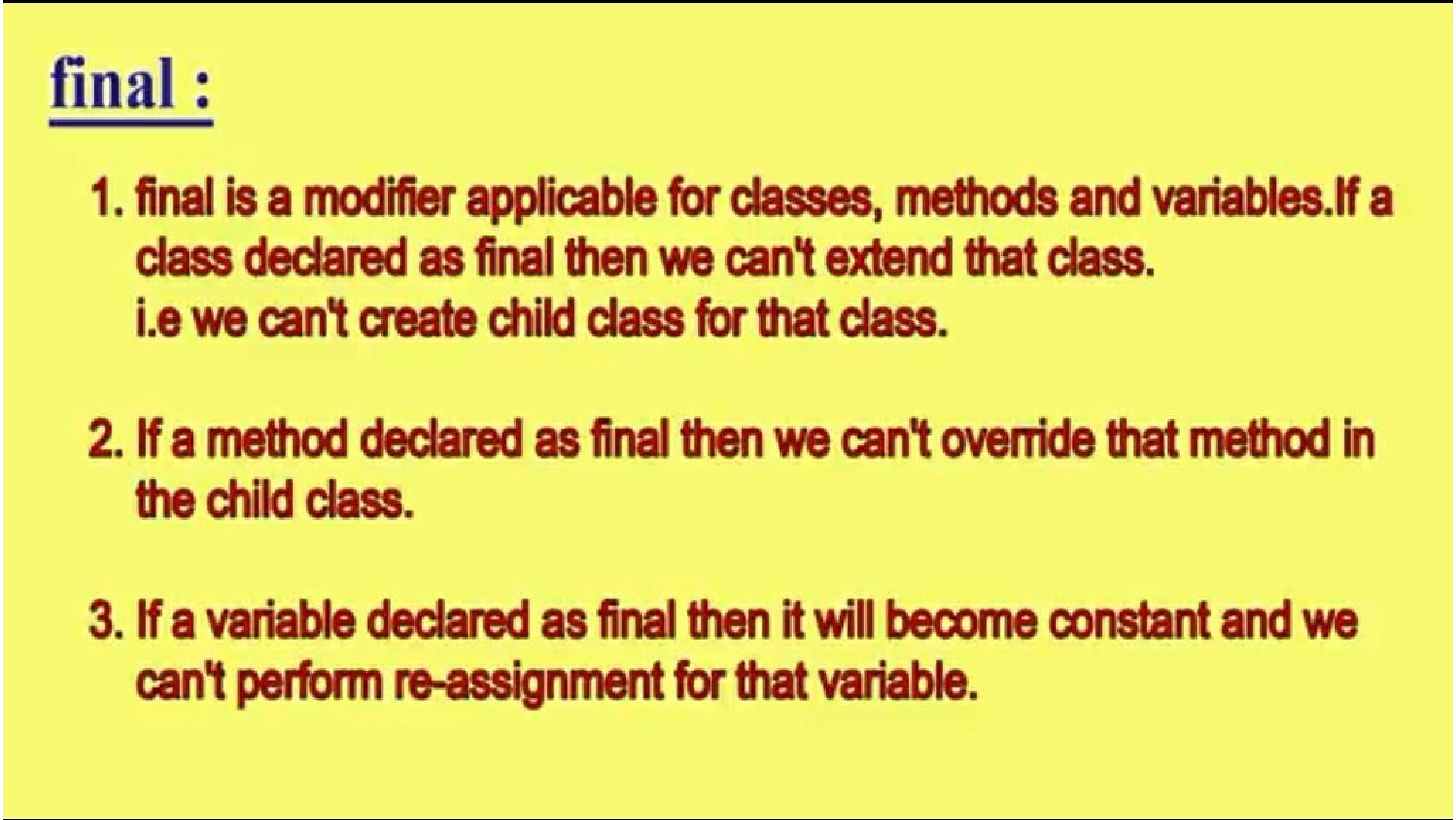


final :

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
}
```

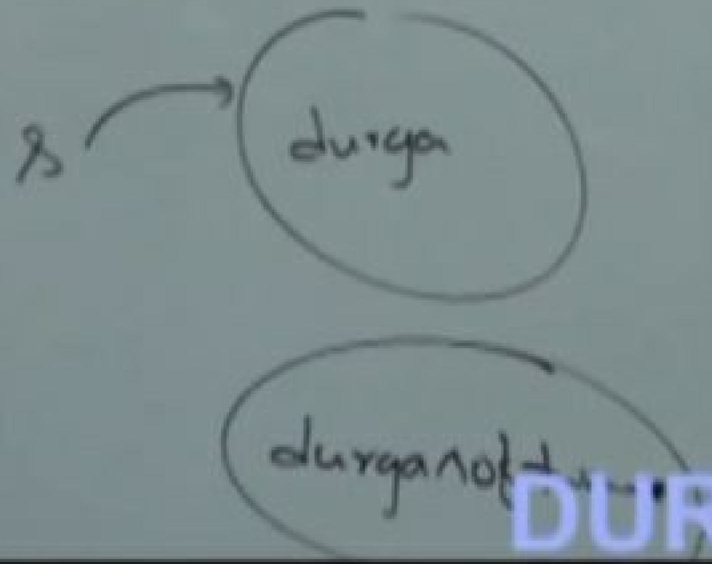
finalize():

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

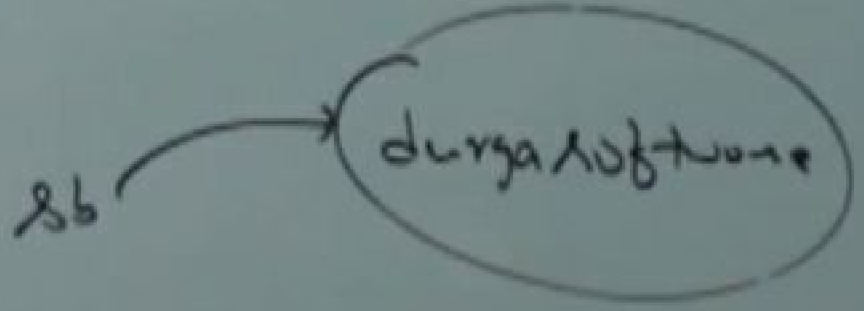
Note :

* finally meant for cleanup activities related to try block. where as finalize() meant for cleanup activities related to object.

```
String s = new String("durga");
s.concat("software");
Sopln(s); durga ✓
```



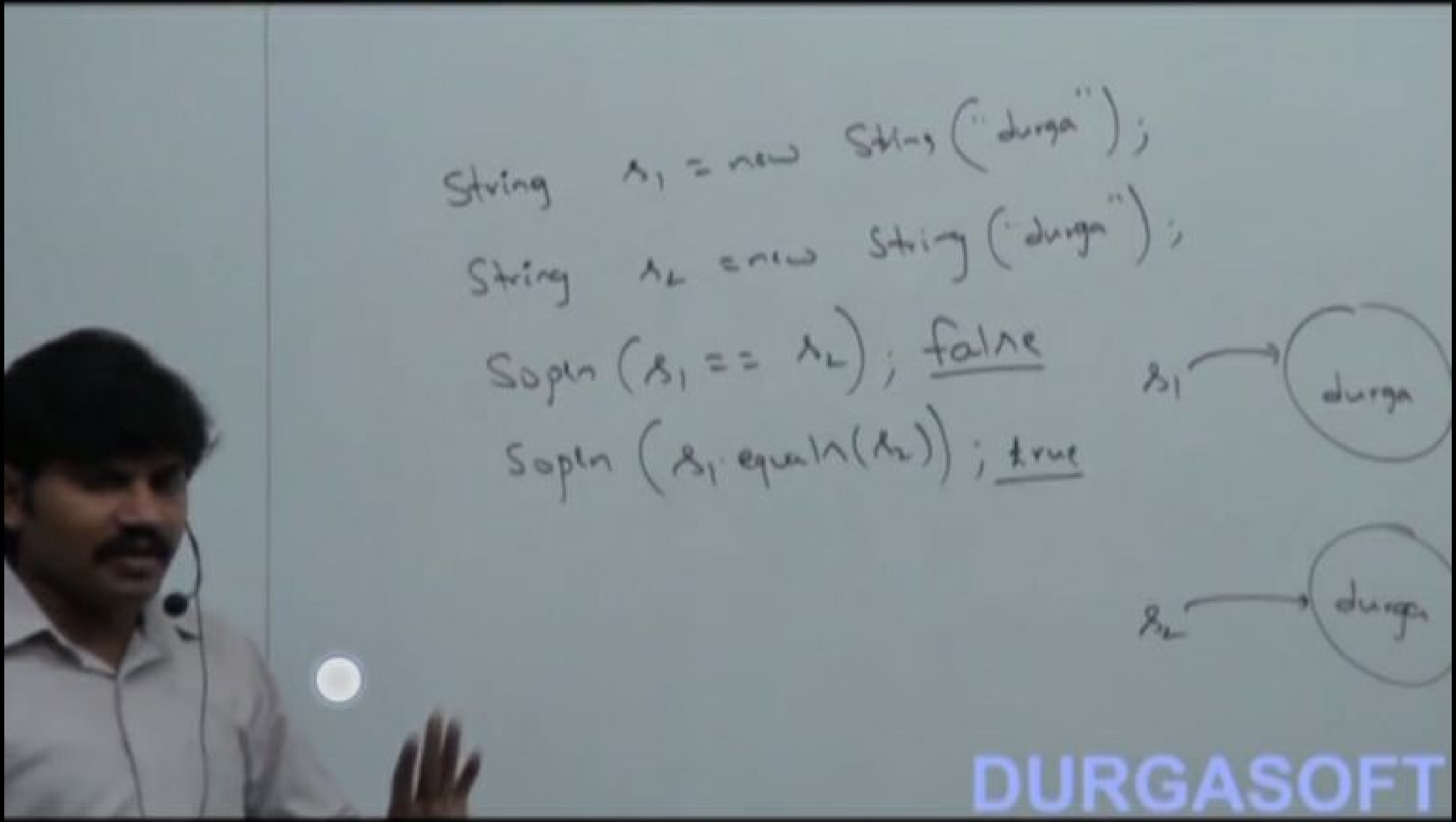
```
StringBuffer sb = new SB("durga");
sb.append("software");
Sopln(sb); durgasoftware
```



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.



