Lecture Notes
Chapter #10
Inheritance & Polymorphism

- **Inheritance** – results from deriving new classes from existing classes
- **Root Class** – all java classes are derived from the `java.lang.Object` class

A child class inherits all accessible data fields and methods from its parent class!
A child class does not inherit the constructors of the parent class!
The child class may also add uniquely new data fields and methods!
1. Implementation

a. GeometricObject1.java

```java
public class GeometricObject1
{
    private String color = "white";
    private boolean filled;
    private java.util.Date dateCreated;

    public GeometricObject1( ) { dateCreated = new java.util.Date( ); }
    public String getColor( ) { return color; }
    public void setColor(String color) { this.color = color; }
    public boolean isFilled( ) { return filled; }
    public void setFilled(boolean filled) { this.filled = filled; }
    public java.util.Date getDateCreated( ) { return dateCreated; }
    public String toString( ) { return "created on " + dateCreated
                              + "ncolor: " + color + " and filled: " + filled; }
}
```

b. Circle4.java

```java
public class Circle4 extends GeometricObject1
{
    private double radius;

    public Circle4( ) {}
    public Circle4(double radius ) { this.radius = radius; }

    public double getRadius( ) { return radius; }
    public void setRadius( double radius) { this.radius = radius; }
    public double getArea( ) { return radius * radius * Math.PI; }
    public double getDiameter( ) { return 2 * radius; }
    public double getPerimeter( ) { return 2 * radius * Math.PI; }

    public void printCircle( )
    {
        System.out.println("The circle is created " + getDateCreated( ) + " and the radius is " + radius);
    }
}
```
c. Rectangle1.java

```java
public class Rectangle1 extends GeometricObject1 {
    private double width;
    private double height;

    public Rectangle1() {} 
    public Rectangle1(double width, double height) {
        this.width = width;
        this.height = height;
    }

    public double getWidth() { return width; }
    public void setWidth(double width) { this.width = width; }

    public double getHeight() { return height; }
    public void setHeight(double height) { this.height = height; }

    public double getArea() { return width * height; }
    public double getPerimeter() { return 2 * (width + height); }
}
```

d. TestCircleRectangle.java

```java
public class TestCircleRectangle {
    public static void main(String[] args) {
        Circle4 circle = new Circle4(1);
        System.out.println(circle.toString());
        System.out.println(circle.getRadius());
        System.out.println(circle.getArea());
        System.out.println(circle.getDiameter());

        Rectangle1 rectangle = new Rectangle1(2,4);
        System.out.println(rectangle.toString());
        System.out.println(rectangle.getArea());
        System.out.println(rectangle.getPerimeter());
    }
}
```

See Liang page 334 for output of TestCircleRectangle.java
Remark: A subclass is NOT a subset of its superclass; in fact, since the subclass has access to more items than the superclass, an instance of the superclass can be thought of as a subset of an instance of the subclass!

Remark: Inheritance is used to model *is-a* relationships; e.g., an apple is a fruit! For a class B to extend a class A, class B should contain more detailed information than class A. A subclass and a superclass must have an *is-a* relationship

Remark: C++ allows inheritance from multiple classes; i.e., it supports multiple inheritance.

Remark: Java does not allow inheritance from multiple classes; a Java class may inherit directly only from one superclass, i.e., the restriction is known as single inheritance. If the *extends* keyword is used to define a subclass, it allows only one parent class. Multiple inheritance in java is achieved by the use of *interfaces*.

2. Constructor Chaining

- A child class inherits all accessible data fields and methods from its parent class, BUT the child class does not inherit the constructors of the parent class!

