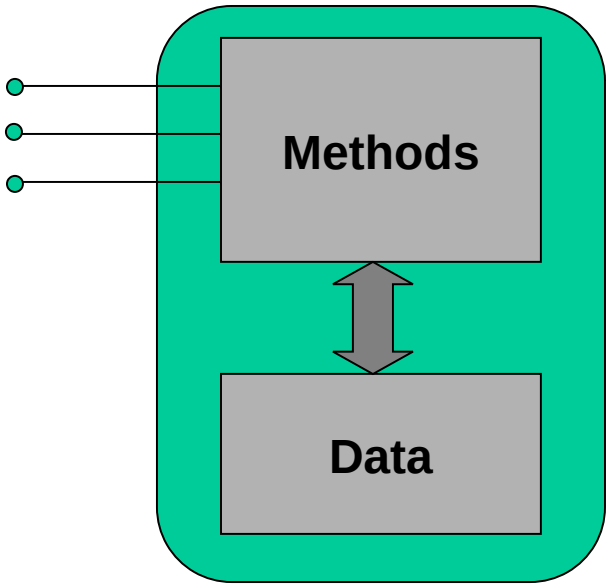
Chapter Four

Encapsulation, Inheritance, Polymorphism and Interfaces

What is encapsulation?

- Hiding the data within the class
- Making it available only through the methods
- Each object protects and manages its own data. This is called self-governing.



Why encapsulation?

- To hide the internal implementation details of your class so they can be easily changed
- To protect your class against accidental or willful mistakes
- In general encapsulation
  - separates the implementation details from the interface

Using Set and Get Methods

- A class's private fields can manipulate only by methods of that class
- How can we access those data outside?
  - Using set and get methods
- Set methods
  - **`public`** method that sets **`private`** variables
  - Does not violate notion of **`private`** data
    - Change only the variables you want
  - Called mutator methods (change value)
- Get methods
  - **`public`** method that displays **`private`** variables
  - Again, does not violate notion of **`private`** data
    - Only display information you want to display
  - Called accessor or query methods

With out Encapsulation

```
Class Person {
    String firstName;
    String lastName;
    int age;
Void display(){
    System.out.print(“welcome”);
    System.out.print(firstName);
    System.out.print(lastName);
    }
    }
```

```
class PersonTest{
public static void
    main(String args[]){
Person p1 = new Person();
p1.age = 20;
// p1.age = -21 possible
p1.lastName=”Dawit”;
p1.firstname = “Kassa”;
p1.dispaly();
}
}
```

With Encapsulation

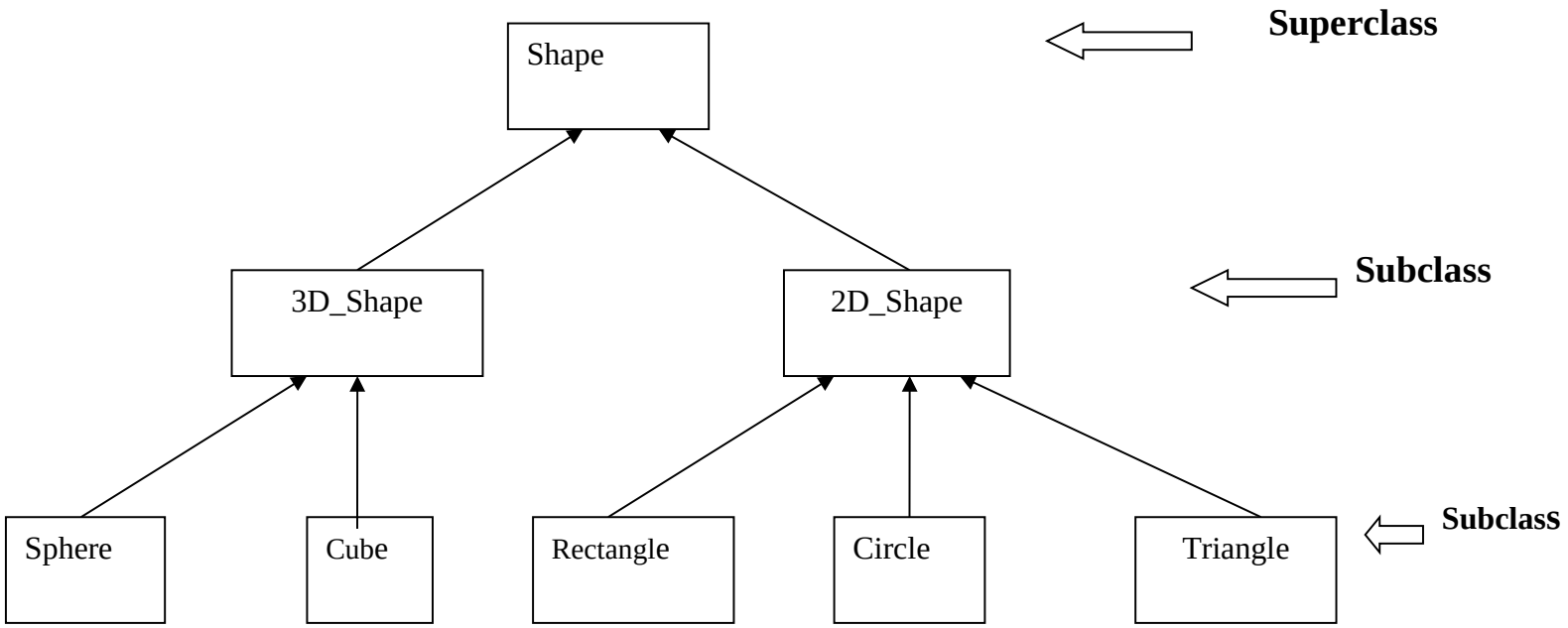
```
Class Person {
    Private String firstName;
    Private String lastName;
    Private int age;
public String getfirsName(){
      return firstName;
    }
public String getlastName(){
      return lastName;
    }
public int getAge(){
    return age;
    }
public void setAge(int i){
      If(i<=0)
          System.out.println(“wrong age”);
            else
        age = i;
    }
public setfirstName(String name)
   {
       firstName = name;
    }
public setlastName(String name){
        lastName = name;
    }
}
```

```
class PersonTest{
  public static void main(String args[]){
      Person p1 = new Person();
        p1.age = 20; // error b/c it is private
        p1.setAge(20); // correct b/c setAge() is public
        p1.setAage() = -21  //  output : wrong age
        p1.lastName=”Dawit”; // error b/c it is private
        p1.setFirstName(“Dawit);
       p1.setLastName (“Kassa”);
       p1.dispaly();
     }
}
```

