CompSci 230 S2
Object Oriented Software Development

Understanding Inheritance

Review Quizzes

- Consider the following code, what will be the value of finalAmount when it is displayed?
  - 528.00
  - 580.00
  - 522.00

```
public class Order{
  private int orderNum;
  private double orderAmount;
  private double orderDiscount;
  public Order(int orderNumber, double orderAmt,double orderDisc) {
    orderNum = orderNumber;
      orderAmount = orderAmt;
      orderDiscount = orderDisc;
   }
   public double finalOrderTotal(){
      return orderAmount - orderAmount *orderDiscount;
   }
}
```

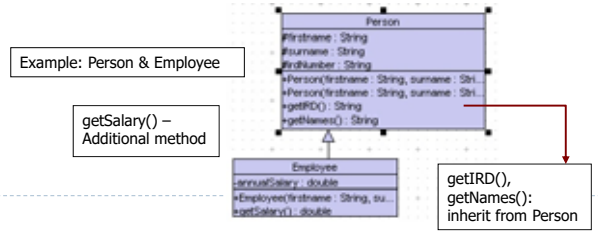
```
Order order = new 1234, 580.00, .1);
double finalAmount = order.finalOrderTotal();
System.out.println("Final order amount = $" + finalAmount);
```


Agenda & Reading

- Topics:
  - Introduction
  - Inheritance
  - Constructors
  - public, private & protectedMethod Overriding
  - The Object class
- Reading
  - Java how to program Late objects version (D & D)
    - Chapter 9
  - The Java Tutorial :
    - https://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html
    - Multiple Inheritance of State, Implementation, and Type
    - Overriding and Hiding Methods
    - Polymorphism
    - Hiding Fields
    - Using the Keyword super
    - Object as a Superclass
    - Writing Final Classes and Methods

1.Introduction

- Inheritance
  - A **new** class is created by acquiring an **existing** class's members and possibly embellishing them with new or modified capabilities.
    - Existing class is called the **superclass/base**
    - New class is called the **subclass/derived**
  - Can save time during program development by basing new classes on **existing** proven and **debugged** high-quality software.
  - Increases the likelihood that a system will be implemented and maintained effectively.
  - The former, known as **derived** classes, take over (or inherit) attributes and behaviour of the latter, which are referred to as **base** classes.

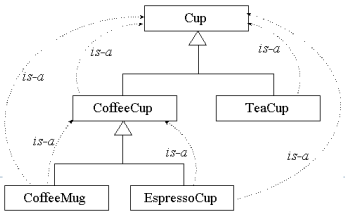


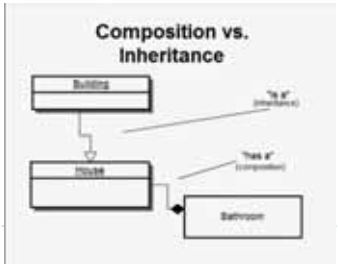
1.Introduction
Inheritance

- When creating a class, rather than declaring completely new members, you can designate that the new class
  - should inherit the members of an existing class
  - adds its own variables and methods
  - can change the meaning of inherited methods that are specific to the subclass.
- A subclass is more specific than its superclass and represents a more specialized group of objects.
- Every class in Java extends (or "inherits from") Object implicitly.
- A subclass can be a superclass of future subclasses.
- Java supports only single inheritance, in which each class is derived from exactly one direct superclass.

1.Introduction
Has-a Vs is-a relationship

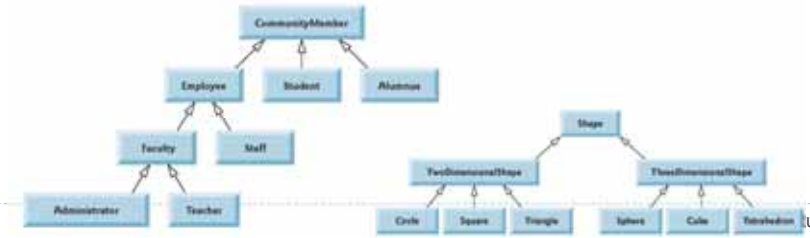
- We distinguish between the is-a relationship and the has-a relationship
- Is-a represents **inheritance**
  - In an is-a relationship, an object of a **subclass** can also be treated as an object of its superclass
- Has-a represents **composition**
  - In a has-a relationship, an object contains as members references to other objects





1.Introduction
Superclasses and Subclasses

- A superclass exists in a hierarchical relationship with its subclasses.
- Each arrow in the hierarchy represents an is-a relationship
- Starting from the bottom, you can follow the arrows and apply the is-a relationship up to the topmost superclass.
  - A Triangle is a TwoDimensionalShape and is a Shape
  - A Sphere is a ThreeDimensionalShape and is a Shape.



