

# **Object Oriented Programming**

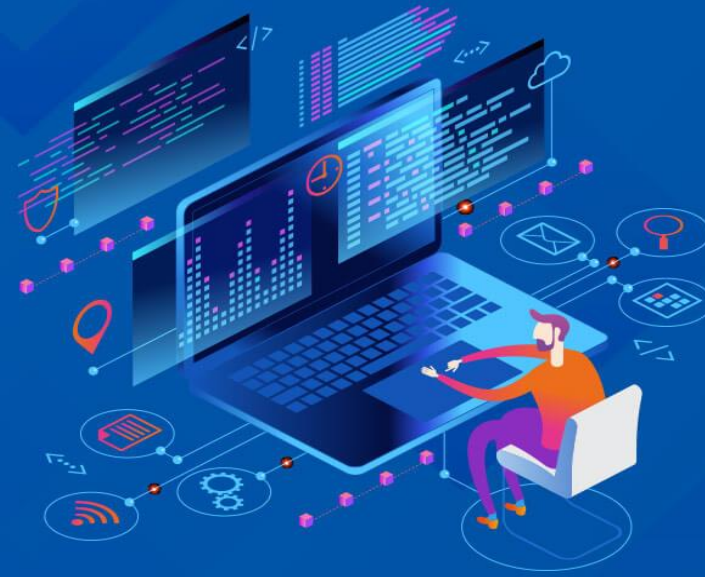
**20EC405**  
**(Revision of concepts)**

**Harshala Khedlekar-Bodas**



# Agenda

- Principles of Object-Oriented Programming
- Classes and objects, methods
- Object creation, constructors, method overloading, static, final, access specifiers



# What is OOP ???

- OOP stands for **O**bject **O**riented **P**rogramming
- OOP is a *programming style*
- OOP is about creating programs which consist of **Objects** and their communication with each other.
- Objects are created using a template which is called as a **Class**.

# How Procedural Languages Work ?

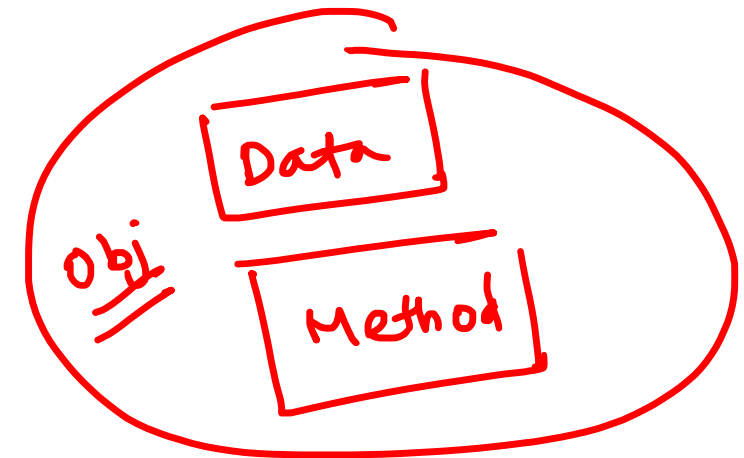
- They use a list of instructions which tell the compiler what to do in which order.
- They are action oriented in the sense that they indicate what actions are to be performed by a program.
- Examples : COBOL, C.

# How Object Oriented Languages Work ?

- They rely on **objects** which represent **real life entities**.
- Objects hold data which depict their state.
- Objects communicate with each other by sharing and accessing data
- Objects can expose functionality for other objects to invoke.
- Examples: SIMULA, Smalltalk, C++, Java

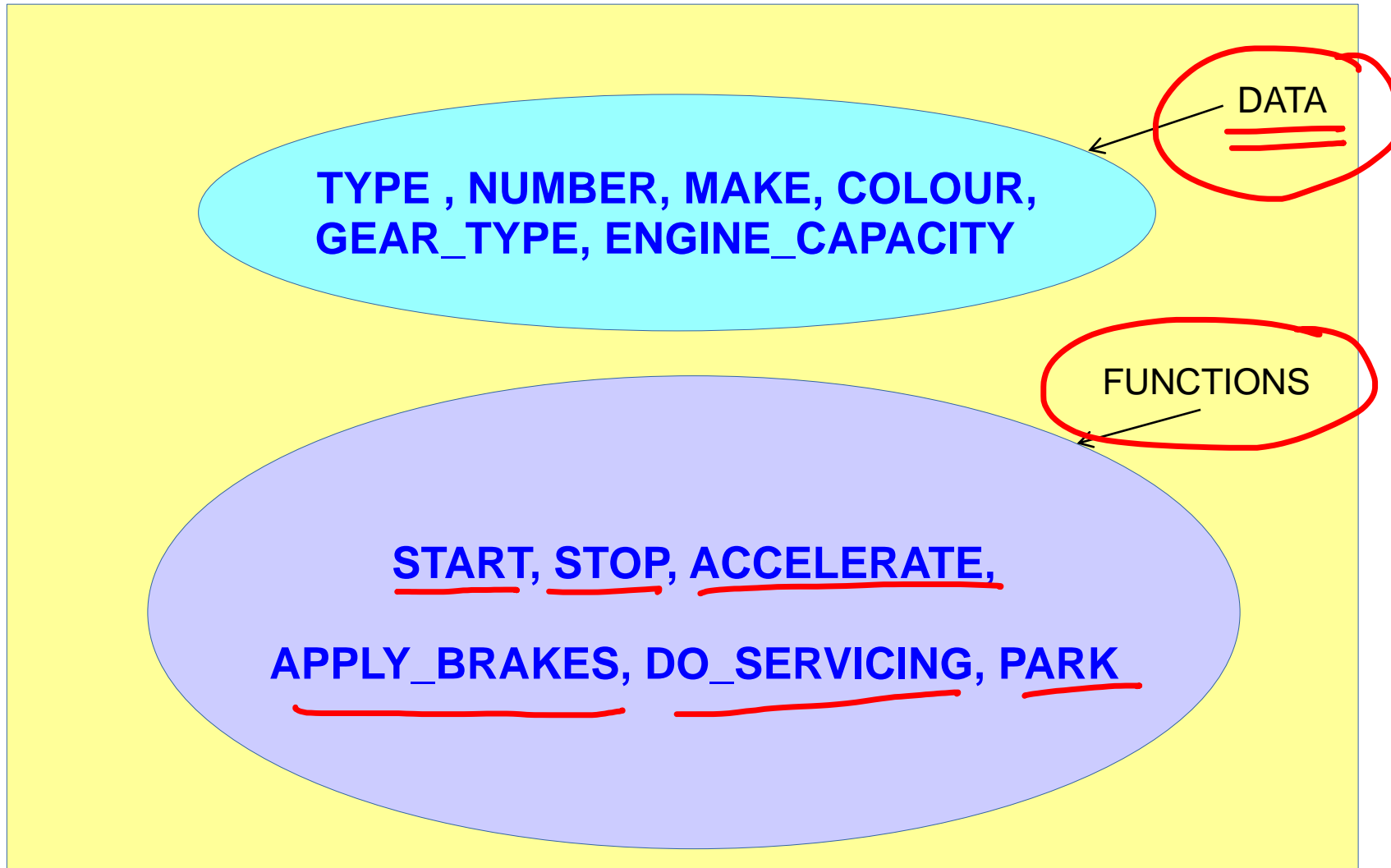
# What an object typically has ?

- ✓ **Data** - This is called Data or Fields or Instance Variables in various languages. This typically indicates state of object and represents the data that object holds
- ✓ **Functions** - This is called method in Java. This typically provides actions allowed on the data that the object holds.



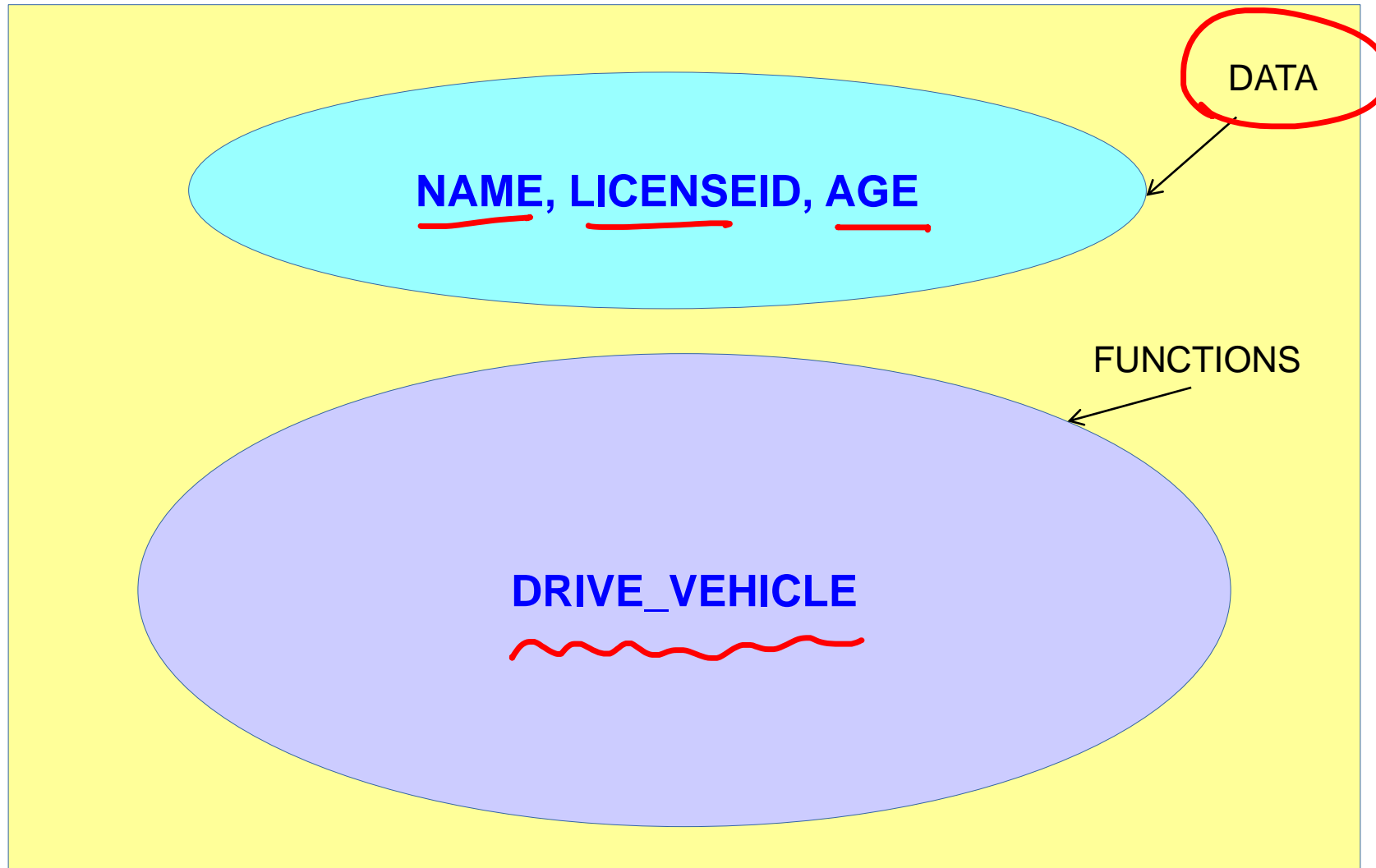
# Example : A Car Object

has



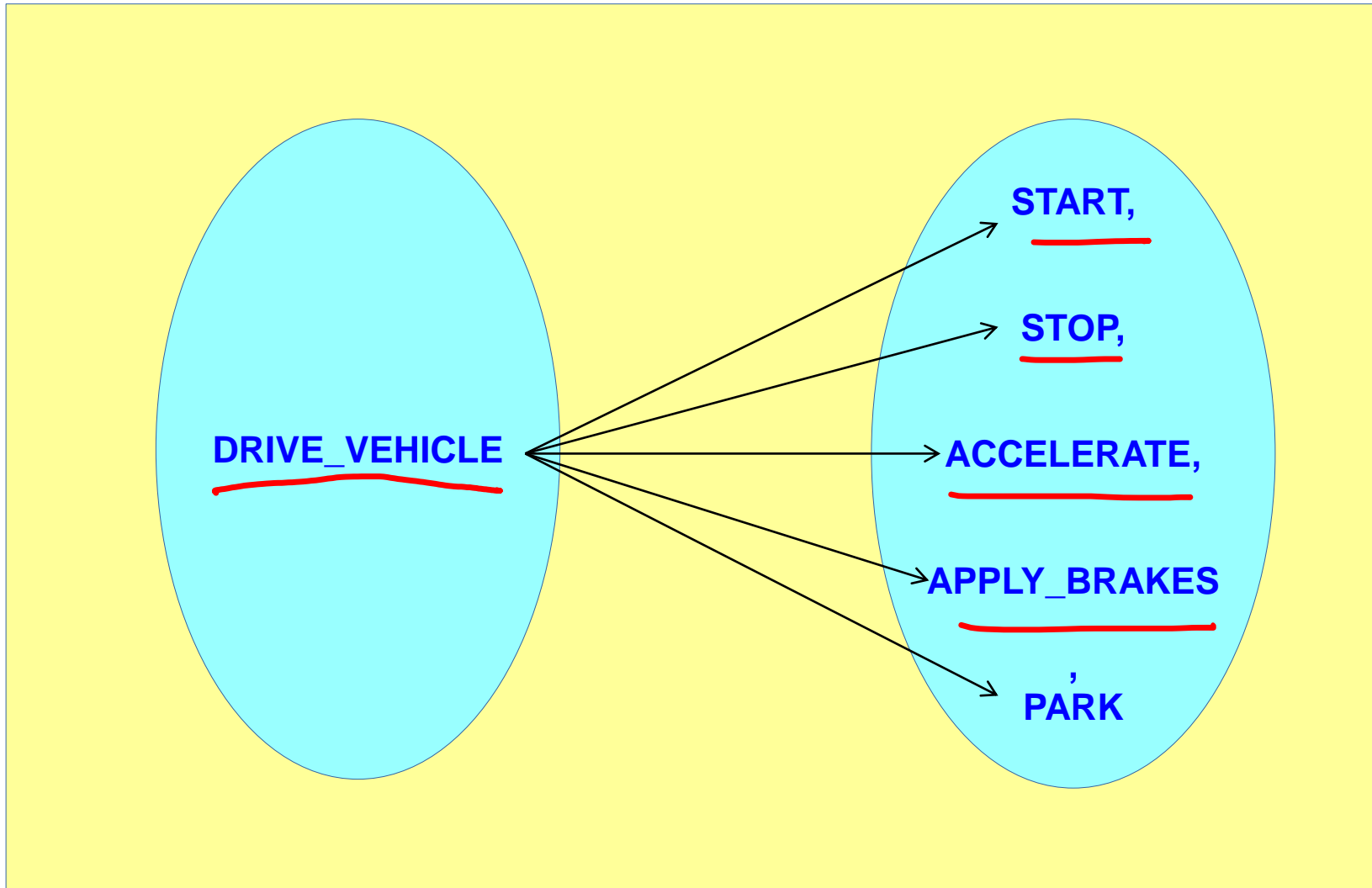
Methods

# Example : A *Driver* Object





# Example : Driver - Car Interaction



# So what is an Object ?

- Consider real world objects such as your vehicle, your TV set, laptop.
- ALL these objects have two basic characteristics - They have **state** and they have **behaviour**.
- For example your computer or laptop has a state such as color, memory, processor speed etc. and it has methods such as start, shutdown, restart etc.

