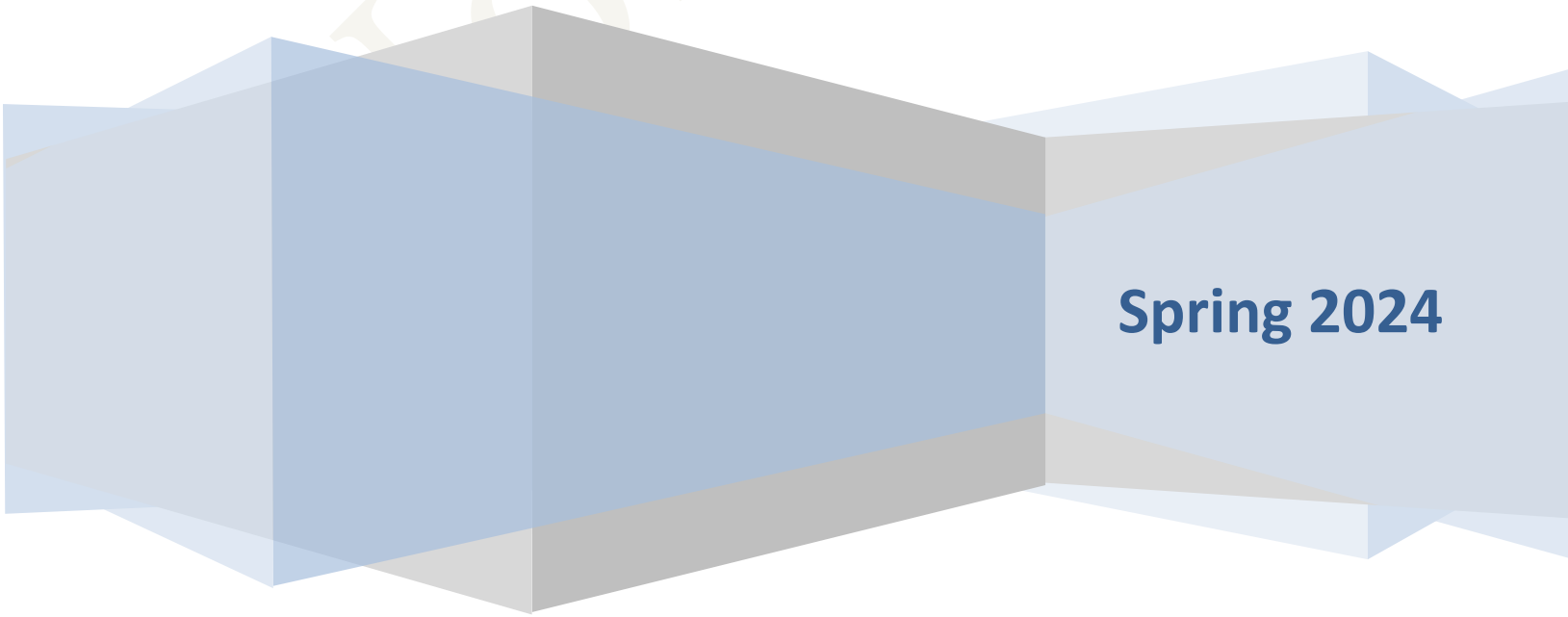
Northern University, Nowshera

Abstraction in Java

Week # 14 - Lecture 27 - 28



Spring 2024

Learning Objectives:

- **Review – Week 13**
    a. **Polymorphism**
    b. **Types of polymorphism (static & dynamic)**
    c. **Method overloading**
    d. **Operator overloading**
    e. **Method overriding**
    f. **Heading towards dynamic polymorphism**
- **Virtual methods**
- **Abstraction**
    a. **Abstract classes**
    b. **Inheriting abstract classes**
    c. **Abstract methods**
    d. **Understanding the real scenario of abstract classes**
    e. **Why we need an abstract class?**
    f. **Why we can't instantiate objects of abstract class?**

Review (Week 13)

Polymorphism is the same entity (method or operator or object) behaves differently in different scenarios.

