Java Important Topics & Answers

1) Principles of OOP

Principles of Object Oriented Programming

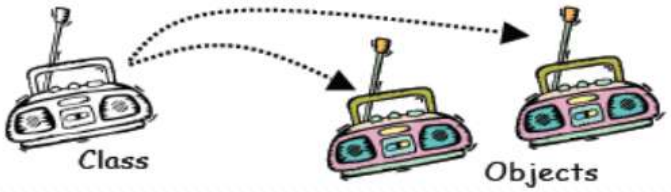
- **Object** means a real word entity such as pen, chair, table etc.
- **Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects. It simplifies the software development and maintenance by providing some concepts:

1. **Object**
2. **Class**
3. **Encapsulation**
4. **Inheritance**
5. **Polymorphism**
6. **Abstraction**

Dr. Y. J. Nagendra Kumar - 6

Class

- A class is a set of objects with:
  - Same interface, same behavior, **different identify**
- Classes allow modeling of simple classification in the real world, and simplify design
- An object is an "instance" of a class



Dr. Y. J. Nagendra Kumar - 7

Objects

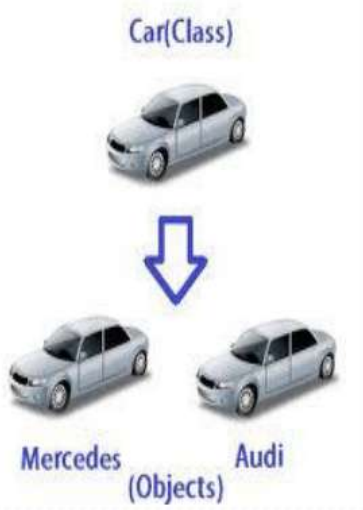
- An object has "memory"

Object State:

– **A bank account:** a number, a name, an address, a balance, an overdraft limit, a branch, a PIN number

Object Behaviour

– **A bank account:** withdraw, deposit, calculate interest, print statement.



Encapsulation

- **Binding (or wrapping) code and data together into a single unit is known as encapsulation**. Example: Class
- For example: capsule, it is wrapped with different medicines.

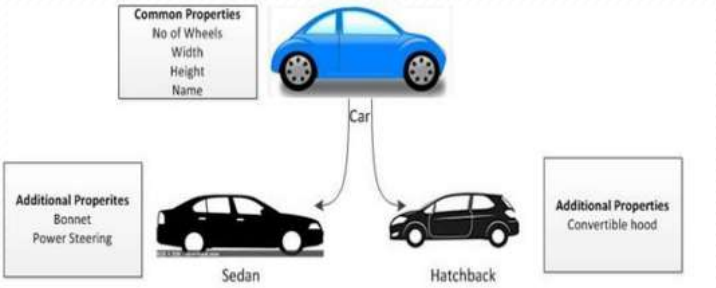
- **Data Fields are private**
- **Constructors and accessor methods are defined**

Person
-name : String -age : int
+Person(String name, int age) +getName() : String +setName(String name) +getAge() : int


Capsule

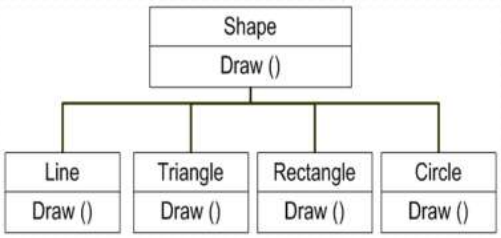
Inheritance

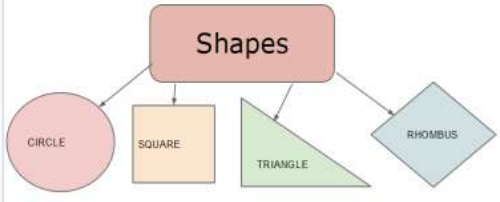
- When one object acquires all the properties and behaviours of parent object i.e. known as **Inheritance.**
- It provides code reusability.



Polymorphism

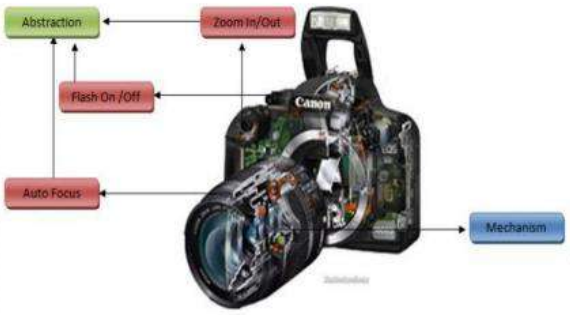
- A property of object oriented software by which an abstract operation may be performed in different ways in different classes.
  - Requires that there be multiple methods of the same name
  - The choice of which one to execute depends on the object.





Abstraction

- **Hiding internal details and showing functionality** is known as abstraction.
- For example: phone call, we don't know the internal processing.



