CS212: Object-Oriented Programming

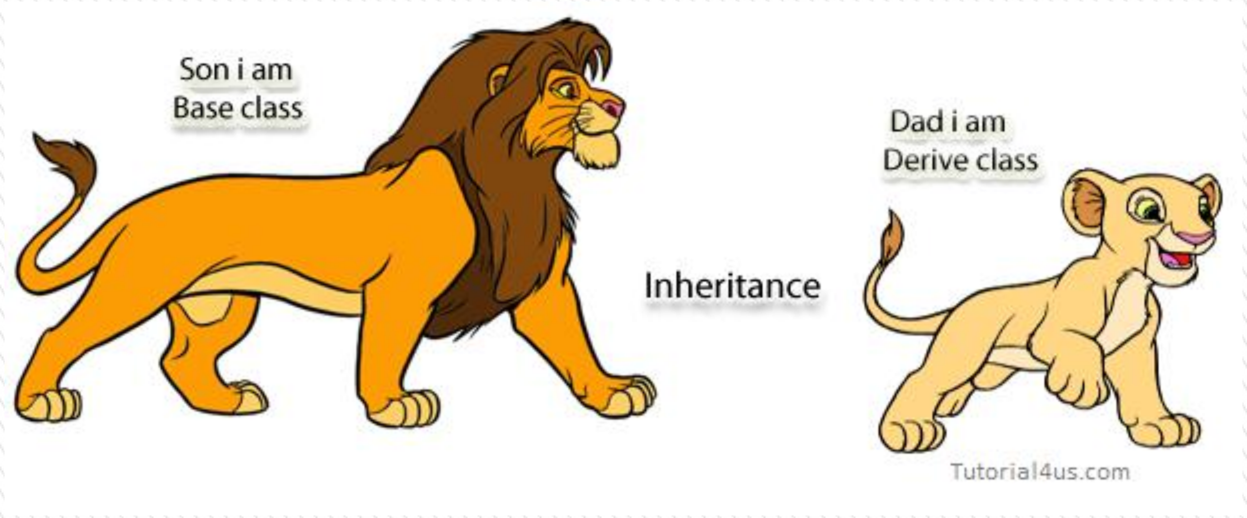
Introducing Inheritance

Instructor: Hirra Anwar

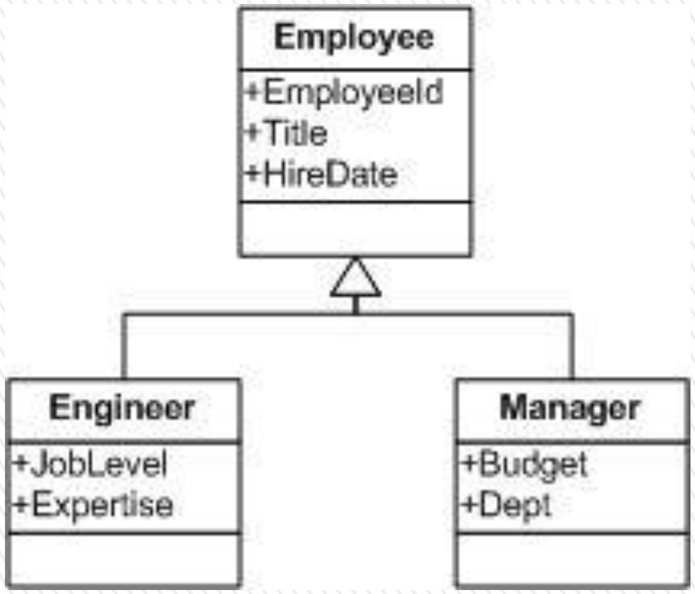
Objectives

- How inheritance promotes software reusability?
- The notions of super classes and subclasses.
- To use keyword `extends` to create a class that inherits attributes and behaviors from another class.
- To use access modifier `protected` to give subclass methods access to superclass members.
- To access superclass members with `super`.
- How constructors are used in inheritance hierarchies.
- The methods of class `Object`, the direct or indirect superclass of all classes in Java.

Introduction



INTRODUCTION



- Inheritance
  - Software reusability
  - Create new classes from existing classes
    - Absorb existing class’s data and behaviors
    - Enhance with new capabilities

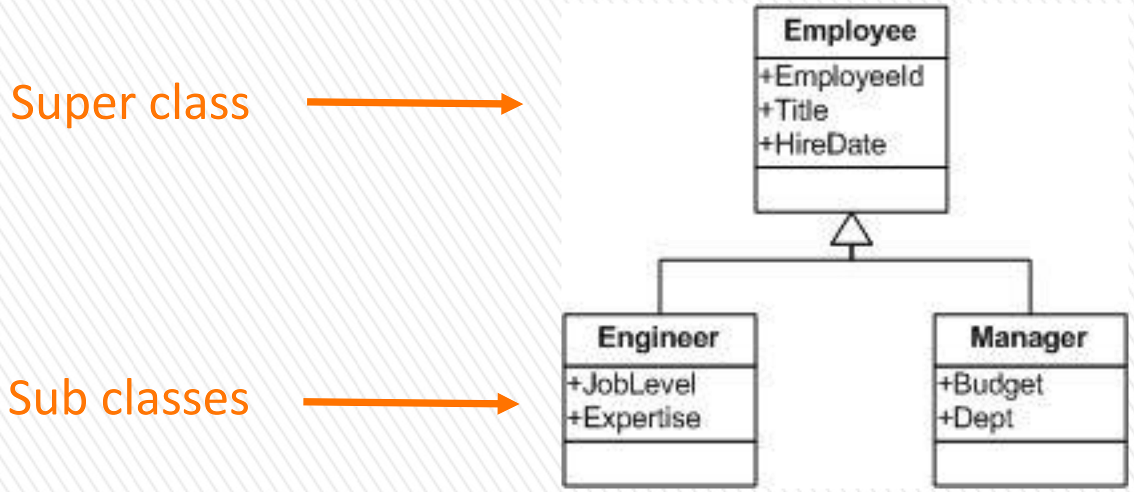
Introduction to Inheritance and Class Hierarchies

- **Main objective of Inheritance is**
  - To reuse previously written code saved as classes
  - Reduce program development time
  - Re-Use debugged and error free classes

- Inheritance in OOP is analogous to inheritance in humans
- Inheritance and hierarchical organization allow you to capture the idea that one thing may be a refinement or extension of another

Inheritance

- Superclass (Base class) & Subclass (Derived class)
  - Subclass extends superclass
  - Superclass
    - Typically contains common features
    - Represents a larger set of objects than subclass
  - Subclass
    - Represents more specialized group of objects
    - Attributes & behaviors inherited from superclass
      - Can customize behaviors
    - Can add additional attributes & behaviors



INHERITANCE

Superclass	Subclasses
`Student`	`GraduateStudent`, `UndergraduateStudent`
`Shape`	`Circle`, `Triangle`, `Rectangle`
`Loan`	`CarLoan`, `HomeImprovementLoan`, `MortgageLoan`
`Employee`	`Faculty`, `Staff`
`BankAccount`	`CheckingAccount`, `SavingsAccount`

Fig. 1 | **Inheritance examples.**

Inheritance Hierarchy

- Inheritance relationships: tree-like hierarchy structure
- Each class becomes
  - Superclass
    - Supply members to other classes
- OR
  - Subclass
    - Inherit members from other classes

Inheritance Hierarchy

- » Direct superclass
  - > Inherited explicitly (one level up hierarchy)
- » Indirect superclass
  - > Inherited two or more levels up hierarchy
- » Single inheritance
  - > Inherits from one superclass
- » Multiple inheritance
  - > Inherits from multiple super classes
    - ▪ Java does not support multiple inheritance
    - ▪ But C++ Does!

