1) Can abstract class have constructors in Java?

Ans: Yes ,all the classes including the abstract classes can have

constructors. Abstract class constructors will be called when its concrete

subclass will be instantiated

2) Can abstract class implements interface in Java? do they require to

implement all methods?

Ans: Yes ,an abstract class can implement an interface.Not required to

implement all methods but the subclasses will be required to

implement any interface methods that the abstract class left out.

3) Can abstract class be final in Java?

Ans: NO. Because

For final classes we can't create child class whereas for abstract classes we should create child class to provide implementation. Hence final, abstract combination is illegal for classes.

and also Abstract methods compulsory we should override in child classes to provide implementation. Whereas we can't override final methods. Here final, abstract combination is illegal combination for methods.

4) Can abstract class have static methods in Java?

Ans: Yes,Abstract Class can have static methods

abstract class Foo{

static void bar(){}

}

This is allowed because that method can be called directly, even if you do not have an instance of the abstract class:

Foo.bar();

 But Abstract class cannot create an abstract static method because we can't create an instance of an abstract class. But since a static method can be called directly, making it abstract would make it possible to call an undefined method.

5) Can you create instance of abstract class?

Ans: No, you cannot create an instance of an abstract class because it does not have a complete implementation. The purpose of an abstract class is to function as a base for subclasses. It acts like a template, or an empty or partially empty structure, you should extend it and build on it before you can use it.

6) Is it necessary for abstract class to have abstract method?

Ans: No ,not necessary

7) Difference between abstract class and interface in Java?

Ans:

|  |  |
| --- | --- |
| Abstract Class | Interface |
| An abstract class can have instance methods that would have default implementation | Interface is implicitly abstract and cannot have any default method implementation. |
| Abstract class **doesn’t support multiple inheritance**. | Interface **supports multiple inheritance**. |
| Abstract class can have constructors. | Interfaces cannot declare a constructor. |
| Abstract class can extend only one class or one abstract class at a time | Interface can extend any number of interfaces at a time |
| In abstract class keyword ‘abstract’ is mandatory to declare a method as an abstract | In an interface keyword ‘abstract’ is optional to declare a method as an abstract |
| Abstract class can have protected , public and public abstract methods. There is no implicit behavior for the members. | Interface can have only public abstract methods. The members of interface are implicitly public by default. Also interfaces cannot have static methods. |
| Abstract class can have  static, final  or static final  variable with any access specifiers. | Interface members are implicitly static and final by default. |

8) When do you favor abstract class over interface?

Ans: We can favor abstract classes in following situation:

1. You want to share code among several closely related classes.
2. You expect that classes that extend your abstract class have many common methods or fields or require access modifiers other than public (such as protected and private).
3. You want to declare non-static or non-final fields. This enables you to define methods that can access and modify the state of the object to which they belong.

Whereas interface can be used :

1. You expect that unrelated classes would implement your interface. For example, the interfaces Comparable and Cloneable are implemented by many unrelated classes.
2. You want to specify the behavior of a particular data type, but not concerned about who implements its behavior.
3. You want to take advantage of multiple inheritances.

9) What is abstract method in Java?

Ans:  A method without body (no implementation) is known as abstract

method. A method must always be declared in an abstract class

10) Can abstract class contains main method in Java ?

Ans: Yes

Abstract just means you can't instantiate the class directly. You can have constructors if you want - they might be needed for subclasses to initiate the object state. You can have static methods, including main() and they don't need an object so calling them is fine.

11) what is static block in java?

Ans: Static block is used for initializing the static variables.This block gets executed when the class is loaded in the memory. A class can have multiple Static blocks, which will execute in the same sequence in which they have been written into the program.

12) What is the need of static block?

Ans: The static block is neded when you do have to do some action even if

no instances is still created. As example, for initializing a static

Variable with non static value.

Other advantages of static blocks are

* If you’re loading drivers and other items into the namespace. For ex, Class class has a static block where it registers the natives.
* If you need to do computation in order to initialize your static variables,you can declare a static block which gets executed exactly once,when the class is first loaded.
* Security related issues or logging related tasks

13) Can we overload static methods in java?

Ans:The answer is **Yes**. We can have two or more static methods with same name, but differences in input parameters. We **cannot** overload two methods in Java if they differ only by static keyword (number of parameters and types of parameters is same).

