Using Objects and APIs

Today, we look at one of the core features of Java and similar languages: objects. We will also learn about a resizeable alternative to arrays.

Content Learning Targets

After completing this activity, you should be able to say:

- I can describe the purpose of String objects in Java.
- I can read Java API documentation to learn how to use built-in classes.
- I can explain differences between primitive and reference type variables.
- I can apply basic String operations to solve small problems.
- I can distinguish ArrayLists and arrays.
- I can apply basic ArrayList operations to solve small problems.
- I can read and write enhanced for loops.
- I can draw box-and-pointer memory diagrams of Strings, arrays, and ArrayLists.

Process Skill Goals

During the activity, you should make progress toward:

- Leveraging prior knowledge and experience of other students. (Teamwork)
- Justifying answers based on evidence provided in the model. (Problem Solving)
- Tracing execution of while/for loops and predicting their output. (Critical Thinking)

Facilitation Notes

The meta activity reinforces the importance of roles. Successful teams are able to accomplish all the tasks outlined on the Role Cards, and there's too many things for one person to keep track. Ask the reflectors to pay special attention to their role during today's activity, and invite them at some point to report to share what they have observed.

In Model 1, teams learn about basic String operations.

Model 2 introduces ArrayLists by side-by-side comparison with arrays.

Model 3 shows how memory diagrams represent ArrayList, String, and Integer objects.

Key questions: #3, #5, #10, #12, #16

Source files: *StringDemo.java*, *ArrayListDemo.java*, *stringUsage/StringProbs.java*

Meta Activity: Team Disruptions

Common disruptions to learning in teams include: talking about topics that are off-task, teammates answering questions on their own, entire teams working alone, limited or no communication between teammates, arguing or being disrespectful, rushing to complete the activity, not being an active teammate, not coming to a consensus about an answer, writing incomplete answers or explanations, ignoring ideas from one or more teammates.

Questions (10 min) **Start time:**

1. Pick four of the disruptions listed above. For each one, find something from the role cards that could help improve the team's success. Use a different role for each disruption.

a) Manager:

limited communication between teammates

b) Presenter:

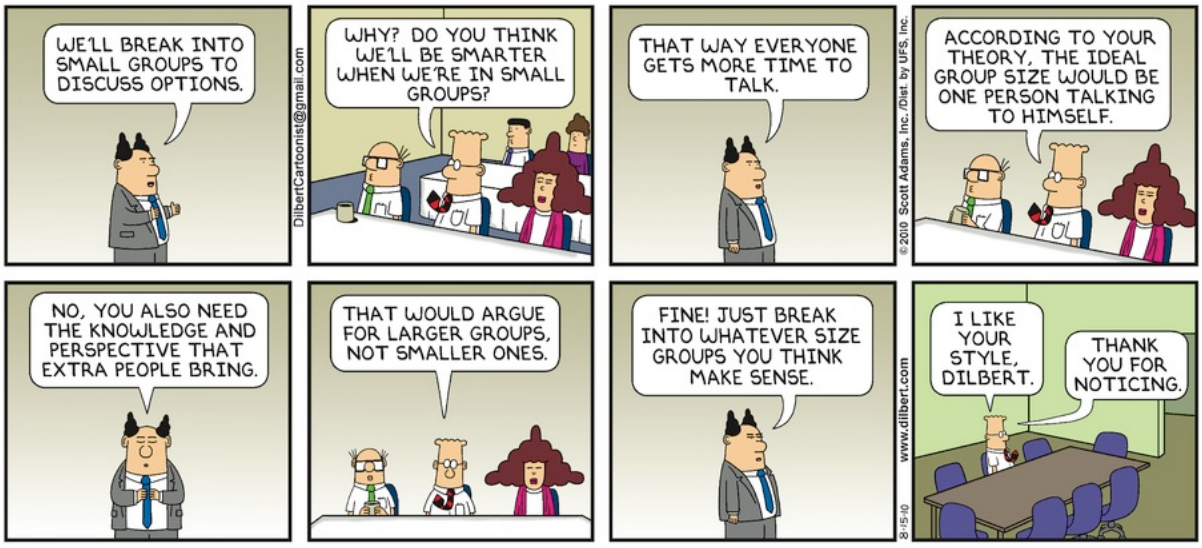
ignoring ideas from one or more teammates

c) Recorder:

writing incomplete answers or explanations

d) Reflector:

teammates answering questions on their own



Dilbert by Scott Adams. © Andrews McMeel Syndication. http://dilbert.com/strip/2010-08-15

Model 1 Strings

As we learned previously, Java has eight primitive types (int, double, etc.). All other types of data, like arrays, are called ***reference*** types, because **their value is a memory address**. In box-and-pointer memory diagrams, we use an arrow to reference other memory locations. The real power of Java comes not from its primitive types, but from its ***objects***: built-in reference types, and new, custom types that we create. The first built-in object we'll look at is `String`.

```
String name = "Rosie the Elephant";
// index help: 012345678901234567
char singleLetter = name.charAt(2);
String partOfString = name.substring(6, 9);
int numCharacters = name.length();
```

A `String` in Java represents a sequence of characters. This could be accomplished by a `char[]`, but String objects are much easier to work with thanks to built-in operations.

Questions (20 min) Start time:

2. Notice the pattern Java uses for primitive type names and reference type names. What is different about `String` compared to `int`, `double`, etc.? it is capitalized

3. Java variables can use up to eight bytes of memory. What does this tell us about the way Java will store the value `"Rosie the Elephant"` using the `name` variable?

