Interface	Abstract Class
1. If we don't know anything about implementation just we have requirement specification then we should go for interface.	1. If we are talking about implementation but not completely (partial implementation) then we should go for Abstract class.
2. Inside Interface every method is always public and abstract whether we are declaring or not. Hence interface is also considered as 100% pure Abstract class.	2. Every method present in Abstract class need not be public and Abstract. In addition to abstract methods we can take concrete methods also.
3. We can't declare interface method with the following modifiers. Public ---> private, protected, Abstract ---> final, static, synchronized, native, strictfp	3. There are no restrictions on Abstract class method modifiers.
4. Every variable present inside interface is always public, static and final whether we are declaring or not.	4. The variables present inside Abstract class need not be public static and final.
5. We can't declare interface variables with the following modifiers. private, protected, transient, volatile.	5. There are no restrictions on Abstract class variable modifiers.
6. For Interface variables compulsory we should perform initialization at the time of declaration otherwise we will get compile time error.	6. For Abstract class variables it is not required to perform initialization at the time of declaration.
7. Inside interface we can't declare instance and static blocks. Otherwise we will get compile time error.	7. Inside Abstract class we can declare instance and static blocks.
8. Inside interface we can't declare constructors.	8. Inside Abstract class we can declare constructor, which will be executed at the time of child object creation.

final :

1. final is a modifier applicable for classes, methods and variables.If a class declared as final then we can't extend that class.
   i.e we can't create child class for that class.

2. If a method declared as final then we can't override that method in the child class.

3. If a variable declared as final then it will become constant and we can't perform re-assignment for that variable.

finally :

* finally is a block always associated with try catch to maintain cleanup code.

```
try
{
    // risky code…
}
catch( X e)
{
    // Handling code
}
finally
{
    //  cleanup code
}
```

finalize():

* finalize() is a method which is always invoked by garbage collector just before destroying an object to perform cleanup activities.

Difference between String and StringBuffer

1. once we creates a string object we can't perform any changes in the existing object. if we are trying to perform any changes with those changes a new object will be created. this non changeable nature is nothing but immutability of the string object.

2. Once we creates a StringBuffer object we can perform any type of changes in the existing object. this changeble is nothing but mutability of the StringBuffer object.

Difference between == operater and .equals() method?

* In general we use == operator for reference comparison, whereas .equals() method for content comparision.

Note :

* .equals() method present in object class also meant for reference comparison only based on our requirement we can override for content comparison .

* In String class, all wrapper class and all collection classes .equals() method is overridden for content comparison.


modifier
class
Outer
class
methods
variables
Local
Outer
Inner
enum
Outer
Inner
constructor
public
private
protected
default
final
abstract
static
synchronized
native
strictfp
transient
volatile
DURGASOFT

Important Conclusions :

1. The modifiers which are applicable for inner classes but not for outer classes.
   A) private, protected, static.
2. The modifiers which are applicable for classes but not for interface are final.
3. The modifiers which are applicable for classes but not for enums are final and abstract.
4. The modifiers which are applicable only for methods and which we can't use anyware else native.
5. The only modifiers which are applicable for contractors are.
   A) public ,private, protected, default
6. The only applicable modifier for local variable is final.

1. The interface which is declared inside a class is always static where we are declaring or not.

2. The interface which is declared inside interface is always public and static whether we are declaring or not.

3. The class which is declared inside interface is always public and static whether we are declaring or not.

When to use String , StringBuffer and StringBuilder?

1. If the content is fixed and won't change frequently then we should go for String.

2. If the content is not fixed and keep on changing but Thread Safety is required then we should go for StringBuffer.

3. If the content is not fixed and keep on changing and thread safety is not required then we should go for StringBuilder.

