II B.Tech I Sem

Narsimha Reddy Engineering College

Unit II

CLASSES, INHERITANCE, POLYMORPHISM Classes and Objects- Classes, Objects, creating objects, methods, constructors- constructor overloading, cleaning up unused objects- Garbage collector, class variable and methods- static keyword, this keyword, arrays, Command line arguments, Nested Classes Strings: String, StringBuffer,StringTokenizerInheritance and Polymorphism- Types of Inheritance, deriving classes using extends keyword, super keyword, Polymorphism – Method Overloading, Method Overriding, final keyword, abstract classes.

CLASSES

- **What is a class in Java**
- A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.
- A class in Java can contain:
- **Fields**
- **Methods**
- **Constructors**
- **Blocks**
- **Nested class and interface**

Object and Class Example: main within the class

```
//Java Program to illustrate how to define a class and fields
//Defining a Student class.
class Student{
 //defining fields
 int id;//field or data member or instance variable
 String name;
 //creating main method inside the Student class
 public static void main(String args[]){
  //Creating an object or instance
  Student s1=new Student();//creating an object of Student
  //Printing values of the object
  System.out.println(s1.id);//accessing member through reference variable
  System.out.println(s1.name);
 }
}
```

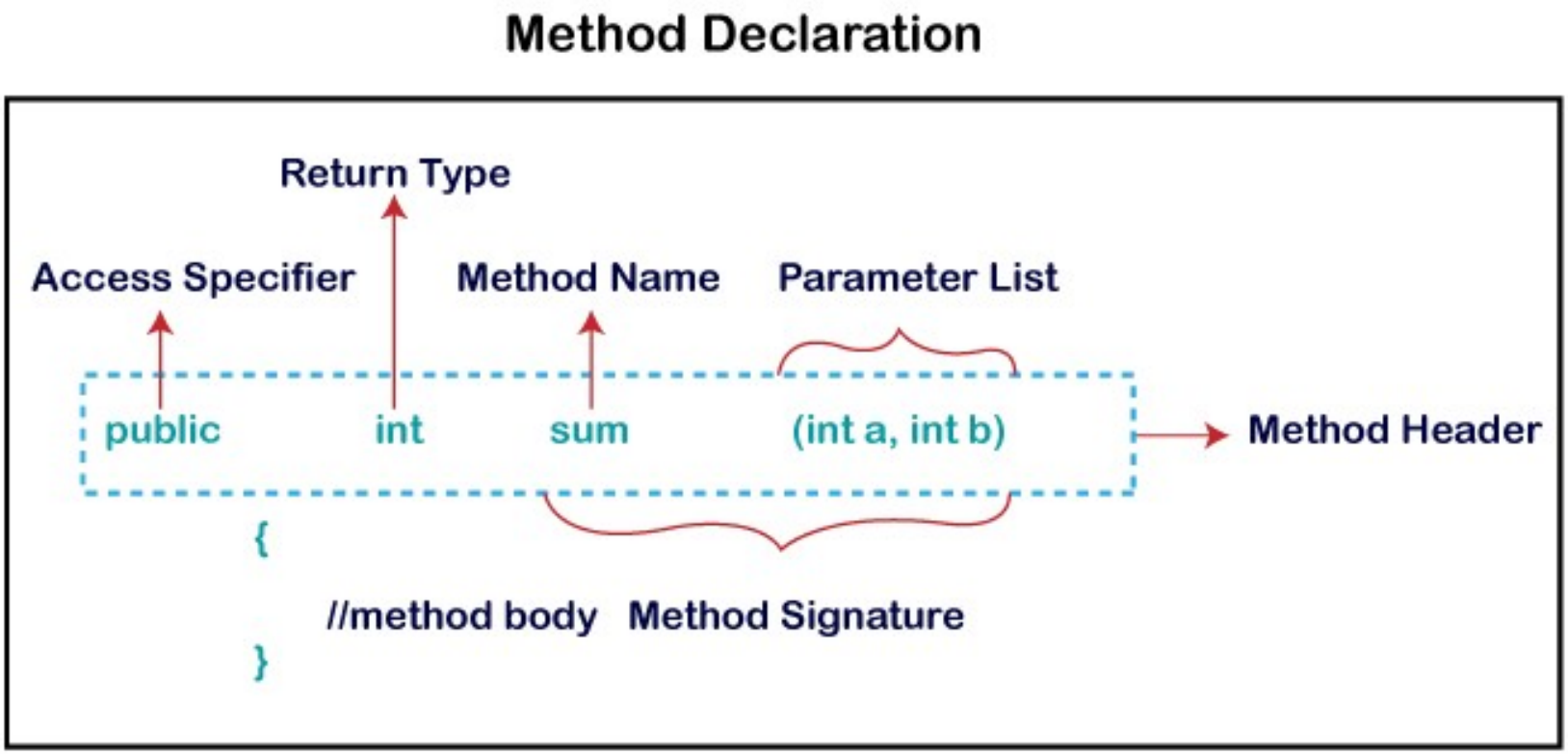
Object and Class Example: main outside the class

- In real time development, we create classes and use it from another class. It is a better approach than previous one.
- We can have multiple classes in different Java files or single Java file.
- If you define multiple classes in a single Java source file, it is a good idea to save the file name with the class name which has main() method.

```java
//Java Program to demonstrate having the main method in
//another class
//Creating Student class.
class Student{
 int id;
 String name;
}
//Creating another class TestStudent1 which contains the main method
class TestStudent1{
 public static void main(String args[]){
  Student s1=new Student();
  System.out.println(s1.id);
  System.out.println(s1.name);
 }
}
```

Method in Java

method in Java is a collection of instructions that performs a specific task. It provides the reusability of code.



- **Access Specifier:** Access specifier or modifier is the access type of the method. It specifies the visibility of the method. Java provides **four** types of access specifier:
- **Public:** The method is accessible by all classes when we use public specifier in our application.
- **Private:** When we use a private access specifier, the method is accessible only in the classes in which it is defined.
- **Protected:** When we use protected access specifier, the method is accessible within the same package or subclasses in a different package.
- **Default:** When we do not use any access specifier in the method declaration, Java uses default access specifier by default. It is visible only from the same package only.

- **Return Type:** Return type is a data type that the method returns. It may have a primitive data type, object, collection, void, etc. If the method does not return anything, we use void keyword.
- **Method Name:** It is a unique name that is used to define the name of a method. It must be corresponding to the functionality of the method. Suppose, if we are creating a method for subtraction of two numbers, the method name must be **subtraction().** A method is invoked by its name.
- **Parameter List:** It is the list of parameters separated by a comma and enclosed in the pair of parentheses. It contains the data type and variable name. If the method has no parameter, left the parentheses blank.
- **Method Body:** It is a part of the method declaration. It contains all the actions to be performed. It is enclosed within the pair of curly braces.

Method Overloading in Java

- If a class having multiple methods with same name but different in parameters, it is known as **Method Overloading**.
- If we have to perform only one operation, having same name of the methods increases the readability of the program
- There are three ways to overload the method in java
- By changing number of arguments
- By changing the data type
- By changing the order

1) Method Overloading: changing no. of arguments

```
class Adder{
static int add(int a,int b){return a+b;}
static int add(int a,int b,int c){return a+b+c;}
}
class TestOverloading1{
public static void main(String[] args){
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(11,11,11));
}}
```

Method Overloading: changing data type of arguments

```
class Adder{
static int add(int a, int b){return a+b;}
static double add(double a, double b){return a+b;}
}
class TestOverloading2{
public static void main(String[] args){
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(12.3,12.6));
}}
```

Java Variables

- A variable is a container which holds the value while the Java program is executed. A variable is assigned with a data type.
- Variable is a name of memory location. There are three types of variables in java: local, instance and static.

Types of Variables

- There are three types of variables in Java:
- local variable
- instance variable
- static variable

1) Local Variable

A variable declared inside the body of the method is called local variable. You can use this variable only within that method and the other methods in the class aren't even aware that the variable exists.

A local variable cannot be defined with "static" keyword.

2) Instance Variable

A variable declared inside the class but outside the body of the method, is called an instance variable. It is not declared as static.

It is called an instance variable because its value is instance-specific and is not shared among instances.

3) Static variable

A variable that is declared as static is called a static variable. It cannot be local. You can create a single copy of the static variable and share it among all the instances of the class. Memory allocation for static variables happens only once when the class is loaded in the memory.

```
public class A {
    static int m = 100; // Static variable

    int data; // This is an instance variable

    void method() {
        int n = 90; // Local variable
    }

    public static void main(String args[]) {
        int data = 50; // Local variable
    }
}
```

Java static keyword

- The static keyword belongs to the class than an instance of the class.
- The static can be:
- Variable (also known as a class variable)
- Method (also known as a class method)
- Block
- Nested class

1) Java static variable

- If you declare any variable as static, it is known as a static variable.
- The static variable can be used to refer to the common property of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc.
- The static variable gets memory only once in the class area at the time of class loading.

```java
//Java Program to demonstrate the use of static variable
class Student{
   int rollno;//instance variable
   String name;
   static String college ="ITS";//static variable
   //constructor
   Student(int r, String n){
   rollno = r;
   name = n;
   }
   //method to display the values
   void display (){System.out.println(rollno+" "+name+" "+college);}
}
//Test class to show the values of objects
public class TestStaticVariable1{
 public static void main(String args[]){
 Student s1 = new Student(111,"Karan");
 Student s2 = new Student(222,"Aryan");
 //we can change the college of all objects by the single line of code
 //Student.college="BBDIT";
 s1.display();
 s2.display();
 }
}
```

2) Java static method

- if you apply static keyword with any method, it is known as static method.
- A static method belongs to the class rather than the object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- A static method can access static data member and can change the value of it.

