Programming in Java

Dr. Nyein Aye Maung Maung
Dr. Eng (Ritsumeikan University, Japan)
Lecturer
Computer Engineering and Information Technology Dept.
Yangon Technological University

Course Schedule

Week	Topics
Week 1	Overview of JAVA, Data Types, Variables and Arrays
Week 2	Operators and Control Statements
Week 3	Classes and A closer look at Methods and Classes
Week 4	Inheritance, Polymorphism, Abstraction and Encapsulation
Week 5	Interfaces and Packages
Week 6	Exception Handling and Multi-threaded Programming
Week 7	String Handling
Week 8	Exploring java.lang and More utilities classes
Week 9	Java Collections Framework
Week 10	Java I/O
Week 11	Basic Graphical User Interface
Week 12	Event Handling
Week 13	Database Programming
Week 14	Applet and Networking

Lecture 4

Inheritance, Polymorphism, Abstraction and Encapsulation

Outline of Class (Lecture 4)

- Inheritance Basic
- Polymorphism
- Class Abstraction
- Encapsulation

Lecture Objectives

- To design programs using the object-oriented paradigm
- To define a subclass from a superclass through inheritance
- To invoke the superclass's constructors and methods using the super keyword
- To override instance methods in the subclass
- To distinguish differences between overriding and overloading
- To apply class abstraction to develop software
- To define private data fields with appropriate get and set methods
- To encapsulate data fields to make classes easy to maintain

Topic 1

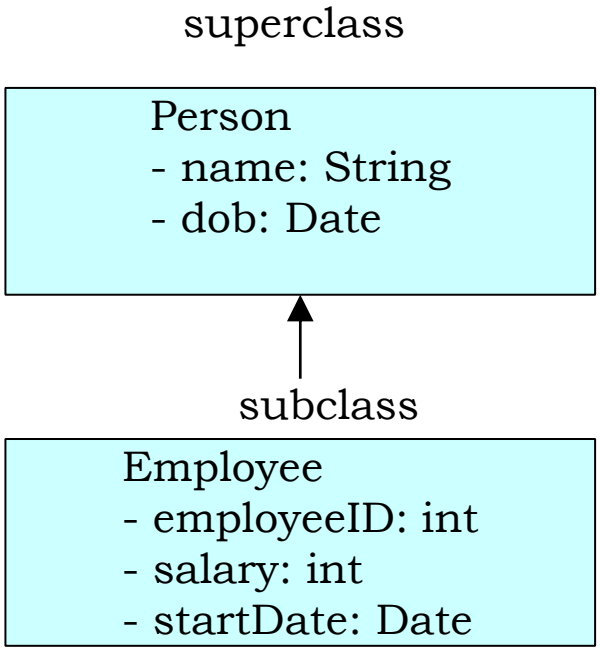
Inheritance

Inheritance

- Object-oriented programming allows you to define new classes from existing classes, called inheritance.
- It helps to reuse the code and establish a relationship between different classes.
- A class can be defined as a "subclass" of another class.
  - The subclass inherits all data attributes of its superclass
  - The subclass inherits all methods of its superclass
  - The subclass inherits all associations of its superclass
- The subclass can:
  - Add new functionality
  - Use inherited functionality
  - Override inherited functionality

Terms used in Inheritance

- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.



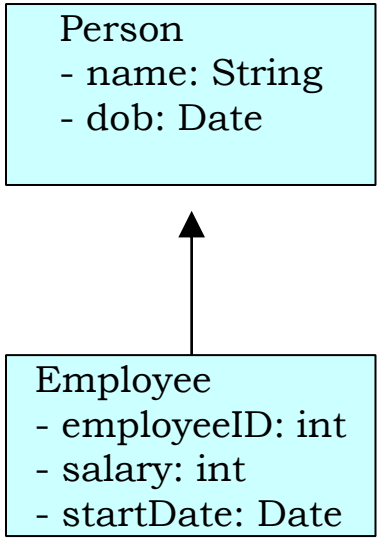
How inheritance is used in Java

- Inheritance is declared using the "**extends**" key word
  - If inheritance is not defined, the class extends a class called Object

```
public class Person
{
    private String name;
    private Date dob;
    [...]
```

