Unit 9: Inheritance
Polymorphism

Adapted from:

1) Building Java Programs: A Back to Basics Approach
by Stuart Reges and Marty Stepp

2) Runestone CSAwesome Curriculum

Textbook Reference

Online Textbook Think Java - 2nd Edition by Allen Downey and Chris Mayfield

For this lecture use Chapter 17

Polymorphism

polymorphism: Ability for an object or method to take on many forms("poly" = many, "morphism" = forms)

- method overriding(**run-time polymorphism**)
- method overloading(**compile-time polymorphism**)

Employee and Lawyer

Suppose we have the following classes. We'll use these classes for the next few examples in the following slides.

```
public class Employee{
  public double getSalary(){return 50000.0}
  public String getVacationForms(){return "pink";}
}
public class Lawyer extends Employee{
  public double getSalary(){return 60000.0}
  public String getVacationForms(){return "yellow";}
  public void sue(){System.out.println("I will see you in court!");}
}
public class Secretary extends Employee{
  public void takeDictation(String str){…}
}
public class LegalSecretary extends Secretary{
  public void filLegalBriefs(){…}
}
```

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` even though ed is an Employee reference.

```
System.out.println(ed.getSalary());// Lawyer's salary 60000.0
System.out.println(ed.getVacationForm()); // Lawyer's(yellow)
```

Method overriding is also known as **run-time polymorphism** or **dynamic binding**. Java selects the correct method at **run-time**.

Polymorphism and parameters

polymorphism: Ability for the same code to be used with different types of objects and behave differently with each. The printInfo method below will behave different depending on the type of object in the parameter.

```
public class EmployeeMain {
    public static void main(String[] args) {
        Employee steve = new Employee();
        Lawyer sarah = new Lawyer();
        printInfo(steve);
        printInfo(sarah);
    }
    public static void printInfo(Employee empl) {
        System.out.println("salary: " + empl.getSalary());
        System.out.println("v.form: " + empl.getVacationForm());
        System.out.println();
    }
}
```

OUTPUT:

```
salary: 50000.0
v.form: pink
salary: 60000.0
v.form: yellow
```

If not for polymorphism, we need to write a **different** printInfo, one for each subclass(Employee, Secretary, Lawyer)!
This code will remain the same regardless of how many subclasses of Employee we add later in our code.

Polymorphism and arrays

Arrays/Arraylists of superclass types can store any subtype as elements.

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

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

Output:

```
salary: 50000.0
forms: pink
salary: 60000.0
forms: yellow
salary: 50000.0
forms: pink
salary: 50000.0
forms: pink
```

Polymorphism allows us to store ALL employees regardless of their job positions in the same array and our code can behave correctly depending on the object type.

Otherwise, we need to create a different array for each position(i.e, one for Employees, one for Lawyers, etc…)

Casting references

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

```
Employee ed = new Lawyer();
int hours = ed.getSalary();  // 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,two sets of()
```

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();  // Class cast
                                        //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)
```

Review Example

```
Employee one = new Secretary(); //upcasts, always ok

one.getSalary();
//calls Secretary’s version

one.takeDictation("hi");
//error, even though one holds a
//Secretary object, one is an Employee reference
//and can only call Employee’s methods.

//here’s how to fix the above error.
((Secretary) one).takeDictation("hi");//cast then call
//can’t cast too far down the tree
((LegalSecretary) one).fileLegalBriefs(); //error
```

Review Example 2

```
LegalSecretary two = new LegalSecretary();

((Secretary) two).getSalary();
//upcast doesn't change behavior.
//still LegalSecretary's version

((Employee) two).getSalary();
//still LegalSecretary's version

//can't cast sideways
((Lawyer) two).sue(); //error
```

Overloading

A class can have **many forms** of the same method. Methods are said to be **overloaded** when there are multiple methods with the same name but a different signature in the **same class**.

The methods are distinguished by:

1. Number of parameters
2. Type of the parameters
3. Order of the parameters

Method overloading is also known as **compile-time polymorphism** or **static binding**. Java selects the correct method at compile-time.

Number of Parameters

Methods with the same name can be distinguished by the **number** of parameters.

