2023/2024(1)
EF234302 Object Oriented Programming

Lecture #4a

Array, ArrayList & Scanner

Misbakhul Munir IRFAN SUBAKTI
司馬伊凡
Мисбакхул Мунир Ирфан Субакти

All about array

- We've seen before, arrays are very, very common in all programming languages.

```
public class Hello {
    public static void main(String[] args) {
        …
```

- See the `String[] args` in the main method declaration? That's an array. An array is like a pigeonhole rack, with a fixed number of slots.
- The idea is that we create an array knowing beforehand how big it needs to be.

String array: Example

- Let's try an example. We only like five fruits, so we'll have an array of five Strings to represent their names. The syntax for using an array is as follows:

```
String[] fruitArray = new String[5];
fruitArray[0] = "srikaya";
```

- In the code above, we have created a *new instance* of String array, and said the maximum number of Strings that can go in this array is five. We're then putting the string "`srikaya`" in the first position in the array (again, we number from zero).
- Exercise: What would happen if we wrote `fruitArray[5] = "durian"`? Why?
  - Note that an array can store any type: Strings, BankAccounts, other arrays, etc., but also any *primitive type* (int, char, boolean, etc.). This is very useful!
- Exercise: How would we create an array of `int`s, of size ten?

Array: Addressing and reading

- In an array, we use a number in square brackets to denote the position we want to set or modify. Note that it's not possible to just add something to an array, we have to know where we want to put it.
- Assume the fruit array is now full of its five elements. We can print them out using a `for`-loop:

```
System.out.println("The element at position 0 is " + fruitArray[0]);
for (int fruitIndex = 0; fruitIndex < 5; fruitIndex++) {
    System.out.println("Fruit " + fruitIndex + " is " + fruitArray[fruitIndex]);
}
```

- This is fine, but it may not always work—what if we don't know how big the array is? What we actually want to do is continue printing elements for the whole length of the array, and for that we can use the `.length` command. This gives the size of the array, but notice—there are no parenthesis "`()`":

```
for (int fruitIndex = 0; fruitIndex < fruitArray.length; fruitIndex++) {
    System.out.println("Fruit " + fruitIndex + " is " + fruitArray[fruitIndex]);
}
```

- We should always use `.length` when traversing an array.

Array shorthand

- If we know what we want to put in our array as soon as the array is created, there is a shorthand way to fill it. The two examples below are equivalent:

```
String[] firstFruitArray = new String[3];
firstFruitArray[0] = "strawberry";
firstFruitArray[1] = "raspberry";
firstFruitArray[2] = "blackcurrant";
String[] secondFruitArray = {"strawberry",
   "raspberry", "blackcurrant"};
```

- Note that in the second example, we don't specify a size, because the size is implicit from the number of elements we put in the curly brackets. A useful shortcut, but be sure to understand it first.

Back to the `main` method

- We saw at the previous lecture that the `main` method contains an array in its parameters:

```
public class Hello2 {
   public static void main(String[] args) { ...
```

- We know that the `main` method is the first thing called when we type `java Hello2`, so what could that array of Strings be for?
- In fact, it's the command line arguments, such that if we type

```
D:\OOP>java Hello2 one two three
```

- the array `args` will hold the Strings `"one","two","three"`. This array is very important, and is used in a variety of situations when information that a program needs is variable even after compilation.

Input

- By now, we all know how to make Java print, using the `System.out.println` command. Quite often though, we need also to be able to read in:

```
Scanner sc = new Scanner(System.in);
String aLineOfText = sc.nextLine();
```

- `Scanner` is a useful tool that allows us to grab input from the keyboard. In order to use it, we need to add `import java.util.Scanner` to the top of our classes, to tell Java to include it in our code. In this example, when we write `sc.nextLine()`, we're reading a line of text from the keyboard, and assigning it to `aLineOfText`. `Scanner` provides a number of other useful input types, such as `sc.nextInt()` or `sc.nextDouble()`:

```
System.out.println("Pick an integer: ");
int value = sc.nextInt();
System.out.println("You picked " + value +
    ". The square of that is " + (value * value) + ".");
```

- If we enter something that isn't an `int`, Java will throw an error and crash. Don't worry, that's normal—we'll explain how to deal with those kinds of errors in a better way.