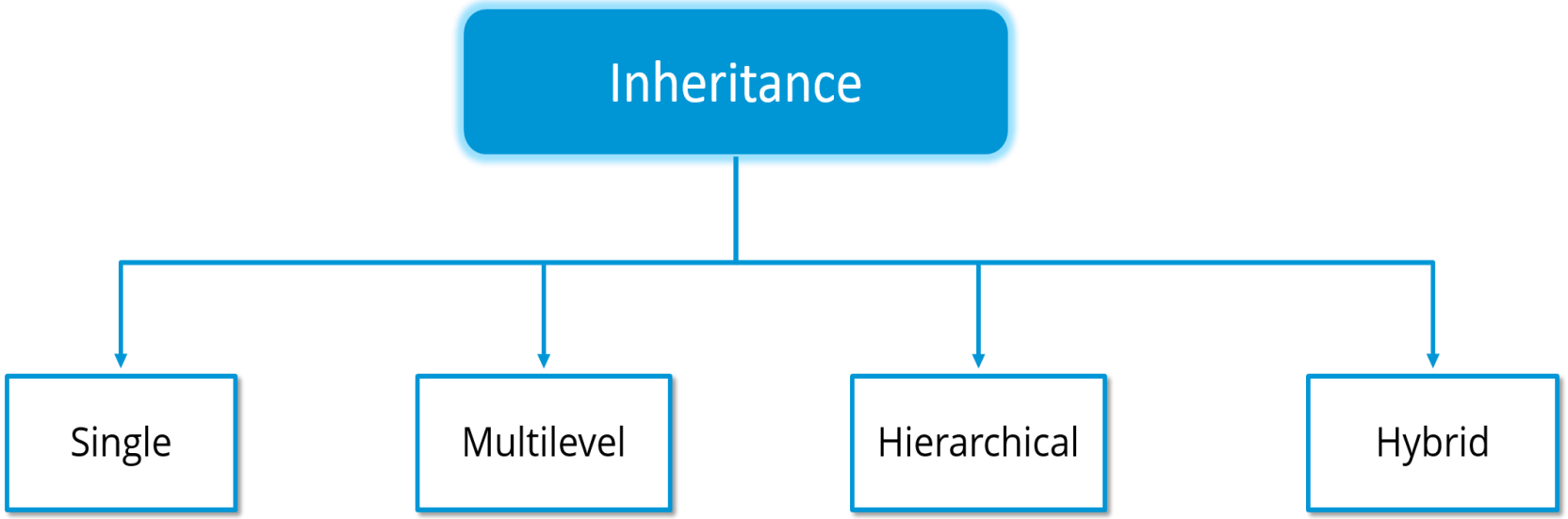
```
public class Employee extends Person
{
    private int employeID;
    private int salary;
    private Date startDate;
    [...]
```

```
Employee anEmployee = new Employee();
```



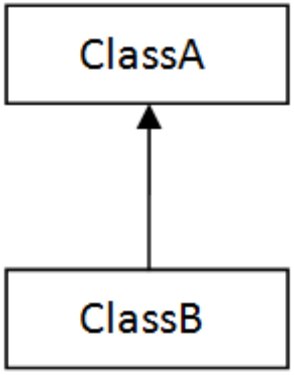
Types of Inheritance

1. Single
2. Multilevel
3. Hierarchical
4. Hybrid



Types of Inheritance

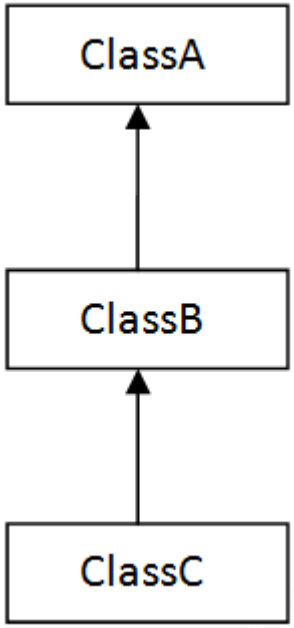
1. Single Level Inheritance




```
class Animal{
      void eat(){System.out.println("eating...");}
      }
      class Dog extends Animal{
      void bark(){System.out.println("barking...");
      }
      }
      class TestInheritance{
            public static void main(String args[]){
            Dog d=new Dog();
            d.bark();
            d.eat();
}
}
```

Types of Inheritance

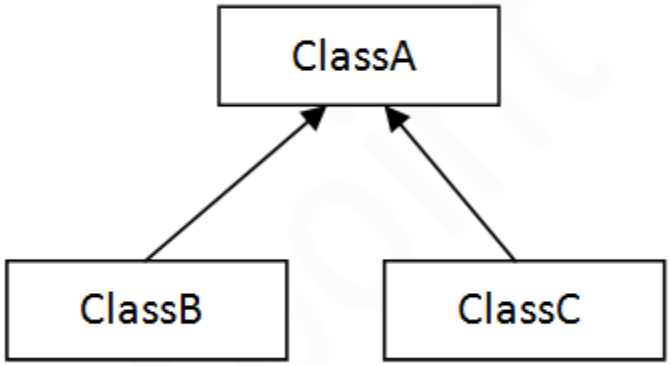
2. Multilevel Inheritance




```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");
}
}
class BabyDog extends Dog{
void weep(){System.out.println("weeping...")
;}
}
class TestInheritance2{
public static void main(String args[]){
BabyDog d=new BabyDog();
d.weep();
d.bark();
d.eat();
}}
```