14) Can we call super class static methods from sub class?

* Yes we can call super class static method inside sub class using super\_class\_method();
* We can also call super class static method using Sub\_class\_name.superclass\_staticMethod()

15) What is the difference between final and static keywords?

1. The static keyword  is applicable to a nested static class, variables, methods and blocks. On the other hand, final keyword is applicable to class methods and variables.
2. Static variable can be initialized any time whereas, a final variable must be initialized at the time of declaration.
3. A static variable can be reinitialized whereas, once initialized a final variable can never be reinitialized.
4. A static method can access the static member of the class and can only be invoked by other static methods. On the other hand, the final method can never be inherited by any class.
5. Static block is used to initialize the static variables whereas, final keyword does not support any block.

16) Write a note on covariant return type with example code.

Java 5.0 onwards it is possible to have different return type for a overriding method in child class, but child’s return type should be **sub-type** of parent’s return type. Overriding method becomes **variant** with respect to return type.

Co-variant return type is based on [Liskov substitution principle](https://en.wikipedia.org/wiki/Liskov_substitution_principle). i.e

Let T be a super type and S be its subtype (parent and child class). Then, instances (objects) of T can be substituted with instances of S. Parent’s instances can be replaced with the child’s instances without change in behavior of the program.

Below is the simple example to understand the co-variant return type with method overriding.

|  |
| --- |
| // Two classes used for return types.  class A {}  class B extends A {}    class Base  {      A fun()      {          System.out.println("Base fun()");          return new A();      }  }    class Derived extends Base  {      B fun()      {          System.out.println("Derived fun()");          return new B();      }  }    public class Main  {      public static void main(String args[])      {         Base base = new Base();         base.fun();           Derived derived = new Derived();         derived.fun();      }  } |

17) Write a note on Enum with example code.

Enumerations serve the purpose of representing a group of named constants in a programming language. For example the 4 suits in a deck of playing cards may be 4 enumerators named Club, Diamond, Heart, and Spade, belonging to an enumerated type named Suit. Other examples include natural enumerated types (like the planets, days of the week, colors, directions, etc.).  
Enums are used when we know all possible values at **compile time**, such as choices on a menu, rounding modes, command line flags, etc. It is not necessary that the set of constants in an enum type stay **fixed** for all time.

In Java (from 1.5), enums are represented using **enum** data type. Java enums are more powerful than [C/C++ enums](http://quiz.geeksforgeeks.org/enumeration-enum-c/). In Java, we can also add variables, methods and constructors to it. The main objective of enum is to define our own data types(Enumerated Data Types).

**Declaration of enum in java :**

* Enum declaration can be done outside a Class or inside a Class but not inside a Method.

|  |
| --- |
| // A simple enum example where enum is declared  // outside any class (Note enum keyword instead of  // class keyword)  enum Color  {      RED, GREEN, BLUE;  }    public class Test  {      // Driver method      public static void main(String[] args)      {          Color c1 = Color.RED;          System.out.println(c1);      }  } |

* Run on IDE
* Output :
* RED

|  |
| --- |
| // enum declaration inside a class.  public class Test  {      enum Color      {          RED, GREEN, BLUE;      }        // Driver method      public static void main(String[] args)      {          Color c1 = Color.RED;          System.out.println(c1);      }  } |

* Run on IDE
* Output :
* RED
* First line inside enum should be list of constants and then other things like methods, variables and constructor.
* According to [Java naming conventions](http://www.oracle.com/technetwork/java/codeconventions-135099.html), it is recommended that we name constant with all capital letters

**Important points of enum :**

* Every enum internally implemented by using Class.
* /\* internally above enum Color is converted to
* class Color
* {
* public static final Color RED = new Color();
* public static final Color BLUE = new Color();
* public static final Color GREEN = new Color();
* }\*/
* Every enum constant represents an **object** of type enum.
* enum type can be passed as an argument to **switch** statement.

