CS 106A, Lecture 26
Polymorphism; Interfaces

reading:

Art & Science of Java, 6.6

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

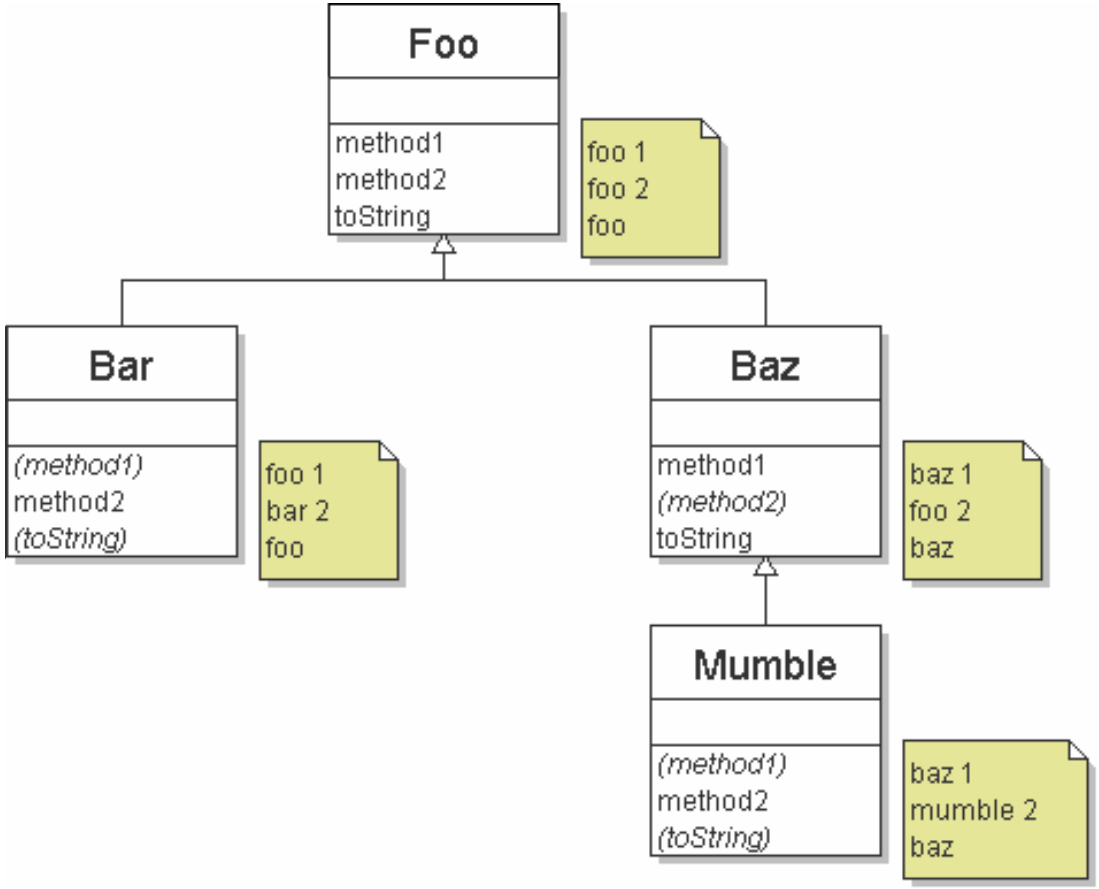
- What would be the output of the following client code?

```
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";
    }
}
```