- “*this*” keyword – refers to the calling object – *self-referential*

- “*super*” keyword – refers to the parent of the calling object – used to
  - call a superclass constructor
    - super( ) invokes the no-arg constructor of its superclass
    - super(argument list) invokes the superclass constructor that matches the argument list
    - the call for a superclass constructor must be the first statement in the subclass constructor
    - invoking a superclass constructor name in a subclass causes a syntax error
    - if a subclass does not explicitly invoke its superclass constructor, the compiler places the “super( )” statement as the first line in the subclass constructor, i.e.,

      ```java
      public A( ){ }  public A( ){ super( ); }
      ```
public class Faculty extends Employee {
    public static void main(String[] args) {
        new Faculty();
    }
    public Faculty() {
        System.out.println("(4) Faculty no-arg constructor invoked");
    }
}
class Employee extends Person {
    public Employee() {
        this("(2) Employee's overloaded constructor invoked");
        System.out.println("(3) Employee's no-arg constructor invoked");
    }
    public Employee(String s) {
        System.out.println(s);
    }
}
class Person {
    public Person() {
        System.out.println("(1) Person's no-arg constructor invoked");
    }
}

Construction of the Faculty Object
The Parent Constructor is always invoked before the Child Constructor
The object is built like a layer cake from the bottom-up
public class Apple extends Fruit {
    public Apple() {
    }
}

class Fruit {
    public Fruit(String name) {
        System.out.println("Fruit constructor is invoked");
    }
}

Since the Apple class does not have any constructors, a no-arg constructor is implicitly declared.

The Apple no-arg constructor automatically invokes the Fruit no-arg constructor; but Fruit does not have a no-arg constructor. But since Fruit has an explicitly declared constructor with a parameter, i.e.,
public Fruit(String name), then the compiler cannot implicitly invoke a no-arg constructor.

Hence, an Apple object cannot be created and the program cannot be compiled!

Best Practices
PROVIDE EVERY CLASS WITH A NO-ARG CONSTRUCTOR
SUCH A POLICY AIDS THE EXTENSION OF THE CLASS, I.E.,
IT AVOIDS THE ERROR DELINEATED ABOVE
3. Overriding Methods

- "super" keyword is also used to call a superclass method
- subclasses inherit methods from their superclasses
- a subclass may modify the definition of an inherited method for use in that subclass – method overriding

```java
public class GeometricObject1 {
    private String color = "white";
    private boolean filled;
    private java.util.Date dateCreated;
    public GeometricObject1() { dateCreated = new java.util.Date(); }
    public String getColor() { return color; }
    public void setColor(String color) { this.color = color; }
    public boolean isFilled() { return filled; }
    public void setFilled(boolean filled) { this.filled = filled; }
    public java.util.Date getDateCreated() { return dateCreated; }
    public String toString() {
        return "created on " + dateCreated + "\ncolor: " + color + " and filled: " + filled;
    }
}
```

```java
public class Circle4 extends GeometricObject1 {
    private double radius;
    public Circle4() {
    }
    public Circle4(double radius) {
        this.radius = radius;
    }
    public double getRadius() {
        return radius;
    }
    public void setRadius(double radius) {
        this.radius = radius;
    }
    public double getArea() {
        return radius * radius * Math.PI;
    }
    public double getDiameter() {
        return 2 * radius;
    }
    public double getPerimeter() {
        return 2 * radius * Math.PI;
    }
    public void printCircle() {
        System.out.println("The circle is created " + getDateCreated() + " and the radius is " + radius);
    }
    public String toString() {
        return super.toString() + "\nradius is " + radius;
    }
}
```

The `Circle4` `toString()` method overrides the `GeometricObject1` `toString()` method; it invokes the `GeometricObject1` `toString()` method and then modifies it to specify information specific to the `Circle4` object.
a. Rules for Overriding Inherited Methods

- **private data fields** in a superclass are not accessible outside of that class, hence they cannot be used directly by a subclass; they can be accessed &/or mutated by **public accessor &/or mutators** defined in the superclass.

- An **instance method** can be overridden only if it is accessible; **private methods cannot be overridden**.

- If a method defined in a subclass is private in its superclass, the two methods are completely unrelated.

- A **static method** can be inherited, but a static method **cannot be overridden**.

- Remember that **static methods are class methods**.

