CS5000:
Foundations of Programming

Mington Kang, PhD
Computer Science, Kennesaw State University
Three main programming mechanisms that constitute object-oriented programming (OOP)

- Encapsulation
- Inheritance
- Polymorphism
Encapsulation

- To hide the implementation details from users
- In Java, a mechanism of wrapping the data and code acting one the data together.
  - The variables of a class will be hidden from other class with access modifiers.
  - Then, the variables can be accessed only through the methods of their current class. → Data hiding

http://beginnersbook.com/2013/05/encapsulation-in-java/
Encapsulation

- Conventional Approach
  - Declare the variables of a class as private
  - Provide public setter/getter methods to modify and view the variables values

http://www.tutorialspoint.com/java/java_encapsulation.htm
Benefits of Encapsulation:

- The fields of a class can be made read-only or write-only.
- A class can have total control over what is stored in its fields.
- The users of a class do not know how the class stores its data.
- A class can change the data type of a field and users of the class do not need to change any of their code.

[http://www.tutorialspoint.com/java/java_encapsulation.htm](http://www.tutorialspoint.com/java/java_encapsulation.htm)
Polymorphism

- Literally means
  - a state of having many shapes
  - The capacity to take on different forms
- In Java, process objects of various types and classes through a single/uniform interface.

Polymorphism

Example:

In Pac-man game

- Two objects: Pac-man and Ghost
- Both pac-mans and ghosts can be derived from a PacManObject class that defines basic behaviors of the objects: Move()
- However, Pac-man and Ghost may have different details in the Move() method.
- Pac-man eats pac-dots, while ghosts do not
Method Overloading

- One solution for polymorphism
  - Overloaded Methods

```java
public class Pacman {
    public void move() {
        System.out.println("Pacman::move()"坪);
    }

    public void move(boolean eatPacdots){
        if (eatPacdots) eatPacdots();
    }
}
```
Method Overloading

- One solution for polymorphism
  - Overloaded Methods

```java
public class PacmanGame {
    public static void main(String[] args) {
        Pacman pacman = new Pacman();
        pacman.move();
        pacman.move(true);
    }
}
```
Method Overriding

- A variable is declared as superclass but is created with child class.
  - Java decides what method to call during runtime

```java
public class PacmanObject {
    public void move() {
        System.out.println("PacmanObject:Move()");
    }
}

public class Ghost extends PacmanObject {
}
```
Method Overriding

- A variable is declared as superclass but is created with child class.
  - Java decides what method to call during runtime

```java
public class Pacman extends PacmanObject {
    public void move() {
        super.move();
        System.out.println("Pacman::move()");
    }

    public void move(boolean eatPacdots){
        super.move();
        if (eatPacdots) eatPacdots();
    }
}
```
Method Overriding

- A variable is declared as superclass but is created with child class.
  - Java decides what method to call during runtime

```java
public class PacmanGame {
    public static void main(String[] args) {
        // obj is declared as an PacmanObject
        PacmanObject obj1 = new Pacman();
        PacmanObject obj2 = new Ghost();
        obj1.move();
        obj2.move();
    }
}
```
Method Overriding

- A variable is declared as superclass but is created with child class.
  - Java decides what method to call during runtime

```java
public class PacmanGame {
    public static void main(String[] args) {
        PacmanObject[2] objs;
        objs[0] = new Pacman();
        objs[1] = new Ghost();
        for (int i = 0; i < objs.length; i++)
            objs[i].move();
    }
}
```
Method Binding

- Process to decide what method to call
- Two types
  - Static Polymorphism (early binding)
    - Compile time polymorphism
    - Private, final, and static methods
    - Method overloading
  - Dynamic polymorphism (late binding)
    - Runtime binding
    - Method overriding
class Top {
    public String f(Object o) { return "Top"; }
}

class Sub extends Top {
    public String f(String s) { return "Sub"; }
    public String f(Object o) { return "SubObj"; }
}
public class Test {
    public static void main(String[] args) {
        Sub sub = new Sub();
        Top top = sub;
        String str = "Something";
        Object obj = str;
        System.out.println(top.f(obj));
        System.out.println(top.f(str));
        System.out.println(sub.f(obj));
        System.out.println(sub.f(str));
    }
}
Sub sub = new Sub(); // Ref: Sub, Act: Sub
Top top = sub; // Ref: Top, Act: Sub
String str = "Something"; // Ref: String, Act: String
Object obj = str; // Ref: Object, Act: String
top.f(obj); // Top.f(Object) at compile-time
    // Sub.f(Object) at run-time
top.f(str); // Top.f(Object) at compile-time
    // Sub.f(Object) at run-time
sub.f(obj);
sub.f(str); // Static Binding (overloading)