Restrictions for the static method

- There are two main restrictions for the static method. They are:
- The static method can not use non static data member or call non-static method directly.
- this and super cannot be used in static context.

Why is the Java main method static?

Ans) It is because the object is not required to call a static method. If it were a non-static method, JVM creates an object first then call main() method that will lead the problem of extra memory allocation.

3) Java static block

- Is used to initialize the static data member.
- It is executed before the main method at the time of class loading.

```
class A2{
  static
{
System.out.println("static block is invoked");}
  public static void main(String
  args[]){ System.out.println("Hello main");
  }
}
```

What is a Constructor?

- Constructor is a special method that gets invoked "automatically" at the time of object creation.
- Constructor is normally used for initializing objects with default values unless different values are supplied.
- Constructor has the same name as the class name.
- Constructor cannot return values.
- A class can have more than one constructor as long as they have different signature (i.e., different input arguments syntax).

Defining a Constructor

- Like any other method

```
public class ClassName {

    // Data Fields…

    // Constructor
    public ClassName()
    {
        // Method Body Statements initialising Data Fields
    }

    //Methods to manipulate data fields
}
```

- Invoking:
  - There is NO explicit invocation statement needed: When the object creation statement is executed, the constructor method will be executed automatically.

Defining a Constructor: Example

```
public class Counter
    { int
    CounterIndex;

    // Constructor
    public Counter()
    {
        CounterIndex = 0;
    }
    //Methods to update or access counter
    public void increase()
    {
        CounterIndex = CounterIndex + 1;
    }
    public void decrease()
    {
        CounterIndex = CounterIndex - 1;
    }
    int getCounterIndex()
    {
      return CounterIndex;
    }
}
```

Trace counter value at each statement and What is the output ?

```
class MyClass {
 public static void main(String args[])
 {      Counter counter1 = new Counter();
     counter1.increase();
     int a = counter1.getCounterIndex();
     counter1.increase();
     int b = counter1.getCounterIndex();
     if ( a > b )
          counter1.increase();
     else
          counter1.decrease();
     System.out.println(counter1.getCounterIndex());
} }
```

A Counter with User Supplied Initial Value ?

- This can be done by adding another constructor method to the class.

```
public class Counter {
    int CounterIndex;

    // Constructor 1
    public Counter()
    {
        CounterIndex = 0;
    }
    public Counter(int InitValue )
     {
          CounterIndex = InitValue;
     }
}

// A New User Class: Utilising both constructors
  Counter counter1 = new Counter();
    Counter counter2 = new Counter (10);
```

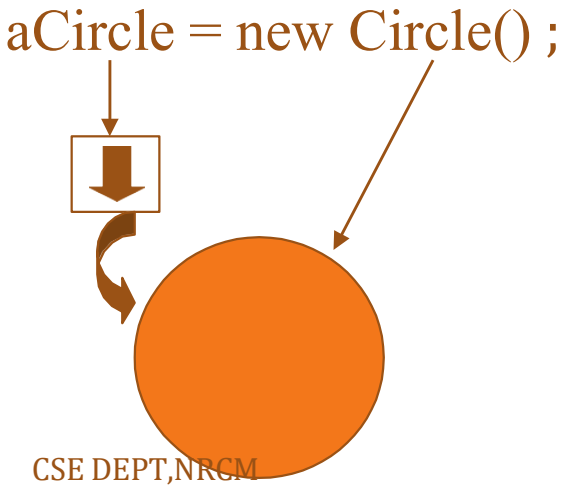
Adding a Multiple-Parameters Constructor to our Circle Class

```
public class Circle {
      public double x,y,r;
     // Constructor
     public Circle(double centreX, double centreY,
                                                    double radius)
     {
            x = centreX;
            y = centreY;
            r = radius;
      }
     //Methods to return circumference and area
      public double circumference() { return 2*3.14*r; }
      public double area() { return 3.14 * r * r; }
}
```

Constructors initialise Objects

- Recall the following OLD Code Segment:

```
Circle aCircle = new Circle();
aCircle.x = 10.0; // initialize center and radius
aCircle.y = 20.0
aCircle.r = 5.0;
```



At creation time the center and radius are not defined.

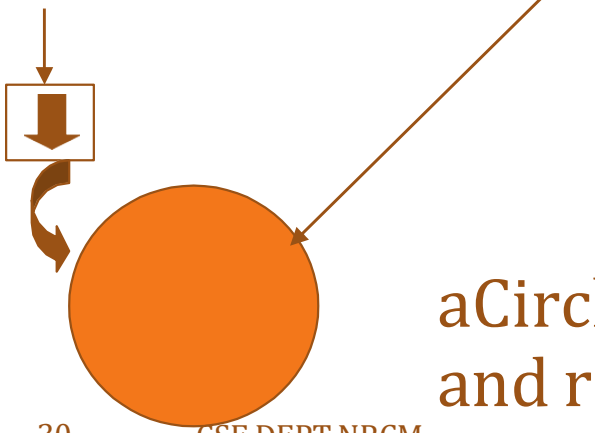
These values are explicitly set later.

Constructors initialise Objects

- With defined constructor

```
Circle aCircle = new Circle(10.0, 20.0, 5.0);
```

aCircle = new Circle(10.0, 20.0, 5.0) ;



aCircle is created with center (10, 20) and radius 5

Multiple Constructors

- Sometimes want to initialize in a number of different ways, depending on circumstance.

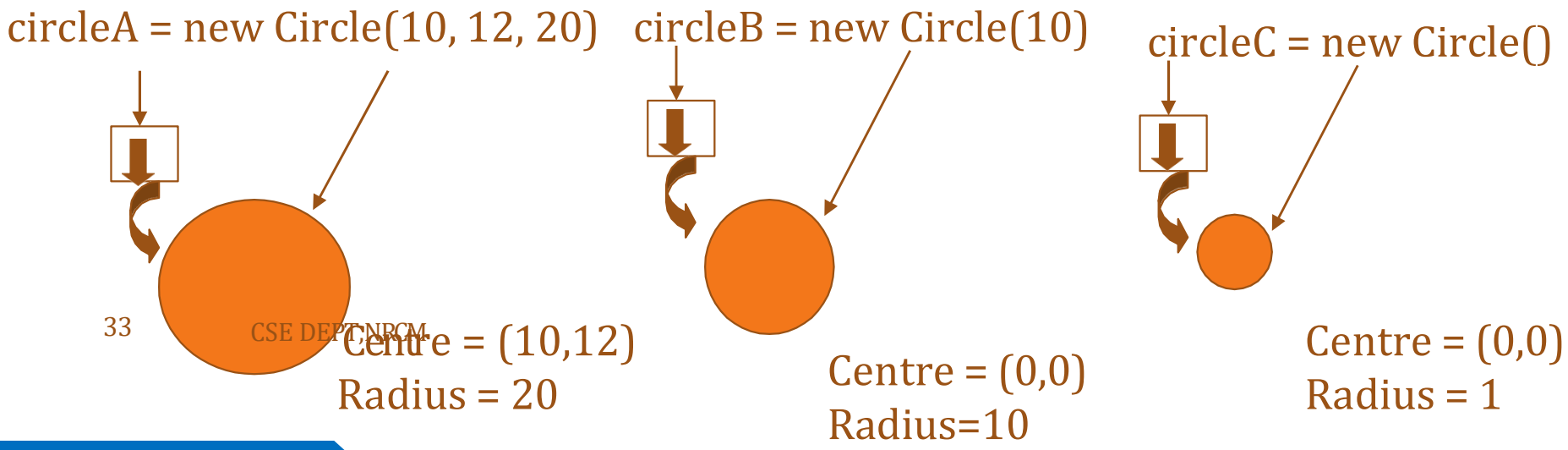
- This can be supported by having multiple constructors having different input arguments.

Multiple Constructors

```
public class Circle {
      public double x,y,r; //instance variables
      // Constructors
      public Circle(double centreX, double cenreY, double radius) {
            x = centreX;  y = centreY; r = radius;
      }
      public Circle(double radius) { x=0; y=0; r = radius; }
      public Circle() { x=0; y=0; r=1.0; }

       //Methods to return circumference and area
      public double circumference() { return 2*3.14*r; }
      public double area() { return 3.14 * r * r; }
}
```

Initializing with constructors

```
public class TestCircles {

        public static void main(String args[]){
            Circle circleA = new Circle( 10.0, 12.0, 20.0);
            Circle circleB = new Circle(10.0);
            Circle circleC = new Circle();
        }
}
```



Method Overloading

- Constructors all have the same name.
- Methods are distinguished by their signature:
  - name
  - number of arguments
  - type of arguments
  - position of arguments
- That means, a class can also have multiple usual methods with the same name.
- Not to confuse with *method overriding* (coming up), *method overloading:*

Polymorphism

- Polymorphism in Java is a concept by which we can perform a *single action in different ways*. Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.

- There are two types of polymorphism in Java: compile-time polymorphism and runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding.

