Polymorphism

Coding with polymorphism

- 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`.

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


Polymorphism and parameters

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

```
public class EmployeeMain {
    public static void main(String[] args) {
        Lawyer lisa = new Lawyer();
        Secretary steve = new Secretary();
        printInfo(lisa);
        printInfo(steve);
    }

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

OUTPUT:

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


Polymorphism and arrays

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

```
public class EmployeeMain2 {
    public static void main(String[] args) {
        Employee[] e = { new Lawyer(),   new Secretary(),
                         new Marketer(), new LegalSecretary() };

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

Output:

```
salary: 50000.0
v.days: 15

salary: 50000.0
v.days: 10

salary: 60000.0
v.days: 10

salary: 55000.0
v.days: 10
```

Polymorphism problems

- 4-5 classes with inheritance relationships are shown.
- A client program calls methods on objects of each class.
- You must read the code and determine the client's output.
- We always put such a question on our midterms!

A polymorphism problem

```
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");}
}
```

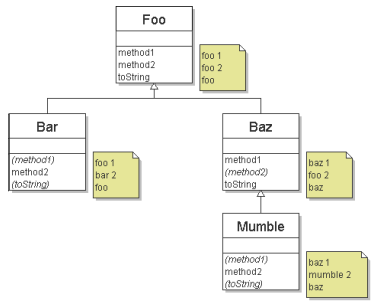

A polymorphism problem

- 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 the classes

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



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*

Polymorphism answer

```
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();
}
```

- Output:

```
baz
baz 1
foo 2

foo
foo 1
bar 2

baz
baz 1
mumble 2

foo
foo 1
foo 2
```


Casting references

- A variable can only call that type's methods, not a subtype's.

```
Employee ed = new Lawyer();
int hours = ed.getHours();  // ok; it's in Employee
ed.sue();                   // compiler error
```

  - The compiler's reasoning is, variable `ed` could store any kind of employee, and not all kinds know how to `sue` .

- To use `Lawyer` methods on `ed`, we can type-cast it.

```
Lawyer theRealEd = (Lawyer) ed;
theRealEd.sue();                  // ok
((Lawyer) ed).sue();              // shorter version
```


More about casting

- The code crashes if you cast an object too far down the tree.

```
Employee eric = new Secretary();
((Secretary) eric).takeDictation("hi");     // ok
((LegalSecretary) eric).fileLegalBriefs();  // exception

// (Secretary object doesn't know how to file briefs)
```

- You can cast only up and down the tree, not sideways.

```
Lawyer linda = new Lawyer();
((Secretary) linda).takeDictation("hi");    // error
```

- Casting doesn't actually change the object's behavior.
  It just gets the code to compile/run.

```
((Employee) linda).getVacationForm()    // pink (Lawyer's)
```

Run-Time Type Information

- You can check the legality of a cast before you do it:

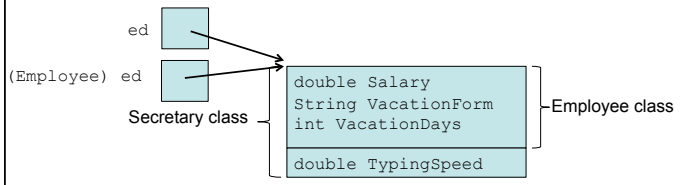
```
Lawyer linda = new Lawyer();
if (linda instanceof Secretary) {
  ((Secretary) linda).takeDictation("hi");
}
```

- It's generally best to avoid casts as much as possible.
- In many cases, reliance on instanceof can be replaced by proper use of polymorphism.

31

How Does Inheritance Work?

- When class B extends class A, the fields in class A are a subset:

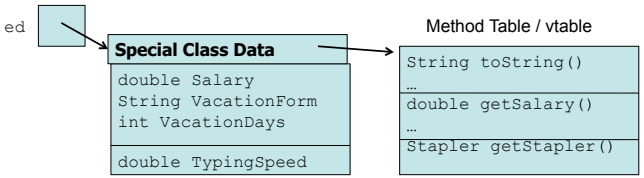


- Therefore, every Secretary can be treated as an Employee by only looking at the first three fields

32

How Does Polymorphism Work?

- A subclass has all the methods of the superclass:



- Overriding a method changes the entry in the method table for this class.
- When we cast a Secretary as an Employee, the method table is unchanged: toString() and getSalary() still point to the Secretary-specific code

33

Real Interview Question

```
/* What does the following program print? */
public class Test {
  public boolean equals( Test other ) {
    System.out.println( "Inside of Test.equals" );
    return false;
  }

  public static void main( String [] args ) {
    Object t1 = new Test();
    Object t2 = new Test();
    Test t3 = new Test();
    Object o1 = new Object();

    System.out.println("1");  t1.equals(t2);
    System.out.println("2");  t1.equals(t3);
    System.out.println("3");  t3.equals(o1);
    System.out.println("4");  t3.equals(t3);
    System.out.println("5");  t3.equals(t2);
  }
}
```

34

Polymorphism vs. Overloading

- **Overloading:** Two or more methods with different parameters and the same name. Which method to call is chosen statically (at compile time).

```
public void add(int value);
public void add(int value, int index);
public void add(ArrayIntList list);
```

- **Polymorphism:** Related classes define a method with the same name and parameters. Which method to call is chosen dynamically (at runtime).

35