Inheritance

- Inheritance is one of the cornerstones of object-oriented programming
- because it allows the creation of hierarchical classifications
- a class that is inherited is called a ***superclass***.
- The class that does the inheriting is called a ***subclass***
- A subclass inherits all of the instance variables and methods defined by the superclass and adds its own, unique elements
- Advantages of Inheritance
  - reuse
  - enhancement,
  - adaptation, etc

- Two ways of expressing relationships
  - Generalization/Specialization
    - **‘is a”** relationship
      - **Example**: Circle is a shape
  - Whole-part
    - Part of or **“has a”** relationship
      - **Example**: Employee class has a BirthDate class
    - Called **aggregation**
- Inheritance Creates an is-a relationship

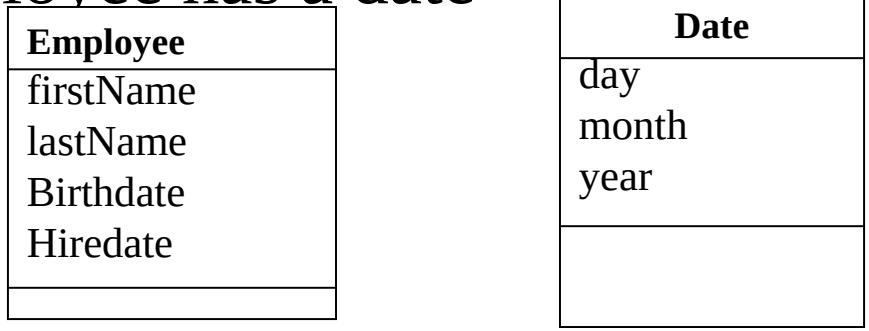


Above: is-a

Circle is a 2D_Shape

below: has-a

Employee has a date



- **Inheritance Basics**
  - To inherit a class, you simply incorporate the definition of one class into another by using the **extends** keyword
  - The general form is:

Class SubClassName extends SuperClassName

{

// body of class

}

  - Java does not support the **multiple inheritance**
  - Subclasses cannot access private members

```
// This program uses inheritance to extend Box.
class Box {
   double width;
   double height;
   double depth;
   // construct clone of an object
   Box(Box ob) { // pass object to constructor
   width = ob.width;
   height = ob.height;
   depth = ob.depth;
   }
   // constructor used when all dimensions specified
   Box(double w, double h, double d) {
   width = w;
   height = h;
   depth = d;
   }
```

```
// constructor used when no dimensions specified
Box() {
   width = -1; // use -1 to indicate
   height = -1; // an uninitialized
   depth = -1; // box
}
// constructor used when cube is created
Box(double len) {
   width = height = depth = len;
}
// compute and return volume
double volume() {
   return width * height * depth;
}
}
```

```
// Here, Box is extended to include weight.
class BoxWeight extends Box {
   double weight; // weight of box
   // constructor for BoxWeight
   BoxWeight(double w, double h, double d, double m) {
                        width = w;
                        height = h;
                        depth = d;
                        weight = m;
   }
}
```

```
class DemoBoxWeight {
   public static void main(String args[]) {
   BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
   BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
   double vol;
   vol = mybox1.volume();
   System.out.println("Volume of mybox1 is " + vol);
   System.out.println("Weight of mybox1 is “+
   mybox1.weight);
   System.out.println();
   vol = mybox2.volume();
   System.out.println("Volume of mybox2 is " + vol);
   System.out.println("Weight of mybox2 is " +
   mybox2.weight);
}
}
```

- The output from this program is shown here:

  Volume of mybox1 is 3000.0

  Weight of mybox1 is 34.3

  Volume of mybox2 is 24.0

  Weight of mybox2 is 0.076

- .