Access Specifiers vs Access Modifiers

1. In old languages like C++ public, private, protected, default are considered as Access Specifiers. Except this the remaining (like static) are considered as Access Modifiers.
2. But in Java there is no terminology like specifiers. all are by default considered as modifiers only

public	synchronized
private	abstract
protected	native
default	Stictfp(1.2v)
final	transient
static	volatile

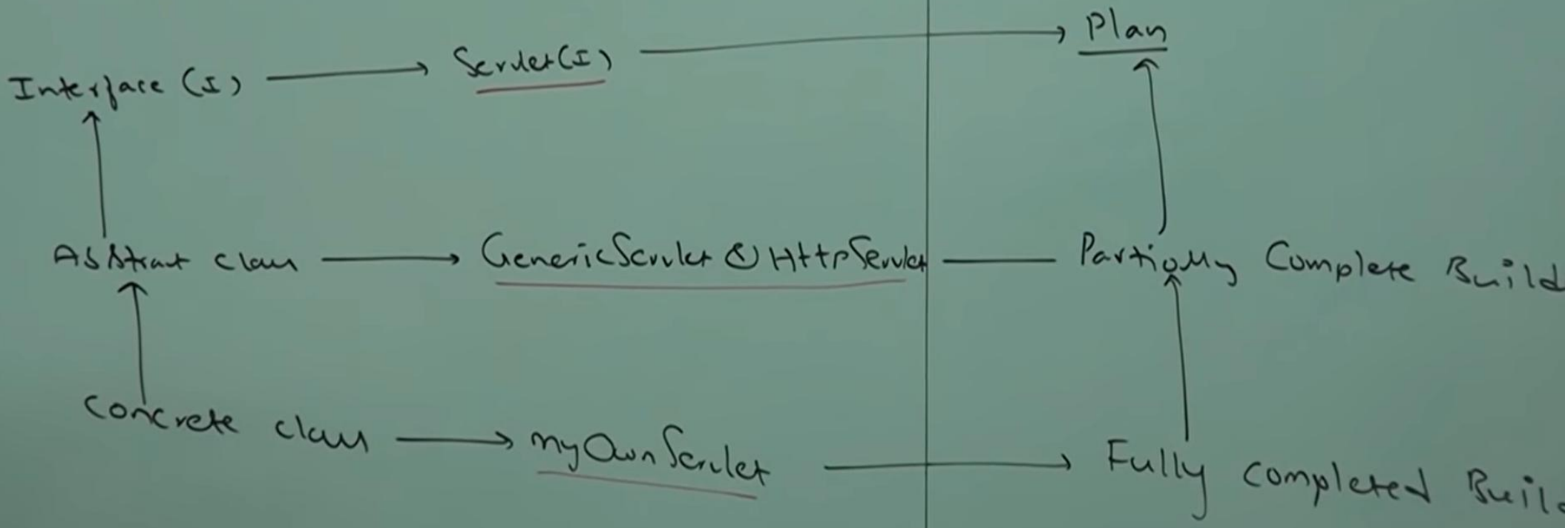
Interface VS Abstract class VS Concrete class?

1. If we don't know anything about implementation just we have requirement specification (100% Abstraction) then we should go for interface.
   Example : Servlet

2. If we are talking about implementation but not completely (partial implementation) then we should go for Abstract class.
   Example : GenericServlet & HTTPServlet.

3. If we are talking about implementation completely and ready to provide service then we should go for concrete class.
   Example : MyOwnServlet.

face vs Abstract class vs Concrete class



Differences Between StringBuffer and StringBuilder?

StringBuffer	StringBuilder
1. Every method present in StringBuffer is synchronized.	1.No method present in StringBuilder is synchronized .
2. At a time only one thread is allow to operate on StringBuffer object. Hence StringBuffer object is Thread safe	2. At a time multiple threads are allow to operate on StringBuilder object and hence StringBuilder object is not Thread Safe.
3. It Increases waiting time of threads and hence relatively performance is low.	3. Threads are not required to wait to operate on StringBuilder object and hence relatively performance is high.
4. Introduced in 1.0 version	4. Introduced in 1.5 version

```
class Test {
     static String s = ”java”;
}

Test.s.length();
```

* Test is a class name
* ‘s’ is a static variable present in Test class of type String.
* length() is method present in String class.

```
class System {
    static PrintStream out ;
}
```

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

* System is class present in java.lang package
* 'out' is a static variable present in system class of type PrintStream
* Println() is a method present in PrintStream class.

Case 1

* Overloading of the main method is possible but JVM will always call String[] argument main method only.

* The other overloaded method we have to call explicitly then it will be executed just a normal method call.

overloaded methods

```
class Test
{
 public static void main(String[] args)
 {
  System.out.println("String[]");
 }overloaded methods
 public static void main(int[] args)
 {
  System.out.println("int[]");
 }
}
```

output: String[]

Case 2

1. Inheritance concept applicable for the main method. Hence while executing child class if child class doesn't contain main method then parent class main method will be executed.

