Inheritance

Ebey S.Raj

Introduction

- Inheritance is one of the cornerstones of object-oriented programming because it allows the creation of hierarchical classifications.
- Using inheritance, we can create a general class that defines traits common to a set of related items.
- This class can then be inherited by other, more specific classes, each adding those things that are unique to it.
- Reusability can be achieved by Inheritance.
- The mechanism of deriving a new class from an old one is called **Inheritance.**

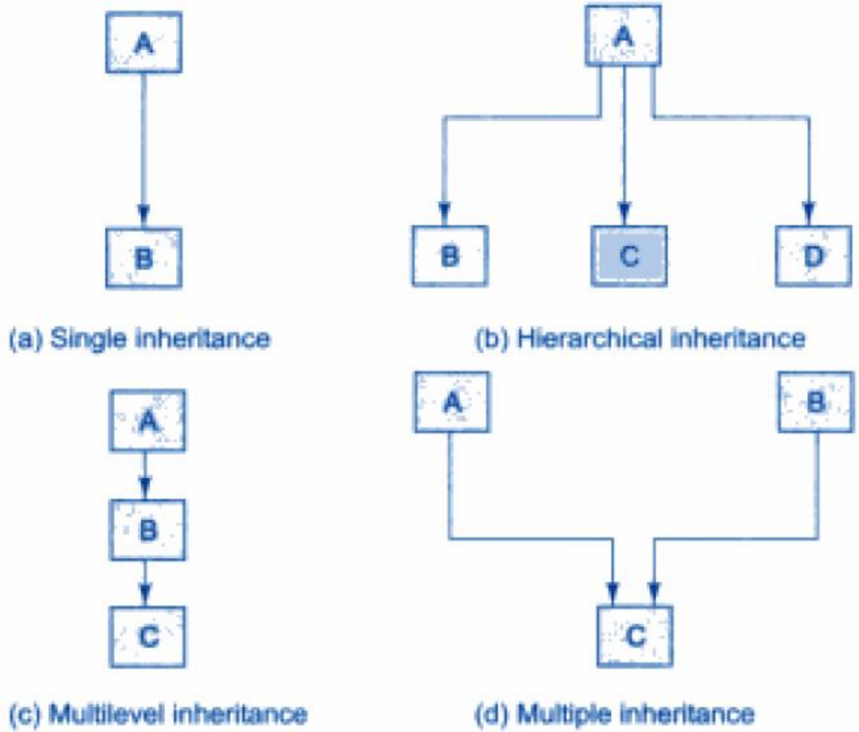
Introduction

- A class that is inherited (old class) is called a ***superclass*** *or* ***base class*** *or* ***parent class***.
- The class that does the inheriting (new class) is called a ***subclass*** *or* ***derived class*** *or* ***child class***.
  - A subclass is a specialized version of a superclass.
  - It inherits all of the members defined by the superclass and adds its own, unique elements.

Forms of inheritance

- Inheritance may take different forms:
  - Single Inheritance(Simple Inheritance)
    - Only one super class
  - Multiple Inheritance
    - Several super classes
  - Hierarchical Inheritance
    - One super class, many subclasses
  - Multilevel Inheritance
    - Derived from a derived class

Forms of inheritance



(a) Single inheritance

(b) Hierarchical inheritance

(c) Multilevel inheritance

(d) Multiple inheritance

Inheritance basics

- To inherit a class, you simply incorporate the definition of one class into another by using the **extends** keyword.

Simple inheritance Example: SimpleInheritance.java

- Even though **A** is a superclass for **B**, it is also a completely independent, stand-alone class.
- The superclass can be used by itself.

Inheritance basics

- The general form of a **class** declaration that inherits a superclass is shown here:

```
class subclass-name extends superclass-name {
        // body of class
}
```

Member Access and Inheritance

- Although a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared as **private**.

Private member Access Example: Access.java

A class member that has been declared as private will remain private to its class. It is not accessible by any code outside its class, including subclasses.

Member Access and Inheritance

- A major advantage of inheritance is that once you have created a superclass that defines the attributes common to a set of objects, it can be used to create any number of more specific subclasses.
  - Each subclass can precisely tailor its own classification.