- **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.
  - For example:

```
class RefDemo {
    public static void main(String args[]) {
    BoxWeight weightbox = new BoxWeight(3, 5, 7, 8.37);
    Box plainbox = new Box();
    double vol;
    vol = weightbox.volume();
    System.out.println("Volume of weightbox is " + vol);
    System.out.println("Weight of weightbox is " +
            weightbox.weight);
    System.out.println();
```

```
        // assign BoxWeight reference to Box reference
        plainbox = weightbox;
        vol = plainbox.volume(); // OK, volume() defined in Box
        System.out.println("Volume of plainbox is " + vol);
        /* The following statement is invalid because plainbox
        does not define a weight member. */
        // System.out.println("Weight of plainbox is " +
        plainbox.weight);
        }
    }
```

- Here, **weightbox** is a reference to **BoxWeight** objects, and **plainbox** is a reference to Box objects.
- Since **BoxWeight** is a subclass of **Box,** it is permissible to assign **plainbox** a reference to the **weightbox** object**.**

- It is important to understand that 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
- When a reference to a subclass object is assigned to a superclass reference variable, you will have access only to those parts of the object defined by the superclass
- This is why **plainbox** can't access weight even when it refers to a **BoxWeight** object**.**

- **Type Casting**
  - Syntax:

    <Sub_Ref.> = (SubClassName)<sup_Ref.> ;

    For example: Person p = new Employee();

    Employee emp = (Employee)p;
  - the super class reference variable should refer to a subclass object.
  - Otherwise it will compile correctly but there will be an error at runtime

The protected Modifier

- Visibility modifiers determine which class members are inherited and which are not
- Variables and methods declared with public visibility are inherited; those with private visibility are not
- But public variables violate the principle of encapsulation
- There is a third visibility modifier that helps in inheritance situations: protected
- The **protected** modifier allows a member of a base class to be inherited into a child
- Protected visibility provides
  - more encapsulation than public visibility does
  - the best possible encapsulation that permits inheritance

```
Class Person {
    Private String firstName;
    Private String lastName;
    Private int age;
public String getfirsName(){
       return firstName;
    }
public String getlastName(){
       return lastName;
    }
public int getAge(){
      return age;
    }
public void setAge(int i)
    {
      age = i;
    }
public setfirstName(String
   name)
    {
       firstName = name;
    }
public setlastName(String name)
    {
       lastName = name;
    }
}
```

Subclass Student

```
class Student extends Person{
        private String stuId;
        public String getStuId( ){
           return stuId;
        }
            public void setStuId(String id){
           stuId = id;
        }
            Public void display(){
                        System.out.println(this.fisrtName);
                        System.out.println(this.lastname);
                        System.out.println(this age);
                        Sytsem.out.println(this.stuId);
        }

}
```

```
class StudentTest{
    public static void main(String args[]){
            Student stud1 = new Student ();
            Student stud2 = new Student();
            Person per1 = new Person ();
                System.out.println(stud1.setFirstName("Alef"));
                System.out.println(stud1.setLastName("Mulugeta"));
                System.out.println(stud1.setAge(28));
                System.out.println(stud1.getFirstName());
                System.out.println(stud1.getLastName());
                 // can not inherits class's private members
                System.out.println(stud1.firstname);
                System.out.println(stud1.getAge());
                 // can not inherits class's private members
                System.out.println(stud1.age);
```

```
            System.out.println(stud1.getFirstName());
            System.out.println(per1.getStuId());   // error
            Stud2.setFirstName(“ Kalid”);
            System.out.println(stud2.getFirstname());
        }
}
```

- **Using super**
  - Whenever a subclass needs to refer to its immediate superclass, it can do so by using of the keyword **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**
    - super(*parameter-list);*
    - *parameter-list* specifies any parameters needed by the constructor in the superclass.
    - **super( )** must always be the first statement executed inside a subclass‘ constructor.

```
// BoxWeight now uses super to initialize its Box attributes.
class BoxWeight extends Box {
                double weight; // weight of box
                // initialize width, height, and depth using super()
                BoxWeight(double w, double h, double d, double
  m) {
                        super(w, h, d); // call superclass
  constructor
                        weight = m;
                }
}
```

- When a subclass calls **super( ),** it is calling the constructor of its immediate superclass.
- **super( )** always refers to the superclass immediately above the calling class.

- The second form of **super**

  super.*member*

  - *member* can be either a method or an instance variable.
  - This second form of **super** is most applicable to situations in which member names of a subclass hide members by the same name in the superclass.

```
// Using super to overcome name hiding.
class A {
   int i;
}
// Create a subclass by extending class A.
class B extends A {
   int i; // this i hides the i in A
   B(int a, int b) {
   super.i = a; // i in A
   i = b; // i in B
}
```

```
void show() {
                    System.out.println("i in superclass: " + super.i);
                    System.out.println("i in subclass: " + i);
}
}
class UseSuper {
   public static void main(String args[]) {
                    B subOb = new B(1, 2);
                    subOb.show();
   }
}
```

- This program displays the following:

```
i in superclass: 1
i in subclass: 2
```

- **Creating a Multilevel Hierarchy**

```
class BoxWeight extends Box {
  double weight; // weight of box
  // construct clone of an object
  BoxWeight(BoxWeight ob) { // pass object to constructor
                super(ob);
                weight = ob.weight;
  }
  // constructor when all parameters are specified
  BoxWeight(double w, double h, double d, double m) {
                super(w, h, d); // call superclass constructor
                weight = m;
  }
  // default constructor
```

```
    BoxWeight() {
    super();
    weight = -1;
    }
    // constructor used when cube is created
    BoxWeight(double len, double m) {
    super(len);
    weight = m;
    }
}
class Shipment extends BoxWeight {
    double cost;
    // construct clone of an object
    Shipment(Shipment ob) { // pass object to constructor
    super(ob);
    cost = ob.cost;
    }
```

```
// constructor when all parameters are specified
Shipment(double w, double h, double d, double m, double c) {
                    super(w, h, d, m); // call superclass constructor
                    cost = c;
}
// default constructor
Shipment() {
                    super();
                    cost = -1;
}
```

```
    Shipment(double len, double m, double c) {
                        super(len, m);
                        cost = c;
    }
}
```