# So what is an Object ?

- Software objects are similar.
- They store the object data into fields which maintain state
- They define methods/actions which can happen *on* that object.

# And what is that Class thing ?

- A class is nothing but a **blueprint** or **template** to create objects.
- When we say "a object has so-and-so method or function", what it really means is that its corresponding class has that method defined in it.
- ✓ To create a object, you need a class definition.
- Think of class as a data type defined by you !!

# And what is that Class thing ?

- For example, to create a car object, first you define a Car class.
- Then you can create 1 or 10 or 1000 or as many cars you want.

# Remember...

- Class is a blueprint.
- Objects are *created* based on that class blueprint.
- One class <===> Many objects

# Formally speaking..

- Classes are user defined data types that behave similar to built-in data types
- Objects are the basic "run-time" entities created using class definition.

# OOP Cornerstones

Object Oriented languages are famous by the principles they follow. These are:

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism.



# Abstraction

- Abstraction refers to the act of representing essential features without including the background details.
- For example, If a class defines a method to sort integers, you simply need to know how to invoke the method. You really don't care how sorting works inside...

# Encapsulation

- Encapsulation is hiding the data from external world.
- You keep your data *private*
- The only way to access the data is in controlled manner via functions which you define to be publicly available.
- Example: Whatsapp profile photo.



```
class student {  
    private int roll ;  
}  
  
    roll  
  
    { getRoll  
      setRoll }
```

# Inheritance

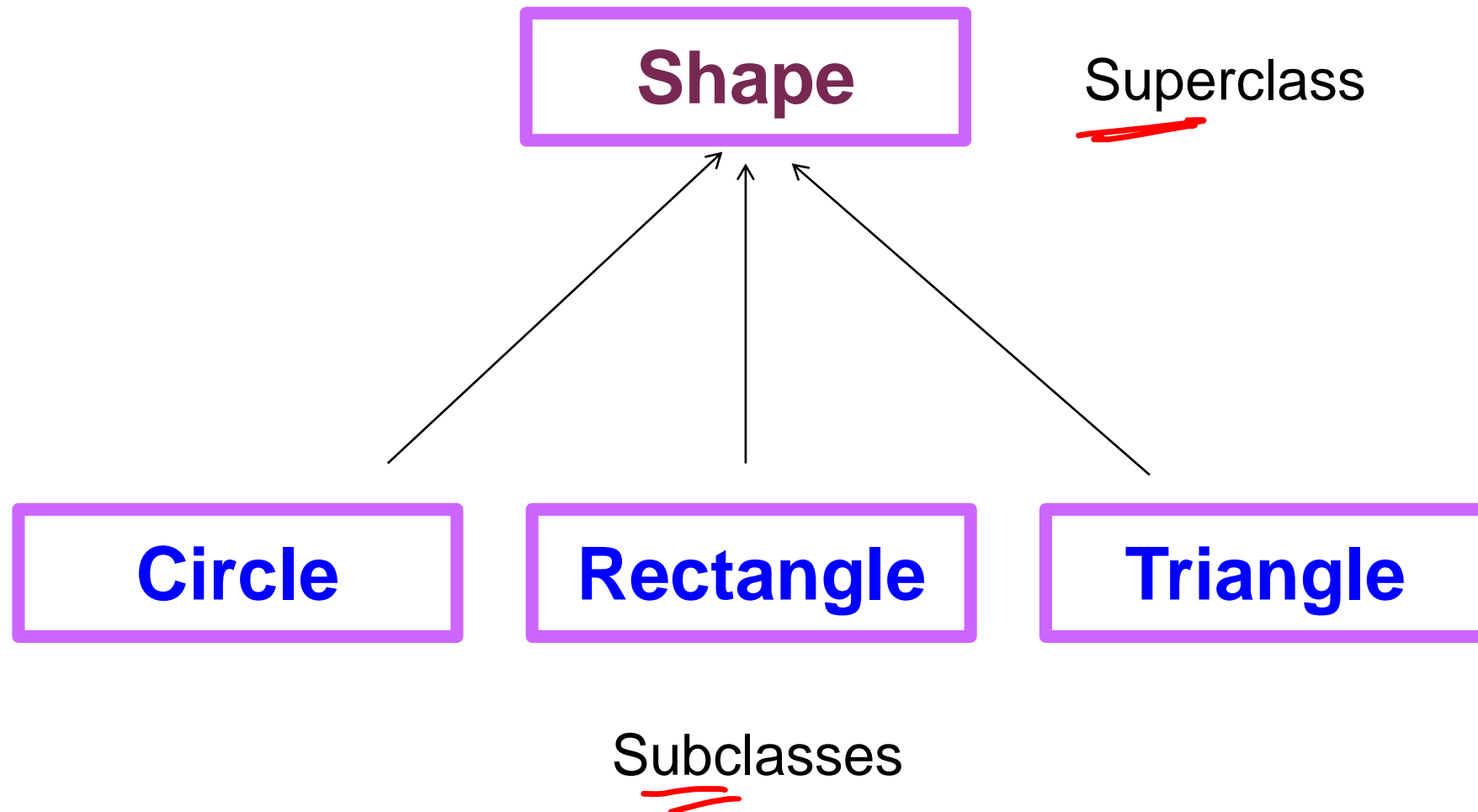
- Inheritance is a process by which objects of one class acquire properties and behaviour of another class.
- The class who is acquiring the behaviour is called subclass or child class, the class whose behaviour is acquired is called superclass or parent class.

# Inheritance



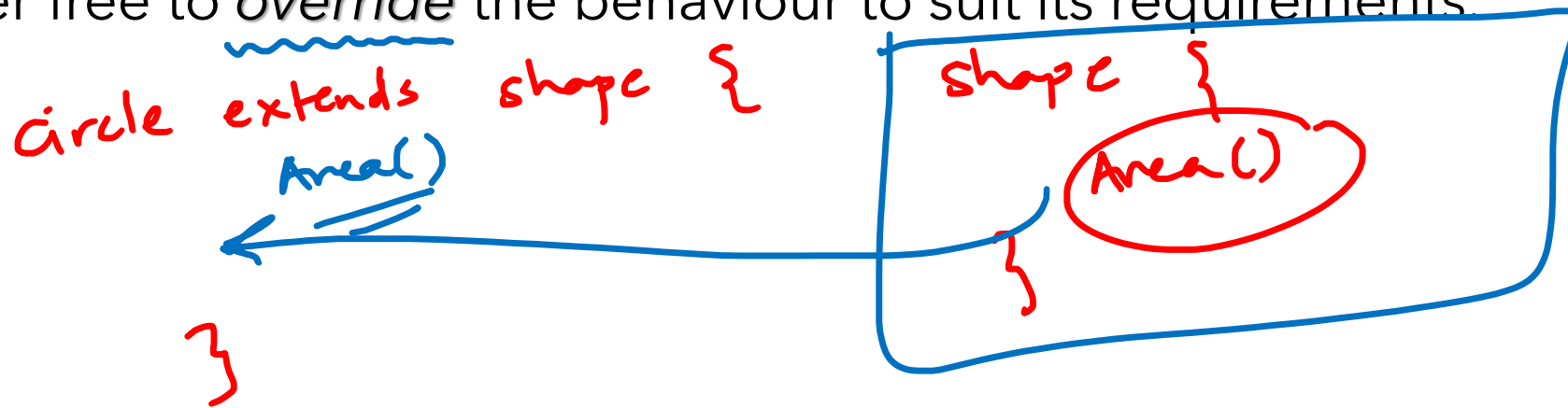
- Consider shape for example:
- A **Cirle** is a **Shape**
- A **Rectangle** is a **Shape**
- A **Triangle** is a **Shape**
- So there are four classes – Circle, Rectangle, Triangle *and* Shape.

# Inheritance



# What is inherited ?

- Inheritance allows subclass to access both data fields and functions/methods of superclass.
- For example, if Shape class has a function called "calculateArea", all the subclasses would *automatically* have a function called "calculateArea" with the same implementation as defined in Shape class. Each subclass is however free to override the behaviour to suit its requirements.



# Polymorphism

- Polymorphism is ability to take many forms.
- In object oriented languages, it is possible to create functions which exhibit different behaviours in different instances.
- For example, a same function "calculateArea" can be used to calculate area for circle, rectangle and triangle based on the context.

# Polymorphism - Example

- Consider the example of Shapes.
- As described earlier, all the three classes Circle, Rectangle and Triangle would *inherit* the method "calculateArea" from parent class.
- But each shape would change this method implementation to suit its requirement.



# Polymorphism - Example

- Now imagine that there is a program function which is required to return area for any shape provided to it.
- Here's where polymorphism comes into picture !

# Polymorphism - Example

- In object oriented language, a single method accepts all shapes and then calls "calculateArea" on any shape provided as input.
- At runtime, the program decides which subclass of Shape was given as input, and invokes the calculateArea function on the subclass.
- The advantage is that, even if you add new Shape classes, this method does not change. It simply delegates the work to the respective class for actual calculation.

# Defining a class

- A class is defined using a keyword **class**, followed by class name and followed by its contents enclosed in curly braces.

```
class name {
```

```
}
```

- A class would typically contain variables and methods.

# Defining a method

A method declaration typically follows following format:

```
<modifiers> type name (paramlist) {  
    statements;  
}
```

Where,

- modifiers* are access modifiers and other modifiers which define the visibility and other facets
- type* is the return type of method
- name* is method name
- paramlist* is comma separated list of parameters

# Defining method - Modifiers

- A method can be declared using zero or more modifiers.
- These modifiers can be of two types.
  - Access level modifiers: private, protected and public default
  - Modifiers which define the method behaviour: Ex. static, final, abstract, synchronized etc.

# Defining method - type

- A method **must** specify a return type.
- A return type can be
  - any primitive type (such as int, char, double etc)
  - or a built in type (such as String, StringBuilder, File, Connection etc)
  - or any user defined data type (such as Person, Shape, Employee, Rectangle etc)
- If a method does not return anything, the return type is specified as **void**.

# Defining method - name

- A method name in java can be any legal identifier.
- However, there are certain code conventions which java developers all around the world follow.
- By convention:
  - A method name begins with a verb which represents the action (behavior) exhibited by that method.
  - A method name starts with lower case character. In multi-word method names, first letter of every word is Capitalized except first word. This is called camel case.
  - Method names do not have underscores.
- Examples: getInstance, run, doSomething, drive, getFile, deleteRecord etc.

# Defining method - name

- Note that, in java, a method name need not be unique inside a class.
- What needs to be unique is method signature, which constitutes of **method name and parameter list**. You can have any number of methods with same name as long as the parameter list for them is different

```
int get Integer (   ) {  
}  
void get Integer (   ) {  
}
```

```
int getInteger ( ) {  
}  
int getInteger (int a) {  
}
```



# Defining method - paramlist

- This is, list of parameters separated by comma.
- We can also have variable arguments aka varargs.

# Creating objects

- In java, to create an object, we use **new** operator.
- There *is* another way called reflection, but it is somewhat advanced concept and you normally would not use it.
- The syntax to create a new object is:

```
suptype obj_name = new type();  
* Student s = new Student();
```

# Creating objects

- Suptype is a data type (either standard or user defined) which is same as type or superclass of type
- obj\_name is name of the object to be created
- type() is the constructor inside the class defined by type.

# Creating a object

- Consider the Rectangle example.

Rectangle rect = new Rectangle();

- In this statement, leftside "Rectangle" denotes the type of the reference.
- "rect" is the object name
- Rectangle() is the constructor in Rectangle.java class which is invoked for creating the object.

# What is a reference ?

- Java does not have pointers.
- *So how do you point to an object ?*
- *How do you manipulate and use an object ?*
- In java this is done using **reference variables**.

