CSE 373

Objects in Collections:

Object; equals; compareTo; mutability

slides created by Marty Stepp
http://www.cs.washington.edu/373/

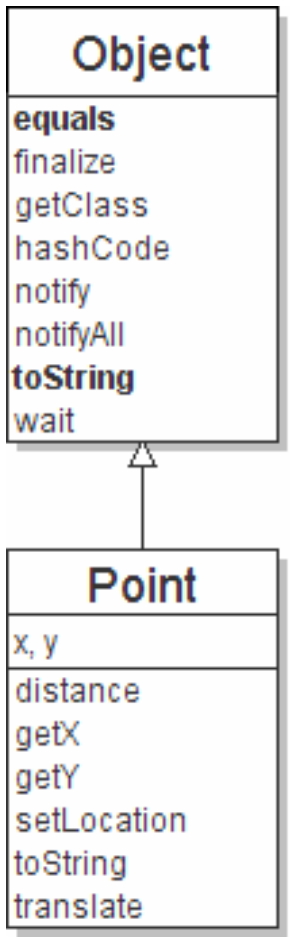
Recall: A typical Java class

```
public class Point {
    private int x;                          // fields
    private int y;

    public Point(int x, int y) {            // constructor
        this.x = x;
        this.y = y;
    }

    public int getX() { return x; }         // accessor
    public int getY() { return y; }

    public void translate(int dx, int dy) {
        x += dx;
        y += dy;                            // mutator
    }

    public String toString() {              // for printing
        return "(" + x + ", " + y + ")";
    }
}
```

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?

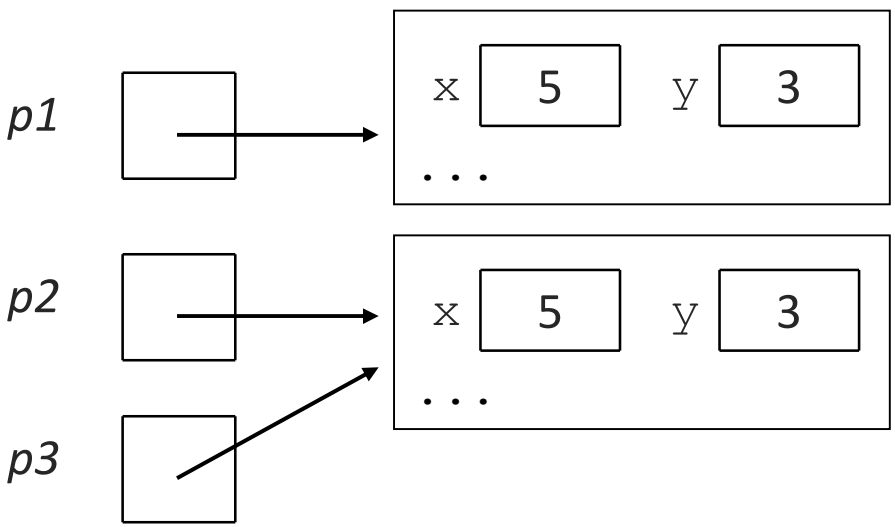
Recall: comparing objects

- 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?
    - Classes can *override* equals to provide their own equality test.
  - The Java API documentation for equals is elaborate. Why?
    - The equality test must meet various guidelines to be a proper test.

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."

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`

Correct equals method

```
// 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;
    }
}
```

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

Comparing objects

- Operators like < and > do not work with objects in Java.
  - But we do think of some types as having an ordering (e.g. `Dates`).
  - (In other languages, we can enable <, > with *operator overloading*.)

- **natural ordering**: Rules governing the relative placement of all values of a given type.
  - Implies a notion of equality (like `equals`) but also < and > .
  - **total ordering**: All elements can be arranged in A ≤ B ≤ C ≤ ... order.

- **comparison function**: Code that, when given two values *A* and *B* of a given type, decides their relative ordering:
  - A < B, A == B, A > B

The Comparable interface

- The standard way for a Java class to define a comparison function for its objects is to implement the `Comparable` interface.

```
public interface Comparable<T> {
    public int compareTo(T other);
}
```

- A call of **A**`.compareTo(`**B**`)` should return:

  a value `<` 0 if **A** comes "before" **B** in the ordering,

  a value `>` 0 if **A** comes "after" **B** in the ordering,

  or exactly 0 if **A** and **B** are considered "equal" in the ordering.

compareTo example

```
public class Point implements Comparable<Point> {
    // sort by x and break ties by y
    public int compareTo(Point other) {
        if (x < other.x) {
            return -1;
        } else if (x > other.x) {
            return 1;
        } else if (y < other.y) {
            return -1;   // same x, smaller y
        } else if (y > other.y) {
            return 1;    // same x, larger y
        } else {
            return 0;    // same x and same y
        }
    }
}
```

compareTo tricks

- *subtraction trick* - Subtracting `ints` works well for `compareTo`:

```
public int compareTo(Point other) {
    if (x != other.x) {
        return x - other.x;    // sort by x first
    } else {
        return y - other.y;    // if same x, break tie by y
    }
}
```