Types of Inheritance

3. Hierarchical Inheritance

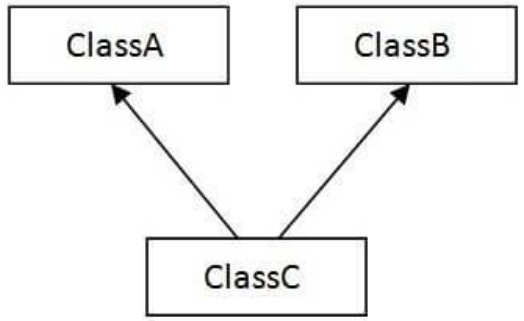



```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");
}
}
class Cat extends Animal{
void meow(){System.out.println("meowing...
");}
}
class TestInheritance3{
public static void main(String args[]){
Cat c=new Cat();
c.meow();
c.eat();
//c.bark();//Error
}}
```

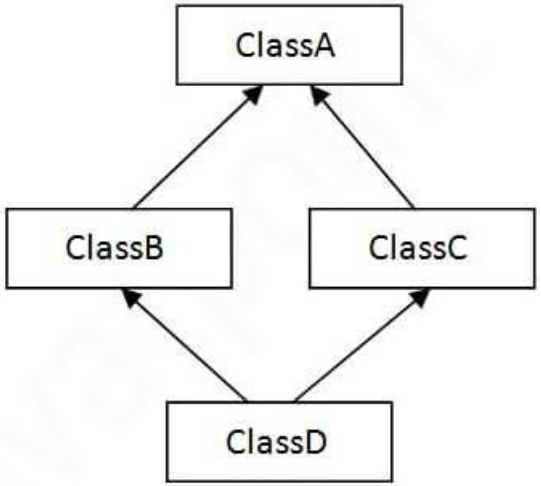
Types of Inheritance

4. Hybrid Inheritance and Multilevel Inheritance

- Not supported in Java, only possible with ‘interfaces’
- E.g. Class D inherits both Class B and Class C
- If B and C classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class
- Java renders compile-time error if you inherit 2 classes



Multilevel



Hybrid

Using Super

- The keyword super refers to the superclass and can be used to invoke the superclass's methods and constructors
- It can be used in two ways:
  - To call a superclass constructor
  - To call a superclass method
- Calling Superclass Constructors
  - Unlike properties and methods, the constructors of a superclass are not inherited by a subclass.
  - They can only be invoked from the constructors of the subclasses using the keyword **super**.

super(), or **super**(parameters);

invokes the no-arg constructor of its superclass

invokes the superclass constructor that matches the **arguments**

Example 4.1: Demonstrate the use of Super

```
class Box {
  private double width;
  private double height;
  private 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;
  }
}
```

```
// BoxWeight now fully implements all constructors.
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;
  }
}
```

width=w;
height=h;
depth=d;

```
class DemoSuper {
  public static void main(String args[]) {
    BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
    BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
    BoxWeight mybox3 = new BoxWeight(); // default
    BoxWeight mycube = new BoxWeight(3, 2);
    BoxWeight myclone = new BoxWeight(mybox1);
    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);
    System.out.println();

    vol = mybox3.volume();
    System.out.println("Volume of mybox3 is " + vol);
    System.out.println("Weight of mybox3 is " + mybox3.weight);
    System.out.println();

    vol = myclone.volume();
    System.out.println("Volume of myclone is " + vol);
    System.out.println("Weight of myclone is " + myclone.weight);
    System.out.println();

    vol = mycube.volume();
    System.out.println("Volume of mycube is " + vol);
    System.out.println("Weight of mycube is " + mycube.weight);
    System.out.println();
  }
}
```

Example 4.2: Using super to access Superclass's instance variables and methods

```
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();
    }
}
```

