

# Fundamentals of Java

## Session: 10

## Inheritance and Polymorphism





- ◆ Describe inheritance
- ◆ Explain the types of inheritance
- ◆ Explain super class and subclass
- ◆ Explain the use of super keyword
- ◆ Explain method overriding
- ◆ Describe Polymorphism
- ◆ Differentiate type of reference and type of objects
- ◆ Explain static and dynamic binding
- ◆ Explain virtual method invocation
- ◆ Explain the use of abstract keyword



- ◆ In the world around us, there are many animals and birds that eat the same type of food and have similar characteristics.
- ◆ Therefore, all the animals that eat plants can be categorized as herbivores, those that eat animals as carnivores, and those that eat both plants and animals as omnivores.
- ◆ This kind of grouping or classification of things is called subclassing and the child groups are known as subclasses.
- ◆ Similarly, Java provides the concept of inheritance for creating subclasses of a particular class.
- ◆ Also, animals such as chameleon change their color based on the environment.
- ◆ Human beings also play different roles in their daily life such as father, son, husband, and so on.
- ◆ This means, that they behave differently in different situations.
- ◆ Similarly, Java provides a feature called polymorphism in which objects behave differently based on the context in which they are used.



In daily life, one often comes across objects that share a kind-of or is-a relationship with each other.

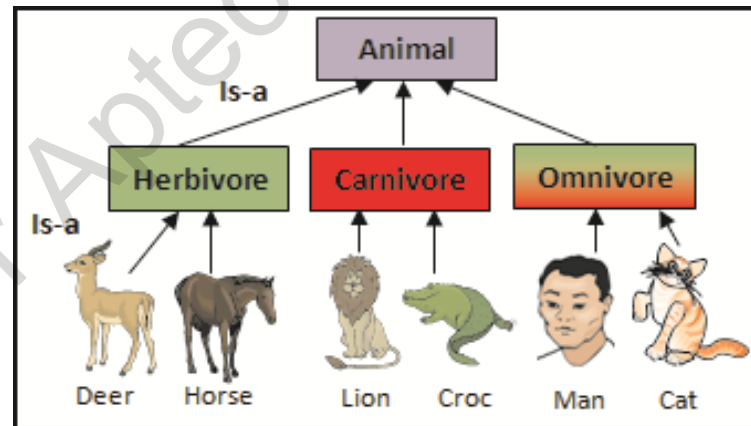
For example, Car is-a four-wheeler, a four-wheeler is-a vehicle, and a vehicle is-a machine.

Similarly, many other objects can be identified having such relationship. All such objects have properties that are common.

For example, all four wheelers have wipers and a rear view mirror.

All vehicles have a vehicle number, wheels, and engine irrespective of a four-wheeler or two-wheeler.

- ◆ Following figure shows some examples of is-a relationship:





- ◆ The figure shows is-a relationship between different objects.
- ◆ For example, Deer is-a herbivore and a herbivore is-a animal.
- ◆ The common properties of all herbivores can be stored in class herbivore.
- ◆ Similarly, common properties of all types of animals such as herbivore, carnivore, and omnivore can be stored in the Animal class.
- ◆ Thus, the class Animal becomes the top-level class from which the other classes such as Herbivore, Carnivore, and Omnivore inherit properties and behavior.
- ◆ The classes Deer, Horse, Lion, and so on inherit properties from the classes Herbivore, Carnivore, and so on.
- ◆ This is called inheritance.
- ◆ Thus, inheritance in Java is a feature through which classes can be derived from other classes and inherit fields and methods from those classes.

# Features and Terminologies 1-2



The class that is derived from another class is called a subclass, derived class, child class, or extended class.

The class from which the subclass is derived is called a super class, base class, or parent class.

The derived class can reuse the fields and methods of the existing class without having to re-write or debug the code again.

A subclass inherits all the members such as fields, nested classes, and methods from its super class except those with `private` access specifier.

Constructors of a class are not considered as members of a class and are not inherited by subclasses.

The child class can invoke the constructor of the super class from its own constructor.

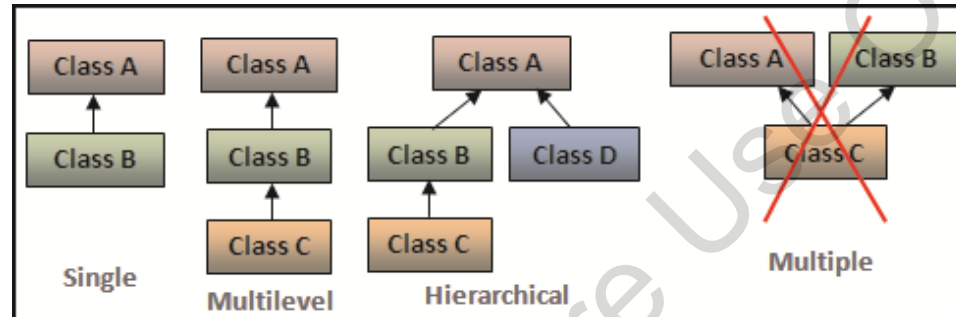
Members having default accessibility in the super class are not inherited by subclasses of other packages.

