CS 142

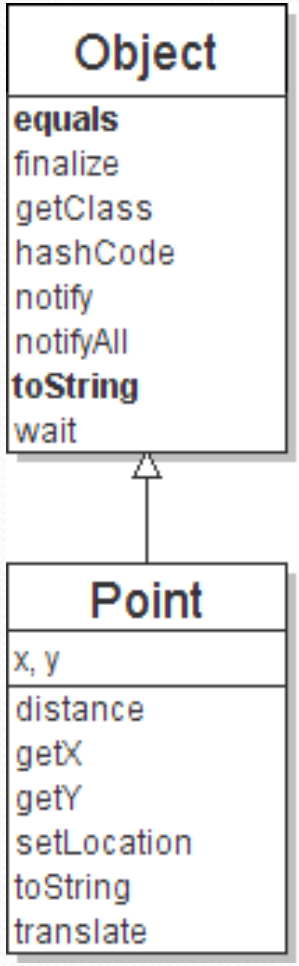
Lecture 32: `Object`; `equals`; generics

Thanks to Marty Stepp and Stuart Reges for parts of these slides

The class Object

- The class `Object` forms the root of the overall inheritance tree of all Java classes.
  - Every class is implicitly a subclass of `Object`

- The `Object` class defines several methods that become part of every class you write. For example:
  - `public String toString()`
    Returns a text representation of the object, usually so that it can be printed.



Object methods

method	description
`protected Object clone()`	creates a copy of the object
`public boolean equals(Object o)`	returns whether two objects have the same state
`protected void finalize()`	called during garbage collection
`public Class<?> getClass()`	info about the object's type
`public int hashCode()`	a code suitable for putting this object into a hash collection
`public String toString()`	text representation of the object

`public void notify()`
`public void notifyAll()`
`public void wait()`
`public void wait(...)`
	methods related to concurrency and locking (seen later)

- What does this list of methods tell you about Java's design?

Common properties of objects

- When Sun designed Java, they felt that *every* object (including arrays) should be able to:
  - be compared to other objects (equals)
  - be printed on the console or converted into a string (toString)
  - ask questions at runtime about what type/class it is (getClass)
  - be created (constructors), copied (clone), and destroyed (finalize)
  - be used in hash-based collections (hashCode)
  - perform multi-threaded synchronization and locking (notify/wait)

- This powerful and broad set of capabilities helped Java's adoption as an object-oriented language.

Using the Object class

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

```
Object o1 = new Point(5, -3);
Object o2 = "hello there";
```

- You can write methods that accept an `Object` parameter.

```
public void example(Object o) {
    if (o != null) {
        System.out.println("o is " + o.toString());
    }
```

- You can make arrays or collections of `Objects`.

```
Object[] a = new Object[5];
a[0] = "hello";
a[1] = new Random();
List<Object> list = new ArrayList<Object>();
```

Recall: comparing objects

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

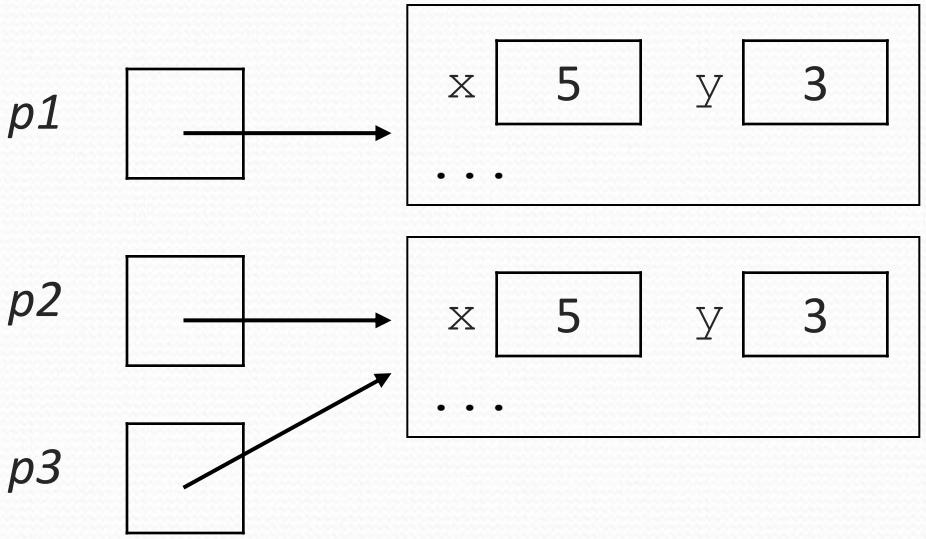
  == tests for **referential equality**, not state-based equality.

