Creating a Class Beyond the Basics (pt 1)

Based on slides by Prof. Burton Ma

Comparable Objects

- Many value types have a natural ordering
  - That is, for two objects `x` and `y`, `x` is less than `y` is meaningful
    - **`Short`**, **`Integer`**, **`Float`**, **`Double`**, etc
    - **`String`**s can be compared in dictionary order
    - **`Date`**s can be compared in chronological order
    - you might compare **`Vector2d`**s by their length
    - **`Die`**s can be compared by their face value
- If your class has a natural ordering, consider implementing the **`Comparable`** interface
  - Doing so allows clients to sort arrays or **`Collection`**s of your object

Interfaces

- An interface is (usually) a group of related methods with empty bodies
  - The **`Comparable`** interface has just one method

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

- A class that implements an interfaces promises to provide an implementation for every method in the interface

`compareTo()`

- Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer if this object is less than, equal to, or greater than the specified object.
- Throws a **`ClassCastException`** if the specified object type cannot be compared to this object.

Die compareTo()

```
public class Die implements Comparable<Die> {
  // attributes, constructors, methods ...


  public int compareTo(Die other) {
    int result = 0;
    if (this.getValue() < other.getValue()) {
      result = -1;
    }
    else if (this.getValue() > other.getValue()) {
      result = 1;
    }
    return result;
  }
}
```

Die compareTo()

- the following also works for the Die class, but is dangerous in general:

```
public int compareTo(Die other) {
  int result = this.getValue() – other.getValue();
  return result;
}
```

Comparable Contract

1. The sign of the returned **`int`** must flip if the order of the two compared objects flip
   - if **`x.compareTo(y) > 0`** then **`y.compareTo(x) < 0`**
   - if **`x.compareTo(y) < 0`** then **`y.compareTo(x) > 0`**
   - if **`x.compareTo(y) == 0`** then **`y.compareTo(x) == 0`**

Comparable Contract

2. **`compareTo()`** must be transitive
   - if `x.compareTo(y) > 0 && y.compareTo(z) > 0` then `x.compareTo(z) > 0`
   - if `x.compareTo(y) < 0 && y.compareTo(z) < 0` then `x.compareTo(z) < 0`
   - if `x.compareTo(y) == 0 && y.compareTo(z) == 0` then `x.compareTo(z) == 0`

Comparable Contract

3. If **`x.compareTo(y) == 0`** then the signs of **`x.compareTo(z)`** and **`y.compareTo(z)`** must be the same

Consistency with equals

- An implementation of **`compareTo()`** is said to be consistent with **`equals()`** when

  if **`x.compareTo(y) == 0`** then
  `x.equals(y) == true`

  and

  if **`x.equals(y) == true`** then
  `x.compareTo(y) == 0`

Not in the Comparable Contract

- It is not required that **`compareTo()`** be consistent with **`equals()`**
  - That is if **`x.compareTo(y) == 0`** then
    `x.equals(y) == false` is acceptable
  - Similarly if **`x.equals(y) == true`** then
    `x.compareTo(y) != 0` is acceptable

- Try to come up with examples for both cases above

Implementing **`compareTo`**

- Implementing **`compareTo`** is similar to implementing equals
- You need to compare all of the attributes
  - Starting with the attribute that is most significant for ordering purposes and working your way down

PhoneNumber compareTo()

```
public class PhoneNumber implements Comparable<PhoneNumber> {
  // attributes, constructors, methods ...


  public int compareTo(PhoneNumber other) {
    int result = 0;
    result = this.getAreaCode() – other.getAreaCode();
    if (result == 0) {
      result = this.getExchangeCode() – other.getExchangeCode();
    }
    if (result == 0) {
      result = this.getStationCode() – other.getStationCode();
    }
    return result;
  }
}
```

Implementing **`compareTo`**

- If you are comparing attributes of type **`float`** or **`double`** you should use **`Float.compare`** or **`Double.compare`** instead of **`<`**, **`>`**, or **`==`**

- If your **`compareTo`** implementation is broken, then any classes or methods that rely on **`compareTo`** will behave erratically
  - **`TreeSet`**, **`TreeMap`**
  - Many methods in the utility classes **`Collections`** and **`Arrays`**

Privacy Leaks

- A mutable object that is passed to or returned from a method can be changed

- Problems:
  - Private attributes become publicly accessible
  - Objects can be put into an inconsistent state

- Solution:
  - Make a copy of the object and save the copy
    - Use copy constructors

Avoiding Privacy Leaks

- Bad

```
public Date getDueDate()
{
        return dueDate; // Unsafe
}
```

- Good

```
public Date getDueDate()
{
        return new Date(dueDate.getTime()); // Avoid leak
}
```

Avoiding Privacy Leaks (con't)

- Bad

```
public void setDueDate(Date newDate)
{
        dueDate = newDate; // Unsafe
}
```

- Good

```
public void setDueDate(Date newDate)
{
        dueDate = new Date(newDate.getTime()); // Avoid leak
}
```