in Java

In Java, if name of a derived class static function is same as base class static function then the derived class static function shadows (or conceals) the base class static function. For example, the following Java code prints*“A.fun()”*

|  |
| --- |
| // file name: Main.java  class A {     static void fun() {        System.out.println("A.fun()");     }  }    class B extends A {     static void fun() {        System.out.println("B.fun()");     }  }    public class Main {     public static void main(String args[]) {        A a = new B();        a.fun();  // prints A.fun()     }  } |

If we make both A.fun() and B.fun() as non-static then the above program would print “B.fun()”.

the call to a static method is decided based on the reference type. So since since object "a" is of type A, A.fun() is called in the above code.

Comparison of Exception Handling in C++ and Java

Both languages use *try*, *catch*and *throw*keywords for exception handling, and meaning of *try*, *catch*and *free*blocks is also same in both languages. Following are the differences between Java and C++ exception handling.

**1)** In C++, all types (including primitive and pointer) can be thrown as exception. But in Java only throwable objects (Throwable objects are instances of any subclass of the Throwable class) can be thrown as exception. For example, following type of code works in C++, but similar code doesn’t work in Java.

|  |
| --- |
| #include <iostream>  using namespace std;  int main()  {     int x = -1;       // some other stuff     try {        // some other stuff        if( x < 0 )        {           throw x;        }     }     catch (int x ) {        cout << "Exception occurred: thrown value is " << x << endl;     }     getchar();     return 0;  } |

Output:  
*Exception occurred: thrown value is -1*  
  
  
  
**2)**In C++, there is a special catch called “catch all” that can catch all kind of exceptions.

|  |
| --- |
| #include <iostream>  using namespace std;  int main()  {     int x = -1;     char \*ptr;       ptr = new char[256];       // some other stuff     try {        // some other stuff        if( x < 0 )        {           throw x;        }        if(ptr == NULL)        {           throw " ptr is NULL ";        }     }     catch (...) // catch all     {        cout << "Exception occurred: exiting "<< endl;        exit(0);     }       getchar();     return 0;  } |

Output:  
*Exception occurred: exiting*

In Java, for all practical purposes, we can catch Exception object to catch all kind of exceptions. Because, normally we do not catch Throwable(s) other than Exception(s) (which are Errors)

|  |
| --- |
| catch(Exception e){    …….  } |

**3)** In Java, there is a block called [*finally*](http://download.oracle.com/javase/tutorial/essential/exceptions/finally.html)that is always executed after the try-catch block. This block can be used to do cleanup work. There is no such block in C++.

|  |
| --- |
| // creating an exception type  class Test extends Exception { }    class Main {     public static void main(String args[]) {          try {           throw new Test();        }        catch(Test t) {           System.out.println("Got the Test Exception");        }        finally {           System.out.println("Inside finally block ");        }    }  } |

Output:  
Got the Test Exception *Inside finally block*

**4)** In C++, all exceptions are unchecked. In Java, there are two types of exceptions – checked and unchecked. See [this](http://tutorials.jenkov.com/java-exception-handling/checked-or-unchecked-exceptions.html)for more details on checked vs Unchecked exceptions.

**5)**In Java, a new keyword *throws*is used to list exceptions that can be thrown by a function. In C++, there is no*throws*keyword, the same keyword *throw*is used for this purpose also.

There is more to point 4 than mentioned here. There is a reason why C++ doesnt have finally , it is because it doesn't need it. Finally is required by languages in which objects dont have deterministic cleanup ie their lifetime is not determined by scope. In Java objects are garbage-collected so their life-time is indeterministic with respect to their scope.If an exception occurs and object goes out of scope , no gurantees can be made about its cleanup-time.If that object encapsulates a scarce resource , this means that no gurantee can be placed on cleanup of that scarce resource , which is bad. So in Java we have finally block you can free the underlying native resource as the object goes out-of-scope. Compiler gaurantees that finally block will always be called. In C++ on the other hand for auto variables we have lifetime determined by scope and so if we create the object on stack as local compiler gurantees its cleanup as soon as object goes out of scope. This technique is also called RAII.

Does Java support goto?

Unlike C/C++, Java does not have *goto*statement, but java supports label. The only place where a label is useful in Java is right before nested loop statements. We can specify label name with **break** to break out a specific outer loop. Similarly, label name can be specified with **continue**.

See following program for example.

|  |
| --- |
| // file name: Main.java  public class Main {    public static void main(String[] args) {      outer: //label for outer loop      for (int i = 0; i < 10; i++) {        for (int j = 0; j < 10; j++) {          if (j == 1)            break outer;          System.out.println(" value of j = " + j);        }      } //end of outer loop    } // end of main()  } //end of class Main |

Output:  
*value of j = 0*

How to compare two arrays in Java?

Predict the output of following Java program.

|  |
| --- |
| class Test  {      public static void main (String[] args)      {          int arr1[] = {1, 2, 3};          int arr2[] = {1, 2, 3};          if (arr1 == arr2) // Same as arr1.equals(arr2)              System.out.println("Same");          else              System.out.println("Not same");      }  } |

Output:

Not Same

In Java, [arrays are first class objects](http://www.geeksforgeeks.org/g-fact-65/). In the above program, arr1 and arr2 are two references to two different objects. So when we compare arr1 and arr2, two reference variables are compared, therefore we get the output as “Not Same” (See [this](http://www.geeksforgeeks.org/final-arrays-in-java/)for more examples).

