

Learning Java - A Foundational Journey

Session: 7

Inheritance and Polymorphism

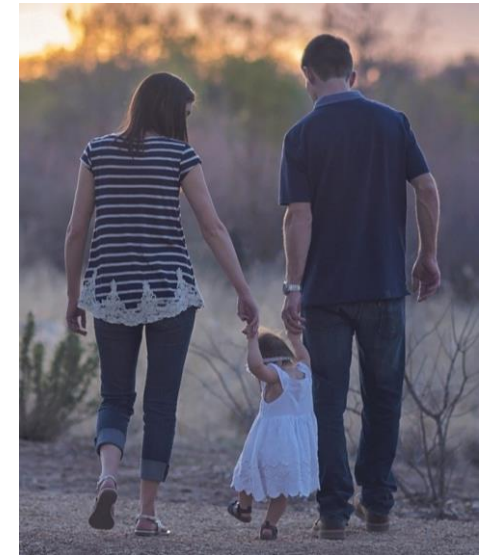
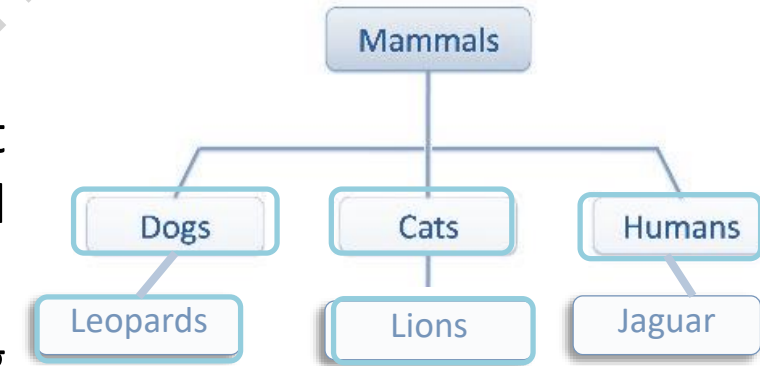




- ◆ Describe inheritance
- ◆ Explain the types of inheritance
- ◆ Explain super class and subclass
- ◆ Explain the use of super keyword
- ◆ Explain method overriding
- ◆ Describe Polymorphism
- ◆ Differentiate type of reference and type of objects
- ◆ Explain static and dynamic binding
- ◆ Explain virtual 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.
- ◆ 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 concept of **inheritance** for creating subclasses of a particular class.
- ◆ Human beings also play different roles in their daily life such as father, son, husband, daughter, 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.

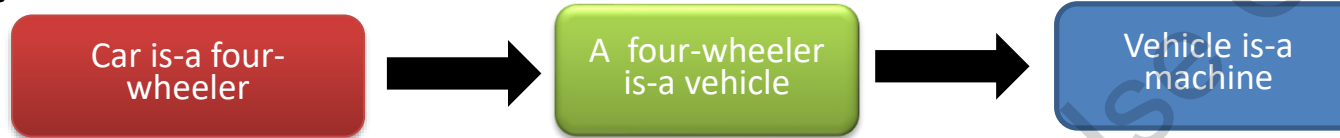


Inheritance



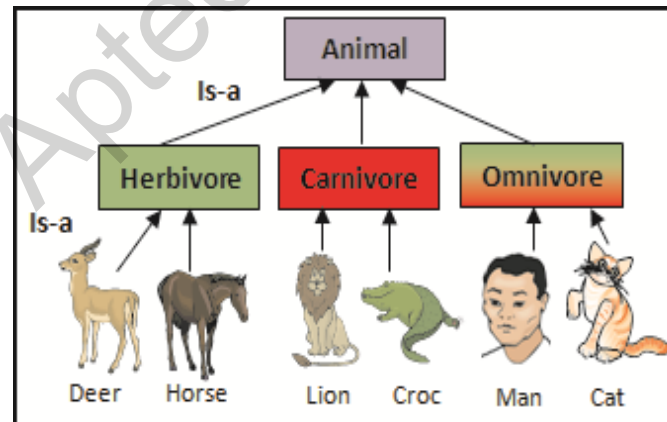
In our day-to-day life, we often come across objects that share a kind-of or is-a relationship with each other.