2) Java Buzzwords or Features

Java Features / Buzzwords

- Simple
- Secure
- Portable
- Object-oriented
- Robust
- Multithreaded
- Architecture-neutral
- Interpreted
- High performance
- Distributed
- Dynamic

Simple

• Java was designed to be easy for the professional programmer to learn and use effectively.

• If we already understand the basic concepts of object-oriented programming, learning Java will be even easier.

• If we are an experienced C++ programmer, moving to Java will require very little effort. Because Java inherits the C/C++ syntax and many of the object-oriented features of C++

Secure

• Java achieved this protection by digital signatures for JAR files

Portable

• Portability is a major aspect of the Internet because there are many different types of computers and operating systems connected to it.

• If a Java program were to be run on virtually any computer connected to the Internet, there needed to be some way to enable that program to execute on different systems.

• Write once, run everywhere

Object Oriented

• Everything in Java is an Object. All program code and data reside within objects and classes.

•Java comes with an extensive set of classes arranged in Packages.

Robust

- Strong typing + no pointer + garbage collection
- The program must execute reliably in a variety of systems.
- Java is a strictly typed language, it checks your code at compile time. However, it also checks your code at run time.

Multithreaded

- Java was designed to meet the real-world requirement of creating interactive, networked programs.
- To accomplish this, Java supports multithreaded programming, which allows you to write programs that do many things simultaneously.

Architecture - Neutral

- Operating system upgrades, processor upgrades, and changes in core system resources can all combine to make a program malfunction.
- The Java designers made several hard decisions in the Java language and the Java Virtual Machine in an attempt to alter this situation. Their goal was "write once; run anywhere, any time, forever."

Distributed

- Java is designed for the distributed environment of the Internet because it handles TCP/IP protocols.
- Java also supports *Remote Method Invocation (RMI). This feature enables a program to* invoke methods across a network.

Interpreted and High Performance

- Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode.
- This code can be executed on any system that implements the Java Virtual Machine.
- Java bytecode was carefully designed so that it would be easy to translate directly into native machine code for very high performance by using a just-in-time compiler.

Dynamic

- Java programs carry with them substantial amounts of run-time type information that is used to verify and resolve accesses to objects at run time.

3) Control Structures

Flow of Control

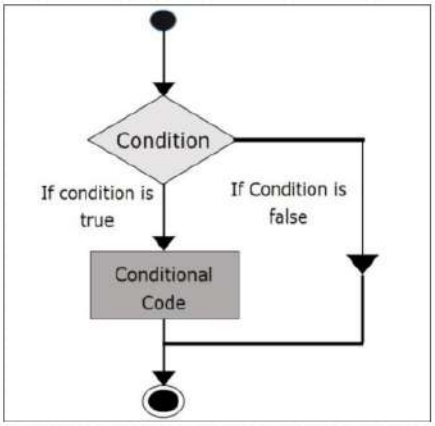
- Java executes one statement after the other in the order they are written
- Many Java statements are flow control statements:

Alternation	:	if, if else, Nested If, Else If, switch
Looping	:	while, do while, for
Escapes	:	break, continue, return

Simple If

The if statement evaluates an expression and if that evaluation is true then the specified action is taken

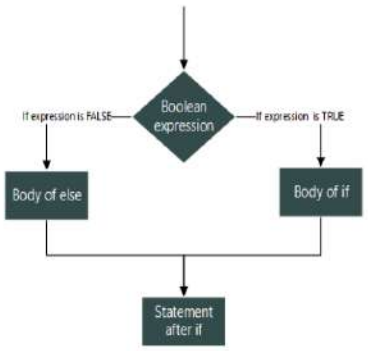
```
if ( x < 10 )
        x = 10;
```



If... else

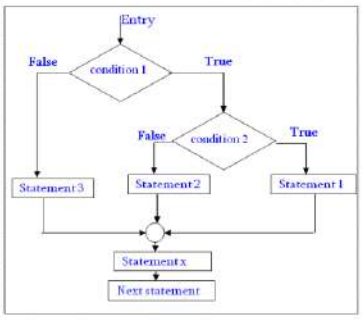
The if ... else statement evaluates an expression and performs one action if that evaluation is true or a different action if it is false.

```
if (x != oldx) {
        System.out.print("x was changed");
}
else {
        System.out.print("x is unchanged");
}
```



Nested if ... else

```
if ( myVal > 100 )
{
        if ( remainderOn == true)
 {
                myVal = mVal % 100;
        }
        else {
                myVal = myVal / 100.0;
        }
}
else
{
        System.out.print("myVal is in range");
}
```