# What is a reference ?

- Note that a reference is NOT same as pointer. It cannot be used to manipulate memory of objects.
- A reference in java is simply ***a way to get a handle to the object***, which is used to invoke methods on that object.

# What is a reference ?

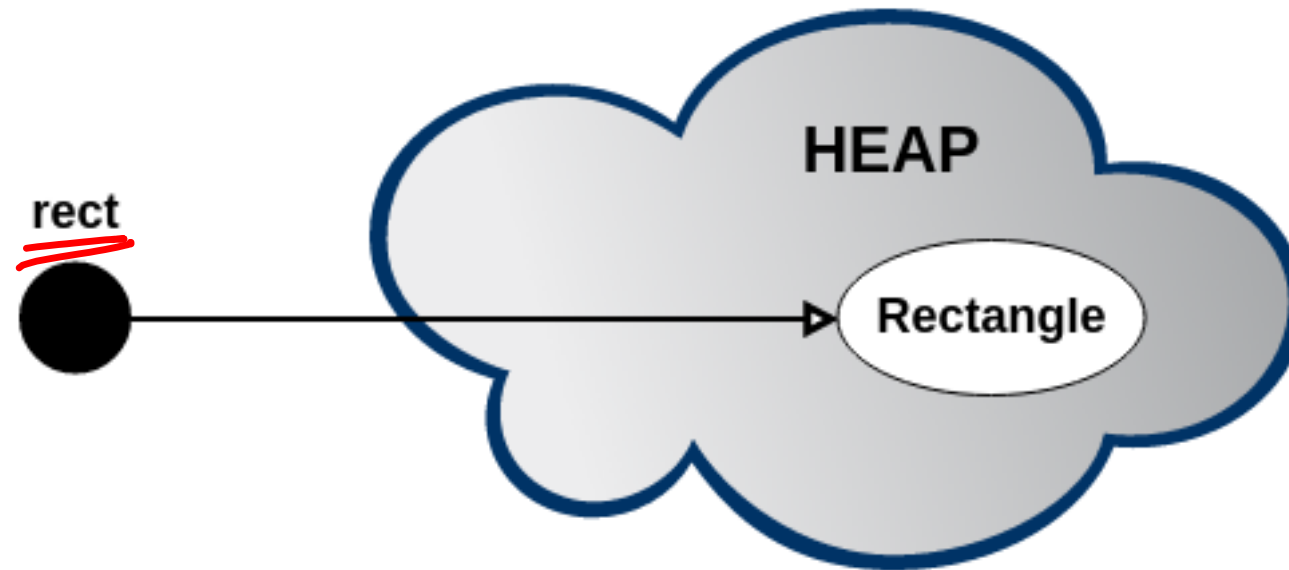
- So when we write a statement like following:

✓ Rectangle rect = new Rectangle( ); ✓

- We are creating a Rectangle object on heap and providing a reference to it using the name rect.
- Note that, in this statement, we have combined declaration and instantiation at same time. We can separate two statements as

{ Rectangle rect; // declaration  
rect = new Rectangle( ); // instantiation.

# What is a reference ?





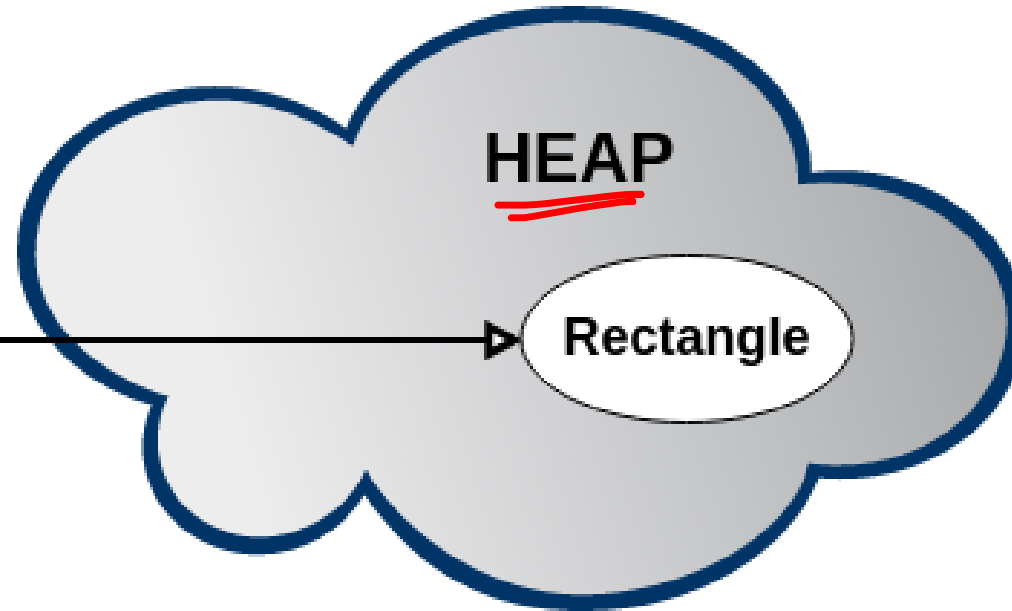
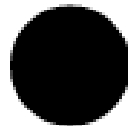
# What is a reference ?

```
{ Rectangle rect1; ✓  
  Rectangle rect2 = new Rectangle();
```

rect1



rect2



# What is a reference ?

- We can also have multiple references pointing to same object.
- For example, consider following:

Rectangle r1;

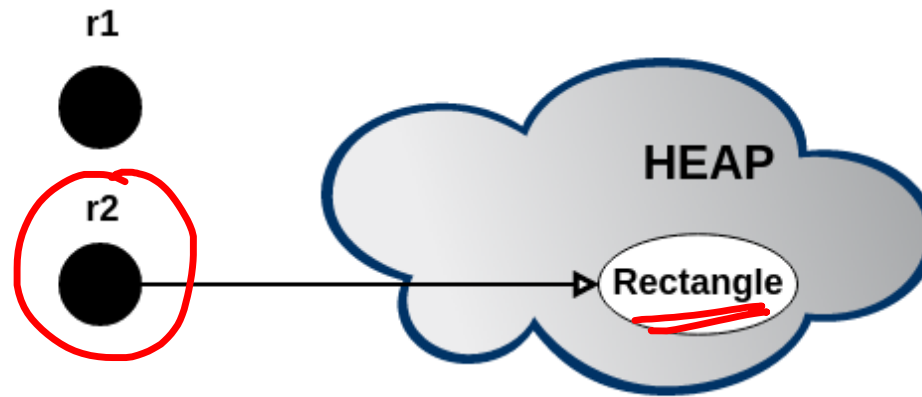
Rectangle r2 = new Rectangle( );

r1 = r2;

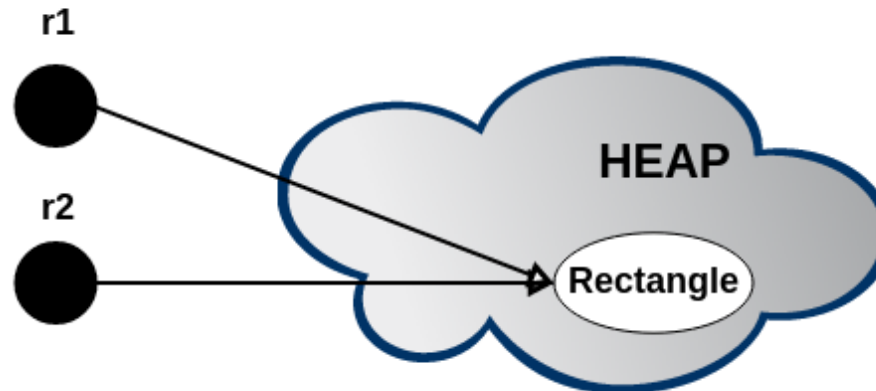
- In this case, both r1 and r2 point to same object in memory.

# What is a reference ?

```
✓ Rectangle r1;  
✓ Rectangle r2 = new Rectangle();
```



```
r1 = r2;
```



# Using references to access members

- In java, we can use dot operator(.) to access members of a class.
- The dot operator is applied on the reference variable.
- For example, in case of rectangle, `rect.getArea( )` would invoke "getArea( )" method of Rectangle object, which is referred to using the reference "rect"
- Similarly, we can access variables.
- `rect.length = 10;` would assign value 10 to a variable called length for the object pointed by reference rect.
- `System.out.println(rect.width);` would print value of width.

# Using references to access members

- Note that, again, the rules of access apply here.
- You can only invoke those members which are visible depending upon the access level definition of member.
- There are four levels of access namely **private**, **protected**, **public**, and **default**.

# Constructors

---

Constructors, are special methods which are used to create and initialize objects of a class.

---

A constructor must have same name as that of class. Also, it does not have any return type.

---

There can be multiple constructors in a java program

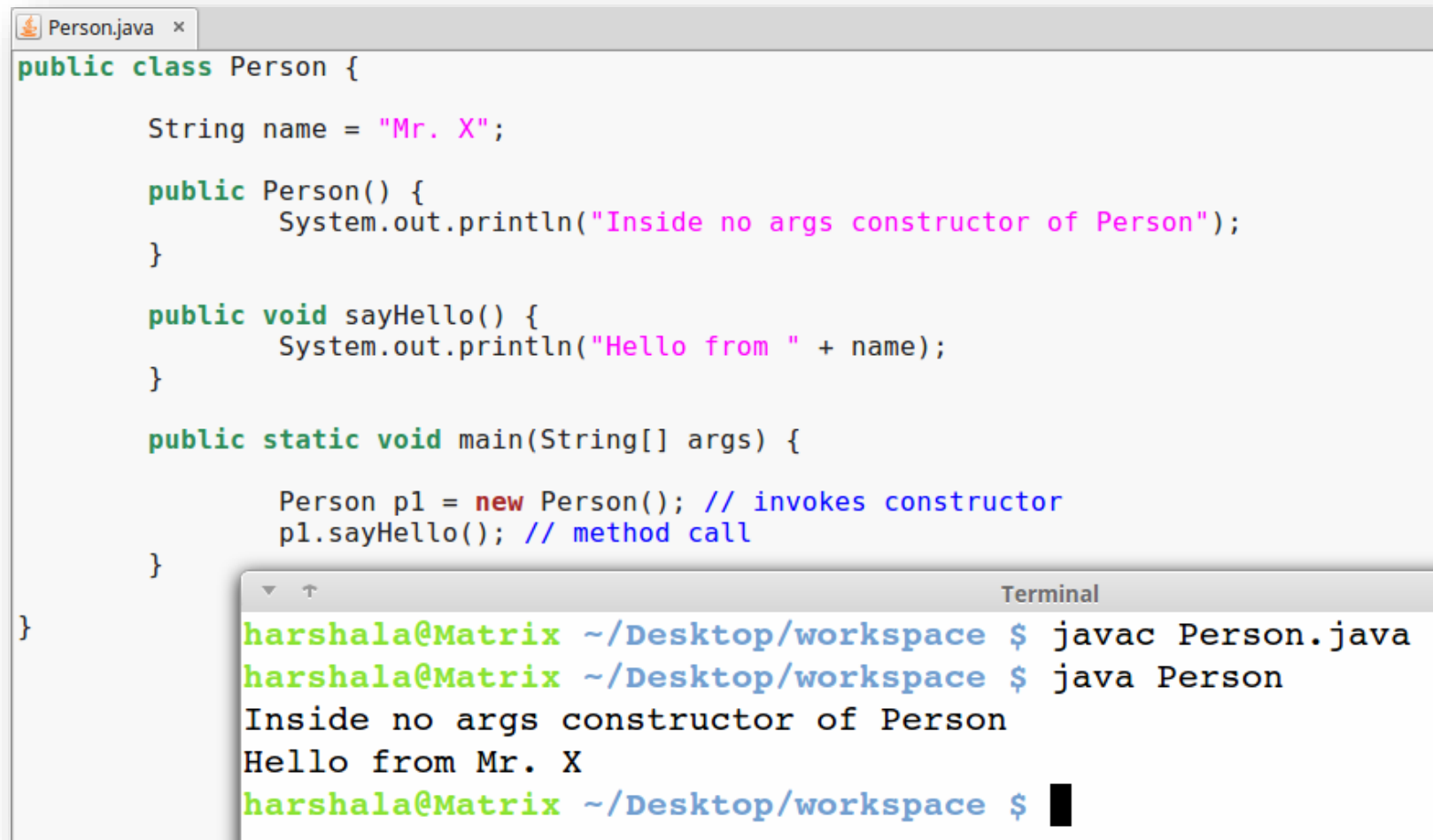
# Constructors

- A constructor in java is either a **no-argument** constructor or a **parameterized** constructor.
- If you do not provide any constructor for a class, java compiler will add a no args constructor by itself. But if you provide any other constructor, then you have to add the no args constructor yourself.

```
class Rectangle {  
    int length;  
    int width;  
    Rectangle(int length, int width) {  
    }  
}
```

```
Rectangle rect1 =  
    new Rectangle(10, 20);  
Rectangle rect2 =  
    new Rectangle();
```

# Example-1



```
Person.java x
public class Person {
    String name = "Mr. X";

    public Person() {
        System.out.println("Inside no args constructor of Person");
    }

    public void sayHello() {
        System.out.println("Hello from " + name);
    }

    public static void main(String[] args) {
        Person p1 = new Person(); // invokes constructor
        p1.sayHello(); // method call
    }
}

Terminal
harshala@Matrix ~/Desktop/workspace $ javac Person.java
harshala@Matrix ~/Desktop/workspace $ java Person
Inside no args constructor of Person
Hello from Mr. X
harshala@Matrix ~/Desktop/workspace $
```



