CS161
Intro to Computer Science I
Dynamic Binding, The Object Class, and Polymorphism
Chap. 9.2 – 9.3

Dynamic Binding

```java
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;
    }
    ...
}

• Notice: getVacationDays() uses the getSeniorityBonus();
```

Dynamic Binding

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

• What makes this dynamic binding not overriding?
  – getVacationDays() dynamically binds the method based on type
```
The Object Class

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

Object Objects

- 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
  String line = o3.nextLine();  // error

- You can write methods that accept an Object parameter.
  public void checkForNull(Object o) {
    if (o == null) {
      throw new IllegalArgumentException();
    }
  }

Recall: comparing objects

- The == operator does not work well with objects.
  == compares references to objects, not their state.

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.
  ```java
  if (str1.equals(str2)) {
      System.out.println("the strings are equal");
  }
  ```
- But if you write a class, its equals method behaves like ==
  ```java
  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 Points by default.

Our own equals method

- override the default behavior from class Object.
  - The method should compare the state of the two objects and return true if they have the same x/y position.
- A flawed implementation:
  ```java
  public boolean equals(Point other) {
      if (x == other.x && y == other.y) {
          return true;
      } else {
          return false;
      }
  }
  ```
  - We think this is better:
    ```java
    return x == other.x && y == other.y;
    ```

Flaws in our method

- The parameter to equals must be of type Object.
  - Object is a general type that can match any object.
- Having an Object parameter means any object can be passed.
  ```java
  // this should be allowed
  Point p = new Point(7, 2);
  if (p.equals("hello")) { // false
  ```
  - equals should always return false if a non-Point is passed.
  - If we don't know what type it is, how can we compare it?
equals and Object

- Another flawed equals implementation:
  ```java
class Point {
  private int x, y;

  public boolean equals(Object o) {
    return x == o.x && y == o.y;
  }
}
```
- It does not compile:
  ```java
  Point p = new Point(7, 2);
  if (p.equals("hello") { // should be false
    System.out.println("Hello");
  }...
  ```
  - The compiler is saying, "o could be any object. Not every object has an x field."

Type-casting objects

- Solution: Type-cast the object parameter to a Point.
  ```java
  public boolean equals(Object o) {
    Point other = (Point) o;
    return x == other.x && y == other.y;
  }
  ```
- Casting objects is different than casting primitives.
  - Doesn't actually change the object that was passed.
  - Tells the compiler to assume that o refers to a Point object.

Comparing different types

- Current our method crashes on the above code:
  ```java
  Exception in thread "main"
  java.lang.ClassCastException: java.lang.String
  at Point.equals(Point.java:25)
  at PointMain.main(PointMain.java:25)
  ```
- The culprit is the line with the type-cast:
  ```java
  public boolean equals(Object o) {
    Point other = (Point) o;
  }
  ```
The `instanceof` keyword

- Error Handling
- Ask if a variable refers to an object of a given type.
  - Used as a boolean test.

```java
if (variable instanceof type) {
    statement(s);
}
```

Final `equals` method

```java
// Returns whether o refers to a Point object
// with same (x, y) coordinates as this Point.
public boolean equals(Object o) {
    if (o instanceof Point) {
        // o is a Point; cast and compare it
        Point other = (Point) o;
        return x == other.x && y == other.y;
    } else {
        // o is not a Point; cannot be equal
        return false;
    }
}
```

Polymorphism

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

  - `System.out.println` can print any type of object.
    - Each one displays in its own way on the console.
Polymorphism and parameters

• You can pass any subtype of a parameter’s type.

```java
class Employee {
    public String getSalary();
    public int getVacationDays();
    public String getVacationForm();
}
class Secretary extends Employee {
    public String takeDictation();
}
```

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

```java
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());
}
```

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

Coding with polymorphism

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

```java
Employee ed = new Secretary();
```

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

• When a method is called with ed, it behaves as a Secretary.

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

• Can we call methods from Secretary?

```java
System.out.println(ed.takeDictation()); // not allowed
```

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.
A polymorphism problem

• Suppose that the following four classes have been declared:

```java
public class Foo {
    public void method1() {
        System.out.println("foo 1");
    }
    public void method2() {
        System.out.println("foo 2");
    }
    public String toString() {
        return "foo";
    }
}
```

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

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

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

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

```java
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)

Finding output with tables

<table>
<thead>
<tr>
<th>method</th>
<th>Foo</th>
<th>Bar</th>
<th>Baz</th>
<th>Mumble</th>
</tr>
</thead>
<tbody>
<tr>
<td>method1</td>
<td>foo 1</td>
<td>foo 1</td>
<td>baz 1</td>
<td>baz 1</td>
</tr>
<tr>
<td>method2</td>
<td>foo 2</td>
<td>bar 2</td>
<td>foo 2</td>
<td>mumble 2</td>
</tr>
<tr>
<td>toString</td>
<td>foo</td>
<td>foo</td>
<td>baz</td>
<td>baz</td>
</tr>
</tbody>
</table>