Inheritance Hierarchy

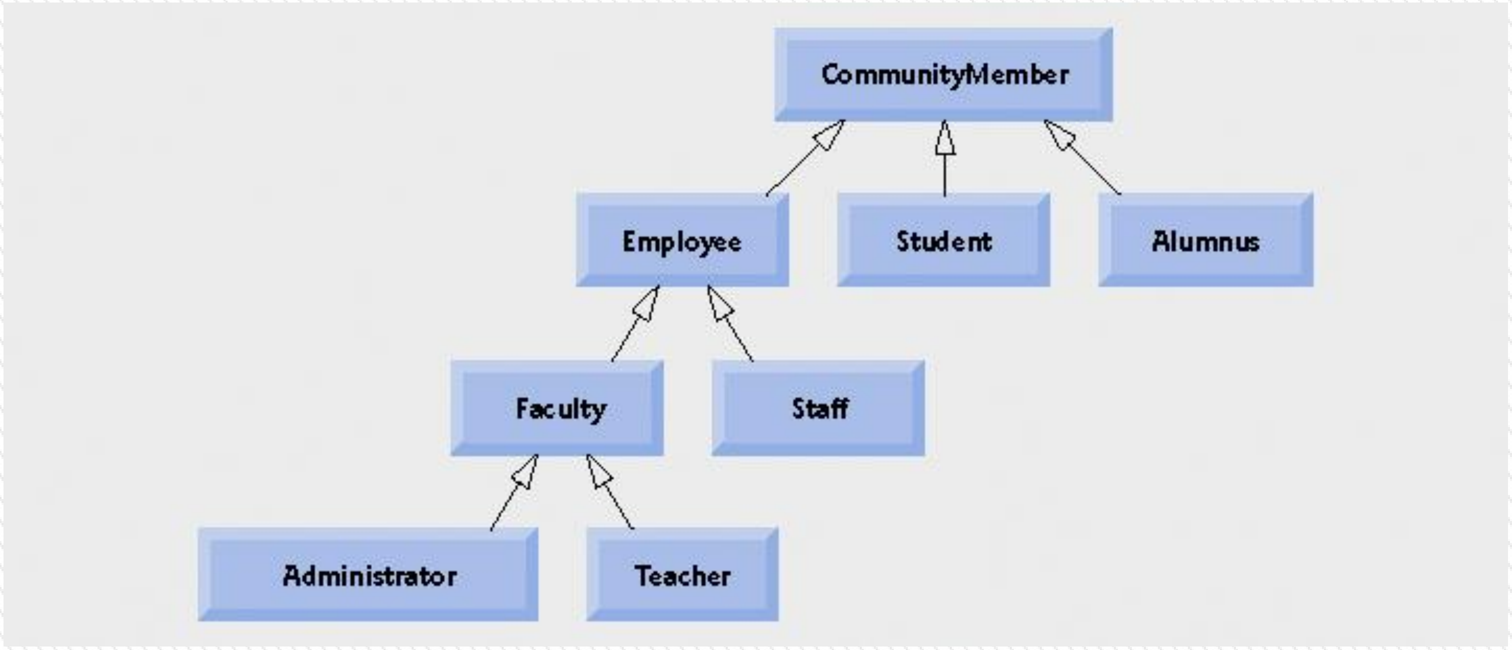


Fig. 2 | Inheritance hierarchy for university `CommunityMembers`

INHERITANCE HIERARCHY

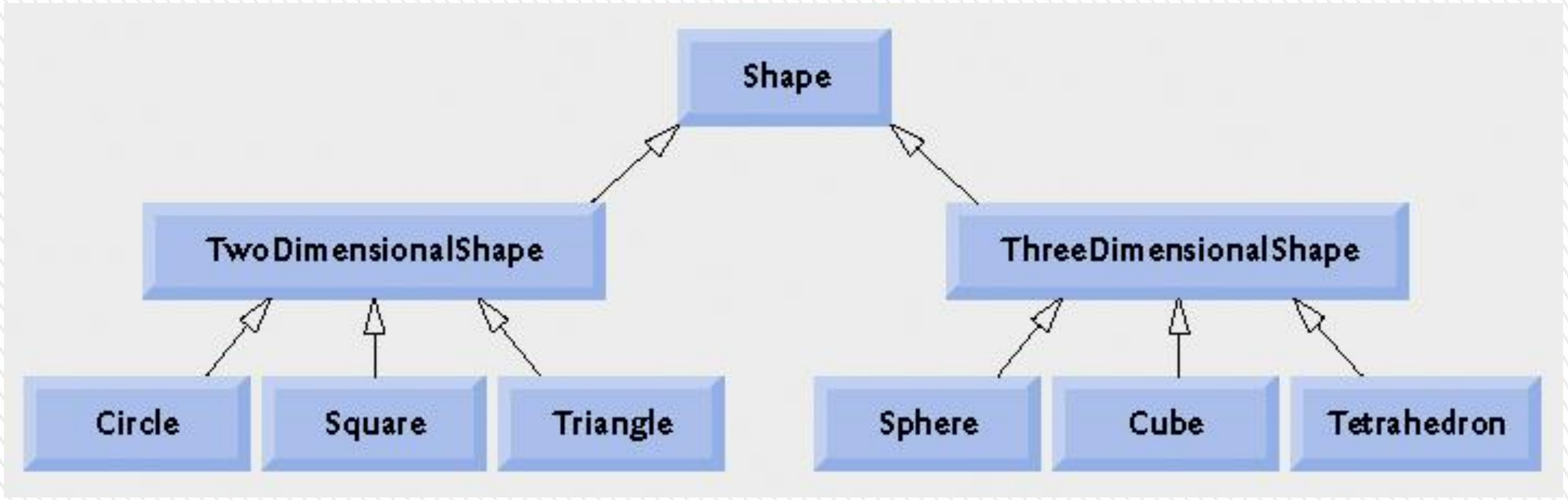


Fig. 3 | Inheritance hierarchy for Shapes.