ArrayList

- It's very common when programming that we'll want to store more than one of something, but we might not know how many things we want to store. Take the following example:

```
System.out.println("Enter some fruit names. Enter #stop# to finish");
Scanner s_in = new Scanner(System.in);
String line = s_in.nextLine();
while (!line.equals("#stop#") {
    System.out.println("You just typed " + line);
    line = s_in.nextLine();
}
```

- What does this code do? What if we wanted to store all of the fruit that the user entered? We don't know how many String variables are required. To get around this, we use an `ArrayList`, which we can think of like a row of pigeonholes, which we can add extra racks to when full.

`ArrayList` (continued)

- We can create and fill an `ArrayList` by modifying the code above. Note that we now need to import `java.util.ArrayList` too:

```
System.out.println("Enter some fruit names. Enter #stop# to finish");
Scanner s_in = new Scanner(System.in);
ArrayList<String> fruit = new ArrayList<String>();
String line = s_in.nextLine();
// Keep adding fruit to the list
while (!line.equals("#stop#") {
    System.out.println("You just typed " + line + ". Adding it!");
    fruit.add(line);
    line = s_in.nextLine();
}
System.out.println("You entered " + fruit.size() +
    " fruit. Can't you do better?");
```

- Have a look at what's going on here. As before, we're setting up the `Scanner`, but this time we're creating an `ArrayList` (which will hold `String`s), and adding each line that the user enters to the `ArrayList` using the `add` method. At the end, the size of the `ArrayList` is obtained using the `size` method.
- What's clever about this is that the list will expand by itself, when it gets full, without we having to do anything.

Getter, Setter and other methods

- There are many operations we can perform on ArrayLists. The most obvious is to get an element at a specific position. Let's say we want the first element:

```
fruit.get(0);
```

- Exercise: Write code to get the 10th fruit from the list. Bear in mind that there may not be ten fruit in the list, and we should handle this eventuality.
- We might also want to check whether any given fruit is in our list of fruit names—we learned that the `boolean` type can store this sort of information:

```
boolean peachIsIn = fruit.contains("peach");
if (peachIsIn) {
    System.out.println("You entered peach, somewhere, apparently");
} else {
    System.out.println("No peaches");
}
```

- ... and if so, we might want to know the position within the ArrayList that `peach` is:

```
int positionOfPeach = fruit.indexOf("peach");
int positionOfBanana = fruit.indexOf("banana");
```

Getter, Setter and other methods (cont'd)

- If the String we look for isn't in the list, `indexOf` will return -1.
- It's also possible to set a value at a particular position. We've gone off plums of late, and prefer lemons. Plums are in position 3:

```
fruit.set(3, "lemon");
```

- Alternatively we might decide to remove a fruit completely:

```
fruit.remove(3);
fruit.remove("plum");
```

- Note that these two methods are very different. In the first case, we just remove the element at position 3, irrespective of what it is (actually, that element is returned by the `remove` call). In the second, we try to remove the first instance of a string "`plum`" in the list—this returns a `boolean` (why?).

Repetition & conditionals with ArrayList

- ArrayLists are the perfect excuse to get more practice with `for`, `while` and `if`! We've already used `if` above, to check if something was in the list:

```
if (fruit.contains("nectarine")) {
    System.out.println("Yes to nectarines!");
}
```

- Something quite important is to be able to print out all of the elements in the list. A `for`-loop is perfect for that:

```
for (int j = 0; j < fruit.size(); j++) {
    System.out.println(fruit.get(j));
}
```

- Note that the `for`-loop will stop repeating when `j` is equal to the size of the list, and that we're always printing the `j`-th element of the list (from `0` through `size()-1`).
- The `while` loop is also useful for list processing. We saw above that we can keep adding elements to a list until the user wishes to stop.

Array or ArrayList?

- Good question.
- Arrays are, we might say, more basic than ArrayLists. They don't require any extra imports, and are compatible with more types. It's also easier to visualise something like a two-dimensional array, and to access its elements.
- However, arrays are *immutable*. If you need to make one bigger, you're stuck (there is a way, but it's long-winded. Any ideas?). It's far better to just use an ArrayList in these circumstances.
  - java.lang.System.arrayCopy, java.util.ArrayList<T>, java.util.Arrays.copyOf