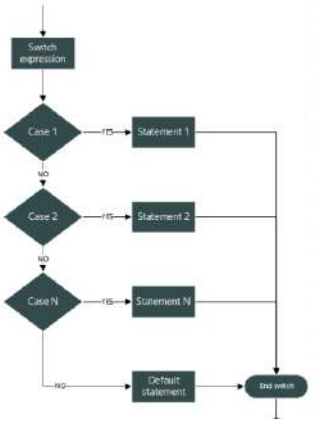
Else If Ladder

Useful for choosing between alternatives:

```
if ( n == 1 )
{
        // execute code block #1
}
else if ( j == 2 ) {
        // execute code block #2
}
else {
        // if all previous tests have failed,
execute code block #3
}
```

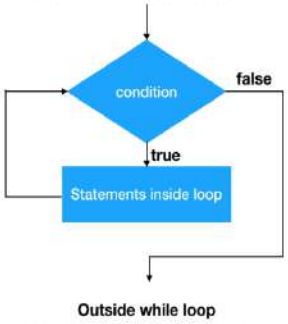
The switch Statement

```
switch ( n ) {
        case 1:
                // execute code block #1
                break;
        case 2:
                // execute code block #2
                break;
        default:
        // if all previous tests fail then
        //execute code block
                break;
}
```



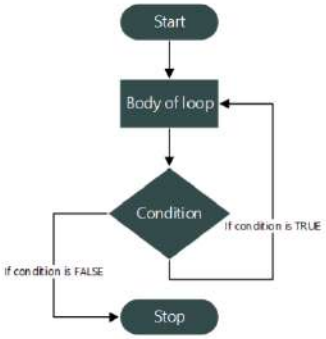
while loop

```
int i=0;
while(i<10)
{
        System.out.println(i);
        i++;
}
```



do {... } while loops

```
int i=0;
do
{
        System.out.println(i);
        i++;
} while(i<10);
```



The for loop

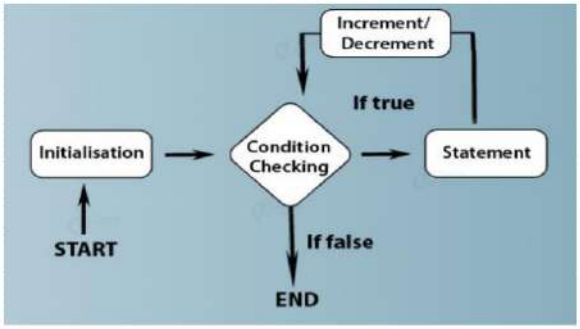
Loop n times

```
for ( i = 0; i < n; n++ ) {
        // this code body will execute n times
        // i from  0 to n-1
}
```

Nested for:

```
for ( j = 0; j < 10; j++ ) {
        for ( i = 0; i < 20; i++ ){
                // this code body will execute 200 times
        }
}
```


For loops

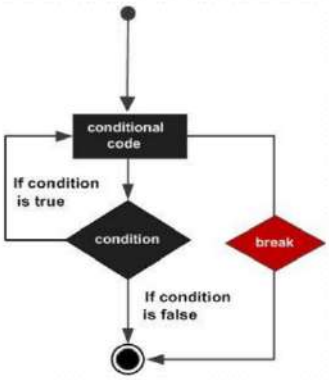


Break

- A break statement causes an exit from the innermost containing while, do, for or switch statement.

```
for ( int i = 0; i < 10, i++ ) {
if ( i == 3 )
{       break;
}
System.out.println(i);
}
```


Break

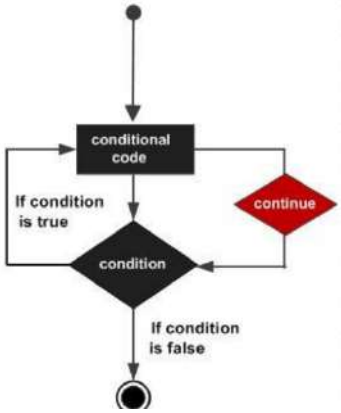


Continue

- Can only be used with while, do or for.
- The continue statement causes the innermost loop to start the next iteration immediately

```
for ( int i = 0; i < 10; i++ ) {
        if ( i==3 ) continue;
System.out.println(i);
}
```


Continue



Labeled break and continue

- Labeled block
  - Set of statements enclosed by {}
  - Preceded by a label
- Labeled break statement
  - Exit from nested control structures
  - Proceeds to end of specified labeled block
- Labeled continue statement
  - Skips remaining statements in nested-loop body
  - Proceeds to beginning of specified labeled block