**How to compare array contents?**  
A simple way is to run a loop and compare elements one by one. Java provides a direct method *Arrays.equals()*to compare two arrays. Actually, there is a list of equals() methods in Arrays class for different primitive types (int, char, ..etc) and one for Object type (which is base of all classes in Java).

|  |
| --- |
| // we need to import java.util.Arrays to use Arrays.equals().  import java.util.Arrays;  class Test  {      public static void main (String[] args)      {          int arr1[] = {1, 2, 3};          int arr2[] = {1, 2, 3};          if (Arrays.equals(arr1, arr2))              System.out.println("Same");          else              System.out.println("Not same");      }  } |

Output:

Same

**How to Deep compare array contents?**  
As seen above, the Arrays.equals() works fine and compares arrays contents. Now the questions, what if the arrays contain arrays inside them or some other references which refer to different object but have same values. For example, see the following program.

|  |
| --- |
| import java.util.Arrays;  class Test  {      public static void main (String[] args)      {          // inarr1 and inarr2 have same values          int inarr1[] = {1, 2, 3};          int inarr2[] = {1, 2, 3};          Object[] arr1 = {inarr1};  // arr1 contains only one element          Object[] arr2 = {inarr2};  // arr2 also contains only one element          if (Arrays.equals(arr1, arr2))              System.out.println("Same");          else              System.out.println("Not same");      }  } |

Output:

Not Same

So *Arrays.equals()* is not able to do deep comparison. Java provides another method for this Arrays.deepEquals() which does deep comparison.

|  |
| --- |
| import java.util.Arrays;  class Test  {      public static void main (String[] args)      {          int inarr1[] = {1, 2, 3};          int inarr2[] = {1, 2, 3};          Object[] arr1 = {inarr1};  // arr1 contains only one element          Object[] arr2 = {inarr2};  // arr2 also contains only one element          if (Arrays.deepEquals(arr1, arr2))              System.out.println("Same");          else              System.out.println("Not same");      }  } |

Output:

Same

**How does Arrays.deepEquals() work?**  
It compares two objects using any custom equals() methods they may have (if they have an equals() method implemented other than Object.equals()). If not, this method will then proceed to compare the objects field by field, recursively. As each field is encountered, it will attempt to use the derived equals() if it exists, otherwise it will continue to recurse further.  
This method works on a cyclic Object graph like this: A->B->C->A. It has cycle detection so ANY two objects can be compared, and it will never enter into an endless loop (Source: <https://code.google.com/p/deep-equals/>).

**Exercise:** Predict the output of following program

|  |
| --- |
| import java.util.Arrays;  class Test  {     public static void main (String[] args)     {        int inarr1[] = {1, 2, 3};        int inarr2[] = {1, 2, 3};        Object[] arr1 = {inarr1};  // arr1 contains only one element        Object[] arr2 = {inarr2};  // arr2 also contains only one element        Object[] outarr1 = {arr1}; // outarr1 contains only one element        Object[] outarr2 = {arr2}; // outarr2 also contains only one element        if (Arrays.deepEquals(outarr1, outarr2))            System.out.println("Same");        else            System.out.println("Not same");      }  } |

O/P: Same

Use DeepEquals.hashCode(obj) to compute a hashCode() for any object. Like deepEquals(), it will attempt to call the hashCode() method if a custom hashCode()method (below Object.hashCode()) is implemented, otherwise it will compute the hashCode field by field, recursively (Deep). Also like deepEquals(), this method will handle Object graphs with cycles. For example, A->B->C->A. In this case,hashCode(A) == hashCode(B) == hashCode(C).DeepEquals.deepHashCode() has cycle detection and therefore will work on ANY object graph.

**When would you use it?**

If you had two 'Person' objects (person1, person2) with firstName, lastName, with the same values for these fields, yet the actual Person instance was different, and this Person object did not have a custom equals() or hashCode() method on it, then calling person1.equals(person2) will return false, because it will default to Object.equals() which relies on the storage location of the object (identity hash). In most cases, you would rather these objects be considered equals. In these cases, calling DeepEquals.deepEquals(person1, person2) will return true. This allows you to properly handle (essentially fix) objects that do not have equals() and hashCode() methods implemented on them.

Can we override private methods in Java?

Let us first consider the following Java program as a simple example of Overriding or Runtime Polymorphism.

|  |
| --- |
| class Base {    public void fun() {       System.out.println("Base fun");    }  }    class Derived extends Base {    public void fun() {  // overrides the Base's fun()       System.out.println("Derived fun");    }    public static void main(String[] args) {        Base obj = new Derived();        obj.fun();    }  } |

The program prints “Derived fun”.  
The Base class reference ‘obj’ refers to a derived class object (see expression “Base obj = new Derived()”). When fun() is called on obj, the call is made according to the type of referred object, not according to the reference.

***Is Overiding possible with private methods?***  
Predict the output of following program.

|  |
| --- |
| class Base {    private void fun() {       System.out.println("Base fun");    }  }    class Derived extends Base {    private void fun() {       System.out.println("Derived fun");    }    public static void main(String[] args) {        Base obj = new Derived();        obj.fun();    }  } |

We get compiler error “fun() has private access in Base” (See [this](http://ideone.com/arKk3c)). So the compiler tries to call base class function, not derived class, means fun() is not overridden.

***An inner class can access private members of its outer class. What if we extend an inner class and create fun() in the inner class?***  
An Inner classes can access private members of its outer class, for example in the following program, *fun()* of*Inner*accesses private data member *msg* which is fine by the compiler.

|  |
| --- |
| /\* Java program to demonstrate whether we can override private method     of outer class inside its inner class \*/  class Outer {       private String msg = "GeeksforGeeks";       private void fun() {            System.out.println("Outer fun()");       }         class Inner extends Outer {           private void fun()  {                 System.out.println("Accessing Private Member of Outer: " + msg);           }       }         public static void main(String args[])  {              // In order to create instance of Inner class, we need an Outer            // class instance. So, first create Outer class instance and then            // inner class instance.            Outer o = new Outer();            Inner  i   = o.new Inner();              // This will call Inner's fun, the purpose of this call is to            // show that private members of Outer can be accessed in Inner.            i.fun();              // o.fun() calls Outer's fun (No run-time polymorphism).            o = i;            o.fun();       }  } |

Output:

Accessing Private Member of Outer: GeeksforGeeks

Outer fun()

In the above program, we created an outer class and an inner class. We extended Inner from Outer and created a method fun() in both Outer and Inner. If we observe our output, then it is clear that the method fun() has not been overriden. It is so because ***private methods are bonded during compile time and it is the type of the reference variable – not the type of object that it refers to – that determines what method to be called.***. As a side note, private methods may be performance-wise better (compared to non-private and non-final methods) due to static binding.