Polymorphism

- Allows a single *method or operator* associated with different meaning depending on the type of data passed to it. It can be realised through:
  - Method Overloading
  - Operator Overloading (Supported in C++, but not in Java)
- Defining the same *method* with different argument types (method overloading) - polymorphism.
- The method body can have different logic depending on the date type of arguments.

Scenario

- A Program needs to find a maximum of two numbers or Strings. Write a separate function for each operation.
  - In C:
    - int max_int(int a, int b)
    - int max_string(char *s1, char *s2)
    - max_int (10, 5) or max_string (“melbourne”, “sydney”)
  - In Java:
    - int max(int a, int b)
    - int max(String s1, String s2)
    - max(10, 5) or max(“melbourne”, “sydney”)
  - Which is better ? Readability and intuitive wise ?

A Program with Method Overloading

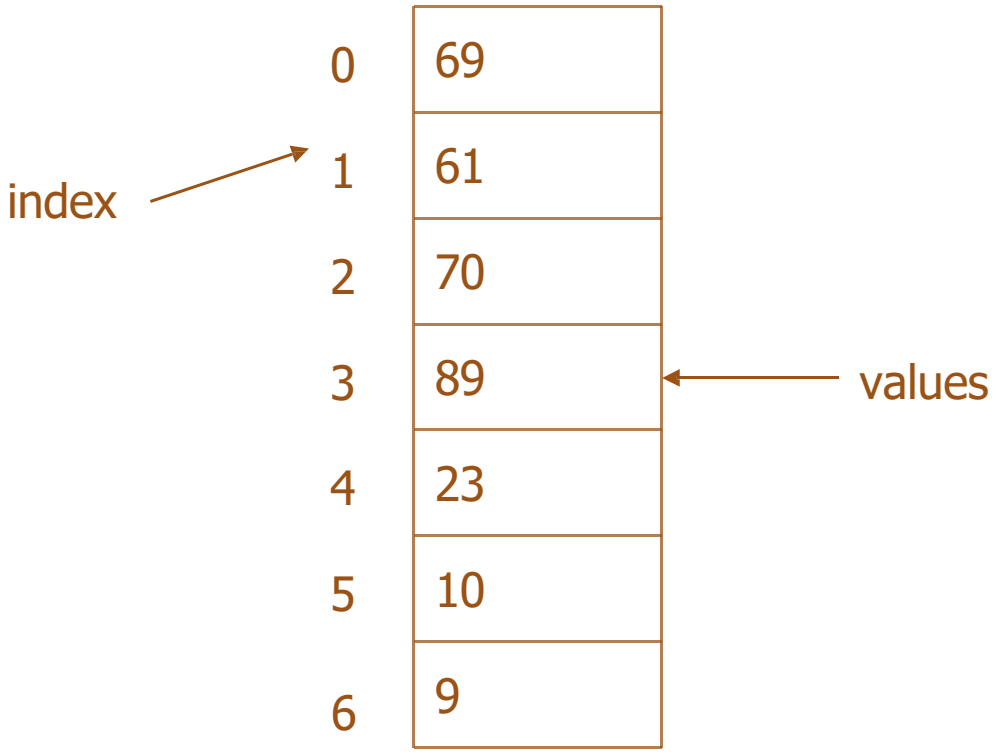
```
// Compare.java: a class comparing
   different items
class Compare {
  static int max(int a, int b)
  {
     if( a > b)
          return a;
     else
     return b;
  }
  static String max(String a, String b)
  {
     if( a.compareTo (b) > 0)
          return a;
     else
          return b;
}
```

```
public static void main(String args[])
{
     String s1 = "Melbourne";
     String s2 = "Sydney";
     String s3  = "Adelaide";

     int a = 10;
     int b = 20;

     System.out.println(max(a, b)); //
 which number is big
     System.out.println(max(s1, s2));
 // which city is big
     System.out.println(max(s1, s3));
 // which city is big
}}
```

Arrays - Introduction

- An array is a group of contiguous or related data items that share a common name.
- Used when programs have to handle large amount of data
- Each value is stored at a specific position
- Position is called a index or superscript. Base index = 0
- A loop with index as the control variable can be used to read the entire array, perform calculations, and print out the results.

Arrays - Introduction



- Like any other variables, arrays must declared and created before they can be used. Creation of arrays involve three steps:
  - Declare the array
  - Create storage area in primary memory.
  - Put values into the array (i.e., Memory location)
- Declaration of Arrays:
  - Form 1:

    Type arrayname[]
  - Form 2:
    - Type [] arrayname;
  - Examples:

    int[] students;

    int students[];
  - Note: we don't specify the size of arrays in the declaration.

Creation of Arrays

- After declaring arrays, we need to allocate memory for storage array items.
- In Java, this is carried out by using “new” operator, as follows:
  - Arrayname = **new** type[size];
- Examples:
  - students = new int[7];
  - Type arrayname[]=**new** type[size];

Initialisation of Arrays

- Once arrays are created, they need to be initialised with some values before access their content. A general form of initialisation is:
  - Arrayname [index/subscript] = value;
- Example:
  - students[0] = 50;
  - students[1] = 40;
- Like C, Java creates arrays starting with subscript 0 and ends with value one less than the size specified.

Arrays – Initializing at Declaration

- Arrays can also be initialised like standard variables at the time of their declaration.
  - Type arrayname[] = {list of values};
- Example:

  int students [] = {55, 69, 70, 30, 80};
- Creates and initializes the array of integers of length 5.
- In this case it is not necessary to use the *new* operator.

Arrays – Length

- Arrays are fixed length
- Length is specified at create time
- We can access the length of arrays as arrayName.length:

  e.g. int x = students.length; // x = 7
- Accessed using the index

  e.g. int x = students [1]; // x = 40

Arrays – Example

```
// StudentArray.java: store integers in arrays and access

public class StudentArray{

    public static void main(String[] args) {

        int[] students;

        students = new int[7];

        System.out.println("Array Length = " + students.length);

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

            students[i] = 2*i;

        System.out.println("Values Stored in Array:");

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

            System.out.println(students[i]);
```



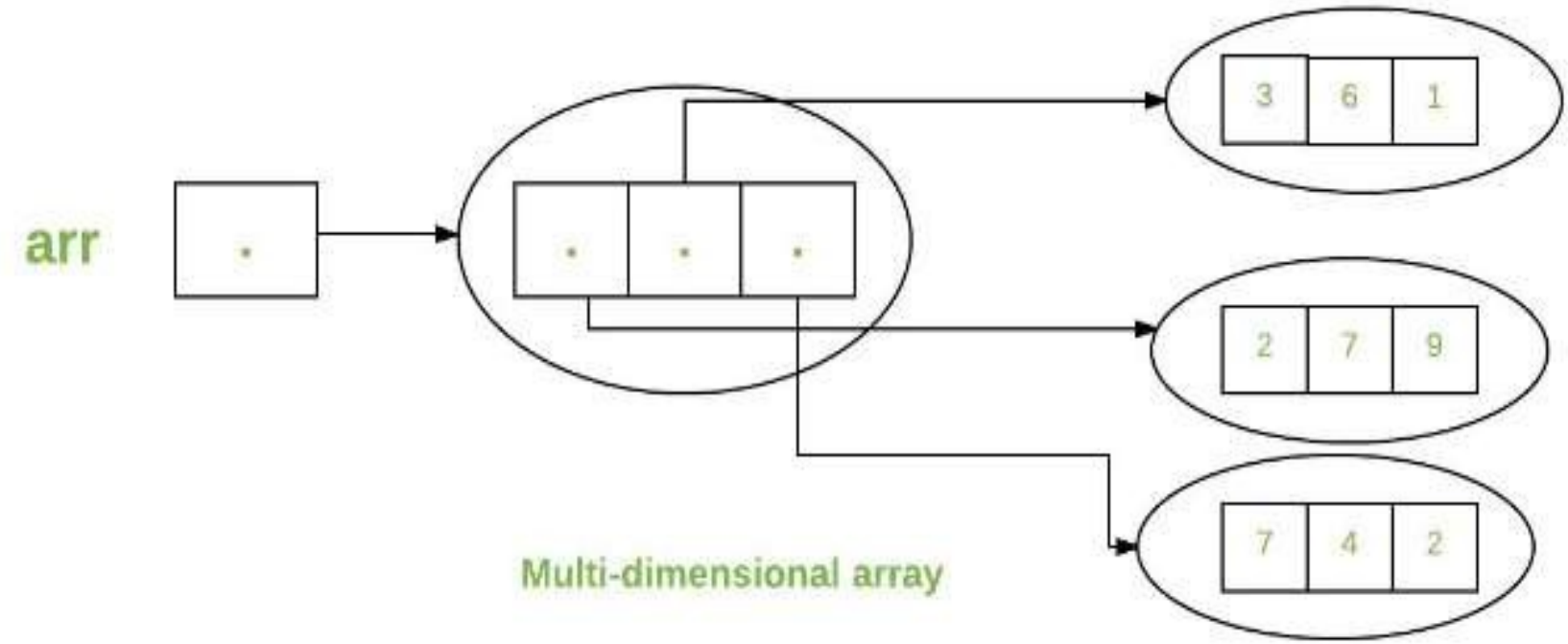
```
}}
```

```
// StudentArray.java: store integers in arrays and access
public class StudentArray{
        public static void main(String[] args) {
         int[] students = {55, 69, 70, 30, 80};

         System.out.println("Array Length = " + students.length);
        System.out.println("Values Stored in Array:");
         for ( int  i=0;  i < students.length;  i++)
                System.out.println(students[i]);
        }
}
```

Two Dimensional Arrays

- Two dimensional arrays allows us to store data that are recorded in table. For example:
- Table contains 12 items, we can think of this as a matrix consisting of 4 rows and 3 columns.