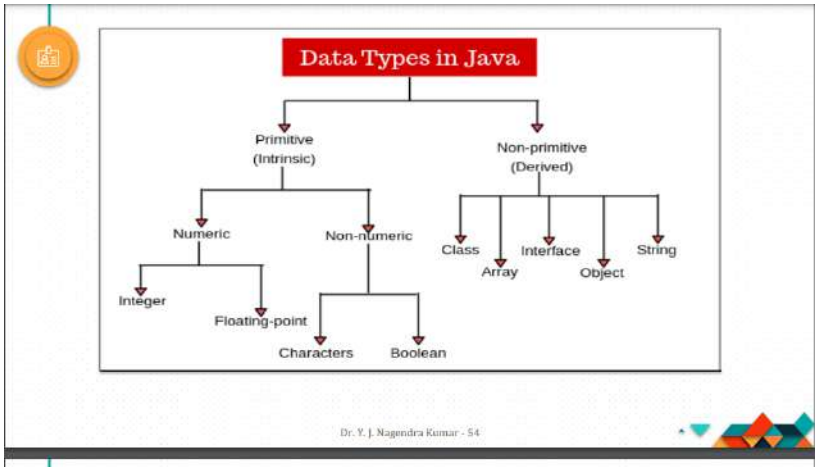
Labeled Continue example

```
public class LabelledContinue
{
 public static void main( String args[] )
 {
   stop:
          for ( int r = 1; r <= 10; r++ )
          {
          for ( int c = 1; c <= 5 ; c++ )
          {
                if ( r == 5 )
                continue stop;
              System.out.print(c);
            }
              System.out.println();
        }
      }
}
```

Labelled break example

```
public class LabelledContinue
{
 public static void main( String args[] )
 {
   stop:
          for ( int r = 1; r <= 10; r++ )
          {
          for ( int c = 1; c <= 5 ; c++ )
          {
                if ( r == 5 )
                break stop;
              System.out.print(c);
            }
              System.out.println();
        }
      }
}
```

4) Data types of Java



Data types

int	4 byte integer (whole number) range -2147483648 to +2147483648
float	4 byte floating point number decimal points, numbers outside range of int
double	8 byte floating point number 15 decimal digits (float has 7) so bigger precision and range
char	2 byte letter
String	string of letters
boolean	true or false (not 1 or 0)

Primitive Data Types

- A ***data type*** is defined by a set of values and the operators you can perform on them
- The Java language has several predefined types, called ***primitive data types***
- The following reserved words represent the eight different primitive data types:
  - **byte, short, int, long, float, double, boolean, char**

Dr. Y. J. Nagendra Kumar - 56

Integers

- There are four integer data types. They differ by the amount of memory used to store them

Type	Bits	Value Range
byte	8	-127 … 128
short	16	-32768 … 32767
int	32	about 9 decimal digits
long	64	about 18 decimal digits

Floating Point

There are two floating point types

Type	Bits	Range (decimal digits)	Precision (decimal digits)
float	32	38	7
double	64	308	15

Characters

- A **char** value stores a single character from the *Unicode character set*
- A *character set* is an ordered list of characters

  'A', 'B', 'C', ... , 'a', 'b', ... ,'0', '1', ... , '$', ...

Boolean

- A **boolean** value represents a true/false condition.
- It can also be used to represent any two states, such as a light bulb being on or off
- The reserved words **true** and **false** are the only valid values for a boolean type

Datatypes

Type	Size	Range	Default
boolean	1 bit	true or false	false
byte	8 bits	[-128, 127]	0
short	16 bits	[-32,768, 32,767]	0
char	16 bits	['\u0000', '\uffff'] or [0, 65535]	'\u0000'
int	32 bits	[-2,147,483,648 to 2,147,483,647]	0
long	64 bits	$[-2^{63}, 2^{63}-1]$	0
float	32 bits	32-bit IEEE 754 floating-point	0.0
double	64 bits	64-bit IEEE 754 floating-point	0.0

5) Constructor Overloading

Overloading Constructors

- In addition to overloading normal methods, we can also overload constructor methods.

Overloading Constructors Example

```
class rect
{    double l,b;
     rect()
     { l=10;b=20; }
     rect(int x,int y)
     { l=x; b=y; }
     rect(rect n)
     { l=n.l;
       b=n.b;  }
     rect(double x)
     { l=b=x;  }
     void area()
     { System.out.println("Area :"+l*b);      }
}
```

```
class rectarea
{    public static void main(String ar[])
     {
     rect r1=new rect();
     rect r2=new rect(22,33);
     rect r3=new rect(r2);
     rect r4=new rect(55.6);

     r1.area();
     r2.area();
     r3.area();
     r4.area();
     }
}
```

6) Classes and Objects

Classes

A class is a *template for an object, and an object is an instance of a class.*

```
class classname
{
        type instance-variable1;
        type instance-variable2;
        // ...
        type instance-variableN;

type methodname1(parameter-list) {
// body of method
}
// ...
type methodnameN(parameter-list) {
// body of method
}
}
```


Classes contd.,

- The data, or variables, defined within a class are called ***instance variables.***
- *The code is* contained within ***methods.***
- *Collectively, the methods and variables defined within a class are* called ***members of the class.***
- Variables defined within a class are called instance variables because each instance of the Class contains its own copy of these variables.

