Topic 31 - inheritance

Hierarchies

- Hierarchies used to organize information

Kingdom - Animalia

Phylum - Chordata

Class- Mammalia

Order - Carnivora

Family - Felidae

Genus - Felinae

Species - F. catus

More general

More specific

Hierarchies

- Object oriented languages provide a mechanism to create hierarchies among data types in a program and in code libraries
- Used for organization, modeling the problem, and to avoid redundant code
- When a new data type is a specialization or variation on an existing data type use inheritance to capture the relationship and avoid redundant of code

Inheritance in Practice

1. extends keyword
2. inheritance of instance methods
3. inheritance of instance variables
4. object initialization and constructors
5. calling a parent constructor with **`super()`**
6. overriding methods
7. partial overriding, **`super.parentMethod()`**
8. inheritance requirement in Java
9. the **`Object`** class

Pretty Stone

- Implement a Pretty Stone class
- Same as a Stone except alternates Color every (N + 1) times based on an int parameter to the constructor
- If parameter is [0..2] alternates between BLUE and RED
- If parameter [3..5] alternates between GREEN and YELLOW
- If parameter > 5 alternates between MAGENTA and ORANGE
- Pretty stones always return true when asked to eat.

Rolling Stone

- Implement a Rolling Stone class
- Same as a Pretty Stone ...
- except when a Rolling Stone is created it picks a random number of turns based on the int sent to the constructor.
  0 -> 0-99, 1 -> 100-199, 2 -> 200-299
- Stays still until asked to move that number of times, then moves North.
- fights: if not moving same as stone, but if moving a random attack

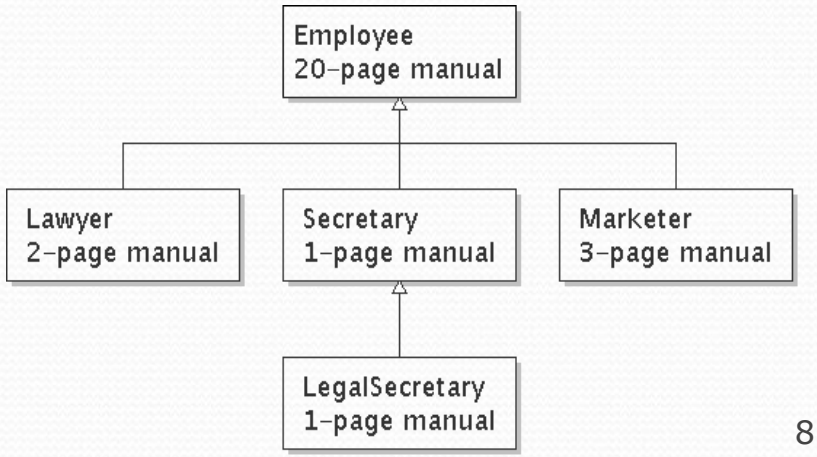
Another Example

Following slides contain another example of an inheritance hierarchy and Java syntax for implementing it.

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



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.

An `Employee` class

```
// 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:

```
// 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)

Improved `Secretary` code

```
// 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.

Lawyer class

```
// 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 ($5,000 more) and can file legal briefs.

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

  - Exercise: Complete the `LegalSecretary` class.

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
    }
}
```

Changes to common behavior

- Imagine a company-wide change affecting all employees.

Example: Everyone is given a $10,000 raise due to inflation.

- The base employee salary is now $50,000.
- Legal secretaries now make $55,000.
- Marketers now make $60,000.

- We must modify our code to reflect this policy change.

Modifying the superclass

```
// 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 50000.0;         // $50,000.00 / year
    }

    ...
}
```

- Are we finished?

- The `Employee` subclasses are still incorrect.
  - They have overridden `getSalary` to return other values.

An unsatisfactory solution

```
public class LegalSecretary extends Secretary {
    public double getSalary() {
        return 55000.0;
    }
    ...
}

public class Marketer extends Employee {
    public double getSalary() {
        return 60000.0;
    }
    ...
}
```

- Problem: The subclasses' salaries are based on the Employee salary, but the `getSalary` code does not reflect this.

Calling overridden methods

- Subclasses can call overridden methods with `super`

  `super.`***<method>***`(`***<parameters>***`)`

  - Example:

```
public class LegalSecretary extends Secretary {
    public double getSalary() {
        double baseSalary = super.getSalary();
        return baseSalary + 5000.0;
    }
    ...
}
```

  - Exercise: Modify `Lawyer` and `Marketer` to use `super`.

Improved subclasses

```
public class Lawyer extends Employee {
    public String getVacationForm() {
        return "pink";
    }

    public int getVacationDays() {
        return super.getVacationDays() + 5;
    }

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

public class Marketer extends Employee {
    public void advertise() {
        System.out.println("Act now while supplies last!");
    }

    public double getSalary() {
        return super.getSalary() + 10000.0;
    }
}
```

Given the Employee class to the right what is output by the following code?

```
Employee e1;
e1 = new Employee("#1");
String str;
str = e1.toString();
System.out.println(str);
```

A. #1

B. "#1"

C. output varies each time

D. syntax error

E. runtime error

```
// A class to represent employees
public class Employee {

    private String id;

    public Employee(String id) {
        this.id = id;
    }

    public int getHours() {
        return 40;
    }

    public double getSalary() {
        return 40000.0;
    }

    public int getVacationDays() {
        return 10;
    }

    public String getVacationForm() {
        return "yellow";
    }
}
```