2.Inheritance

- Inheritance issue
  - A subclass can **inherit** methods that it does not need or should not have.
  - Even when a superclass method is appropriate for a subclass, that subclass often needs a **customized** version of the method.
  - The subclass can **override** (redefine) the superclass method with an appropriate implementation.
- For example: Sphere & ColoredSphere
  - The Sphere class is a base class
    - Instance Variable: radius
    - Instance Methods: setRadius(…), diameter(), area(), circumference(), toString()
    - …

```
public class Sphere {
  protected double theRadius;
  public Sphere() {
    setRadius(1.0);
  }
  ...
}
```

2.Inheritance
Example: Sphere & ColoredSphere

- The ColoredSphere is a derived class that
  - Inherits a field (radius) from the base class,
  - Inherits a few methods (setRadius, diameter … etc) from the base class,
  - Adds a new field: color
  - Adds two new methods: setColor, getColour
  - Overrides an existing method (toString) from the base class.

```
public class ColoredSphere extends Sphere {
  private Color color;
  public ColoredSphere(Color c) {
       super();
       color = c;
    }
  public Color getColor() {...}
  public String toString() {
    return super.toString() + ...}
  ...
}
```

```
Radius = 5.0,
Colour=java.awt.Color[r=255,g=255,b=255]
The ball color is
java.awt.Color[r=255,g=255,b=255]
```

```
ColoredSphere ball = new ColoredSphere(Color.white, 5);
System.out.println(ball);
System.out.println("The ball color is " + ball.getColor());
```


3.Constructors

B1.java

- Instantiating a subclass object begins a chain of constructor calls
- Each Subclass constructor must always call the superclass constructor (explicitly or implicitly)
  - Implicitly: calls its superclass default no-argument constructor if the code does not include an explicit call to the superclass constructor
  - Explicitly: using super()
    - If you call the superclass constructor explicitly, then the compiler will not call it implicitly.

```
class A1 {
  int x;
  public A1() {
    x=1;
    System.out.println("called A1()");
  }
  public A1(int x) {
    this.x = x;
    System.out.println("called A1(x)");
  }
}
```

```
class B1 extends A1 {
  int y;
  public B1() {
    System.out.println("called B1()");
  }
  public B1(int x, int y) {
    super(x);
    this.y = y;
    System.out.println("called B1(x,y)");
  }
}
```

Calls constructor implicitly

Calls constructor explicitly

3 2 4 1

```
B1 b1 = new B1();
B1 b2 = new B1(10, 100);
```

```
called A1()
called B1()
called A1(x)
called B1(x,y)
```


3.Constructors

B2.java

- A subclass cannot inherit constructors from the base class. Each subclass should define its constructor
  - If no constructor is defined, the compiler adds a single zero-parameter default constructor for the class and applies the default initialization for any data fields.

```
class A1 {
  int x;
  public A1() {
    x=1;
    System.out.println("called A1()");
  }
  public A1(int x) {
    this.x = x;
    System.out.println("called A1(x)");
  }
}
```

```
class B2 extends A1 {
  int y = 10;
}
```

Add a single zero-parameter default constructor

Calls base class's constructor implicitly

3 2

```
B2 b = new B2();
System.out.println("b.x=" + b.x);
System.out.println("b.y=" + b.y);
```

1

```
called A1()
b.x=1
b.y=10
```


3.Constructors

B2.java

- A subclass cannot inherit constructors from the base class. Each subclass should define its constructor
  - If the subclass does not have such a constructor, the compiler would issue an error.

```
class A1 {
  int x;
  public A1() {
    x=1;
    System.out.println("called A1()");
  }
  public A1(int x) {
    this.x = x;
    System.out.println("called A1(x)");
  }
}
```

```
class B2 extends A1 {
  int y = 10;
}
```

```
B2 b = new B2(10);
```