```
public class Overload{

public void method1(int c)
{…}

public void method1(int c, double d)
{…}

}
```

Type of Parameters

Methods with the same name can be distinguished by the **type** of the parameters.

```
public class Overload{

public void method1(int c)
{…}

public void method1(double c)
{…}

}
```

Order of Parameters

Methods with the same name can be distinguished by the **order** of the parameters.

```
public class Overload{

public void method1(int c, double d)
{…}

public void method1(double d, int c)
{…}

}
```

Invalid Overloading

```
Case 1:
public void method1(int c, double d)
{…}

public void method1(int e, double f)
{…}
```

Compile error. Same number, data types and sequence. Methods cannot be overloaded with just different variable names.

One reason is because a call like this would be ambiguous:

```
method1(3, 4.1);  // which method1?
```

Invalid Overloading

```
Case 2:
public void method1(int c, double d)
{…}

public boolean method1(int e, double f)
{…}
```

Compile error. Same number, data types and sequence. Even though the return type is different, this is not valid.

One reason is because a call like this would be ambiguous:

```
method1(3, 4.1);   // which method1?, both work.
```

Ambiguous Call

The following correctly implements method overloading. However, it is possible that a method call is ambiguous.

```
public static void method1(int a, double b)
{…}
public static void method1(double a, int b)
{…}
public static void main(String[] args)
{
    method1(3, 4.0); // ok, calls the first one above
    method1(3.3, 4); // ok, calls the second one above
    method1(3, 4); //error, ambiguous call, which one?
}
```

Compile-time vs Runtime

An error is a **compile-time error** if it happens when the program compiles.

- All method overloading errors are compile-time errors.

An error is a **runtime error** if it happens when the program runs.

- Casting too far down, sideways or calling methods not in reference class are run-time errors.

A runtime error compiles without errors.

Compile-time vs Runtime

```
Employee Sean = new Secretary();

Sean.takeDictation(“hi”);
//compile-time error, no such method in Employee

Sean.fileLegalBriefs();
// compile-time error, no such method in Employee

Sean.getSalary();
//ok
```

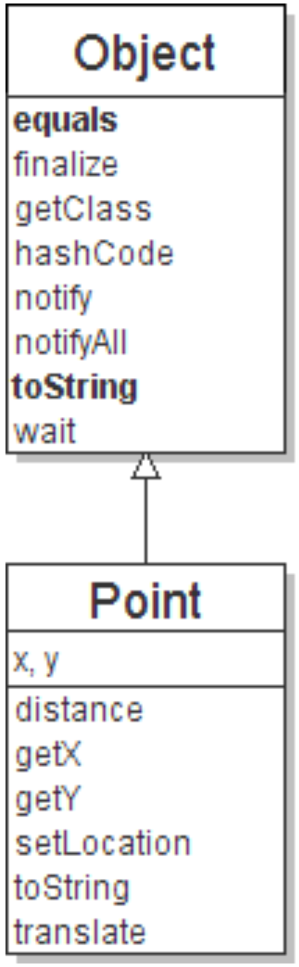
Compile-time vs Runtime

```
((LegalSecretary) Sean).sue();
//compile-time error,sue() isn’t in LegalSecretary

((LegalSecretary) Sean).fileLegalBriefs();
//runtime error, cast too far down the tree
//the program compiles without errors.

((Lawyer) Sean).sue();
//runtime error, horizontal casting not allowed;
//the program compiles without errors.
```

The Cosmic SuperClass `Object`

All types of objects have a superclass named `Object`.

- Every class implicitly extends `Object`



The `Object` class defines several methods:

- `public String toString()`
  Returns a text representation of the object, often so that it can be printed. We have seen this in Unit 5.

- `public boolean equals(Object other)`
  Compare the object to any other for equality. Returns `true` if the objects have equal state.

`Object` variables

You can store any object in a variable of type `Object`.