# Example-2

```
Person.java x
public class Person {
    String name = "Mr. X";

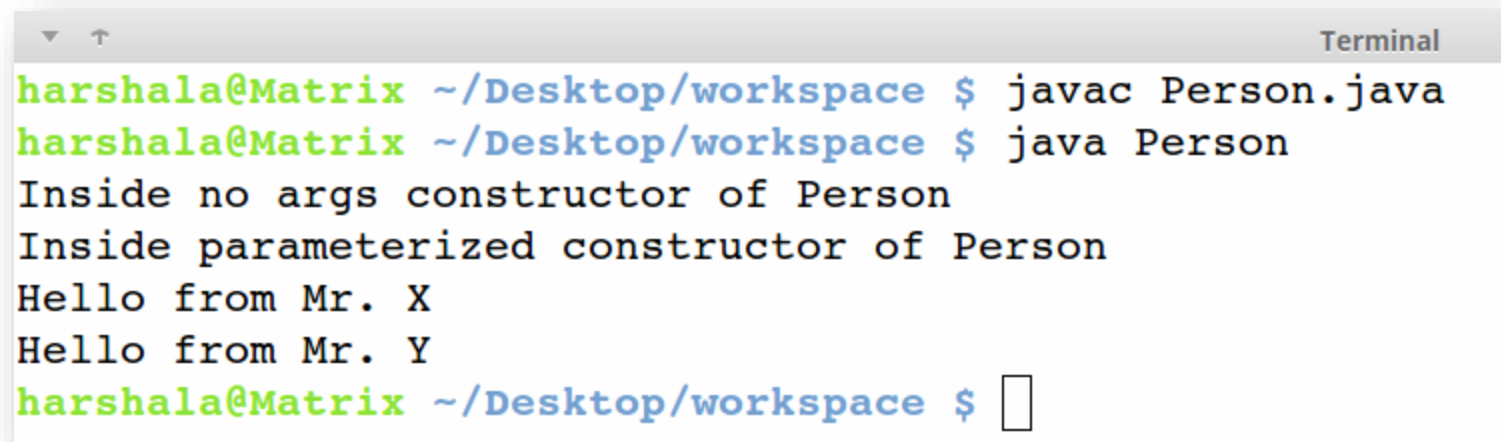
    public Person() {
        System.out.println("Inside no args constructor of Person");
    }

    public Person(String n) {
        System.out.println("Inside parameterized constructor of Person");
        name = n;
    }

    public void sayHello() {
        System.out.println("Hello from " + name);
    }

    public static void main(String[] args) {
        Person p1 = new Person(); // create object using no args constructor
        Person p2 = new Person("Mr. Y"); // create object using parameterized
        p1.sayHello();
        p2.sayHello();
    }
}
```

# Example-2 Out

A terminal window titled "Terminal" with a standard Linux prompt. The user 'harshala@Matrix' is in the directory '~/Desktop/workspace'. They run 'javac Person.java' and then 'java Person'. The output shows two constructor calls: one with no arguments and one with a parameter, followed by two "Hello from Mr." messages.

```
harshala@Matrix ~/Desktop/workspace $ javac Person.java
harshala@Matrix ~/Desktop/workspace $ java Person
Inside no args constructor of Person
Inside parameterized constructor of Person
Hello from Mr. X
Hello from Mr. Y
harshala@Matrix ~/Desktop/workspace $
```

# Variable Hiding

- In the previous example, second constructor has following statement for assignment:

`name = n;`

- Where *name* is instance variable and *n* is a local variable (A parameter is a local variable - only accessible inside that method).
- Here, name of instance variable and name of local variable is different.

# Variable Hiding

- But if name of the parameter is same as name of the instance variable, then what happens? Consider the constructor signature as:

```
public Person(String name){  
    }  
}
```

- Now, *name* is also an instance variable and *name* is a local variable. This is called as variable hiding.
- When such a scenario occurs, inside the method “name” would represent the local variable and is said to hide the instance variable.
- **So the question is, how do we assign the value of parameter *name* to the instance variable *name*?**

# What is *this* ?

- Well, to answer *that* question in the *this* context of java..
- **this** is a keyword in java which represents the current instance/object context of that class.
- In the current example, inside the constructor we are facing problem of single name for parameter and instance variable. This can be solved using *this* keyword.
- Exercise: Count how many **this** on *this* page 😊

# With and without *this*

- When we say :
- name, it would mean parameter
- this.name, it would mean instance variable.
- In general this.something, or this.doSomething() would refer to the variable "something" or method "doSomething" for that object instance.

# An example

```
Person.java x
public class Person {
    String name = "Mr. X";

    public Person() {
        System.out.println("Inside no args constructor of Person");
    }

    public Person(String name) {
        System.out.println("Inside parameterized constructor of Person");
        this.name = name;
    }

    public void sayHello() {
        System.out.println("Hello from " + name);
    }

    public static void main(String[] args) {
        Person p1 = new Person();
        Person p2 = new Person("Mr. Y");
        p1.sayHello();
        p2.sayHello();
    }
}
```

# An example

- Note that, **this** keyword is only required when there is any ambiguity.
- For example, in the method sayHello(), it is perfectly clear to compiler that name is instance variable. So this is not required (You can still go ahead and use this.name in sayHello() method, but it is not required by compiler)



# Another use of *this*

- It is also possible to invoke one constructor from other !
- For example, in the person class, we have two constructors – a no args and a parameterized constructor. It is possible to invoke the parameterized constructor from no-args constructor.

# Example of constructor call

```
Person.java x
public class Person {
    String name;

    public Person() {
        this("Mr. X");
        System.out.println("Inside no args constructor of Person");
    }

    public Person(String name) {
        System.out.println("Inside parameterized constructor of Person");
        this.name = name;
    }

    public void sayHello() {
        System.out.println("Hello from " + name);
    }

    public static void main(String[] args) {
        Person p1 = new Person();
        Person p2 = new Person("Mr. Y");
        p1.sayHello();
        p2.sayHello();
    }
}
```

Annotations in the code:

- A red circle highlights the `String name;` declaration.
- A red bracket is next to the `public Person() {` block.
- A red arrow points from the text "Invoke parameterized constructor" to the `this("Mr. X");` line.
- A red checkmark is next to the `Person p1 = new Person();` line in the `main` method.

`this(-----);`

`this();` ✓

X `this(name);`

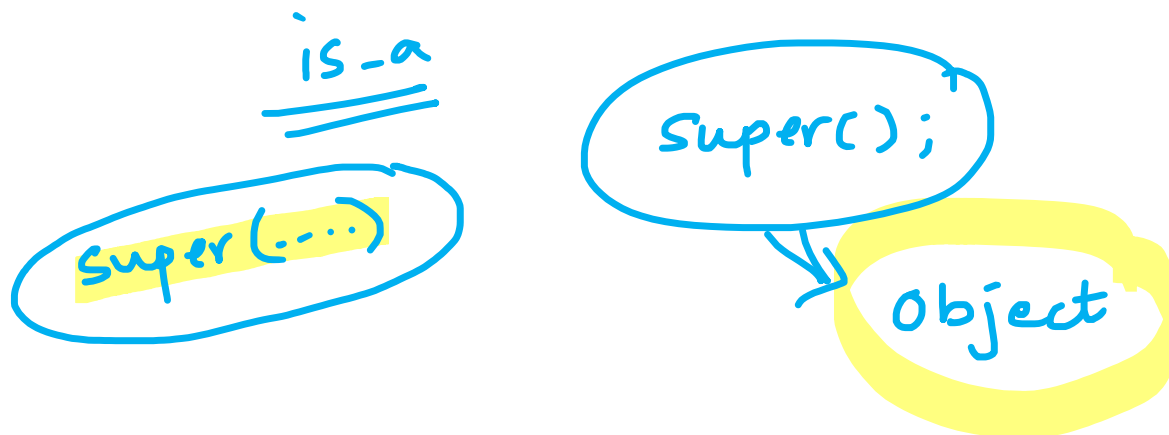
# Some points to note

- When using this(..) to invoke another constructor, there are some points which need to be taken care of.
  - ✘ The parameters to be passed can be variables only if they are passed as parameters themselves in the first constructor or defined locally.
  - For example, when no arg constructor invokes parameterized constructor, it passes "Mr X" as a name to it. But it cannot pass name as argument because name is not yet defined for this object. Remember that object is not yet created.
- 
- ```
person (String name) {  
    * this (name, id);  
    this.name = name;  
}  
person (string name, int id) {
```

# Some points to note

- ✱ • Another point to remember is that if you are using this(..), then it should be the **first statement** present in the constructor. You cannot have this(..) called after some other statement.
- ✱ • If you are not using this(..), implicit call to super class constructor is first line..

```
public Person () {
```



# A Rectangle example

Rectangle.java

```
public class Rectangle {  
    private int length, width;  
    public Rectangle() {  
        this(1, 1);  
    }  
    public Rectangle(int length, int width) {  
        this.length = length;  
        this.width = width;  
    }  
    public int getLength() {  
        return length;  
    }  
    public int getWidth() {  
        return width;  
    }  
    public void setWidth(int width) {  
        this.width = width;  
    }  
    public void setLength(int length) {  
        this.length = length;  
    }  
}
```

Handwritten annotations:

- state / Data (points to `private int length, width;`)
- No-arg (points to `public Rectangle() {`)
- param. (points to `public Rectangle(int length, int width) {`)
- getter (points to `public int getLength() {`)
- getter (points to `public int getWidth() {`)
- setter (points to `public void setWidth(int width) {` and `public void setLength(int length) {`)

# A Rectangle example

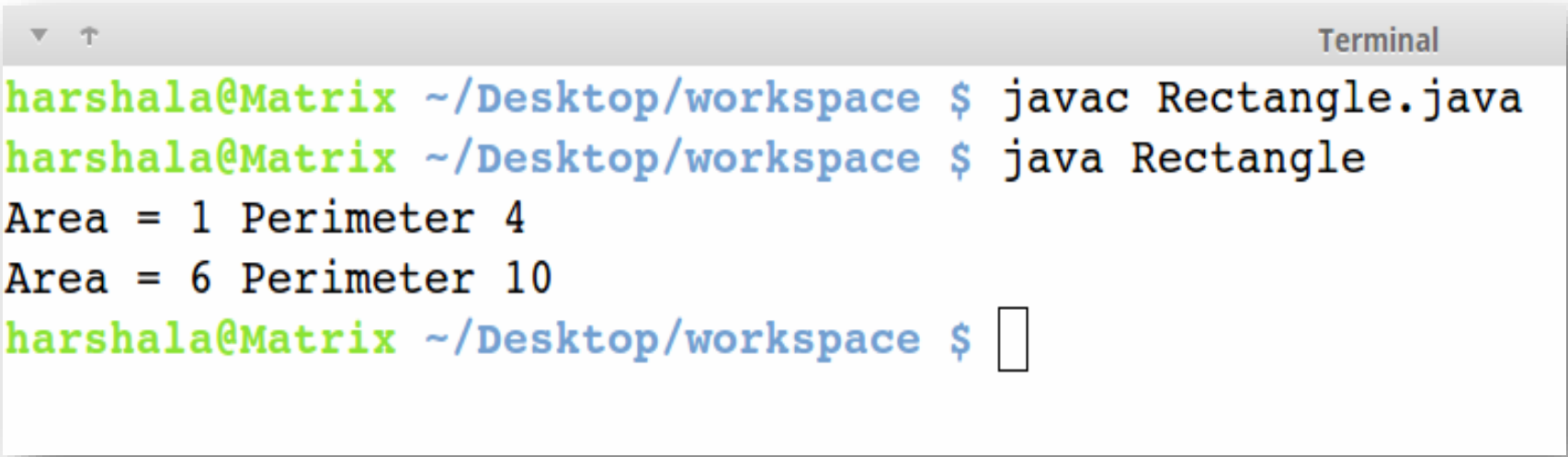
Method/behaviour

```
{
    public int getArea() {
        return length * width;
    }

    public int getPerimeter() {
        return 2 * (length + width);
    }

    public static void main(String[] args) {
        ✓ Rectangle r1 = new Rectangle();
        System.out.println("Area = " + r1.getArea() + " Perimeter "
            + r1.getPerimeter());
        ✓ Rectangle r2 = new Rectangle(2, 3);
        System.out.println("Area = " + r2.getArea() + " Perimeter "
            + r2.getPerimeter());
    }
}
```

# Output



A terminal window titled "Terminal" with a standard macOS-style title bar (red, yellow, green buttons). The terminal shows the following commands and output:

```
harshala@Matrix ~/Desktop/workspace $ javac Rectangle.java
harshala@Matrix ~/Desktop/workspace $ java Rectangle
Area = 1 Perimeter 4
Area = 6 Perimeter 10
harshala@Matrix ~/Desktop/workspace $
```

# Garbage Collector

- Java frees developer from burdon of destructing objects. JVM has a component called Garbage Collector which manages object deletion for you !
- Note that, however, you do not have control on when JVM runs Garbage Collector. In fact there is no gurantee that garbage collection will happen.
- All objects have a method called finalize(). Before removing any object, garbage collector invokes this method. But again remember: there is no guarantee when this method would be called. In fact, there is no guarantee that this method would be called at all.
- So remember: Do not rely your business logic on finalize method.