Compile time error :
No 1-argument constructor in the subclass

B2.java

3.Constructors

- A subclass cannot inherit constructors from the base class. Each subclass should define its constructor
  - If the subclass does not have such a constructor, the compiler would issue an error.

```
class A1 {
  int x;
  public A1() {
    x=1;
    System.out.println("called A1()");
  }
  public A1(int x) {
    this.x = x;
    System.out.println("called A1(x)");
  }
}
```

```
class B extends A1 {
  int y = 10;
  public B(int x) {
    super(x);
  }
}
```

```
B b = new B();
```

Compile time error:
No zero-parameter constructor in the subclass

B3.java

3.Constructors

- You can only call a superclass constructor from a subclass constructor, not from any other subclass method
- Never place any subclass constructor code ahead of its superclass constructor call (reason: a subclass constructor's initialisation may depend on the values declared in a superclass)

```
class A1 {
  int x;
  public A1() {
    x=1;
    System.out.println("called A1()");
  }
  public A1(int x) {
    this.x = x;
    System.out.println("called A1(x)");
  }
}
```

3

```
class B3 extends A1 {
  int y = x + 10;

  public B3(int x) {
    super(x);
  }
}
```

2

```
B3 b = new B3(100);
System.out.println("b.x=" + b.x);
System.out.println("b.y=" + b.y);
```

1

```
called A1(x)
b.x=100
b.y=110
```


4.public, private & protected

- Classes, and their fields and methods have access levels to specify how they can be used by other objects during execution
  - A **private** field or method is accessible only to the class in which it is defined.
  - A **protected** field or method is accessible to the class itself, its subclasses,
  - A **public** field or method is accessible to any class of any parentage in any package

4.public, private & protected

- A superclass's **private** members are **hidden** from its subclasses
  - They can be accessed only through the **public** or **protected** methods inherited from the superclass
- Subclass methods can refer to public and protected members inherited from the superclass simply by using the **member names**.
- When a subclass method overrides an inherited superclass method, the superclass version of the method can be accessed from the subclass by preceding the superclass method name with keyword ***super*** and a dot (.) separator

4.public, private & protected

- Example:

```
public class Sphere {
  protected double theRadius;
  public Sphere() {
    setRadius(1.0);
  }
  public Sphere(double r) {...}
  public void setRadius(double r) {...}
  public double radius() { ... }
  public String toString() {...}
  ...
}
```

```
public class ColoredSphere extends SimpleSphere {
  private Color color;
  public ColoredSphere(Color c) {
       super();
       color = c;
     }
  public void setColor(Color c) { ... }
  public Color getColor() {...}
  public String toString() {
    return super.toString() + ...}
  ...
  public void method1() {
    System.out.println(super.theRadius);
  }
}
```


5.Method Overriding

- You can change the meaning (override) of the method declared in the superclass
  - Completely / new implementation , or
  - Add more functionality to the method
    - The new method can call the original method in the parent class by specifying "super" before the method name.
- Rules:
  - A Subclass cannot override ***final*** methods declared in the base class.
  - The Overridden method must have the same arguments as the inherited method from the base class.

```
class Base2 {
  public final void finalMethod() {
    System.out.println("called Base:finalMethod");
  }
}
```

Final methods cannot be overridden.

Example: Derived2.java

```
public class Derived2 extends Base2 {
  public final void finalMethod() {
     System.out.println("called Derived:finalMethod");
  ...
```


5.Method Overriding New implementation

Derived.java

- To override a superclass method, a subclass must declare a method with the same signature as the superclass method
- @Override annotation
  - Indicates that a method should override a superclass method with the same signature.
  - If it does not, a compilation error occurs.

This method defined by the subclass is overridden to the method defined in the superclass.

```
class Base {
  public void aMethod() {
    System.out.println("called Base:aMethod");
  }
}
```

```
public class Derived extends Base
  public void aMethod() {
    System.out.println("called Derived:aMethod");
  }
  public static void main(String[] args) {
    Derived d = new Derived();
    d.aMethod();
  }
}
```

2

1

```
called Derived:aMethod
```


5.Method Overriding Overridden method from the superclass

Derived.java

- Placing the keyword super and a dot (.) separator before the superclass method name invokes the superclass version of an overridden method.
  - Good software engineering practice
  - If a method performs all or some of the actions needed by another method, call that method rather than duplicate its code.

```
class Base {
  public void add() {
    System.out.println("called Base:add");
  }
}
```

3

```
public class Derived extends Base {
  public void add() {
    super.add();
    System.out.println("called Derived:add");
  }
  public static void main(String[] args) {
    Derived d = new Derived();
    d.add();
  }
}
```