For example



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

All vehicles irrespective of a four-wheeler or a two-wheeler have:



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.

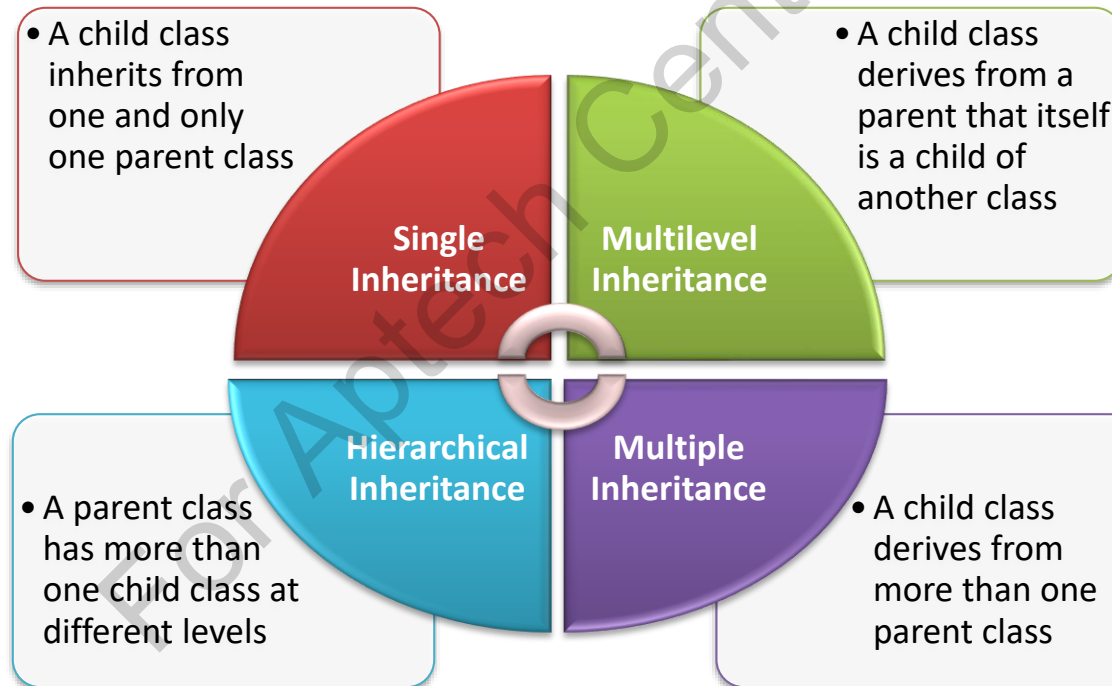
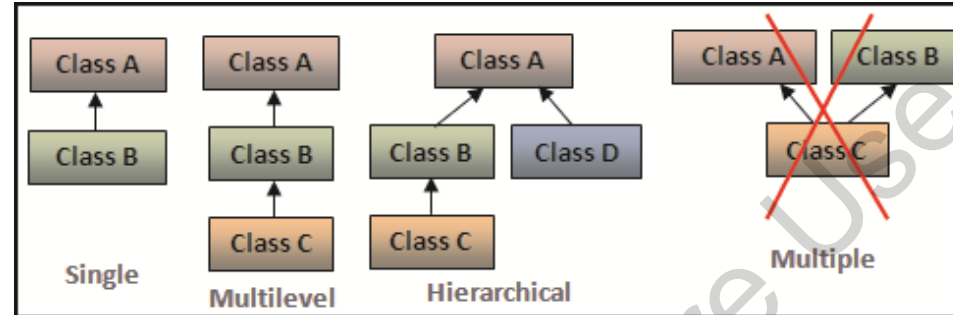
Constructors of a class are not considered as members of a class and are not inherited by subclasses.

