Inheritance

Readings: 9.1

Writing classes

- Write an `Employee` class with methods that return values for the following properties of employees at a particular company:
  - Work week: 40 hours
  - Annual salary: $40,000
  - Paid time off: 2 weeks
  - Leave of absence form: Yellow form

`Employee` class

```
// A class to represent employees
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
    }
}
```

Writing more classes

- Write a `Secretary` class with methods that return values for the following properties of secretaries at a particular company:
  - Work week: 40 hours
  - Annual salary: $40,000
  - Paid time off: 2 weeks
  - Leave of absence form: Yellow form
- Add a method `takeDictation` that takes a string as a parameter and prints out the string prefixed by "Taking dictation of text: ".

`Secretary` class

```
// A 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);
    }
}
```

Why are they very similar?

```
// A class to represent employees
public class Employee {
    public int getHours() {
        return 40;
    }

    public double getSalary() {
        return 40000.0;
    }

    public int getVacationDays() {
        return 10;
    }

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

```
// A class to represent secretaries
public class Secretary {
    public int getHours() {
        return 40;
    }

    public double getSalary() {
        return 40000.0;
    }

    public int getVacationDays() {
        return 10;
    }

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

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

Is-a relationship

- **is-a relationship**: A hierarchical connection where one category can be treated as a specialized version of another.
- Examples:
  - Every secretary is an employee.
  - Every dog is a mammal.
  - Every refrigerator is an appliance.

7

Reusing code: why re-invent the wheel?

- **code reuse**: The practice of writing program code once and using it in many contexts.
- We'd like to be able to say the following:

```
// A class to represent secretaries
public class Secretary {
    <copy all the contents from Employee class>

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

8

Inheritance

- **inheritance**: A way to specify a relationship between two classes where one class *inherits* the state and behavior of another.
- The *child* class (also called subclass) inherits from the *parent* class (also called superclass).
- The subclass receives a copy of every field and method from the superclass.

9

Inheritance syntax

- Creating a subclass, general syntax:

```
public class <subclass name> extends <superclass name> {
```

- Example:

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

- By extending `Employee`, each `Secretary` object receives a `getHours`, `getSalary`, `getVacationDays`, and `getVacationForm` method automatically.

10

Improved `Secretary` class

```
// A class to represent secretaries
public class Secretary extends Employee {
    public void takeDictation(String text) {
        System.out.println("Taking dictation of text: "
                           + text);
    }
}
```

11

Writing even more classes

- Write a `Marketer` class that represents marketers who have the same properties as general employees, but instead of making only a paltry $40,000, marketers make $50,000!
- Can we still leverage the `Employee` class or do we have to re-write everything, because one method (`getSalary`) is different?
- If only `Marketer` could write a new version of the `getSalary` method, but inherit everything else…

12

Overriding methods

- **override**: To write a new version of a method in a subclass to replace the superclass's version.
- To override a superclass method, just write a new version of it in the subclass. This will replace the inherited version.

`Marketer` class

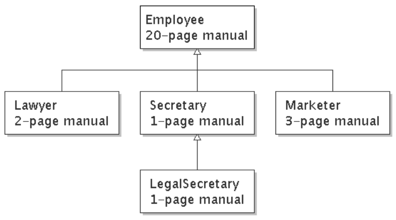
```
// A class to represent marketers
public class Marketer extends Employee {
    public void advertise() {
        System.out.println("Act now while supplies last!");
    }

    public double getSalary() {
        return 50000.0;          // $50,000.00 / year
    }
}
```

Based in reality or too convenient?

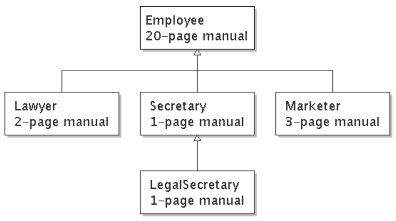
- At many companies, all new employees attend a common orientation to learn general rules (e.g., what forms to fill out when).
- Each person receives a big manual of these rules.
- Each employee also attends a subdivision-specific orientation to learn rules specific to their subdivision (e.g., marketing department).
- Everyone receives a smaller manual of these rules.

Rules, rules, everywhere

- The smaller manual adds some rules and also changes (read: overrides) some rules from the large manual (e.g., "use the pink form instead of the yellow form")



Why bother with separate manuals?



- Why not just have a 22-page manual for lawyers, 21-page manual for secretaries, 23-page manual for marketers, etc…?

Advantages of separate manuals

- maintenance: If a common rule changes, only the common manual needs to be updated.
- locality: A person can look at the manual for lawyers and quickly discover all rules that are specific to lawyers.

Key ideas

- It is useful to be able to specify general rules that will apply to many groups (the 20-page manual).
- It is also useful to specify a smaller set of rules for a particular group, including being able to replace rules from the overall set (e.g., "use the pink form instead of the yellow form").

Exercise: `LegalSecretary`

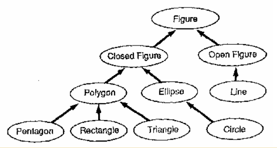
- Write a `LegalSecretary` class that represents legal secretaries—a special type of secretary that can file legal briefs. Legal secretaries also earn more money ($45,000).

Solution: `LegalSecretary`

```
// 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
    }
}
```

Inheritance hierarchies

- Deep hierarchies can be created by multiple levels of subclassing.
- **inheritance hierarchy**: A set of classes connected by is-a relationships that can share common code.



Exercise: `Lawyer`

- Lawyers are employees that know how to sue. They get an extra week of paid vacation (a total of 3) and have to use the pink form when applying for vacation leave. Write the `Lawyer` class.

Solution: `Lawyer`

```
// 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 getVacation() {
        return 15;          // 3 weeks vacation
    }

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

Polymorphism

Readings: 9.2

Motivation

- Given the following:
  ```
  Lawyer laura = new Lawyer();
  Marketer mark = new Marketer();
  ```
- Write a program that will print out the salaries and the color of the vacation form for each employee.