- Polymorphism allows us to perform a single action in different ways. In other words, polymorphism allows you to define one interface and have multiple implementations.
- The word “poly” means many and “morphs” means forms, so it means many forms.
- Polymorphism is a property through which any message can be sent to objects of multiple classes, and every object has the tendency to respond in an appropriate way depending on the class properties.
- It is so important that languages that don’t support polymorphism cannot advertise themselves as Object-Oriented languages.
- Languages that possess classes but have no ability of polymorphism are called object-based languages.

Thus it is very vital for an object-oriented programming language.

Types of Polymorphism

In Java, Polymorphism can be divided into two types:

- Compile-time Polymorphism: The compile-time polymorphism can be achieved through method overloading and operator overloading in Java.
- Run-time Polymorphism: a process in which a function call to the overridden method is resolved at Runtime. This type of polymorphism is achieved by Method Overriding.

Method Overloading

In a Java class, we can create methods with the same name if they differ in parameters. For example,

```
void func() { ... }
void func(int a) { ... }
```

```
float func(double a) { ... }
float func(int a, float b) { ... }
```

This is known as method overloading in Java. Let's take a working example of method overloading.

Operator Overloading

Some operators in Java behave differently with different operands. For example,

- + operator is overloaded to perform numeric addition as well as string concatenation,
- operators like &, |, and ! are overloaded for logical and bitwise operations.

Note: ***In languages like C++, we can define operators to work differently for different operands. However, Java doesn't support user-defined operator overloading.***

Method overriding on the other hand, occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be **overridden**.

Rules for Method Overriding

- The method signature i.e. method name, parameter list and return type have to match exactly.
- The overridden method can widen the accessibility but not narrow it, i.e. if it is private in the base class, the child class can make it public but not vice versa.

Suppose the same method is created in the superclass and its subclasses. In this case, the method that will be called depends upon the object used to call the method.

Difference between Overloading and Overriding

Method Overloading	Method Overriding
Method overloading is in the same class, where more than one method have the same name but different signatures.	Method overriding is when one of the methods in the super class is redefined in the sub-class. In this case, the signature of the method remains the same.

What is Dynamic Polymorphism?

Dynamic Polymorphism is the mechanism by which multiple methods can be defined with same name and signature in the super class and subclass and the call to an overridden method are resolved at run time.

Step towards Dynamic Polymorphism

A reference variable of the super class can refer to a sub class object

```
Doctor obj = new Surgeon();
```

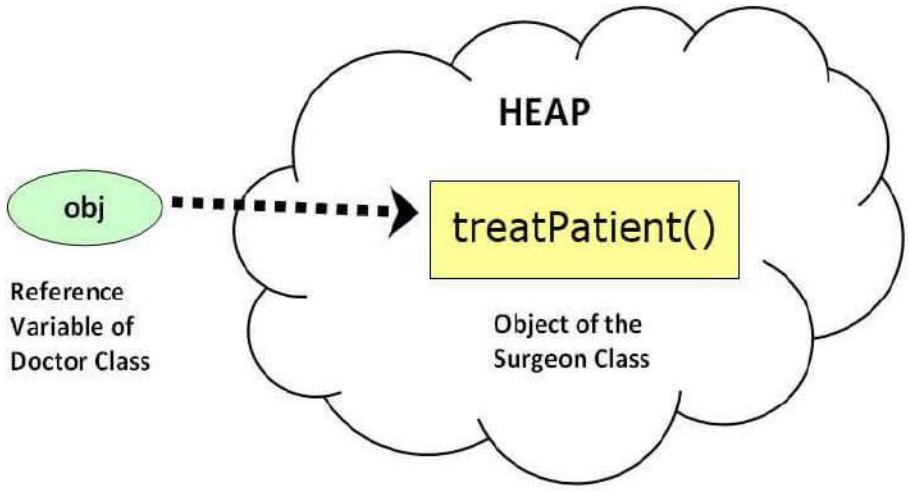
Consider the statement

```
obj.treatPatient();
```

Here the reference variable "obj" is of the parent class, but the object it is pointing to is of the child class (as shown in the diagram).

obj.treatPatient() will execute treatPatient() method of the sub-class – Surgeon. If a base class reference is used to call a method, the method to be invoked is decided by the JVM, depending on the object the reference is pointing to.