2

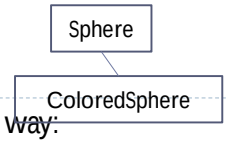
4

1

Use super.methodName() to call the method defined in the superclass

```
called Base:add
called Derived:add
```

5.Method Overriding Sphere & ColoredSphere



- We normally create an instance in the following way:

```
ColoredSphere s1 = new ColoredSphere(Color.blue);
```

Type of the variable (LHS) is the same as the type of the ColoredSphere created (RHS)

- An ColoredSphere object is also an Sphere object and it is also an Object. Therefore, we can assign …

Type of the variable (LHS) : Sphere

```
Sphere s2 = new ColoredSphere(Color.green);
```

Type of the variable (LHS) : Object

```
Object obj1 = new ColoredSphere(Color.red);
```

- And …

```
System.out.println(s1); // call the toString() from ColoredSphere
System.out.println(s2); // call the toString() from ColoredSphere
System.out.println(obj1); // call the toString() from ColoredSphere

Sphere s3 = new Sphere(10);
System.out.println(s3);     //call the toString() from Sphere
```


Exercise 1

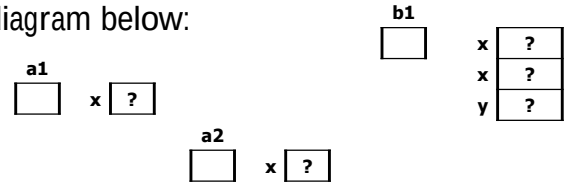
- Consider the following code fragment:

```
class B extends A {
  int x = 10;
  int y = 1;
  public B() {}
  public B(int y) { this.y = this.y + y; }
  public B(int x, int y) {
    super(x);
    this.y = this.y + y;
  }
}
```

```
class A {
  int x;
  public A() {
    this(100);
  }
  public A(int x) {
    this.x = x;
  }
}
```

- Complete the diagram below:

```
A a1 = new A();
A a2 = new A(10);
B b1 = new B();
```



Exercise 2

- Car & FunCar

```
public class FunCar extends Car{
  public FunCar() {
    // implicit call to super( ), which is Car( )
  }
  public FunCar(String color, String body) {
    super(color, body);
  }
  public String playCD() {
    return "(Beautiful music fills the passenger compartment.)";
  }
}
```


Exercise 2

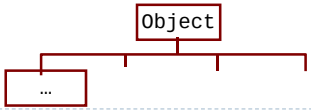
- What is the output of the following?

```
FunCar momsCar = new FunCar( );
System.out.println("Mom's car is a " + momsCar.toString( ));
System.out.println( momsCar.playCD( ) );
System.out.println( );

FunCar dadsCar = new FunCar("red", "convertible");
System.out.println("Dad's car is a " + dadsCar.toString( ));
System.out.println( dadsCar.playCD( ) );
```

```
class Car {
  String theColor = "blue";
  String theBody  = "wagon";
  public Car() {
    System.out.println("Called the default constructor Car().");
  }
  public Car(String color, String body) {
    System.out.println("Called the 2 args constructor Car().");
    theColor = color;
    theBody  = body;
  }
  public String toString() {
    return theColor + " " + theBody + ".";
  }
}
```

6.The Object class



- Every java class has Object as its superclass and thus inherits the Object methods.
- Object is a **non-abstract** class
- Many Object methods, however, have implementations that aren't particularly useful in general
- In most cases it is a good idea to override these methods with more useful versions.
  - equals: compares two objects for equality and returns true if they are equal.
  - toString: returns a String representation of the object

```
public class Object {
 ...
  public boolean equals(Object obj) {
    return (this == obj);
  }
  public String toString() {
    return getClass().getName() ...
  }
}
```


6.The Object class
The toString() method

- It is intended to return a readable textual representation of the object upon which it is called. This is great for debugging!
  - Returns a String representing an object.
  - Called implicitly whenever an object must be converted to a String representation.

```
System.out.println(someObject);
```

- Every class has a toString, even if it isn't in your code.
  - The default toString returns the class's name followed by a hexadecimal (base-16) number
- Replace the default behaviour by overriding the toString method in your class

```
public String toString() {
  return "(" + x + ", " + y + ")";
}
```