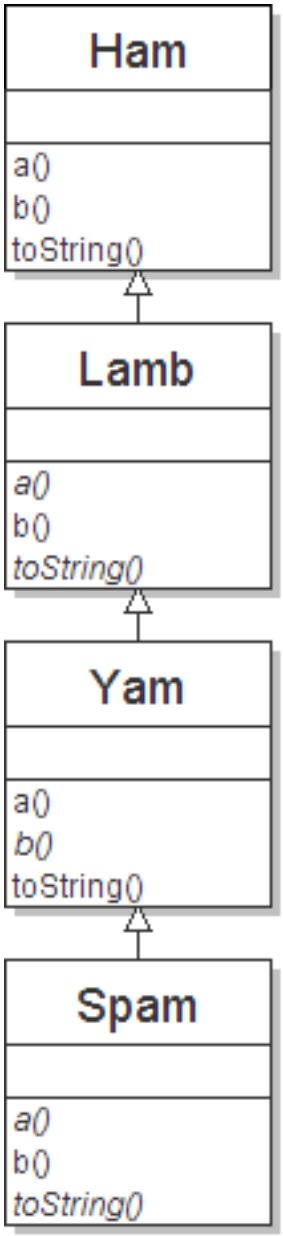
Polymorphism mystery 2

- 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
```

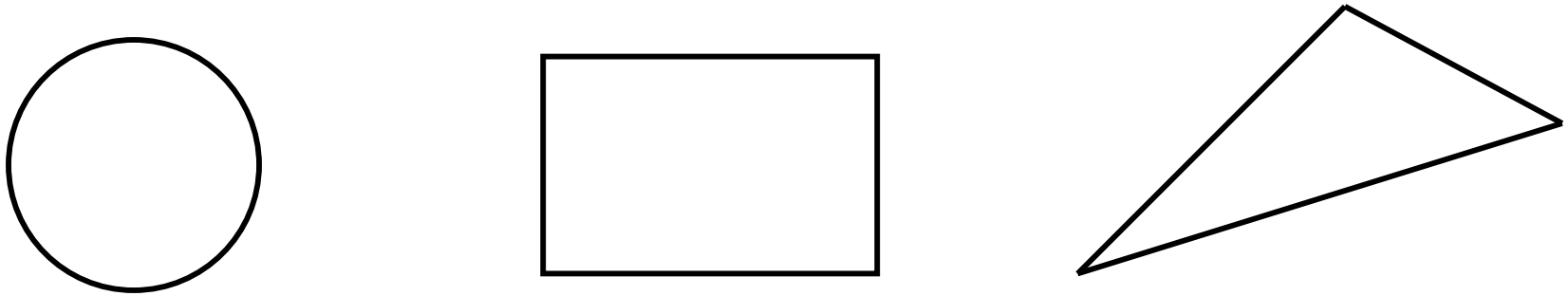
Interfaces

Shapes example

- Consider the task of writing classes to represent 2D shapes such as `Circle`, `Rectangle`, and `Triangle`.

- Certain attributes or operations are common to all shapes:
  - perimeter: distance around the outside of the shape
  - area: amount of 2D space occupied by the shape

  - Every shape has these, but each computes them differently.



Interfaces

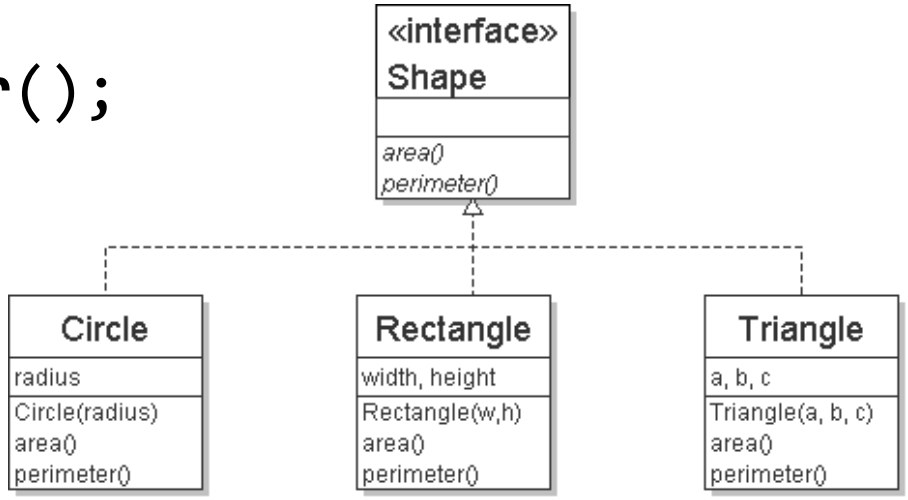
- **interface**: A list of methods that a class can promise to implement.
  - Inheritance gives you an is-a relationship *and* code sharing.
    - A `Lawyer` can be treated as an `Employee` and inherits its code.
  - Interfaces give you an is-a relationship *without* code sharing.
    - A `Rectangle` object can be treated as a Shape but inherits no code.
  - Analogous to non-programming idea of roles or certifications:
    - "I'm certified as a CPA accountant.
      This assures you I know how to do taxes, audits, and consulting."
    - "I'm 'certified' as a Shape, because I implement the Shape interface.
      This assures you I know how to compute my area and perimeter."

Interface syntax

```
public interface Name {
    public type name(type name, ..., type name);
    public type name(type name, ..., type name);
    ...
    public type name(type name, ..., type name);
}
```

Example:

```
public interface Shape {
    public double area();
    public double perimeter();
}
```



Implementing an interface

```
public class Name implements Interface {
    ...
}
```

– The class promises to contain each method in that interface.
  (Otherwise it will fail to compile.)

Example:

```
public class Rectangle implements Shape {
    ...
    public double area() { ... }
    public double perimeter() { ... }
}
```

Interfaces + polymorphism

- Interfaces benefit the *client code* author the most.
  - they allow **polymorphism**
    (the same code can work with different types of objects)

```
public void printInfo(Shape s) {
    println("The shape: " + s);
    println("area : " + s.area());
    println("perim: " + s.perimeter());
    println();
}
...
Circle circ = new Circle(12.0);
Triangle tri = new Triangle(5, 12, 13);
printInfo(circ);
printInfo(tri);
```

Interface of constants

- A methodless interface can store shared **constants**.
  - Files can 'implement' the interface to refer to those constants easily.

```
public interface FinanceConstants {
    public static final double INTEREST = 0.025;
    public static final String NAME = "Wells GoFar";
}
...
public class BankAccount implements FinanceConstants {
    public void accrueInterest() {
        balance += INTEREST * balance;
    }

    public String toString() {
        return NAME + ": " + id + " $" + balance;
    }
}
```

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;
    ...
}
```