|  |
| --- |
| // A Java program to demonstrate working on enum  // in switch case (Filename Test. Java)  import java.util.Scanner;    // An Enum class  enum Day  {      SUNDAY, MONDAY, TUESDAY, WEDNESDAY,      THURSDAY, FRIDAY, SATURDAY;  }    // Driver class that contains an object of "day" and  // main().  public class Test  {      Day day;        // Constructor      public Test(Day day)      {          this.day = day;      }        // Prints a line about Day using switch      public void dayIsLike()      {          switch (day)          {          case MONDAY:              System.out.println("Mondays are bad.");              break;          case FRIDAY:              System.out.println("Fridays are better.");              break;          case SATURDAY:          case SUNDAY:              System.out.println("Weekends are best.");              break;          default:              System.out.println("Midweek days are so-so.");              break;          }      }        // Driver method      public static void main(String[] args)      {          String str = "MONDAY";          Test t1 = new Test(Day.valueOf(str));          t1.dayIsLike();      }  } |

* Run on IDE
* Output:
* Mondays are bad.
* Every enum constant is always implicitly **public static final**. Since it is **static**, we can access it by using enum Name. Since it is **final**, we can’t create child enums.
* We can declare **main() method** inside enum. Hence we can invoke enum directly from the Command Prompt.

|  |
| --- |
| // A Java program to demonstrate that we can have  // main() inside enum class.  enum Color  {      RED, GREEN, BLUE;        // Driver method      public static void main(String[] args)      {          Color c1 = Color.RED;          System.out.println(c1);      }  } |

* Run on IDE
* Output :
* RED

**Enum and Inheritance :**

* All enums implicitly extend **java.lang.Enum class**. As a class can only extend **one**parent in Java, so an enum cannot extend anything else.
* **toString() method** is overridden in **java.lang.Enum class**,which returns enum constant name.
* enum can implement many interfaces.

**values(), ordinal() and valueOf() methods :**

* These methods are present inside **java.lang.Enum**.
* **values() method** can be used to return all values present inside enum.
* Order is important in enums.By using **ordinal() method**, each enum constant index can be found, just like array index.
* **valueOf() method** returns the enum constant of the specified string value, if exists.

|  |
| --- |
| // Java program to demonstrate working of values(),  // ordinal() and valueOf()  enum Color  {      RED, GREEN, BLUE;  }    public class Test  {      public static void main(String[] args)      {          // Calling values()          Color arr[] = Color.values();            // enum with loop          for (Color col : arr)          {              // Calling ordinal() to find index              // of color.              System.out.println(col + " at index "                               + col.ordinal());          }            // Using valueOf(). Returns an object of          // Color with given constant.          // Uncommenting second line causes exception          // IllegalArgumentException          System.out.println(Color.valueOf("RED"));          // System.out.println(Color.valueOf("WHITE"));      }  } |

* Run on IDE
* Output :
* RED at index 0
* GREEN at index 1
* BLUE at index 2
* RED

**enum and constructor :**

* enum can contain constructor and it is executed separately for each enum constant at the time of enum class loading.
* We can’t create enum objects explicitly and hence we can’t invoke enum constructor directly.

**enum and methods :**

* enum can contain **concrete** methods only i.e. no any **abstract** method.

|  |
| --- |
| // Java program to demonstrate that enums can have constructor  // and concrete methods.    // An enum (Note enum keyword inplace of class keyword)  enum Color  {      RED, GREEN, BLUE;        // enum constructor called separately for each      // constant      private Color()      {          System.out.println("Constructor called for : " +          this.toString());      }        // Only concrete (not abstract) methods allowed      public void colorInfo()      {          System.out.println("Universal Color");      }  }    public class Test  {      // Driver method      public static void main(String[] args)      {          Color c1 = Color.RED;          System.out.println(c1);          c1.colorInfo();      }  } |

Run on IDE

Output:

Constructor called for : RED

Constructor called for : GREEN

Constructor called for : BLUE

RED

Universal Color

Enum with Customized Value in Java

By default enums have their own string values, we can also assign some custom values to enums. Consider below example for that.

Examples:

enum Fruits

{

APPLE(“RED”), BANANA(“YELLOW”), GRAPES(“GREEN”);

}

In above example we can see that the Fruits enum have three members i.e APPLE, BANANA and ORANGE with have their own different custom values RED, YELLOW and GREEN respectively.

