CSE 331

Comparing objects; `Comparable`, `compareTo`, and `Comparator`

slides created by Marty Stepp
based on materials by M. Ernst, S. Reges, D. Notkin, R. Mercer,

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.

- **Effective Java Tip #12**: Consider implementing `Comparable`.

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
        }
    }

    // subtraction trick:
    // return (x != other.x) ? (x - other.x) : (y - other.y);
}
```

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]
```

Flawed compareTo method

```
public class BankAccount implements Comparable<BankAccount> {
    private String name;
    private double balance;
    private int id;
    ...
    public int compareTo(BankAccount other) {
        return name.compareTo(other.name);  // order by name
    }

    public boolean equals(Object o) {
        if (o != null && getClass() == o.getClass()) {
            BankAccount ba = (BankAccount) o;
            return name.equals(ba.name)
                && balance == ba.balance && id == ba.id;
        } else {
            return false;
        }
    }
}
```

- What's bad about the above? Hint: See Comparable API docs.

The flaw

```
BankAccount ba1 = new BankAccount("Jim", 123, 20.00);
BankAccount ba2 = new BankAccount("Jim", 456, 984.00);

Set<BankAccount> accounts = new TreeSet<BankAccount>();
accounts.add(ba1);
accounts.add(ba2);
System.out.println(accounts);     // [Jim($20.00)]
```

- Where did the other account go?
  - Since the two accounts are "equal" by the ordering of `compareTo`, the set thought they were duplicates and didn't store the second.

compareTo and equals

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

- from `Comparable` Java API docs:
  - ... sorted sets (and sorted maps) without explicit comparators behave strangely when they are used with elements (or keys) whose natural ordering is inconsistent with equals. In particular, such a sorted set (or sorted map) violates the general contract for set (or map), which is defined in terms of the equals method.
  - For example, if one adds two keys a and b such that (!a.equals(b) && a.compareTo(b) == 0) to a sorted set that does not use an explicit comparator, the second add operation returns false (and the size of the sorted set does not increase) because a and b are equivalent from the sorted set's perspective.

What's the "natural" order?

```
public class Rectangle implements Comparable<Rectangle> {
    private int x, y, width, height;

    public int compareTo(Rectangle other) {
        // ...?
    }
}
```

- What is the "natural ordering" of rectangles?
  - By x, breaking ties by y?
  - By width, breaking ties by height?
  - By area? By perimeter?

- Do rectangles have any "natural" ordering?
  - Might we ever want to sort rectangles into some order anyway?

Comparator interface

```
public interface Comparator<T> {
    public int compare(T first, T second);
}
```

- Interface `Comparator` is an external object that specifies a comparison function over some other type of objects.
  - Allows you to define multiple orderings for the same type.
  - Allows you to define a specific ordering for a type even if there is no obvious "natural" ordering for that type.

Comparator examples

```
public class RectangleAreaComparator
        implements Comparator<Rectangle> {
    // compare in ascending order by area (WxH)
    public int compare(Rectangle r1, Rectangle r2) {
        return r1.getArea() - r2.getArea();
    }
}

public class RectangleXYComparator
        implements Comparator<Rectangle> {
    // compare by ascending x, break ties by y
    public int compare(Rectangle r1, Rectangle r2) {
        if (r1.getX() != r2.getX()) {
            return r1.getX() - r2.getX();
        } else {
            return r1.getY() - r2.getY();
        }
    }
}
```

Using Comparators

- `TreeSet` and `TreeMap` can accept a `Comparator` parameter.

```
Comparator<Rectangle> comp = new RectangleAreaComparator();
Set<Rectangle> set = new TreeSet<Rectangle>(comp);
```

- Searching and sorting methods can accept `Comparators`.

```
Arrays.binarySearch(array, value, comparator)
Arrays.sort(array, comparator)
Collections.binarySearch(list, comparator)
Collections.max(collection, comparator)
Collections.min(collection, comparator)
Collections.sort(list, comparator)
```

- Methods are provided to reverse a `Comparator`'s ordering:

```
Collections.reverseOrder()
Collections.reverseOrder(comparator)
```

Using `compareTo`

- `compareTo` can be used as a test in an `if` statement.

```
String a = "alice";
String b = "bob";
if (a.compareTo(b) < 0) {   // true
    ...
}
```

Primitives	Objects
`if (a < b) { ...`	`if (a.compareTo(b) < 0) { ...`
`if (a <= b) { ...`	`if (a.compareTo(b) <= 0) { ...`
`if (a == b) { ...`	`if (a.compareTo(b) == 0) { ...`
`if (a != b) { ...`	`if (a.compareTo(b) != 0) { ...`
`if (a >= b) { ...`	`if (a.compareTo(b) >= 0) { ...`
`if (a > b) { ...`	`if (a.compareTo(b) > 0) { ...`

compareTo tricks

- *subtraction trick* - Subtracting related numeric values produces the right result for what you want `compareTo` to return:

```
// sort by x and break ties by y
public int compareTo(Point other) {
    if (x != other.x) {
        return x - other.x;   // different x
    } else {
        return y - other.y;   // same x; compare 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`

      - NOTE: This trick doesn't work for `doubles` (but see `Math.signum`)

`compareTo` tricks 2

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

```
// sort by employee name, e.g. "Jim" < "Susan"
public int compareTo(Employee other) {
    return name.compareTo(other.getName());
}
```

- *`toString` trick* - If your object's `toString` representation is related to the ordering, use that to help you:

```
// sort by date, e.g. "09/19" > "04/01"
public int compareTo(Date other) {
    return toString().compareTo(other.toString());
}
```