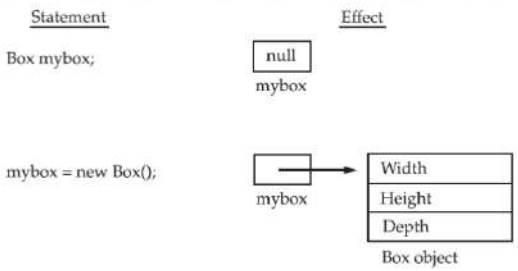
Declaring Objects

- Objects of a class is a two-step process.
- First, we must declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can *refer to an object.*
- *Second, we must acquire an actual,* physical copy of the object and assign it to that variable. We can do this using the **new operator.**
- The new operator dynamically allocates (that is, allocates at run time) memory for an object

Declaring Objects Contd.,

- Box mybox; // declare reference to object
- After this line executes, mybox contains the value null.
- mybox = new Box(); // allocate a Box object
- This line allocates an actual object and assigns a reference to it to mybox.
- We can combine the above statements into a singe one as follows:
- Box mybox = new Box();

Declaring Objects Contd.,



A class is a logical construct. An object has physical reality.

Declaring Objects Contd.,

```
class Box
{
    double width;
    double height;
    double depth;
}
class BoxDemo
{   public static void main(String args[])
    {       Box mybox = new Box();
            double vol;
            mybox.width = 10;
            mybox.height = 20;
            mybox.depth = 15;

        vol = mybox.width * mybox.height *
        mybox.depth;

        System.out.println("Volume is " + vol);
        }
}
```

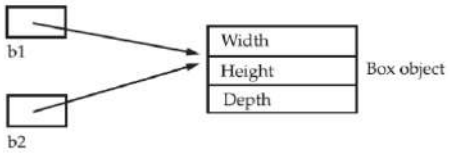
Assigning Object Reference Variables

Box b1 = new Box();

Box b2 = b1;

- b1 and b2 will both refer to the *same object.*
- *The assignment of b1 to b2 did not allocate any memory.*
- It simply makes b2 refer to the same object as does b1.
- Thus, any changes made to the object through b2 will affect the object to which b1 is referring

Assigning Object Reference Variables contd.,




```
Box b1 = new Box();
Box b2 = b1;
// ...
b1 = null;
```

Here, **b1 has been set to null, but b2 still points to the original object.**

7) Method Overloading and Method Overriding

Method Overloading

- In Java, it is possible to create methods that have the same name, but different parameter lists and different definitions. This is called *"Method Overloading".*
- Method Overloading is used when objects are required to perform similar tasks but using different input parameters. This process is known as ***Polymorphism.***

Method Overloading Example

```
class A
{    int i,j;
     A(int a, int b)
     {        i=a;j=b;     }
     void display()
     {
     System.out.println(" i and j "+i+" "+j);
     }
}
class B extends A
{    int k;
     B(int a,int b,int c)
     {        super(a,b);
              k=c;          }
     void display(String s)
     {        System.out.println(s+k);
     }
}
class overload
{
     public static void main(String ar[])
     {
              B sub=new B(10,20,30);

              sub.display(" this is :");
              sub.display();
     }
}
```

Method Overriding

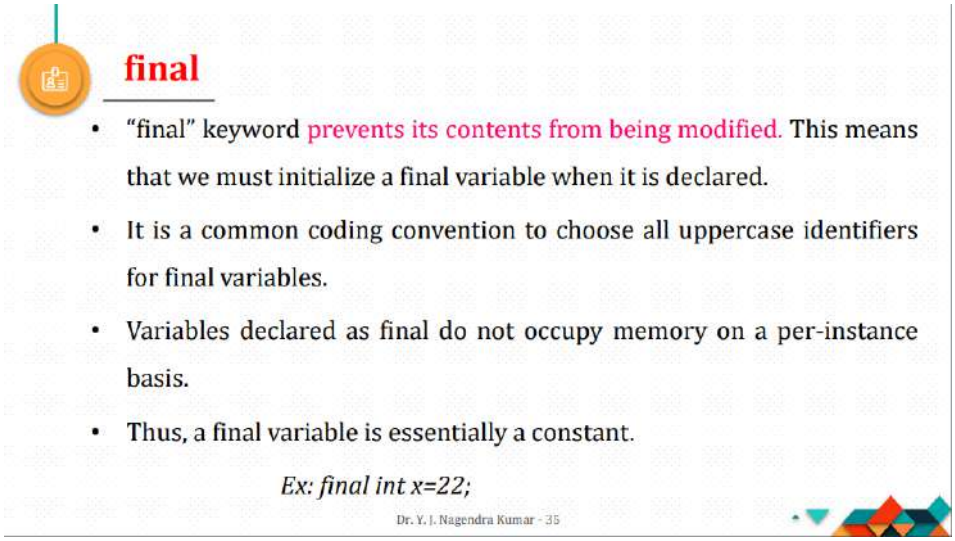
- In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to ***override*** the method in the superclass.
- When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden.