**Now to use this enum in code, there are some points we have to follow:-**

1. We have to create parameterized constructor for this enum class. Why? Because as we know that enum class’s object can’t be create explicitly so for initializing we use parameterized constructor. And the constructor cannot be the public or protected it must have private or default modifiers. Why? if we create public or protected, it will allow initializing more than one objects. This is totally against enum concept.
2. We have to create one getter method to get the value of enums.

|  |
| --- |
| // Java program to demonstrate how values can  // be assigned to enums.  enum TrafficSignal  {      // This will call enum constructor with one      // String argument      RED("STOP"), GREEN("GO"), ORANGE("SLOW DOWN");        // declaring private variable for getting values      private String action;        // getter method      public String getAction()      {          return this.action;      }        // enum constructor - cannot be public or protected      private TrafficSignal(String action)      {          this.action = action;      }  }    // Driver code  public class EnumConstructorExample  {      public static void main(String args[])      {          // let's print name of each enum and there action          // - Enum values() examples          TrafficSignal[] signals = TrafficSignal.values();            for (TrafficSignal signal : signals)          {              // use getter method to get the value              System.out.println("name : " + signal.name() +                          " action: " + signal.getAction() );          }      }  } |

Run on IDE

Output:

name : RED action: STOP

name : GREEN action: GO

name : ORANGE action: SLOW DOWN

18) Write a note on use of super keyword and super() method.

The**super** keyword in java is a reference variable that is used to refer parent class objects. :

**Use of super with variables:**This scenario occurs when a derived class and base class has same data members. In that case there is a possibility of ambiguity for the JVM. We can understand it more clearly using this code snippet:

|  |
| --- |
| /\* Base class vehicle \*/  class Vehicle  {      int maxSpeed = 120;  }    /\* sub class Car extending vehicle \*/  class Car extends Vehicle  {      int maxSpeed = 180;        void display()      {          /\* print maxSpeed of base class (vehicle) \*/          System.out.println("Maximum Speed: " + super.maxSpeed);      }  }    /\* Driver program to test \*/  class Test  {      public static void main(String[] args)      {          Car small = new Car();          small.display();      }  } |

Run on IDE

Output:

Maximum Speed: 120

In the above example, both base class and subclass have a member maxSpeed. We could access maxSpeed of base class in sublcass using super keyword.

Super()method

**Use of super with methods:**This is used when we want to call parent class method. So whenever a parent and child class have same named methods then to resolve ambiguity we use super keyword. This code snippet helps to understand the said usage of super keyword.

|  |
| --- |
| /\* Base class Person \*/  class Person  {      void message()      {          System.out.println("This is person class");      }  }    /\* Subclass Student \*/  class Student extends Person  {      void message()      {          System.out.println("This is student class");      }        // Note that display() is only in Student class      void display()      {          // will invoke or call current class message() method          message();            // will invoke or call parent class message() method          super.message();      }  }    /\* Driver program to test \*/  class Test  {      public static void main(String args[])      {          Student s = new Student();            // calling display() of Student          s.display();      }  } |

Run on IDE

Output:

This is student class

This is person class

In the above example, we have seen that if we only call method message() then, the current class message() is invoked but with the use of super keyword, message() of superclass could also be invoked.

**Use of super with constructors:**super keyword can also be used to access the parent class constructor. One more important thing is that, ‘’super’ can call both parametric as well as non parametric constructors depending upon the situation. Following is the code snippet to explain the above concept:

|  |
| --- |
| /\* superclass Person \*/  class Person  {      Person()      {          System.out.println("Person class Constructor");      }  }    /\* subclass Student extending the Person class \*/  class Student extends Person  {      Student()      {          // invoke or call parent class constructor          super();            System.out.println("Student class Constructor");      }  }    /\* Driver program to test\*/  class Test  {      public static void main(String[] args)      {          Student s = new Student();      }  } |

Run on IDE

Output:

Person class Constructor

Student class Constructor

In the above example we have called the superclass constructor using keyword ‘super’ via subclass constructor.

**Other Important points:**

1. Call to super() must be first statement in Derived(Student) Class constructor.
2. If a constructor does not explicitly invoke a superclass constructor, the Java compiler automatically inserts a call to the no-argument constructor of the superclass. If the superclass does not have a no-argument constructor, you will get a compile-time error. Object *does* have such a constructor, so if Object is the only superclass, there is no problem.
3. If a subclass constructor invokes a constructor of its superclass, either explicitly or implicitly, you might think that a whole chain of constructors called, all the way back to the constructor of Object. This, in fact, is the case. It is called *constructor chaining*..