***Comparison With C++***  
**1)**In Java, inner Class is allowed to access private data members of outer class. This behavior is same as C++

**2)** In Java, methods declared as private can never be overridden, they are in-fact bounded during compile time. This behavior is different from C++. In C++, we can have virtual private methods.

Can we Overload or Override static methods in java ?

Let us first define Overloading and Overriding.

[**Overriding**](http://en.wikipedia.org/wiki/Method_overriding): Overriding is a feature of OOP languages like Java that is related to run-time polymorphism. A subclass (or derived class) provides a specific implementation of a method in superclass (or base class).  
The implementation to be executed is decided at run-time and decision is made according to the object used for call. Note that signatures of both methods must be same.

[**Overloading**](http://en.wikipedia.org/wiki/Function_overloading): Overloading is also a feature of OOP languages like Java that is related to compile time (or static) polymorphism. This feature allows different methods to have same name, but different signatures, especially number of input parameters and type of input paramaters. Note that in both C++ and Java, [methods cannot be overloaded according to return type.](http://www.geeksforgeeks.org/g-fact-75/)

**Can we overload static methods?**  
The answer is ‘Yes’. We can have two ore more static methods with same name, but differences in input parameters. For example, consider the following Java program.

|  |
| --- |
| // filename Test.java  public class Test {      public static void foo() {          System.out.println("Test.foo() called ");      }      public static void foo(int a) {          System.out.println("Test.foo(int) called ");      }      public static void main(String args[])      {          Test.foo();          Test.foo(10);      }  } |

Output:

Test.foo() called

Test.foo(int) called

**Can we overload methods that differ only by static keyword?**  
We cannot overload two methods in Java if they differ only by static keyword (number of parameters and types of parameters is same). See following Java program for example. This behaviour is same in C++ (See point 2 of[this](http://www.geeksforgeeks.org/function-overloading-in-c/)).

|  |
| --- |
| // filename Test.java  public class Test {      public static void foo() {          System.out.println("Test.foo() called ");      }      public void foo() { // Compiler Error: cannot redefine foo()          System.out.println("Test.foo(int) called ");      }      public static void main(String args[]) {          Test.foo();      }  } |

Output: Compiler Error, cannot redefine foo()

**Can we Override static methods in java?**  
We can declare static methods with same signature in subclass, but it is not considered overriding as there won’t be any run-time polymorphism. Hence the answer is ‘No’.  
If a derived class defines a static method with same signature as a static method in base class, the method in the derived class hides the method in the base class.

|  |
| --- |
| /\* Java program to show that if static method is redefined by     a derived class, then it is not overriding. \*/    // Superclass  class Base {        // Static method in base class which will be hidden in subclass      public static void display() {          System.out.println("Static or class method from Base");      }         // Non-static method which will be overridden in derived class       public void print()  {           System.out.println("Non-static or Instance method from Base");      }  }    // Subclass  class Derived extends Base {        // This method hides display() in Base      public static void display() {           System.out.println("Static or class method from Derived");      }        // This method overrides print() in Base      public void print() {           System.out.println("Non-static or Instance method from Derived");     }  }    // Driver class  public class Test {      public static void main(String args[ ])  {         Base obj1 = new Derived();           // As per overriding rules this should call to class Derive's static         // overridden method. Since static method can not be overridden, it         // calls Base's display()         obj1.display();           // Here overriding works and Derive's print() is called         obj1.print();      }  } |

Output:

Static or class method from Base

Non-static or Instance method from Derived

Following are some important points for method overriding and static methods in Java.  
**1)** For class (or static) methods, the method according to the type of reference is called, not according to the abject being referred, which means method call is decided at compile time.

**2)** For instance (or non-static) methods, the method is called according to the type of object being referred, not according to the type of reference, which means method calls is decided at run time.

**3)** An instance method cannot override a static method, and a static method cannot hide an instance method. For example, the following program has two compiler errors.

|  |
| --- |
| /\* Java program to show that if static methods are redefined by     a derived class, then it is not overriding but hidding. \*/    // Superclass  class Base {        // Static method in base class which will be hidden in subclass      public static void display() {          System.out.println("Static or class method from Base");      }         // Non-static method which will be overridden in derived class       public void print()  {           System.out.println("Non-static or Instance method from Base");      }  }    // Subclass  class Derived extends Base {        // Static is removed here (Causes Compiler Error)      public void display() {          System.out.println("Non-static method from Derived");      }        // Static is added here (Causes Compiler Error)      public static void print() {          System.out.println("Static method from Derived");     }  } |

**4)** In a subclass (or Derived Class), we can overload the methods inherited from the superclass. Such overloaded methods neither hide nor override the superclass methods — they are new methods, unique to the subclass.

**References:**  
<http://docs.oracle.com/javase/tutorial/java/IandI/override.html>

Static class in Java

**Can a class be static in Java ?**  
The answer is YES, we can have static class in java. In java, we have [static instance variables](http://www.geeksforgeeks.org/static-keyword-in-java/) as well as [static methods](http://www.geeksforgeeks.org/static-keyword-in-java/) and also [static block](http://www.geeksforgeeks.org/g-fact-79/). Classes can also be made static in Java.

Java allows us to define a class within another class. Such a class is called a nested class. The class which enclosed nested class is known as Outer class. In java, we can’t make Top level class static. ***Only nested classes can be static***.

**What are the differences between static and non-static nested classes?**  
Following are major differences between static nested class and non-static nested class. Non-static nested class is also called Inner Class.

**1)** Nested static class doesn’t need reference of Outer class, but Non-static nested class or Inner class requires Outer class reference.

**2)** Inner class(or non-static nested class) can access both static and non-static members of Outer class. A static class cannot access non-static members of the Outer class. It can access only static members of Outer class.

