Building Java Programs

Chapter 8
Lecture 19: encapsulation, inheritance

reading: 8.5 - 8.6

(Slides adapted from Stuart Reges, Hélène Martin, and Marty Stepp)
WE TOOK THE HOSTAGES, SECURED THE BUILDING, AND CUT THE COMMUNICATION LINES LIKE YOU SAID.

EXCELLENT.

BUT THEN THIS GUY CLIMBED UP THE VENTILATION DUCTS AND WALKED ACROSS BROKEN GLASS, KILLING ANYONE WE SENT TO STOP HIM.

AND HE RESCUED THE HOSTAGES?

NO, HE IGNORED THEM. HE JUST RECONNECTED THE CABLES WE CUT, MUTTERING SOMETHING ABOUT "UPTIME".

SHIT, WE'RE DEALING WITH A SYSADMIN.
**Encapsulation**

- **encapsulation**: Hiding implementation details of an object from its clients.
  - Encapsulation provides *abstraction*.
    - separates external view (behavior) from internal view (state)
  - Encapsulation protects the integrity of an object's data.
Private fields

• A field can be declared *private*.  
  • No code outside the class can access or change it.

  ```java
  private type name;
  ```

• Examples:

  ```java
  private int id;
  private String name;
  ```

• Client code sees an error when accessing private fields:

  ```java
  PointMain.java:11: x has private access in Point
  System.out.println("p1 is (" + p1.x + ", " + p1.y + ")");
  ^
  ```
Accessing private state

- We can provide methods to get and/or set a field's value:

  // A "read-only" access to the x field ("accessor")
  public int getX() {
      return x;
  }

  // Allows clients to change the x field ("mutator")
  public void setX(int newX) {
      x = newX;
  }

- Client code will look more like this:

  System.out.println("pl: (" + pl.getX() + ", " + pl.getY() + ")");
  pl.setX(14);
// A Point object represents an (x, y) location.
public class Point {
    private int x;
    private int y;

    public Point(int initialX, int initialY) {
        x = initialX;
        y = initialY;
    }

    public double distanceFromOrigin() {
        return Math.sqrt(x * x + y * y);
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    public void setLocation(int newX, int newY) {
        x = newX;
        y = newY;
    }

    public void translate(int dx, int dy) {
        x = x + dx;
        y = y + dy;
    }
}
public class PointMain4 {
   public static void main(String[] args) {
      // create two Point objects
      Point p1 = new Point(5, 2);
      Point p2 = new Point(4, 3);

      // print each point
      System.out.println("p1: (" + p1.getX() + ", " + p1.getY() + ")");
      System.out.println("p2: (" + p2.getX() + ", " + p2.getY() + ")");

      // move p2 and then print it again
      p2.translate(2, 4);
      System.out.println("p2: (" + p2.getX() + ", " + p2.getY() + ")");
   }
}

OUTPUT:
p1 is (5, 2)
p2 is (4, 3)
p2 is (6, 7)
Benefits of encapsulation

- Provides abstraction between an object and its clients.
- Protects an object from unwanted access by clients.
  - A bank app forbids a client to change an `Account`'s balance.
- Allows you to change the class implementation.
  - `Point` could be rewritten to use polar coordinates (radius $r$, angle $\theta$), but with the same methods.
- Allows you to constrain objects' state (invariants).
  - Example: Only allow `Points` with non-negative coordinates.
Inheritance

reading: 9.1
Law firm employee analogy

- common rules: hours, vacation, benefits, regulations …
  - all employees attend a common orientation to learn general company rules
  - each employee receives a 20-page manual of common rules

- each subdivision also has specific rules:
  - employee receives a smaller (1-3 page) manual of these rules
  - smaller manual adds some new rules and also changes some rules from the large manual
Separating behavior

• Why not just have a 22 page Lawyer manual, a 21-page Secretary manual, a 23-page Marketer manual, etc.?

• Some advantages of the separate manuals:
  • maintenance: Only one update if a common rule changes.
  • locality: Quick discovery of all rules specific to lawyers.

• Some key ideas from this example:
  • General rules are useful (the 20-page manual).
  • Specific rules that may override general ones are also useful.
Is-a relationships, hierarchies

- **is-a relationship**: A hierarchical connection where one category can be treated as a specialized version of another.
  - every marketer *is an* employee
  - every legal secretary *is a* secretary

- **inheritance hierarchy**: A set of classes connected by is-a relationships that can share common code.
Employee regulations

- Consider the following employee regulations:
  - Employees work 40 hours / week.
  - Employees make $40,000 per year, except legal secretaries who make $5,000 extra per year ($45,000 total), and marketers who make $10,000 extra per year ($50,000 total).
  - Employees have 2 weeks of paid vacation leave per year, except lawyers who get an extra week (a total of 3).
  - Employees should use a yellow form to apply for leave, except for lawyers who use a pink form.