The subclass will have its own specific characteristics along with those inherited from the super class.

# Features and Terminologies 2-2



- ◆ There are several types of inheritance as shown in the following figure:



- A child class inherits from one and only one parent class

**Single Inheritance**

- A child class derives from a parent that itself is a child of another class

**Multilevel Inheritance**

- A parent class has more than one child classes at different levels

**Hierarchical Inheritance**

- A child class derives from more than one parent class

**Multiple Inheritance**

# Working with Super class and Subclass 1-6



- ◆ Within a subclass, one can use the inherited members as is, hide them, replace them, or enhance them with new members as follows:

The inherited members, including fields and methods, can be directly used just like any other fields.

One can declare a field with the same name in the subclass as the one in the super class. This will lead to hiding of super class field which is not advisable.

One can declare new fields in the subclass that are not present in the super class. These members will be specific to the subclass.

One can write a new instance method with the same signature in the subclass as the one in the super class. This is called method overriding.

A new `static` method can be created in the subclass with the same signature as the one in the super class. This will lead to hiding of the super class method.

One can declare new methods in the subclass that are not present in the super class.

A subclass constructor can be used to invoke the constructor of the super class, either implicitly or by using the keyword `super`.

The `extends` keyword is used to create a subclass. A class can be directly derived from only one class.



# Working with Super class and Subclass 2-6



- ◆ If a class does not have any super class, it is implicitly derived from Object class.
- ◆ The syntax for creating a subclass is as follows:

## Syntax

```
public class <class1-name> extends <class2-name>
{
...
...
}
```

where,

class1-name: Specifies the name of the child class.

class2-name: Specifies the name of the parent class.

- ◆ Following code snippet demonstrates the creation of super class **Vehicle**:

```
package session10;
public class Vehicle {

    // Declare common attributes of a vehicle
    protected String vehicleNo; // Variable to store vehicle number
    protected String vehicleName; // Variable to store vehicle name
    protected int wheels; // Variable to store number of wheels
```

# Working with Super class and Subclass 3-6



```
/**
 * Accelerates the vehicle
 *
 * @return void
 */
public void accelerate(int speed) {
    System.out.println("Accelerating at:" + speed + " kmph");
}
}
```

- ◆ The parent class **Vehicle** consists of common attributes of a vehicle such as **vehicleNo**, **vehicleName**, and **wheels**.
- ◆ Also, it consists of a common behavior of a vehicle, that is, **accelerate()** that prints the speed at which the vehicle is accelerating.
- ◆ Following code snippet demonstrates the creation of subclass **FourWheeler**:

```
package session10;
class FourWheeler extends Vehicle{

    // Declare a field specific to child class
    private boolean powerSteer; // Variable to store steering information
}
```

# Working with Super class and Subclass 4-6



```
/**
 * Parameterized constructor to initialize values based on user input
 *
 * @param vId a String variable storing vehicle ID
 * @param vName a String variable storing vehicle name
 * @param numWheels an integer variable storing number of wheels
 * @param pSteer a String variable storing steering information
 */
public FourWheeler(String vId, String vName, int numWheels, boolean
pSteer) {

    // Attributes inherited from parent class
    vehicleNo = vId;
    vehicleName = vName;
    wheels = numWheels;

    // Child class' own attribute
    powerSteer = pSteer;
}
```

# Working with Super class and Subclass 5-6



```
/**
 * Displays vehicle details
 *
 * @return void
 */
public void showDetails() {

    System.out.println("Vehicle no:" + vehicleNo);
    System.out.println("Vehicle Name:" + vehicleName);
    System.out.println("Number of Wheels:" + wheels);

    if(powerSteer == true)
        System.out.println("Power Steering:Yes");
    else
        System.out.println("Power Steering:No");
    }
}

/**
 * Define the TestVehicle.java class
 */
public class TestVehicle {
```

# Working with Super class and Subclass 6-6



```
/**
 * @param args the command line arguments
 */
public static void main(String[] args) {

    // Create an object of child class and specify the values
    FourWheeler objFour = new FourWheeler("LA-09 CS-1406", "Volkswagen",
    4, true);
    objFour.showDetails(); // Invoke the child class method
    objFour.accelerate(200); // Invoke the inherited method
}
}
```