```
String s1 = new String("durga");
String s2 = new String("durga");
Sopln(s1 == s2); false
Sopln(s1.equals(s2)); true
```

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

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

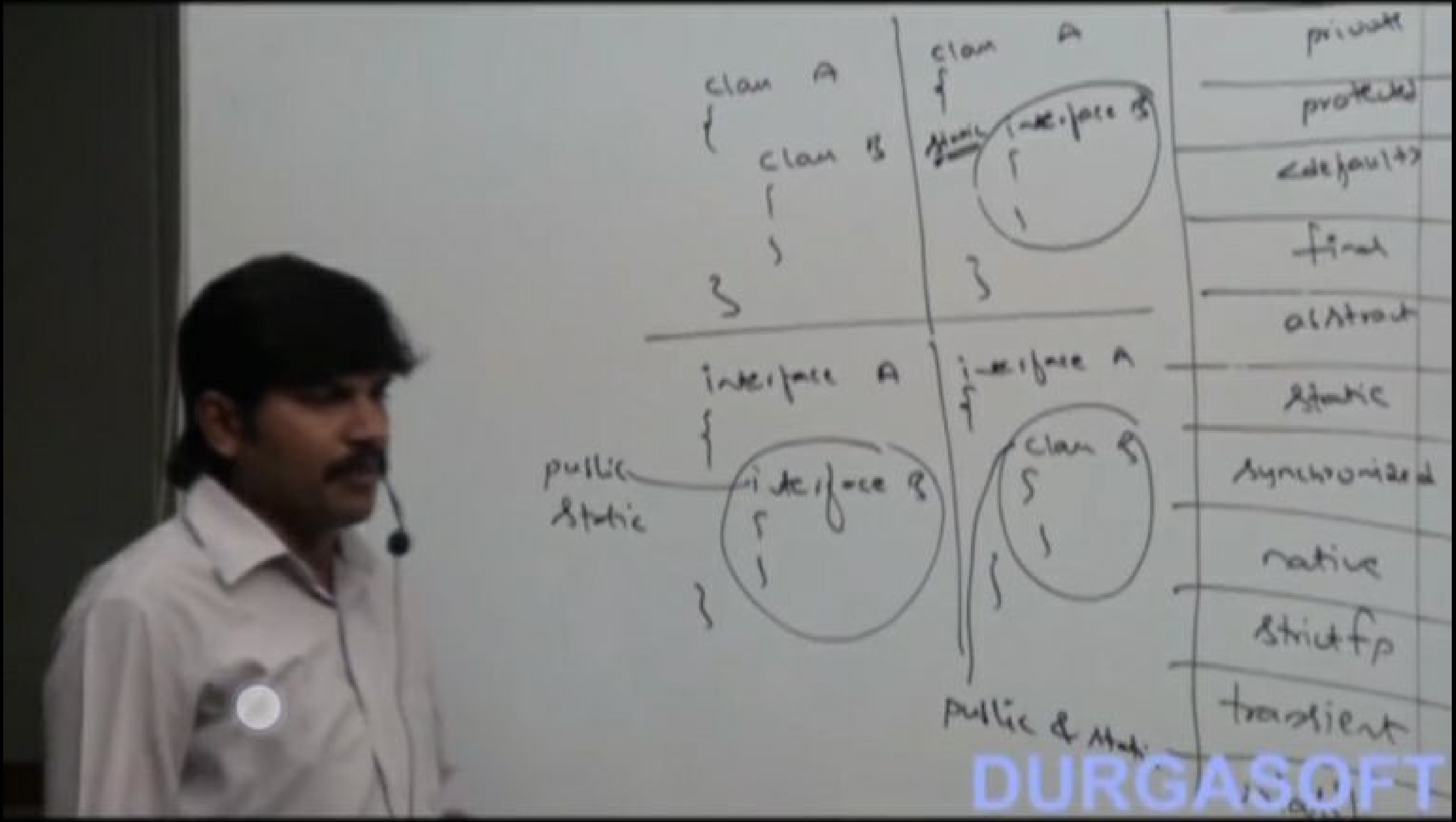
Note :

* .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.

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.