For example, even though **obj** is a reference to **Doctor**, it calls the method of **Surgeon**, as it points to a Surgeon object. This is decided during run-time and hence termed ***dynamic*** or ***run-time polymorphism***



Another example of method overriding is as follows.

Example 1: Dynamic Polymorphism → Method Overriding

```
class Animal {
   public void makeSound(){
      System.out.println("Animal sound...");
   }
}

class Dog extends Animal {

   public void makeSound() {
      System.out.println("Dog: Bark bark...");
   }
}

class Cat extends Animal {

   public void makeSound() {
      System.out.println("Cat :Meow meow...");
   }
}
class Main {
   public static void main(String[] args) {
     Dog  d1 = new Dog();
     d1.makeSound();
```

```
     Cat c1 = new Cat();
     c1.makeSound();
     System.out.println("\nwhat if we create object of parent: Animal??");
     System.out.println("Do you think the output change??\n");

     Animal a=new Dog();
     a.makeSound();
     a=new Cat();
     a.makeSound();
     System.out.println("Great...This is called dynamic polymorphism
");

   }
}
```

Output:

```
Dog: Bark bark. . .
Cat: Meow-meow. . .

what if we create object of parent: Animal??
Do you think the output change??
Dog: Bark bark...
Cat: Meow meow...
Great...This is called dynamic polymorphism
```

In the above example, the method makeSound() has different implementations in two different classes. When we run the program,

- the expression d1.makeSound() will call the method of Dog class. It is because d1 is an object of the Dog class.
- the expression c1.makeSound() will call the method of Cat class. It is because c1 is an object of the cat class.
- But the call a.makeSound() calls method based on referenced class object

Virtual Methods

In this section, I will show you how the behavior of overridden methods in Java allows you to take advantage of polymorphism when designing your classes.

We already have discussed method overriding, where a child class can override a method in its parent. An overridden method is essentially hidden in the parent class, and is not invoked unless the child class uses the super keyword within the overriding method.

Example

```
public class Employee {
  private String name;
  private String address;
  private int number;
  public Employee(String name, String address, int number) {
    System.out.println("Constructing an Employee");
    this.name = name;
    this.address = address;
    this.number = number;
  }
  public void mail_Cheque() {
    System.out.println("Mailing a Cheque to " + this.name + " " + this.address);
  }
  public String toString() {
    return name + " " + address + " " + number;
  }
  public String getName() {
    return name;
  }
```

```
  public String getAddress() {
    return address;
  }
  public int getNumber() {
    return number;
  }
  public void setAddress(String newAddress) {
    address = newAddress;
  }
}
```

Now suppose we extend Employee class as follows:

```
public class Salary extends Employee {
  private double salary; // Annual salary
  public Salary(String name, String address, int number, double salary) {
    super(name, address, number);
    setSalary(salary);
  }
  public void mail_ Cheque() {
    System.out.println("Within mail_Cheque  of Salary class ");
    System.out.println("Mailing Cheque  to " + getName() + " with salary " + salary);
  }
  public double getSalary() {
    return salary;
  }
  public void setSalary(double newSalary) {
    if(newSalary >= 0.0) {
      salary = newSalary;
    }
```

```
 }
  public double computePay() {
    System.out.println("Computing salary pay for " + getName());
    return salary;
  }
}
```

Now, you study the following program carefully and try to determine its output –

```
public class VirtualDemo {
  public static void main(String [] args) {
    Salary s = new Salary("Mohd Ali", "DHA Phase 4", 3, 3600.00);
    Employee e = new Salary("M. Azeem", "i-10/4, ISB", 2, 2400.00);
    System.out.println("Call mailCheque using Salary reference --");
    s.mailCheque();
    System.out.println(" Call mailCheque using Employee reference--");
    e.mailCheque();
  }}
```

This will produce the following result

```
Constructing an Employee
Constructing an Employee
This will produce the following result –
Call mailCheque using Salary reference --
Within mailCheque of Salary classs
Mailing checque to Mohd Ali with salary 3600.0
Call mailCheque using Employee reference--
Within mailCheque of Salary class
Mailing cheque M. Azeem with salary 2400.0
```

Here, we instantiate two Salary objects, one using a Salary reference **s**, and the other using an Employee reference **e**.