- If a static method defined in a superclass is **redefined** in a subclass, the method defined in the superclass is hidden; the hidden static method can be invoked by using the syntax “**SuperClassName.staticMethodName();**”

b. Overriding versus Overloading

i. **Overloading** – same name, **different signatures**

ii. **Overriding** – method defined in the superclass, overridden in a subclass using the same name, **same signature**, and **same return type** as defined in the superclass.

```java
public class Test {
    public static void main(String [] args) {
        A a = new A();
        a.p(10);
    }
}

class B {
    public void p(int i){ }
}

class A extends B {
    public void p(int i) {
        System.out.println(i);
    }
}
```

```java
public class Test {
    public static void main(String [] args) {
        A a = new A();
        a.p(10);
    }
}

class B {
    public void p(int i){ }
}

class A extends B {
    public void p(double i) {
        System.out.println(i);
    }
}
```

- a.p(10) invokes class A method; hence prints 10
- a.p(10) invokes class B method; hence prints nothing
- overloads
4. **Object** Class & Methods

- Every class in Java is descended from `java.lang.Object`

- If no inheritance is declared when a class is defined, the class is a subclass of **Object** by default

- **public String toString( );**
  - returns a string consisting of the objects name, the `@` sign, and the objects memory address in hexadecimal, e.g., `student@B7F9A1`

  - Override the `toString( )` method to produce relevant information concerning the subclass objects

  - `System.out.println(student); ➔ System.out.println(student.toString( ));`

- **public boolean equals(Object obj) { return (this == obj); }**
  - default implementation tests whether two reference variables point to the same object

  - Override the `equals( )` method to test whether two distinct objects have the same content, e.g.,

    ```java
    public boolean equals(Object o)
    {
        if (o instanceof Circle)
        {
            return radius == ((Circle)o).radius;
        }
        else return false;
    }
    ```

- **Comparison Operators/Methods**

  - “==” operator is used to compare primitive data type values

  - “==” operator is also used to compare whether two reference variables refer to the same object (where arrays may be considered to be objects)

  - The modified “equals( )” method can be used to determine whether two objects have the same contents

  - The “equals( )” method can be modified to test the contents of all or a selected subset of the data fields in the class

---

**INSTANCEOF OPERATOR**

- `o instanceof Circle`
  - returns true if `o` is an instance of Circle

  - Do not use `(Circle o)` as the argument when overriding the `equals( )` method, i.e., do not use the signature `public boolean equals( Circle o)` see page 355 #10.12
5. Polymorphism, Dynamic & Genetic Programming

- a class defines a type
- a type defined by a subclass is a subtype
- a type defined by a superclass is a supertype

- a variable must be declared to be of a specific type
- the type of a variable called it's declared type
- a variable of a reference type can hold a null value or a reference to an object

- an object is an instance of a class
- a subclass is a specialization of its superclass

  every instance of a subclass is an instance of its superclass
  - every circle is an object

  an instance of a superclass is not an instance of a subclass
  - not every object is a circle

- an instance of a subclass can be passed to a parameter of its superclass, i.e., a Circle object can be passed to a GeometricObject class parameter

- polymorphism – an object of a subtype can be used whenever its superclass object is required; i.e., a variable of a supertype can refer to a subtype object

- dynamic binding – given an inheritance chain as follows,

```
class C1
class C2
class C3
class C4
```

and the object

```
C1 o = new C1();
```

if the object o were to invoke a method, i.e., o.p(); then the JVM searches for the method p() in the classes in the order C1, C2, C3, C4, java.lang.Object once an implementation of p() is found, the search stops and that implementation of p() is invoked
public class PolymorphismDemo
{
    public static void main(String[ ] args)
    {
        m(new GraduateStudent( ));
        m(new Student( ));
        m(new Person( ));
        m(new Object( ));
    }

    public static void m(Object x)
    {
        System.out.println(x.toString( ));
    }
}

class GraduateStudent extends Student { }
class Student extends Person { public String toString( ) { return “Student”; } }
class Person extends Object { public String toString( ) { return “Person”; } }

A reference variable’s declared type determines which method is matched at compile time; i.e., the compiler uses the parameter type, the number & order of parameters to determine the matching method.