- ◆ Following figure shows the output of the program:

```
run:
Vehicle no:LA-09 CS-1406
Vehicle Name:Volkswagen
Number of Wheels:4
Power Steering:Yes
Accelerating at:200 kmph
```

# Overriding Methods 1-5



- ◆ Java allows creation of an instance method in a subclass having the same signature and return type as an instance method of the super class.
- ◆ This is called method overriding.
- ◆ Method overriding allows a class to inherit behavior from a super class and then, to modify the behavior as needed.
- ◆ Rules to remember when overriding:

The overriding method must have the same name, type, and number of arguments as well as return type as the super class method.

An overriding method cannot have a weaker access specifier than the access specifier of the super class method.

- ◆ The **accelerate()** method can be overridden in the subclass as shown in the following code snippet:

```
package session10;
class FourWheeler extends Vehicle{

    // Declare a field specific to child class
    private boolean powerSteer; // Variable to store steering information
```

# Overriding Methods 2-5



```
/**
 * Parameterized constructor to initialize values based on user input
 *
 * @param vID a String variable storing vehicle ID
 * @param vName a String variable storing vehicle name
 * @param numWheels an integer variable storing number of wheels
 * @param pSteer a String variable storing steering information
 */
public FourWheeler(String vId, String vName, int numWheels, boolean
pSteer) {

    // attributes inherited from parent class
    vehicleNo=vId;
    vehicleName=vName;
    wheels=numWheels;

    // child class' own attribute
    powerSteer=pSteer;
}
```

# Overriding Methods 3-5



```
/**
 * Displays vehicle details
 *
 * @return void
 */
public void showDetails() {

    System.out.println("Vehicle no:" + vehicleNo);
    System.out.println("Vehicle Name:" + vehicleName);
    System.out.println("Number of Wheels:" + wheels);

    if (powerSteer == true) {
        System.out.println("Power Steering:Yes");
    } else {
        System.out.println("Power Steering:No");
    }

}

/**
 * Overridden method
 * Accelerates the vehicle
 *
 * @return void
```



# Overriding Methods 4-5



```
*/
@Override
public void accelerate(int speed) {
    System.out.println("Maximum acceleration:" + speed + " kmph");
}
}

/**
 * Define the TestVehicle.java class
 */
public class TestVehicle {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // Create an object of child class and specify the values
        FourWheeler objFour = new FourWheeler("LA-09 CS-1406", "Volkswagen",
        4, true);
        objFour.showDetails(); // Invoke child class method
        objFour.accelerate(200); // Invoke inherited method
    }
}
```

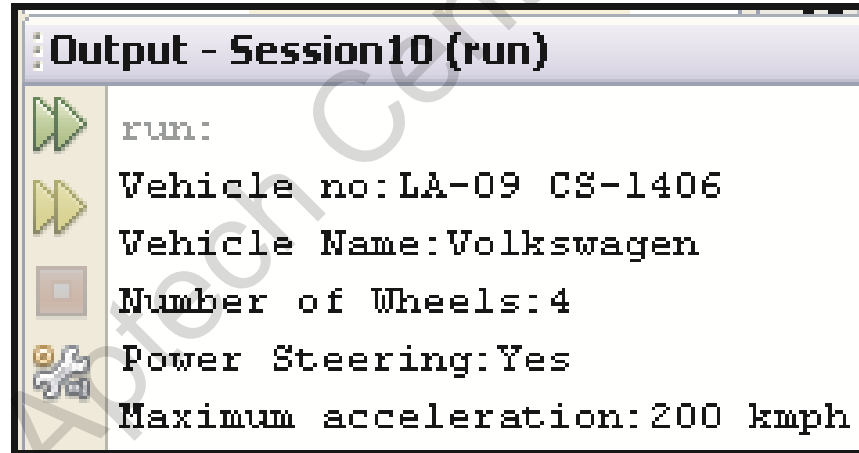
# Overriding Methods 5-5



- ◆ The **accelerate()** method is overridden in the child class with the same signature and return type but with a modified message.
- ◆ Notice the use of `@Override` annotation on top of the method.

Annotations provide additional information about a program. Annotations have no direct effect on the functioning of the code they annotate.

- ◆ Following figure shows the output of the code:



```
Output - Session10 (run)
run:
Vehicle no:LA-09 CS-1406
Vehicle Name:Volkswagen
Number of Wheels:4
Power Steering:Yes
Maximum acceleration:200 kmph
```

- ◆ The **accelerate()** method now prints the message specified in the subclass.
- ◆ Since the **accelerate()** method is overridden in the subclass, the subclass version of the **accelerate()** method is invoked and not the super class **accelerate()** method.