Output

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

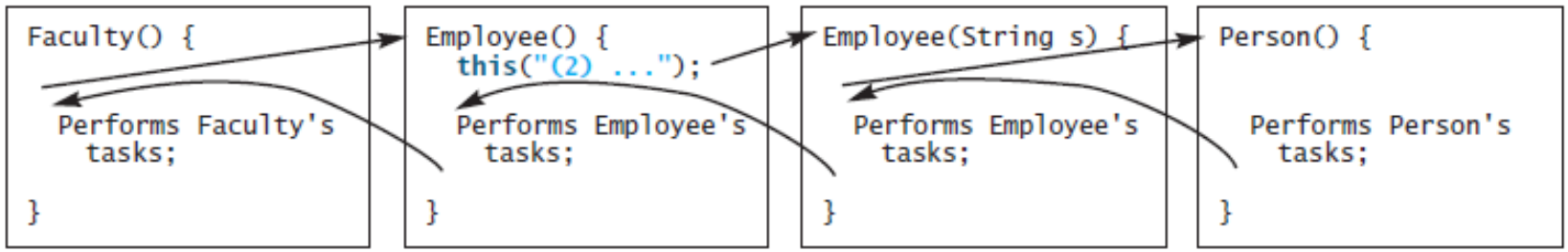
Constructor Chaining

- When constructing an object of a subclass
  - The subclass constructor first invokes its superclass constructor before performing its own tasks.
  - If the superclass is derived from another class, the superclass constructor invokes its parent-class constructor before performing its own tasks.
  - This process continues until the last constructor along the inheritance hierarchy is called. This is called **constructor chaining** .
- If a call to super is not made, the system will automatically attempt to invoke the no-argument constructor of the superclass.
  - If a class is designed to be extended, it is better to provide a no-arg constructor to avoid programming errors.

```
public class Faculty extends Employee{
public static void main(String[] args) {
 new Faculty();
 }
 public Faculty() {
 System.out.println("(4) Performs Faculty's tasks");
 }
 }
```

```
class Employee extends Person {
public Employee(){
this("(2) Invoke Employee's overloaded constructor");
System.out.println("(3) Performs Employee's tasks ");
}
public Employee(String s){
System.out.println(s);
}
}
```

```
class Person {
public Person(){
System.out.println("(1) Performs Person's tasks");
}
}
```



Constructor Chaining

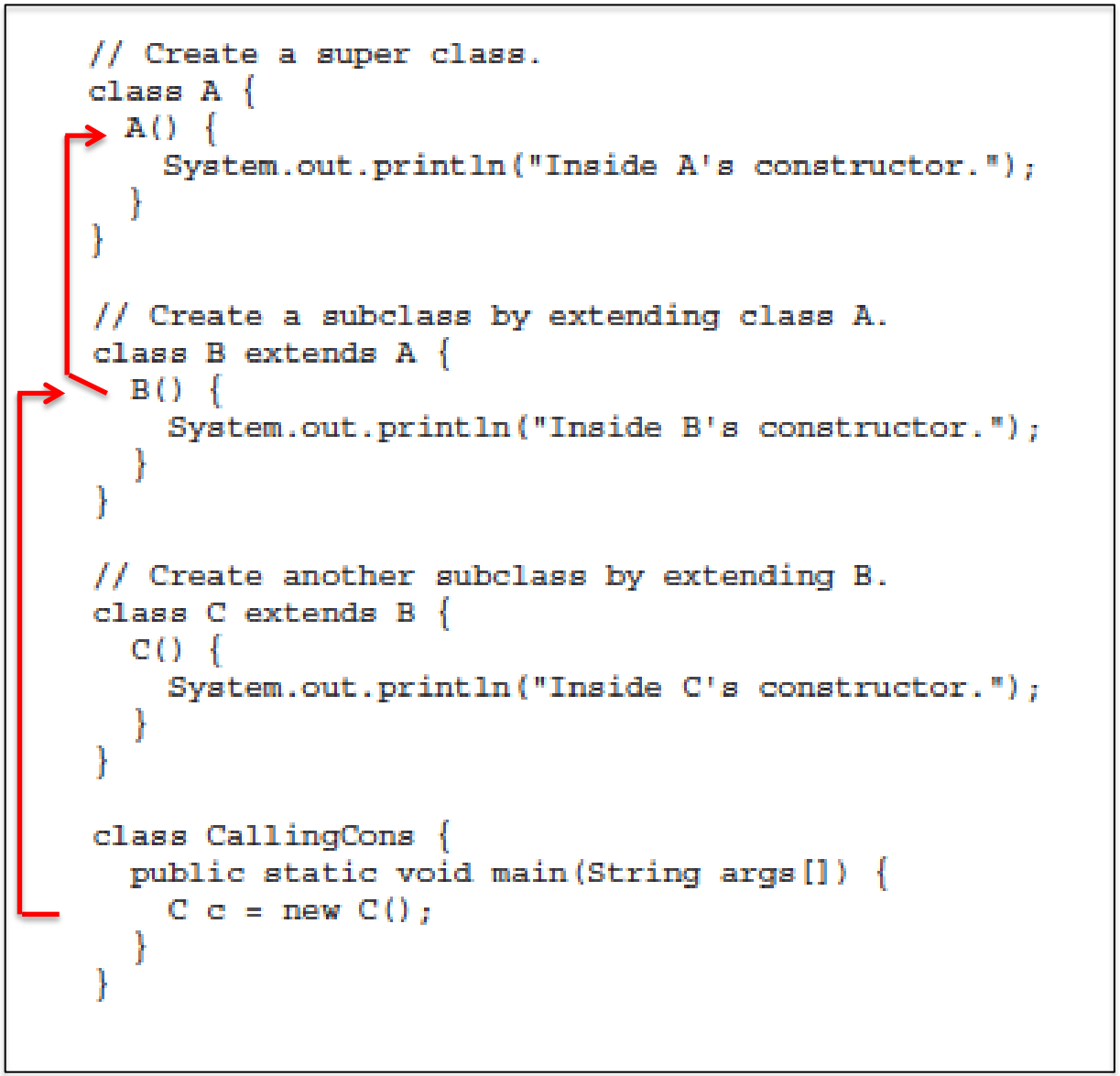
Example 4.3: Demonstrate constructor chaining

```
// 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();
  }
}
```