  - The idea:
    - if `x > other.x`, then `x - other.x > 0`
    - if `x < other.x`, then `x - other.x < 0`
    - if `x == other.x`, then `x - other.x == 0`

- To easily compare two `doubles`, try `Double.compare`:

```
public int compareTo(Employee other) {
    return Double.compare(salary, other.salary);
}
```

compareTo tricks 2

- *delegation trick* - If your object's fields are comparable (such as strings), use their `compareTo` results to help you:

```
// compare by name, e.g. "Joe" < "Suzy"
public int compareTo(Employee other) {
    return name.compareTo(other.getName());
}
```

- Guava has a nice `ComparisonChain` class for comparisons:

```
// compare by name, break tie by salary, then id
public int compareTo(Employee other) {
    return ComparisonChain.start()
            .compare(name, other.name)
            .compare(salary, other.salary)
            .compare(id, other.id)
            .result();
}
```

compareTo and equals

- `compareTo` should generally be consistent with `equals`.
  - `a.compareTo(b) == 0` should imply that `a.equals(b)`.

- *equals-compareTo trick* - If your class needs to implement both `equals` and `compareTo`, you can take advantage:

```
public boolean equals(Object o) {
    if (o instanceof Employee) {
        Employee other = (Employee) o;
        return this.compareTo(other) == 0;
    } else {
        return false;
    }
}
```

`compareTo` and collections

- Java's binary search methods call `compareTo` internally.

```
String[] a = {"al", "bob", "cari", "dan", "mike"};
int index = Arrays.binarySearch(a, "dan");  // 3
```

- Java's `TreeSet`/`Map` use `compareTo` internally for ordering.
  - Only classes that implement `Comparable` can be used as elements.

```
Set<String> set = new TreeSet<String>();
for (int i = a.length - 1; i >= 0; i--) {
    set.add(a[i]);
}
System.out.println(s);
// [al, bob, cari, dan, mike]
```

Mutation

- **mutation**: A modification to the state of an object.

```
Point p = new Point(3, 5);
p.translate(1, 3);          // mutator;  (4, 8)
```

- **immutable**: Unable to be changed (mutated).
  - Java example: Strings (can't change one, only produce a new one)

- Why? What is good about immutability?
  - easier to design, implement, and use immutable objects
  - less prone to developer error, misuse by clients
  - more secure (sometimes)
  - can be optimized for better performance / memory use (sometimes)

Making a class immutable

1. Don't provide any methods that modify the object's state.
2. Ensure that the class cannot be extended.
3. Declare all fields **`final`** (unable to be modified once set).
   - local variables (value can be set once, and can never be changed)
   - fields (they can be set only once, in the constructor)
   - static fields (they become "class constants")
   - classes (the class becomes unable to be subclassed)
   - methods (the method becomes unable to be overridden)
4. Declare all fields `private`. (ensure encapsulation)
5. Ensure exclusive access to any mutable object fields.
   - Don't let a client get a reference to a field that is a mutable object.

Mutable Fraction class

```
public class Fraction implements Cloneable, Comparable<Fraction> {
    private int numerator, denominator;

    public Fraction(int n)
    public Fraction(int n, int d)
    public int getNumerator(),  getDenominator()
    public void setNumerator(int n),  setDenominator(int d)

    public Fraction clone()
    public int compareTo(Fraction other)
    public boolean equals(Object o)
    public String toString()

    public void add(Fraction other)
    public void subtract(Fraction other)
    public void multiply(Fraction other)
    public void divide(Fraction other)
}
```

- How would we make this class immutable?

Immutable Fraction class

```
public final class Fraction implements Comparable<Fraction> {
    private final int numerator, denominator;

    public Fraction(int n)
    public Fraction(int n, int d)
    public int getNumerator(),  getDenominator()
    // no more setN/D methods

    // no clone method needed
    public int compareTo(Fraction other)
    public boolean equals(Object o)
    public String toString()

    public Fraction add(Fraction other)        // past mutators
    public Fraction subtract(Fraction other)   // are producers
    public Fraction multiply(Fraction other)   // (return a new
    public Fraction divide(Fraction other)     //  object)
}
```

Immutable methods

```
// mutable version
public void add(Fraction other) {
    numerator = numerator * other.denominator
              + other.numerator * denominator;
    denominator = denominator * other.denominator;
    reduce();   // private helper to reduce fraction
}

// immutable version
public Fraction add(Fraction other) {
    int n = numerator  * other.denominator
          + other.numerator * denominator;
    int d = denominator * other.denominator;
    return new Fraction(n, d);
}
```

- former mutators become *producers*
  - create/return a new immutable object rather than modifying this one

Mutability and collections

```
Set<Time> times = new HashSet<Time>();
times.add(new Time(11, 30, "AM"));
times.add(new Time(12, 45, "PM"));
Course c = new Course("CSE 373", 3, times, ...);
...

// c will be modified!
times.add(new Time(3, 30, "PM"));
```

- Since the course stores the set of times passed in, the object is in fact *mutable*. (called "representation exposure")
  - What could the `Course` author do to provide an immutable class?
    - copy the set of times in constructor;
      don't return a direct reference to it in `getTimes()`