While invoking *s.mailCheque()*, the compiler sees mail*Cheque*() in the Salary class at compile time, and the JVM invokes **mailCheque**() in the Salary class at run time.

mailCheque () on **e** is quite different because **e** is an Employee reference. When the compiler sees ***e.mailCheque*** *()*, the compiler sees the **mailCheque** () method in the Employee class.

Here, at compile time, the compiler used **mailCheque** () in Employee to validate this statement. At run time, however, the JVM invokes **mailCheque** () in the Salary class.

This behavior is referred to as virtual method invocation, and these methods are referred to as virtual methods. An overridden method is invoked at run time, no matter what data type the reference is that was used in the source code at compile time.

Abstraction

In Object-oriented programming, *abstraction is a process of hiding the implementation details from the user, only the functionality will be provided to the use*r. In other words, the user will have the information on ***what the object does*** instead of ***how it does it***. For example, when you consider the case of e-mail, complex details such as what happens as soon as you send an e-mail, the protocol your e-mail server uses are hidden from the user. Therefore, to send an e-mail you just need to type the content, mention the address of the receiver, and click send.

In Java, abstraction is achieved using Abstract classes and interfaces.

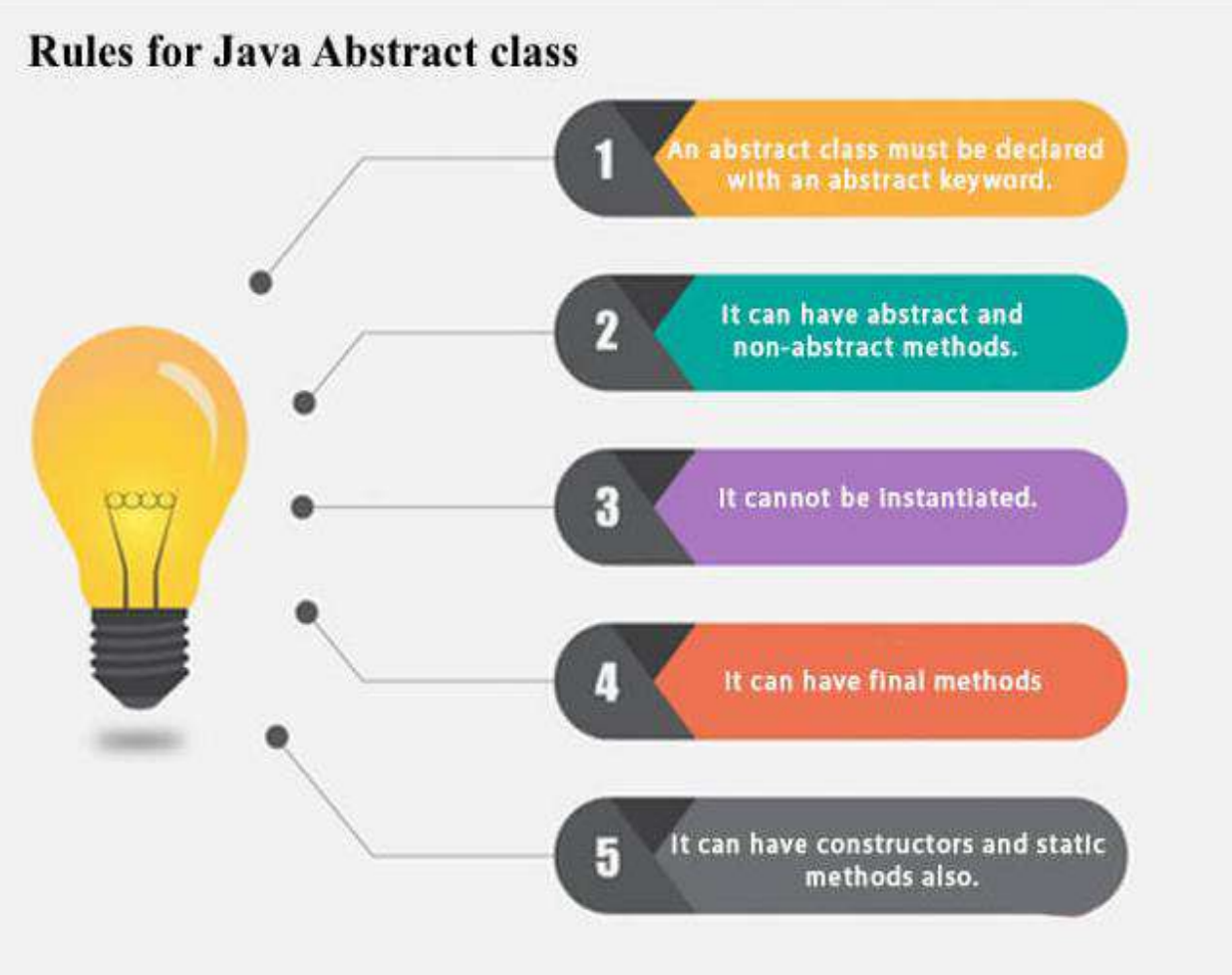
Abstract Class

A class which contains the **abstract** keyword in its declaration is known as abstract class.

Abstract classes may or may not contain *abstract methods*, i.e., methods without body (public void get();)

- But, if a class has at least one abstract method, then the class **must** be declared abstract.
- If a class is declared abstract, it cannot be instantiated.

- To use an abstract class, you have to inherit it from another class, provide implementations to the abstract methods in it.
- If you inherit an abstract class, you have to provide implementations to all the abstract methods in it.



Example

This section provides you an example of the abstract class. To create an abstract class, just use the **abstract** keyword before the class keyword, in the class declaration.

```
public abstract class Employee {
  private String name;
  private String address;