P.java

```
class P
{
  public static void main(String[] args)
  {
    System.out.println("parent main");
  }
}
class  C extends P
{
}
```

Case 3

It seems overriding concept applicable for main method but it is not overriding it is method hiding.

It is method hiding but not overriding

```
class  p
{
public static void main(String[] args)
{
System.out.println("parent main");
}
}
class c extends p
{
public static void main(String[] args)
{
System.out.println("child main");
}
}
```

Case 1:

Until 1.6 version if the class doesn't contain main() method then we will get runtime exception saying no such method error. But form 1.7version onwards instead of NoSuchMethodError we will get more meaningful error information.

```
class Test
{
}
```

1.6 version	1.7 version
javac Test.java	javac Test.java
java Test	java Test
RE: NoSuchMethodError: main	Error: Main method not found in class test, please define main method as public static void main(String[] args)

Case 2:

From 1.7 version onwards to run a java program main method is mandatory. Hence, even though class contains static blocks they wont be executed if the class doesn't contain main() method.

Example 1

```
class Test
{
static
{
System.out.println("static block");
}
}
```

1.6 version	1.7 version
javac Test.java	javac Test.java
java Test	java Test
output: static block RE: NoSuchMethodError: main	Error: main method not found in class

Example 2

```
class Test
{
static
{
System.out.println(“static block”);
System.exit(0);
}
}
```

Example 3

If the class contains main() method whether it is 1.6 or 1.7 version there is no change in execution sequence.

```
class Test
{
static
{
System.out.println(“static block”);
}
public static void main(String[] args)
{
System.out.println(“main method”);
}
}
```

1.6 version	1.7 version
javac Test.java	javac Test.java
java Test	java Test
output: static block method	output: static block main method

With out writing main() method is it possible to print some statements to the console?

Yes we can print by using static block.
But this rule is applicable until 1.6version only from 1.7version onwards main() method is mandatory to print some statements to the console.

Property	Overloading	Overriding
1. Method names	Must be same	Must be same
2. Argument Types	Must be different (at least order)	Must be same (Including order)
3. Method signatures	Must be different	Must be same.
4. Return Type	No Restrictions	Must be same but this rule is applicable until 1.4version only. From 1.5v onwards co-variant return types are allowed.
5.private,static & final methods	Can be overloaded	Cannot be overridden
6. Access modifiers	No Restrictions	We can't reduce scope of Access modifier but we can increase.
7. throws clause	No Restrictions	If child class method throws any checked exception compulsory parent class method should throw the same checked exception are its parent otherwise we will get compile time error but there are no restrictions for Unchecked Exceptions.
8. Method Resolution	Always takes care by compiler based on reference type.	Always takes care by JVM based on Runtime object.
9.Also Known as	Compile time polymorphism or static polymorphism or early binding.	Runtime polymorphism, dynamic polymorphism or late binding.

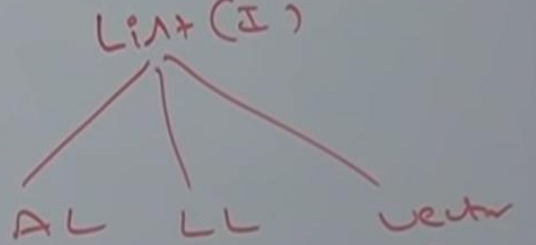
* Whether class contains main() method or not and whether main() method is declared according to requirement or not thess things won't be checked by compiler. At runtime, JVM is responsible to check thess things.

* At runtime if JVM is unable to find required main() method then we will get runtime exception saying NoSuchMethodError:main

```
class Test
{
}
```

```
javac Test.java
java Test
RuntimeException : NoSuchMethodError:main
```

p p = new C();



List m1()

{

List l = m1();

ret AL | LL | Vector | Stack

}

C c= new C();	P p=new C()
1. If we know exact runtime type of object then we should use this approach	1. if we don't know exact runtime type of object then we should use this approach. (polymorphism)
2. By using child reference we can call both parent and child class methods.	2. By using parent reference we can call only methods available in parent class and child specific methods we cant call.
3. we can use child reference to hold only for that particular child class object only.	3. we can use parent reference to hold any child class object.