```
class A
{
    class B
    {
    }
}
```

```
class A
{
static interface B
    {
    }
}
```

```
interface A
{
public
static  interface B
    {
    }
}
```

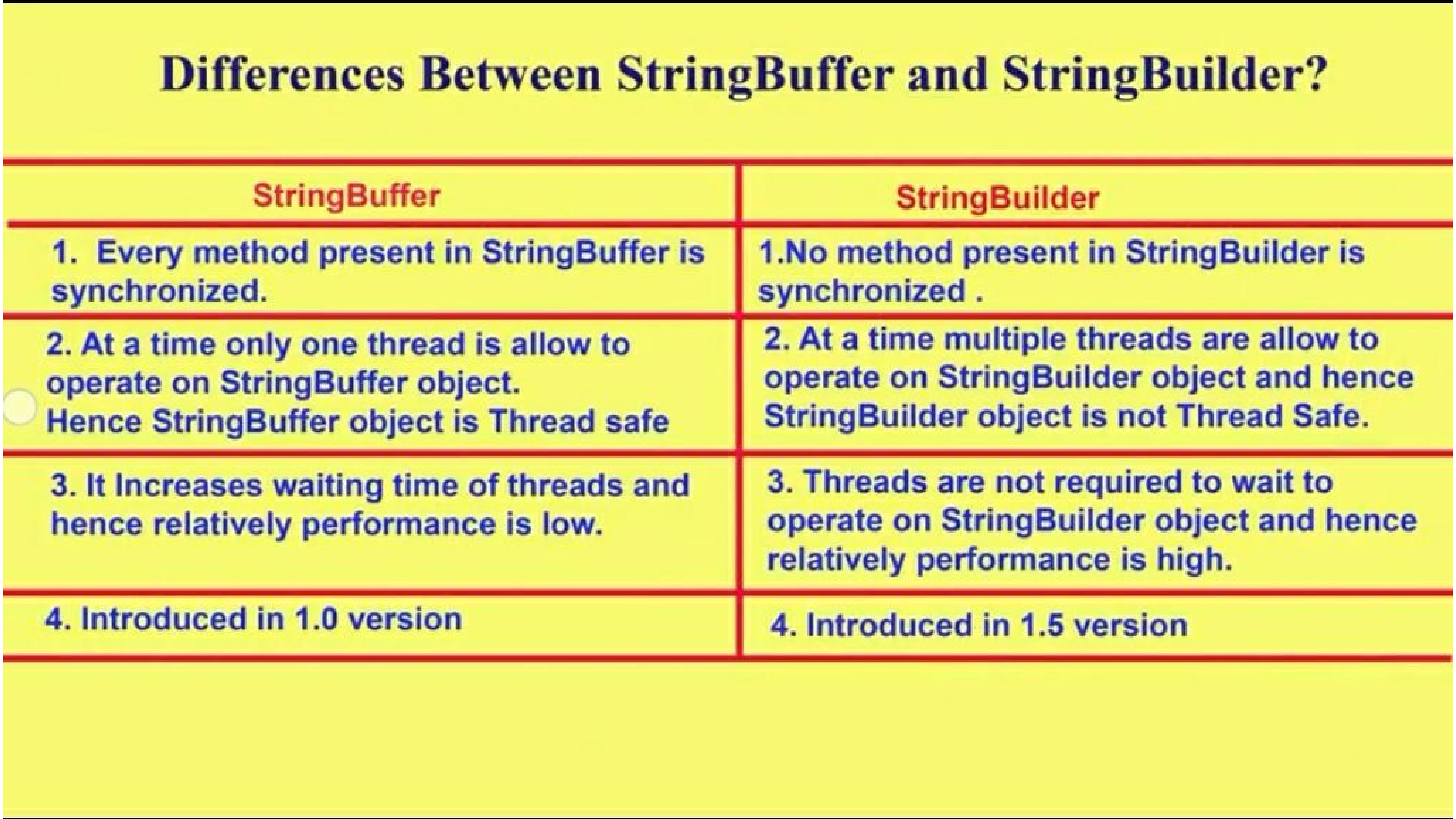
```
interface A
{
    class B
    {
    }
}
public & static
```

private
protected
<default>
final
abstract
static
synchronized
native
strictfp
transient

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.



Differences Between StringBuffer and StringBuilder?

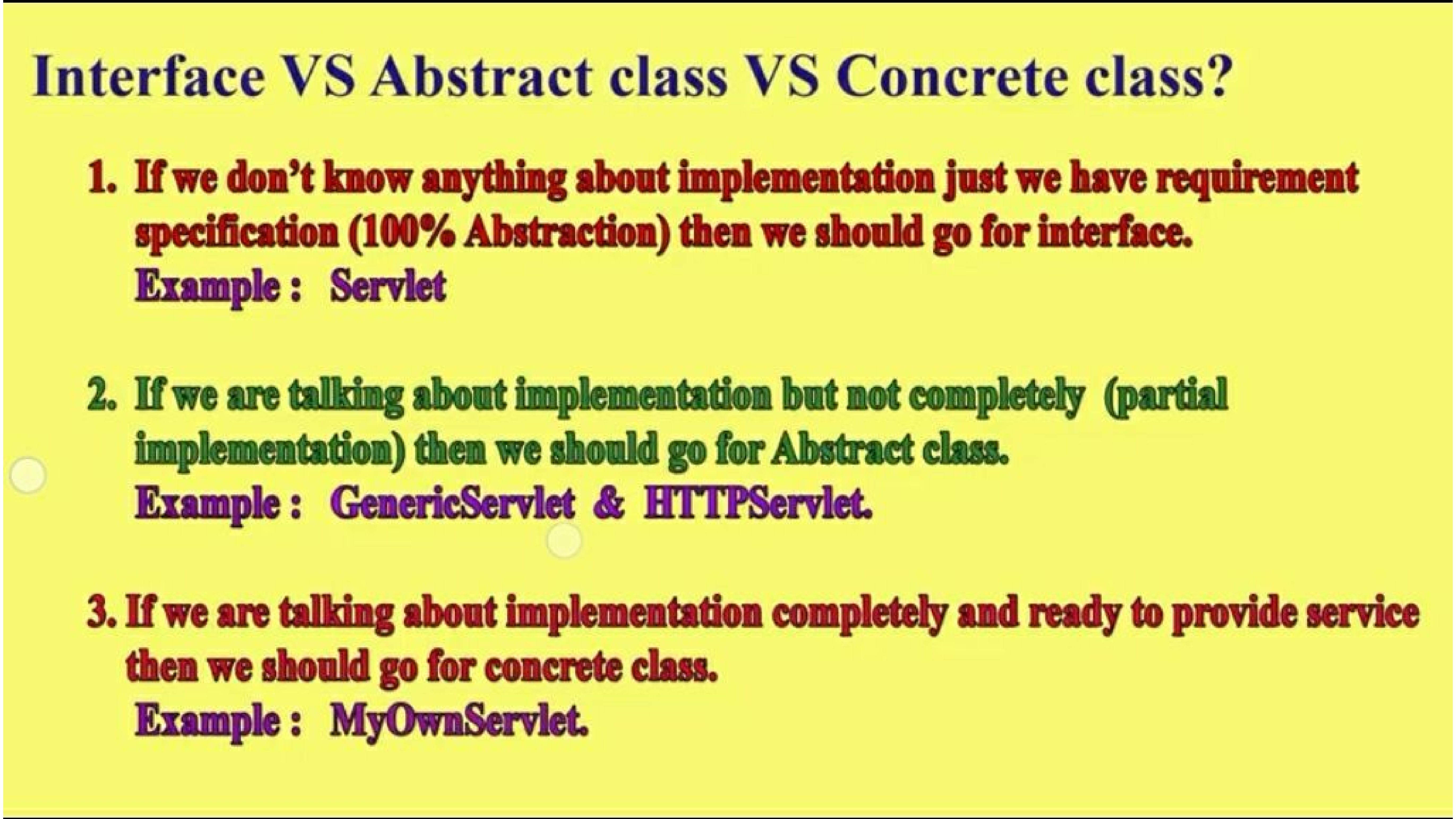
StringBuffer	StringBuilder
1. Every method present in StringBuffer is synchronized.	1.No method present in StringBuilder is synchronized .
2. At a time only one thread is allow to operate on StringBuffer object. Hence StringBuffer object is Thread safe	2. At a time multiple threads are allow to operate 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

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.

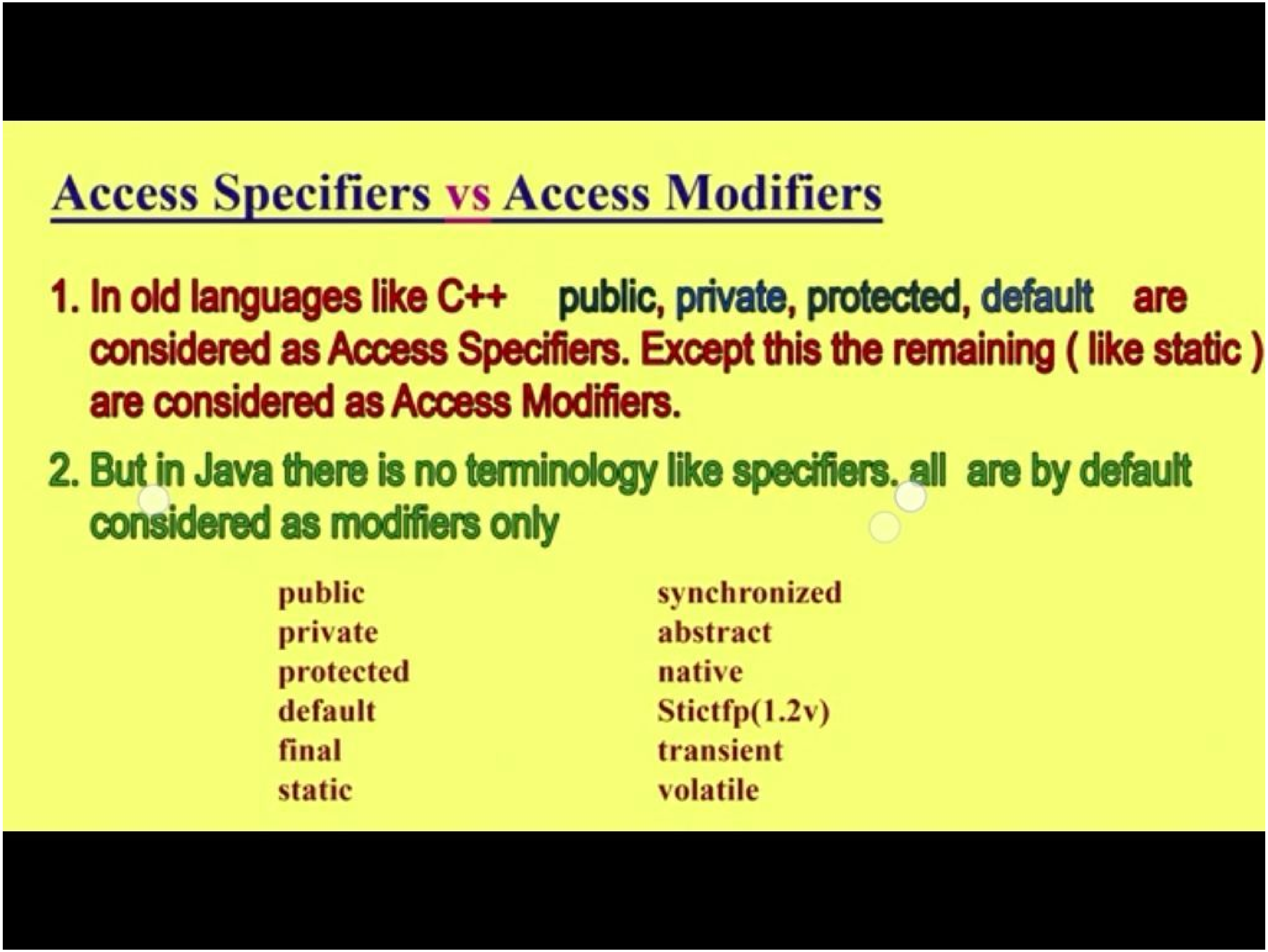


Interface VS Abstract class VS Concrete class?

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.

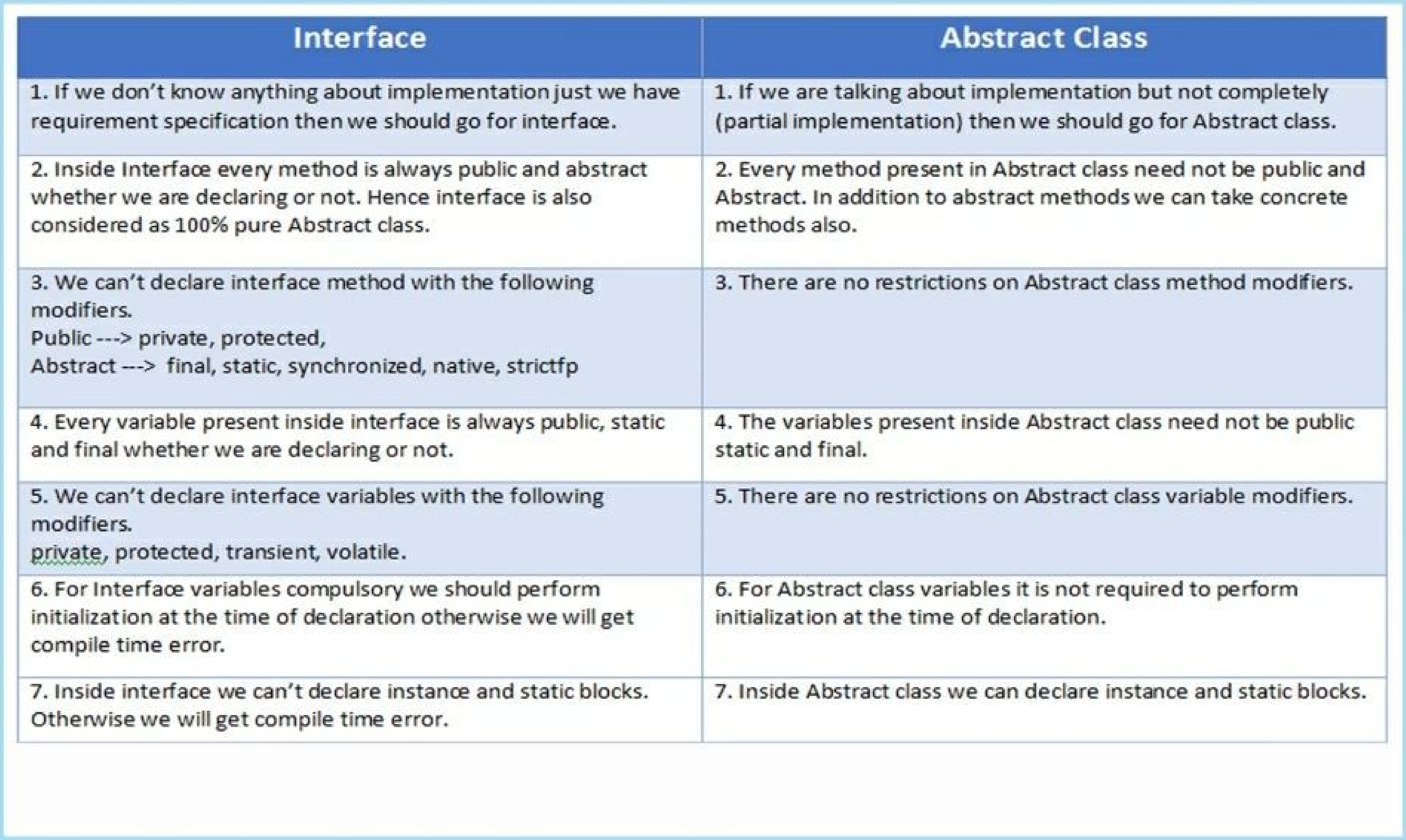


Access Specifiers vs Access Modifiers

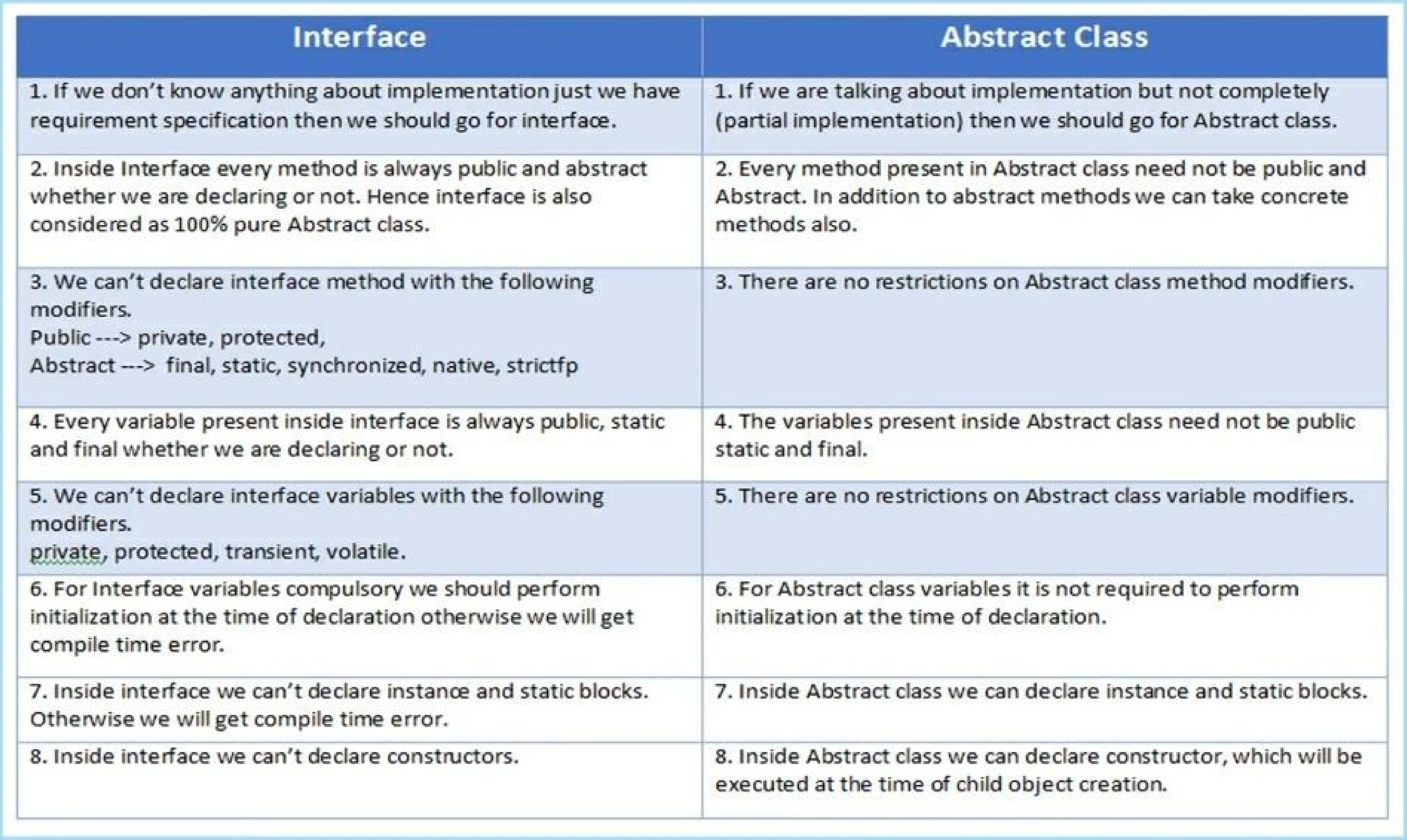
1. In old languages like C++ public, private, protected, default are considered as Access Specifiers. Except this the remaining (like static) are considered as Access Modifiers.
2. But in Java there is no terminology like specifiers. all are by default considered as modifiers only

public
private
protected
default
final
static

synchronized
abstract
native
Stictfp(1.2v)
transient
volatile



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.



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.