**3)**An instance of Inner class cannot be created without an instance of outer class and an Inner class can reference data and methods defined in Outer class in which it nests, so we don’t need to pass reference of an object to the constructor of the Inner class. For this reason Inner classes can make program simple and concise.

|  |
| --- |
| /\* Java program to demonstrate how to implement static and non-static     classes in a java program. \*/  class OuterClass{     private static String msg = "GeeksForGeeks";       // Static nested class     public static class NestedStaticClass{           // Only static members of Outer class is directly accessible in nested         // static class         public void printMessage() {             // Try making 'message' a non-static variable, there will be           // compiler error           System.out.println("Message from nested static class: " + msg);         }      }        // non-static nested class - also called Inner class      public class InnerClass{           // Both static and non-static members of Outer class are accessible in         // this Inner class         public void display(){            System.out.println("Message from non-static nested class: "+ msg);         }      }  }  class Main  {      // How to create instance of static and non static nested class?      public static void main(String args[]){           // create instance of nested Static class         OuterClass.NestedStaticClass printer = new OuterClass.NestedStaticClass();           // call non static method of nested static class         printer.printMessage();           // In order to create instance of Inner class we need an Outer class         // instance. Let us create Outer class instance for creating         // non-static nested class         OuterClass outer = new OuterClass();         OuterClass.InnerClass inner  = outer.new InnerClass();           // calling non-static method of Inner class         inner.display();           // we can also combine above steps in one step to create instance of         // Inner class         OuterClass.InnerClass innerObject = new OuterClass().new InnerClass();           // similarly we can now call Inner class method         innerObject.display();      }  } |

Output:

Message from nested static class: GeeksForGeeks

Message from non-static nested class: GeeksForGeeks

Message from non-static nested class: GeeksForGeeks

Reference Book:  
Introduction To Java Programming by Y. DANIEL LIANG

Do we need forward declarations in Java?

Predict output of the following Java program.

|  |
| --- |
| // filename: Test2.java    // main() function of this class uses Test1 which is declared later in  // this file  class Test2 {      public static void main(String[] args) {           Test1 t1 = new Test1();           t1.fun(5);      }  }  class Test1 {      void fun(int x) {          System.out.println("fun() called: x = " + x);      }  } |

Output:

fun() called: x = 5

The Java program compiles and runs fine. Note that *Test1*and *fun()* are not declared before their use. Unlike C++, we don’t need [forward declarations](http://en.wikipedia.org/wiki/Forward_declaration) in Java. Identifiers (class and method names) are recognized automatically from source files. Similarly, library methods are directly read from the libraries, and there is no need to create header files with declarations. Java uses naming scheme where package and public class names must follow directory and file names respectively. This naming scheme allows Java compiler to locate library files.

Checked vs Unchecked Exceptions in Java

In Java, there two types of exceptions:

**1) Checked:** are the exceptions that are checked at compile time. If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using*throws*keyword.

For example, consider the following Java program that opens file at locatiobn “C:\test\a.txt” and prints first three lines of it. The program doesn’t compile, because the function main() uses FileReader() and FileReader() throws a checked exception *FileNotFoundException*. It also uses readLine() and close() methods, and these methods also throw checked exception *IOException*

|  |
| --- |
| import java.io.\*;    class Main {      public static void main(String[] args) {          FileReader file = new FileReader("C:\\test\\a.txt");          BufferedReader fileInput = new BufferedReader(file);            // Print first 3 lines of file "C:\test\a.txt"          for (int counter = 0; counter < 3; counter++)              System.out.println(fileInput.readLine());            fileInput.close();      }  } |

Output:

Exception in thread "main" java.lang.RuntimeException: Uncompilable source code -

unreported exception java.io.FileNotFoundException; must be caught or declared to be

thrown

at Main.main(Main.java:5)

To fix the above program, we either need to specify list of exceptions using throws, or we need to use try-catch block. We have used throws in the below program. Since *FileNotFoundException* is a subclass of *IOException*, we can just specify *IOException* in the throws list and make the above program compiler-error-free.

|  |
| --- |
| import java.io.\*;    class Main {      public static void main(String[] args) throws IOException {          FileReader file = new FileReader("C:\\test\\a.txt");          BufferedReader fileInput = new BufferedReader(file);            // Print first 3 lines of file "C:\test\a.txt"          for (int counter = 0; counter < 3; counter++)              System.out.println(fileInput.readLine());            fileInput.close();      }  } |

Output: First three lines of file “C:\test\a.txt”

**2) Unchecked** are the exceptions that are not checked at compiled time. In C++, all exceptions are unchecked, so it is not forced by the compiler to either handle or specify the exception. It is up to the programmers to be civilized, and specify or catch the exceptions.  
In Java exceptions under *Error*and *RuntimeException*classes are unchecked exceptions, everything else under throwable is checked.

+-----------+

| Throwable |

+-----------+

/ \

/ \

+-------+ +-----------+

| Error | | Exception |

+-------+ +-----------+

/ | \ / | \

\\_\_\_\_\_\_\_\_/ \\_\_\_\_\_\_/ \

unchecked checked +------------------+

| RuntimeException |

+------------------+

/ | | \

\\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_/

unchecked

Consider the following Java program. It compiles fine, but it throws *ArithmeticException* when run. The compiler allows it to compile, because *ArithmeticException* is an unchecked exception.

|  |
| --- |
| class Main {     public static void main(String args[]) {        int x = 0;        int y = 10;        int z = y/x;    }  } |

Output:

Exception in thread "main" java.lang.ArithmeticException: / by zero

at Main.main(Main.java:5)

Java Result: 1

**Why two types?**  
See [Unchecked Exceptions — The Controversy](http://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html) for details.

**Should we make our exceptions checked or unchecked?**  
Following is the bottom line from [Java documents](http://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html)  
*If a client can reasonably be expected to recover from an exception, make it a checked exception. If a client cannot do anything to recover from the exception, make it an unchecked exception*

Bitwise right shift operators in Java

In C/C++ there is only one right shift operator ‘>>’ which should be used only for positive integers or unsigned integers. Use of right shift operator for negative numbers is not recommended in C/C++, and when used for negative numbers, output is compiler dependent (See [this](https://www.securecoding.cert.org/confluence/display/seccode/INT13-C.+Use+bitwise+operators+only+on+unsigned+operands)). Unlike C++, Java supports following two right shift operators.