Output

Inside A's constructor.

Inside B's constructor.

Inside C's constructor.

Topic 2

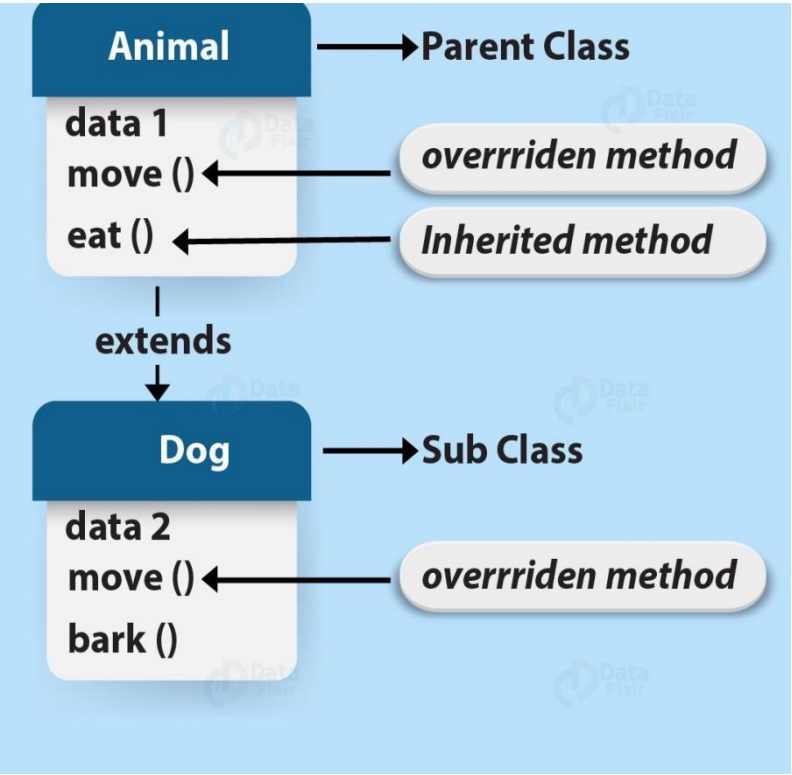
Polymorphism

Polymorphism

- Polymorphism in Java is of two types:
  1. **Run time polymorphism**
     - Refers to a process in which a call to an overridden method is resolved at runtime rather than at compile-time.
     - Method overriding is an example of run time polymorphism.
  2. **Compile time polymorphism**
     - Refers to a process in which a call to an overloaded method is resolved at compile time rather than at run time
     - Method overloading is an example of compile time polymorphism

Method Overriding

- Subclasses inherit all methods from their superclass
  - Sometimes, the implementation of the method in the superclass does not provide the functionality required by the subclass.
  - In these cases, the method must be overridden.
- To override a method, provide an implementation in the subclass.
  - The method in the subclass MUST have the exact same signature as the method it is overriding.



Example 4.4: A program that demonstrates method overriding.

```
class Parent
{
void show()  ← This method overrides show() of Parent
{
        System.out.println("Parent's show()");
}
}
class Child extends Parent      ←Inherited class
{
@Override
void show()
{
        System.out.println("Child's show()");
}
class Main
{
    public static void main(String[] args)
    {
    Parent obj1 = new Parent();
    obj1.show();
    Parent obj2 = new Child();
    obj2.show();
    }
}
```

If a Parent type reference refers to a Parent object, then Parent's show is called

If a Parent type reference refers a Child object Child's show() is called.