```
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 ;
}
```

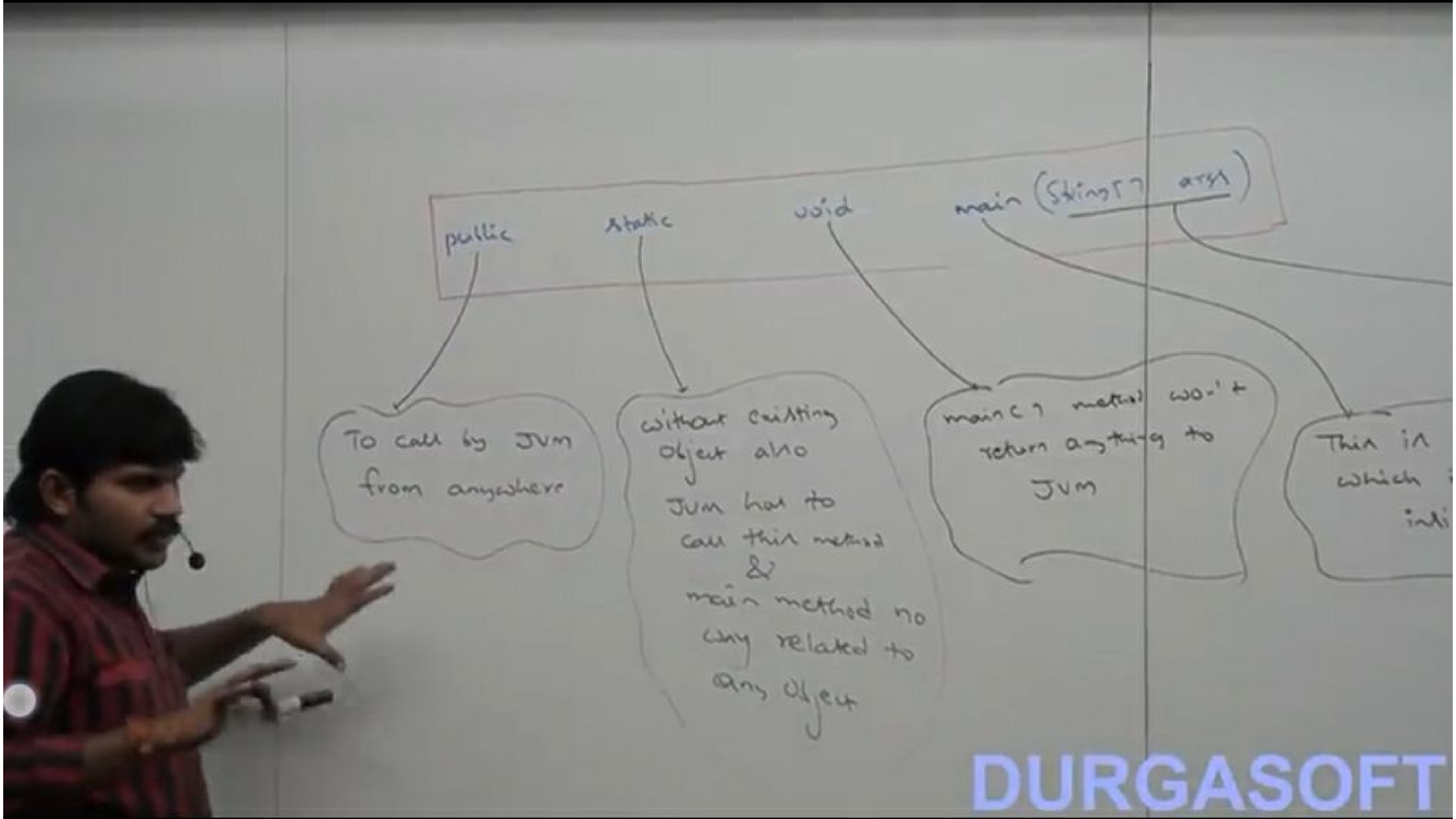
```
System.out.println("Hello");
```

* 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.

* 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
```



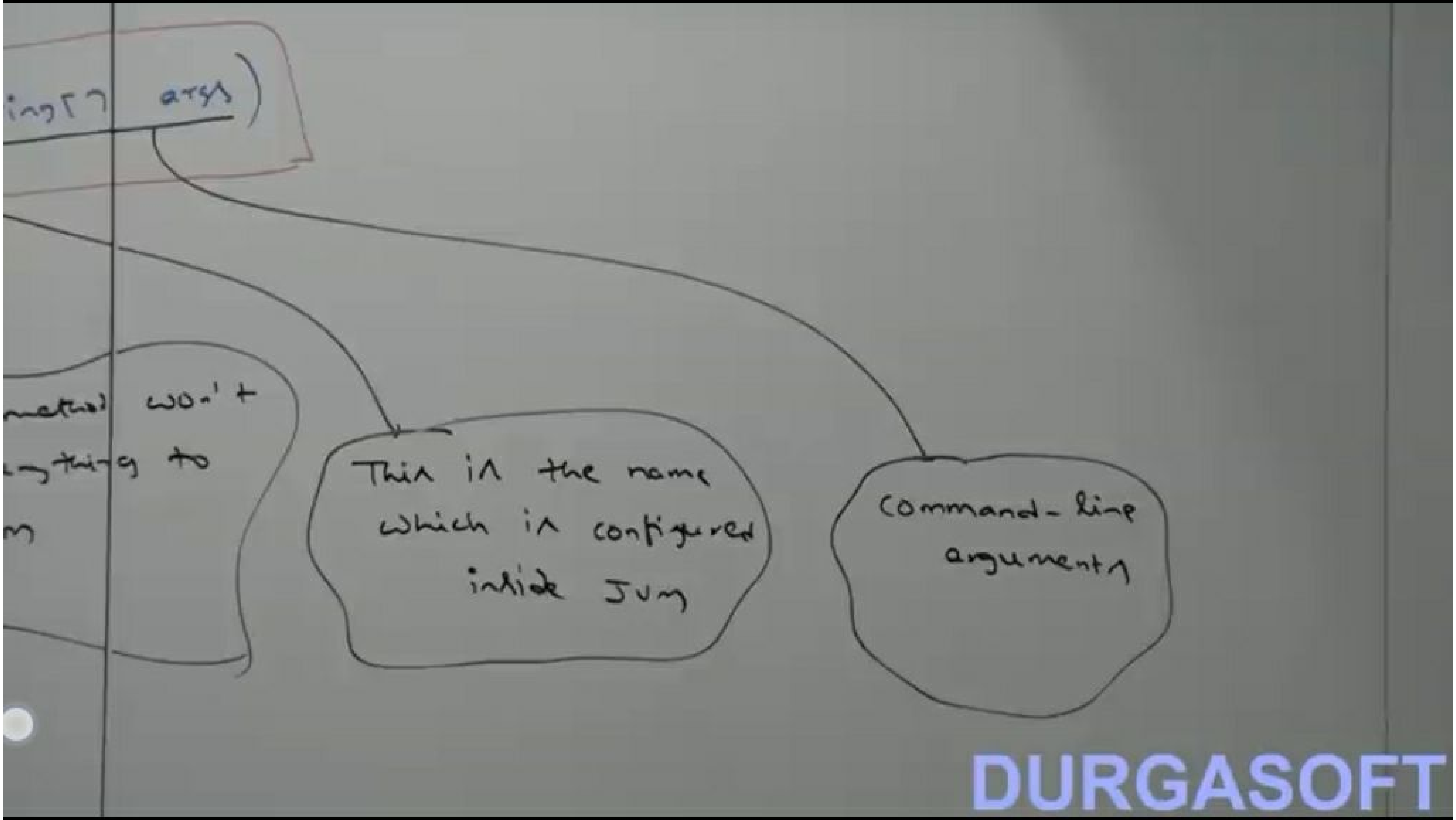
public static void main(String[] args)

To call by JVM from anywhere

without existing object also JVM has to call this method & main method no way related to any object

main() method won't return anything to JVM

This is
which
ini



ing[] args)