  It produces `true` only when you compare an object to itself.

```
Point p1 = new Point(5, 3);
Point p2 = new Point(5, 3);
Point p3 = p2;

// p1 == p2 is false;
// p1 == p3 is false;
// p2 == p3 is true

// p1.equals(p2)?
// p2.equals(p3)?
```



Default equals method

- The `Object` class's `equals` implementation is very simple:

```
public class Object {
    ...
    public boolean equals(Object o) {
        return this == o;
    }
}
```

- However:
  - When we have used equals with various kinds of objects, it didn't behave like `==`. Why not?
  - The Java API documentation for equals is elaborate. Why?

Overriding equals

- The Object class is designed for inheritance.
  - Its description and specification will apply to all other Java classes.

- So, its specification must be flexible enough to apply to all classes.
  - Subclasses will *override* equals to test for equality in their own way.
  - The Object equals spec enumerates basic properties that clients can rely on that method to have in all subtypes of Object.
    - (this == o) is compatible with these properties, but so are other tests.

Flawed `equals` method 1

```
public boolean equals(Point other) {     // bad
    if (x == other.x && y == other.y) {
        return true;
    } else {
        return false;
    }
}
```

- Let's write an `equals` method for a `Point` class.
  - The method should compare the state of the two objects and return `true` if they have the same x/y position.
  - What's wrong with the above implementation?

Flaws in the method

- The body can be shortened to the following (boolean zen):

```
return x == other.x && y == other.y;
```

- The parameter to equals must be of type `Object`, not `Point`.
  - It should be legal to compare a `Point` to *any* other object:

```
// 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.
  - By writing ours to accept a Point, we have *overloaded* equals.
    - Point has two equals methods: One takes an Object, one takes a Point.

Flawed `equals` method 2

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

- What's wrong with the above implementation?
  - It does not compile:

```
Point.java:36: cannot find symbol
symbol  : variable x
location: class java.lang.Object
return x == o.x && y == o.y;
            ^
```

  - The compiler is saying,
    "`o` could be any object. Not every object has an `x` field."

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
int len = o2.length();          // error
String line = o3.nextLine();    // error
```

  - (The objects referred to by `o1`, `o2`, and `o3` still do have those methods. They just can't be called through those variables because the compiler isn't sure what kind of object the variable refers to.)

Casting references

```
Object o1 = new Point(5, -3);
Object o2 = "hello there";

((Point) o1).translate(6, 2);       // ok
int len = ((String) o2).length();   // ok
Point p = (Point) o1;
int x = p.getX();                   // ok
```

- Casting references is different than casting primitives.
  - Doesn't actually change the object that is referred to.
  - Tells the compiler to *assume* that `o1` refers to a `Point` object.

- Watch out for precedence mistakes:

```
int len = (String) o2.length();     // error
```

Flawed equals method 3

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

- What's wrong with the above implementation?
  - It compiles and works when other `Point` objects are passed, but it throws an exception when a non-`Point` is passed (see next slide).

Comparing different types

```
Point p = new Point(7, 2);
if (p.equals("hello")) {   // should be false
    ...
}
```

- Currently our method crashes on the above code:

```
Exception in thread "main"
java.lang.ClassCastException: java.lang.String
        at Point.equals(Point.java:25)
        at PointMain.main(PointMain.java:25)
```

- The problem is the cast (cannot cast sideways in an inheritance tree):

```
public boolean equals(Object o) {
    Point other = (Point) o;
```

The instanceof keyword

reference `instanceof` **type**

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

- A binary, infix, `boolean` operator.
- Tests whether **variable** refers to an object of class **type** (or any subclass of **type**).

```
String s = "hello";
Point p = new Point();
```

expression	result
`s instanceof Point`	`false`
`s instanceof String`	`true`
`p instanceof Point`	`true`
`p instanceof String`	`false`
`p instanceof Object`	`true`
`s instanceof Object`	`true`
`null instanceof String`	`false`
`null instanceof Object`	`false`

Flawed equals method 4

```
// Returns true if o refers to a Point object
// with the same (x, y) coordinates as
// this Point; otherwise returns false.
public boolean equals(Object o) {
    if (o instanceof Point) {
        Point other = (Point) o;
        return x == other.x && y == other.y;
    } else {
        return false;
    }
}
```

- What's wrong with the above implementation?
  - It behaves incorrectly if the `Point` class is extended (see next slides).