# Accessing Super class Constructor and Methods 1-6



In the code, the subclass constructor is used to initialize the values of the common attributes inherited from the super class **Vehicle**.

This is not the correct approach because all the subclasses of the **Vehicle** class will have to initialize the values of common attributes every time in their constructors.

Java allows the subclass to invoke the super class constructor and methods using the keyword `super`.

Also, when the `accelerate()` method is overridden in the subclass, the statement(s) written in the `accelerate()` method of the super class, **Vehicle**, are not printed.

- ◆ To address these issues, one can use:
  - the `super` keyword to invoke the super class method using `super.method-name()`.
  - the `super()` method to invoke the super class constructors from the subclass constructors.

# Accessing Super class Constructor and Methods 2-6



- ◆ Following code snippet demonstrates the modified super class **Vehicle.java** using a parameterized constructor:

```
package session10;
class Vehicle {

    protected String vehicleNo; // Variable to store vehicle number
    protected String vehicleName; // Variable to store vehicle name
    protected int wheels; // Variable to store number of wheels

    /**
     * Parameterized constructor to initialize values based on user input
     *
     * @param vId a String variable storing vehicle ID
     * @param vName a String variable storing vehicle name
     * @param numWheels an integer variable storing number of wheels
     */
    public Vehicle(String vId, String vName, int numWheels){
        vehicleNo=vId;
        vehicleName=vName;
        wheels=numWheels;
    }
}
```

# Accessing Super class Constructor and Methods 3-6



```
/**
 * Accelerates the vehicle
 *
 * @return void
 */
public void accelerate(int speed){
    System.out.println("Accelerating at:"+ speed + " kmph");
}
}
```

- ◆ Following code snippet depicts the modified subclass **FourWheeler.java** using the `super` keyword to invoke super class constructor and methods:

```
package session10;
class FourWheeler extends Vehicle{
    private boolean powerSteer; // Variable to store steering information

    /**
     * Parameterized constructor to initialize values based on user input
     *
     * @param vID a String variable storing vehicle ID
     * @param vName a String variable storing vehicle name
     * @param numWheels an integer variable storing number of wheels
     */
}
```

# Accessing Super class Constructor and Methods 4-6



```
* @param pSteer a String variable storing steering information
*/
public FourWheeler(String vId, String vName, int numWheels, boolean
pSteer) {
    // Invoke the super class constructor
    super(vId,vName,numWheels);
    powerSteer=pSteer;
}

/**
 * Displays vehicle details
 *
 * @return void
 */
public void showDetails(){
    System.out.println("Vehicle no:"+ vehicleNo);
    System.out.println("Vehicle Name:"+ vehicleName);
    System.out.println("Number of Wheels:"+ wheels);

    if(powerSteer==true){
        System.out.println("Power Steering:Yes");
    }
}
```

# Accessing Super class Constructor and Methods 5-6



```
        else{
            System.out.println("Power Stearing:No");
        }
    }
    /**
     * Overridden method
     * Displays the acceleration details of the vehicle
     *
     * @return void
     */
    @Override
    public void accelerate(int speed){
        // Invoke the super class accelerate() method
        super.accelerate(speed);
        System.out.println("Maximum acceleration:"+ speed + " kmph");
    }
}
/**
 * Define the TestVehicle.java class
 */
public class TestVehicle {
    /**
```

# Accessing Super class Constructor and Methods 6-6



```
* @param args the command line arguments
*/
public static void main(String[] args){
    FourWheeler objFour = new FourWheeler("LA-09 CS-1406", "Volkswagen", 4,
    true);
    objFour.showDetails();
    objFour.accelerate(200);
}
}
```

- ◆ The `super()` method is used to call the super class constructor from the child class constructor.
- ◆ Similarly, the `super.accelerate()` statement is used to invoke the super class `accelerate()` method from the child class.
- ◆ Following figure shows the output of the code:

```
Output - Session10 (run)
run:
Vehicle no:LA-09 CS-1406
Vehicle Name:Volkswagen
Number of Wheels:4
Power Steering:Yes
Accelerating at:200 kmph
Maximum acceleration:200 kmph
```





The word polymorph is a combination of two words namely, 'poly' which means 'many' and 'morph' which means 'forms'.

Thus, polymorph refers to an object that can have many different forms.

This principle can also be applied to subclasses of a class that can define their own specific behaviors as well as derive some of the similar functionality of the super class.

The concept of method overriding is an example of polymorphism in object-oriented programming in which the same method behaves in a different manner in super class and in subclass.