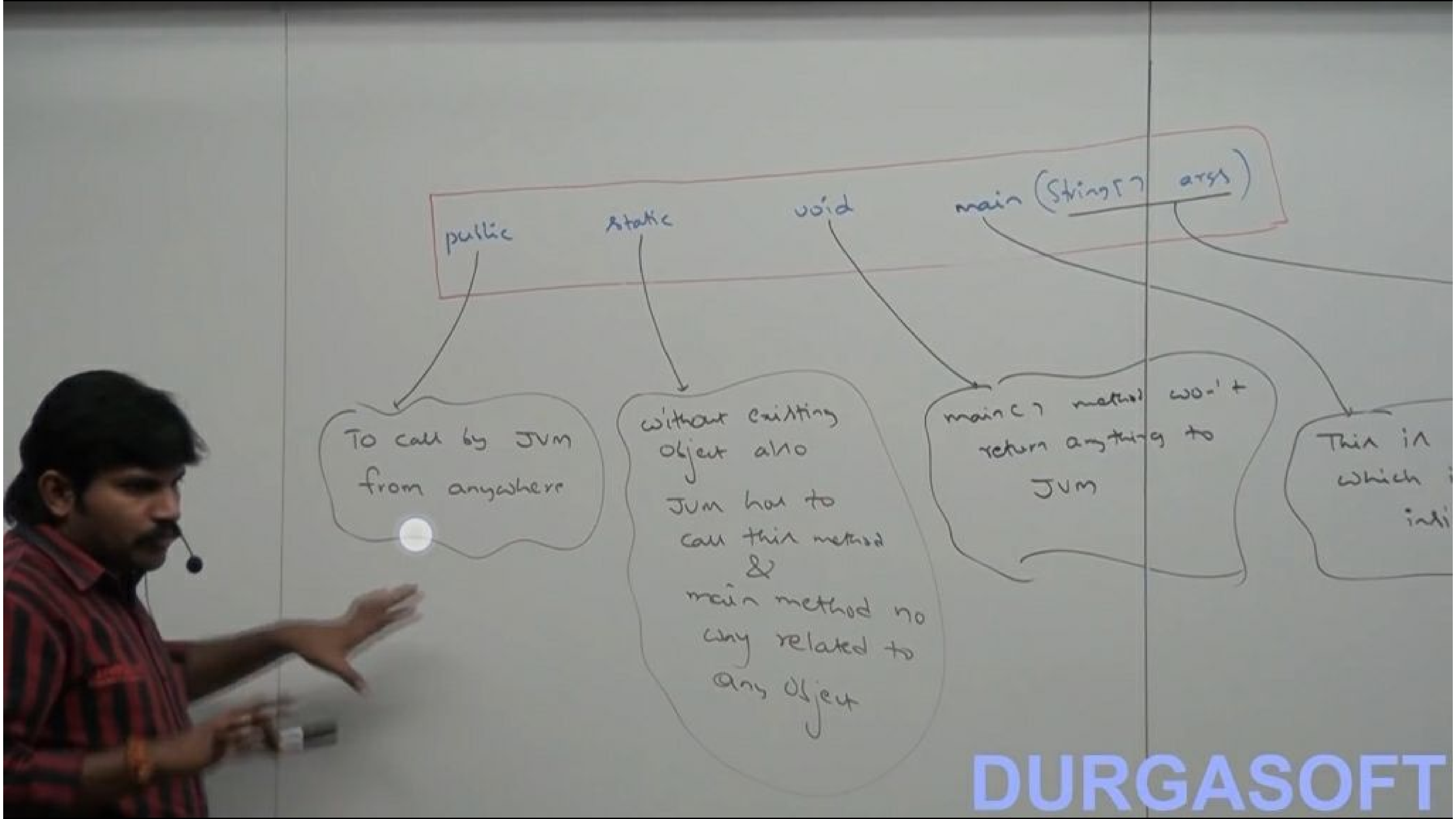
method won't

ything to

m

This is the name which is configured inside JVM

command-line arguments



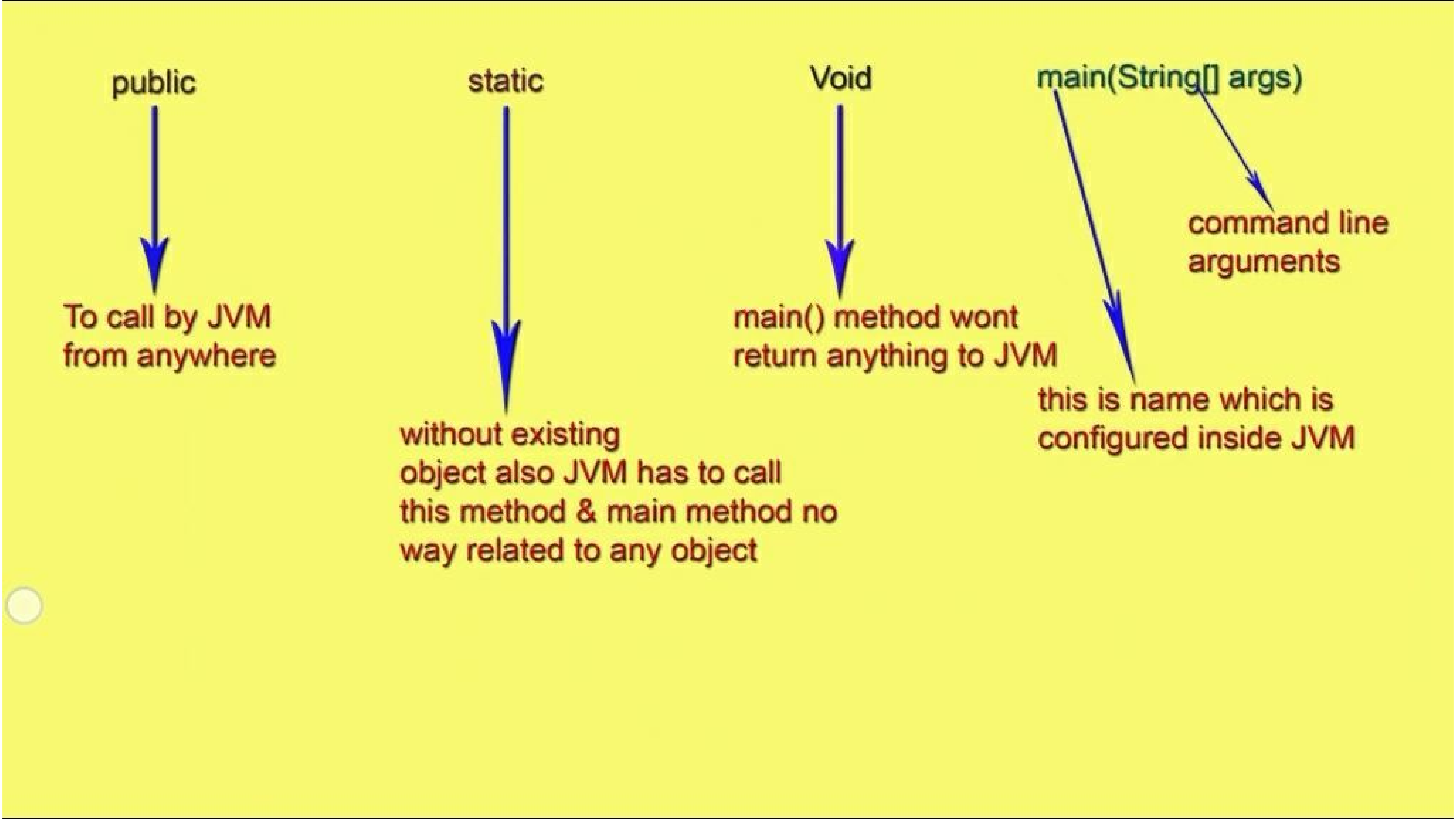
public static void main(String[] args)

To call by JVM from anywhere

without existing object also JVM has to call this method & main method no way related to any object

main() method won't return anything to JVM

This is
which
ini



public → To call by JVM from anywhere

static → without existing object also JVM has to call this method & main method no way related to any object

Void → main() method wont return anything to JVM

main(String[] args)

main → this is name which is configured inside JVM

String[] args → command line arguments

* The above syntax is very strict if we perform any change we will get runtime exception saying NoSuchMethodError:main.

* Even though the above syntax is very strict the following changes are acceptable.

1. The order of modifiers is not important that is instead of “public static” we can take “static public” also.

2. We can declare "String[]" in any acceptable form.

```
main(String[] args)

main(String []args)

main(String args[])
```

3. instead of ‘args’ we can take any valid java identifier.

4. we can replace String[] with var arg perameter

```
main(String[] args) ==> main(String… args)
```

5. We can declare main() method with the following modifiers also.

final
synchronized
strictfp

Q. which of the following are valid main method declarations ?

✗ 1) public static void Main(String[] args)

✗ 2) public static int main(String[] args)