Subclassing and equals

```
public class Point3D extends Point {
    private int z;
    public Point3D(int x, int y, int z) { ... }
    ...
}

Point3D p1 = new Point3D(4, 5, 0);
Point3D p2 = new Point3D(4, 5, 6);
Point   p3 = new Point  (4, 5);
```

- All objects above will report that they are "equal" to each other.
  - But they shouldn't be. The objects don't have equal state.
  - In some cases, they aren't even the same type.

Flawed equals method 5

```
public class Point3D extends Point {
    ...
    public boolean equals(Object o) {
        if (o instanceof Point3D) {
            Point3D other = (Point3D) o;
            return super.equals(o)
                   && z == other.z;
        } else {
            return false;
        }
    }
}
```

- What's wrong with the above implementation?
  - It produces *asymmetric* results when Point and Point3D are mixed.

A proper equals method

- Equality is reflexive:
  - `a.equals(a)` is `true` for every object `a`
- Equality is symmetric:
  - `a.equals(b)` ↔ `b.equals(a)`
- Equality is transitive:
  - `(a.equals(b) && b.equals(c))` ↔ `a.equals(c)`
- No non-`null` object is equal to `null`:
  - `a.equals(null)` is `false` for every object `a`
- **Effective Java Tip #8:**
  Obey the general contract when overriding `equals`.

The getClass method

- `getClass` returns information about the type of an object.
  - Commonly used for *reflection* (seen later).
  - Uses run-time type information, not the type of the variable.
  - Stricter than `instanceof`; subclasses return different results.

- `getClass` should be used when implementing `equals`.
  - Instead of `instanceof` to check for same type, use `getClass`.
  - This will eliminate subclasses from being considered for equality.
  - Caution: Must check for `null` before calling `getClass`.

Correct equals methods

```
public boolean equals(Object o) {    // Point
    if (o != null && getClass() == o.getClass()) {
        Point other = (Point) o;
        return x == other.x && y == other.y;
    } else {
        return false;
    }
}

public boolean equals(Object o) {    // Point3D
    if (o != null && getClass() == o.getClass()) {
        Point3D other = (Point3D) o;
        return super.equals(o) && z == other.z;
    } else {
        return false;
    }
}
```

Implementing generics

```
// a parameterized (generic) class
public class name<Type> {
    ...
}
```

- Forces any client that constructs your object to supply a type.
  - Don't write an actual type such as String; the client does that.
  - Instead, write a type variable name such as `E` or `T`.
  - You can require multiple type parameters separated by commas.
- The rest of your class's code can refer to that type by name.

- Exercise: Convert our list classes to use generics.

Generics and arrays

```
public class Foo<T> {
    private T myField;                          // ok

    public void method1(T param) {
        myField = new T();                      // error
        T[] a = new T[10];                      // error

        myField = param;                        // ok
        T[] a2 = (T[]) (new Object[10]);        // ok
    }
}
```

- You cannot create objects or arrays of a parameterized type.
- You can create variables of that type, accept them as parameters, return them, or create arrays by casting from `Object[]`.

Comparing generic objects

```
public class ArrayList<E> {
    ...
    public int indexOf(E value) {
        for (int i = 0; i < size; i++) {
        // if (elementData[i] == value) {
            if (elementData[i].equals(value)) {
                return i;
            }
        }
        return -1;
    }
}
```

- When testing objects of type E for equality, must use `equals`

Generic interface

```
// Represents a list of values.
public interface List<E> {
    public void add(E value);
    public void add(int index, E value);
    public E get(int index);
    public int indexOf(E value);
    public boolean isEmpty();
    public void remove(int index);
    public void set(int index, E value);
    public int size();
}

public class ArrayList<E> implements List<E> { …
public class LinkedList<E> implements List<E> { …
```