# Understanding Static and Dynamic Binding 1-8



- ◆ Some important differences between static and dynamic binding are listed in the following table:

Static Binding	Dynamic Binding
Static binding occurs at compile time.	Dynamic binding occurs at runtime.
Private, static, and final methods and variables use static binding and are bounded by compiler.	Virtual methods are bounded at runtime based upon the runtime object.
Static binding uses object type information for binding. That is, the type of class.	Dynamic binding uses reference type to resolve binding.
Overloaded methods are bounded using static binding.	Overridden methods are bounded using dynamic binding.

- ◆ Following code snippet demonstrates an example of static binding:

```
package session10;  
class Employee {  
  
    String empId; // Variable to store employee ID  
    String empName; // Variable to store employee name  
    int salary; // Variable to store salary  
    float commission; // Variable to store commission  
}
```

# Understanding Static and Dynamic Binding 2-8



```
/**
 * Parameterized constructor to initialize the variables
 *
 * @param id a String variable storing employee id
 * @param name a String variable storing employee name
 * @param sal an integer variable storing salary
 *
 */
public Employee(String id, String name, int sal) {
    empId=id;
    empName=name;
    salary=sal;
}
/**
 * Calculates commission based on sales value
 * @param sales a float variable storing sales value
 *
 * @return void
 */
public void calcCommission(float sales){
    if(sales > 10000)
        commission = salary * 20 / 100;
```

# Understanding Static and Dynamic Binding 3-8



```
        else
            commission=0;
    }

    /**
     * Overloaded method. Calculates commission based on overtime
     * @param overtime an integer variable storing overtime hours
     *
     * @return void
     */
    public void calcCommission(int overtime){
        if(overtime > 8)
            commission = salary/30;
        else
            commission = 0;
    }

    /**
     * Displays employee details
     *
     * @return void
     */
```

# Understanding Static and Dynamic Binding 4-8



```
public void displayDetails(){
    System.out.println("Employee ID:"+empId);
    System.out.println("Employee Name:"+empName);
    System.out.println("Salary:"+salary);
    System.out.println("Commission:"+commission);
}
}
/**
 * Define the EmployeeDetails.java class
 */
public class EmployeeDetails {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args)
    {
        // Instantiate the Employee class object
        Employee objEmp = new Employee("E001","Maria Nemeth", 40000);
        // Invoke the calcCommission() with float argument
        objEmp.calcCommission(20000F);
        objEmp.displayDetails(); // Print the employee details
    } }
```

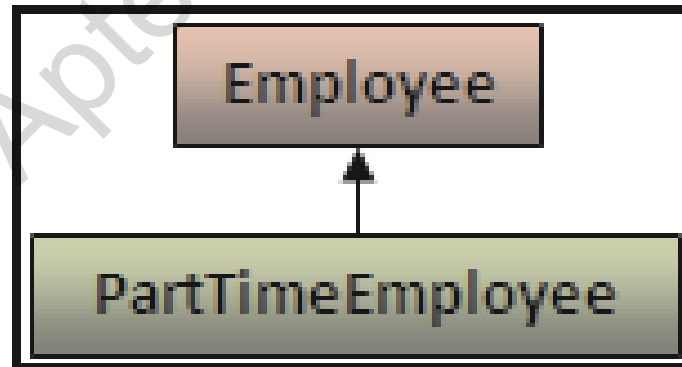
# Understanding Static and Dynamic Binding 5-8



- ◆ In the example, when the `calcCommission()` method is executed, the method with `float` argument gets invoked because it was bounded during compile time based on the type of variable, that is, `float`.
- ◆ Following figure shows the output of the code:

```
run:  
Employee ID:E001  
Employee Name:Maria Nemeth  
Salary:40000  
Commission:8000.0
```

- ◆ Now, consider the class hierarchy shown in the following figure:



# Understanding Static and Dynamic Binding 6-8



- ◆ Following code snippet demonstrates an example of dynamic binding:

```
package session10;

class PartTimeEmployee extends Employee{

    // Subclass specific variable
    String shift; // Variable to store shift information

    /**
     * Parameterized constructor to initialize values based on user input
     *
     * @param id a String variable storing employee ID
     * @param name a String variable storing employee name
     * @param sal an integer variable storing salary
     * @param shift a String variable storing shift information
     */
    public PartTimeEmployee(String id, String name, int sal, String shift)
    {
        // Invoke the super class constructor
        super(id, name, sal);
        this.shift=shift;
    }
}
```