Rules for Method Overriding in Java

- Final Methods cannot be Overridden Methods
  - When we don't want the method to be overridden we declare it as final.
- Static Methods cannot be Overridden
- Private Methods cannot be Overridden Method
- Overriding Method must have the same Return Type or Subtype
- Invoking Overridden Method from Sub-Class
  - To call the parent class method during overriding we use the keyword super.

Method Overloading

- Overloading is the Java method which allows the methods to have a similar name but with the difference in signatures which is by input parameters
- Overloading supports compile (static) polymorphism in Java

Example 4.5: A program that demonstrates method overloading.

```
public class Sum
{
public int sum(int x, int y)
{
return (x + y);
}
public int sum(int x, int y, int z)
{
return (x + y + z);
}
public double sum(double x, double y)
{
return (x + y);
}
public static void main(String args[])
{
        Sum s = new Sum();
        System.out.println(s.sum(10, 20));
        System.out.println(s.sum(10, 20, 30));
        System.out.println(s.sum(10.5, 20.5));
}
}
```

Overloaded sum(). This sum takes two int parameters

Overloaded sum(). This sum takes three int parameters

Overloaded sum(). This sum takes two double parameters

Different Methods of Method Overloading

- By number of parameters in two methods
  - public int add(int a, int b)
  - public int add(int a, int b, int c)
- By data types of the parameters of methods
  - public int add(int a, int b, int c)
  - public double add(double a, double b, double c)
- By order of the parameters of methods
  - public void display(String a, int b)
  - public void display(int b, String a)

Method Overriding and Overloading

- Overridden methods are in different classes related by inheritance; overloaded methods can be either in the same class or different classes related by inheritance.
- Overridden methods have the same signature and return type; overloaded methods have the same name but a different parameter list.

```
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);
  }
}
```

Overriding

```
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 overloads the method in B
  public void p(int i) {
    System.out.println(i);
  }
}
```

Overloading

Exercise 4.1. Design a class named BankAccount and its subclass OverdraftAccount.

- BankAccount class has
  - instance variables: **ownersName, accountNumber, balance**,
  - three argument constructor and
  - methods: deposit(double anAmount), withdraw(double anAmount) and getBalance() which returns the current balance. deposit method adds deposit amount(anAmount) to the current balance if deposit amount is above 0.0. withdraw method updates the current balance by subtracting withdraw amount(anAmount) from it if the withdraw amount does not exceeds the current balance.
- For OverdraftAccount,
  - there is one instance variable called limit which is used to limit the withdraw amount of OverdraftAccount holders.
  - Override the withdraw method of its superclass to add limit to the withdraw amount.
  - Then, write a main method which creates one BankAccount object and OverdraftAccount object, do deposits and printout balance, and do withdraws and printout balance.

```
class BankAccount
{
    private String ownersName;
    private int accountNumber;
    protected float balance;
    BankAccount(String oName,int accNumber,  double b)
    {
    ownersName=oName;
    accountNumber=accNumber;
    balance=b;
    }
    public void deposit(float anAmount)
    {
    if (anAmount>0.0)
            balance = balance + anAmount;
    }
    public void withdraw(double anAmount)
    {
    if ((anAmount>0.0) && (balance>anAmount))
            balance = balance - anAmount;
    }
    public float getBalance()
    {
    return balance;
    }
}
```

```
public class OverdraftAccount extends BankAccount
{
   private double limit;
   OverdraftAccount(String oName,int accNumber,double b,double l)
   {
         super(oName,accNumber,b);
         limit=l;
   }
    @Override
   public void withdraw(double anAmount)
   {
         if ((anAmount>0.0) && (getBalance()>anAmount) && anAmount<limit)
                balance = balance - anAmount;
   }
   public static void main(String args[])
   {
         BankAccount ba=new BankAccount("Hla Hla", 123, 600);
         OverdraftAccount oa=new OverdraftAccount("Mya Mya", 456, 6000, 1000);
         ba.deposit(300000);
         oa.deposit(600000);
         System.out.println("Current balance of first account is "+ ba.getBalance());
         System.out.println("Current balance of first account is "+ oa.getBalance());
         ba.withdraw(300000);
         oa.withdraw(600000);
         System.out.println("Current balance of first account is "+ ba.getBalance());
         System.out.println("Current balance of first account is "+ oa.getBalance());

   }}
```

Topic 3

Abstraction

Using Abstract Classes

- Define generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details
- Determine the nature of the methods that the subclasses must implement
- Java Abstract class is used for abstraction in Java
- They are meant to hide the internal implementation and only showing the functionality to the users, i.e. showing what is works and hiding how it works.

abstract *type* name (parameter-list)

- When to use?
  - When some classes need to share some codes then we put these classes in abstract classes.
  - When a state of an object is to define because we need to define a non-static or non-final fields.