Rectangle r1 ;





# Overloading of Methods

- In java, it is possible to have two methods with the same name as far as their parameter list is different.
- Examples of overloaded methods:

```
public double getArea(double r);
```

```
public double getArea(double length, double width);
```

```
{ public double getArea( double r );  
  public double getArea( double radius );
```

# Example

```
Shape.java x
public class Shape {
    public static void main(String[] args) {
        Shape shape = new Shape();
        System.out.println("circle area = " + shape.getArea(10));
        System.out.println("square area = " + shape.getArea(10, 10));
    }

    public double getArea(double r) {
        return Math.PI * r * r;
    }

    public double getArea(double length, double width) {
        return length * width;
    }
}
```

Terminal

```
harshala@Matrix ~/Desktop/workspace $ javac Shape.java
harshala@Matrix ~/Desktop/workspace $ java Shape
circle area = 314.1592653589793
square area = 100.0
harshala@Matrix ~/Desktop/workspace $
```


Handwritten annotations in blue:

- ① points to the `getArea(10)` call in the main method.
- ② points to the `getArea(10, 10)` call in the main method.
- ① points to the `getArea(double r)` method signature.
- ② points to the `getArea(double length, double width)` method signature.

# Objects as Parameters

- It is possible (in fact, very common) to pass objects as arguments to methods.
- For example, following method returns if a person is adult:

```
public boolean isAdult(Person p){  
    int age = p.getAge();  
    if(age > 18){  
        return true;  
    } else {  
        return false;  
    }  
}
```

- A useful program for movie theatres, some might say.. 

# Objects as return type

- Again, it is very common to use Object as return values. A typical example is writing a method which creates an instance of object and returns it:

```
public Person getInstance(String name, int age){  
    Person p = new Person();  
    setName(name);  
    p.setAge(age);  
    return p;  
}
```

# How to count ?

- Suppose you want to count how many objects of a particular class your program is creating.
- How would you do it ? You need to define a variable in the class which stores the count of number of objects created. But where would you increment this counter?
- Each object has a different copy of instance variables. So each time a object is created, a different "count" variable is created for that object.

# How to count ?

1) Constructor:

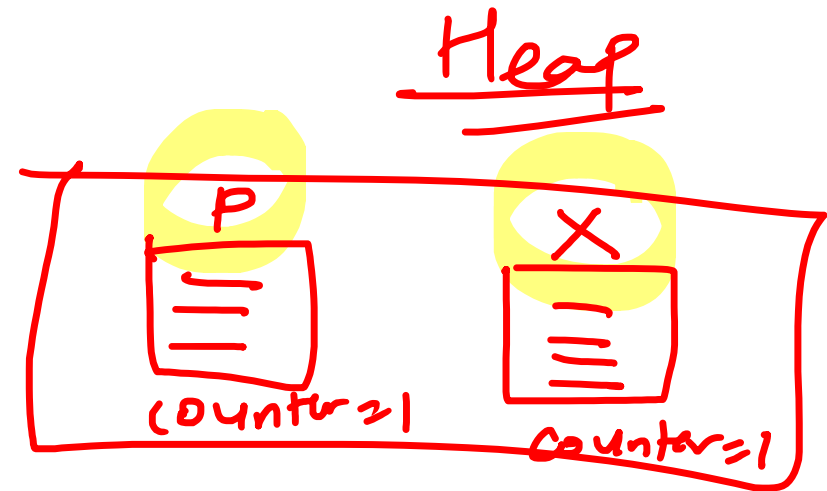
```
class Person {  
    Person() {  
        int counter;  
        counter++;  
    }  
}
```

2) class Person {  
 ✓ int counter;  
 person() {  
 counter++;  
 }  
}



main

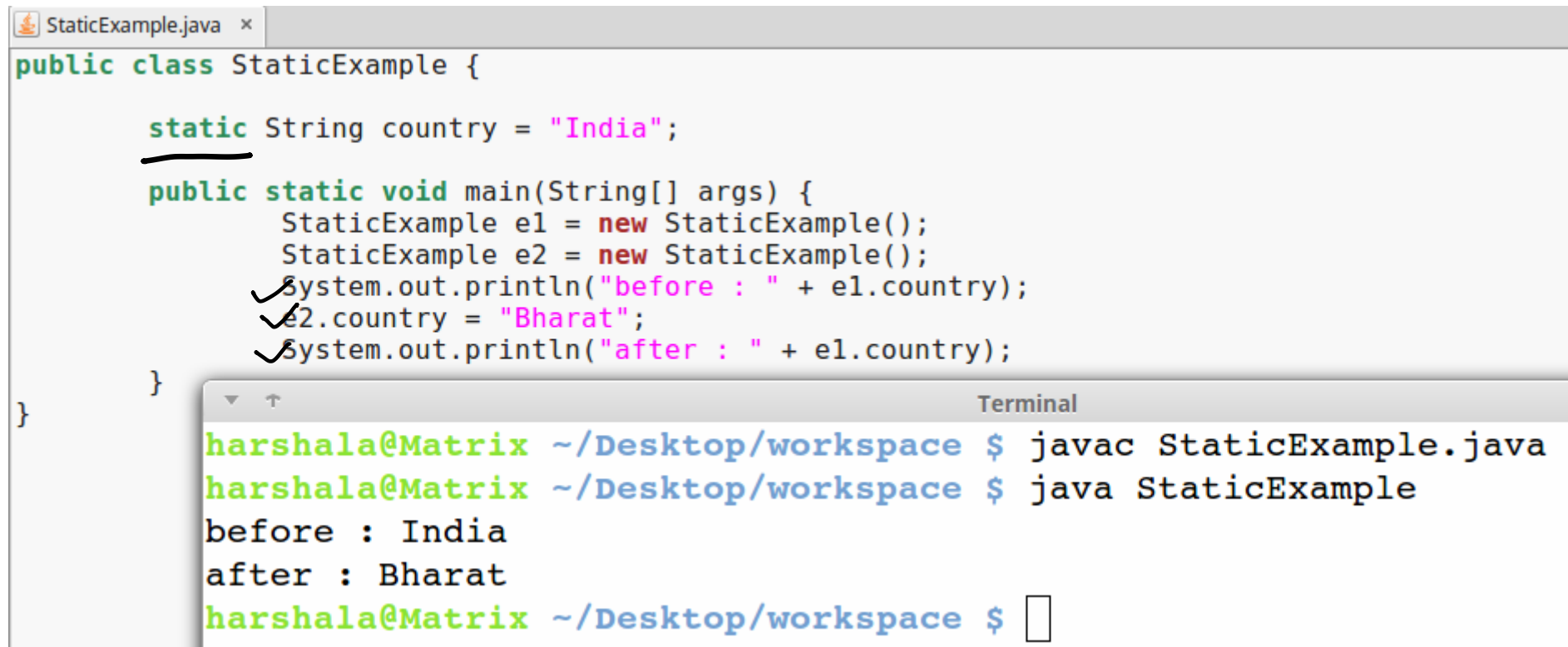
```
int counter;  
✓ Person P = new Person();  
=  
✓ Person X = new Person();
```



# *static* to the rescue

- Java has a keyword called `static`, which is `associated with class` instead of object.
- When you mark any class level variable as static, there is single copy of that variable per class. No matter how many objects you create, all objects would have exactly same value for this variable. If you change value in one instance, value is updated in all instances !
- Static can also be applied to methods. When used for a method, that method belongs to class and not object. In other words, difference between static and non-static (instance) methods is that static methods cannot be invoked using object references (or using *this* keyword. Remember *this* ? .. you better..)

# Example



```
StaticExample.java x
public class StaticExample {

    static String country = "India";

    public static void main(String[] args) {
        StaticExample e1 = new StaticExample();
        StaticExample e2 = new StaticExample();
        ✓ System.out.println("before : " + e1.country);
        ✓ e2.country = "Bharat";
        ✓ System.out.println("after : " + e1.country);
    }
}
```

```
Terminal
harshala@Matrix ~/Desktop/workspace $ javac StaticExample.java
harshala@Matrix ~/Desktop/workspace $ java StaticExample
before : India
after : Bharat
harshala@Matrix ~/Desktop/workspace $
```



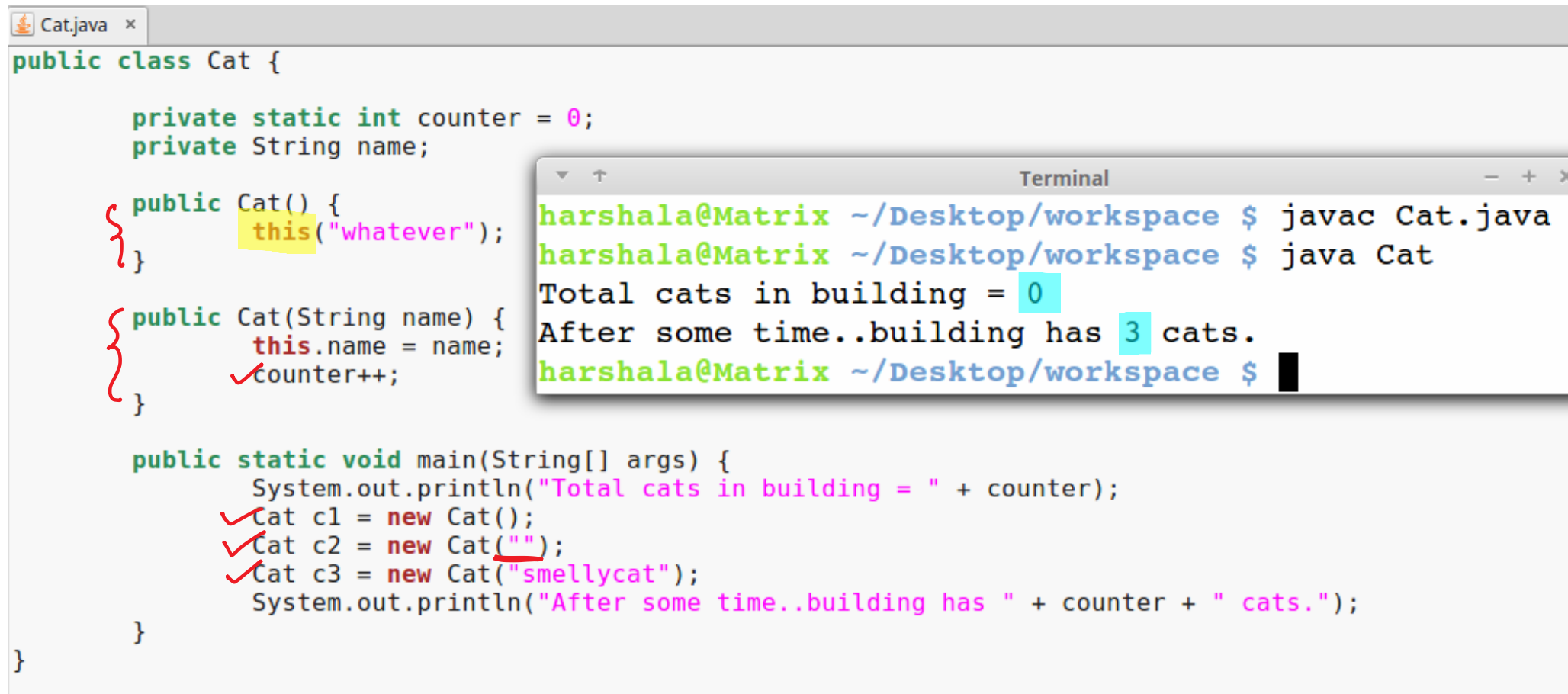
# Example

- As we can see, if we change value of static variable in one object, it is also reflected in another object.
- But note that although what we have done is syntactically correct, it is not correct way to do the things. Since static variables are class level, they are not tied with with any instance and it is incorrect to code as such.
- Java allows (and expects) us to directly invoke the variable from Class like this: `System.out.println(StaticExample.country);`

# Back to counter

- So how would static be used to count instances ?
- Simple, just create a static variable and increment it whenever any object is created..that is whenever any constructor is invoked !

# Counter Attack !



```
public class Cat {  
    private static int counter = 0;  
    private String name;  
  
    public Cat() {  
        this("whatever");  
    }  
  
    public Cat(String name) {  
        this.name = name;  
        counter++;  
    }  
  
    public static void main(String[] args) {  
        System.out.println("Total cats in building = " + counter);  
        Cat c1 = new Cat();  
        Cat c2 = new Cat("");  
        Cat c3 = new Cat("smellycat");  
        System.out.println("After some time..building has " + counter + " cats.");  
    }  
}
```

```
harshala@Matrix ~/Desktop/workspace $ javac Cat.java  
harshala@Matrix ~/Desktop/workspace $ java Cat  
Total cats in building = 0  
After some time..building has 3 cats.  
harshala@Matrix ~/Desktop/workspace $
```

# Some points to remember

- When using static methods, following must be taken into account:
  - Static methods can only call other static methods
  - Static methods can only access static variables
  - Static methods cannot use this or super keywords

# Inheritance in Java

- Strict use of IS-A relationship
- No direct support for multiple inheritance

# IS-A relationship

- In java, inheritance must follow an IS-A relationship along with code reuse. An IS-A relationship means that when we want to check if class A can inherit from class B, it should satisfy the statement class A IS-A class B. An example is a class called Player and class called CricketPlayer.
- A CricketPlayer IS-A Player and hence, the two classes qualify for inheritance

# Multiple Inheritance

- **Java does not support Multiple Inheritance.**
- In java, it is NOT possible to inherit from more than one class.
- We will soon learn a concept called Interfaces. Java allows a class to *implement* more than one interface. In a very limited sense, you can say that using multiple interfaces, java allows you to emulate multiple inheritance.