# Understanding Static and Dynamic Binding 7-8



```
/**
 * Overridden method to display employee details
 *
 * @return void
 */
@Override
public void displayDetails(){
    calcCommission(12); // Invoke the inherited method
    super.displayDetails(); // Invoke the super class display method
    System.out.println("Working shift:"+shift);
}
}
/**
 * Modified EmployeeDetails.java
 */
public class EmployeeDetails {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args)
    {
```



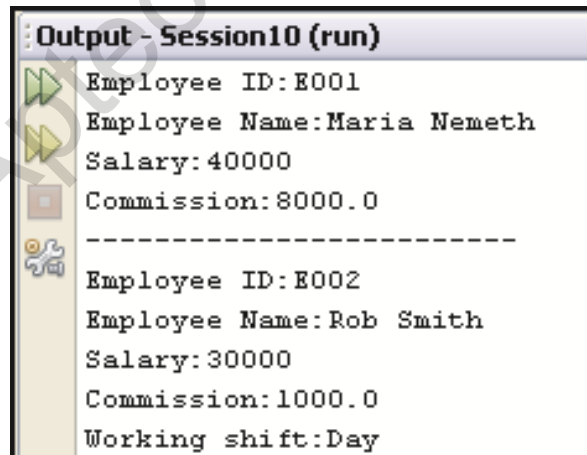
# Understanding Static and Dynamic Binding 8-8



```
// Instantiate the Employee class object
Employee objEmp = new Employee("E001","Maria Nemeth", 40000);
objEmp.calcCommission(20000F); // Calculate commission
objEmp.displayDetails(); // Print the details
System.out.println("-----");

// Instantiate the Employee object as PartTimeEmployee
Employee objEmp1 = new PartTimeEmployee("E002", "Rob Smith", 30000,
    "Day");
objEmp1.displayDetails(); // Print the details
}
}
```

- ◆ Following figure shows the output of the code:



```
Output - Session10 (run)
Employee ID:E001
Employee Name:Maria Nemeth
Salary:40000
Commission:8000.0
-----
Employee ID:E002
Employee Name:Rob Smith
Salary:30000
Commission:1000.0
Working shift:Day
```



## Upcasting

- ◆ In the earlier code, type of object of **objEmp1** is **Employee**.
- ◆ This means that the object will have all characteristics of an **Employee**.
- ◆ However, the reference assigned to the object was of **PartTimeEmployee**.
- ◆ This means that the object will bind with the members of **PartTimeEmployee** class during runtime.
- ◆ That is, object type is **Employee** and reference type is **PartTimeEmployee**.
- ◆ This is possible only when the classes are related with a parent-child relationship.
- ◆ Java allows casting an instance of a subclass to its parent class.
- ◆ This is known as upcasting.
- ◆ For example,  

```
PartTimeEmployee objPT = new PartTimeEmployee();  
Employee objEmp = objPT; // upcasting
```
- ◆ While upcasting a child object, the child object **objPT** is directly assigned to the parent class object **objEmp**.
- ◆ However, the parent object cannot access the members that are specific to the child class and not available in the parent class.



## Downcasting

- ◆ Java also allows casting the parent reference back to the child type.
- ◆ This is because parent references an object of type child.
- ◆ Casting a parent object to child type is called downcasting because an object is being casted to a class lower down in the inheritance hierarchy.
- ◆ However, downcasting requires explicit type casting by specifying the child class name in brackets.
- ◆ For example,  

```
PartTimeEmployee objPT1 = (PartTimeEmployee) objEmp;  
    // downcasting
```



In the earlier code, during execution of the statement `Employee objEmp1= new PartTimeEmployee (...)`, the runtime type of the `Employee` object is determined.

The compiler does not generate error because the `Employee` class has a `displayDetails()` method.

At runtime, the method executed is referenced from the `PartTimeEmployee` object. This aspect of polymorphism is called virtual method invocation.

- ◆ The difference here is between the compiler and the runtime behavior.
  - The compiler checks the accessibility of each method and field based on the class definition whereas the behavior associated with an object is determined at runtime.
  - Since the object created was of `PartTimeEmployee`, the `displayDetails()` method of `PartTimeEmployee` is invoked even though the object type is `Employee`.
  - This is referred to as virtual method invocation and the method is referred to as virtual method.
- ◆ In other languages such as C++, the same can be achieved by using the keyword `virtual`.

# Using the 'abstract' Keyword 1-9



Java provides the `abstract` keyword to create a super class that serves as a generalized form that will be inherited by all of its subclasses.

The methods of the super class serve as a contract or a standard that the subclass can implement in its own way.

## Abstract method

An abstract method is one that is declared with the `abstract` keyword and is without an implementation, that is, without any body.