Marks / Person	Subject1	Subject2	Subject3
Student #1	10	15	30
Student #2	14	30	33
Studentl #3	200	32	1
Student#4	10	200	4

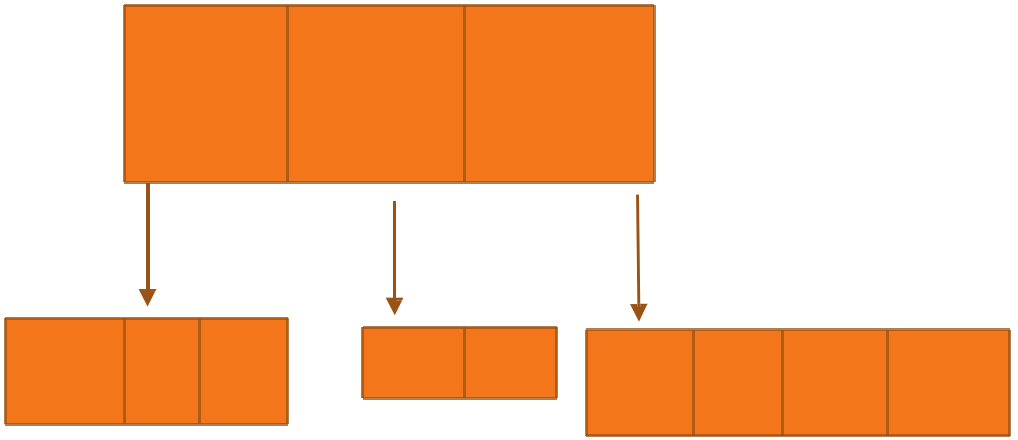

arr
3
6
1
2
7
9
7
4
2
Multi-dimensional array

2D arrays manipulations

- Declaration:
  - int myArray [][];
- Creation:
  - myArray = new int[4][3]; // OR
  - int myArray [][] = new int[4][3];
- Initialisation:
  - Single Value;
    - myArray[0][0] = 10;
  - Multiple values:
    - int tableA[][] = {{10, 15, 30}, {14, 30, 33}};

Variable Size Arrays

- Java treats multidimensional arrays as “arrays of arrays”. It is possible to declare a 2D arrays as follows:
  - int a[][] = new int [3][];
  - a[0]= new int [3];
  - a[1]= new int [2];
  - a[2]= new int [4];



Try: Write a program to Add to Matrix

- Define 2 dimensional matrix variables:
  - Say: int a[][], b[][];
- Define their size to be 2x3
- Initialise like some values
- Create a matrix c to storage sum value
  - c[0][0] = a[0][0] + b[0][0]
- Print the contents of result matrix.

Java Inner Classes / Nested Classes

- **Java inner class** or nested class is a class that is declared inside the class. The scope of a nested class is bounded by the scope of its enclosing class.
- Ex: if class B is defined within class A, then B is known to A, but not outside of A.
- A nested class has access to the members, including private members, of the class in which it is nested. However, the enclosing class does not have access to the members of the nested class.
- We use inner classes to logically group classes and interfaces in one place to be more readable and maintainable.

Syntax of Inner class

```
class Java_Outer_class{
 //code
 class Java_Inner_class{
  //code
 }
}
```

Static Nested Classes

- There are two types of nested classes:
  - static
  - non-static.
- A static nested class is one which has the static modifier applied. Because it is static, it must access the members of its enclosing class through an object.
- That is, it cannot refer to members of its enclosing class directly. Because of this restriction, static nested classes are seldom used.

Static Nested Classes Example

```
class Outer
{    int outer_x = 100;
     void test()
     {       Inner in=new Inner();
             in.display();
     }
     static class Inner
     {       void display()
             {   Outer out=new Outer();
     System.out.println("display: outer_x = " +
     out.outer_x);
             }
     }}
```

```
class StaticNestedDemo
{
     public static void main(String args[])
     {
             Outer outer = new Outer();
             outer.test();
     }
}
```

Non Static Nested Classes (Inner Classes)

- An inner class is a non-static nested class.
- It has access to all of the variables and methods of its outer class and may refer to them directly.

Inner Classes Example

```
class Outer
{     int outer_x = 100;
      void test()
      {         Inner in=new Inner();
                in.display();
      }
      class Inner
      {         void display()
                {
    System.out.println("display: outer_x = " + outer_x);
                }
      }
}
```

```
class NonStaticClassDemo
{
      public static void main(String args[])
      {
                Outer outer = new Outer();
                outer.test();
      }
}
```

String Operations in Java

String

String manipulation is the most common operation performed in Java programs. The String in java is a sequence of characters

Following are some useful classes that Java provides for String operations.

- String Class
- StringBuffer Class
- StringTokenizer Class

String Class

- String objects are read-only (immutable)
- There are two ways to create String object:
- By string literal
- By new keyword

Strings Basics

- Declaration and Creation:
  - **String** stringName;
  - stringName = **new String** (“string value”);
  - Example:
    - String city;
    - city = new String (“Bangalore”);
    - String s="java";
  - Length of string can be accessed by invoking length() method defined in String class:
    - int len = city.length();

String – Some useful operations

public int length()	Returns the length of the string.
public charAt(int index)	Returns the character at the specified location (*index)*
public int compareTo(String anotherString) public int compareToIgnoreCase (String anotherString)	Compare the Strings.
reigonMatch(int start, String other, int ostart, int count)	Compares a region of the Strings with the specified start.

String – Some useful operations

public String replace(char oldChar, char newChar)	Returns a new string with all instances of the *oldChar* replaced with *newChar.*
public trim()	Trims leading and trailing white spaces.
public String toLowerCase() public String toUpperCase()	Changes as specified.

String Class - example

```
// StringDemo.java: some operations on strings
class StringDemo {
        public static void main(String[] args)
        {
                String s = new String("Have a nice Day");

                // String Length =  15
                System.out.println("String Length = " + s.length() );
               // Have a Nice Day
               System.out.println("Modified String = " + s.replace('n', 'N'));

                // Converted to Uppercse =  HAVE A NICE DAY"
                System.out.println("Converted to Uppercase = " + s.toUpperCase());

                // Converted to Lowercase =  have a nice day"
                System.out.println("Converted to Lowercase = " + s.toLowerCase());

        }
}
```

Java StringBuffer Class

- Java StringBuffer class is used to create mutable (modifiable) String objects. The StringBuffer class in Java is the same as String class except it is mutable i.e. it can be changed.

Important Constructors of StringBuffer Class

Constructor	Description
StringBuffer()	It creates an empty String buffer with the initial capacity of 16.
StringBuffer(String str)	It creates a String buffer with the specified string..
StringBuffer(int capacity)	It creates an empty String buffer with the specified capacity as length.

```
StringBuffer s=new StringBuffer();
StringBuffer sb=new StringBuffer("Hello ");
StringBuffer sb1=new StringBuffer(25);
```

Important methods of StringBuffer class

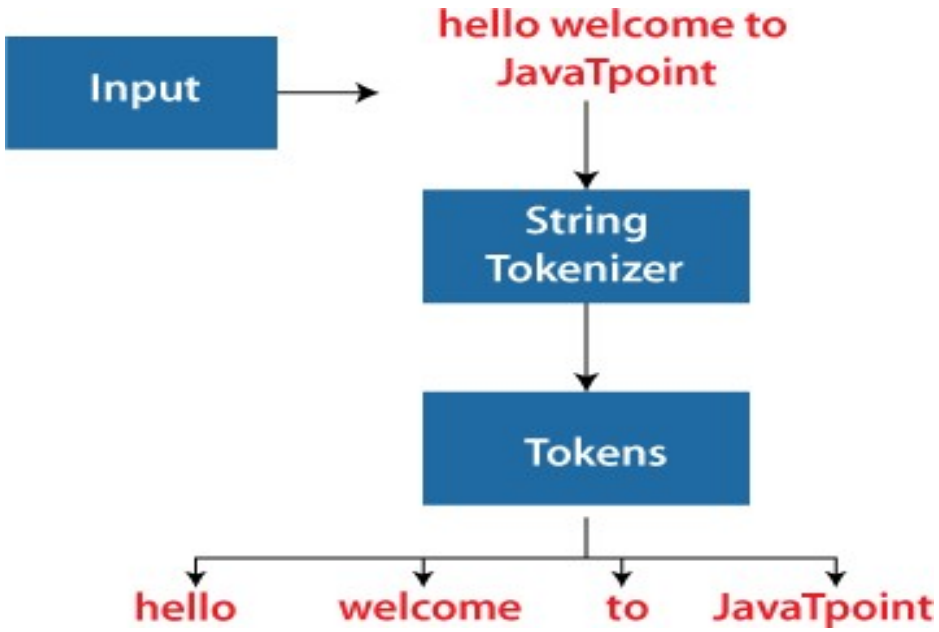
	Method	Description
	append(String s)	It is used to append the specified string with this string. The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc.
	insert(int offset, String s)	It is used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc.
	replace(int startIndex, int endIndex, String str)	It is used to replace the string from specified startIndex and endIndex.
	delete(int startIndex, int endIndex)	It is used to delete the string from specified startIndex and endIndex.
	substring(int beginIndex, int endIndex)	It is used to return the substring from the specified beginIndex and endIndex.