# Inheritance in Java

- As already discussed, in java inheritance is achieved by satisfying "IS-A" relationship. This relationship is expressed in java using keyword **extends**. The syntax for this is:

```
class subclass extends superclass {  
    //Subclass body;  
}
```



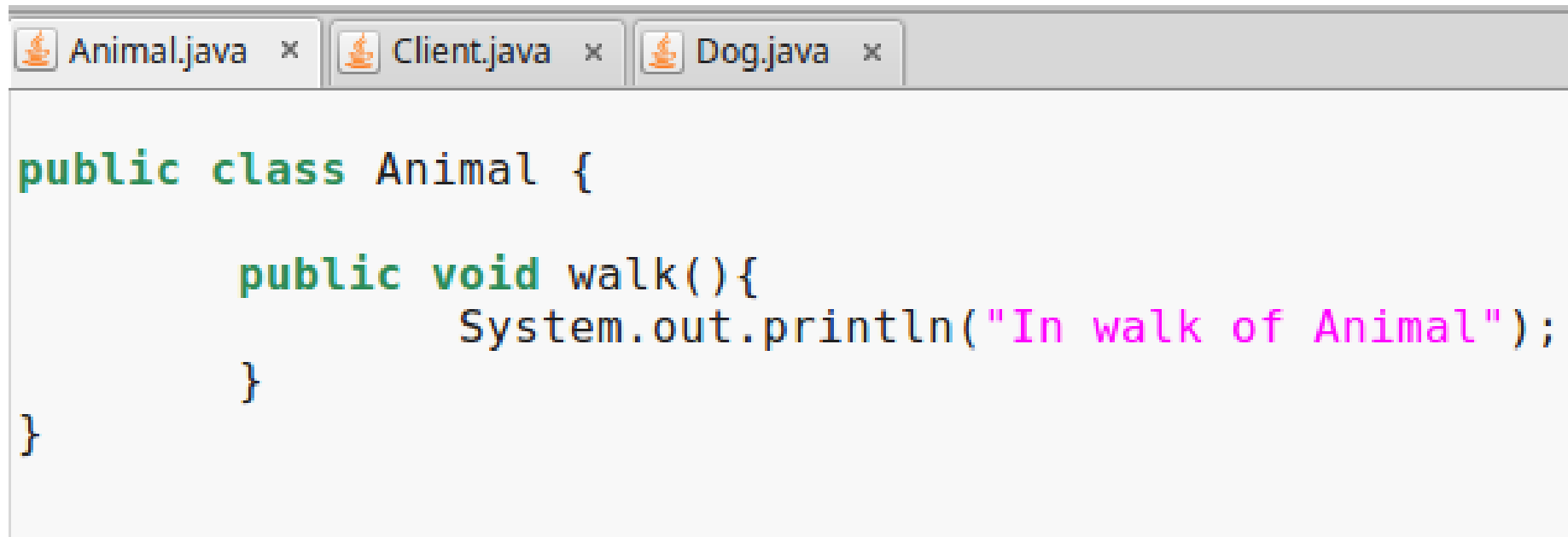
# Example of Inheritance

- We are going to consider a simple example of **Animal** and **Dog** classes.
- A **Dog IS-A Animal** and hence we can have Animal as superclass and Dog as subclass.
- In our example, we are going to have a simple method called “walk”. We will see a basic example, and then see some additions to it.

# Dog and Animal

- We have three classes in our example. Each is in separate file.
- First class is Animal class, which has a method called walk
- Second class is Dog class, which extends Animal and has a method called bark
- Third is a class called Client, which has main method.

# Animal Class



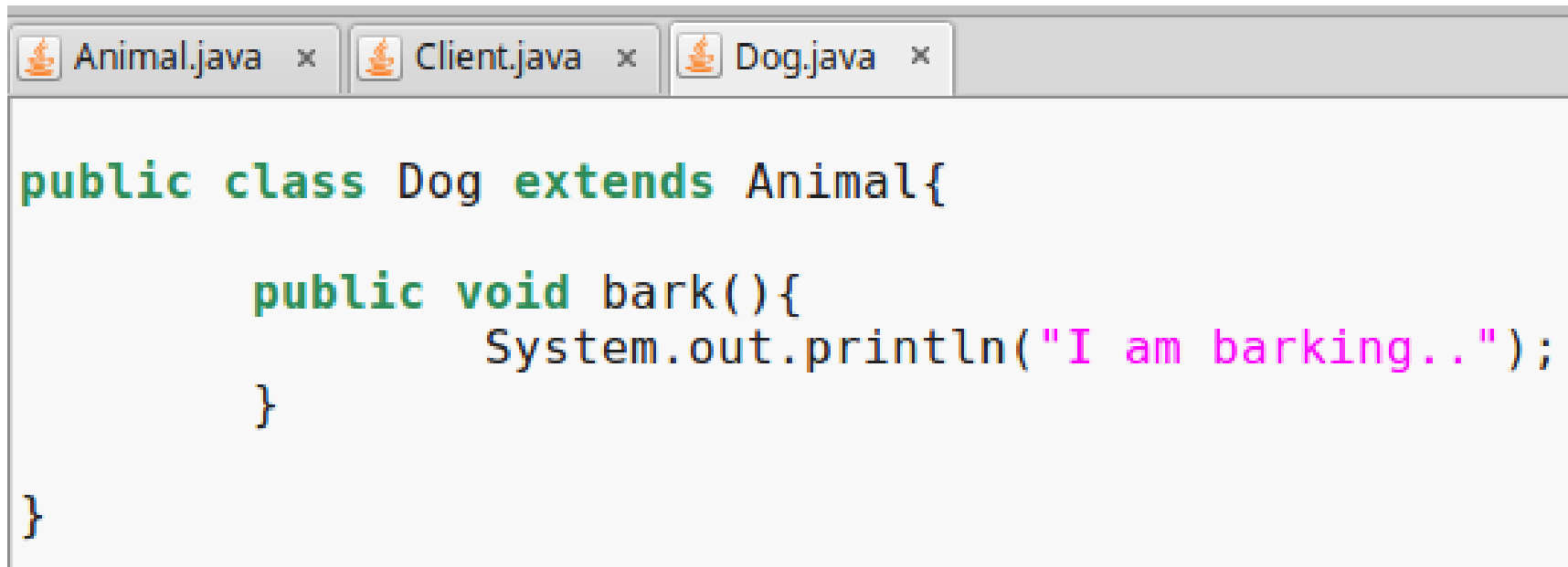
The screenshot shows a Java IDE window with three tabs: Animal.java, Client.java, and Dog.java. The Animal.java tab is active, displaying the following code:

```
public class Animal {  
    public void walk(){  
        System.out.println("In walk of Animal");  
    }  
}
```

# Animal Class

- As we can see, this is a typical public class, in file called Animal.java. It has a single method called walk().

# Dog Class



The screenshot shows a Java IDE window with three tabs: Animal.java, Client.java, and Dog.java. The Dog.java tab is active, displaying the following code:

```
public class Dog extends Animal{  
  
    public void bark(){  
        System.out.println("I am barking..");  
    }  
  
}
```

# Dog Class

- Dog class uses **extends** keyword to inherit from Animal class.
- Due to this, Dog class has now two methods – Method called bark() which is its own method, and method called walk(), which is a public method in Animal class and hence inherited.
- Only those methods from superclass **can be inherited** which are **public** or **protected**.

# Client Class

```
Animal.java x Client.java x Dog.java x

public class Client {

    public static void main(String[] args) {
        ✓ Animal a = new Animal();
        ✓ a.walk();
        ✓ Dog d = new Dog();
        d.walk();
        ✓ d.bark();
    }
}
```

# Client Class

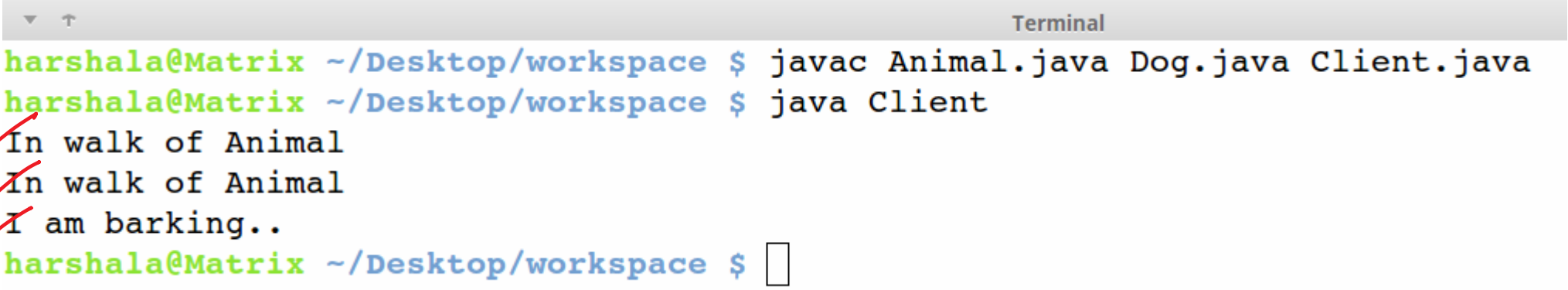
- We have created a separate class called "Client" for adding main function. Theoratically, we could add main to any of the Dog or Animal classes, but it is good practice to keep it seperate unless any of Animal or Dog must be starting points of your application.



# Client Class

- In client class, we are creating two objects – one of Animal and one of Dog. Note that, we can call both “walk” and “bark” methods on dog object.

# Output



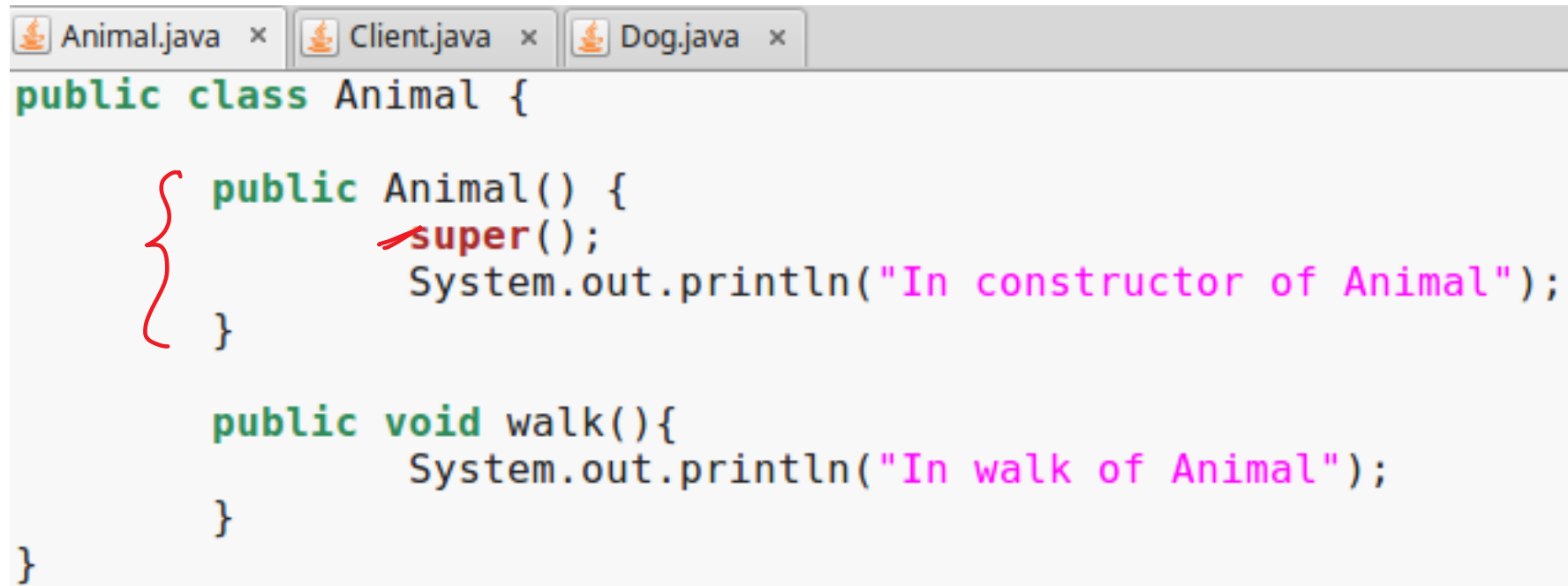
```
▼ ↑ Terminal
harshala@Matrix ~/Desktop/workspace $ javac Animal.java Dog.java Client.java
harshala@Matrix ~/Desktop/workspace $ java Client
✓ In walk of Animal
✓ In walk of Animal
✓ I am barking..
harshala@Matrix ~/Desktop/workspace $
```

A terminal window titled "Terminal" with a dropdown arrow and an up arrow icon. It shows the execution of Java code. The prompt is "harshala@Matrix ~/Desktop/workspace". The first command is "javac Animal.java Dog.java Client.java". The second command is "java Client". The output consists of three lines, each preceded by a red checkmark: "In walk of Animal", "In walk of Animal", and "I am barking..". The prompt "harshala@Matrix ~/Desktop/workspace \$" is followed by a small square cursor.

