



# CS205 Object Oriented Programming in Java

## Module 2 - Core Java Fundamentals (Part 11)

**Prepared by**

**Renetha J.B.**

AP

Dept.of CSE,

Lourdes Matha College of Science and Technology

# Topics



- Core Java Fundamentals:
  - ✓ Inheritance :
    - ✓ Abstract Classes and Methods
    - ✓ using *final* with Inheritance.

# Abstract Classes and Methods



- Sometimes we may want to create a **superclass** that only defines a generalized form which will be shared by all of its subclasses and leaves the implementation to be filled by each subclass.
  - To ensure that a **subclass should override all necessary methods(implementations)**, we have to make them **abstract methods in superclass**.
- For making a method an **abstract method** we have use **abstract** type modifier.

# Abstract Classes and Methods(contd.)



- Abstract methods have **no implementation(function body)** in **the superclass** .
  - so they are also called as **subclasser responsibility**
  - the **implementation** should be there in subclasses by **overriding** those methods.
- To declare an **abstract method in superclass**, syntax is :

**abstract** *type name(parameter-list);*

- The semicolon ; after the function header shows that abstract function has no body in superclass.

# Abstract Classes and Methods(contd.)



- **ABSTRACT CLASS**

- Any **class** that contains one or more abstract methods must also be declared **abstract**.
- To declare a class abstract, use the **abstract** keyword in front of the class keyword at the beginning of the class declaration.

**abstract** class classname

```
{  
//members.abstract or nonabstract method  
}
```

- Abstract class can have non abstract methods(concrete methods) also.

# Abstract Classes and Methods(contd.)



- Abstract classes cannot be instantiated using **new** operator.
  - i.e. **Objects are not created** from abstract class.
  - Such objects would be *useless*, because an abstract class is not fully defined.
- There are **no** abstract constructors, or **no** abstract static methods.
- Any **subclass** of an **abstract class** must either implement all of the abstract methods in the superclass, *or* it should be declared abstract class.



// A Simple demonstration of abstract  
with abstract and concrete  
methods.

```
abstract class A
{
    abstract void callme();

    void callmetoo()
    {
        System.out.println("concrete method.");
    }
}

class B extends A {
    void callme() {
        System.out.println("callme in B");
    }
}
```

```
class AbstractDemo {
    public static void main(String args[])
    {
        B b = new B();
        b.callme();
        b.callmetoo();
    }
}
```

OUTPUT  
callme in B  
concrete method.

# Abstract Classes(contd.)



- Although abstract classes cannot be used to instantiate objects, abstract classes can be used to create object references,

```
superclassname superclassobjectreference ;  
superclassobjectreference =subclassobjectreference;
```

- Java's *run-time polymorphism(dynamic binding)* is *implemented through the use of superclass references.*



**// DYNAMIC(run-time)**  
**BINDING(polymorphism).**

```
abstract class Figure
{
    double dim1;
    double dim2;
    Figure(double a, double b)
    {
        dim1 = a;
        dim2 = b;
    }
    abstract double area();
}

class Rectangle extends Figure
{
    Rectangle(double a, double b)
    {
        super(a, b);
    }
    double area()
    {
        System.out.println("Rectangle Area");
        return dim1 * dim2;
    }
}
```

```
class Triangle extends Figure {
    Triangle(double a, double b)
    {
        super(a, b);
    }
    double area()
    {
        System.out.println("Triangle Area");
        return dim1 * dim2 / 2;
    }
}

class AbstractAreas {
    public static void main(String args[]) {
        // Figure f = new Figure(10, 10); // illegal
        Rectangle r = new Rectangle(9, 5);
        Triangle t = new Triangle(10, 8);
        Figure figref; // superclass object reference
        figref = r; //figref refers to object of Rectangle
        System.out.println("Area is " + figref.area());
        figref = t; //figref refers to object of Triangle
        System.out.println("Area is " + figref.area());
    }
}
```



# Dynamic binding(program)contd.



## OUTPUT

```
Rectangle Area  
Area is 45.0  
Triangle Area  
Area is 40.0
```

- Here all subclasses of abstract class **Figure** must override abstract method `area()`.
- Here the statement **Figure figref;** is not creating object but creating an object reference.
- The statement **figref = r;** means that superclass reference `figref` now points to subclass(`Rectangle`) object `r`. So the value of `dim1` and `dim2` are the values in `r`. The statement **figref.area()** will call `area()` method in that subclass(`Rectangle`)
- The statement **figref = t;** means that superclass reference `figref` now points to subclass(`Triangle`) object `t`. So the value of `dim1` and `dim2` are the values in `t`. The statement **figref.area()** will call `area()` method in that subclass(`Triangle`)

# Using final with Inheritance



- Use of **final** keyword
  - **final** can be used to create the equivalent of a named constant(*final variable*). E.g. **final** int TOTAL=0;
  - **final** helps to prevent overriding in inheritance
  - **final** helps to prevent inheritance.

# Using final with Inheritance



- Using **final** to **Prevent Overriding**
  - If we don't want to allow subclass to override a method of supeclases, we can use **final** as a modifier at the start of its method declaration in superclass.
  - Methods declared as **final** cannot be overridden by subclass.

# Using **final** to Prevent Overriding(contd.)



```
class A {  
    final void show()  
    { System.out.println("This is a final method.");  
    }  
}  
class B extends A {  
    void show() // ERROR! Can't override.  
    { System.out.println("Illegal!");  
    }  
}
```

Here show() method is declared as **final** in A. So it cannot be overridden(redefined) in subclass B. If we try to override, COMPILE ERROR will occur in the program.

# Using **final** to Prevent Overriding(contd.)



- Methods declared as **final** can sometimes provide a **performance enhancement**:
  - The compiler is free **call them inline** because it “knows” they will not be overridden by a subclass.
- When a small **final** method is called, Java compiler can copy the bytecode for the subroutine directly inline with the compiled code of the calling method, thus *eliminating the costly overhead associated with a method call*.
- Inlining is only an option with final methods.

## Using **final** to Prevent Overriding(contd.)



- Normally, Java resolves calls to **methods** dynamically, at run time. This is called **late binding**.
- However, since **final methods** cannot be overridden, a call to final method can be resolved at compile time. This is called **early binding**.

# Using final to Prevent Inheritance



- To prevent a class from being inherited it can be declared as final.
  - We cannot create subclasses from a final class.
- Class with **final** modifier cannot be inherited. It **cannot act as superclass**.
- To make a class a final class, precede the class declaration with the modifier **final**.
- If we declare a class as **final**, it implicitly declares **all of its methods as final**.
- It is **illegal** to declare a class as both **abstract** and **final** since an abstract class is incomplete by itself.



# Using final to Prevent Inheritance

```
final class A {  
    // ...  
}
```

**// The following class is illegal.**

```
class B extends A { //ERROR!cannot create a subclass for final class A  
    // ...  
}
```

It is illegal for B to inherit A since class A is declared as **final**.

# Reference



- Herbert Schildt, Java: The Complete Reference, 8/e, Tata McGraw Hill, 2011.