Another Inheritance Example: BoxWeightDemo.java

A Superclass Variable Can Reference a Subclass Object

- A reference variable of a superclass can be assigned a reference to any subclass derived from that superclass.
  - We will have access **only to those parts of the object defined by the superclass.**

Reference Example: RefDemo.java

It is the **type of the reference variable** — not the type of the object that it refers to — that determines what members can be accessed.

Using Super

- Problems with BoxWeightDemo example
  - Code inside the constructor of BoxWeight is duplicate code which is same code found in its superclass, which is inefficient.
    - It implies that a subclass must be granted access to these members.
  - When we want to keep data of super class private, we cannot use like the example.

Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword **super**.

Using Super

- super has two general forms.
    - The first calls the superclass’ constructor.
    - The second is used to access a member of the superclass that has been hidden by a member of a subclass.

Using super to Call Superclass Constructors

- A subclass can call a constructor defined by its superclass by use of the following form of super:

  super(*arg-list*);

  Here, *arg-list* specifies any arguments needed by the constructor in the superclass.
- super( ) must always be the **first** statement executed inside a subclass' constructor.

Super Use1 Example: SuperUse1.java

super( ) always refers to the superclass immediately above the calling class.

A Second Use for super

- The second form of **super** acts somewhat like **this**, except that it always refers to the superclass of the subclass in which it is used.
- This usage has the following general form:

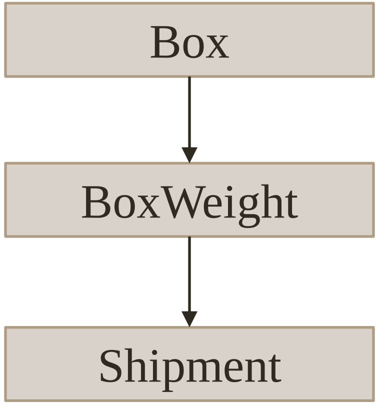
  super.member

  Here, member can be either a method or an instance variable.
- Most applicable to situations in which member names of a subclass hide members by the same name in the superclass.

Super Use2 Example: SuperUse2.java

Creating a Multilevel Hierarchy

- Use a subclass as a super class of another class.
- In this situation, each subclass inherits all the traits found in all its super classes.



[Multilevel Inheritance Example: ShipmentDemo.java]

Shipment inherits all of the traits of **BoxWeight** and **Box**, and adds a field called **cost**

When constructors are called

- Constructors are called in order of derivation, from superclass to subclass.
- Since super( ) must be the first statement executed in a subclass' constructor, this order is the same whether or not super( ) is used.
- If super( ) is not used, then the default or parameter-less constructor of each superclass will be executed.

Constructor Calling Example: ConsCalling.java

Prepared by Ebey S.Raj

Method Overriding

- In a class hierarchy, when a method in a subclass has the **same name and type signature** as a method in its superclass, then the method in the subclass is said to **override** the method in the superclass.
- When an overridden method is called from within its subclass, it will always refer to the version of that **method defined by the subclass**.
  - The version of the method defined by the superclass will be hidden.

Method Overriding Example: Override.java

Method Overriding

- If you wish to access the superclass version of an overridden method, you can do so by using **super**.

Method Overriding Example2: Override1.java

- Method overriding occurs only when the names and the type signatures of the two methods are identical.
  - If they are not, then the two methods are simply overloaded.

Different name and signature Example: NoOverride.java

Dynamic Method Dispatch

- Method overriding forms the basis for one of Java’s most powerful concepts: **Dynamic Method Dispatch.**
- Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.
- Dynamic method dispatch is important because this is how Java implements **run-time polymorphism**.

Dynamic Method Dispatch

- A superclass reference variable can refer to a subclass object.
- When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs.
- Determination of the version of overridden method is made at run time.
- When different types of objects are referred to, different versions of an overridden method will be called.

Dynamic Method Dispatch Example: Dispatch.java

It is the **type of the object being referred** to (not the type of the reference variable) that determines which version of an overridden method will be executed.

Why Overridden Methods?