# Order of constructors

- When using inheritance, the constructors are called in top-down fashion. Constructor of superclass is called first and the child class constructor is called last. This can be easily seen by adding constructors to our classes

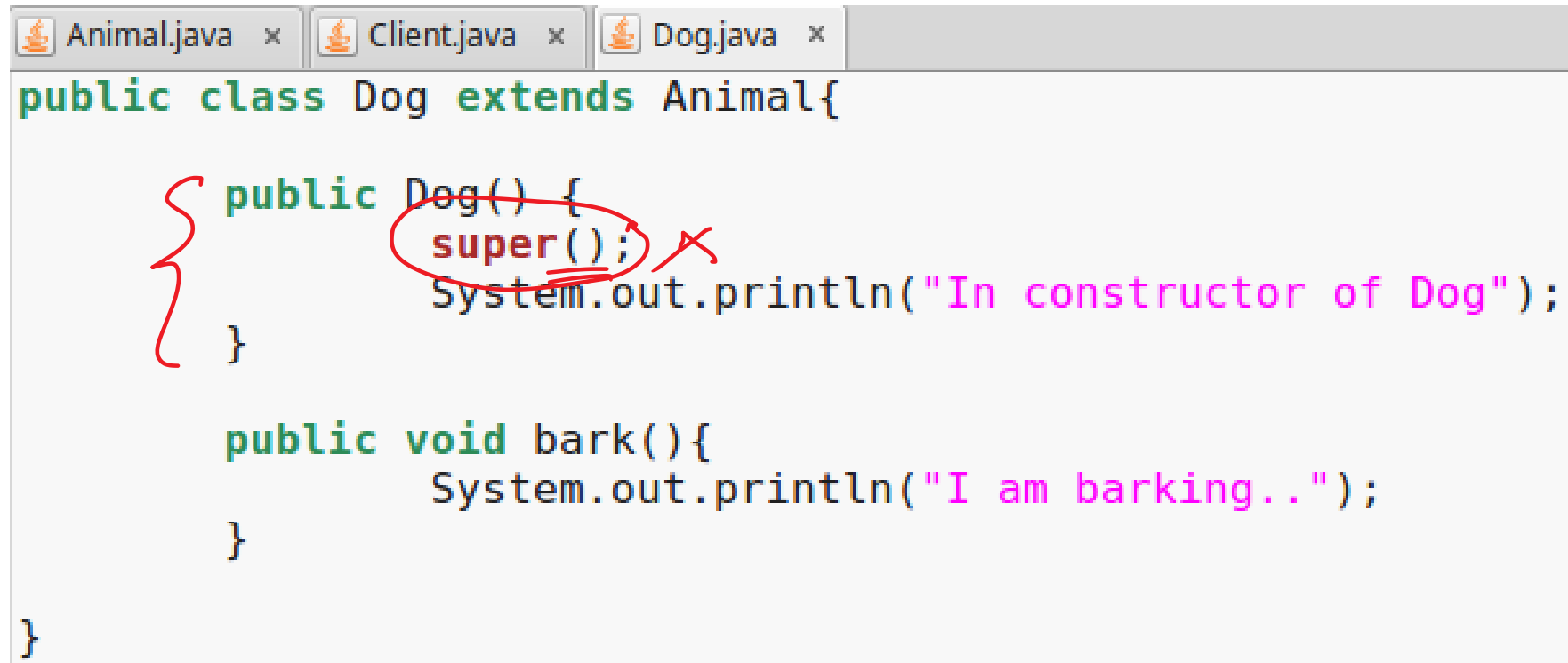
# Animal Class



```
Animal.java x Client.java x Dog.java x
public class Animal {
    public Animal() {
        super();
        System.out.println("In constructor of Animal");
    }

    public void walk(){
        System.out.println("In walk of Animal");
    }
}
```

# Dog Class



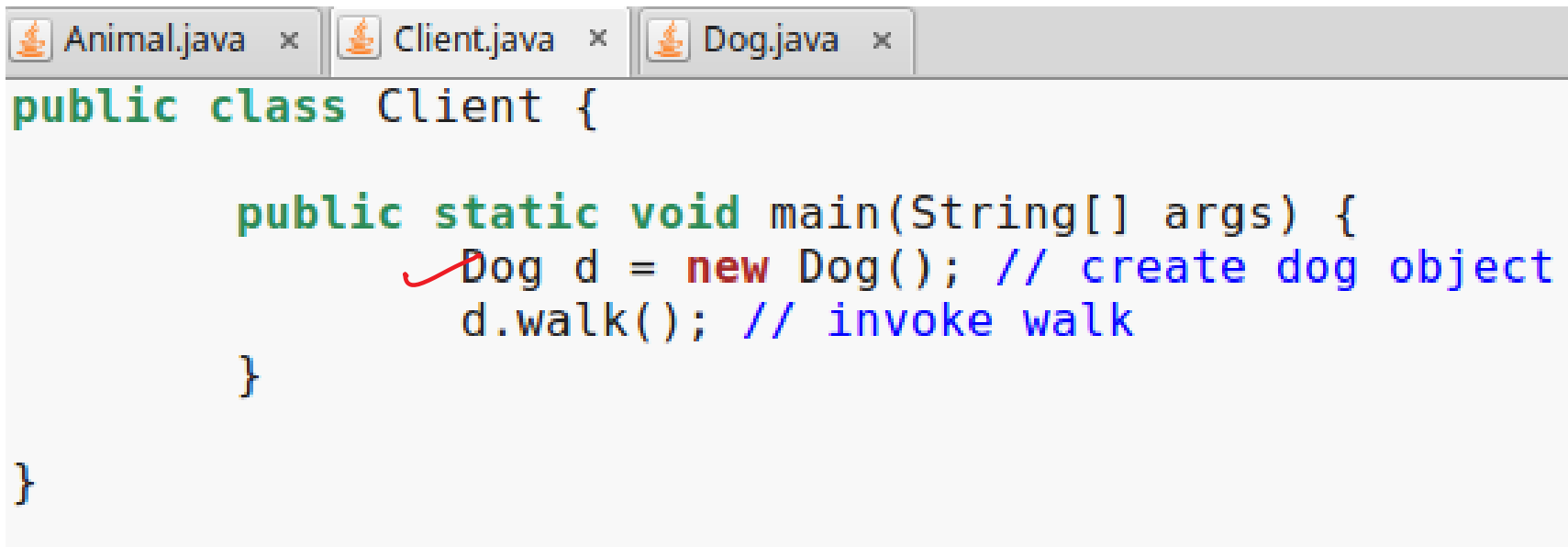
```
Animal.java x Client.java x Dog.java x
public class Dog extends Animal{

    public Dog() {
        super();
        System.out.println("In constructor of Dog");
    }

    public void bark(){
        System.out.println("I am barking..");
    }

}
```

# Client Class



The screenshot shows a Java IDE window with three tabs: Animal.java, Client.java, and Dog.java. The Client.java tab is active, displaying the following code:

```
public class Client {  
  
    public static void main(String[] args) {  
        ✓ Dog d = new Dog(); // create dog object  
        d.walk(); // invoke walk  
    }  
  
}
```

# Output

```
Terminal
harshala@Matrix ~/Desktop/workspace $ javac Animal.java Dog.java Client.java
harshala@Matrix ~/Desktop/workspace $ java Client
In constructor of Animal
In constructor of Dog
In walk of Animal
harshala@Matrix ~/Desktop/workspace $
```

# What is super() ?

- Like **this**, **super** is a special keyword in java.
- **this** means current object in java
- **super** means parent object of current object.
- **this()** means invoking no-args constructor of current object
- **super()** means invoking no-args constructor of parent object. We can also pass arguments to invoke parameterized constructors for super class.



# Calling super() in Animal ?

- Calling super() in constructor of Dog would invoke constructor of Animal.
- But what about call to super() in constructor of Animal ? Well, it calls constructor of Object class. In java every class implicitly extends object class if explicit extends is not used. Calling super() in Animal constructor would call constructor of Object class to create base object of class Object !

# super() is optional

- Well, after all that super calls, here's a thing to note: Call to super() is optional ! Java compiler would add a call to super() itself, if you do not add it. The only reason to use these calls in our programs was to make sure we understand what the concept is.
- Also, remember that for any constructor, the first line is either call to **super()** or call to **this()**. If first line is anything else, constructor would implicitly add super() during compile time.

# Overriding methods

- Java also supports **overriding** of methods from parent class. Overriding refers to re-implementing the parent class method in child class. When re-implementing, the method name and argument list must be same, and return type must be same as defined in parent or it's subclass.
- In our example, we can override walk method from Animal class in Dog class.
- The only change is in class Dog. Class Animal and class Client do not change !

# Class Dog with overriding

Annotation

```
Animal.java x Client.java x Dog.java x
public class Dog extends Animal {
    * @Override
    public void walk() {
        ✓ super.walk();
        ✓ System.out.println("In walk of Dog");
    }

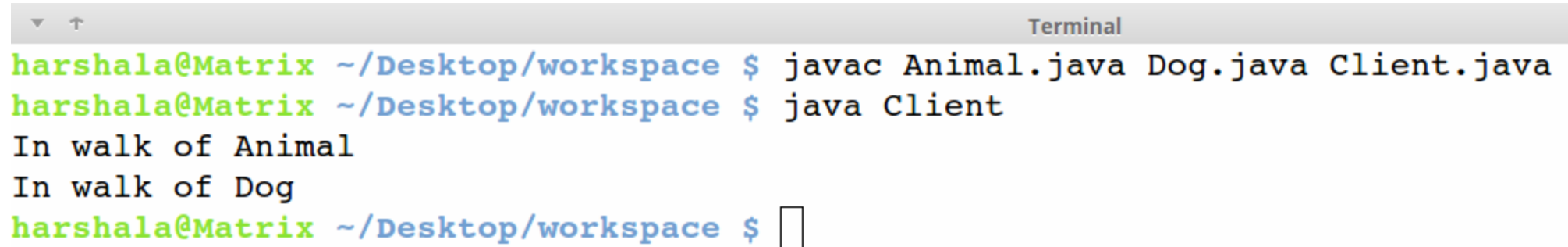
    public void bark() {
        System.out.println("I am barking..");
    }
}
```

@Override  
public void walk() {  
} - - - -

# Overriding - Details

- @Overriding - This is an annotation. An annotation is a metadata about a method for the compiler. In this case annotation is Overriding and it informs the compiler that this method is overridden. We will discuss annotations in some time.
- ✓ • super.walk() - This allows you to invoke "walk" of parent class. This is not mandatory call, but added here just to show how it can be done.

# Overriding - Output




A terminal window titled "Terminal" with a grey header bar containing a dropdown arrow and an upward arrow. The terminal shows the following text:

```
harshala@Matrix ~/Desktop/workspace $ javac Animal.java Dog.java Client.java
harshala@Matrix ~/Desktop/workspace $ java Client
In walk of Animal
In walk of Dog
harshala@Matrix ~/Desktop/workspace $
```

# A *final* fantasy

- No, this is not regarding that online role playing game... This is regarding the keyword **final** in java..
- In java a keyword final can be used at different positions with different meanings. These are - variable, method modifier, method argument modifier, class modifier. Let us see these one by one.

# final variables

- As we have already seen, in java, when a variable is marked as final, you cannot change its value. For example consider:
- `final int a = 10;`
-  `a = 15; // compile time error`



# final methods

- We have seen how to override a method. But what if you do not want a method to be overridden ?
- Marking a method as final prevents it from getting it overridden. For example, if we mark "walk" as final in Animal as:
- ✓ `public final void walk(){ };`
- Then it would not be possible to override it in Dog (Try it out !)

# Final arguments to method

*add(<sup>final</sup>int a, final int b)*

- It is also possible to add final keyword to the parameter in method. This informs the user of the method that the method would not be modifying the parameter passed to it.
- Note: This has a very special usage in java for a certain scenario. When using the argument in method local classes, the argument can only be used if it is final.
- Note for the non-sleepy people: Method local classes are classes defined inside a method. And yes, you can do that kind of stuff.

# final class

- If you want to ensure that a class should never be inherited, mark it as final !
- For example, if we have

```
public final class Animal { }
```

- Then,

```
public class Dog extends Animal
```

- would be compile error.
- Note: The above statement won't make sense for now. We would understand it better it during discussion on inheritance.

# Why mark class as final ?

- Mostly, due to security concerns !
- If you want to ensure that business logic in your class is to be used as is and not to be modified, you must mark it as final.
- Also there are some data types, whose behaviour you want to freeze, For example String class in java is final.

# finalize() method

- We have already discussed this. This is reiterated here to enforce that this method is not related to final.
- This method is part of Object class and is used to release the resources before that object is destroyed. As discussed earlier, there is no gurantee when this method would be invoked by Garbage Collector. (As we have seen earlier, Garbage Collector is a component in java which is used to free memory by destroying objects.)
- Note again that, this should never be used as part of your business logic implementation, since you are never sure when this would be called.

# Abstract Methods and Class

- We have seen how to write subclass and superclass methods.
- Now consider an example of superclass as Shape and subclasses as Circle and Rectangle.
- Superclass Shape as methods `getArea()` and `getPerimeter()`.

# Abstract Methods and Class

- Both Circle and Rectangle would have their own implementation for `getArea()` and `getPerimeter()`, since both have different formula for calculating area and perimeter.
- Further, since Shape is a generic class, it does not make sense to provide any implementation for these two methods in Shape class.

# Abstract Methods and Class

- To achieve this, we can provide empty methods for `getArea()` and `getPerimeter()`.
- But if a new subclass say `Triangle` is added, and developer forgets to add implementation for these two methods, then the base implementation would be called, which is of no use.
- To ensure that every new subclass provides implementation for these methods, we need to mark them as abstract. If a method is marked as abstract, it is a rule that any subclass of that class must implement it.



# Abstract Methods and Class

- To mark a method as abstract, we do two things – First, use keyword abstract in method declaration and second, remove method body.
- For example,
- ✓ public abstract double getArea();
- public abstract double getPerimeter();
- Note that the curly braces are not provided.