Example 4.6: A program that demonstrates using Abstract class and methods.

```
// Using abstract methods and classes.
abstract class Figure {
        double dim1;
        double dim2;
        Figure(double a, double b) {
        dim1 = a;
        dim2 = b;
        }
        // area is now an 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;
      }
}
class AbstractAreas {
      public static void main(String args[]) {
            // Figure f = new Figure(10, 10); // illegal now
            Rectangle r = new Rectangle(9, 5);
            Triangle t = new Triangle(10, 8);
            Figure figref; // this is OK, no object is created
            figref = r;
            System.out.println("Area is " + figref.area());
            figref = t;
            System.out.println("Area is " + figref.area());
      }
}
```

Using final

- To prevent Overriding
  - Methods can be qualified with the final modifier
  - Final methods cannot be overridden.
  - This can be useful for security purposes.

```
public final boolean validatePassword(String username, String Password)
{
   […]
```

- To prevent Inheritance
  - Classes can be qualified with the final modifier
  - The class cannot be extended
  - This can be used to improve performance.

```
public final class Color
{
   […]
```

Topic 4

Encapsulation

Encapsulation

- The process in which we wrap the data into a single unit.
- Encapsulation binds the data and code.
- It basically creates a shield and code cannot be accessed outside the shield or by any code outside the shield
- We can achieve encapsulation in Java by:
  - Declaring the variables of a class as private.
  - Providing public setter and getter methods to modify and view the variables values.

Data Field Encapsulation

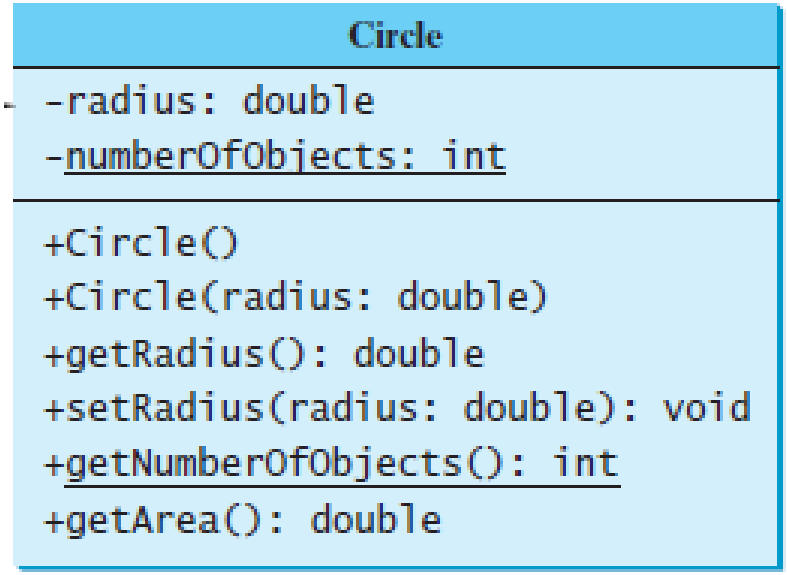
- **Accessor and Mutator**
  - To make a private data field accessible, provide a get method (accessor) to return the value of the data field.
  - To enable a private data field to be updated, provide a set method (mutator) to set a new value.
  - A get method is referred to as a **getter** (or accessor), and a set method is referred to as a **setter** (or mutator).
  - A get method has the following signature:

    public returnType getPropertyName()
  - If the returnType is boolean, the get method should be defined as follows by convention:

    public boolean isPropertyName()
  - A set method has the following signature:

    public void setPropertyName(dataType propertyValue)

Exercise 3.2. Create a new circle class with a private data-field radius, numberOfObjects and its associated accessor and mutator methods for the class diagram shown in Figure.

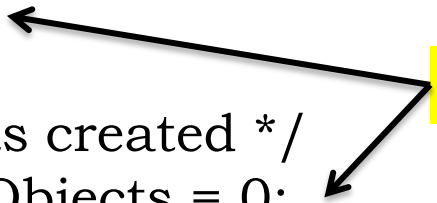


```
public class CircleWithPrivateDataFields {
  /** The radius of the circle */
  private double radius = 1;

  /** The number of the objects created */
  private static int numberOfObjects = 0;

  /** Construct a circle with radius 1 */
  public CircleWithPrivateDataFields() {
    numberOfObjects++;
  }

  /** Construct a circle with a specified radius */
  public CircleWithPrivateDataFields(double newRadius) {
    radius = newRadius;
    numberOfObjects++;
  }
```



```
  /** Return radius */
  public double getRadius() {
    return radius;
  }

  /** Set a new radius */
  public void setRadius(double newRadius) {
    radius = (newRadius >= 0) ? newRadius : 0;
  }

  /** Return numberOfObjects */
  public static int getNumberOfObjects() {
    return numberOfObjects;
  }

  /** Return the area of this circle */
  public double getArea() {
    return radius * radius * Math.PI;
  }
}
```