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

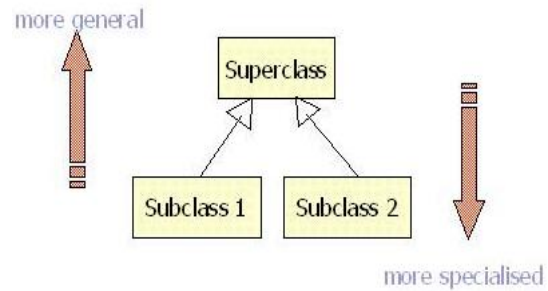
Features and Terminologies 2-2



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



Working with Super Class and Subclass 1-3



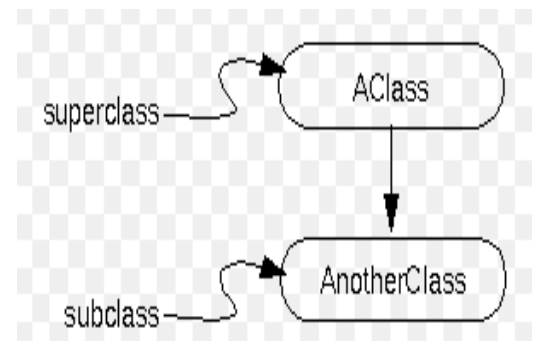
One can declare a field with the same name in the subclass as the one in the super class. This will lead to hiding of super class field which is not advisable.

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

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.

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 2-3



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.

Code Snippet:

```
package session7;
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
/**
 * Accelerates the vehicle
 *
 * @return void
 */
public void accelerate(int speed) {
System.out.println("Accelerating at:"+ speed + " kmph");
}
}
```


Working with Super Class and Subclass 3-3



Creation of subclass **FourWheeler**:

```
package session7;
class FourWheeler extends Vehicle{
// Declare a field specific to child class
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
 * @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;
}
/**
 * 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 TestVehicle 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 1-3



- ◆ 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.
- ◆ Method overriding allows a class to inherit behavior from a super class and then, to modify the behavior as required.

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.

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

For Aptech
Centre Use Only

Overriding Methods 2-3



```
package session7;
class FourWheeler extends Vehicle{
// Declare a field specific to child class
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
 * @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;
}
```

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

Overriding Methods 3-3



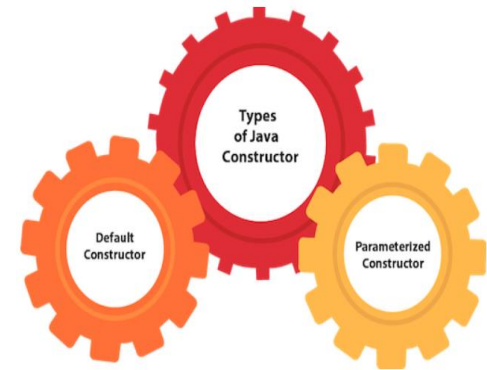
```
/**
 * Define the TestVehicle class
 */
public class TestVehicle {

    /**
     * @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
    }
}
```

A screenshot of an IDE's output window titled "Output - session7 (run)". The window shows the output of a Java program. On the left side of the window, there are icons for running (a green play button), stepping through (a green play button with a magnifying glass), and debugging (a red bug icon). The output text is as follows:

```
run:
Vehicle no:LA-09 CS-1406
Vehicle Name:Volkswagen
Number of Wheels:4
Power Steering:Yes
Maximum acceleration:200 kmph
BUILD SUCCESSFUL (total time: 3 seconds)
```

Accessing Super Class Constructor and Methods





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 1-6



- ◆ 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.
<code>private</code> , <code>static</code> , and <code>final</code> 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:

```
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
    /**
     * 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
     *
     */
}
```

Understanding Static and Dynamic Binding 2-6



```
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;
    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;
}
```


Understanding Static and Dynamic Binding 3-6

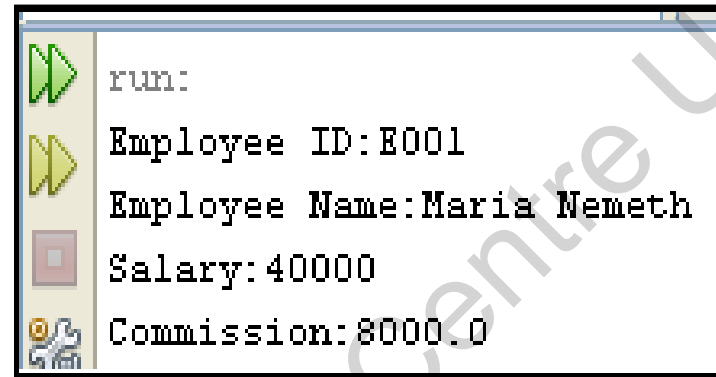