- **When Constructors Are Called**
  - in a class hierarchy, 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.

```
// Demonstrate when constructors are called.
// Create a super class.
class A {
  A() {
        System.out.println("Inside A's
  constructor.");
  }
}
// Create a subclass by extending class A.
class B extends A {
  B() {
        System.out.println("Inside B's
  constructor.");
  }
}
```

```
// Create another subclass by extending B.
class C extends B {
  C() {
        System.out.println("Inside C's
  constructor.");
  }
}
class CallingCons {
  public static void main(String args[]) {
        C c = new C();
  }
}
```

- The output is
  - Inside A's constructor
  - Inside B's constructor
  - Inside C's constructor

Method Overriding

- 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

```
// Method overriding.
class A {
  int i, j;
  A(int a, int b) {
  i = a;
  j = b;
  }
  // display i and j
  void show() {
  System.out.println("i and j: "+i+" "+j);
  }
}
```

```
class B extends A {
  int k;
  B(int a, int b, int c) {
  super(a, b);
  k = c;
  }
  // display k – this overrides show() in A
  void show() {
  System.out.println("k: " + k);
  }
}
class Override {
  public static void main(String args[]) {
  B subOb = new B(1, 2, 3);
  subOb.show(); // this calls show() in B
  }
}
```

The output

k: 3

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

```
class B extends A {
  int k;
  B(int a, int b, int c) {
  super(a, b);
  k = c;
  }
  void show() {
  super.show(); // this calls A's show()
  System.out.println("k: " + k);
  }
}
```

- output:

```
i and j: 1 2   k: 3
```

- The access specifiers for the overriding method can allow more access than the overridden method, but not less
  - **For example**, a protected method in the superclass can be made public but not private
- it is a syntax error to override a method with a more restricted access modifier
- **Overloading vs. Overriding**
  - Don't confuse the concepts of overloading and overriding
  - Overloading deals with multiple methods with the same name in the same class, but with different signatures
  - Overriding deals with two methods, one in a parent class and one in a child class, that have the same signature
  - Overloading lets you define a similar operation in different ways for different data
  - Overriding lets you define a similar operation in different ways for different object types

- **Dynamic Method Dispatch**
  - is the mechanism by which a call to an overridden function is resolved at run time, rather than compile time.
  - Dynamic method dispatch is important because this is how Java implements **run-time polymorphism**.
  - Method overriding forms the basis for this concept
  - 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
  - When different types of objects are referred to, different versions of an overridden method will be called

```
// Dynamic Method Dispatch
class A {
  void callme() {
  System.out.println("Inside A's callme method");
  }
}
class B extends A {
  //override callme()
  void callme() {
  System.out.println("Inside B's callme method");
  }
}
class C extends A {
  // override callme()
  void callme() {
  System.out.println("Inside C's callme method");
  }
}
```

```
class Dispatch {
 public static void main(String args[]) {
 A a = new A(); // object of type A
 B b = new B(); // object of type B
 C c = new C(); // object of type C
 A r; // obtain a reference of type A
 r = a; // r refers to an A object
 r.callme(); // calls A's version of callme
 r = b; // r refers to a B object
 r.callme(); // calls B's version of callme
 r = c; // r refers to a C object
 r.callme(); // calls C's version of callme
 }
}
```

- The output from the program is shown here:

```
Inside A's callme method
Inside B's callme method
Inside C's callme method
```

- **Why Overridden Methods?**
  - Overridden methods are another way that Java implements the "one interface, multiple methods" aspect of polymorphism
  - 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.

Abstract Classes

- In certain situation, we want to properties of classes to be always extended and used. Such classes are called Abstract Classes.
- An ***Abstract*** class is a conceptual class.
- An Abstract class cannot be instantiated – objects cannot be created.
- Abstract classes provides a common root for a group of classes, nicely tied together in a package:
- general form:

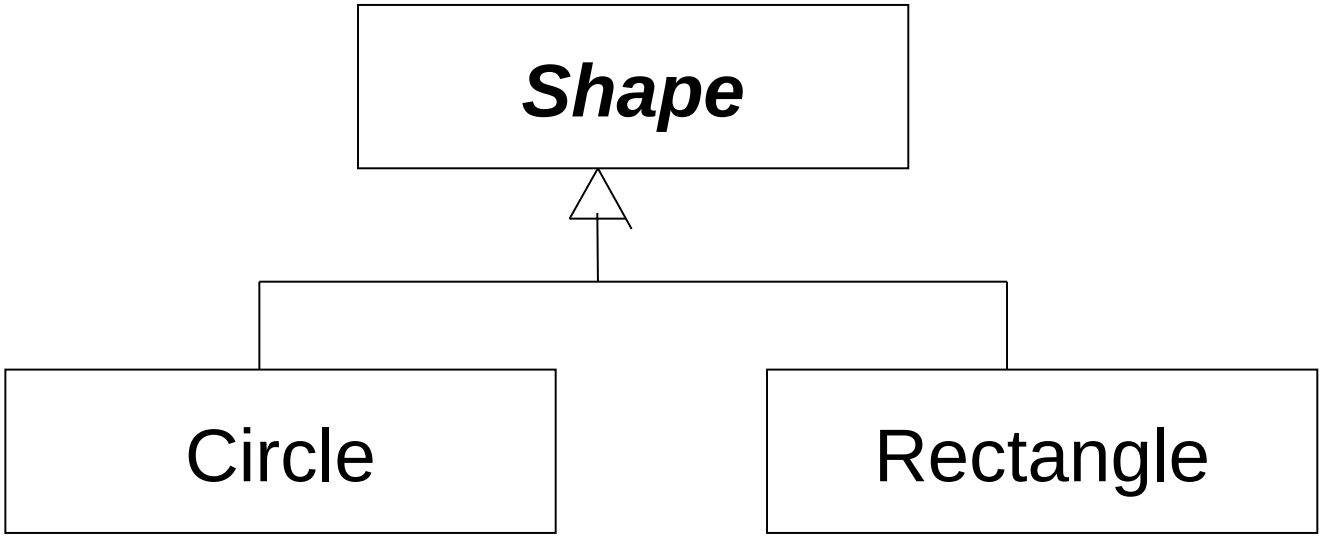
```
abstract class ClassName{ }
```

- Abstract class contains one or more abstract methods.

```
abstract type methodName([parametr list]);
```

- The abstract methods of an abstract class must be defined or implemented in its subclass (called **concrete class**)

- 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 itself declared **abstract.**
- Example: Shape is an abstract class



The Shape Abstract Class

```
public abstract class Shape {
      public abstract double area();
      public void move(){// non-abstract method
            // implementation
     }
}
```

```
public Circle extends Shape {
  protected double r;
  protected static final double PI = 3.1415926535;
  public Circle() { r = 1.0; }
  public double area() {
     return PI * r * r;
  }
…
}
public Rectangle extends Shape {
  protected double w, h;
  public Rectangle() {
     w = 0.0; h=0.0;
  }
  public double area() {
    return w * h;
  }
}
```

Abstract Classes Properties

- A class with one or more abstract methods is automatically abstract and it cannot be instantiated.
- A class declared abstract, even with no abstract methods can not be instantiated.
- A subclass of an abstract class can be instantiated if it overrides all abstract methods by implementing them.
- A subclass that does not implement all of the superclass abstract methods is itself abstract; and it cannot be instantiated.
- Although abstract classes cannot be used to instantiate objects, they can be used to create object references
- it must be possible to create a reference to an abstract class so that it can be used to point to a subclass object.

Using final with 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!");
  }
}
```

- Methods declared as **final** can sometimes provide a performance enhancement:
  - The compiler is free to ***inline*** calls to them because it "knows" they will not be overridden by *a* subclass.
- When a small **final** function is called, often the 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 Inheritance**
  - We can prevent an inheritance of classes by other classes by declaring them as **final** classes.
  - Declaring a class as **final** implicitly declares all of its methods as **final,** too**.**
  - 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.
  - This is achieved in Java by using the keyword **final** as follows:

```
final class Marks {
  // members
}
final class Student extends Person {
  // members
}
```

  - Any attempt to inherit these classes will cause an error.

Final Class Members

- All methods and variables can be overridden by default in subclasses.
- This can be prevented by declaring them as final using the keyword “final” as a modifier. For example:

```
final int marks = 100;
final void display(){ ...}
```

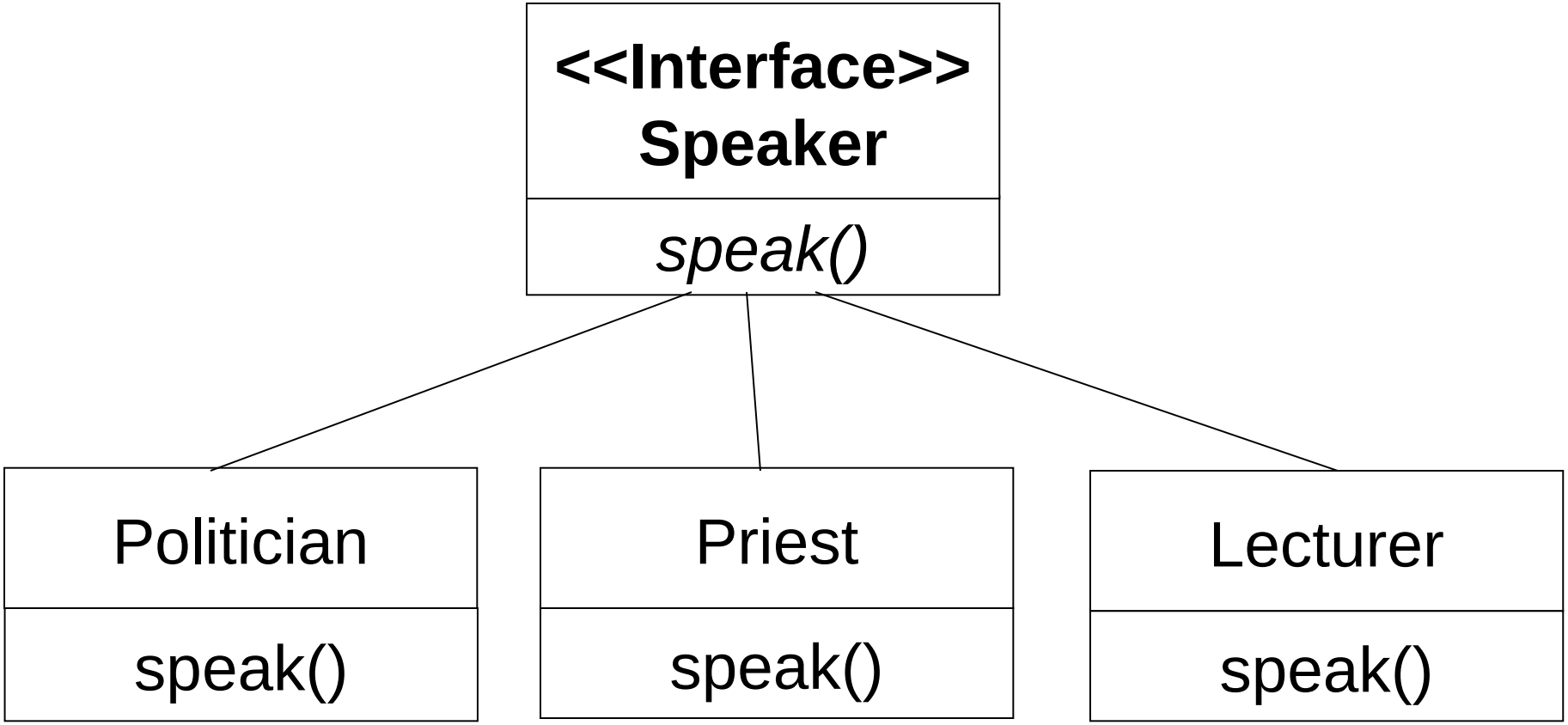
- This ensures that functionality defined in this method cannot be altered any. Similarly, the value of a final variable cannot be altered.

Interfaces

- ***Interface*** is a conceptual entity similar to a Abstract class.
- using **interface,** you can specify what a class must do, but not how it does it.
- Can contain only **constants (final variables)** and **abstract method** (no implementation) **-** Different from Abstract classes.
- Use when a number of classes share a common interface.
- Each class should implement the interface.
- Any number of classes can implement an **interface.**
- One class can implement any number of interfaces
- Interfaces are designed to support dynamic method resolution at run time

- disconnect the definition of a method or set of methods from the inheritance hierarchy
- Since interfaces are in a different hierarchy from classes, it is possible for classes that are unrelated in terms of the class hierarchy to implement the same interface.
- A class can implement any number of interfaces, but cannot extend more than one class at a time.
- Therefore, interfaces are considered as an **informal way of realizing multiple inheritance** in Java.
- Access modifier is either ***public*** *or* ***not used***
- All methods and variables are implicitly **public** if the interface, itself, is declared as **public**

Interface - Example



Interfaces Definition

- Syntax (appears like abstract class):