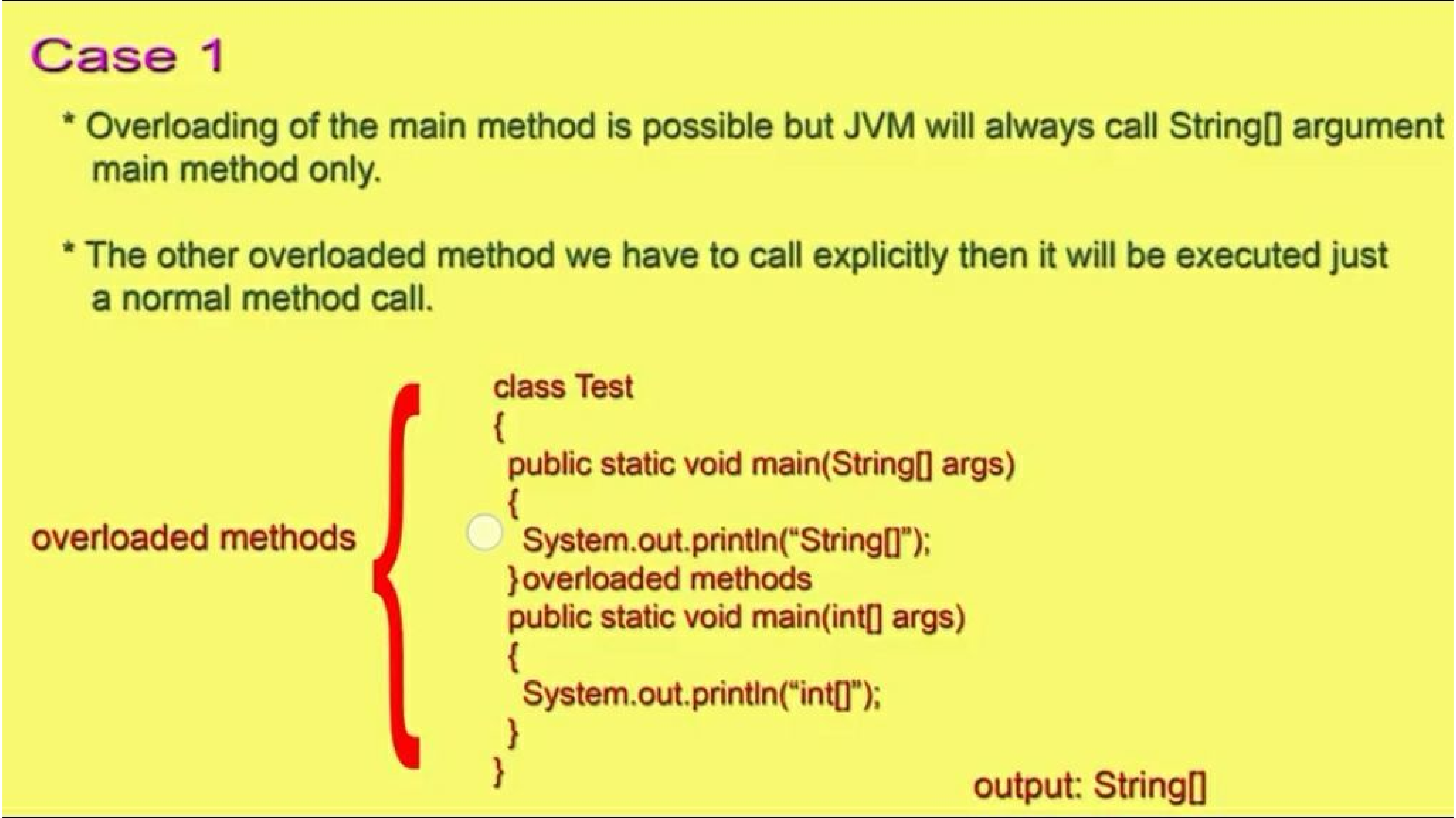
✗ 3) public static void main(String args)

✗ 4) public final synchronized strictfp void main(String[] args)

✓ 5) public static final synchronized strictfp void main(String[] args)

✓ 6) public static void main(String... args)

* we won't get compile time error any where but at runtime we will get exception in all cases except last two.



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[]");
 }
}
```

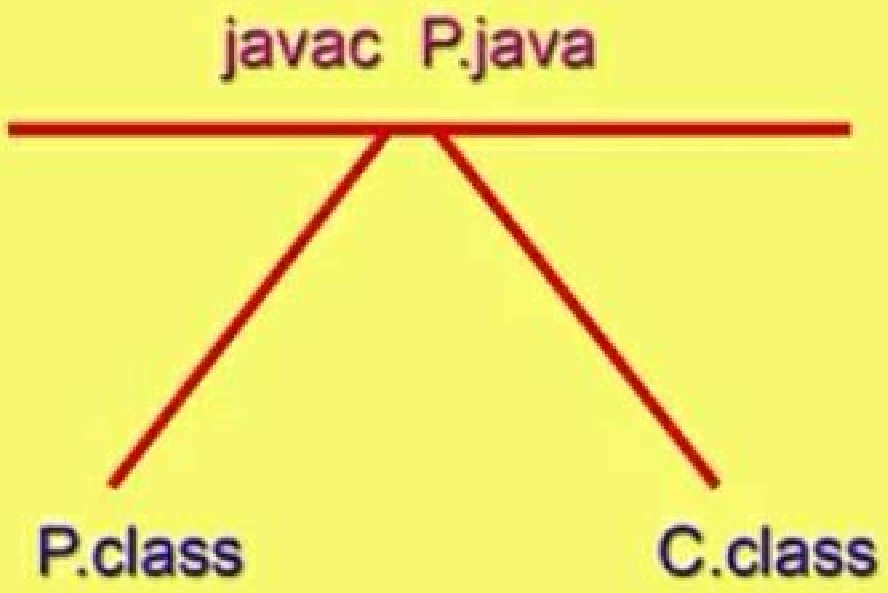
output: String[]

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
{
}
```



java P

output: parent main

java C

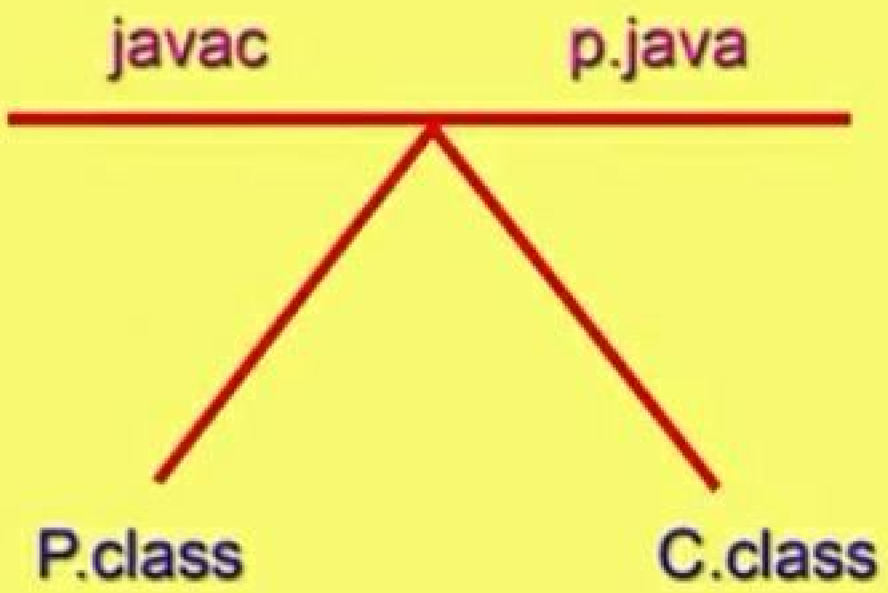
output: parent main

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");
}
}
```



java P

output: parent main

java C

output: child main

Note:

*. For main method inheritance and overloading concepts are applicable but overriding concept is not applicable instead of overriding method hiding concept is applicable.

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 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");
}
}
```

Page 1 of 2

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 3

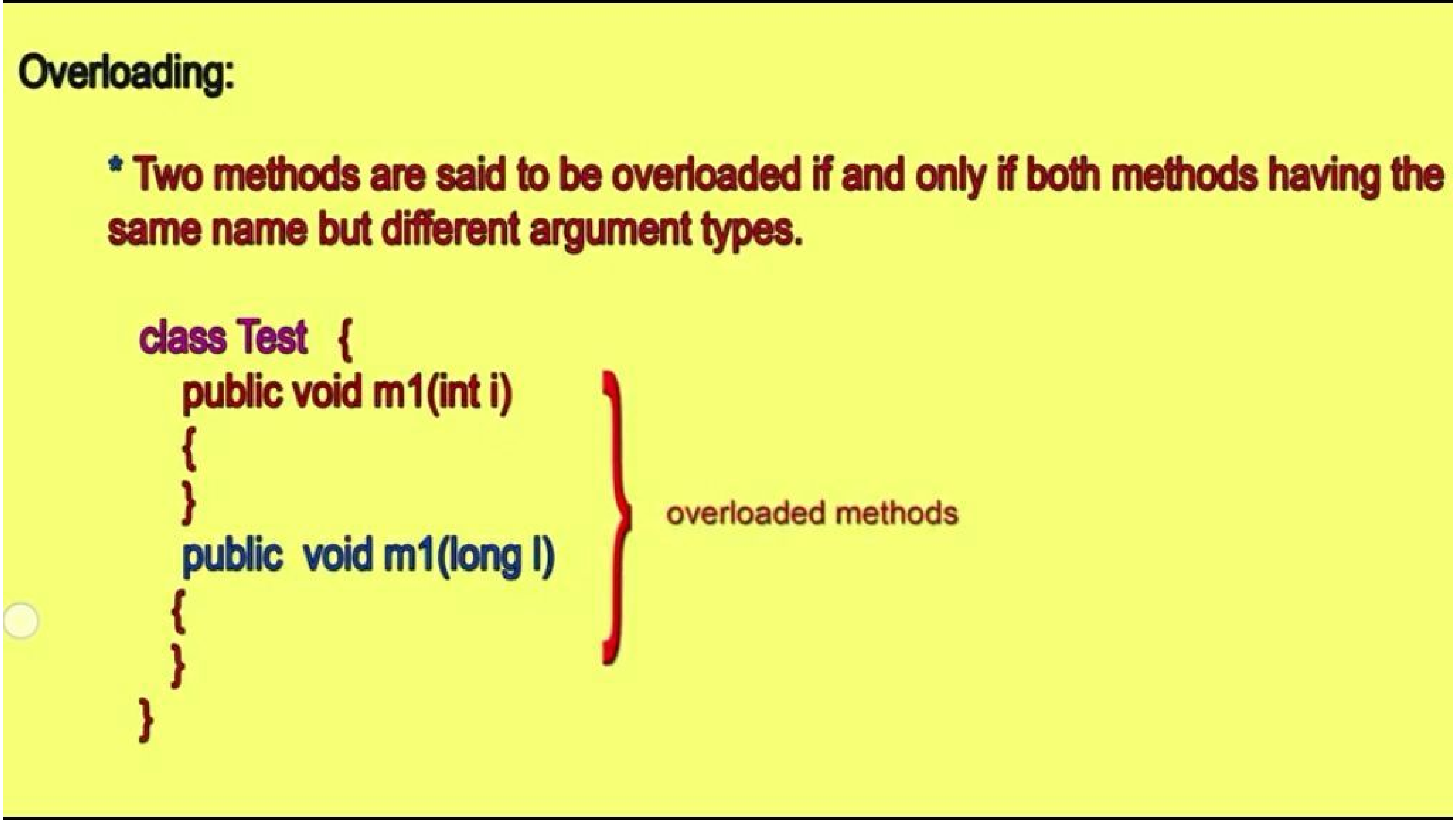
If the class contains main() method whether it is 1.6 or 1.7 version there is no change in execution sequence.

```
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.

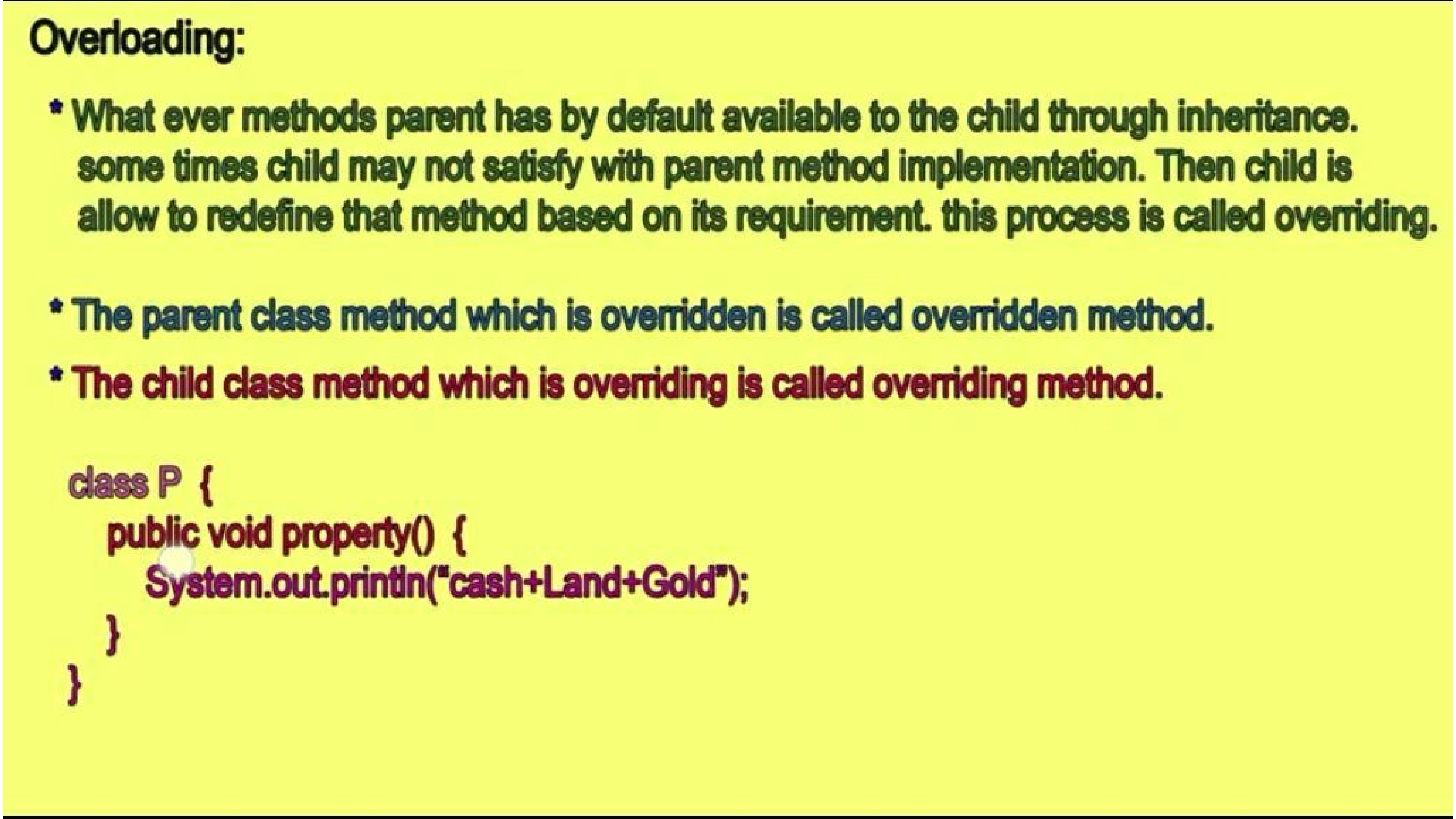


Overloading:

* Two methods are said to be overloaded if and only if both methods having the same name but different argument types.