  private int number;
  public Employee(String name, String address, int number) {
```

```
    System.out.println("Constructing an Employee");
    this.name = name;
    this.address = address;
    this.number = number;
  }
  public double computePay() {
   System.out.println("Inside Employee computePay");
   return 0.0;
  }
public void mailCheck() {
    System.out.println("Mailing a check to " + this.name + " " + this.address);
  }
  public String toString() {
    return name + " " + address + " " + number;
  }
public String getName() {
    return name;
  }
public String getAddress() {
    return address;
  }
  public void setAddress(String newAddress) {
    address = newAddress;
  }
  public int getNumber() {
    return number;
  }
}
```

You can observe that except abstract methods the Employee class is same as normal class in Java. The class is now abstract, but it still has three fields, seven methods, and one constructor.

Now you can try to instantiate the Employee class in the following way –

```
/* File name : AbstractDemo.java */
public class AbstractDemo {

  public static void main(String [] args) {
    /* Following is not allowed and would raise error */
    Employee e = new Employee("Ali ahmed.", "H-9/2", 43000);
    System.out.println("\n Call mailCheck using Employee reference--");
    e.mailCheck();
  }
}
```

When you compile the above class, it gives you the following error –

Employee.java:46: Employee is abstract; cannot be instantiated

```
    Employee e = new Employee("Ali ahmed.", "H-9/2", 43000);
```

Inheriting Abstract Class

We can inherit the properties of Employee class just like concrete class in the following way –

Example

```
/* File name : Salary.java */
public class Salary extends Employee {
  private double salary;   // Annual salary
  public Salary(String name, String address, int number, double salary) {
    super(name, address, number);
    setSalary(salary);
```

```
  }
  public void mailCheck() {
    System.out.println("Within mailCheck of Salary class ");
    System.out.println("Mailing check to " + getName() + " with salary " + salary);
  }
   public double getSalary() {
    return salary;
  }
  public void setSalary(double newSalary) {
    if(newSalary >= 0.0) {
      salary = newSalary;
    }
  }
  public double computePay() {
    System.out.println("Computing salary pay for " + getName());
    return salary;
  }
}
```

