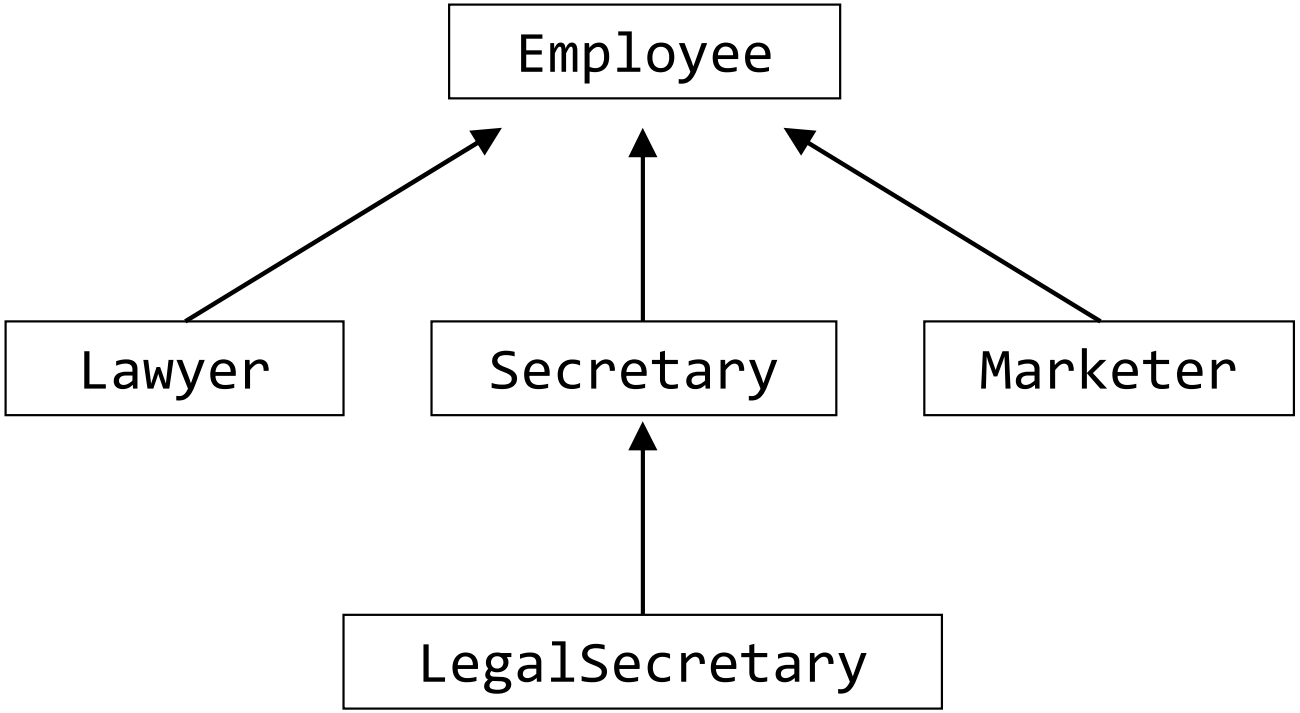
CS 106A, Lecture 19
Inheritance and Polymorphism

reading:

Art & Science of Java, 6.6

Lecture at a glance

- Today we will learn more about **inheritance**.
  - …



System.out.println

```
System.out.println(expression);
System.out.print(expression);
```

- Essentially the `println` / `print` commands we've used before.
  - But you can use them from classes other than your `Program` class.
  - `System.out.println` is standard Java;
    `println` (without System.out) is specific to the Stanford libraries.

Law firm employees

- Recall our **law firm** that employs lawyers, secretaries, legal secretaries, marketers, etc.

- The company has the following employee policies:
  - *hours:* Employees work **40 hours** / week.
  - *salary:* Employees make **$40,000 per year**, except legal secretaries **$45,000**; marketers **$50,000**.
  - *vacation:* Employees have **2 weeks** of paid vacation leave per year, except lawyers get **3 weeks**.
  - *forms:* Employees use a **yellow form** to apply for leave, except lawyers use a **pink form**.

- Also, 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**.