**1) >> (Signed right shift)** In Java, the operator ‘>>’ is signed right shift operator. All integers are signed in Java, and it is fine to use >> for negative numbers. The operator ‘>>’ uses the sign bit (left most bit) to fill the trailing positions after shift. If the number is negative, then 1 is used as a filler and if the number is positive, then 0 is used as a filler. For example, if binary representation of number is **1**0….100, then right shifting it by 2 using >> will make it **11**…….1.  
See following Java programs as example ‘>>’

|  |
| --- |
| class Test {      public static void main(String args[])  {         int x = -4;         System.out.println(x>>1);         int y = 4;         System.out.println(y>>1);      }  } |

Output:

-2

2

**2) >>> (Unsigned right shift)** In Java, the operator ‘>>>’ is unsigned right shift operator. It always fills 0 irrespective of the sign of the number.

|  |
| --- |
| class Test {      public static void main(String args[])  {           // x is stored using 32 bit 2's complement form.         // Binary representation of -1 is all 1s (111..1)         int x = -1;           System.out.println(x>>>29);  // The value of 'x>>>29' is 00...0111         System.out.println(x>>>30);  // The value of 'x>>>30' is 00...0011         System.out.println(x>>>31);  // The value of 'x>>>31' is 00...0001     }  } |

Output:

7

3

1

# Private and final methods in Java

When we use final specifier with a method, the method cannot be overridden in any of the inheriting classes. Methods are made final due to design reasons.   
Since private methods are inaccessible, they are implicitly final in Java. So adding finalspecifier to a private method doesn’t add any value. It may in-fact cause unnecessary confusion.

Final arrays in Java

Predict the output of following Java program.

|  |
| --- |
| class Test  {      public static void main(String args[])      {         final int arr[] = {1, 2, 3, 4, 5};  // Note: arr is final         for (int i = 0; i < arr.length; i++)         {             arr[i] = arr[i]\*10;             System.out.println(arr[i]);         }      }  } |

Output:

10

20

30

40

50

The array *arr* is declared as final, but the elements of array are changed without any problem. [Arrays are objects](http://www.geeksforgeeks.org/archives/9562)and [object variables are always references](http://www.geeksforgeeks.org/archives/8926) in Java. So, when we declare an object variable as final, it means that the variable cannot be changed to refer to anything else. For example, the following program 1 compiles without any error and program fails in compilation.

|  |
| --- |
| // Program 1  class Test  {      int p = 20;      public static void main(String args[])      {         final Test t = new Test();         t.p = 30;         System.out.println(t.p);      }  } |

Output: 30

|  |
| --- |
| // Program 2  class Test  {      int p = 20;      public static void main(String args[])      {         final Test t1 = new Test();         Test t2 = new Test();         t1 = t2;         System.out.println(t1.p);      }  } |

Output: Compiler Error: cannot assign a value to final variable t1

*So a final array means that the array variable which is actually a reference to an object, cannot be changed to refer to anything else, but the members of array can be modified.*

|  |
| --- |
| class Test  {      public static void main(String args[])      {         final int arr1[] = {1, 2, 3, 4, 5};         int arr2[] = {10, 20, 30, 40, 50};         arr2 = arr1;         arr1 = arr2;         for (int i = 0; i < arr2.length; i++)            System.out.println(arr2[i]);      }  } |

Compiler Error

Abstract Classes in Java

In C++, if a class has at least one pure virtual function, then the class becomes abstract. Unlike C++, in Java, a separate keyword *abstract* is used to make a class abstract.

|  |
| --- |
| // An example abstract class in Java  abstract class Shape {      int color;        // An abstract function (like a pure virtual function in C++)      abstract void draw();  } |

Following are some important observations about abstract classes in Java.

**1)** Like C++, in Java, an instance of an abstract class cannot be created, we can have references of abstract class type though.

|  |
| --- |
| abstract class Base {      abstract void fun();  }  class Derived extends Base {      void fun() { System.out.println("Derived fun() called"); }  }  class Main {      public static void main(String args[]) {            // Uncommenting the following line will cause compiler error as the          // line tries to create an instance of abstract class.          // Base b = new Base();            // We can have references of Base type.          Base b = new Derived();          b.fun();      }  } |

Output:

Derived fun() called

**2)** Like C++, an abstract class can contain constructors in Java. And a constructor of abstract class is called when an instance of a inherited class is created. For example, the following is a valid Java program.

|  |
| --- |
| // An abstract class with constructor  abstract class Base {      Base() { System.out.println("Base Constructor Called"); }      abstract void fun();  }  class Derived extends Base {      Derived() { System.out.println("Derived Constructor Called"); }      void fun() { System.out.println("Derived fun() called"); }  }  class Main {      public static void main(String args[]) {         Derived d = new Derived();      }  } |

Output:

Base Constructor Called

Derived Constructor Called

**3)**In Java, we can have an abstract class without any abstract method. This allows us to create classes that cannot be instantiated, but can only be inherited.

|  |
| --- |
| // An abstract class without any abstract method  abstract class Base {      void fun() { System.out.println("Base fun() called"); }  }    class Derived extends Base { }    class Main {      public static void main(String args[]) {          Derived d = new Derived();          d.fun();      }  } |

Output:

Base fun() called

**4)** Abstract classes can also have final methods (methods that cannot be overridden). For example, the following program compiles and runs fine.

|  |
| --- |
| // An abstract class with a final method  abstract class Base {      final void fun() { System.out.println("Derived fun() called"); }  }    class Derived extends Base {}    class Main {      public static void main(String args[]) {         Base b = new Derived();         b.fun();      }  } |

Output:

Derived fun() called

**Exercise:**  
**1.** Is it possible to create abstract and final class in Java? NO  
**2.** Is it possible to have an abstract method in a final class? NO  
**3.** Is it possible to inherit from multiple abstract classes in Java? NO

Comparison of static keyword in C++ and Java

Static keyword is used for almost same purpose in both C++ and Java. There are some differences though. This post covers similarities and differences of static keyword in C++ and Java.