```
class Test  {
   public void m1(int i)
   {
   }
   public  void m1(long l)
  {
  }
}
```

overloaded methods



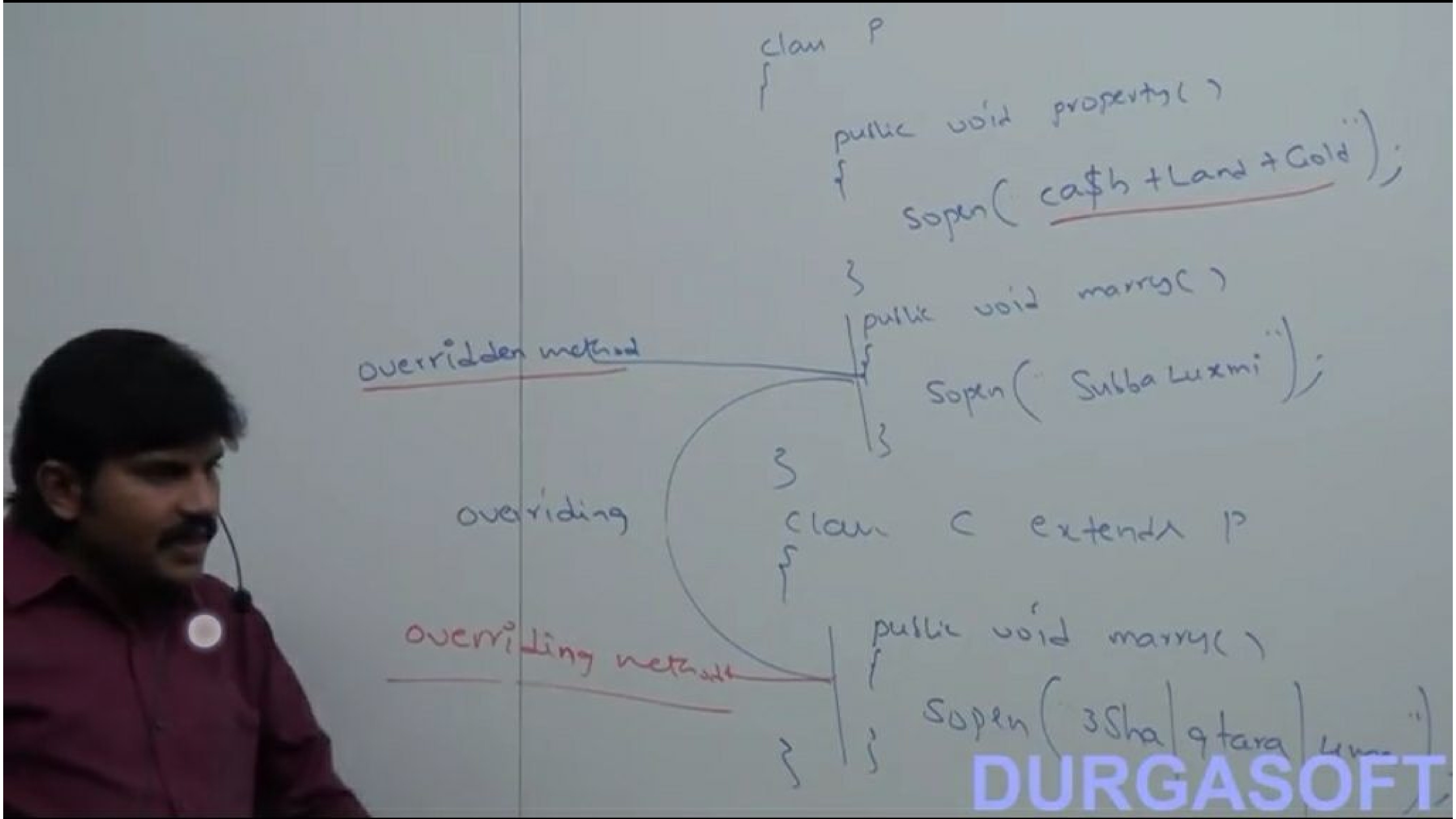
Overloading:

* What ever methods parent has by default available to the child through inheritance. some times child may not satisfy with parent method implementation. Then child is allow to redefine that method based on its requirement. this process is called overriding.

* The parent class method which is overridden is called overridden method.

* The child class method which is overriding is called overriding method.

```
class P {
   public void property() {
      System.out.println("cash+Land+Gold");
   }
}
```

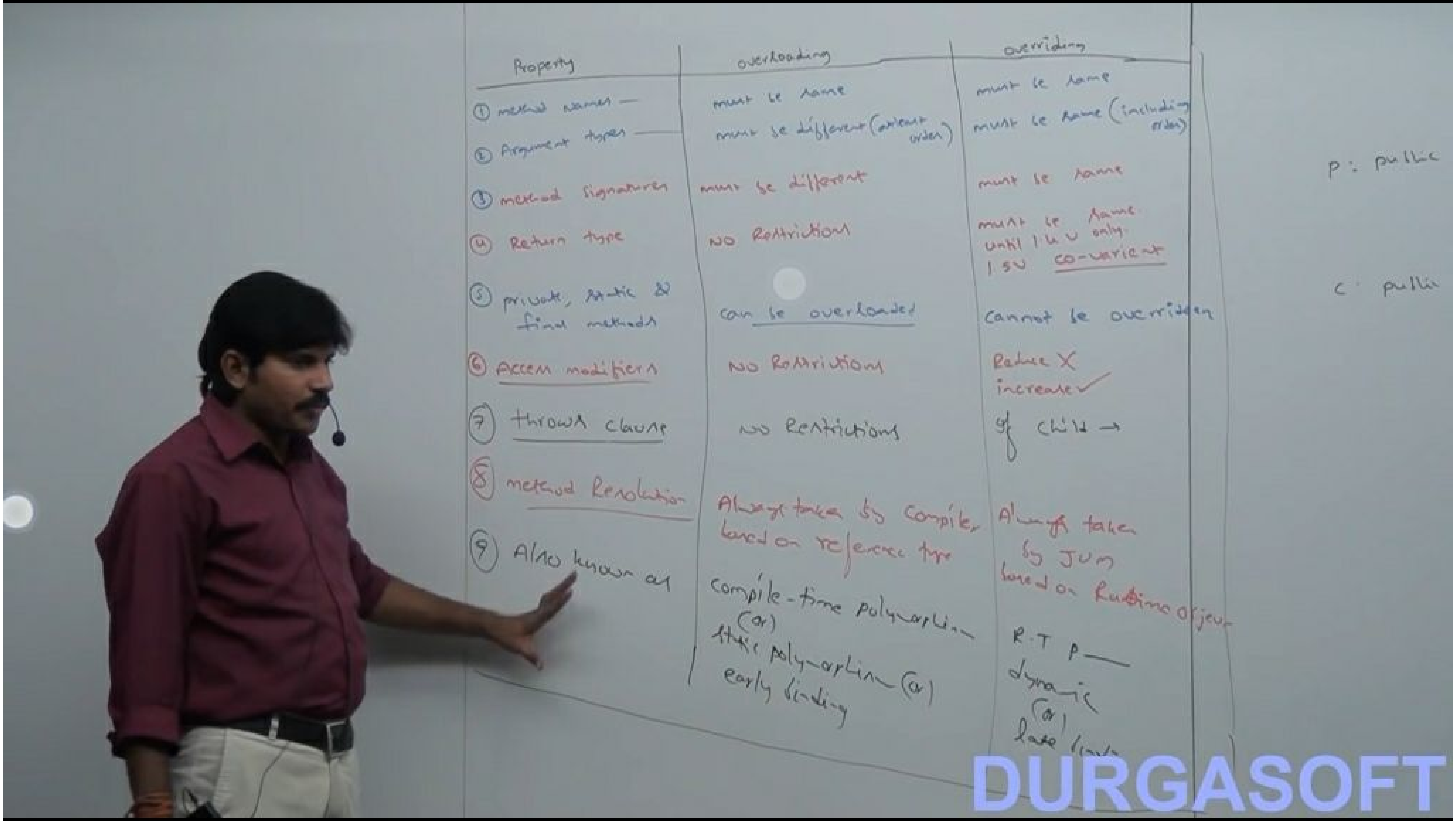