```
/**
 * Displays employee details
 *
 * @return void
 */
public void displayDetails(){
    System.out.println("Employee ID:"+empId);
    System.out.println("Employee Name:"+empName);
    System.out.println("Salary:"+salary);
    System.out.println("Commission:"+commission);
}
}
/**
 * Define the EmployeeDetails.java 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 4-6

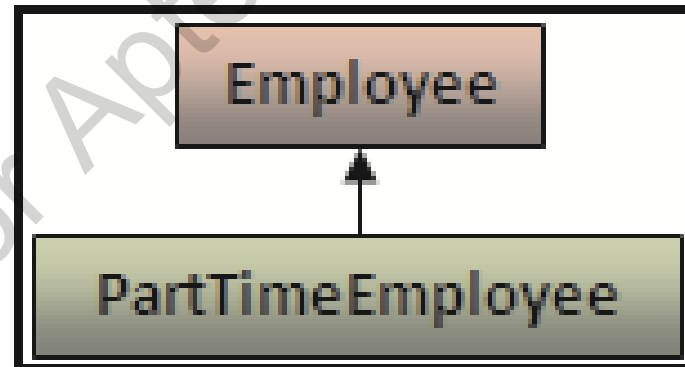


- ◆ In the example, when the `calcCommission()` method is executed it gets invoked because it was bounded during compile time based on the type of variable, that is, `float`.
- ◆ Following figure shows the output of the code:



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

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





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

```
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;
    }
    /**
     * 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);
    }
}
```

Understanding Static and Dynamic Binding 6-6



```
/**
 * Modified EmployeeDetails.java
 */
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);
        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:

A screenshot of a Java IDE's output window. It displays the output of the program, showing details for two employees. The first employee is E001, Maria Nemeth, with a salary of 40000 and a commission of 8000.0. The second employee is E002, Rob Smith, with a salary of 30000, a commission of 1000.0, and a working shift of Day. The output is formatted with a separator line between the two employee details.

```
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
```

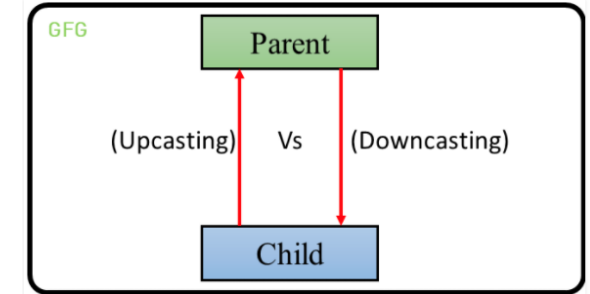
Differentiate Between Type of Reference and Type of Object 1-2



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.
- ◆ This is possible only when the classes are related with a parent-child relationship.
- ◆ This is known as upcasting.
- ◆ For example,

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



For Aptech Centre Use Only



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.

Using the 'abstract' Keyword 1-7



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.

Abstract method

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

Syntax

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

Abstract class

An `abstract` class is one that consists of `abstract` methods.

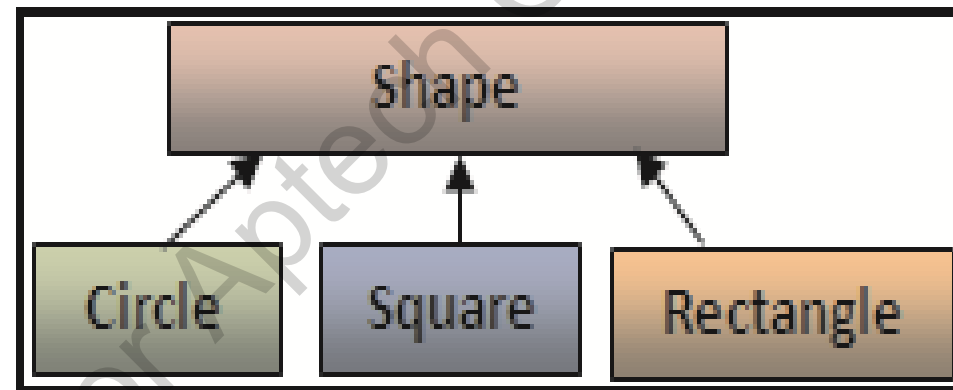
Using the 'abstract' Keyword 2-7



Syntax

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

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



Using the 'abstract' Keyword 3-7



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