```
interface InterfaceName {
   // Constant/Final Variable Declaration
   // Methods Declaration – only abstract method
}
```

- Example:

```
interface Speaker {
   public void speak( );
}
```

Implementing Interfaces

- Interfaces are used like super-classes whose properties are inherited by classes.
- To implement an interface, include the **implements** clause in a class definition, and then create the methods defined by the interface
- The general form of a class that includes the implements clause:

```
class ClassName implements InterfaceName [, InterfaceName2, …]
{
    // Body of Class
}
```

- If a class implements more than one interface, the interfaces are separated with a **comma**
- The methods that implement an interface must be declared **public**

Implementing Interfaces Example

```
class  Politician implements Speaker {
      public void speak(){
         System.out.println(“Talk politics”);
      }
}
```

```
class  Priest implements Speaker {
      public void speak(){
         System.out.println(“Religious Talks”);
      }
}
```

```
class  Lecturer implements Speaker {
   public void speak(){
     System.out.println(“Talks Object Oriented Design and
                              Programming!”);
      }
}
```

- **Partial Implementations**
  - If a class includes an interface but does not fully implement the methods defined by that interface, then that class must be declared as **abstract.**
- **Extending Interfaces**
  - Like classes, interfaces can also be extended.
  - This is achieved by using the keyword **extends** as follows:

```
interface InterfaceName2 extends
InterfaceName1 {
    // Body of InterfaceName2
}
```

```
// One interface can extend another.
interface A {
  void meth1();
  void meth2();
}
// B now includes meth1() and meth2() –
  // it adds meth3().
interface B extends A {
  void meth3();
}
// This class must implement all of A and B
class MyClass implements B {
  public void meth1() {
        System.out.println("Implement meth1().");
  }
  public void meth2() {
        System.out.println("Implement meth2().");
  }
```

```
public void meth3() {
        System.out.println("Implement meth3().");
  }
}
class IFExtend {
  public static void main(String arg[]) {
        MyClass ob = new MyClass();
        ob.meth1();
        ob.meth2();
        ob.meth3();
  }
}
```

Inheritance and Interface Implementation

- A general form of interface implementation:

```
class ClassName extends SuperClass implements
                        InterfaceName [, InterfaceName2, …]{
   // Body of Class
}
```

- This shows a class can extended another class while implementing one or more interfaces.
- It appears like a multiple inheritance (if we consider interfaces as special kind of classes with certain restrictions or special features).

Polymorphism

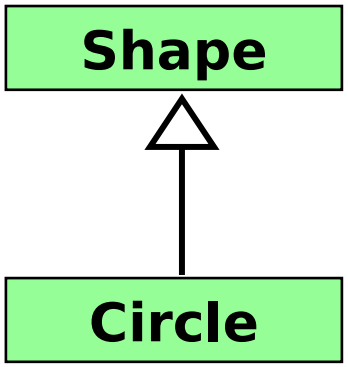
- The term *polymorphism* literally means "having many forms“
- also called ***dynamic binding or late binding or run-time binding***
- allows to create versatile software designs
- In OOP, polymorphism promotes code reuse by calling the method in a generic way.
- Suppose we create the following reference variable:

```
Animal myPets;
```

- Java allows this reference to point to an Animal object, or to any object of any compatible type
- This compatibility can be established using inheritance or using interfaces
- Careful use of polymorphic references can lead to elegant, robust software designs

References and Inheritance

- An object reference can refer to an object of its class, or to an object of any class related to it by inheritance
- For example, if the **`Shape`** class is used to derive a class called **`Circle`**, then a **`Shape`** reference could be used to point to a **`Circle`** object




```
Shape s;
s = new Circle();
```

- Assigning a child object to a parent reference is considered to be a **widening conversion**, and can be performed by simple assignment
- Assigning a parent object to a child reference can be done also, but it is considered a **narrowing conversion** and must be done with a **cast**
- The widening conversion is the most useful

Polymorphism via Inheritance

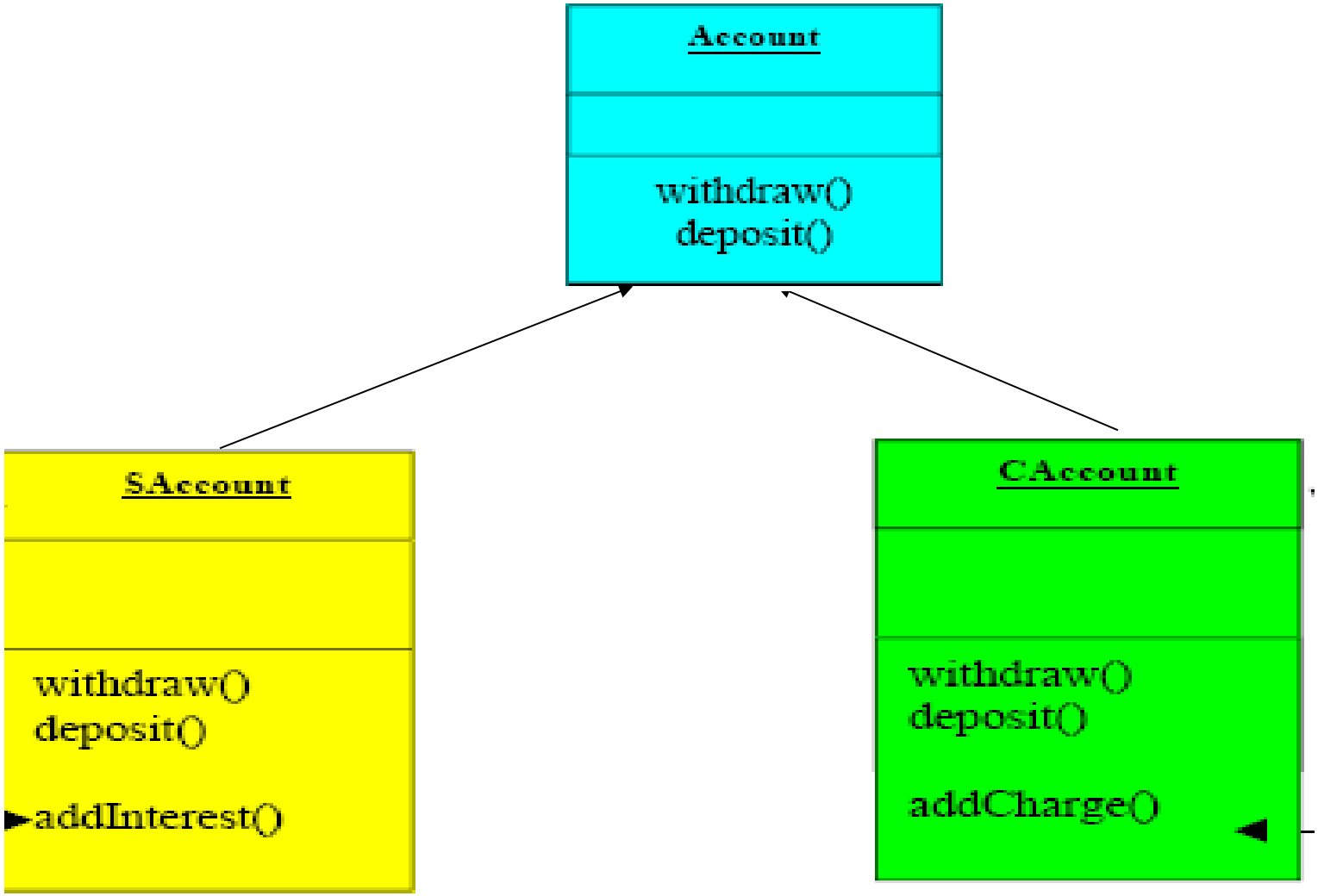
- It is the type of the object being referenced, not the reference type, that determines which method is invoked
- Suppose the **Shape** class has a method called **area**, and the **Circle** class overrides it
- Now consider the following invocation:

```
s.area();
```

- If **s** refers to a **Shape** object, it invokes the **Shape** version of **area**;
  - if it refers to a **Circle** object, it invokes the **Circle** version

Polymorphism

- Consider the following inheritance



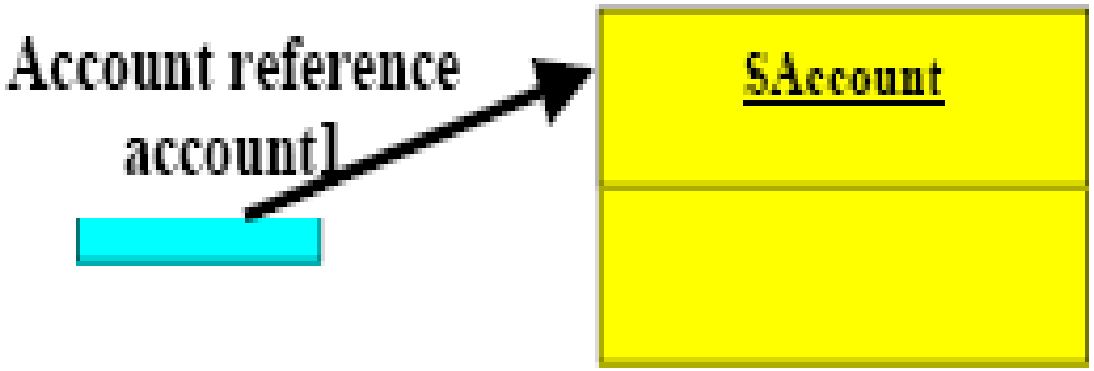
Object Reference Conversion

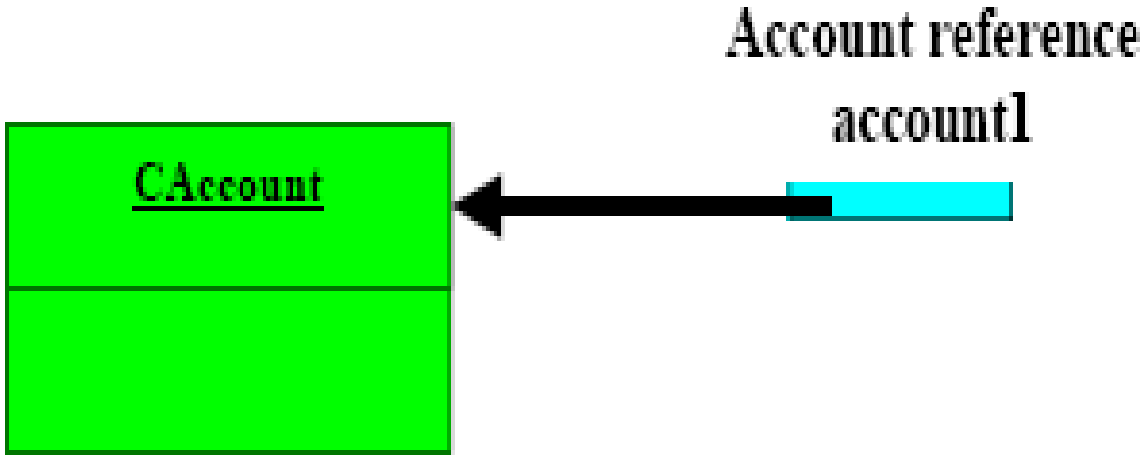
- A ***polymorphic reference*** is a variable that can refer to different types of objects at different points in time
- The method invoked through a polymorphic reference can change from one invocation to the next
- All object references in Java are potentially polymorphic

```
Account account1;
...
account1 = new SAccount("s123","Tom", 100,0);
...
account1 = new Account("c123","Kim",2000,1000);
```

- To enable methods to be called in a polymorphic way, Java allows a superclass reference to refer to a subclass object.

- As **SAccount** and **CAccount** extend the class **Account** both of these statements are valid.





Invoking methods through Superclass Reference

- When **Account** reference **account1** referring to a subclass object is used to invoke a method such as **withdraw**(), the overridden **withdraw**() of that subclass is invoked.

```
Account account1 = new SAccount(“s123”,”Tom”,100, 0);
account1.withdraw(100);//withdraw of SAccount invoked
account1 = new CAccount (“c343”,”Kim”,2000,1000);
account1.withdraw(200);//withdraw of CAccount invoked
```

Use of Polymorphism

```
Account[] accounts = new Account[3];
accounts[0] = new Account("a12345", "Charles", 1000);
accounts[1] = new SAccount("s12", "Craig", 1200, 1000);
accounts[2] = new CAccount("c12", "George", 200, 1000);
```

// Deduct fixed amount $500 from all accounts

```
for (int i = 0; i < 3; i++)
  accounts[i].withdraw(500);
```

- Which **withdraw()** method is called ?
- Though **accounts[i]** is an **Account** reference in Java, actual method called is determined at run-time based on the type of object being referred.

Method overloading and overriding

- In Java, one form of polymorphism refers to the fact that you can have multiple methods with the same name in the same class
- There are two kinds of polymorphism:
  - **Overloading**
    - Two or more methods with different signatures
  - **Overriding**
    - Replacing an inherited method with another having the same signature

Polymorphism via Interfaces

- An interface name can be used as the type of an object reference variable

```
Speaker current;
```

- The current reference can be used to point to any object of any class that implements the **Speaker** interface
- The version of **speak** that the following line invokes depends on the type of object that current is referencing

```
current.speak();
```

Polymorphism via Interfaces

- Suppose two classes, Lecturer and Politician, both implement the Speaker interface, providing distinct versions of the speak method
- In the following code, the first call to speak invokes one version and the second invokes another:

```
Speaker guest = new lecturer();
guest.speak();
guest = new politician();
guest.speak();
```

```
public class Plant {
     public void grow() {
          System.out.println(“Plant
growing”);
     }
}
class Tree extends Plant {
     public void grow() {
                    System.out.println(“Tree
growing”);
     }
     public void leavesShade() {
          System.out.println(“Leaves
Shading”);
     }
}
```

```
public class App {
	public static void main(String [] args)
{
		Plant plant1 = new Plant();
		Tree tree = new Tree();
		Plant plant2 = tree;
		plant2.grow();
		doGrow(tree);
	}
	public static void doGrow (Plant plant)
{
		plant.grow();
	}
}
```