```
class P
{
    public void property()
    {
        sopln("ca$h+Land+Gold");
    }
    public void marry()
    {
        sopln("Subba Luxmi");
    }
}
class C extends P
{
    public void marry()
    {
        sopln("3Sha|9tara|4[illegible]");
    }
}
```

overridden method

overriding

overriding method



Property	overloading	overriding
① method names —	must be same	must be same
② Argument types —	must be different (atleast order)	must be same (including order)
③ method signatures	must be different	must be same
④ Return type	No Restrictions	must be same until 1.4 v only 1.5 v co-varient
⑤ private, static & final methods	can be overloaded	cannot be overridden
⑥ Access modifiers	No Restrictions	Reduce ✗ increase ✓
⑦ throws clause	No Restrictions	If child →
⑧ method Resolution	Always taken by compiler based on reference type	Always taken by JVM based on Runtime object
⑨ Also known as	compile-time polymorphism (or) static polymorphism (or) early binding	R.T.P — dynamic (or) late binding

P : public

C : public

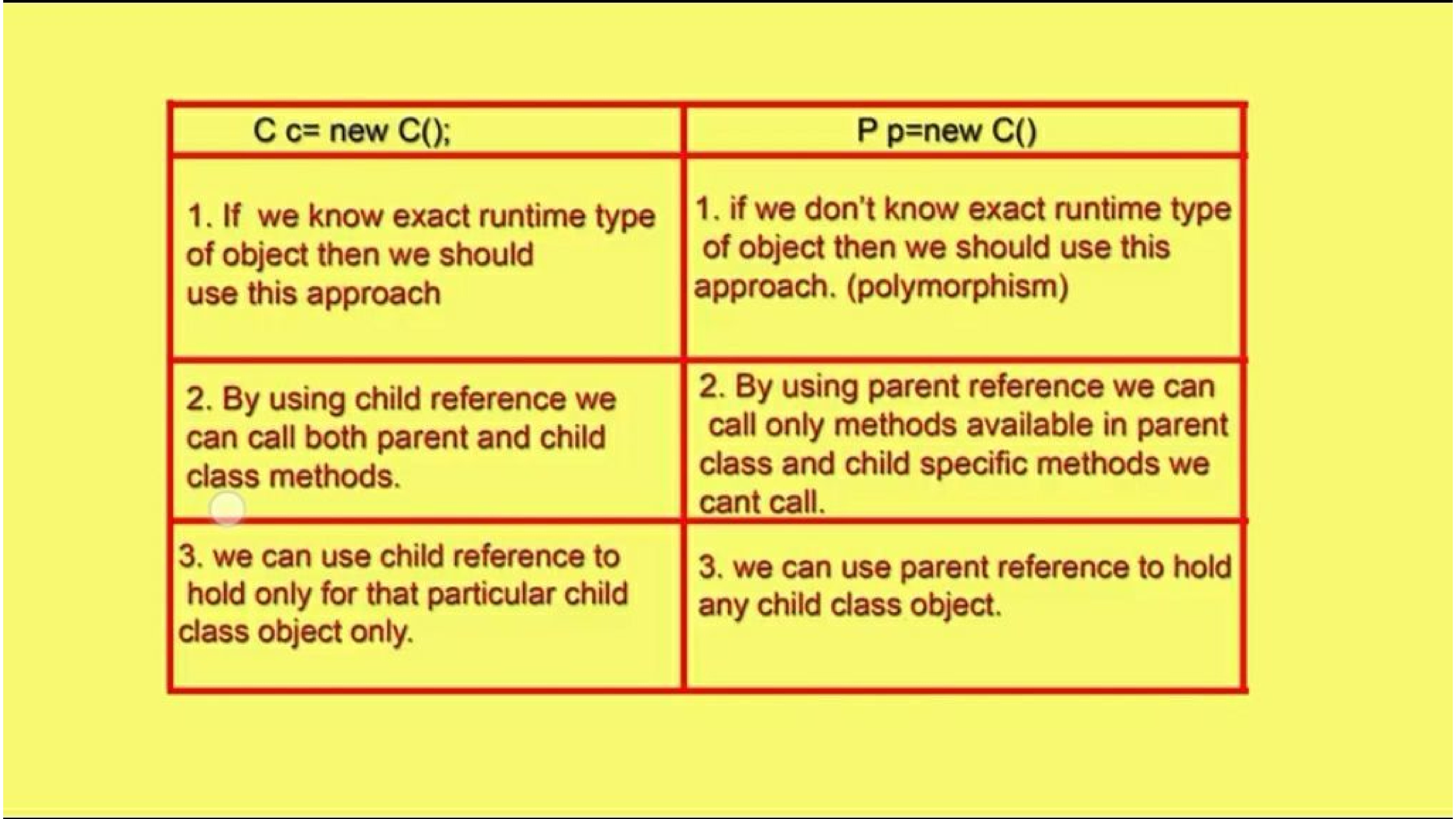
Property	Overloading	Overriding
1. Method names	Must be same	Must be same
2. Argument Types	Must be different (at least order)	Must be same (Including order)
3. Method signatures	Must be different	Must be same.
4. Return Type	No Restrictions	Must be same but this rule is applicable until 1.4version only. 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.

Note:

* In overloading we have to check only method names (must be same) and argument types (must be different) except this the remaining like return types, access modifiers etc., are not required to check.

* But in overriding every thing we have to check like method names argument types, return types, access modifiers etc.

Difference between C c =new C(); and P p =new C()



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.

Various possible combinations of try- catch- finally?

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.

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

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.

Throwable acts as root for java exception hierarchy

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

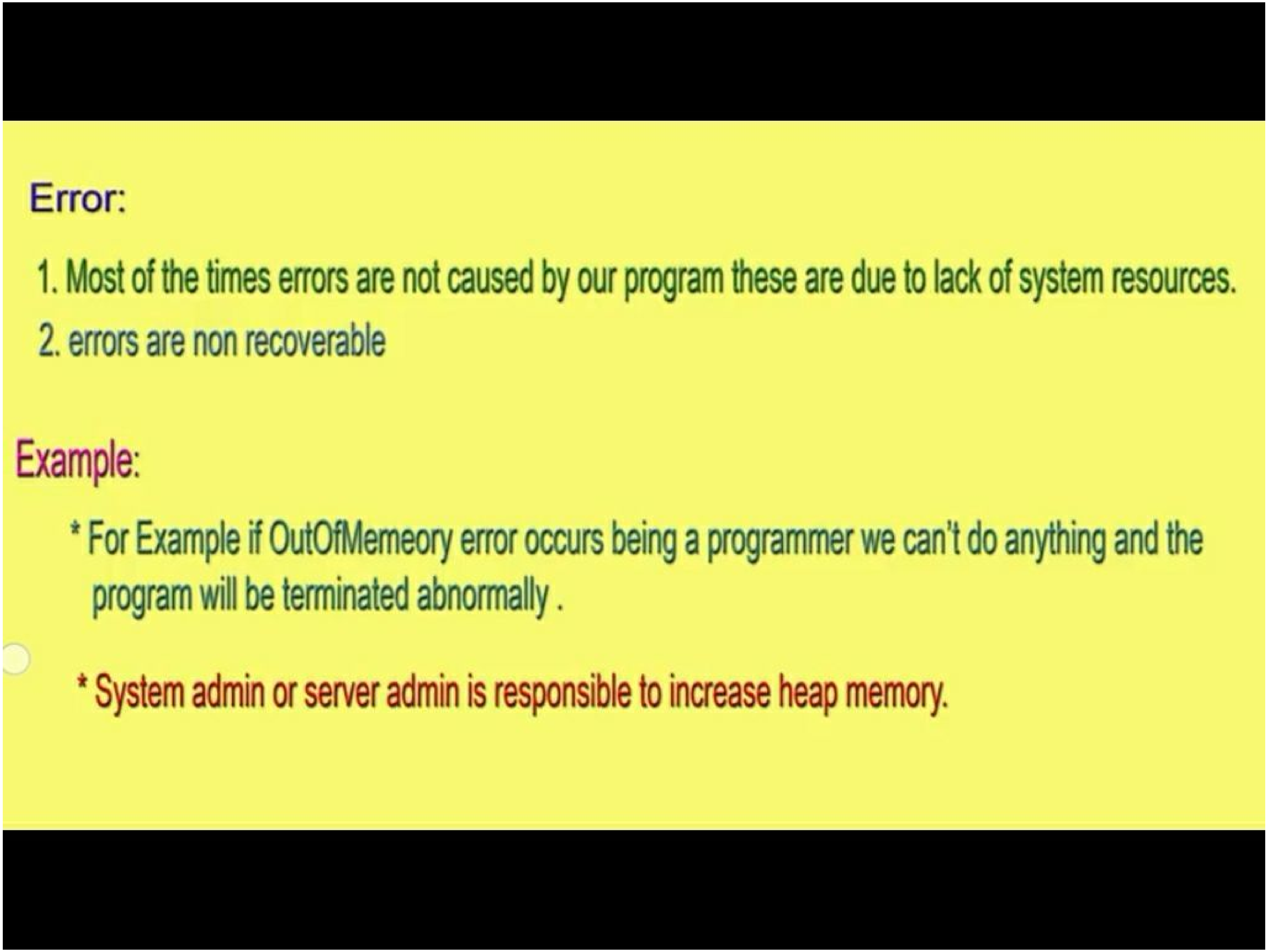
Example:

* 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.

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.

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.

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.

Differences between fully checked and partially 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