Here, you cannot instantiate the Employee class, but you can instantiate the Salary Class, and using this instance you can access all the three fields and seven methods of Employee class as shown below.

```
public class AbstractDemo {
  public static void main(String [] args) {
    Salary s = new Salary("Mohd Mohtashim", "Ambehta, UP", 3, 3600.00);
    Employee e = new Salary("John Adams", "Boston, MA", 2, 2400.00);
    System.out.println("Call mailCheck using Salary reference --");
    s.mailCheck();
    System.out.println("\n Call mailCheck using Employee reference--");
```

```
    e.mailCheck();
  }
}
```

This produces the following result

Output

Constructing an Employee

Constructing an Employee

Call mailCheque using Salary reference --

Within mailCheque of Salary class

Mailing cheque to Mohd Mohtashim with salary 3600.0

Call mailCheque using Employee reference--

Within mailCheque of Salary class

Mailing cheque to John Adams with salary 2400.0

Abstract Methods

If you want a class to contain a particular method but you want the actual implementation of that method to be determined by child classes, you can declare the method in the parent class as an abstract.

- Abstract keyword is used to declare the method as abstract.
- You have to place the abstract keyword before the method name in the method declaration.
- An abstract method contains a method signature, but no method body.
- Instead of curly braces, an abstract method will have a semoi colon (;) at the end.

Following is an example of the abstract method.

Example

```
public abstract class Employee {
  private String name;
  private String address;
  private int number;
  public abstract double computePay();
  // Remainder of class definition
}
```

Declaring a method as abstract has two consequences –

- The class containing it must be declared as abstract.
- Any class inheriting the current class must either override the abstract method or declare itself as abstract.

Note – eventually, a descendant class has to implement the abstract method; otherwise, you would have a hierarchy of abstract classes that cannot be instantiated.

Suppose Salary class inherits the Employee class, then it should implement the computePay() method as shown below –

```
/* File name : Salary.java */
public class Salary extends Employee {
  private double salary;   // Annual salary

  public double computePay() {
    System.out.println("Computing salary pay for " + getName());
    return salary;
  }
  // Remainder of class definition
}
```

Understanding the real scenario of Abstract class

In this example, Shape is the abstract class, and its implementation is provided by the Rectangle and Circle classes.

In this example, if you create the instance of Rectangle class, draw() method of Rectangle class will be invoked.

```
abstract class Shape{
abstract void draw();
}
//In real scenario, implementation is provided by others i.e. unknown by end user
class Rectangle extends Shape{
void draw(){
 System.out.println("drawing rectangle");
}
}
class Circle1 extends Shape{
void draw(){
System.out.println("drawing circle");
}
}
//In real scenario, method is called by programmer or user
class TestAbstraction1{
public static void main(String args[]){
Shape s=new Circle1();//In a real scenario, object is provided through method, e.g., getS
hape() method
s.draw();
}
}
```

OUTPUT: drawing circle

Example of Abstract class in java: Figure class

```
// Using run-time polymorphism.
class Figure {
     double dim1;
     double dim2;

  Figure(double a, double b) {
       dim1 = a;
       dim2 = b;
       }

  double area() {
       System.out.println("Area for Figure is undefined.");
       return 0;
       }
  }

class Rectangle extends Figure {
     Rectangle(double a, double b) {
          super(a, b);
          }


  // override area for rectangle
double area() {
     System.out.println("Inside Area for Rectangle.");
     return dim1 * dim2;
     }
}

class Triangle extends Figure {
     Triangle(double a, double b) {
          super(a, b);
          }


  // override area for right triangle
double area() {
     System.out.println("Inside Area for Triangle.");
     return dim1 * dim2 / 2;
     }
}
class FindAreas {
     public static void main(String args[]) {
          Figure f = new Figure(10, 10);
          Rectangle r = new Rectangle(9, 5);
```

```
            Triangle t = new Triangle(10, 8);
            Figure figref;
            figref = r;
            System.out.println("Area is " + figref.area());
            figref = t;
            System.out.println("Area is " + figref.area());
            figref = f;
            System.out.println("Area is " + figref.area());
    }
}
```

Output:

```
Inside Area for Rectangle.
Area is 45.0
Inside Area for Triangle.
Area is 40.0
Area for Figure is undefined.
Area is 0.0
```