- ◆ The abstract method does not contain any '{}' brackets and ends with a semicolon.
- ◆ The syntax for declaring an abstract method is as follows:

## Syntax

```
abstract <return-type> <method-name> (<parameter-list>);
```

where,

`abstract`: Indicates that the method is an abstract method.

- ◆ For example,

```
public abstract void calculate();
```

# Using the 'abstract' Keyword 2-9



## Abstract class

An abstract class is one that consists of abstract methods.

- ◆ Abstract class serves as a framework that provides certain behavior for other classes.
- ◆ The subclass provides the requirement-specific behavior of the existing framework.
- ◆ Abstract classes cannot be instantiated and they must be subclassed to use the class members.
- ◆ The subclass provides implementations for the abstract methods in its parent class.
- ◆ The syntax for declaring an abstract class is as follows:

## Syntax

```
abstract class <class-name>
{
// declare fields
// define concrete methods
[abstract <return-type> <method-name>(<parameter-list>);]
}
```

# Using the 'abstract' Keyword 3-9



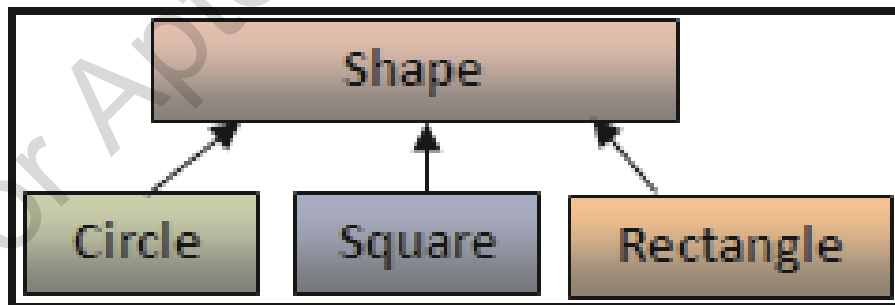
where,

`abstract`: Indicates that the method is an abstract method.

- ◆ For example,

```
public abstract Calculator
{
    public float getPI(){ // Define a concrete method
        return 3.14F;
    }
    abstract void Calculate(); // Declare an abstract method
}
```

- ◆ Consider the class hierarchy as shown in following figure:



# Using the 'abstract' Keyword 4-9



- ◆ Following code snippet demonstrates creation of abstract class and abstract method:

```
package session10;
abstract class Shape {
    private final float PI = 3.14F; // Variable to store value of PPI
    /**
     * Returns the value of PI
     *
     * @return float
     */
    public float getPI(){
        return PI;
    }

    /**
     * Abstract method
     * @param val a float variable storing the value specified by user
     *
     * @return float
     */
    abstract void calculate(float val);
}
```



# Using the 'abstract' Keyword 5-9



- ◆ Following code snippet demonstrates two subclasses **Circle** and **Rectangle** inheriting the abstract class **Shape**:

```
package session10;
/**
 * Define the child class Circle.java
 */
class Circle extends Shape{
    float area; // Variable to store area of a circle

    /**
     * Implement the abstract method to calculate area of circle
     *
     * @param rad a float variable storing value of radius
     * @return void
     */
    @Override
    void calculate(float rad){
        area = getPI() * rad * rad;
        System.out.println("Area of circle is:" + area);
    }
}
```

# Using the 'abstract' Keyword 6-9



```
/**
 * Define the child class Rectangle.java
 */
class Rectangle extends Shape{

    float perimeter; // Variable to store perimeter value
    float length=10; // Variable to store length

    /**
     * Implement the abstract method to calculate the perimeter
     *
     * @param width a float variable storing width
     * @return void
     */
    @Override
    void calculate(float width){

        perimeter = 2 * (length+width);
        System.out.println("Perimeter of the Rectangle is:"+ perimeter);
    }
}
```

# Using the 'abstract' Keyword 7-9



- ◆ Following code snippet depicts the code for **Calculator** class that uses the subclasses based on user input:

```
package session10;

public class Calculator {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args)
    {
        Shape objShape; // Declare the Shape object
        String shape; // Variable to store the type of shape

        if(args.length==2) { // Check the number of command line arguments
            //Retrieve the value of shape from args[0]
            shape = args[0].toLowerCase(); // converting to lower case
            switch(shape) {

                // Assign reference to Shape object as per user input
                case "circle": objShape = new Circle();
                               objShape.calculate(Float.parseFloat(args[1]));
                               break;
```