Method Overriding Example

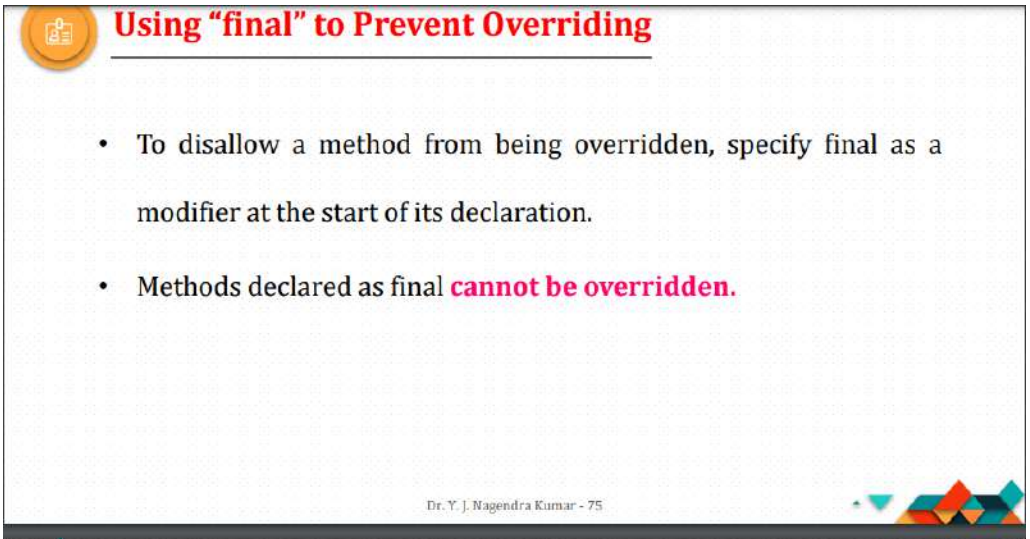
```
class A
{    int i,j;
     A(int a, int b)
     {        i=a;j=b;     }
     void display()
     {
     System.out.println(" i and j "+i+" "+j);
     }
}
class B extends A
{    int k;
     B(int a,int b,int c)
     {        super(a,b);
              k=c;          }
     void display()
     {
              System.out.println("k: " + k);
     }
}
class Override
{
     public static void main(String args[])
     {
              B subOb = new B(1, 2, 3);
     subOb.display();  // this calls show() in B
     }
}
```

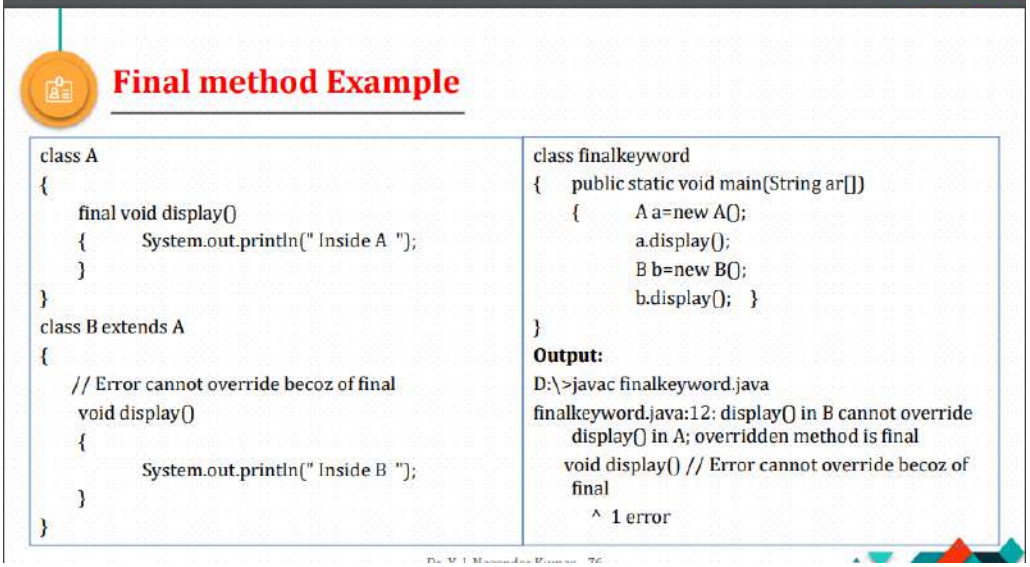
8) Final Keyword – 3 Forms

a. Final Variable



b. Final Method





c. Final Class

Using final to Prevent Inheritance

- Sometimes we will want to **prevent a class from being inherited.** To do this, precede the class declaration with final.
- Declaring a class as final implicitly declares all of its methods as final, too.
- It is illegal to declare a class as both abstract and final since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations.