Polymorphism

- A reference variable of type T can refer to an object of any subclass of T.
  ```
  Employee person = new Lawyer();
  ```
- **polymorphism**: The ability for the same code to be used with several different types of objects and behave differently depending on the type of object used.

Properties of polymorphism

```
Employee person = new Lawyer();
System.out.println(person.getSalary());        // 40000.0
System.out.println(person.getVacationForm());  // "pink"
```

- You can call any method from `Employee` on the `person` variable, but not any method specific to `Lawyer` (such as `sue`).
- Once a method is called on the object, it behaves in its normal way (as a `Lawyer`, not as a normal `Employee`).

Polymorphism and parameters

```
public class EmployeeMain {
    public static void main(String[] args) {
        Lawyer laura = new Lawyer();
        Marketer mark = new Marketer();
        printInfo(laura);
        printInfo(mark);
    }

    public static void printInfo(Employee empl) {
        System.out.println("salary = " + empl.getSalary());
        System.out.println("days = " + empl.getVacationDays());
        System.out.println("form = " + empl.getVacationForm());
        System.out.println();
    }
}
```

Output:

```
salary = 40000.0
vacation days = 15
vacation form = pink

salary = 50000.0
vacation days = 10
vacation form = yellow
```

Polymorphism and arrays

```
public class EmployeeMain2 {
    public static void main(String[] args) {
        Employee[] employees = { new Lawyer(), new Secretary(),
                                 new Marketer(), new LegalSecretary() };
        for (int i = 0; i < employees.length; i++) {
            System.out.println("salary = " + employees[i].getSalary());
            System.out.println("vacation days = " +
                               employees[i].getVacationDays());
            System.out.println();
        }
    }
}
```

Output:

```
salary = 40000.0
vacation days = 15

salary = 40000.0
vacation days = 10

salary = 50000.0
vacation days = 10

salary = 45000.0
vacation days = 10
```

Exercise 1

- Assume that the following four classes have been declared:

```
public class Foo {
    public void method1() {
        System.out.println("foo 1");
    }

    public void method2() {
        System.out.println("foo 2");
    }

    public String toString() {
        return "foo";
    }
}

public class Bar extends Foo {
    public void method2() {
        System.out.println("bar 2");
    }
}
```

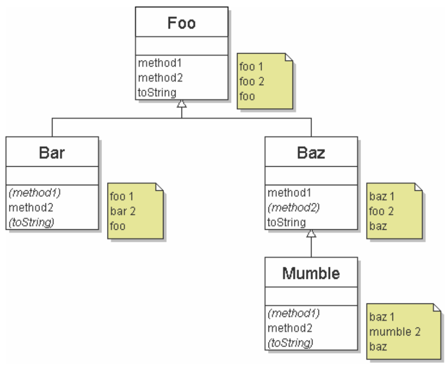
```
public class Baz extends Foo {
    public void method1() {
        System.out.println("baz 1");
    }

    public String toString() {
        return "baz";
    }
}

public class Mumble extends Baz {
    public void method2() {
        System.out.println("mumble 2");
    }
}
```

Exercise 1

- What would be the output of the following client code?

```
Foo[] pity = { new Baz(), new Bar(),
               new Mumble(), new Foo() };

for (int i = 0; i < pity.length; i++) {
    System.out.println(pity[i]);
    pity[i].method1();
    pity[i].method2();
    System.out.println();
}
```

Diagramming polymorphic code



Finding output with tables

method	Foo	Bar	Baz	Mumble
method1	foo 1	*foo 1*	baz 1	*baz 1*
method2	foo 2	bar 2	*foo 2*	mumble 2
toString	foo	*foo*	baz	*baz*

Solution 1

- The code produces the following output:

```
baz
baz 1
foo 2

foo
foo 1
bar 2

baz
baz 1
mumble 2

foo
foo 1
foo 2
```

Exercise 2

- Assume that the following four classes have been declared:

```
public class Lamb extends Ham {
    public void b() {
        System.out.println("Lamb b");
    }
}

public class Ham {
    public void a() {
        System.out.println("Ham a");
    }

    public void b() {
        System.out.println("Ham b");
    }

    public String toString() {
        return "Ham";
    }
}
```

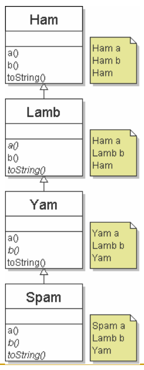
```
public class Spam extends Yam {
    public void a() {
        System.out.println("Spam a");
    }
}

public class Yam extends Lamb {
    public void a() {
        System.out.println("Yam a");
    }

    public String toString() {
        return "Yam";
    }
}
```

Exercise 2

- What would be the output of the following client code?

```
Ham[] food = { new Spam(), new Yam(),
               new Ham(), new Lamb() };

for (int i = 0; i < food.length; i++) {
    System.out.println(food[i]);
    food[i].a();
    food[i].b();
    System.out.println();
}
```

Diagramming polymorphic code



Finding output with tables

method	**Ham**	**Lamb**	**Yam**	**Spam**
a	Ham a	*Ham a*	Yam a	Spam a
b	Ham b	Lamb b	*Lamb b*	*Lamb b*
toString	Ham	*Ham*	Yam	*Yam*

Solution 2

- The code produces the following output:

```
Yam
Spam a
Lamb b

Yam
Yam a
Lamb b

Ham
Ham a
Ham b

Ham
Ham a
Lamb b
```