Accessor

Mutator

Accessor

Accessor

```
public class TestCircleWithPrivateDataFields {
  /** Main method */
  public static void main(String[] args) {
    // Create a Circle with radius 5.0
    CircleWithPrivateDataFields myCircle =
      new CircleWithPrivateDataFields(5.0);
    System.out.println("The area of the circle of radius "
      + myCircle.getRadius() + " is " + myCircle.getArea());

    // Increase myCircle's radius by 10%
    myCircle.setRadius(myCircle.getRadius() * 1.1);
    System.out.println("The area of the circle of radius "
      + myCircle.getRadius() + " is " + myCircle.getArea());
  }
}
```

Quiz

1. What keyword do you use to define a subclass?
2. Does Java support multiple inheritance?
3. True or false? You can override a private method defined in a superclass.
4. True or false? You can override a static method defined in a superclass.
5. How do you explicitly invoke a superclass's constructor from a subclass?

5. How do you invoke an overridden superclass method from a subclass?
6. If a method in a subclass has the same signature as a method in its superclass with the same return type, is the method overridden or overloaded?
7. If a method in a subclass has the same signature as a method in its superclass with a different return type, will this be a problem?
8. If a method in a subclass has the same name as a method in its superclass with different parameter types, is the method overridden or overloaded?
9. How does a subclass invoke its superclass's constructor?
10. True or false? When invoking a constructor from a subclass, its superclass's no-arg constructor is always invoked.

11. What is the printout of running the class C in Figure.

```
class A {
public A() {
      System.out.println(
      "A's no-arg constructor is invoked");
      }
}
class B extends A {
}
public class C {
      public static void main(String[] args) {
      B b = new B();
      }
}
```

12. What is the printout of the following codes.

```
public class Test {
    public static void main(String[] args) {
    A a = new A(3);
    }
}
class A extends B {
    public A(int t) {
    System.out.println("A's constructor is invoked");
    }
}
class B {
    public B() {
    System.out.println("B's constructor is invoked");
    }
}
```

Practical Assignments

1. Design a class named **Rectangle** to represent a rectangle. The class contains:

- Two **double** data fields named **width** and **height** that specify the width and height of the rectangle. The default values are **1** for both **width** and **height**.
- A no-arg constructor that creates a default rectangle.
- A constructor that creates a rectangle with the specified **width** and **height**.
- A method named **getArea()** that returns the area of this rectangle.
- A method named **getPerimeter()** that returns the perimeter.
- Write a test program that creates two **Rectangle** objects—one with width **4** and height **40** and the other with width **3.5** and height **35.9**.
- Display the width, height, area, and perimeter of each rectangle in this order.

2. Design a class named Fan to represent a fan. The class contains:

- Three constants named SLOW, MEDIUM, and FAST with the values 1, 2, and 3 to denote the fan speed.

- A private **int** data field named **speed** that specifies the speed of the fan (the default is **SLOW**).
- A private **boolean** data field named **on** that specifies whether the fan is on (the default is **false**).
- A private **double** data field named **radius** that specifies the radius of the fan (the default is **5**).
- A string data field named **color** that specifies the color of the fan (the default is **blue**).
- The accessor and mutator methods for all four data fields.
- A no-arg constructor that creates a default fan.
- A method named **toString()** that returns a string description for the fan. If the fan is on, the method returns the fan speed, color, and radius in one combined string. If the fan is not on, the method returns the fan color and radius along with the string "fan is off" in one combined string.

Write a test program that creates two **Fan** objects. Assign maximum speed, radius **10**, color **yellow**, and turn it on to the first object. Assign medium speed, radius **5**, color **blue**, and turn it off to the second object. Display the objects by invoking their **toString** method.

3. Design a class named **Person** and its two subclasses named **Student** and **Employee**. Make **Faculty** and **Staff** subclasses of **Employee**. A person has a name, address, phone number, and email address. A student has a class status (freshman, sophomore, junior, or senior). Define the status as a constant. An employee has an office, salary, and date hired. A faculty member has office hours and a rank. A staff member has a title.

Override the **toString** method in each class to display the class name and the person's name.

Write a test program that creates a **Person**, **Student**, **Employee**, **Faculty**, and **Staff**, and invokes their **toString()** methods.

Next Week Lecture

- **Packages**
- **Interfaces**

Thank you!