```
Object o1 = new Point(5, -3);
Object o2 = "hello there";
Object o3 = new Scanner(System.in);
```

An `Object` variable only knows how to do general things.

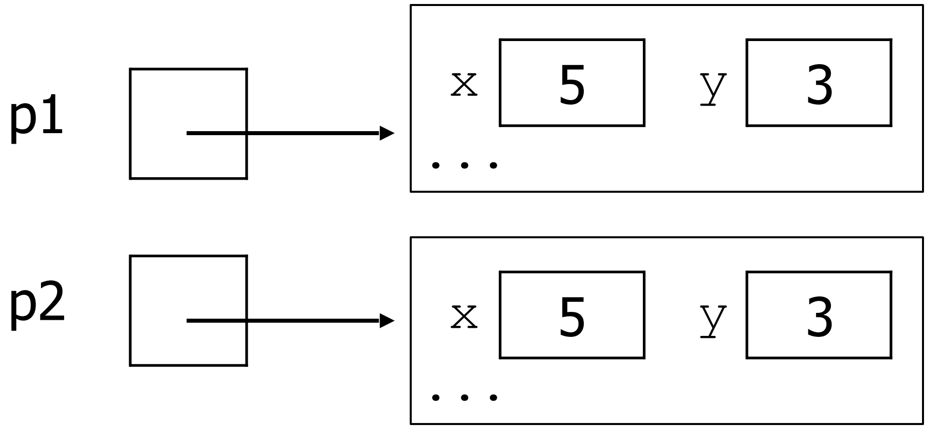
```
String s = o1.toString(); // ok (memory address)
int len = o2.length();    // compile-time error
String line = o3.nextLine(); // compile-time error
```

Recall: comparing objects

The `==` operator does not work well with objects.

- `==` compares references to objects, not their state. It only produces `true` when you compare an object to itself.

```
Point p1 = new Point(5, 3);
Point p2 = new Point(5, 3);
if (p1 == p2) {    // false
    System.out.println("equal");
}
```



The `equals` method

The `equals` method compares the state of objects.

```
    if (str1.equals(str2)) {
        System.out.println("the strings are 
  equal");
    }
```

But if you write a class, its `equals` method behaves like `==`

```
    if (p1.equals(p2)) {    // false :-(
        System.out.println("equal");
    }
```

- This is the behavior we inherit from class `Object`.
- Java doesn't understand how to compare `Point`s by default.

`equals` method

We can change this behavior by writing an `equals` method that **overrides** the one inherited from `Object`.

- **Note the method header including the parameter Object o below.**
- The method should compare the state of the two objects and return `true` if they have the same x/y position.

```
  public boolean equals(Object o) {
          Point other = (Point) o;
      return (x == other.x && y == other.y)
 }
```

An Implementation of Point

Here's the Point class with both `toString` and `equals` overriden.

```
public class Point {
    private int x;
    private int y;
    public Point(int newX, int newY){
        x = newX;
        y = newY;
    }
    public boolean equals(Object o) {
                Point other = (Point) o;
            return (x == other.x && y == other.y);
    }
    public String toString(){
                return "(" + x + ", " + y + ")";
    }

}
```

Main

```
public class Main {
  public static void main(String[] args){
    Point x = new Point(2, -5);
    Point y = new Point(2, -5);
    Point z = new Point(3, 8);
    Point w = z;
    System.out.println(x == y); // false
    System.out.println(z == w); // true
    System.out.println(x.equals(y)); // true
    System.out.println(x.equals(w)); // false
    System.out.println(x); //(2, -5)
    // call toString() implicitly
  }
}
```

Lab 1

Modify the previous lab(Inheritance Lecture Lab 1) which contains Student and GradStudent classes.

The Student class now has an additional private variable double gpa. Modify the constructor accordingly.

Add getGpa() and setGpa() methods.

Add a isGraduating() method which returns whether the Student is graduating. A student graduates if his gpa is at least a 2.0.

Lab 1

Modify the GradStudent class. Override the isGraduating method from Student. A graduate student graduates if his gpa is at least a 3.0.

Write the driver class. Create an array containing at least one Student object and one GradStudent object. Use a loop to print out welcome messages and whether they graduate.

Notice polymorphism at work.

References

1) CPJava Website
2) CPJava Google Classroom
3) CPJava trinket.io Classroom
4) Runestone CSAwesome BUSHSCHOOL_CPJAVA Course
5) Online Textbook Think Java - 2nd Edition by Allen Downey and Chris Mayfield
6) Building Java Programs: A Back to Basics Approach by Stuart Reges and Marty Stepp