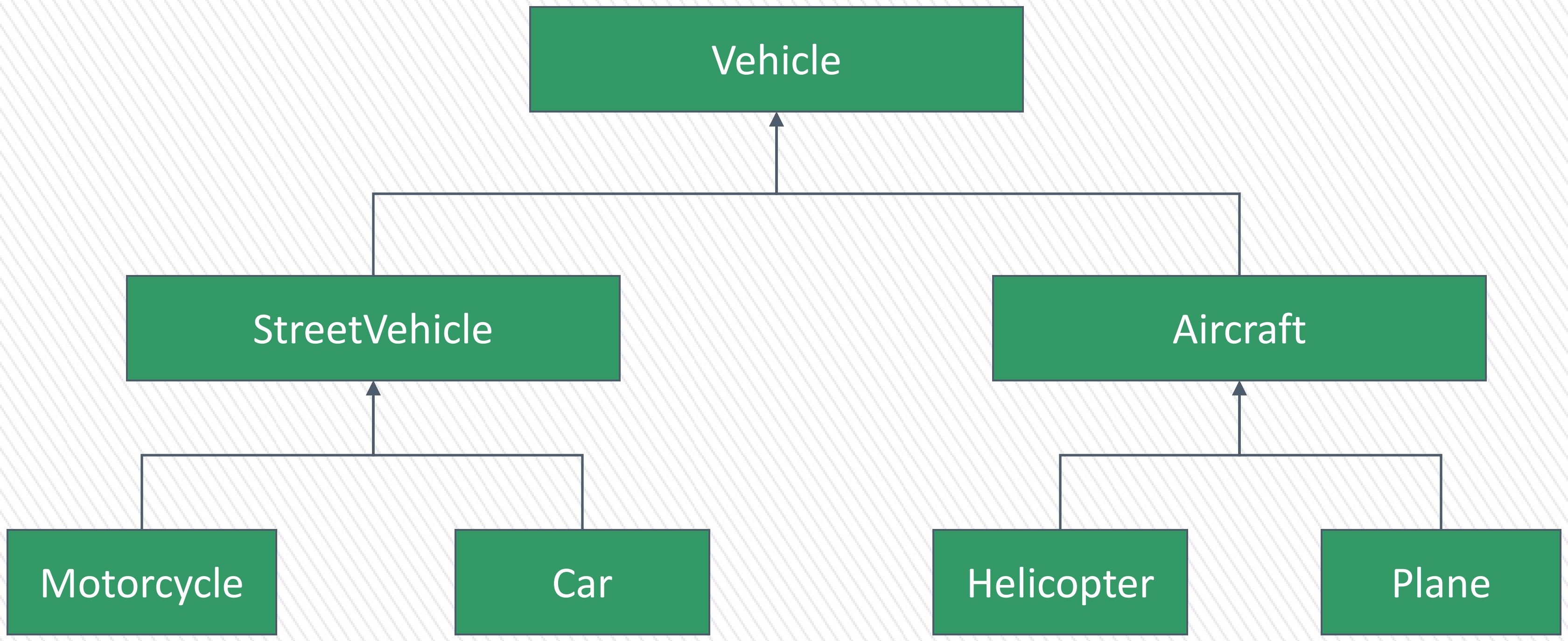
Protected Members

`protected` access

- Intermediate level of protection between `public` and `private`
- `protected` members accessible by
  - superclass members
  - subclass members
  - Class members in the same package
- Subclass access to superclass member
  - Keyword `super` and a dot (.)

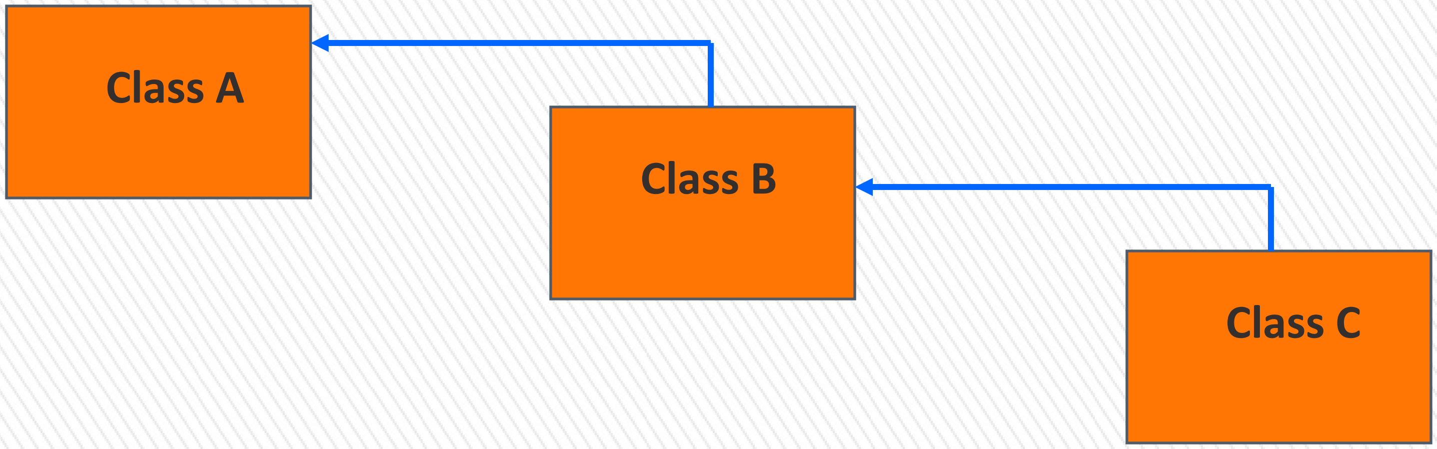
Note

Methods of a subclass cannot directly access `private` members of their superclass. A subclass can change the state of `private` superclass instance variables only through non-`private` methods provided in the superclass and inherited by the subclass.


Vehicle
StreetVehicle
Aircraft
Motorcycle
Car
Helicopter
Plane

Introduction to Inheritance and Class Hierarchies

- Defining a new class based on an existing class is called **derivation**
- The derived class is also called the **direct subclass** of the base or **super class**
- You can also derive classes from the derived class and so on



- **class B extends A{**

```
class B extends A{
                    //definition of class B
                 }
```

- The keyword extends identifies that class B is a direct subclass of class A
- The class B can have additional members in addition to the inherited members of class A

- An inherited member of a base class is one that is accessible within the derived class
- Base class members that are not inherited still form part of a derived class object
- An inherited member of a derived class is a full member of that class and is freely accessible to any method in the class
- Which members of the base class are inherited?