Cont..

	reverse()	is used to reverse the string.
	capacity()	It is used to return the current capacity.
	ensureCapacity(int minimumCapacity)	It is used to ensure the capacity at least equal to the given minimum.
	charAt(int index)	It is used to return the character at the specified position.
	length()	It is used to return the length of the string i.e. total number of characters.
	substring(int beginIndex)	It is used to return the substring from the specified beginIndex.

String Tokenizer in Java

- The **java.util.StringTokenizer** class allows you to break a String into tokens. It is simple way to break a String.



Constructor	Description
StringTokenizer(String str)	It creates StringTokenizer with specified string.
StringTokenizer(String str, String delim)	It creates StringTokenizer with specified string and delimiter.
StringTokenizer(String str, String delim, boolean returnValue)	It creates StringTokenizer with specified string, delimiter and returnValue. If return value is true, delimiter characters are considered to be tokens. If it is false, delimiter characters serve to separate tokens.

Methods	Description
boolean hasMoreTokens()	It checks if there is more tokens available.
String nextToken()	It returns the next token from the StringTokenizer object.
String nextToken(String delim)	It returns the next token based on the delimiter.
boolean hasMoreElements()	It is the same as hasMoreTokens() method.
int countTokens()	It returns the total number of tokens.

```
import java.util.StringTokenizer;

public class Simple{

 public static void main(String args[]){

   StringTokenizer st = new StringTokenizer("my name is kh
an"," ");

    while (st.hasMoreTokens())

        {

        System.out.println(st.nextToken());

  }}

}
```

my name is khan
Output

Type Casting in Java

- Convert a value from one data type to another data type is known as **type casting**.
- In Java, **type casting** is a method or process that converts a data type into another data type in both ways manually and automatically.
- The automatic conversion is done by the compiler and manual conversion performed by the programmer.

Type Casting

In Java, **type casting** is a method or process that converts a data type into another data type in both ways manually and automatically. The automatic conversion is done by the compiler and manual conversion performed by the programmer.

Types of Type Casting

- There are two types of type casting:
- Widening Type Casting
- Narrowing Type Casting

Widening Type Casting

- Converting a lower data type into a higher one is called **widening** type casting. It is also known as **implicit conversion** or **casting down**. It is done automatically. It is safe because there is no chance to lose data. It takes place when:
- Both data types must be compatible with each other.
- The target type must be larger than the source type.

byte -> short -> char -> int -> long -> float -> double

```java
public class WideningTypeCastingExample
{
public static void main(String[] args)
{
int x = 7;
//automatically converts the integer type into long type
long y = x;
//automatically converts the long type into float type
float z = y;
System.out.println("Before conversion, int value "+x);
System.out.println("After conversion, long value "+y);
System.out.println("After conversion, float value "+z);
}
}
```

```
Before conversion, the value is: 7
After conversion, the long value is: 7
After conversion, the float value is: 7.0
```

In the above example, we have taken a variable x and converted it into a long type. After that, the long type is converted into the float type.

Narrowing Type Casting

Converting a higher data type into a lower one is called **narrowing** type casting. It is also known as **explicit conversion** or **casting up**. It is done manually by the programmer. If we do not perform casting then the compiler reports a compile-time error.

double -> float -> long -> int -> char -> short -> byte

```
public class NarrowingTypeCastingExample
{
public static void main(String args[])
{
double d = 166.66;
//converting double data type into long data type
long l = (long)d;
//converting long data type into int data type
int i = (int)l;
System.out.println("Before conversion: "+d);
//fractional part lost
System.out.println("After conversion into long type: "+l);
//fractional part lost
System.out.println("After conversion into int type: "+i);
}
}
```

```
Before conversion: 166.66
After conversion into long type: 166
After conversion into int type: 166
```

Inheritance

Inheritance: Introduction

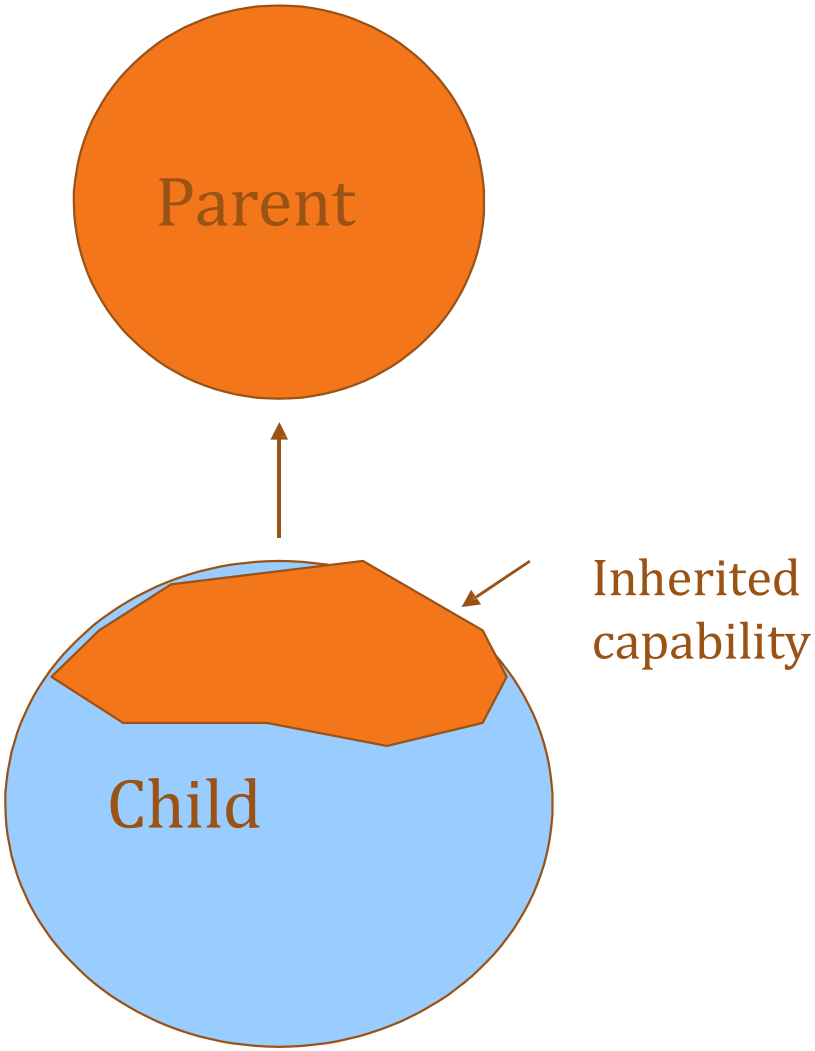
- Reusability--building new components by utilising existing components- is yet another important aspect of OO paradigm.
- It is always good/“productive” if we are able to reuse something that is already exists rather than creating the same all over again.
- This will achieve by creating new classes, reusing the properties of existing classes.

Java - Inheritance

- Inheritance can be defined as the process where one class acquires the properties (methods and fields) of another.
- The class which inherits the properties of other is known as subclass (derived class, child class) and the class whose properties are inherited is known as superclass (base class, parent class).

Inheritance: Introduction

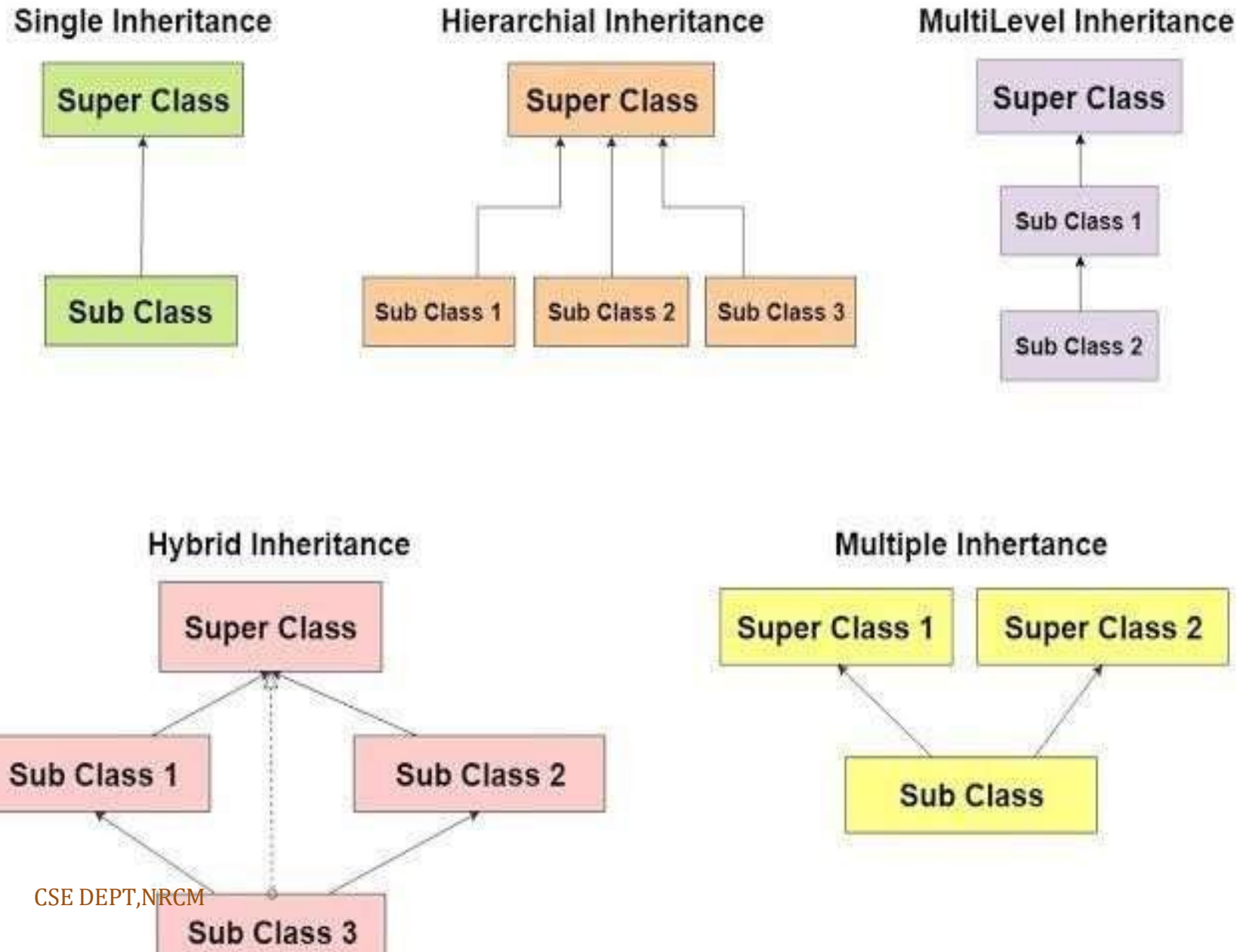
- This mechanism of deriving a new class from existing/old class is called “inheritance”.
- The old class is known as “base” class, “super” class or “parent” class”; and the new class is known as “sub” class, “derived” class, or “child” class.