Above output highlighted is ambiguous that is corrected using abstract class.

```
// Using run-time polymorphism.
abstract class Figure {
     double dim1;
     double dim2;

  Figure(double a, double b) {
       dim1 = a;
       dim2 = b;
       }
 //abstract method
 abstract  double area();
  }

class Rectangle extends Figure {
     Rectangle(double a, double b) {
          super(a, b);
          }

  // override area for rectangle
double area() {
     System.out.println("Inside Area for Rectangle.");
     return dim1 * dim2;
     }
```

```
}

class Triangle extends Figure {
     Triangle(double a, double b) {
          super(a, b);
          }


  // override area for right triangle
double area() {
     System.out.println("Inside Area for Triangle.");
     return dim1 * dim2 / 2;
     }
}
public class FindAreas {
     public static void main(String args[]) {

          Rectangle r = new Rectangle(9, 5);
         Triangle t = new Triangle(10, 8);
         Figure figref;
         figref = r;
         System.out.println("Area is " + figref.area());
         figref = t;
         System.out.println("Area is " + figref.area());


    }
}
```

Another example of Abstract class in java: Bank Class

```
abstract class Bank{
abstract int getRateOfInterest();
}
class SBI extends Bank{
int getRateOfInterest(){return 7;}
}
class PNB extends Bank{
int getRateOfInterest(){return 8;}
}

class TestBank{
public static void main(String args[]){
```

```
Bank b;
b=new SBI();
System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
b=new PNB();
System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
}}
```

Why we need an abstract class?

Let's say we have a class Animal that has a method sound() and the subclasses of it like Dog, Lion, Horse, Cat etc.

- Since the animal sound differs from one animal to another, there is no point to implement this method in parent class.
- This is because every child class must override this method to give its own implementation details, like Lion class will say "Roar" in this method and Dog class will say "Woof".
- Thus, making this method abstract would be the good choice as by making this method abstract we force all the sub classes to implement this method(otherwise you will get compilation error), also we need not to give any implementation to this method in parent class.
- Since the Animal class has an abstract method, you must need to declare this class abstract.
- Now each animal must have a sound, by making this method abstract we made it compulsory to the child class to give implementation details to this method.
- This way we ensures that every animal has a sound.

Rules

Note 1: As we seen in the above example, there are cases when it is difficult or often unnecessary to implement all the methods in parent class. In these cases, we can declare the parent class as abstract, which makes it a special class which is not complete on its own.

A class derived from the abstract class must implement all those methods that are declared as abstract in the parent class.

Note 2: Abstract class cannot be instantiated which means you cannot create the object of it. To use this class, you need to create another class that extends this this class and provides the implementation of abstract methods, then you can use the object of that child class to call non-abstract methods of parent class as well as implemented methods(those that were abstract in parent but implemented in child class).

Note 3: If a child does not implement all the abstract methods of abstract parent class, then the child class must need to be declared abstract as well.

Do you know? Since abstract class allows concrete methods as well, *it does not provide 100% abstraction*. You can say that it provides partial abstraction. Abstraction is a process where you show only "relevant" data and "hide" unnecessary details of an object from the user. *Interfaces on the other hand are used for 100% abstraction* (Discuss in later classes).

Why can't we create the object of an abstract class?

Because these classes are

- incomplete, they have abstract methods that have no body so if java allows you to create object of this class then if someone calls the abstract method using that object then What would happen? There would be no actual implementation of the method to invoke.

- Also because an object is concrete. An abstract class is like a template, so you have to extend it and build on it before you can use it.

Assignment #13

Consider the following Main class and write required classes using the concept “abstract classes and methods”. Example output is given below.

***Note**: viewAccountNumber() is an abstract method while CheckingAccount and SavingsAccount are non-abstract.*

```
public class Week14 {
}
class Main {
    public static void main(String[] args) {
        CheckingAccount  aliCheckingAccount = new CheckingAccount();
        aliCheckingAccount.viewAccountNumber();
        SavingsAccount  aliSavingsAccount = new SavingsAccount();
        aliSavingsAccount.viewAccountNumber();
    }
}
```

Example Output

```
Checking account number: #1932042555
Savings account number: #1932042777
```