Employee class

```
// A class to represent employees in general.
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
    }
}
```

Extending a class

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

- One class can **extend** another, absorbing its data/behavior.

- Secretary *inherits* getHours/Salary/VacationDays/Form.
  - We add the `takeDictation` method as new Secretary behavior.

Design for change

- 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.

Poor solution

```
public class LegalSecretary extends Secretary {
    public double getSalary() {
        return 45000.0;
    }
    ...
}

public class Marketer extends Employee {
    public double getSalary() {
        return 50000.0;
    }
    ...
}
```

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

The super keyword

- 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;
    }
    ...

public class Lawyer extends Employee {
    public int getVacationDays() {
        return super.getVacationDays() + 5;
    }
    ...
```

Fields and constructors

- Inheritance has some subtleties related to fields.

- Imagine that we want to give employees **more vacation days** the longer they've been with the company.
  - For each year worked, we'll award **+2** additional vacation days.
  - When an Employee object is constructed, we'll pass in the number of years the person has been with the company.
    - The years will be stored as a **field**.
  - This will require us to modify our `Employee` class and add some new state and behavior.
  - Exercise: Make necessary modifications to the `Employee` class.

Modified Employee

```
public class Employee {
    private int years;

    public Employee(int initialYears) {
        years = initialYears;
    }

    public int getHours() {
        return 40;
    }

    public double getSalary() {
        return 50000.0;
    }

    public int getVacationDays() {
        return 10 + 2 * years;
    }

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

A new error

- Now that we've added the constructor to the `Employee` class, our subclasses do not compile. The error:

```
Lawyer.java:2: cannot find symbol
symbol  : constructor Employee()
location: class Employee
public class Lawyer extends Employee {
       ^
```

  - *The short explanation:* Once we write a constructor with parameters in the superclass, we must now write constructors for our employee subclasses as well.

  - *The long explanation:* (next slide)

Long explanation

- Constructors are not inherited.
  - Subclasses don't inherit the `Employee(int)` constructor.
  - Subclasses receive a default constructor that contains:

```
// implicit constructor (if you don't write one)
public Lawyer() {
    super();          // calls Employee() constructor
}
```

- But our `Employee(int)` replaces the default `Employee()`.
  - The subclasses' default constructors are now trying to call a non-existent default `Employee` constructor.

Call superclass c'tor

```
super(parameters);
```

– Example:

```
public class Lawyer extends Employee {
    public Lawyer(int years) {
        super(years);  // call Employee constructor
    }
    ...
}
```

– The `super` call must be the *first* statement in the constructor.

– Exercise: Modify the `Secretary` subclass.
  - Secretaries' years of employment are not tracked.
  - They do not earn extra vacation for years worked.

Modified Secretary

```
// A class to represent secretaries.
public class Secretary extends Employee {
    public Secretary() {
        super(0);
    }

    public String takeDictation(String text) {
        return "Taking dictation of text: " + text;
    }
}
```

- Since `Secretary` doesn't require any parameters to its constructor, `LegalSecretary` compiles without a constructor.
  - Its default constructor calls the `Secretary()` constructor.

Inheritance and fields

- Try to give lawyers $5000 for each year at the company:

```
public class Lawyer extends Employee {
    ...
    public double getSalary() {
        return super.getSalary() + 5000 * years;
    }
    ...
}
```

- Does not work; the error is the following:

```
Lawyer.java:7: years has private access in Employee
        return super.getSalary() + 5000 * years;
                                          ^
```

- Private fields cannot be directly accessed from subclasses.
  - One reason: So that subclassing can't break encapsulation.
  - How can we get around this limitation?

Accessors

- Add an accessor for any field needed by the subclass.

```
public class Employee {
    private int years;

    public Employee(int initialYears) {
        years = initialYears;
    }

    public int getYears() {
        return years;
    }
    ...
}

public class Lawyer extends Employee {
    public double getSalary() {
        return super.getSalary() + 5000 * getYears();
    }
    ...
}
```

Revisiting Secretary

- The `Secretary` class currently has a poor solution.
  - We set all Secretaries to 0 years because they do not get a vacation bonus for their service.
  - If we call `getYears` on a `Secretary` object, we'll always get 0.
  - This isn't a good solution; it's not really true that they worked 0 years.
    - What if we need to know how many years the secretary worked?
    - What if we want to give a reward based on years of service?
    - Etc.