Protected Visibility for Superclass Data Fields

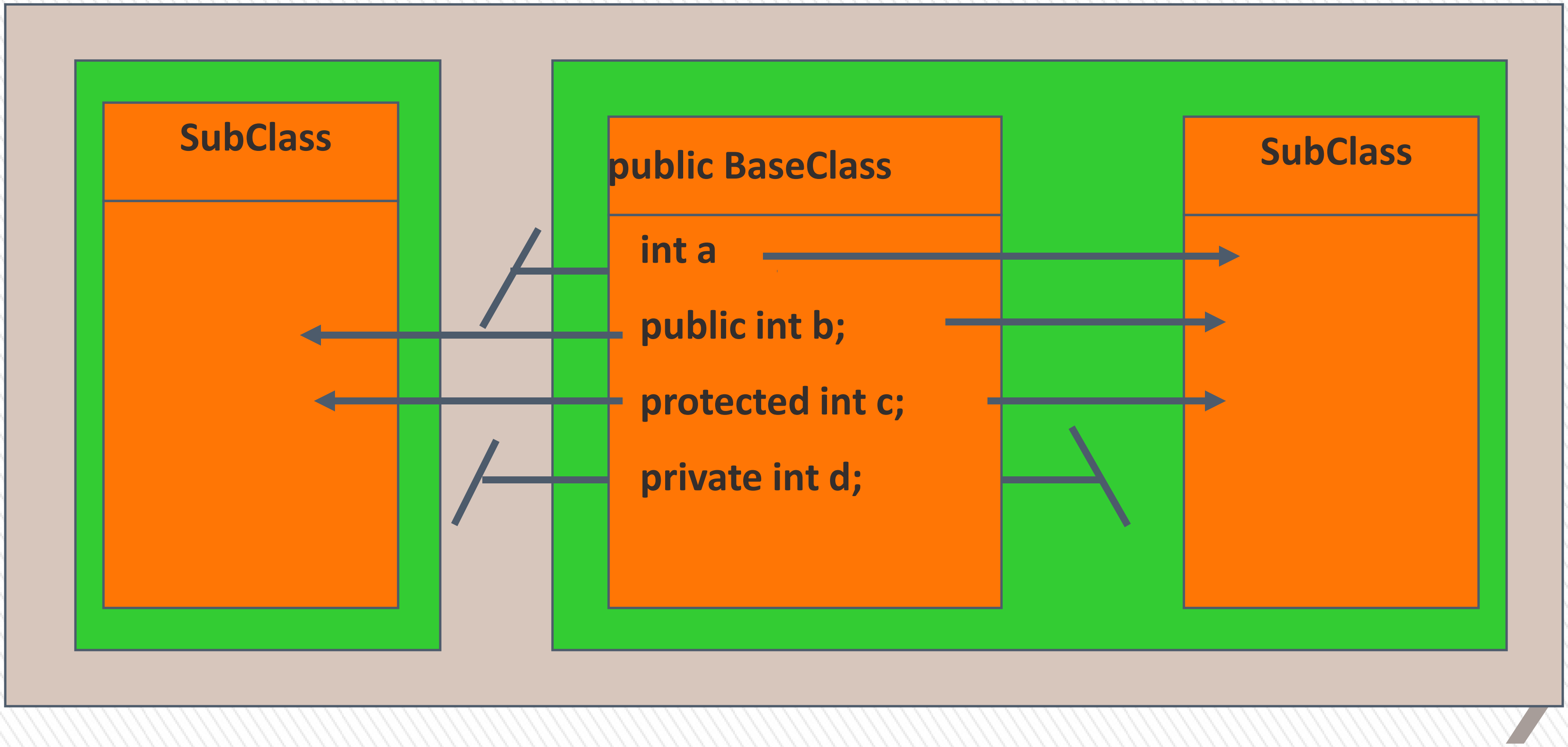
» Private data fields are not accessible to derived classes

» Protected visibility allows data fields to be accessed either by the class defining it or any subclass

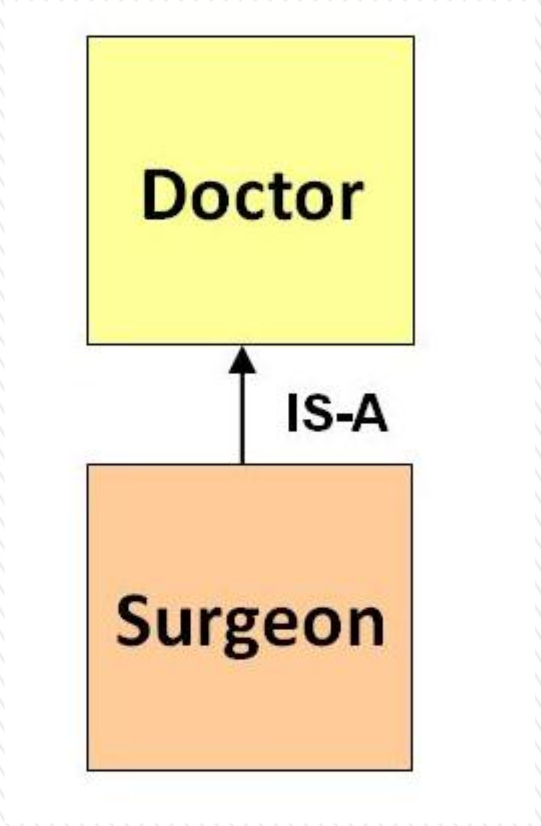
» In general, it is better to use private visibility because subclasses may be written by different programmers and it is always good practice to restrict and control access to the superclass data fields

Inheriting Data Members

- The inheritance rules apply to class variables as well as instance variables



Inheritance - Examples



```
class Doctor {
 void Doctor_Details( ) {
  System.out.println("Doctor Details...");
 }}

class Surgeon extends Doctor {
 void Surgeon_Details( ) {
  System.out.println("Surgen Detail...");
 }}

public class Hospital {
 public static void main(String args[]) {
  Surgeon s = new Surgeon();
  s.Doctor_Details();
  s.Surgeon_Details();
 }
}
```

- **Super Keyword**
- You can define a data member in a derived class with the same name as data member in the base class.
- The data member of the base class is still inherited but is hidden by the derived class member with the same name
- The hiding will occur irrespective if the type or access modifiers are the same or not.
- Any use of the derived member name will always refer to the member defined in derived class
- To refer to the inherited base class member, you must qualify it with the keyword super
- Note that you cannot use super.super.something

- Methods in a base class excluding constructors are inherited in a derived class in the same way as the data members of the base class
- Methods declared as private in a base class are not inherited
- Note: Constructors in the base class are never inherited regardless of their attributes
- Though the base class constructors are not inherited in your derived class, you can still call them or if you don't call a base class constructor from your derived class constructor, the compiler will try to do it for you
- The super class constructor is called in a subclass using super ( );

Method Overriding

- If a derived class has a method found within its base class, that method will override the base class’s method
- The keyword super can be used to gain access to superclass methods overridden by the base class
- A subclass method must have the same return type as the corresponding superclass method