Final Class Example

```
final class A
{
    void display()
    {   System.out.println(" Inside A  ");
    }
}
class B extends A
// Error cannot Inherit becoz of final
{
    void display()
    {
    System.out.println(" Inside B  ");
    }
}
```

```
class finalclass
{   public static void main(String ar[])
    {       A a=new A();
            a.display();
            B b=new B();
            b.display();
    }
}
```

Output:

```
D:\>javac finalclass.java
finalclass.java:10: cannot inherit from final A
class B extends A // Error cannot Inherit becoz of final
          ^  1 error
```

9) Super Keyword – 2 Forms

'super' forms

- super has two general forms.
  - The first calls the superclass' constructor.
  - The second is used to access a member of the superclass

Using super to Call Superclass Constructors

- A subclass can call a constructor method defined by its superclass by using the following form of super:

  super(*parameter-list*);

- Here, *parameter-list* specifies any parameters needed by the constructor in the superclass.
- super( ) must always be the first statement executed inside a subclass' constructor.

Super First form Example

```
class rect
{     double length;
      double breadth;
      rect()
      { length=-1; breadth=-1;          }
      rect(double l,double b)
      {
                  length=l; breadth=b;
      }
      double area()
      {
                  return length*breadth;
      }
}
```

```
class box extends rect
{
    double height;
       box()
      {           super();
                  height=-1;      }
      box(double l,double b,double h)
      {           super(l,b);
                  height=h;       }
      void volume()
      {
System.out.println("Volume : "+length*breadth*height);
      }
}
```


Super First form Example Contd.,

```
class inhert2
{
      public static void main(String ar[])
      {
                  rect r=new rect(20,10);
                  System.out.println("Area : "+r.area());

                  box b=new box(30,40,5);
                  b.volume();
      }
}
```

A Second Use for super

- The second form of super acts like this, except that it always refers to the superclass of the subclass in which it is used.

  Syn: super.*member*

- Here, *member* can be either a method or an instance variable.
- This second form of super is most applicable to situations in which member names of a subclass hide members by the same name in the superclass.

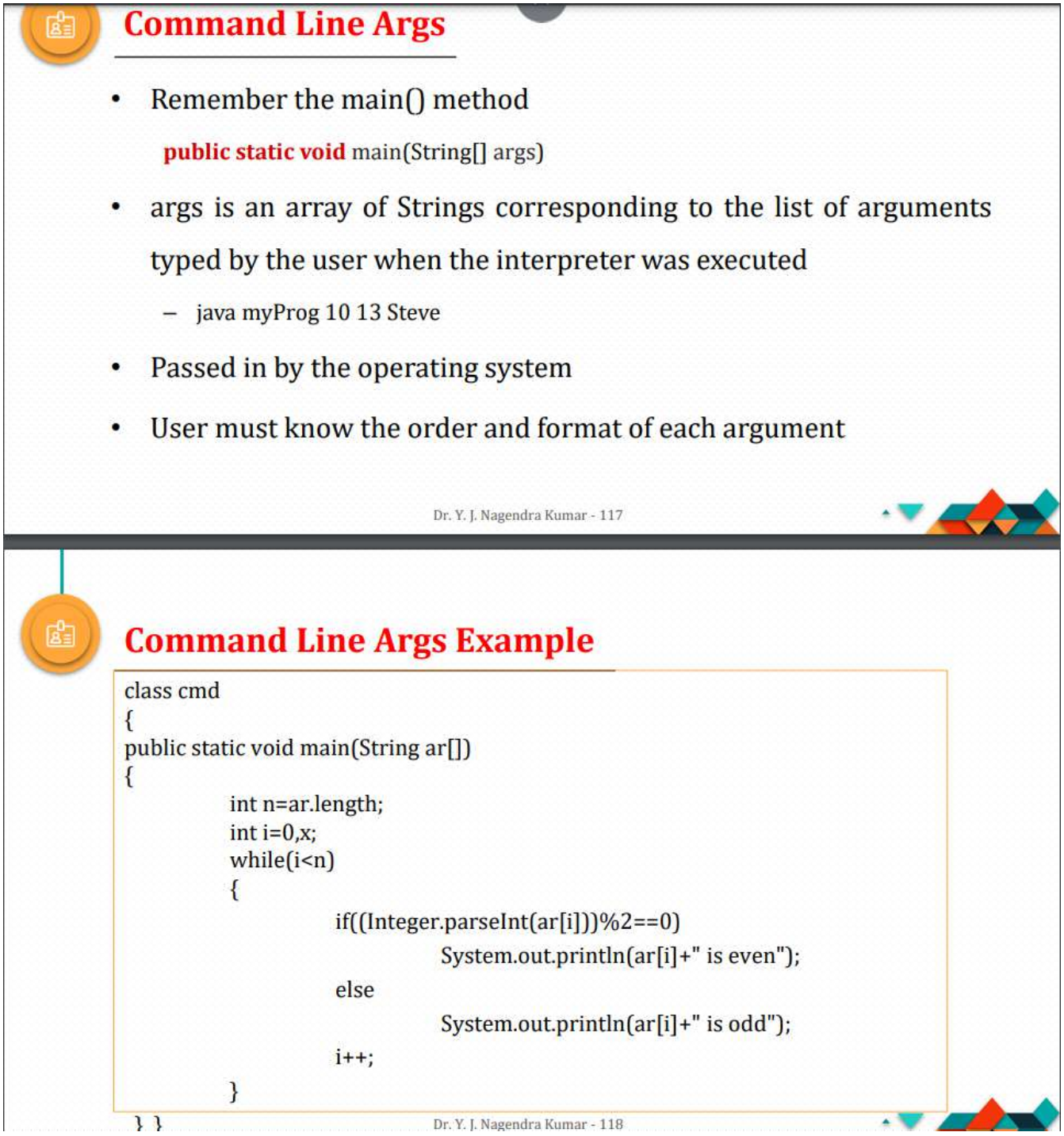
Super Second form Example

```
class A
{
    int i;
}

class B extends A
{
    int i;

    B(int a,int b)
    {
        super.i=a;
        i=b;
    }
void display()
    {
        System.out.println(" Super i : "+super.i);
        System.out.println(" Sub i : "+i);
    }
}
class inhert3
{
    public static void main(String ar[])
    {
        B sub=new B(20,30);
        sub.display();
    }
}
```