- Let's redesign our `Employee` class to allow for a better solution.

Improved Employee

```
// A class to represent employees in general.
public class Employee {
    private int years;

    public Employee(int initialYears) {
        years = initialYears;
    }

    public int getVacationDays() {
        return 10 + getSeniorityBonus();
    }

    // vacation days given for each year in the company
    public int getSeniorityBonus() {
        return 2 * years;
    }
    ...
}
```

- Separate the 10 vacation days from the seniority vacation bonus.
  - How does this help us improve the `Secretary`?

Improved Secretary

```
// A class to represent secretaries.
public class Secretary extends Employee {
    public Secretary(int years) {
        super(years);
    }

    // Secretaries don't get a bonus for their years of service.
    public int getSeniorityBonus() {
        return 0;
    }

    ...
}
```

- `Secretary` can selectively override `getSeniorityBonus`; when `getVacationDays` runs, it will use the new version.
  - Choosing a method at runtime is called *dynamic binding*.

Polymorphism

Polymorphism

- **polymorphism**: Ability for the same code to be used with different types of objects and behave differently with each.

- Examples:
  - `println` can accept any type of parameter and print it.
  - A `GraphicsProgram` can add any type of graphical object to itself.

Poly. and variables

- A variable of type *T* can hold an object of any subclass of *T*.

```
Employee ed = new Lawyer();
```

  - You can call any methods from the `Employee` class on `ed`.

- When a method is called on `ed`, it behaves as a `Lawyer`.

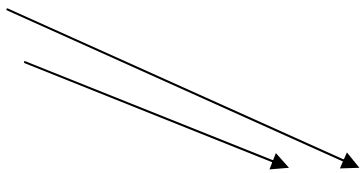
```
println(ed.getSalary());           // 50000.0
println(ed.getVacationForm());     // pink
```

Polym. and parameters

- You can pass any subtype of a parameter's type.

```
public class EmployeeMain extends ConsoleProgram {
    public void run() {
        Lawyer lisa = new Lawyer();
        Secretary steve = new Secretary();
        printInfo(lisa);
        printInfo(steve);
    }

    public void printInfo(Employee empl) {
        println("salary: " + empl.getSalary());
        println("v.days: " + empl.getVacationDays());
        println("v.form: " + empl.getVacationForm());
        println();
    }
}
```



OUTPUT:

```
salary: 50000.0     salary: 50000.0
v.days: 15          v.days: 10
v.form: pink        v.form: yellow
```

Polymorphic arrays

- Arrays of superclass type can store any subtype as elements.

```
public class EmployeeMain2 extends ConsoleProgram {
    public void run() {
        Employee[] e = { new Lawyer(),   new Secretary(),
                         new Marketer(), new LegalSecretary() };

        for (int i = 0; i < e.length; i++) {
            println(i + " salary: " + e[i].getSalary());
            println(i + " v.days: " + e[i].getVacationDays());
            println();
        }
    }
}
```

Output:

```
0 salary: 50000.0
0 v.days: 15

1 salary: 50000.0
1 v.days: 10

2 salary: 60000.0
2 v.days: 10

3 salary: 55000.0
3 v.days: 10
```

Polymorphism mystery

- **Q:** What is the output from the following code?

```
public class Pikachu {
    public void method1() { System.out.println("P1"); }
    public void method2() { System.out.println("P2"); }
}

public class Squirtle extends Pikachu {
    public void method2() { System.out.println("S2"); }
}

public class Charizard extends Squirtle {
    public void method1() {
        method2();
        System.out.println("C1");
    }
}

Pikachu pika = new Charizard();
pika.method1();
```