1. Whenever we are writing try block compulsory we should write catch or finally. That is 'try' without catch or finally is invalid syntax.

2. Whenever we are writing catch block compulsory we should write try block that is catch without try is invalid.

3. Whenever we are writing finally block compulsory we should write try block. That is finally without try is invalid.

4. In try catch finally, order is important.

5. ‘try’ with multiple catch blocks Is valid but the order is important compulsory we should take from child to parent. by mistake if we are trying to take from parent to child then we will get compile time error.

6. if we are defining two catch blocks for the same exception we will get compile time error.

7. we can define try-catch-finally with in the try, with in the catch and with in finally blocks. Hence nesting of try-catch-finally is valid.

8. For try-catch-finally curly braces are mandatory.

1	2	3	4

try
{
}
catch(x e)
{
}
	try
	{
	}
	catch(x e)
	{
	}
	catch(x e)
	{
	}
	
	CE: exception x has already been caught
		try
		{
		}
		catch(x e)
		{
		}
		catch(y e)
		{
		}
			try
			{
			}
			catch(Exception e)
			{
			}
			catch(AE e)
			{
			}
			
			CE: exception java.lang.AE has already been caught

5.	6.	7.	8.

try
{
}
catch(AE e)
{
}
catch(Exception e)
{
}
	try
	{
	}
	CE: try without catch or finally
		catch(x e)
		{
		}
		CE: catch without try
			Finally
			{
			}
			CE: finally without try.

9.	10.	11.	12.

try
{
}
finally
{
}
	try
	{
	}
	system.out.
	println(“hello”);
	catch(x e)
	{
	}
	CE1: try without catch or finally
	CE2: catch without try
		try
		{
		}
		catch(x e)
		{
		}
		system.out.
		println(“hello”);
		catch(y e)
		{
		}
		CE: catch without try
			try
			{
			}
			catch(x e)
			{
			}
			system.out.
			println(“hello”);
			finally
			{
			}
			CE: finally without try

13.

```
try
{
}
catch(x e)
{
}
try
{
}
finally
{
}
```

14.

```
try
{
}
finally
{
}
catch(x e)
{
}
```

CE: catch without try

15.

```
try
{
}
catch(x e)
{
}
finally
{
}
finally
{
}
```

CE: finally without try

16.

```
try
{
try
{
}
catch(x e)
{
}
finally
{
}
}
catch(x e)
```

17.

```
try
{
}
catch(x e)
{
try
{
}
finally
{
}
}
```

18.

```
try
{
}
catch (x e)
{
}
finally
{
try
{
}
finally
{
}
}
```

19.

```
try
{
try
{
}
}
catch(x e)
{
}
```

CE: try without catch or finally

20.

```
try
system.out.
println("hello");
catch(x e)
{
}
```

21.

```
try
{
}
catch(x e)
system.out.
println("catch");
```

22.

```
try
{
}
catch( x e)
{
}
finally
system.out.
println("finally");
```

1.Most of the cases Exceptions are caused by our program and this are recoverable .

Example:

* For Example If our program requirement is to read data from a remote file locating at London at runtime if the London file is not available then we will get FileNotFoundException.

* If FileNotFoundException occurs then we can provide a local file and rest of the program will be continued normally.

Expection

```
try {
    // Read data from a remote file location at London
} catch( FileNotFoundException e) {
    // Use local file & continue rest of the program normally
}
```

Error:

1. Most of the times errors are not caused by our program these are due to lack of system resources.
2. errors are non recoverable

Example:

* For Example if OutOfMemeory error occurs being a programmer we can't do anything and the program will be terminated abnormally .

* System admin or server admin is responsible to increase heap memory.

Unchecked Exceptions :

* The Exceptions which are not checked by compiler are called unchecked exceptions.

Example :

ArithmeticException
BombBlastException
NullPointerException

* In the case of Unchecked Exceptions compiler wont check whether programmerhandling exception or not.

Checked Exceptions :

* The Exceptions which are checked by compiler for smooth execution of the program at runtime are called Checked Exceptions.

Example:

HallticketMissingException

PenNotWorkingException

FileNotFoundException

* In the case of Checked exceptions compiler will check whether we are handling exception if the programmer not handling then we will get compile time error.