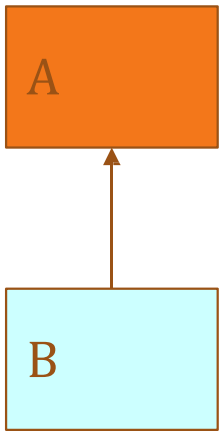
Inheritance Introduction

- The inheritance allows subclasses to inherit all properties (variables and methods) of their parent classes. The different forms of inheritance are:
  - Single inheritance (only one super class)
  - Multiple inheritance (several super classes)
  - Hierarchical inheritance (one super class, many sub classes)
  - Multi-Level inheritance (derived from a derived class)
  - Hybrid inheritance (more than two types)

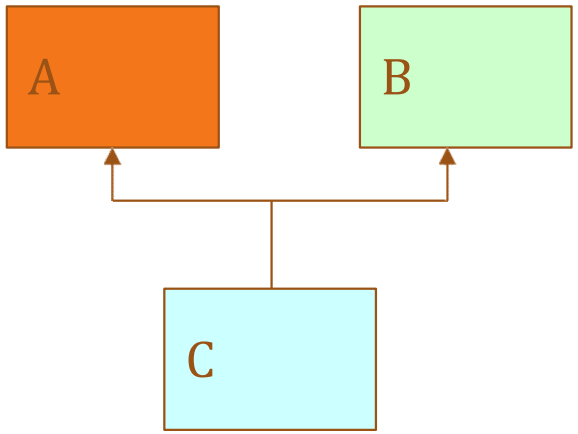
T


Single Inheritance
Super Class
Sub Class
Hierarchial Inheritance
Super Class
Sub Class 1
Sub Class 2
Sub Class 3
MultiLevel Inheritance
Super Class
Sub Class 1
Sub Class 2
Hybrid Inheritance
Super Class
Sub Class 1
Sub Class 2
Sub Class 3
Multiple Inhertance
Super Class 1
Super Class 2
Sub Class

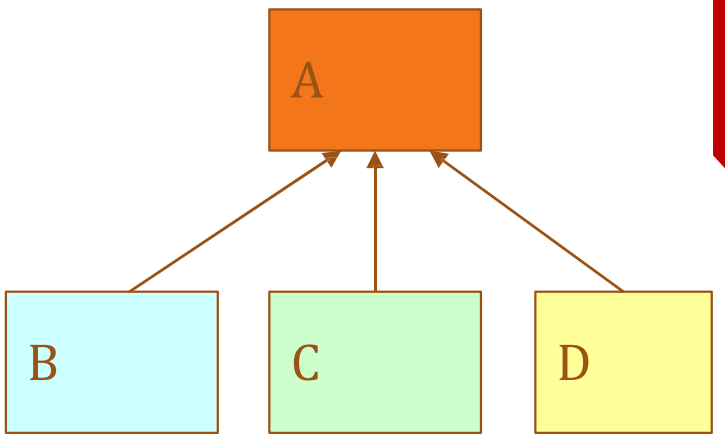
Types / Forms of Inheritance



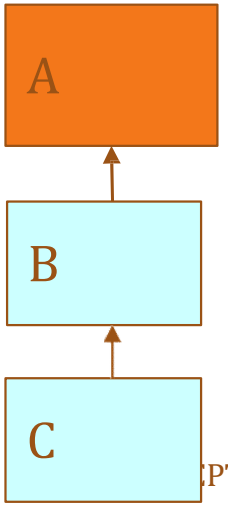
(a) Single Inheritance



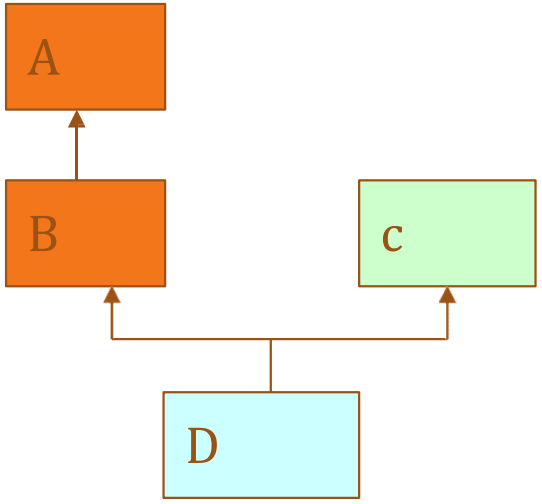
(b) Multiple Inheritance



(c) Hierarchical Inheritance



(a) Multi-Level Inheritance



(b) Hybrid Inheritance

Defining a Sub class

- A subclass/child class is defined as follows:

```
class SubClassName extends SuperClassName
{
        fields declaration;
        methods declaration;
}
```

- The keyword “extends” signifies that the properties of super class are extended to the subclass. That means, subclass contains its own members as well of those of the super class. This kind of situation occurs when we want to enhance properties of existing class without actually modifying it.

```
class Employee{
 float salary=40000;
}
class Programmer extends Employee
{
 int bonus=10000;
 public static void main(String args[])
{
  Programmer p=new Programmer();
System.out.println("Programmer salary is:"+p.salary);
System.out.println("Bonus of Programmer is:"+p.bonus);
}
}
```

Subclasses & Constructors

- Default constructor automatically calls constructor of the base class:

default constructor for Circle class is called

```
GraphicCircle drawableCircle = new GraphicCircle();
```

Subclasses & Constructors

- Defined constructor can invoke base class constructor with *super*:

```
public GraphicCircle(double x, double y, double r,
Color outline, Color fill) {
    super(x, y, r);
    this.outline = outline;
    this fill = fill
}
```

Shadowed Variables

- Subclasses defining variables with the same name as those in the superclass, *shadow* them:

Shadowed Variables - Example

```
public class Circle {
        public  float r = 100;
}

public class GraphicCircle extends Circle {
        public float r = 10;       // New variable, resolution in dots per inch
}

public class CircleTest {
        public static void main(String[]
                args){ GraphicCircle gc = new
                GraphicCircle(); Circle c = gc;
                System.out.println(" GraphicCircleRadius= " + gc.r);    // 10
                System.out.println    (" Circle Radius = " + c.r);              // 100
        }
}
```

Garbage Collection

- Since objects are dynamically allocated by using the **new operator**
- When objects are destroyed and the memory released is used for later reallocation.
- In some languages, such as C++, dynamically allocated objects must be manually released by use of a **delete operator.**
- Java takes a different approach; it handles de allocation for us automatically.
- The technique that accomplishes this is called ***garbage collection.***

The finalize( ) Method

- Sometimes an object will need to perform some action when it is destroyed.
- For example, if an object is holding some non-Java resource such as a file handle or network connections, data base connections, then we might want to make sure these resources are freed before an object is destroyed.
- To handle such situations, Java provides a mechanism called *finalization.*
- To add a finalizer to a class, you simply define the **finalize( ) method.**

The finalize( ) Method Contd.,

- The Java run time calls that method whenever it is about to recycle an object of that class.
- Inside the **finalize( )** method, you will specify those actions that must be performed before an object is destroyed.
- The garbage collector runs periodically, checking for objects that are no longer referenced

```
protected void finalize( )
{
        // finalization code here
}
```

this keyword in Java

- In Java, this is a **reference variable** that refers to the current object.
- Usage of Java this keyword
- Here is given the usage of java this keyword.
- this can be used to refer current class instance variable.
- this can be used to invoke current class method (implicitly)
- this() can be used to invoke current class constructor.