**Static Data Members:** Like C++, static data members in Java are class members and shared among all objects. For example, in the following Java program, static variable *count*is used to count the number of objects created.

|  |
| --- |
| class Test {      static int count = 0;        Test() {         count++;      }      public static void main(String arr[]) {         Test t1 = new Test();         Test t2 = new Test();         System.out.println("Total " + count + " objects created");      }  } |

Output:

Total 2 objects created

**Static Member Methods:** Like C++, methods declared as static are class members and have following restrictions:

**1)** They can only call other static methods. For example, the following program **fails in compilation**. fun() is non-static and it is called in static main()

|  |
| --- |
| class Main {      public static void main(String args[]) {          System.out.println(fun());      }      int fun() {          return 20;      }  } |

**2)** They must only access static data.

**3)** They cannot access [*this*](http://docs.oracle.com/javase/tutorial/java/javaOO/thiskey.html)or [*super*](http://docs.oracle.com/javase/tutorial/java/IandI/super.html). For example, the following program fails in compilation.

|  |
| --- |
| class Base {      static int x = 0;  }    class Derived extends Base  {     public static void fun() {         System.out.println(super.x); // Compiler Error: non-static variable                                    // cannot be referenced from a static context     }  } |

Like C++, static data members and static methods can be accessed without creating an object. They can be accessed using class name. For example, in the following program, static data member count and static method fun() are accessed without any object.

|  |
| --- |
| class Test {      static int count = 0;      public static void fun() {         System.out.println("Static fun() called");      }  }    class Main  {      public static void main(String arr[]) {         System.out.println("Test.count = " + Test.count);         Test.fun();      }  } |

**Static Block:**Unlike C++, Java supports a special block, called static block (also called static clause) which can be used for static initialization of a class. This code inside static block is executed only once. See [Static blocks in Java](http://www.geeksforgeeks.org/archives/10270) for details.

**Static Local Variables:** Unlike C++, Java doesn’t support static **local** variables. For example, the following Java program fails in compilation.

|  |
| --- |
| class Test {     public static void main(String args[]) {       System.out.println(fun());     }     static int fun()  {       static int x= 10; //Compiler Error: Static local variables are not allowed       return x--;     }  } |

Overriding equals method in Java

Consider the following Java program:

|  |
| --- |
| class Complex {      private double re, im;        public Complex(double re, double im) {          this.re = re;          this.im = im;      }  }    // Driver class to test the Complex class  public class Main {      public static void main(String[] args) {          Complex c1 = new Complex(10, 15);          Complex c2 = new Complex(10, 15);          if (c1 == c2) {              System.out.println("Equal ");          } else {              System.out.println("Not Equal ");          }      }  } |

Output:

Not Equal

The reason for printing “Not Equal” is simple: when we compare c1 and c2, it is checked whether both c1 and c2 refer to same object or not ([object variables are always references in Java](http://www.geeksforgeeks.org/archives/8926)). c1 and c2 refer to two different objects, hence the value (c1 == c2) is false. If we create another reference say c3 like following, then (c1 == c3) will give true.

|  |
| --- |
| Complex c3 = c1;  // (c3 == c1) will be true |

So, how do we check for equality of values inside the objects? All classes in Java inherit from the Object class, directly or indirectly (See point 1 of [this](http://www.geeksforgeeks.org/archives/15055)). **The**[**Object class**](http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/Object.html)**has some basic methods like clone(), toString(), equals(),.. etc. We can override the equals method in our class to check whether two objects have same data or not.**

|  |
| --- |
| class Complex {        private double re, im;        public Complex(double re, double im) {          this.re = re;          this.im = im;      }        // Overriding equals() to compare two Complex objects      @Override      public boolean equals(Object o) {            // If the object is compared with itself then return true          if (o == this) {              return true;          }            /\* Check if o is an instance of Complex or not            "null instanceof [type]" also returns false \*/          if (!(o instanceof Complex)) {              return false;          }            // typecast o to Complex so that we can compare data members          Complex c = (Complex) o;            // Compare the data members and return accordingly          return Double.compare(re, c.re) == 0                  && Double.compare(im, c.im) == 0;      }  }    // Driver class to test the Complex class  public class Main {        public static void main(String[] args) {          Complex c1 = new Complex(10, 15);          Complex c2 = new Complex(10, 15);          if (c1.equals(c2)) {              System.out.println("Equal ");          } else {              System.out.println("Not Equal ");          }      }  } |

Output:

Equal

As a side note, when we override equals(), it is recommended to also override the hashCode() method. If we don’t do so, equal objects may get different hash-values; and hash based collections, including HashMap, HashSet, and Hashtable do not work properly (see [this](http://www.technofundo.com/tech/java/equalhash.html)for more details). We will be coverig more about hashCode() in a separate post.

References:  
[Effective Java Second Edition](http://java.sun.com/docs/books/effective/)

Overriding toString() in Java

This post is similar to [Overriding equals method in Java](http://www.geeksforgeeks.org/archives/15616). Consider the following Java program:

|  |
| --- |
| // file name: Main.java    class Complex {      private double re, im;        public Complex(double re, double im) {          this.re = re;          this.im = im;      }  }    // Driver class to test the Complex class  public class Main {      public static void main(String[] args) {          Complex c1 = new Complex(10, 15);          System.out.println(c1);      }  } |

Output:

Complex@19821f

The output is, *class name, then ‘at’ sign, and at the end [hashCode](http://en.wikipedia.org/wiki/Java_hashCode()) of object*. All classes in Java inherit from the Object class, directly or indirectly (See point 1 of [this](http://www.geeksforgeeks.org/archives/15055)). The Object class has some basic methods like clone(), toString(), equals(),.. etc. The default toString() method in Object prints “class name @ hash code”. We can override toString() method in our class to print proper output. For example, in the following code toString() is overridden to print “Real + i Imag” form.

|  |
| --- |
| // file name: Main.java    class Complex {      private double re, im;        public Complex(double re, double im) {          this.re = re;          this.im = im;      }        /\* Returns the string representation of this Complex number.         The format of string is "Re + iIm" where Re is real part         and Im is imagenary part.\*/      @Override      public String toString() {          return String.format(re + " + i" + im);      }  }    // Driver class to test the Complex class  public class Main {      public static void main(String[] args) {          Complex c1 = new Complex(10, 15);          System.out.println(c1);      }  } |

Output:

10.0 + i15.0

In general, it is a good idea to override toString() as we get get proper output when an object is used in print() or println().

