

Revision: sub-class

- Java supports a restricted form of sub-class relationship.
- Each Java class is a subclass of one particular class, except the class Object which is not a subclass of any other class.
- Super class specifies a set of methods that all sub-classes must support. It can also provide default implementations.
- At any place that expects a value of a class, you can give it a value of its sub-class.
- Abstract classes are classes that cannot be instantiated.

0911B L24-1

The Object class

As we have said, all classes are sub-class of Object. That means all classes provide all the services specified by the class Object. E.g.,

- boolean equals(Object obj): return whether another object is equal to the object itself. By default, an object equals only to itself (i.e. a.equals(b) is the same as a==b).
- String toString(): return a string representation of the object. By default, returns a string like "Obj@3179c3"

Both are actually heavily used in the Java language and standard library. For example, whenever you concatenate an object to a String (or use print to print an object), toString() is automatically invoked.

0911B L24-2

Example

For example, currently PlayingCard does not provide its own equals method, meaning that equals have the default behaviour of returning true only if two objects are actually the same object.

We can modify it by adding the following method to PlayingCard:

```
public boolean equals(Object obj) {
    if (! (obj instanceof PlayingCard))
        return false;
    PlayingCard card = (PlayingCard) obj;
    return suit == card.suit && value == card.value;
}
```

Notice that **we must use Object as the argument type as required by the super-class Object**. We cannot use PlayingCard. So we have to make a cast.

0911B L24-3

Method overloading

What will happen if we really define the method it like equals(PlayingCard card) instead?

Answer: a separate, **completely unrelated** method will be created.

- In Java, we can have more than one method with the same name, as long as they have different "signatures".
- The "signature" of a method is its name and the types of its arguments. We write signatures like equals(PlayingCard), or even PlayingCard.equals(PlayingCard).
- Note that **signature does not include return type**.
- It is basically a "convenience feature" of Java, so that the compiler automatically choose the right method to call.

0911B L24-4

Behaviour of method overloading

- **Overloaded methods are completely unrelated methods.**
- Suppose we actually define PlayingCard.equals(PlayingCard), without defining PlayingCard.equals(Object).
- For `Object obj;` and `PlayingCard card;`, `card.equals(obj)` calls `PlayingCard.equals(Object)`—even if `obj` is a `PlayingCard`!
- But for `PlayingCard card1, card2;` (two `PlayingCard` references), `card1.equals(card2)` calls `PlayingCard.equals(PlayingCard)`.
- Due to this strange behaviour, it is not a good idea to define `PlayingCard.equals(PlayingCard)`.

0911B L24-5

When to use method overloading

- Method overloading is best used when something can logically be done to two or more unrelated types.
- Example: see what Java does for the Arrays class:

```
public class Arrays {
    ...
    public static void sort(char[] a) { ... }
    public static void sort(short[] a) { ... }
    public static void sort(int[] a) { ... }
    public static void sort(byte[] a) { ... }
    public static void sort(double[] a) { ... }
    public static void sort(float[] a) { ... }
    public static void sort(long[] a) { ... }
    public static void sort(Object[] a) { ... }
}
```
- The compiler thus automatically finds the correct sort to call.
- Method overloading is not frequently used in our own code, though.

0911B L24-6

Sorting Objects?

- One problem shows up when looking at the sort method of array: how are Object's sorted?
- For any "sort" to make sense, the things to be sorted must have an ordering.
- But objects in general do not have an ordering.
- For Object's to be comparable, it must have the property that given two objects, we can compare them.
- It would be nice if all comparable classes have a common super-class to define a compare operation. But this is impossible since each class can only have one immediate super-class, and something more important probably has used it.

0911B L24-7

Java interface

- In Java, a property is defined by an **interface**. It is very similar to an abstract class, except that interface cannot provide default implementations and cannot define any data field.
- Any class can **implement** any number of interface. This relieves the unnatural design that each class can only have one superclass (since it can implement as many interface it wants).
- For example, any class that has the property that "it can be compared" will implement an interface `Comparable`, and then provide a method `int compareTo(Object obj)`.
- The method should return a negative number, 0 and a positive number depending on whether it is smaller than, equal to or larger than obj.

0911B L24-8

Example: PlayingCard ordering

For example, let's impose a natural ordering for the PlayingCards: first compare the suit, and if they are equal compare the values. Once this is done, an array of PlayingCard can be sorted using `Arrays.sort`.

```
public class PlayingCard implements Comparable {
    ...
    public int compareTo(Object obj) {
        PlayingCard card = (PlayingCard) obj;
        if (suit < card.suit)
            return -1;
        else if (suit > card.suit)
            return 1;
        else
            return value - card.value;
    }
}
```

0911B L24-9