Comparisons and the `Comparable` Interface

Comparison

- Something that we do a lot
- Can compare all kinds of data with respect to all kinds of comparison relations
  - identity
  - equality
  - order
  - lots of others

Identity vs Equality

- identity: == != (primitive and reference types)
- testing equality of objects: use **`equals`**
- **`equals`** is defined in class **`Object`**
- any class you create inherits **`equals`** from its parent class, but you can override it (and probably want to)

Identity vs Equality

- Quiz: What are the results of the following tests?
  - "hello".equals("hello") **`true`**
  - "hello" == "hello" **`true`**
  - new String("hello") == new String("hello") **`false`**

Order

- numeric primitives: use <, >, <=, >=
- objects?
  - **`Integer`** – compare by value
  - **`String`** – compare lexicographically (dictionary order)
  - cannot use <, >, <=, >=

Order

- for reference types, use **`Comparable`** interface

```
interface Comparable {
  int compareTo(Object x);
}
```

- (note: this is Java 1.4.2 – Java 5.0 has generics)
- **`x.compareTo(y)`** returns a negative, zero, or positive integer according as **`x`** is "less than", "equal to", or "greater than" **`y`**, respectively
- "less than", "equal to", and "greater than" are *defined* for that class by the implementation of **`compareTo`**

Example

- Compare people by weight:

```
class Person implements Comparable {
  private int weight;
  ...
  public int compareTo(Object obj) {
    return ((Person)obj).weight - weight;
  }
  public boolean equals(Object obj) {
    return obj instanceof Person &&
      ((Person)obj).weight == weight;
  }
}
```

Note

If a class has an **`equals`** method and also implements **`Comparable`**, then it is advisable (but not enforced) that

`a.equals(b)`

exactly when

`a.compareTo(b) == 0`.

Generic Code

- The **`Comparable`** interface allows generic code for sorting, searching, and other operations that only require comparisons

```
static void mergeSort(Comparable[] a) {...}
static void bubbleSort(Comparable[] a) {...}
```

- The sort methods do not need to know what they are sorting, only how to compare elements

Generic Code

- Finding the max element of an array

```
//return max element of an array
static Comparable max(Comparable[] a) {
  //throws ArrayIndexOutOfBoundsException
  Comparable max = a[0];
  for (Comparable x : a) {
    if (x.compareTo(max) > 0) max = x;
  }
  return max;
}
```

- What is the max element? Whatever **`compareTo`** says it is!

Another Example

- Lexicographic comparison of **`Comparable`** arrays
- for **`int`** arrays, **`a < b`** lexicographically iff either:
  - **`a[i] == b[i]`** for **`i < j`** and **`a[j] < b[j]`**; or
  - **`a[i] == b[i]`** for all **`i < a.length`**, and **`b`** is longer

```
//compare two Comparable arrays lexicographically
static int arrayCompare(Comparable[] a, Comparable[] b) {
  for (int i = 0; i < a.length && i < b.length; i++) {
    int x = a[i].compareTo(b[i]);
    if (x != 0) return x;
  }
  return b.length - a.length;
}
```