A. P1 **B.** S1 **C.** C2 / C1 **D.** S2 / C1 **E.** P2 / C1

Polymorphism mystery

- Suppose 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";
    }
}

...
```

Polymorphism mystery

```
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");
    }
}
```

Polymorphism mystery

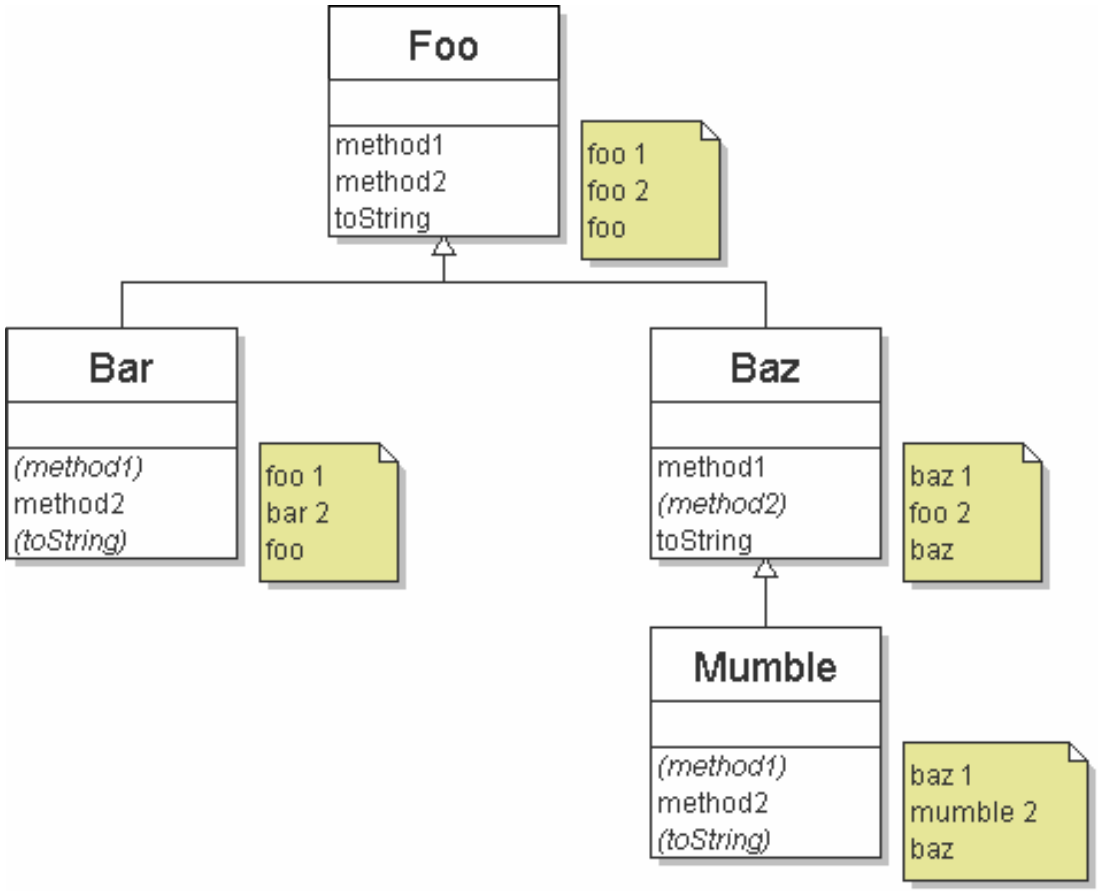
- What would be the output of the following client code?

```
Foo[] pity = new Foo[4]
pity[0] = new Baz();
pity[1] = new Bar();
pity[2] = new Mumble();
pity[3] = new Foo();

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

Class diagram

- Add classes from top (superclass) to bottom (subclass).
- Include all inherited methods.



Output 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*

Mystery solution

```
Foo[] pity = {new Baz(), new Bar(), new Mumble(), new Foo()};
for (int i = 0; i < pity.length; i++) {
    println(pity[i]);
    pity[i].method1();  pity[i].method2();
    println();
}
```

- Output:

```
baz
baz 1
foo 2

foo
foo 1
bar 2

baz
baz 1
mumble 2

foo
foo 1
foo 2
```

Polymorphism mystery 2

- The class order is jumbled; some methods call others (tricky!).