```
package abstractdemo;
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 4-7



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

```
package abstractdemo;
/**
 * 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 5-7



```
/**
 * 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 6-7



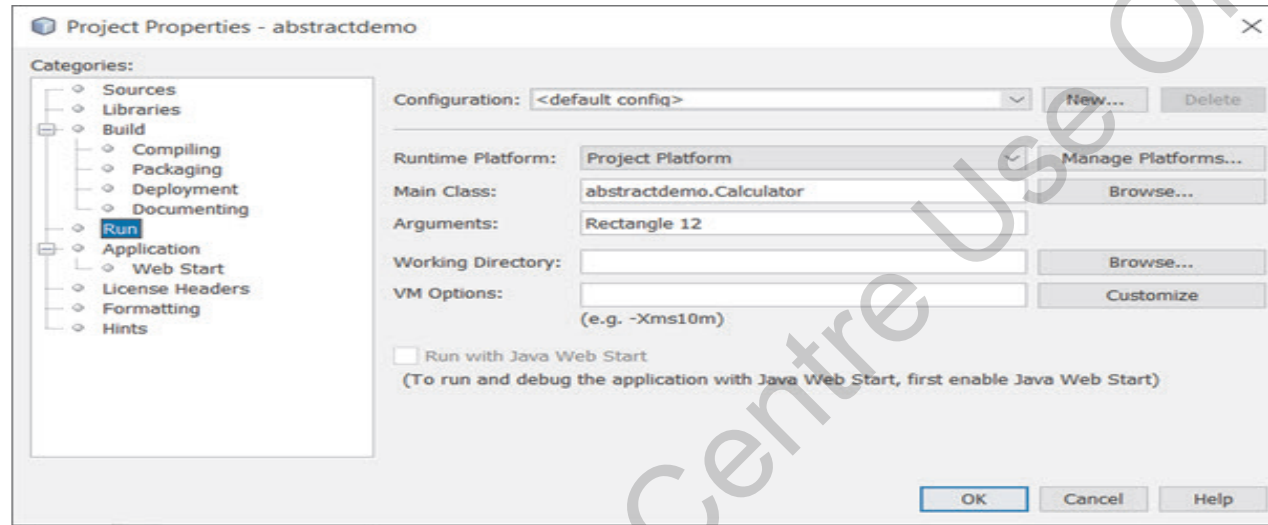
- ◆ Following code snippet depicts the code for **Calculator** class that uses the subclasses based on user input:

```
package abstractdemo;
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;
                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>");
        }
    }
}
```

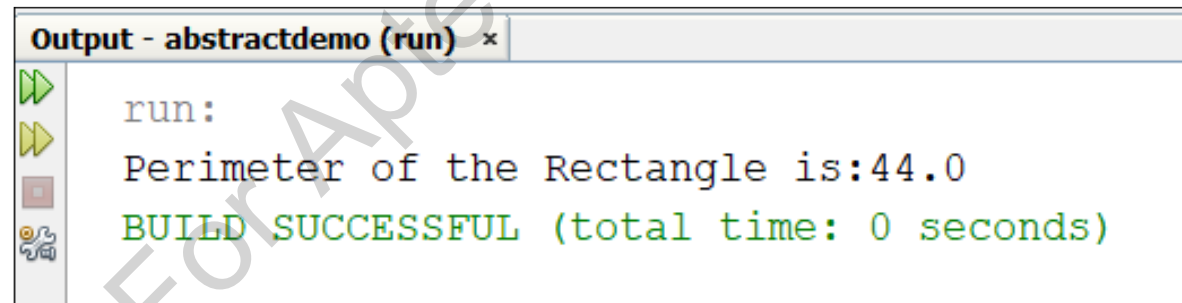
Using the 'abstract' Keyword 7-7



- ◆ 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:





- ◆ 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 and 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.