- Overridden methods allow Java to support run-time polymorphism. Java implements the “**one interface, multiple methods**” aspect of polymorphism.
- By inheritance, the superclass provides all elements that a subclass can use directly.
- By overriding, the super class defines those methods that the derived class must implement on its own.

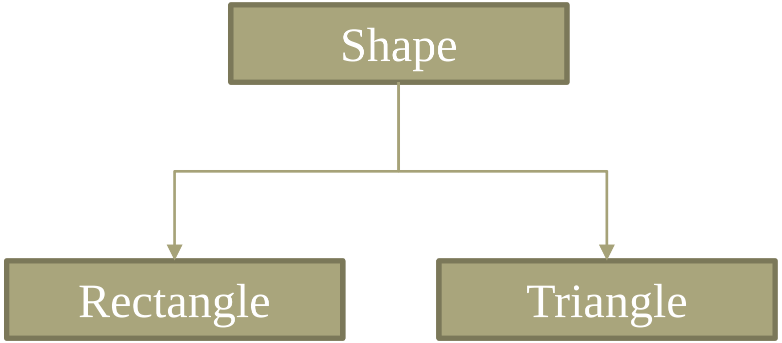
Why Overridden Methods?

- By combining inheritance with overridden methods, a superclass can define the general form of the methods that will be used by all of its subclasses.
- Dynamic, run-time polymorphism is one of the most powerful mechanisms that object oriented design brings to bear on code reuse and robustness.

Applying Method Overriding

Another Method Overriding Example: FindAreas.java

- Here we are implementing Hierarchical Inheritance.



Using Abstract Classes

- There are situations in which we will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method.
- Superclass only defines a generalized form for the methods that will be shared by all of its subclasses, leaving it to each subclass to fill in the details.
  - Eg: When a superclass is unable to create a meaningful implementation for a method.(area() in Shape class)
- Java’s solution to this problem is the **abstract method**.

Using Abstract classes

- Generalized methods should be overridden by subclasses by specifying the **abstract** type modifier.
- These methods are sometimes referred to as **subclasser responsibility** because they have no implementation specified in the superclass.
- To declare an abstract method, use this general form:

abstract return-type name(parameter-list);

Using Abstract classes

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

```
abstract class class-name{
        // one of the methods is abstract…
        // ……
}
```

Prepared by Ebey S.Raj

Using Abstract classes

- There can be **no objects** of an abstract class.
  - An abstract class **cannot be directly instantiated** with the new operator.
- Objects of an abstract class would be useless
  - Because an abstract class is not fully defined.
- We cannot declare abstract constructors, or abstract static methods.
- Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be declared abstract itself.

Abstract class Example: AbstractDemo.java

Using Abstract classes

- Java’s approach to run-time polymorphism is implemented through the use of superclass references.
- Abstract classes can be used to create object references and can be used to point to a subclass object.

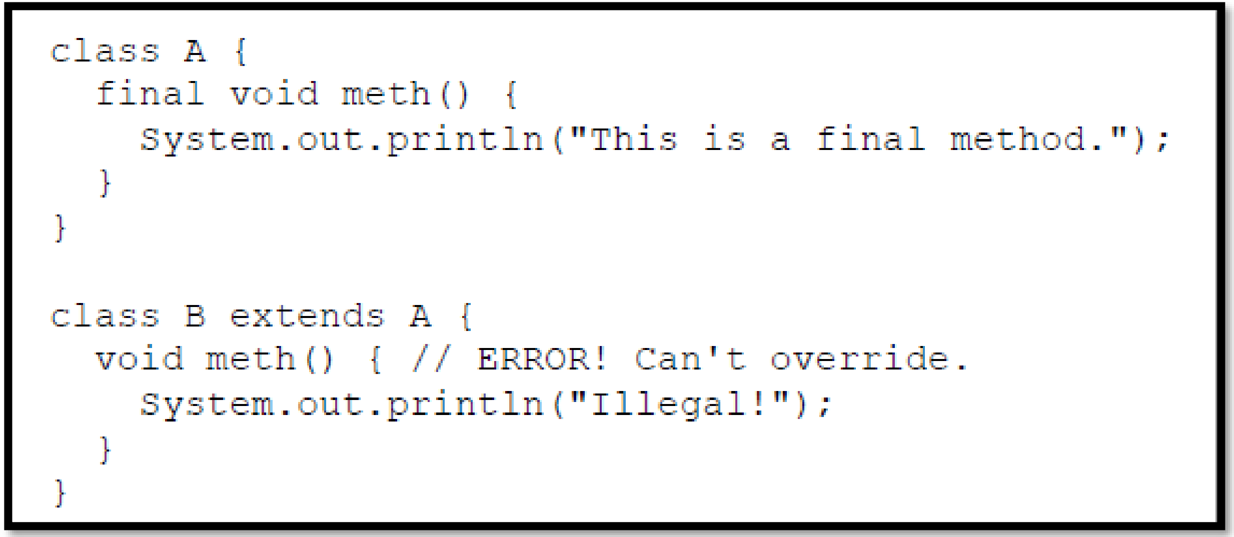
Abstract class Overriding Example: AbstractAreas.java