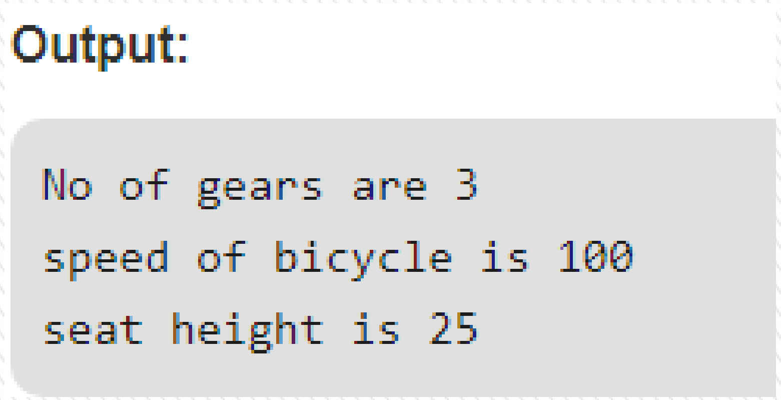
- Example – Bicycle

- class Bicycle is a base class, class MountainBike is a derived class which extends Bicycle class and class Test is a driver class to run program.

```
class Bicycle
{
    // the Bicycle class has two fields
    public int gear;
    public int speed;

    // the Bicycle class has one constructor
    public Bicycle(int gear, int speed)
    {
        this.gear = gear;
        this.speed = speed;
    }

    // the Bicycle class has three methods
    public void applyBrake(int decrement)
    {
        speed -= decrement;
    }

»
»        public void speedUp(int increment)
»        {
»            speed += increment;
»        }
»
»        // toString() method to print info of Bicycle
»        public String toString()
»        {
»            return("No of gears are "+gear
»                    +"\n"
»                    + "speed of bicycle is "+speed);
»        }
»    }
```

```
// derived class
class MountainBike extends Bicycle
{

    // the MountainBike subclass adds one more field
    public int seatHeight;

    // the MountainBike subclass has one constructor
    public MountainBike(int gear,int speed,
                        int startHeight)
    {
        // invoking base-class(Bicycle) constructor
        super(gear, speed);
        seatHeight = startHeight;
    }

» // the MountainBike subclass adds one more method
»     public void setHeight(int newValue)
»     {
»         seatHeight = newValue;
»     }
»
»     // overriding toString() method
»     // of Bicycle to print more info
»     @Override
»     public String toString()
»     {
»         return (super.toString()+
»                 "\nseat height is "+seatHeight);
»     }
» }
```

```
// driver class
public class Test
{
    public static void main(String args[])
    {

        MountainBike mb = new MountainBike(3, 100, 25);
        System.out.println(mb.toString());

    }
}
```



Output:

```
No of gears are 3
speed of bicycle is 100
seat height is 25
```

- If the first statement in a derived class constructor is not a call to a base class constructor, the compiler will insert a call to the default class constructor i.e super ( ), for you
- Default call for base class constructor is with no arguments. This sometimes result in a compiler error. Why?

Constructors

- If the first statement in a derived class constructor is not a call to a base class constructor, the compiler will insert a call to the default class constructor i.e super ( ), for you
- Default call for base class constructor is with no arguments. This sometimes result in a compiler error. Why?

When you define your own constructor in a class, no default constructor is created by the compiler. Thus you have to define the no argument constructor for the class yourself so that in a derived class you don't get the compile error due to call to default constructor of base class

Example

```
public class Person
{
    protected String name;
    protected String address;
    /* Default constructor */
    public Person()
    {
        System.out.println(“Inside Person:Constructor”)
        name = ""; address = "";
    }. . . .
}
public class Student extends Person
{
        public Student()
        {
                System.out.println(“Inside Student:Constructor”);
        }. . . .
}
public static void main( String[] args )
{
        Student ann = new Student();
}
```

The super keyword

- A subclass can also explicitly call a constructor of its immediate super class.
- This is done by using the **super** constructor call.
- A super constructor call in the constructor of a subclass will result in the execution of relevant constructor from the super class, based on the arguments passed.
- Few things to remember when using the super constructor call:
    - The super() call must occur as the first statement in a constructor
    - The super() call can only be used in a constructor (not in ordinary methods)

Super Keyword Example

```
public Student()
{
    super( "SomeName", "SomeAddress" );
    System.out.println("Inside Student:Constructor");
}
```

» Another use of super is to refer to members of the super class (just like the this reference).

```
public Student()
{
    super.name = “somename”;
    super.address = “some address”;
}
```

Method Overriding

- Note that the scheme of overriding applies only to instance methods
- For static methods, it is called hiding methods

```
public class Person {
-----
    public String getName(){
        System.out.println("Parent: getName");
        return name;
    }
}
public class Student extends Person{
-------
    public String getName(){
        System.out.println("Student: getName");
        return name;
    }
}
```

```
Student st=new Student();
st.getName();
```

```java
public class Test {
  public static void
main(String[] args) {
    A a = new A();
    a.p(10);
    a.p(10.0);
  }
}
class B {
  public void p(double i) {
    System.out.println(i * 2);
  }
}
class A extends B {
  // This method overrides the
method in B
  public void p(double i) {
    System.out.println(i);
  }
}
```

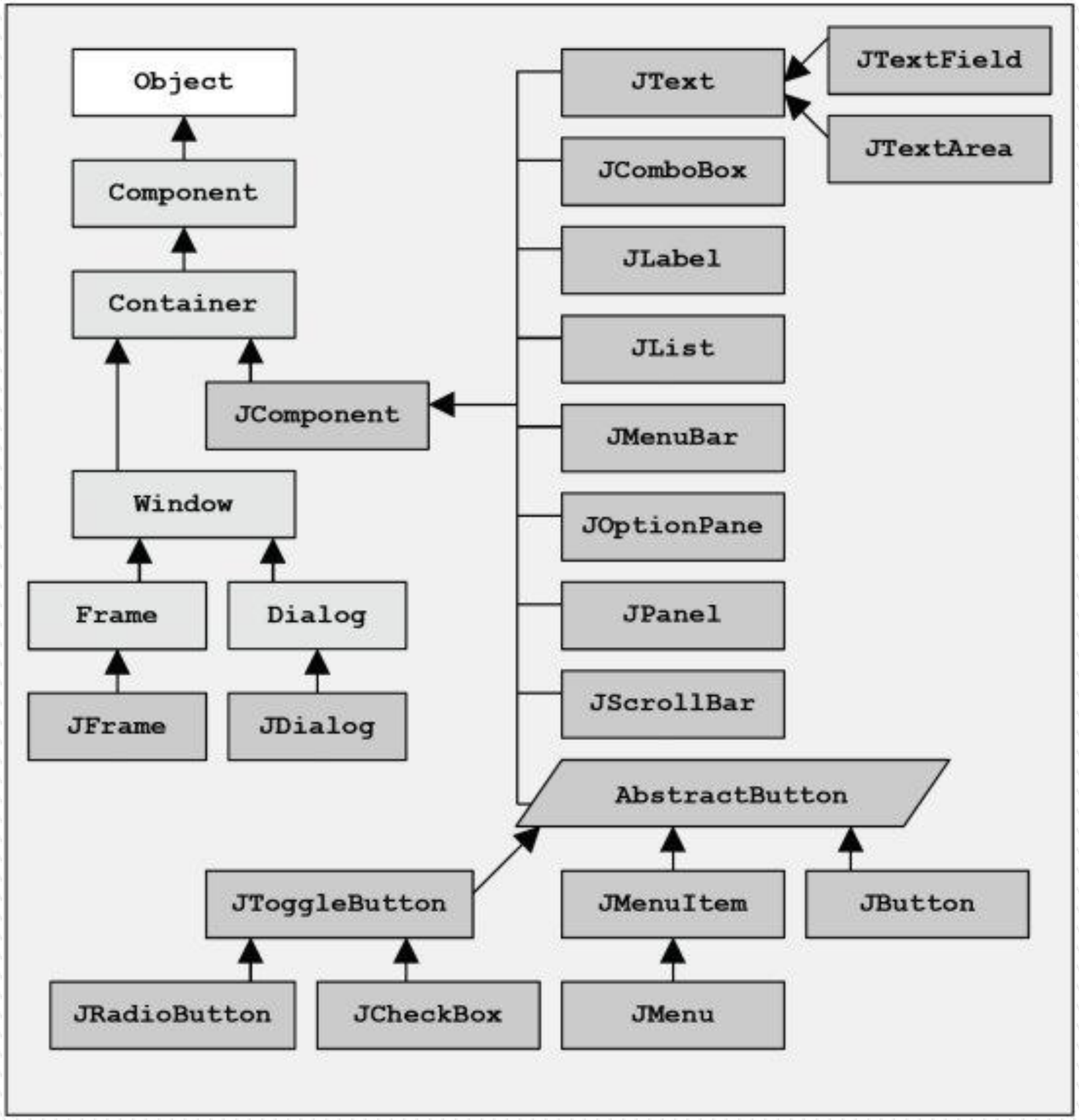
```java
public class Test {
  public static void
main(String[] args) {
    A a = new A();
    a.p(10);
    a.p(10.0);
  }
}
class A {
  public void p(double i) {
    System.out.println(i * 2);
  }
// This method overloads the
method p
  public void p(int i) {
    System.out.println(i);
  }
}
```

```
class Person {
        protected String name;
        private String address;
        private String phoneNumber;
        private String emailAddress;
        protected int age;

    public Person() { // no argument constructor
                name = "";
                address = "";
                phoneNumber = "";
                emailAddress = "";  }

        public Person(String name, String address, String phoneNumber,
String emailAddress) { // constructor with all data members as arguments

                this.name = name;
                this.address = address;
                this.phoneNumber = phoneNumber;
                this.emailAddress = emailAddress;
        }
```

```
protected void display() { // display all the data members

                    System.out.println("Name: " + name);
                    System.out.println("Address: " + getAddress());
                    System.out.println("Phone Number: " + getPhoneNumber());
                    System.out.println("Email Address: " + getEmailAddress());

          }
```

```
class Student extends Person {

      private final String classStatus = "Freshman";
      private int regNo;

    public Student() {
          super();
          super.name = "ali";
        }
 public Student(String name, String address, String phoneNumber, String email) {
                        // call super class constructor
                        super(name, address, phoneNumber, emailAddress);              }

@Override
          public void display() {

          super.display(); // display data members inherited from super class
          System.out.println("Class Status: " + getClassStatus());
          }
```