6.The Object class
The equals() method

- By default, equals(Object o) does exactly what the == operator does – compare object references
  - That is, two object are the same if the point to the same memory.
  - Since java does not support operator overloading, you cannot change this operator.
  - However, the equals method of the Object class gives you a chance to more meaningful compare objects of a given class.
    - returns true if they are actually the same object

```
Sphere sphere1 = new Sphere();
Sphere sphere2 = sphere1;
if (sphere1.equals(sphere2)) {
  System.out.println("same");
} else {
  System.out.println("different");
}
```

same

```
Sphere sphere1 = new Sphere(2.0);
Sphere sphere3 = new Sphere(2.0);
if (sphere1.equals(sphere3)) {
  System.out.println("same");
} else {
  System.out.println("different");
}
```

different

6.The Object class
The customize equals method

- To override, simply override method with version that does more meaningful test, i.e. compares **values** and returns true if equal, false otherwise
  - E.g. An equals methods that determines whether two sphere have the *same radius*

```
public boolean equals(Object rhs) {
 return (rhs instanceof MySphere) &&
         (theRadius == ((MySphere) rhs).theRadius);
}
```

```
MySphere sphere1 = new MySphere();
MySphere sphere2 = sphere1;
if (sphere1.equals(sphere2)) {
  System.out.println("same");
} else {
  System.out.println("different");
}
```

same

```
MySphere sphere1 = new MySphere(2.0);
MySphere sphere3 = new MySphere(2.0);
if (sphere1.equals(sphere3)) {
  System.out.println("same");
} else {
  System.out.println("different");
}
```

same

6.The Object class
The customize equals method

- Consider the equals method:

```
return (obj instanceof MySphere) &&
  (theRadius == ((MySphere) obj).theRadius);
```

  - We wish to implement equals() for MySphere which gives us value equality. Specifically, two MySphere objects should be considered equal if their the **radii** are the same.

```
sphere1.equals(new Point(1,2))
```

    - i.e. If the parameter is not a MySphere object, e.g.
      - The equals method should return false
      - With instanceof, we can check that obj is a MySphere object
  - Next, we need to get the radius from the parameter object for comparison. However, the signature of the equals method is:

```
public boolean equals(Object obj)
```

    - The type of the parameter "obj" is "Object". It does not have any information regarding to theRadius

```
obj.theRadius <- compiler error
```

    - We need to access MySphere instance variables, so that's why we cast obj to a MySphere object.
    - with instanceof again, we can check that obj is a MySphere object, so that doing the cast doesn't fail.

```
((MySphere)obj).theRadius
```


Usage of super & this

- super
  - Constructor : super() or super(…)
    - Automatically called in derived constructor if not explicitly called
    - Call to super() must be the first call in constructor
    - Cannot call super.super()
  - super.member
    - Members can be either method or instance variables
    - Refers to the members of the superclass of the subclass in which it is used
      - Note: a variable that has the same name as a variable in the superclass hides the superclass's member variable. The variable in the superclass cannot be referenced by its name and it must be accessed through "super" (later this week)
    - Used from anywhere within a method of the subclass
- this
  - Can be used inside any method to refer to the current object
  - Constructor: this(), this(…): refer to its constructor
  - this.member
    - Members can be either method or instance variables
    - this.instance_variable:
      - To resolve name-space collisions that might occur between instance variables and local variables

Exercise 3

- Complete the Book class:

The equals() method provided by Object tests whether the object references are equal—that is, if the objects compared are the exact same object. To test whether two objects are equal in the sense of equivalency (containing the same information), you must override the equals() method.

```
public class Book {
   String Title;
   String Author;
   String Publisher;
   String Year;
   String ISBN;
   ...
   public boolean equals(Object obj) {
      // complete this: check ISBN for equality

   }
   ...
 }
```

```
Book firstBook  = new Book("The JFC Swing Tutorial", "Kathy Walrath", "0201914670");
Book secondBook = new Book("The JFC Swing Tutorial", "Kathy Walrath", "0201914670");
if (firstBook.equals(secondBook)) {
  System.out.println("equivalent objects");
} else {
System.out.println("non-equivalent objects");
}
```


Review

- Except for the Object class, a class has exactly one direct superclass.
- A class inherits fields and methods from all its superclasses, whether direct or indirect.
- Overloading VS Overriding
  - Same method name & …

	Within a class	Parent & Child class
Same method signature	Compile-time error	Overriding
different method signature	Overloading	Overloading