**References:**  
[Effective Java by Joshua Bloch](http://www.flipkart.com/effective-java-8131726592/p/itmczzffdyegaf6h?pid=9788131726594&affid=sandeepgfg)

Copy Constructor in Java

Like C++, Java also supports copy constructor. But, unlike C++, Java doesn’t create a default copy constructor if you don’t write your own.

Following is an example Java program that shows a simple use of copy constructor.

|  |
| --- |
| // filename: Main.java    class Complex {        private double re, im;        // A normal parametrized constructor      public Complex(double re, double im) {          this.re = re;          this.im = im;      }        // copy constructor      Complex(Complex c) {          System.out.println("Copy constructor called");          re = c.re;          im = c.im;      }        // Overriding the toString of Object class      @Override      public String toString() {          return "(" + re + " + " + im + "i)";      }  }    public class Main {        public static void main(String[] args) {          Complex c1 = new Complex(10, 15);            // Following involves a copy constructor call          Complex c2 = new Complex(c1);            // Note that following doesn't involve a copy constructor call as          // non-primitive variables are just references.          Complex c3 = c2;            System.out.println(c2); // toString() of c2 is called here      }  } |

Output:

Copy constructor called

(10.0 + 15.0i)

Now try the following Java program:

|  |
| --- |
| // filename: Main.java    class Complex {        private double re, im;        public Complex(double re, double im) {          this.re = re;          this.im = im;      }  }    public class Main {        public static void main(String[] args) {          Complex c1 = new Complex(10, 15);          Complex c2 = new Complex(c1);  // **compiler error here**      }  } |

Accessing Grandparent’s member in Java

**Directly accessing Grandparent’s member in Java:**

Predict the output of following Java program.

|  |
| --- |
| // filename Main.java  class Grandparent {      public void Print() {          System.out.println("Grandparent's Print()");      }  }    class Parent extends Grandparent {      public void Print() {          System.out.println("Parent's Print()");      }  }    class Child extends Parent {      public void Print() {          super.super.Print();  // Trying to access Grandparent's Print()          System.out.println("Child's Print()");      }  }    public class Main {      public static void main(String[] args) {          Child c = new Child();          c.Print();      }  } |

Output: Compiler Error  
There is error in line “super.super.print();”. In Java, a class cannot directly access the grandparent’s members. It is allowed in C++ though. In C++, we can use scope resolution operator (::) to access any ancestor’s member in inheritance hierarchy. ***In Java, we can access grandparent’s members only through the parent class.***For example, the following program compiles and runs fine.

|  |
| --- |
| // filename Main.java  class Grandparent {      public void Print() {          System.out.println("Grandparent's Print()");      }  }    class Parent extends Grandparent {      public void Print() {          super.Print();          System.out.println("Parent's Print()");      }  }    class Child extends Parent {      public void Print() {          super.Print();          System.out.println("Child's Print()");      }  }    public class Main {      public static void main(String[] args) {          Child c = new Child();          c.Print();      }  } |

Output:

Grandparent's Print()

Parent's Print()

Child's Print()

Comparison of Inheritance in C++ and Java

The purpose of inheritance is same in C++ and Java. Inheritance is used in both languages for reusing code and/or creating is-a relationship. There are following differences in the way both languages provide support for inheritance.

**1)** In Java, all classes inherit from the [Object class](http://download.oracle.com/javase/1.5.0/docs/api/java/lang/Object.html)directly or indirectly. Therefore, there is always a single inheritance tree of classes in Java, and [Object class](http://download.oracle.com/javase/1.5.0/docs/api/java/lang/Object.html)is root of the tree. In Java, if we create a class that doesn’t inherit from any class then it automatically inherits from [Object class](http://download.oracle.com/javase/1.5.0/docs/api/java/lang/Object.html). In C++, there is forest of classes; when we create a class that doesn’t inherit from anything, we create a new tree in forest.

Following Java example shows that Test class automatically inherits from Object class.

|  |
| --- |
| class Test {      // members of test  }  class Main {    public static void main(String[] args) {      Test t = new Test();      System.out.println("t is instanceof Object: " + (t instanceof Object));    }  } |

Output:

t is instanceof Object: true

**2)** In Java, members of the grandparent class are not directly accessible. See [this G-Fact](http://www.geeksforgeeks.org/archives/15258) for more details.

**3)** The meaning of protected member access specifier is somewhat different in Java. In Java, protected members of a class “A” are accessible in other class “B” of same package, even if B doesn’t inherit from A (they both have to be in the same package). For example, in the following program, protected members of A are accessible in B.

|  |
| --- |
| // filename B.java  class A {      protected int x = 10, y = 20;  }    class B {      public static void main(String args[]) {          A a = new A();          System.out.println(a.x + " " + a.y);      }  } |

**4)**Java uses *extends*keyword for inheritence. Unlike C++, Java doesn’t provide an inheritance specifier like public, protected or private. Therefore, we cannot change the protection level of members of base class in Java, if some data member is public or protected in base class then it remains public or protected in derived class. Like C++, private members of base class are not accessible in derived class.  
Unlike C++, in Java, we don’t have to remember those rules of inheritance which are combination of base class access specifier and inheritance specifier.

**5)** In Java, methods are virtual by default. In C++, we explicitly use virtual keyword. See [this G-Fact](http://www.geeksforgeeks.org/archives/8876) for more details.

**6)** Java uses a separte keyword *interface*for interfaces, and *abstract*keyword for abstract classes and abstract functions.