For a method defined in several subclasses, the JVM dynamically binds the implementation of a method at runtime decided by the actual class of the object referenced by the variable.

Polymorphism refers to the use a variable of a supertype to refer to an object of a subtype; the implementation is known as generic programming.
If a methods parameter type is a superclass, then an object of any of the subclasses may be passed to the method via that parameter type.

6. Casting Objects & the `instanceof` Operator

a. Implicit Casting
   
   Object o = new Student( );
   m( o );
   
   equivalent statements
   
   \[ \begin{align*}
   m( \text{new Student( )});
   \end{align*} \]

   An instance of Student is automatically an instance of Object

b. Explicit Casting
   
   Object o;
   Student b = o; \Rightarrow \text{compilation error!}

   An instance of Object is not necessarily an instance of Student

   Student b = (Student) o;

c. Up Casting
   
   Casting an instance of a subclass to a variable of a superclass is always possible; implicit casting may be used.

d. Down Casting
   
   Casting an instance of a superclass to a variable of a subclass: must use explicit casting & object cast must be an instance of the subclass

   error message ClassCastException

e. `instanceof` Operator

   Object o = new Circle( );
   if( o instanceof Circle )
   {
       double d = ((Circle) o).getDiameter( );
   }

   The declared type determines which method to match at compile time;
   “o.getDiameter( );” would cause a compile error since Object does not contain a “getDiameter( )” method.

   To enable Generic Programming, declare variables with their supertype; thus they can accept a value of any type.
f. TestPolymorphismCasting.java

```java
public class TestPolymorphismCasting {
    public static void main(String[] args) {
        Object o1 = new Circle4(1);
        Object o2 = new Rectangle1(1, 1);
        displayObject(o1);
        displayObject(o2);
    }
    public static void displayObject(Object o) {
        if (o instanceof Circle4) {
            System.out.println(((Circle4) o).getArea());
            System.out.println(((Circle4) o).getDiameter());
        } else if (o instanceof Rectangle1) {
            System.out.println(((Rectangle1) o).getArea());
        }
    }
}
```

Note: \((\text{Circle4} \ o).\text{getArea}())\n
Casting must be executed before the object member access operator.

7. ArrayList Class

```
Java.util.ArrayList
+ArrayList():
+add(o: Object): void
+add(index: int, o: Object): void
+clear(): void
+contains(o: Object): boolean
+get(index: int): Object
+indexOf(o: Object): int
+isEmpty(): boolean
+lastIndexOf(o: Object): int
+remove(o: Object): boolean
+remove(index: int): boolean
+size(): int
+set(index: int, o: Object): Object
```

See page 346 for description of the ArrayList operators.

See Liang page 348 for a list of the differences & similarities between ArrayList operations and Array operations.

Notice:
- Arrays are fixed in size at creation
- ArrayLists are extensible at any time
Listing 10.8 TestArrayList.java

```java
public class TestArrayList
{
    public static void main(String[ ] args)
    {
        java.util.ArrayList cityList = new java.util.ArrayList();
        cityList.add("London");
        cityList.add("New York");
        cityList.add("Paris");
        cityList.add("Los Angeles");
        cityList.add("Mexico City");
        cityList.add("Toyko");
        System.out.print("List Size: "+cityList.size());
        System.out.print("Toronto is in list: "+cityList.contains("Toronto");
        System.out.print("Index of New York: +cityList.indexOf("New York");
        System.out.println("Empty List: +cityList.isEmpty()");
        cityList.add("Beijing");
        cityList.remove("London");
        cityList.remove(1);
        System.out.println(cityList.toString());
        for( int i = cityList.size() - 1; i >= 0; i-- ) System.out.print(cityList.get(i) + " ");
        System.out.println();

        java.util.ArrayList list = new java.util.ArrayList();
        list.add(new Circle4(2));
        list.add(new Circle4(3));
        System.out.print("Area of first circle: +((Circle4)list.get(0)).getArea()");
        System.out.print("Area of second circle: +((Circle4)list.get(1)).getArea()");
    }
}
```

See page 348 for compiler warning when compiling from the command