10) Command Line Arguments



Command Line Args

- Remember the main() method

  public static void main(String[] args)

- args is an array of Strings corresponding to the list of arguments typed by the user when the interpreter was executed
  - java myProg 10 13 Steve
- Passed in by the operating system
- User must know the order and format of each argument

Dr. Y. J. Nagendra Kumar - 117

Command Line Args Example

```
class cmd
{
public static void main(String ar[])
{
        int n=ar.length;
        int i=0,x;
        while(i<n)
        {
                if((Integer.parseInt(ar[i]))%2==0)
                        System.out.println(ar[i]+" is even");
                else
                        System.out.println(ar[i]+" is odd");
                i++;
        }
} }
```

Dr. Y. J. Nagendra Kumar - 118

11) Interfaces (Multiple Inheritance)

Interfaces

- Java does not support multiple inheritance. That is, classes in java cannot have more than one superclass.
- Java provides an alternate approach known as interfaces to support the concept of multiple inheritance.
- Interfaces are syntactically similar to classes, but they define only abstract methods and final fields.
- This means that interfaces do not specify any code to implement these methods and data fields contain only constants.
- Once it is defined, any number of classes can implement an interface. Also, one class can implement any number of interfaces.

Defining an Interface

- An interface is defined much like a class. This is the general form of an interface:

```
access interface name
{
type final-varname1 = value;
type final-varname2 = value;
// ...
type final-varnameN = value;
return-type method-name1(parameter-list);
return-type method-name2(parameter-list);
// ...
return-type method-nameN(parameter-list);
}
```

Here, *access* is either **public** or not used

Interfaces

- Methods are, essentially, abstract methods
- Each class that includes an interface must implement all of the methods.
- Variables can be declared inside of interface declarations. They are implicitly **final** and **static**, meaning they cannot be changed by the implementing class.
- All methods and variables are implicitly **public** if the interface, itself, is declared as **public**.

Ex:

```
interface shape
{
void area(int param);
}
```


Implementing Interfaces

- Once an **interface** has been defined, one or more classes can implement that interface.
- To implement an interface, include the **implements** clause in a class definition, and then create the methods defined by the interface.

Syntax:

```
access class classname [extends superclass]
[implements interface [,interface...]]
{
        // class-body
}
```

Interfaces contd.,

- Here, *access* is either **public** or not used. If a class implements more than one interface, the interfaces are separated with a comma.
- The methods that implement an interface must be declared **public**.
- Also, the type signature of the implementing method must match exactly the type signature specified in the **interface** definition.

Interfaces

```
class circle implements shape
{
        // Implement shape's interface
        public void area(int p)
        {
        System.out.println("Area of Circle "+3.14*p*p);
        }
}
```

Interfaces Second Example

```
class student
{
        int rollno;

        void getno(int a)
        {
                rollno=a;
        }
        void putno()
        {
System.out.println(" Roll Number : "+rollno);
        }
}
```

```
class test extends student
{
    double m1,m2;

    void getmarks(double a,double b)
    {
        m1=a; m2=b;
    }
    void putmarks()
    {
    System.out.println("M1 : "+m1+" M2 : "+m2);
    }
}
```


Interfaces Second Example contd.,

```
interface sports
{
    final static double spwt=10;
    void putspwt();
}
class result extends test implements sports
{
    double total;
    public void putspwt()
    {
    System.out.println("Sports Marks Weightage
    : "+spwt);
    }
    void show()
    {    total=m1+m2+spwt;
        putno();
        putmarks();
        putspwt();
    System.out.println("Total Marks :"+total);
    }
}
class interface2
{   public static void main(String ar[])
    {    result r=new result(); r.getno(786);
        r.getmarks(78.5,65.25);  r.show();
    }
}
```