# Abstract Methods and Class

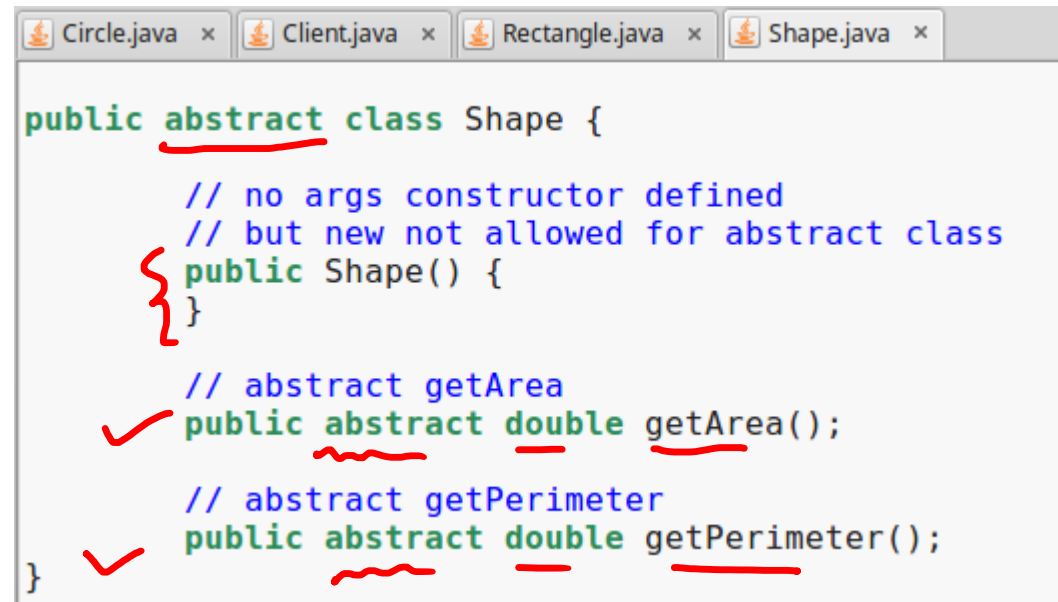
- ✓ Further, it is a rule in Java that whenever any method is marked as abstract, corresponding class must also be marked abstract.
- ✓ When we mark a class as abstract, it is not possible to create an instance of that class. Even though you can define constructor for that class, you cannot use new operator on it. In our example, Shape would be marked as abstract.

```
public abstract class Shape {  
  
    public abstract double  
        getArea();  
}
```

# Example

- In this example, we have four classes (4 files) – Shape, Circle, Rectangle, and Client.
- Shape is abstract class. Rectangle and Circle extend Shape. Client is for main method.

# Shape.java



```
Circle.java x Client.java x Rectangle.java x Shape.java x

public abstract class Shape {
    // no args constructor defined
    // but new not allowed for abstract class
    { public Shape() {
    }

    // abstract getArea
    ✓ public abstract double getArea();

    // abstract getPerimeter
    ✓ public abstract double getPerimeter();
}
```

# Circle.java

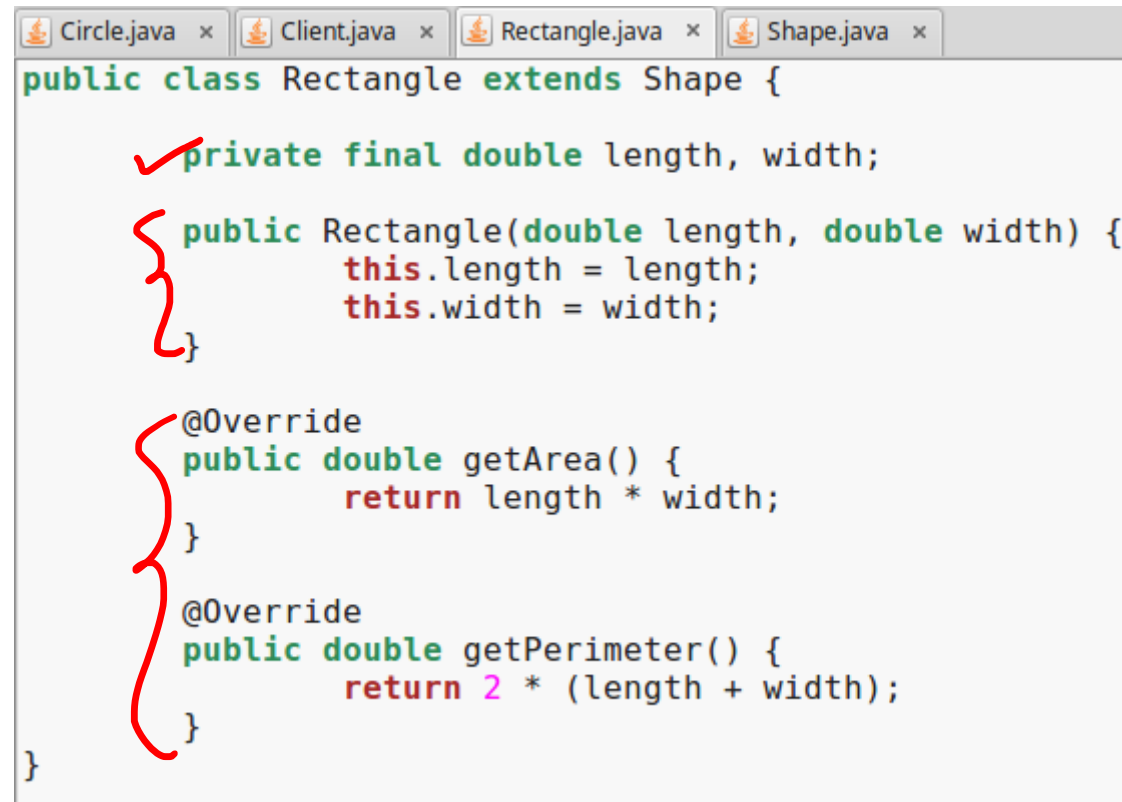
```
Circle.java x Client.java x Rectangle.java x Shape.java x
public class Circle extends Shape {
    private final double radius;

    {
    public Circle(double radius) {
        this.radius = radius;
    }

    @Override
    {
    public double getArea() {
        return Math.PI * radius * radius;
    }

    @Override
    {
    public double getPerimeter() {
        return 2 * Math.PI * radius;
    }
    }
}
```

# Rectangle.java



```
Circle.java x Client.java x Rectangle.java x Shape.java x
public class Rectangle extends Shape {

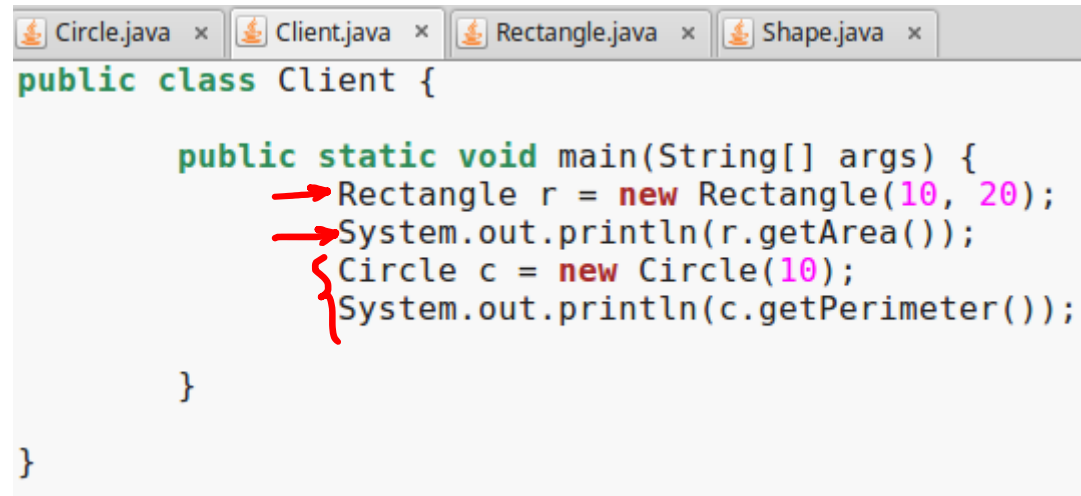
    ✓ private final double length, width;

    {
        public Rectangle(double length, double width) {
            this.length = length;
            this.width = width;
        }

        @Override
        public double getArea() {
            return length * width;
        }

        @Override
        public double getPerimeter() {
            return 2 * (length + width);
        }
    }
}
```

# Client.java



```
Circle.java x Client.java x Rectangle.java x Shape.java x
public class Client {

    public static void main(String[] args) {
        → Rectangle r = new Rectangle(10, 20);
        → System.out.println(r.getArea());
        {
            Circle c = new Circle(10);
            System.out.println(c.getPerimeter());
        }
    }
}
```

# final *and* abstract

- Note that, there is no “and” with **final** *and* **abstract**. They are opposite of each other.
- **final** is used to prevent a class from getting inherited
- **abstract** is used to ensure that a class is inherited.
- You can either mark a class as final or as abstract, but not both.



# Using superclass as reference


- So far in the examples of subclass, whenever we create an object of subclass, we use the subclass name as reference variable type.
- For example to create a rectangle, we say:
- `Rectangle r = new Rectangle(10,20);`

Reference  
type

# Using superclass as reference


- Here Rectangle on left hand side is reference type and Rectangle on right hand side refers to the actual class whose object is being created.
- It is possible to use supertype for reference variable. For example, below is correct:
  - `Shape r = new Rectangle(10,20);`
  - `Shape c = new Circle(10);`

# Using superclass as reference

- When we say
-  Shape r = new Rectangle(10,20);
- What this means is that we create a Rectangle object and point reference r of type Shape to it.
- This is perfectly legal and in fact a best practice in java. A famous principal in java is "**Program to interface**" where interface refers to the common interface (Shape in this example)

```
Shape {  
    display()  
}
```

 r.getDiaglength (Rectangle) r

 getDiaglength() {  
 }  
}

# Using superclass as reference

- A very important point to note here is that you can only invoke methods which are declared in your reference type.
- For example, if your Rectangle has a method called getDiagonalLength(), which is not defined in Shape, it is not possible to directly invoke this method from the reference.
- You would need to first cast reference to Rectangle and then invoke the corresponding method.

# Access Modifiers

- Java supports 4 types of accesses –
  - public access
  - protected access
  - private access
  - default access.
- First three are defined using appropriate access modifiers (public/protected/private). In the absence of any modifier, default access is assumed.

# Using public

- If a variable or method is marked public, it is accessible everywhere
- If a class is marked public, it can be accessed from everywhere

# Using protected

- If a variable or method is marked protected, it can be accessed by other classes in same packages and subclasses (in any package)
- This modifier is not allowed at class level

# Using private

- If a variable or method is marked private, it cannot be accessed by any code outside the class.
- It is also not possible to access a private method on any object of that class.
- ✱ • This modifier is not allowed for outer class declaration. It is however, possible to define a class within another class and mark it as private, in which case, it is only accessible inside that class.



# Using default

- In absence of any of private/protected/public, a default access is applied.
- If a variable or method is default, it can be accessed by other classes in same packages, but not outside.
- If a class is default (public not used), it can be created only from within the package.

# Using Variable Arguments

- Java supports concept of variable arguments or varargs. Using this, you do not need to know exact number of arguments for a method.
- A variable argument is defined using "..." after the data type and before variable name.
- Note that vararg is internally treated as array of that type.

# Example

```

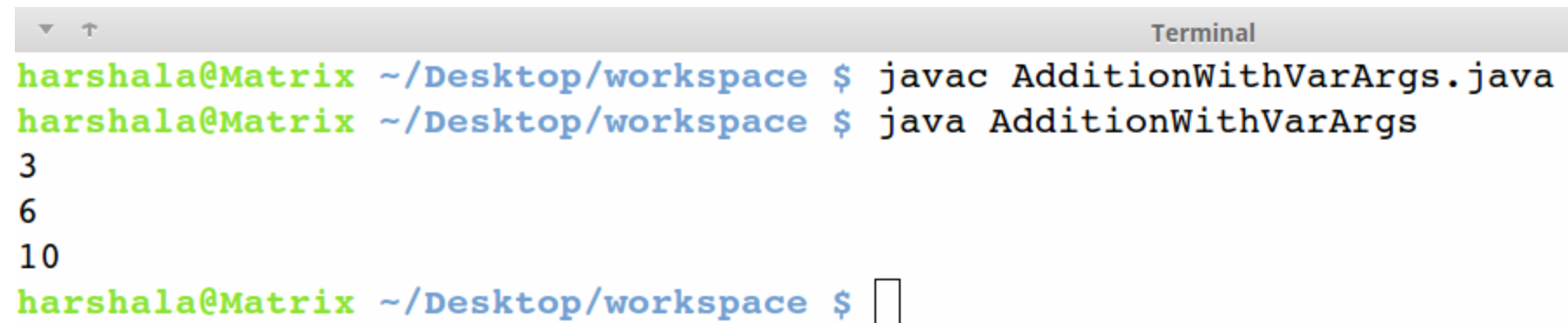
AdditionWithVarArgs.java ×
public class AdditionWithVarArgs {

    public static void main(String[] args) {
        System.out.println(add(1,2));
        System.out.println(add(1,2,3));
        System.out.println(add(1,2,3,4));
    }

    public static int add(int... nums) {
        int sum = 0;
        for (int i = 0; i < nums.length; i++) {
            sum += nums[i];
        }
        return sum;
    }
}

```

# Output



A terminal window titled "Terminal" with a grey header bar containing a dropdown arrow and an upward arrow. The terminal shows the following commands and output:

```
harshala@Matrix ~/Desktop/workspace $ javac AdditionWithVarArgs.java
harshala@Matrix ~/Desktop/workspace $ java AdditionWithVarArgs
3
6
10
harshala@Matrix ~/Desktop/workspace $
```