1) this: to refer current class instance variable

- The this keyword can be used to refer current class instance variable. If there is ambiguity between the instance variables and parameters, this keyword resolves the problem of ambiguity.
- In the bellow example, parameters (formal arguments) and instance variables are same. So, we are using this keyword to distinguish local variable and instance variable.

```
class Student{

int rollno;

String name;

float fee;

Student(int rollno,String name,float fee){

rollno=rollno;

name=name;

fee=fee;

}

void display(){System.out.println(rollno+" "+name
+" "+fee);}

}
```

```
class TestThis1{

public static void main(String args[]){

Student s1=new Student(111,"ankit",50
00f);

Student s2=new Student(112,"sumit",60
00f);

s1.display();

s2.display();

}}
```

Output:

```
0 null 0.0
 0 null 0.0
```

```
class
Student{ int
rollno; String
name; float
fee;
Student(int rollno,String name,float fee){
this.rollno=rollno;
this.name=name;
this.fee=fee;
}
void display(){System.out.println(rollno+" "+name+" "+fee);}
}
```

```
class TestThis2{
public static void main(String args[]){
Student s1=new Student(111,"ankit",5000f);
Student s2=new Student(112,"sumit",6000f);
s1.display();
s2.display();
}}
```

Output:

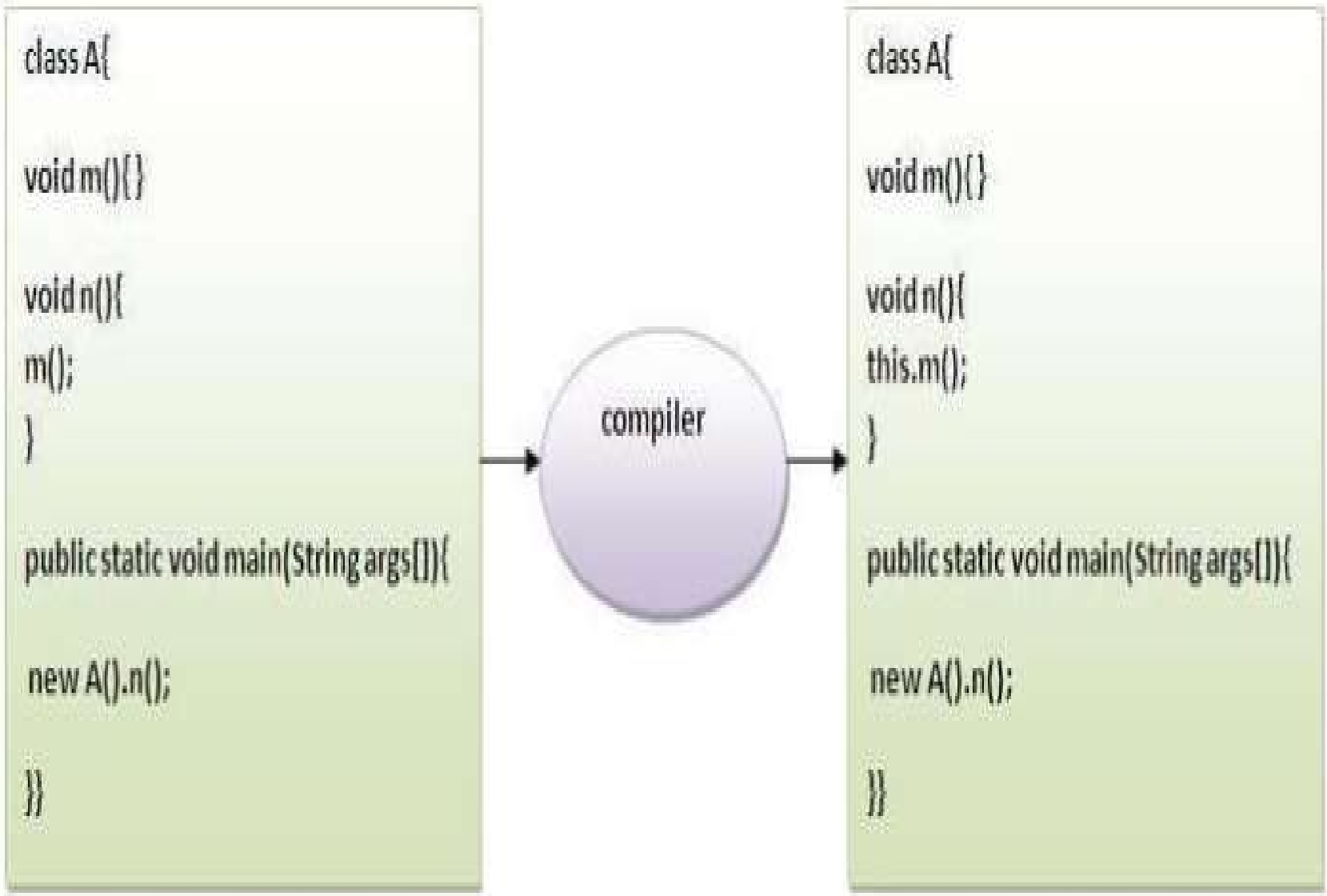
111 ankit 5000.0
112 sumit 6000.0

2) this: to invoke current class method

You may invoke the method of the current class by using the this keyword. If you don't use the this keyword, compiler automatically adds this keyword while invoking the method. Let's see the example

```
class A{

void m(){System.out.println("hello m");}

void n(){

System.out.println("hello n");

//m();//same as this.m()

this.m();

}

}

class TestThis4{

public static void main(String args[]){

A a=new A();

a.n();

}}
```

Output:
hello n
hello m


class A{
void m(){}
void n(){
m();
}
public static void main(String args[]){
new A().n();
}}
compiler
class A{
void m(){}
void n(){
this.m();
}
public static void main(String args[]){
new A().n();
}}

3) this() : to invoke current class constructor

The this() constructor call can be used to invoke the current class constructor. It is used to reuse the constructor.

In other words, it is used for constructor chaining.

```
class A{

A(){System.out.println("hello a");}

A(int x){

this();

System.out.println(x);

}

}

class TestThis5{

public static void main(String args[]){

A a=new A(10);

}}
```

Output:
hello a
10

```
class A{
A(){
this(5);
System.out.println("hello a");
}
A(int
x)
{
System.out.println(x);
}
}
class TestThis6{
public static void main(String
args[]){ A a=new A();
}}
```

Output:
5
hello a

Method Overriding in Java

- If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in java**.

- Derived/sub classes defining methods with same name, return type and arguments as those in the parent/super class, it is known as **method overriding in java**.

- In other words, If subclass provides the specific implementation of the method that has been provided by one of its parent class, it is known as method overriding.

Usage of Java Method Overriding

- Method overriding is used to provide specific implementation of a method that is already provided by its super class.
- Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

- method must have same name as in the parent class
- method must have same parameter as in the parent class.
- must be IS-A relationship (inheritance).

```
class Vehicle{
void run(){System.out.println("Vehicle is running");}
}
class Bike2 extends Vehicle{
void run(){System.out.println("Bike is running safely");}

public static void main(String
args[]){ Bike2 obj = new Bike2();
obj.run();
}
```

Overriding Methods

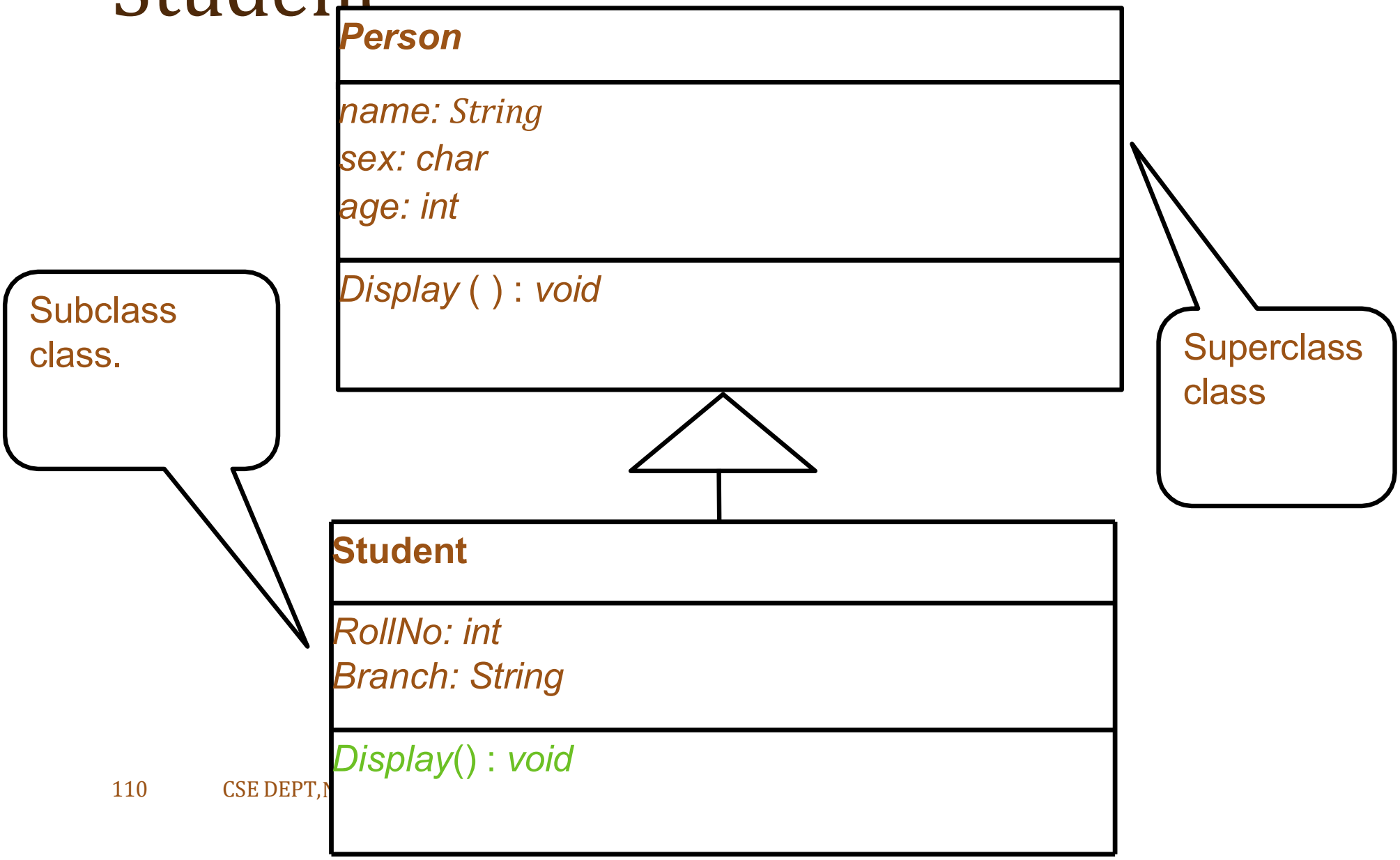
```
class A {
        int j = 1;
        int f( ) { return j; }
}



class B extends A {
        int j = 2;
        int f( )
        { return
        j; }
}
```