Note-1:

* Whether exception is Checked or Unchecked compulsory it will occur only at Runtime.There is no chance of occurring any exception at compile time.

Note-2:

* Runtime exception and its child classes, error and its child classes are unchecked except this remaining are checked exceptions.

* A Checked Exception is said to be fully checked exception if and only if all its child classes also checked.

Example:

IOException

InteruptedException

* A Checked Exception is said to be partially Checked exception if and only if some of its child classes are Unchecked.

Example:

Exception

Note:

The only possible partially checked exceptions in java are

1. Exception
2. Throwable

Conclusions:

* With in the try block if anywhere exception raised then rest of try block won't be excuted even though we handled that exception. Hence length of try block should be as less as possible. And we have to take only resiky code with in try block.

Conclusions:

2 .In addition to try block there may be a chance of raising an exception inside catch and finally blocks.

Conclusions:

3. If any statement raises an exception, which is not part of try block then it is always Abnormal terimination.

Var-arg methods

Introduction and syntax:

* Until 1.4 version we cannot declare a method with variable number of arguments. If there is a change in a number of arguments then compulsory we should declare a new method,which increases length of the code and reduces readability. To overcome this problem SUN people introduced var-arg methods in 1.5 version.

* Hence from 1.5 versions onwards we can declare a method with the variable number of arguments,such type of methods are called Var-arg methods.

Var-arg methods

* We can declare a var-arg method as follows

```
m1(int... a)
```

* We can call this method by passing any number of int values including zero number also.

i . m1 ();
ii. m1(10);
iii. m1(10,20,30);

Example

```
Class Test {
    public static void m1 (int… x) {
        System.out.println (“Var-arg method”);
    }
    public static void main(String[] args) {
        m1 ();
        m2 (10, 20);
        m3 (10, 20 , 30 ,40);
        m4 (10, 20 , 30 ,40 ,50);
    }
}
```

Output:

Var-arg method
Var-arg method
Var-arg method
Var-arg method

Var-arg methods

* Internally var-arg parameters will be converted into one dimensional array.

* Hence within the var-arg method we can differentiate values by using index.

Example

```
Class Test {
    public static void main (String[] args) {
            sum ();                  //Sum is: 0
            sum (10);                //Sum is: 10
            sum (10,20);             //Sum is: 30
            sum (10,20,30,40);       //Sum is: 100
     } //main
     public static void sum (int… x) {
        int total=0;
        for (int x1:x) {
           Total=total + x;
         }  //for
        System.out.println (“Sum is:”+total);
      }  //sum
}  //class
```

Various syntax loop holes for SCJP/OCJP

Case (1) :

Q: which of the following are valid Var-arg methods?

m1 (int... x) ----> // valid

m1 (int ...x) ----> // valid

m1 (int x...) ----> // invalid

m1 (int. ..x) ----> // invalid

m1 (int. x..) ----> // invalid

m1 (int...x) ----> // valid

Various syntax loop holes for SCJP/OCJP

Case (3) :

If we mix normal parameter with Var-arg parameter then Var-arg parameter should be last parameter.

Example:

```
m1 (String... x, int y)   // invalid
m1 (int x, String... y)   // valid
```

Various syntax loop holes for SCJP/OCJP

Case (4) :

Within Var-arg method we can take only one var-arg parameter.
i.e., we can't declare more than one var-arg parameters inside var-arg method.

Example:

```
m1 (int... x, String... y)    // invalid
```

Various syntax loop holes for SCJP/OCJP

Case (5) :

* In general var-arg method will get least priority i.e., if no other method matched then only var-arg method will get the chance. It is exactly same as default case inside switch.

```
class Test {
   public static void m1 (int x) {
      System.out.println (“General Method”);
   }
   public static void m1 (int… x) {
      System.out.println (“Var-arg Method”);
   }
   public static void main (String [] args) {
        m1 (); \\var-method
        m1 (10, 20); \\var-arg method
        m1 (10); \\general method);
     }
}
```

Various syntax loop holes for SCJP/OCJP

Case (6) :

* With in a class if we are declaring var-arg parameter method then in the same class we can't declare corresponding one-dimensional array method.

Example :

```
class Test {
    public static void m1 (int [] x) {
    }
    public static void m1 (int... x) {
    }
}
```

Compile time Error: Cannot declare both m1 (int []) and m1 (int...) in Test