Revision: classes and data fields

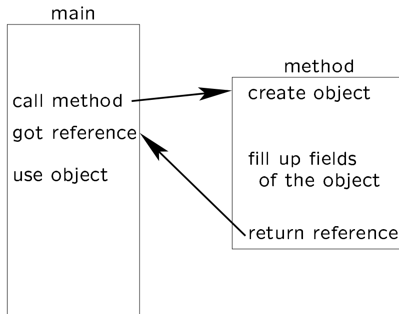
- We can define a class by writing `class ClassName { ... }`.
- We can specify what data fields are in an object of the class we define, by writing things like `public int a_field;` within the class definition.
- Once that is done, every object has a data field called `a_field`.
- Given an object reference `ref` to `ClassName`, we can access a data field by `ref.a_field`.
- If we create an array of `ClassName`, what we actually create is an array of `ClassName` references, initially `null`—meaning "not refering to any object at this time".

The use of objects without instance methods

- You should have noticed that our classes are conceptually different from those defined in other classes: it requires us to know the internal implementation (data fields) of the class.
- It's because we still don't know how to define instance methods.
- However, even classes like that can be useful, usually because of two reasons:
  - It is an object. So we can pass a reference of it into a method, and the method can modify it.
  - It can contain multiple data fields, so a method can return a reference to it to return multiple values.

A method returning a new object

We can return a reference from a method, and let the caller (e.g., main) get the reference.



Example 1

We might want to have a method that compute statistics of an array:

```
class Stats {
  public double max, min, average, sd;

  public static Stats getStats(double[] arr) {
    if (arr == null || arr.length == 0)    // No stat for empty array
      return null;
    Stats ret = new Stats();               // Create object to return
    ret.max = ret.min = arr[0];            // Deal with 1st element
    double sum = arr[0], sum_sqr = arr[0] * arr[0];
    for (int i = 1; i < arr.length; ++i) { // Deal with each other element
      if (arr[i] > ret.max)
        ret.max = arr[i];
      else if (arr[i] < ret.min)
        ret.min = arr[i];
      sum += arr[i];
      sum_sqr += arr[i] * arr[i];
    }
    ret.average = sum / arr.length;
    ret.sd = Math.sqrt(sum_sqr / arr.length - ret.average * ret.average);
    return ret;
  }
}
```


Example 1 (cont'd)

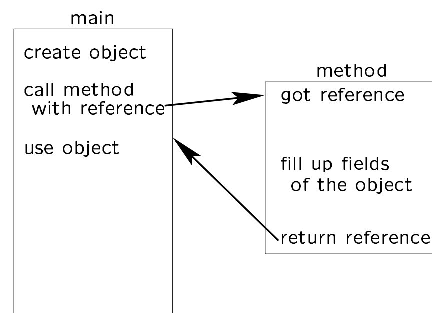
Once you have the method and the class, you can find the statistics of some numbers by calling `getStats`. Example:

```
public static void main(String[] args) {
  StdIn in = new StdIn();
  int n = in.readInt();
  double[] arr = new double[n];
  for (int i = 0; i < n; ++i)
    arr[i] = in.readInt();
  Stats st = Stats.getStats(arr);       // Get the statistics
  if (st != null) {
    System.out.println("Max = " + st.max);
    System.out.println("Min = " + st.min);
    System.out.println("Average = " + st.average);
    System.out.println("SD = " + st.sd);
  }
}
```

Note that in the main method, we did not create a new `Stat` object: it is done by `getStats` already. The reference returned by `getStats` already refers to something.

A method receiving an object and write the values there

The caller can also give a method a reference to object in order for the method to fill.



Example 2

Rather than creating a new `Stat` object for each call to `getStats`, one can instead ask the caller to provide a `Stats` object:

```
class Stats {
  ...
  public static boolean getStats(double[] arr, Stats stats) {
    if (arr == null || arr.length == 0)    // No stat for empty array
      return false;
    if (stats == null)
      return false;
    stats.max = stats.min = arr[0];        // Deal with 1st element
    double sum = arr[0], sum_sqr = arr[0] * arr[0];
    for (int i = 1; i < arr.length; ++i) { // Deal with each other element
      if (arr[i] > stats.max)
        stats.max = arr[i];
      else if (arr[i] < stats.min)
        stats.min = arr[i];
      sum += arr[i];
      sum_sqr += arr[i] * arr[i];
    }
    stats.average = sum / arr.length;
    stats.sd = Math.sqrt(sum_sqr / arr.length - stats.average * stats.average);
    return true;
  }
}
```


Example 2 (cont'd)

The main program then need to create its own Stats object:

```
public static void main(String[] args) {
  StdIn in = new StdIn();
  int n = in.readInt();
  double[] arr = new double[n];
  for (int i = 0; i < n; ++i)
    arr[i] = in.readInt();
  Stats st = new Stats();              // Create its own Stats object
  if (Stats.getStats(arr, st)) {       // Get the statistics
    System.out.println("Max = " + st.max);
    System.out.println("Min = " + st.min);
    System.out.println("Average = " + st.average);
    System.out.println("SD = " + st.sd);
  }
}
```

Note that `getStats` can modify the object referred to by the reference `st`. Were we passing a whole brunch of `double`'s, we cannot do that.

Question: When is this method better?

Adding instance methods

To be able to deal with objects of a class without knowing its internal details, we still need one piece of the puzzle.

- Instance methods will define all the operations that can be done for (to) an object.
- Instance methods are specialized to deal with objects.
- So **an instance method is always called for an object**.
- For example, if we have a PlayingCard object `card`, we might write `card.toString()` to get a String representation of the PlayingCard. This calls the method `toString` **for the object** `card`.

Defining methods

- Definitions of instance methods are very similar to other methods, except that the keyword `static` is dropped.
- Instance methods always receive an extra argument: the object that the method is being called for.
- The extra argument is called `this`. However, in the method, we seldom need to refer to `this`, since we can access its fields without writing out `this`.
- That is, if you write the name of a field `f` without writing a object following a dot, it refers to `this.f`.

Example method

Here we add a `toString` method to the `PlayingCard` class we defined earlier. This overrides the default `toString` method.

```
class PlayingCard {
  public static final String[] SUIT_NAME =
    { "Spade", "Heart", "Diamond", "Club" };
  public static final String[] VALUE_NAME = {
    "", "", "2", "3", "4", "5", "6", "7", "8", "9", "10",
    "Jack", "Queen", "King", "Ace"
  };

  public byte suit;
  public byte value;

  public String toString() {
    return SUIT_NAME[suit] + ' ' + VALUE_NAME[value];
  }
}
```

Question 1: Why we can write `suit` and `value` without quoting an object? (Ans:)

Question 2: What are those `static final` arrays? (Ans:)

Modifiers

- Now it starts to get messy: sometimes we need `static`, sometimes we need `final`, sometimes we need both. (`public` will be dealt with later) They are called "modifiers".
- Static means *not for an object*. Final means *not modifiable*.
- The difference of methods and variables is actually in the pair of parentheses that can be found only for methods.
- 8 variations. Most frequently used:
  - Variables: no modifier (data field), `static` (class variable), `final static` (constant).
  - Methods: no modifier (instance method), `static` (ordinary methods).