# Using the 'abstract' Keyword 8-9



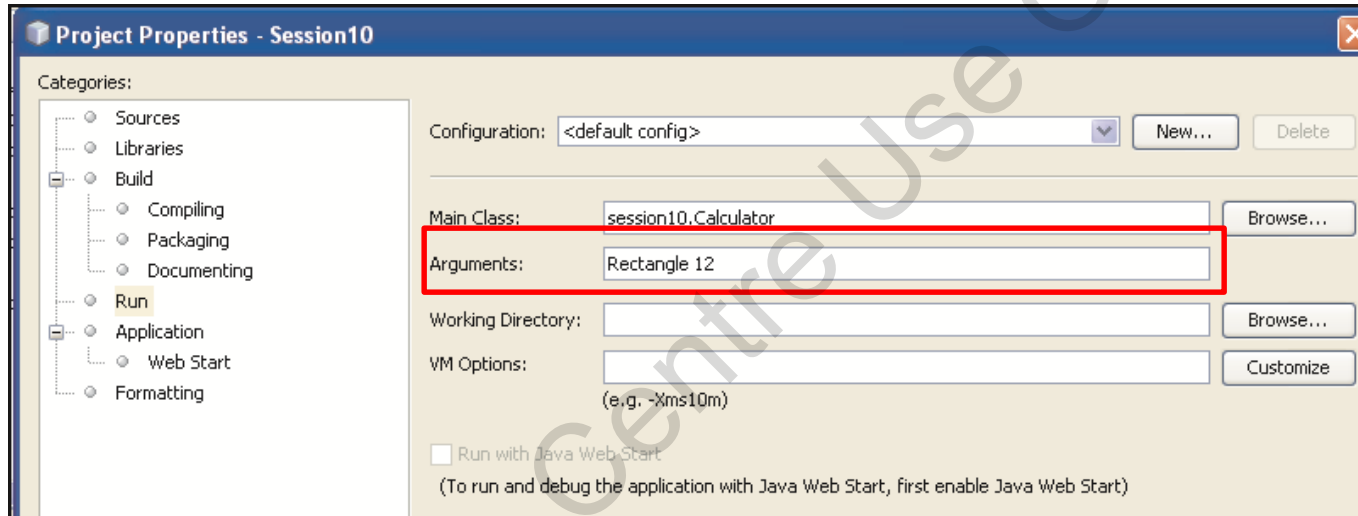
```
        case "rectangle": objShape = new Rectangle();
        objShape.calculate(Float.parseFloat(args[1]));
        break;
    }
}
else{
    // Error message to be displayed when arguments are not supplied
    System.out.println("Usage: java Calculator <shape-name> <value>");
}
}
```

- ◆ To execute the example at command line, write the following command:  
`java Calculator Rectangle 12`
- ◆ Note that the word **Circle** can be in any case.
- ◆ Within the code, it will be converted to lowercase.

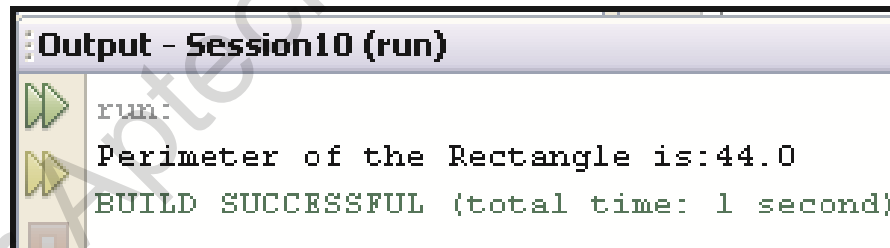
# Using the 'abstract' Keyword 9-9



- ◆ To execute the example in NetBeans IDE, type the arguments in the **Arguments** box of **Run** property as shown in the following figure:



- ◆ Following figure shows the output of the code after execution:



- ◆ The output shows the perimeter of a rectangle.
- ◆ This is because the first command line argument was **Rectangle**.
- ◆ Therefore, within the `main()` method, the `switch` case for rectangle got executed.



- ◆ Inheritance is a feature in Java through which classes can be derived from other classes and inherit fields and methods from classes it is inheriting.
- ◆ The class that is derived from another class is called a subclass, derived class, child class, or extended class. The class from which the subclass is derived is called a super class.
- ◆ Creation of an instance method in a subclass having the same signature and return type as an instance method of the super class is called method overriding.
- ◆ Polymorphism refers to an object that can have many different forms.
- ◆ When the compiler resolves the binding of methods and method calls at compile time, it is called static binding or early binding. If the compiler resolves the method calls and the binding at runtime, it is called dynamic binding or late binding.
- ◆ An abstract method is one that is declared with the abstract keyword without an implementation, that is, without any body.
- ◆ An abstract class is one that consists of abstract methods.
- ◆ An abstract class serves as a framework that provides certain pre-defined behavior for other classes that can be modified later as per the requirement of the inheriting class.