19) Write a code to implement abstraction using interface.

interface shape

 {

     public   String baseclass="shape";

     public void Draw();

 }

 class circle implements shape

 {

    public void Draw() {

        System.out.println("Drawing Circle here");

    }

 }

class Rectangle implements shape

 {

    public void Draw() {

        System.out.println("Drawing Rectangle here");

    }

 }

public class Main {

    public static void main(String[] args) {

          shape circleshape=new circle();

             circleshape.Draw();

shape rectangleshape=new Reactangle();

rectangleshape.Draw();

    }

}

20)Write a Java program to sort a numeric array and a string array.

//Sorting numeric array

**import** java.util.Scanner;

**public** **class** SortingDemo

{

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

{

**int** n, temp;

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

System.***out***.print("Enter no. of elements you want in array:");

n = s.nextInt();

**int** a[] = **new** **int**[n];

System.***out***.println("Enter all the elements:");

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

{

a[i] = s.nextInt();

}

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

{

**for** (**int** j = i + 1; j < n; j++)

{

**if** (a[i] > a[j])

{

temp = a[i];

a[i] = a[j];

a[j] = temp;

}

}

}

System.***out***.print("Ascending Order:");

**for** (**int** i = 0; i < n - 1; i++)

{

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

}

System.***out***.print(a[n - 1]);

}

}

//Sorting String array

**import** java.util.Arrays;

**import** java.util.Scanner;

**public** **class** SortString {

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

{

// create a Java String array

**int** n;

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

System.***out***.print("Enter no. of elements you want in array:");

n = s.nextInt();

String strarr[] = **new** String[n];

System.***out***.println("Enter all the elements:");

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

{

strarr[i] = s.next();

}

// sort the array, using the sort method of the Arrays class

Arrays.*sort*(strarr);

// print the sorted results

**for** (String str : strarr)

{

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

}

}

}

21)Write a Java program to sum values of an array.

**import** java.util.Scanner;

**public** **class** Array\_Sum

{

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

{

**int** n, sum = 0;

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

System.***out***.print("Enter no. of elements you want in array:");

n = s.nextInt();

**int** a[] = **new** **int**[n];

System.***out***.println("Enter all the elements:");

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

{

a[i] = s.nextInt();

sum = sum + a[i];

}

System.***out***.println("Sum:"+sum);

}

}

22)Write a Java program to remove a specific element from an array.

**import** java.util.Arrays;

**import** java.util.Scanner;

**public** **class** Removearray {

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

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

System.***out***.print("Enter no. of elements you want in array:");

**int** n = s.nextInt();

**int**[] my\_array = **new** **int**[n];

System.***out***.println("Enter all the elements:");

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

{

my\_array[i] = s.nextInt();

}

System.***out***.println("Original Array : "+Arrays.*toString*(my\_array));

System.***out***.print("Enter the index of element to be removed:");

**int** removeIndex = s.nextInt();;

**for**(**int** i = removeIndex; i < my\_array.length -1; i++){

my\_array[i] = my\_array[i + 1];

}

System.***out***.println("After removing the element: "+Arrays.*toString*(my\_array));

}

}

23)Write a Java program to reverse an array of integer values.

**import** java.util.Arrays;

**import** java.util.Collections;

**import** java.util.List;

**public** **class** ReverseArray {

/\*\*

\* **@param** args

\*/

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

Integer[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

List<Integer> list = Arrays.*asList*(numbers);

System.***out***.println("Array before reverse:" + list);

Collections.*reverse*(list);

list.toArray(numbers);

System.***out***.print("Array after reverse: ");

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

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

}

}

}

24)Write a Java program to find the duplicate values of an array of integer values.

**import** java.util.HashSet;

**import** java.util.Set;

**public** **class** Duplicate{

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

**int**[] array = {1,1,2,3,4,5,6,7,8,8};

Set<Integer> set = **new** HashSet<Integer>();

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

{

//If same integer is already present then add method will return FALSE

**if**(set.add(array[i]) == **false**)

{

System.***out***.println("Duplicate element found : " + array[i]);

}

}

}

}