- Object class is mother of all classes
  - In Java language, all classes are sub-classed (extended) from the Object super class
  - Object class is the only class that does not have a parent class
- Defines and implements behavior common to all classes including the ones that you write
  - getClass()
  - equals()
  - toString()


Object
Component
Container
JComponent
Window
Frame
Dialog
JFrame
JDialog
JText
JTextField
JTextArea
JComboBox
JLabel
JList
JMenuBar
JOptionPane
JPanel
JScrollBar
AbstractButton
JToggleButton
JMenuItem
JButton
JRadioButton
JCheckBox
JMenu

The Object Class

- Class `Object` methods
  - `clone`
  - `equals`
  - `finalize`
  - `getClass`
  - `hashCode`
  - `notify, notifyAll, wait`
  - `toString`

- **OBJECT CLASS**
- **toString()** : toString() provides String representation of an Object and used to convert an object to String. The default toString() method for class Object returns a string consisting of the name of the class of which the object is an instance, the at-sign character `@’, and the unsigned hexadecimal representation of the hash code of the object.

```
»  Student s = new Student();

»  // Below two statements are equivalent
»  System.out.println(s);
»  System.out.println(s.toString());

»  //Implementation of toString() method in Object class
»  public String toString() {
»  return getClass().getName() + "@" + Integer.toHexString(hashCode())
»  }
```

- **getClass()** : Returns the class object of “this” object and used to get actual runtime class of the object. It can also be used to get metadata of this class.
- As it is final so we don’t override it.

» **equals(Object obj)** : Compares the given object to “this” object (the object on which the method is called).

» It gives a generic way to compare objects for equality. It is recommended to override **equals(Object obj)** method to get our own equality condition on Objects.

» **clone()** : It returns a new object that is exactly the same as this object.

Method	Description
`clone`	This `protected` method, which takes no arguments and returns an `Object` reference, makes a copy of the object on which it is called. When cloning is required for objects of a class, the class should override method `clone` as a `public` method and should implement interface `Cloneable` (package `java.lang`). The default implementation of this method performs a so-called shallow copy—instance variable values in one object are copied into another object of the same type. For reference types, only the references are copied. A typical overridden `clone` method's implementation would perform a deep copy that creates a new object for each reference type instance variable. There are many subtleties to overriding method `clone`. You can learn more about cloning in the following article: `java.sun.com/developer/JDCTechTips/2001/tt0306.html`

Fig. 4 | `Object` methods that are inherited directly or indirectly by all classes. (Part 1 of 4)

Method	Description
`Equals`	This method compares two objects for equality and returns `true` if they are equal and `false` otherwise. The method takes any `Object` as an argument. When objects of a particular class must be compared for equality, the class should override method `equals` to compare the contents of the two objects. The method's implementation should meet the following requirements: • It should return `false` if the argument is `null`. • It should return `true` if an object is compared to itself, as in `object1.equals( object1 )`. • It should return `true` only if both `object1.equals( object2 )` and `object2.equals( object1 )` would return `true`. • For three objects, if `object1.equals( object2 )` returns `true` and `object2.equals( object3 )` returns `true`, then `object1.equals( object3 )` should also return `true`. • If `equals` is called multiple times with the two objects and the objects do not change, the method should consistently return `true` if the objects are equal and `false` otherwise. A class that overrides `equals` should also override `hashCode` to ensure that equal objects have identical hashcodes. The default `equals` implementation uses operator == to determine whether two references *refer to the same object* in memory. Section 29.3.3 demonstrates class `String`'s `equals` method and differentiates between comparing `String` objects with == and with `equals`.

Fig. 5 | `Object` methods that are inherited directly or indirectly by all classes. (Part 2 of 4)

Method	Description
`finalize`	This `protected` method (introduced in Section 8.10 and Section 8.11) is called by the garbage collector to perform termination housekeeping on an object just before the garbage collector reclaims the object's memory. It is not guaranteed that the garbage collector will reclaim an object, so it cannot be guaranteed that the object's `finalize` method will execute. The method must specify an empty parameter list and must return `void`. The default implementation of this method serves as a placeholder that does nothing.
`getClass`	Every object in Java knows its own type at execution time. Method `getClass` (used in Section 10.5 and Section 21.3) returns an object of class `Class` (package `java.lang`) that contains information about the object's type, such as its class name (returned by `Class` method `getName`). You can learn more about class `Class` in the online API documentation at `java.sun.com/j2se/5.0/docs/api/java/lang/Class.html`.

Fig. 6 | `Object` methods that are inherited directly or indirectly by all classes. (Part 3 of 4)

Method	Description
`hashCode`	A hashtable is a data structure (discussed in Section 19.10) that relates one object, called the key, to another object, called the value. When initially inserting a value into a hashtable, the key's `hashCode` method is called. The hashcode value returned is used by the hashtable to determine the location at which to insert the corresponding value. The key's hashcode is also used by the hashtable to locate the key's corresponding value.
`notify`, `notifyAll`, `wait`	Methods `notify`, `notifyAll` and the three overloaded versions of `wait` are related to multithreading, which is discussed in Chapter 23. In J2SE 5.0, the multithreading model has changed substantially, but these features continue to be supported.
`toString`	This method (introduced in Section 9.4.1) returns a `String` representation of an object. The default implementation of this method returns the package name and class name of the object's class followed by a hexadecimal representation of the value returned by the object's `hashCode` method.