Using final with inheritance

- The keyword final has three uses.
  - final can be used to create the equivalent of a named constant.
  - final can be used to Prevent Overriding.
  - final can be used to Prevent Inheritance

Using final to prevent overriding

- To disallow a method from being overridden, specify **final** as a modifier at the start of its declaration.
  - Methods declared as final cannot be overridden.



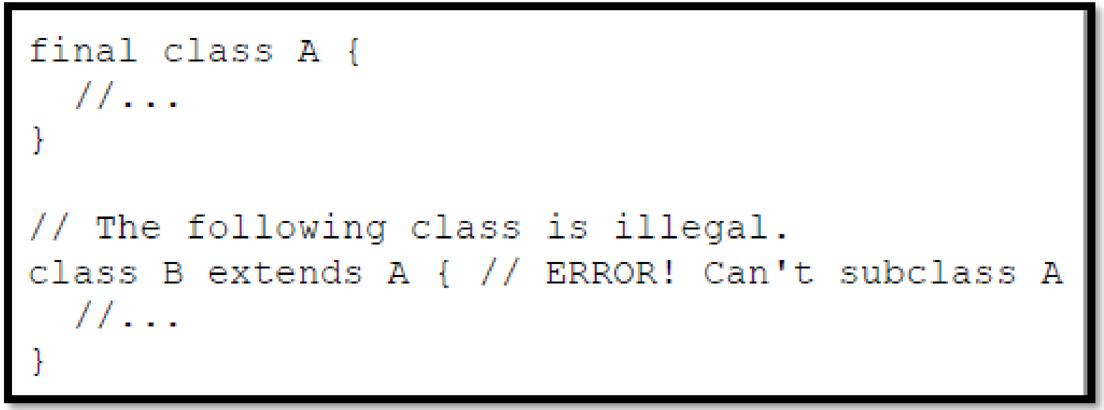
```
class A {
  final void meth() {
    System.out.println("This is a final method.");
  }
}

class B extends A {
  void meth() { // ERROR! Can't override.
    System.out.println("Illegal!");
  }
}
```

Early binding

- The compiler is free to inline calls to final methods.
  - Java compiler can copy the bytecode for the final subroutine directly inline with the compiled code of the calling method.
  - Eliminates the costly overhead associated with a method call.
  - Inlining is an option only with final methods.
- Normally, Java resolves calls to methods dynamically, at run time. This is called **late binding.**
- Since final methods cannot be overridden, a call to one can be resolved at compile time. This is called **early binding.**

Using final to prevent inheritance

- Declaring a class as **final** implicitly declares all of its methods as final.
- It prevents a class from being inherited.
- It is illegal to declare a class as both abstract and final.
  - Since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations.



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

// The following class is illegal.
class B extends A { // ERROR! Can't subclass A
  //...
}
```

References

- Lafore R., Object Oriented Programming in C++, Galgotia Publications, 2001.
- Balagurusamy, Object Oriented Programming with C++, Tata McGraw Hill, 2008.

Thank You

Ebey S.Raj

Asst Professor in IT

Govt Engg College, Sreekrishnapuram