- Each type of employee has some unique behavior:
  - Lawyers know how to sue.
  - Marketers know how to advertise.
  - Secretaries know how to take dictation.
  - Legal secretaries know how to prepare legal documents.
// A class to represent employees in general (20-page manual).
public class Employee {
    public int getHours() {
        return 40;  // works 40 hours / week
    }

    public double getSalary() {
        return 40000.0;  // $40,000.00 / year
    }

    public int getVacationDays() {
        return 10;  // 2 weeks' paid vacation
    }

    public String getVacationForm() {
        return "yellow";  // use the yellow form
    }
}

• Exercise: Implement class Secretary, based on the previous employee regulations. (Secretaries can take dictation.)
Redundant Secretary class

// A redundant class to represent secretaries.
public class Secretary {
    public int getHours() {
        return 40; // works 40 hours / week
    }

    public double getSalary() {
        return 40000.0; // $40,000.00 / year
    }

    public int getVacationDays() {
        return 10; // 2 weeks' paid vacation
    }

    public String getVacationForm() {
        return "yellow"; // use the yellow form
    }

    public void takeDictation(String text) {
        System.out.println("Taking dictation of text: " + text);
    }
}
Desire for code-sharing

- `takeDictation` is the only unique behavior in `Secretary`.

- We'd like to be able to say:

```java
// A class to represent secretaries.
public class Secretary {
    copy all the contents from the Employee class;

    public void takeDictation(String text) {
        System.out.println("Taking dictation of text: " + text);
    }
}
```
Inheritance

• **inheritance**: A way to form new classes based on existing classes, taking on their attributes/behavior.
  - a way to group related classes
  - a way to share code between two or more classes

• One class can *extend* another, absorbing its data/behavior.
  - **superclass**: The parent class that is being extended.
  - **subclass**: The child class that extends the superclass and inherits its behavior.
    - Subclass gets a copy of every field and method from superclass
Inheritance syntax

public class name extends superclass {

• Example:

public class Secretary extends Employee {
    ...
}

• By extending Employee, each Secretary object now:
  • receives a getHours, getSalary, getVacationDays, and getVacationForm method automatically
  • can be treated as an Employee by client code (seen later)
// A class to represent secretaries.
public class Secretary extends Employee {
    public void takeDictation(String text) {
        System.out.println("Taking dictation of text: " + text);
    }
}

- Now we only write the parts unique to each type.
  - Secretary inherits getHours, getSalary, getVacationDays, and getVacationForm methods from Employee.
  - Secretary adds the takeDictation method.
Implementing Lawyer

- Consider the following lawyer regulations:
  - Lawyers who get an extra week of paid vacation (a total of 3).
  - Lawyers use a pink form when applying for vacation leave.
  - Lawyers have some unique behavior: they know how to sue.

- Problem: We want lawyers to inherit *most* behavior from employee, but we want to replace parts with new behavior.
Overriding methods

• **override**: To write a new version of a method in a subclass that replaces the superclass's version.
  
  • No special syntax required to override a superclass method. Just write a new version of it in the subclass.

```java
public class Lawyer extends Employee {
    // overrides getVacationForm method in Employee class
    public String getVacationForm() {
        return "pink";
    }
    ...
}
```

• Exercise: Complete the **Lawyer class**.
  
  • (3 weeks vacation, pink vacation form, can sue)
// A class to represent lawyers.
public class Lawyer extends Employee {
    // overrides getVacationForm from Employee class
    public String getVacationForm() {
        return "pink";
    }

    // overrides getVacationDays from Employee class
    public int getVacationDays() {
        return 15; // 3 weeks vacation
    }

    public void sue() {
        System.out.println("I'll see you in court!");
    }
}

Exercise: Complete the Marketer class. Marketers make $10,000 extra ($50,000 total) and know how to advertise.
Marketer class

// A class to represent marketers.
public class Marketer extends Employee {
    public void advertise() {
        System.out.println("Act now while supplies last!");
    }
    public double getSalary() {
        return 50000.0;  // $50,000.00 / year
    }
}
Levels of inheritance

• Multiple levels of inheritance in a hierarchy are allowed.
  • Example: A legal secretary is the same as a regular secretary but makes more money ($45,000) and can file legal briefs.

  ```java
  public class LegalSecretary extends Secretary {
    ...
  }
  ```

• Exercise: Complete the LegalSecretary class.
// A class to represent legal secretaries.
public class LegalSecretary extends Secretary {
    public void fileLegalBriefs() {
        System.out.println("I could file all day!");
    }

    public double getSalary() {
        return 45000.0;    // $45,000.00 / year
    }
}