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

public class Ham {
    public void a() {
        System.out.print("Ham a   ");
        b();
    }
    public void b() {
        System.out.print("Ham b   ");
    }
    public String toString() {
        return "Ham";
    }
}
```

Polymorphism mystery 2

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

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

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

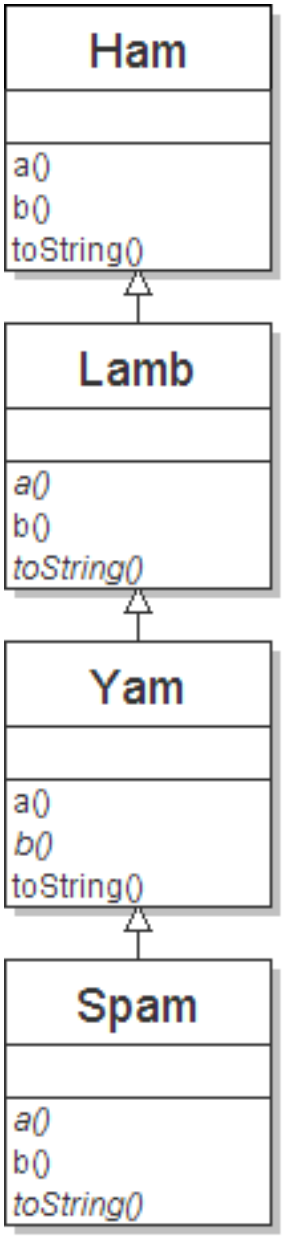
Polymorphism mystery 2

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

```
Ham[] food = {
    new Lamb(),    // 0
    new Ham(),     // 1
    new Spam(),    // 2
    new Yam()      // 3
};

for (int i = 0; i < food.length; i++) {
    println(food[i]);
    food[i].a();
    println();     // to end the line of output
    food[i].b();
    println();     // to end the line of output
    println();
}
```

Class diagram



Polymorphism question

- **Q:** What is Lamb's output from calling method a ?

```
public class Ham {
    public void a() {
        System.out.print("Ham a   ");
        b();
    }
    public void b() {
        System.out.print("Ham b   ");
    }
}

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

- **A.** Ham a / Ham b
- **B.** Ham a / Lamb b
- **C.** compiler error, because class Lamb does not have a method a
- **D.** infinite loop / infinite output
- **E.** none of the above

Polymorphism at work

```
// Lamb inherits a from Ham. a calls b. But Lamb overrides b...
public class Ham {
    public void a() {
        System.out.print("Ham a   ");
        b();
    }
    public void b() {
        System.out.print("Ham b   ");
    }
    public String toString() {
        return "Ham";
    }
}

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

- Lamb's output from calling a:

```
Ham a   Lamb b
```

Output table

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

Mystery 2 solution

```
Ham[] food = {new Lamb(), new Ham(), new Spam(), new Yam()};
for (int i = 0; i < food.length; i++) {
    println(food[i]);
    food[i].a();   food[i].b();   println();
}
```

- Output:

```
Ham
Ham a   Lamb b
Lamb b

Ham
Ham a   Ham b
Ham b

Yam
Yam a   Ham a   Spam b
Spam b

Yam
Yam a   Ham a   Lamb b
Lamb b
```

Overflow (extra) slides

Inheritance question

- **Q:** Which of the following is a **good** usage of inheritance?

A.
```
public class Hexagon extends Square { ...
```
B.
```
public class Melody extends Note { ...
```
C.
```
public class Car extends Minivan { ...
```
D.
```
public class ShoppingCart extends GroceryItem {
```
E.
```
public class Stanford extends Berkeley { ...
```

(In which case is the subclass a natural subcategory of the superclass?)

Protected fields

```
protected type fieldName;          // field
protected type methodName(type name, ..., type name) {
    statements;                    // method
}
```

- a **protected field** or **method** can be seen/called only by:
  - the class itself, and its subclasses
  - also by other classes in the same "package"
    - useful for allowing selective access to inner class implementation

```
public class Employee {
    protected double salary;
    ...
}
```