Fig. 7 | `Object` methods that are inherited directly or indirectly by all classes. (Part 4 of 4)

- **Copy Constructor**

- A copy constructor is a constructor that creates a new object using an existing object of the same class and initializes each instance variable of newly created object with corresponding instance variables of the existing object passed as argument.

- This constructor takes a single argument whose type is that of the class containing the constructor.

```
»   class Rectangle
    {
            int length;
            int breadth;
            //constructor to initialize length and breadth of rectangle
            Rectangle(int l, int b)
            {
              length = l;           breadth= b;
            }
            //copy constructor
            Rectangle(Rectangle obj)
            {
             System.out.println("Copy Constructor Invoked");
             length = obj.length;
             breadth= obj.breadth;
            }
           //method to calcuate area of rectangle
           int area()
           {
             return (length * breadth);
           } }
```

» //class to create Rectangle object and calculate area

```
    class CopyConstructor
{
        public static void main(String[] args)
        {
          Rectangle firstRect = new Rectangle(5,6);
          Rectangle secondRect= new Rectangle(firstRect);
          System.out.println("Area  of First Rectangle : "+
                                          firstRect.area());
          System .out.println("Area of First Second Rectangle:”
                                     + secondRect.area());
        }
}
```

- Activity:

- Create a class that implements a complex no. e.g. 2+i3
- Create data members
- Constructor
- Copy constructor
- Display Method

```
class Complex {
   private double re, im;
     // A parametrized constructor
  public Complex(double re, double im) {
    this.re = re;
    this.im = im;
  }
     // copy constructor
  Complex(Complex c) {
    System.out.println("Copy constructor called");
    re = c.re;
    im = c.im;
  }

  // Overriding the toString of Object class
  @Override
  public String toString() {
    return "(" + re + " + " + im + "i)";
  } }
```

```java
public class Main {

    public static void main(String[] args) {

        Complex c1 = new Complex(10, 15);

        // Following involves a copy constructor call
        Complex c2 = new Complex(c1);

        // Note that following doesn't involve a copy constructor call as
        // non-primitive variables are just references.
        Complex c3 = c2;

        System.out.println(c2); // toString() of c2 is called here
    }
}
```

- **Activity:**
- **CommissionEmployee** class represents an employee paid a percentage of weekly sales.
- **BasePlusCommissionEmployee** class represents an employee who receives a base salary in addition to commission.

- Activity:

- public class CommissionEmployee extends Object
- private String firstName;
- private String lastName;
- private String SSN;
- private double weeklySales; // gross weekly sales
- private double commissionRate;

- public class **BasePlusCommissionEmployee**
- private String firstName;
- private String lastName;
- private String socialSecurityNumber;
- private double grossSales; // gross weekly sales
- private double commissionRate;
- private double baseSalary;

Superclass & Subclass Relationship

- Example: `CommissionEmployee/BasePlusCommissionEmployee` inheritance hierarchy
- `CommissionEmployee`
  - First name, last name, SSN, commission rate, gross sale amount
- `BasePlusCommissionEmployee`
  - First name, last name, SSN, commission rate, gross sale amount
  - Base salary

```java
// Fig. 9.4: CommissionEmployee.java
// CommissionEmployee class represents a commission empl

public class CommissionEmployee extends Object
{
   private String firstName;
   private String lastName;
   private String socialSecurityNumber;
   private double grossSales; // gross weekly sales
   private double commissionRate; // commission percent

   // five-argument constructor
   public CommissionEmployee( String first, S
      double sales, double rate )
   {
      // implicit call to Object constructor occurs h
      firstName = first;
      lastName = last;
      socialSecurityNumber = ssn;
      setGrossSales( sales ); // validate and store gross sales
      setCommissionRate( rate ); // validate and store commission rate
   } // end five-argument CommissionEmployee constructor

   // set first name
   public void setFirstName( String first )
   {
      firstName = first;
   } // end method setFirstName

```

Declare `private` instance variables

Class `CommissionEmployee` `extends` class `Object`

Implicit call to `Object` constructor

Initialize instance variables

Invoke methods `setGrossSales` and `setCommissionRate` to validate data

```
   // return first name
   public String getFirstName()
   {
      return firstName;
   } // end method getFirstName

   // set last name
   public void setLastName( String last )
   {
      lastName = last;
   } // end method setLastName

   // return last name
   public String getLastName()
   {
      return lastName;
   } // end method getLastName

   // set social security number
   public void setSocialSecurityNumber( String ssn )
   {
      socialSecurityNumber = ssn; // should validate
   } // end method setSocialSecurityNumber

   // return social security number
   public String getSocialSecurityNumber()
   {
      return socialSecurityNumber;
   } // end method getSocialSecurityNumber

   // set gross sales amount
   public void setGrossSales( double sales )
   {
      grossSales = ( sales < 0.0 ) ? 0.0 : sales;
   } // end method setGrossSales

   // return gross sales amount
   public double getGrossSales()
   {
      return grossSales;
   } // end method getGrossSales

   // set commission rate
   public void setCommissionRate( double rate )
   {
      commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
   } // end method setCommissionRate

   // return commission rate
   public double getCommissionRate()
   {
      return commissionRate;
   } // end method getCommissionRate

   // calculate earnings
   public double earnings()
   {
      return commissionRate * grossSales;
   } // end method earnings

```

Calculate earnings

```java
   // return String representation of CommissionEmployee object
   public String toString()
   {
      return String.format( "%s: %s %s\n%s: %s\n%s: %.2f\
         "commission employee", firstName, lastName,
         "social security number", socialSecurityNumber,
         "gross sales", grossSales,
         "commission rate", commissionRate );
   } // end method toString
} // end class CommissionEmployee
```

Override method `toString` of class `Object`

```java
// Fig. 9.5: CommissionEmployeeTest.java
// Testing class CommissionEmployee.

public class CommissionEmployeeTest
{
   public static void main( String args[] )
   {
      // instantiate CommissionEmployee object
      CommissionEmployee employee = new CommissionEmployee(
         "Sue", "Jones", "222-22-2222", 10000, .06 );

      // get commission employee data
      System.out.println(
         "Employee information obtained by get methods: \n" );
      System.out.printf( "%s %s\n", "
         employee.getFirstName() );
      System.out.printf( "%s %s\n", "
         employee.getLastName() );
      System.out.printf( "%s %s\n", "Social security number is",
         employee.getSocialSecurityNumber() );
      System.out.printf( "%s %.2f\n", "Gross sales is",
         employee.getGrossSales() );
      System.out.printf( "%s %.2f\n", "Commiss
         employee.getCommissionRate() );

      employee.setGrossSales( 500 ); // set gross sales
      employee.setCommissionRate( .1 ); // set commission rate