Following is a Java abstract class example.

|  |
| --- |
| // An abstract class example  abstract class myAbstractClass {       // An abstract method     abstract void myAbstractFun();       // A normal method     void fun() {        System.out.println("Inside My fun");     }  }    public class myClass extends myAbstractClass {     public void myAbstractFun() {        System.out.println("Inside My fun");     }  } |

Following is a Java interface example

|  |
| --- |
| // An interface example  public interface myInterface {     // myAbstractFun() is public and abstract, even if we don't use these keywords     void myAbstractFun();  // is same as public abstract void myAbstractFun()  }    // Note the implements keyword also.  public class myClass implements myInterface {     public void myAbstractFun() {        System.out.println("Inside My fun");     }  } |

**7)** Unlike C++, Java doesn’t support multiple inheritance. A class cannot inherit from more than one class. A class can implement multiple interfaces though.

**8 )**In C++, default constructor of parent class is automatically called, but if we want to call parametrized constructor of a parent class, we must use [Initializer list](http://www.geeksforgeeks.org/archives/13797). Like C++, default constructor of the parent class is automatically called in Java, but if we want to call parametrized constructor then we must use super to call the parent constructor. See following Java example.

|  |
| --- |
| package main;    class Base {      private int b;      Base(int x) {          b = x;          System.out.println("Base constructor called");      }  }    class Derived extends Base {      private int d;      Derived(int x, int y) {          // Calling parent class parameterized constructor          // Call to parent constructor must be the first line in a Derived class          super(x);          d = y;          System.out.println("Derived constructor called");      }  }    class Main{      public static void main(String[] args) {        Derived obj = new Derived(1, 2);      }  } |

Output:

Base constructor called

Derived constructor called

Access specifiers for classes or interfaces in Java

In Java, methods and data members of a class/interface can have one of the following four access specifiers. The access specifiers are listed according to their restrictiveness order.

1) private  
2) default (when no access specifier is specified)  
3) protected  
4) public

But, the classes and interfaces themselves can have only two access specifiers when declared outside any other class.  
1) public  
2) default (when no access specifier is specified)

We cannot declare class/interface with private or protected access specifiers. For example, following program fails in compilation.

|  |
| --- |
| //filename: Main.java  protected class Test {}    public class Main {    public static void main(String args[]) {      }  } |

Note : Nested interfaces and classes can have all access specifiers.

Static blocks in Java

Unlike C++, Java supports a special block, called static block (also called static clause) which can be used for static initializations of a class. This code inside static block is executed only once: the first time you make an object of that class or the first time you access a static member of that class (even if you never make an object of that class). For example, check output of following Java program.

|  |
| --- |
| // filename: Main.java  class Test {      static int i;      int j;        // start of static block      static {          i = 10;          System.out.println("static block called ");      }      // end of static block  }    class Main {      public static void main(String args[]) {            // Although we don't have an object of Test, static block is          // called because i is being accessed in following statement.          System.out.println(Test.i);      }  } |

Output:  
*static block called  
10*

Also, static blocks are executed before constructors. For example, check output of following Java program.

|  |
| --- |
| // filename: Main.java  class Test {      static int i;      int j;      static {          i = 10;          System.out.println("static block called ");      }      Test(){          System.out.println("Constructor called");      }  }    class Main {      public static void main(String args[]) {           // Although we have two objects, static block is executed only once.         Test t1 = new Test();         Test t2 = new Test();      }  } |

Output:  
*static block called  
Constructor called  
Constructor called*

**What if we want to execute some code for every object?**  
We use [Initializer Block in Java](http://www.geeksforgeeks.org/g-fact-26-the-initializer-block-in-java/)

**References:**  
[Thinking in Java Book](http://geeksquiz.com/amazon-books/)

>>>>>>>> In C++ and Java, functions can not be overloaded if they differ only in the return type.

Access specifier of methods in interfaces

In Java, all methods in an interface are *public* even if we do not specify *public*with method names. Also, data fields are *public static final* even if we do not mention it with fields names. Therefore, data fields must be initialized.

Consider the following example, *x*is by default *public static final* and *foo()* is *public*even if there are no specifiers.

|  |
| --- |
| interface Test {    int x = 10;  // x is public static final and must be initialized here    void foo();  // foo() is public  } |

What happens when more restrictive access is given to a derived class method in Java?

In Java, it is compiler error to give more restrictive access to a derived class function which overrides a base class function. For example, if there is a function *public void foo()* in base class and if it is overridden in derived class, then access specifier for *foo()*cannot be anything other than public in derived class. If *foo()* is private function in base class, then access specifier for it can be anything in derived class.

Consider the following two programs. Program 1 fails in compilation and program 2 works fine.

**Program 1**

|  |
| --- |
| // file name: Main.java  class Base {      public void foo() {}  }    class Derived extends Base {      private void foo() {} // compiler error  }    public class Main {      public static void main(String args[]) {          Derived d = new Derived();      }  } |

**Program 2**

|  |
| --- |
| // file name: Main.java  class Base {      private void foo() {}  }    class Derived extends Base {      public void foo() {} // works fine  }    public class Main {      public static void main(String args[]) {          Derived d = new Derived();      }  } |

Inheritance and constructors in Java

In Java, constructor of base class with no argument gets automatically called in derived class constructor. For example, output of following program is:

*Base Class Constructor Called  
Derived Class Constructor Called*

|  |
| --- |
| // filename: Main.java  class Base {    Base() {      System.out.println("Base Class Constructor Called ");    }  }    class Derived extends Base {    Derived() {      System.out.println("Derived Class Constructor Called ");    }  }    public class Main {    public static void main(String[] args) {      Derived d = new Derived();    }  } |

But, if we want to call parameterized contructor of base class, then we can call it using super(). The point to note is **base class comstructor call must be the first line in derived class constructor**. For example, in the following program, super(\_x) is first line derived class constructor.

|  |
| --- |
| // filename: Main.java  class Base {    int x;    Base(int \_x) {      x = \_x;    }  }    class Derived extends Base {    int y;    Derived(int \_x, int \_y) {      super(\_x);      y = \_y;    }    void Display() {      System.out.println("x = "+x+", y = "+y);    }  }    public class Main {    public static void main(String[] args) {      Derived d = new Derived(10, 20);      d.Display();    }  } |

Output:  
*x = 10, y = 20*