Overriding Methods

```
class override_test {
        public static void main(String args[]) {
                B b = new B();
                System.out.println(b.j);        // refers to B.j prints 2 System.out.println(b.f());
// refers to B.f prints 2

                A a = (A) b;                                    Object Type Casting
                System.out.println(a.j);      // now refers to a.j prints 1
                System.out.println(a.f());    // overridden method still refers to B.f() prints 2 !
        }
}
```

```
inheritance java
override_test
2
2
1
2
```

Using All in One: Person and Student



Person class: Parent class

```
// Student.java: Student inheriting properties of person class


class person
{
        private String name;
        protected char sex; // note protected
        public int age;
        person()
        {
                name = null;
                sex = 'U'; // unknown
                age = 0;
        }
```

```
 person(String name, char sex, int age)
        {
                this.name = name;
                this.sex = sex;
                this.age = age;
        }
        String getName()
        {
                return name;
        }
        void Display()
        {
        System.out.println("Name = "+name);
        System.out.println("Sex = "+sex);
        System.out.println("Age = "+age);

        }
}
```

Student class: Derived class

```
class student extends person
{
        private int RollNo;
        String branch;

student(String name, char sex, int age, int
RollNo, String branch)
        {
                super(name, sex, age);
 // calls parent class's constructor with 3
arguments
                this.RollNo = RollNo;
                this.branch = branch;
        }
        void Display()  // Method Overriding
        {
System.out.println("Roll No = "+RollNo);
System.out.println("Name =
"+getName());
                System.out.println("Sex =
"+sex);
                System.out.println("Age =
"+age);
                System.out.println("Branch =
"+branch);
        }
        void TestMethod()
// test what is valid to access
        {
        //      name = "Mark"; Error: name is
private
                sex = 'M';
                RollNo = 20;
        }
}
```


What happens if super class constructor is not explicitly invoked ? (default constructor will be invoked).

```
class MyTest
{

public static void main(String args[] )
        {
    student s1 = new student("Rama",
'M', 21, 1, "Computer Science");


 student s2 = new student("Sita", 'F',
19, 2, "Software Engineering");
```

```
System.out.println("Student 1
Details...");

s1.Display();
System.out.println("Student 2
Details...");

s2.Display();

 person p1 = new person("Rao",
'M', 45);
  System.out.println("Person
Details...");

 p1.Display();

        }
}
```


Can we create Object of person class ?

Output

```
inheritance [1:154] java MyTest

Student 1 Details...
Roll No = 1
Name = Rama
Sex = M
Age = 21
Branch = Computer Science
Student 2 Details...
Roll No = 2
Name = Sita
Sex = F
Age = 19
Branch = Software Engineering
Person Details...
Name = Rao
Sex = M
Age = 45
inheritance [1:155]
```

Super Keyword in Java

- The **super** keyword in Java is a reference variable which is used to refer immediate parent class object.

Usage of Java super Keyword

- super can be used to refer immediate parent class instance variable.
- super can be used to invoke immediate parent class method.
- super() can be used to invoke immediate parent class constructor.

1) super is used to refer immediate parent class instance variable.

```
class Animal{

String color="white";

int a=10;

}

class Dog extends Animal{

String color="black";

void

printColor()

{

System.out.println(color);

System.out.println(a);11//prints color of Dog c
lass
```

```
class TestSuper1{

public static void main(Strin
g args[]){

Dog d=new Dog();

d.printColor();

}}
```

Output:
Black
white

of Animal class)

2) super can be used to invoke parent class method

The super keyword can also be used to invoke parent class method. It should be used if subclass contains the same method as parent class. In other words, it is used if method is overridden.

```
class Animal{

void eat()

{System.out.println("eating...");}

}

class Dog extends Animal{

void eat()

{System.out.println("eating bread...");}

void bark()

{System.out.println("barking...");}

void work(){

super.eat();

bark();
```



```
}

}
```

```
class TestSuper2{

public static void main(
String args[]){

Dog d=new Dog();

d.work();

}}
```

3) super is used to invoke parent class constructor.

- The super keyword can also be used to invoke the parent class constructor.
- As we know well that default constructor is provided by compiler automatically if there is no constructor. But, it also adds super() as the first statement.
- Let's see a simple example:

```
class Animal{

Animal()

{System.out.println("animal is c
reated");}

}

class Dog extends

Animal{ Dog(){

super();

System.out.println("dog is creat
ed");

}

}
```

```
class TestSuper3{

public static void main(Stri
ng args[]){

Dog d=new Dog();

}}
```

Output:
animal is created
dog is created

Dynamic Method Dispatch

- Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.
- Dynamic method dispatch is important because this is how Java implements run-time polymorphism.

Dynamic Method Dispatch

- When an overridden method is called through a superclass reference,

  Java determines which version of that method to execute based upon

  the type of the object being referred to at the time the call occurs.

- Thus, this determination is made at run time. When

  different types of objects are referred to, different versions

  of an overridden method will be called.

```
// Dynamic Method Dispatch
class A {
void callme()
{ System.out.println("Inside A's
callme method");
}
}
class B extends A {
// override callme()
void callme()
{ System.out.println("Inside B's
callme method");
}
}
class C extends A {
// override callme()
void callme()
{ System.out.println("Inside C's
callme method");
}
}
```

```
class Dispatch {
public static void main(String args[]) {
A a = new A(); // object of type A
B b = new B(); // object of type B
C c = new C(); // object of type C
A r; // obtain a reference of type A
r = a; // r refers to an A object
r.callme(); // calls A's version of callme
r = b; // r refers to a B object
r.callme(); // calls B's version of callme
r = c; // r refers to a C object
r.callme(); // calls C's version of callme
}
}
```

The output from the program is shown here:

Inside A’s callme method
Inside B’s callme method
Inside C’s callme method

Abstract Classes

Using Abstract Classes

- There are situations in which you will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method.
- That is, sometimes you will want to create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details. Such a class determines the nature of the methods that the subclasses must implement.

Abstract Classes

- A superclass that declares the structure of a given abstraction without providing a complete implementation of every method.
- A superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details.
- This situation can occur is when a superclass is unable to create a meaningful implementation for a method.

Abstract Methods

- Java's solution to this problem is the *abstract method.*
- To declare an abstract method, use this general form:

abstract *type methodname(parameter-list)*;

*** no method body is present.

Abstract Classes

- Any class that contains one or more abstract methods must be declared abstract.
- To declare a class abstract, simply use the **abstract** keyword in front of the **class** keyword at the beginning of the class declaration.
- There can be no objects of an abstract class. That is, an abstract class cannot be directly instantiated with the **new** operator.
- Such objects would be useless, because an abstract class is not fully defined.

Abstract Classes

```
abstract class draw
{
    double d1,d2;
    draw(double a,
    double b)
    {
          d1=a;d2=b;
    }
    abstract double
    area();
}
```

```
class triangle extends draw
{
    triangle(double a,double b)
    {
          super(a,b);
    }
    double area()
    {
    System.out.println(" Inside
    Area for Triangle : ");
    return d1*d2/2; // base *
    height / 2
    }
}
```

Abstract Classes

```
class rectangle extends draw
{
rectangle(double a,double b)
{
        super(a,b);
}


double area()
{
System.out.println(" Inside Area
for Rectangle : ");
return d1*d2;  // length *
breadth
}
}
```

```
class abstractclasses
{    public static void main(String ar[])
     {
//  draw d=new draw(10,20);
                                  is illegal
            triangle t=new
            triangle(12,8);  rectangle
            r=new rectangle(6,4);
            draw ref;

            ref=t;
     System.out.println("Area is
     :"+ref.area());
            ref=r;
  }
     System.out.println("Area is
     :"+ref.area());
     }
```