```

Instantiate `CommissionEmployee` object

Use `CommissionEmployee`'s *get* methods to retrieve the object's instance variable values

Use `CommissionEmployee`'s *set* methods to change the object's instance variable values

```java
      System.out.printf( "\n%s:\n\n%s\n",
         "Updated employee information obtained by toString", employee );
   } // end main
} // end class CommissionEmployeeTest
```

Implicitly call object's `toString` method

```
Employee information obtained by get methods:

First name is Sue
Last name is Jones
Social security number is 222-22-2222
Gross sales is 10000.00
Commission rate is 0.06

Updated employee information obtained by toString:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 500.00
commission rate: 0.10
```

Creating a CommissionEmployee-BasePlusCommiionEmployee Inheritance Hierarchy

- Class `BasePlusCommissionEmployee2`
  - Extends class `CommissionEmployee`
  - Is a `CommissionEmployee`
  - Has instance variable `baseSalary`
  - Inherits `public` and `protected` members
  - Constructor not inherited

```java
// Fig. 9.8: BasePlusCommissionEmployee2.java
// BasePlusCommissionEmployee2 inherits from class CommissionEmployee.

public class BasePlusCommissionEmployee2 extends CommissionEmployee
{
   private double baseSalary; // base salary per week

   // six-argument constructor
   public BasePlusCommissionEmployee2( Str
      String ssn, double sales, double rate, double salary )
   {
      // explicit call to superclass CommissionEmployee constructor
      super( first, last, ssn, sales, rate );

      setBaseSalary( amount ); // validate and store base salary
   } // end six-argument BasePlusCommissionEmploye

   // set base salary
   public void setBaseSalary( double salary )
   {
      baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
   } // end method setBaseSalary

```

Class `BasePluCommissionEmployee2` is a subclass of `CommissionEmployee`

Invoke the superclass constructor using the superclass constructor call syntax

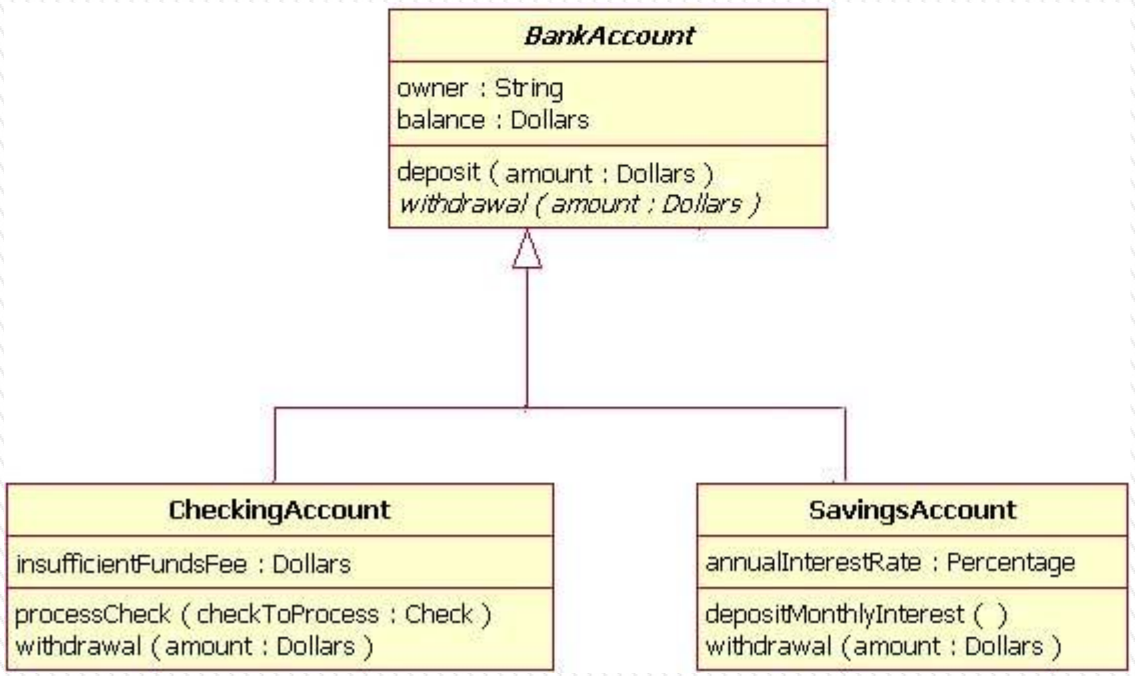
```
   // return base salary
   public double getBaseSalary()
   {
      return baseSalary;
   } // end method getBaseSalary

   // calculate earnings
   public double earnings()
   {
      // not allowed: commissionRate and grossSales private in superclass
      return baseSalary + ( commissionRate * grossSales );
   } // end method earnings

   // return String representation of BasePlusCommissionEmployee2
   public String toString()
   {
      // not allowed: attempts to access private superclass members
      return String.format(
         "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f\n%s: %.2f",
         "base-salaried commission employee", firstName, lastName,
         "social security number", socialSecurityNumber,
         "gross sales", grossSales, "commission rate", commissionRate,
         "base salary", baseSalary );
   } // end method toString
} // end class BasePlusCommissionEmployee2
```

Compiler generates errors because superclass's instance variable `commissionRate` and `grossSales` are `private`

Compiler generates errors because superclass's instance variable `firstName`, `lastName`, `socialSecurityNumber`, `grossSales` and `commissionRate` are `private`

```
BasePlusCommissionEmployee2.java:34: commissionRate has private access in
CommissionEmployee
      return baseSalary + ( commissionRate * grossSales );
                            ^
BasePlusCommissionEmployee2.java:34: grossSales has private access in
CommissionEmployee
      return baseSalary + ( commissionRate * grossSales );
                                             ^
BasePlusCommissionEmployee2.java:43: firstName has private access in
CommissionEmployee
         "base-salaried commission employee", firstName, lastName,
                                              ^
BasePlusCommissionEmployee2.java:43: lastName has private access in
CommissionEmployee
         "base-salaried commission employee", firstName, lastName,
                                                         ^
BasePlusCommissionEmployee2.java:44: socialSecurityNumber has private access in
CommissionEmployee
         "social security number", socialSecurityNumber,
                                   ^
BasePlusCommissionEmployee2.java:45: grossSales has private access in
CommissionEmployee
         "gross sales", grossSales, "commission rate", commissionRate,
                        ^
BasePlusCommissionEmployee2.java:45: commissionRate has private access in
CommissionEmployee
         "gross sales", grossSales, "commission rate", commissionRate,
                                                       ^
7 errors
```

Activity:


BankAccount
owner : String
balance : Dollars
deposit (amount : Dollars)
withdrawal (amount : Dollars)
CheckingAccount
insufficientFundsFee : Dollars
processCheck (checkToProcess : Check)
withdrawal (amount : Dollars)
SavingsAccount
annualInterestRate : Percentage
depositMonthlyInterest ( )
withdrawal (amount : Dollars)

» **Book Reading: (Very important)**
» Chapter 09 – Inheritance
» Book – How to program by Dietel & Dietel

» **Practice Questions**
» Complete the activities given in the slides and practice questions from book exercise.

Questions/Answers & Discussion