The value is much larger than eight bytes, so it cannot be stored directly in the memory location for `name`. Instead, `name` will store a memory location for the String value.

4. Notice the Java syntax for method calls from an object:

```
[variable name].[methodName]([arguments]);
```

The `charAt`, `substring`, and `length` methods are in Java's `String` class. Predict the values:
`singleLetter`: `'s'` `partOfString`: `"the"` `numCharacters`: `18`

5. For built-in Java objects, Oracle provides thorough documentation about what they represent and how they can be used. The ways that your code can interact with Java's built-in code are collectively called an ***Application Programming Interface***, or API. Search the web for "Java 9 API String" to find the current Java documentation for Strings. (You should get a `docs.oracle.com` page.) Find the descriptions of the `charAt`, `substring`, and `length` methods.

a) Did reading the docs change your answers to #4? Why/why not?

Off-by-one errors are common in interpreting `charAt` and `substring` arguments. Notice that String indexing starts at 0, like for arrays, and the two-parameter version of substring ***excludes*** the end index.

b) The value of `name.substring(8)` is: `"e Elephant"`

c) Check your answers to #4 and #5 by running `StringDemo.java`.

6. Java uses different syntax to compare primitive types vs. reference types.

a) To check if two primitive type variables `x` and `y` are the same, the syntax is `x == y`

b) Suppose we have two String variables, `str1` and `str2`. What do you think will happen if we use the primitive equality check syntax to compare `str1` and `str2`? (***Hint:*** Recall #3.)

If we write `str1 == str2`, it will compare the memory addresses, not the Strings' contents, since that is what Java stores for reference type variables.

c) Revisit the Java String documentation. Based on the provided methods, what do you think is the correct way to check whether two Strings match?

We should use `str1.equals(str2)`, which compares their contents and is `true` if they match, or `false` otherwise. You might also find `contentEquals`, which is used for comparing against other String-like objects, and `equalsIgnoreCase`, which does what it sounds like it does.

7. Practice working with Strings by reviewing `StringProbs.java` and completing the unfinished method. Test `StringProbs.java` by running `StringProbsTest.java`.

8. (Optional) If time allows and you are already familiar with Java String objects, take a look at the Java API documentation for the `StringBuffer` and `StringBuilder` classes. Why do these classes exist, and when might you want to use them?

`StringBuffer` is like String, but mutable and thread-safe. We should consider using it when we have multiple threads. `StringBuilder` is a faster version of `StringBuffer` that is designed for use by a single thread. Since both are mutable, but a `String` is immutable, both can improve performance if your code does a lot of string modifications.

Model 2 ArrayLists

Arrays in Java have fixed sizes, but there is an alternative for when we want to store a list of values that can easily increase or decrease in size: `ArrayList` objects.

Example 1:

```
int[] nums = {10, 20, 15};
ArrayList<Integer> numsList = new ArrayList<>();
for (int i = 0; i < nums.length; i++) {
    numsList.add(nums[i]);
}
System.out.println(nums);
System.out.println(numsList);
```

Example 2:

```
String[] leagues = {"MLS", "MLB", "NHL", "NFL", "NBA"};
ArrayList<String> leaguesList = new ArrayList<>();
for (String league : leagues) {
    leaguesList.add(league);
}
System.out.println(leagues);
System.out.println(leaguesList);
leaguesList.remove(2);
leaguesList.add(3, "UFA");
System.out.println(leaguesList);
```

Predict the outputs of these examples, then run `ArrayListDemo.java` and compare.

Questions (20 min) Start time:

9. Look at Example 1. What features do you observe about `ArrayList` objects? How are they different from arrays?

ArrayList declarations use angle brackets "<>" to specify the type they contain. For integers, the type is `Integer` intead of `int`. Instantiating an ArrayList requires the `new` keyword, and we end with angle brackets followed by parentheses. The ArrayList class has an `add` method that appends a new value at the end of the list. Unlike arrays, their size can change. When printed, they display their actual contents, unlike arrays that display their memory addresses.

10. `Integer` is one of the `Number` classes that Java uses to "wrap" primitive types as reference types (read more). For now, the main takeaway is that ArrayLists always store objects, so if we want to store primitives in an ArrayList, we need to use the corresponding wrapper class. How would we declare and instantiate an ArrayList for real numbers, like [1.65, 2.32, 7.91]?

```
ArrayList<Double> reals = new ArrayList<>();
```

11. Look at Example 2. What additional ArrayList features do you notice? Search "Java 9 API ArrayList" and read the docs for these new methods.

We can remove an element from an ArrayList at a particular index. There is also a two-parameter version of the `add` method that lets us specify where to insert the new value.

12. Example 2 also shows a new type of for loop. This is called an ***enhanced*** for loop, which we read as "for each String league in leaguesArray". Notice we need to use the primitive wrapper class `Integer`. In what order does the enhanced for loop iterate through the array? When should we use an enhanced versus a standard for loop?

The enhanced for loop goes in increasing index order, starting at 0. We can use an enhanced for loop any time we only need the ***values*** one at a time, and not their ***indices*** also.

13. In Example 1, what happens if we try to expand `nums` with `nums[3] = 5;`? Try it out.

The program crashes with an `ArrayIndexOutOfBoundsException`. The index 3 is outside the fixed length of `nums`.

14. Consult the Java API docs for ArrayList. Working with `numsList` above, what method call should we use to…

a) retrieve the value at index 2? `numsList.get(2);`

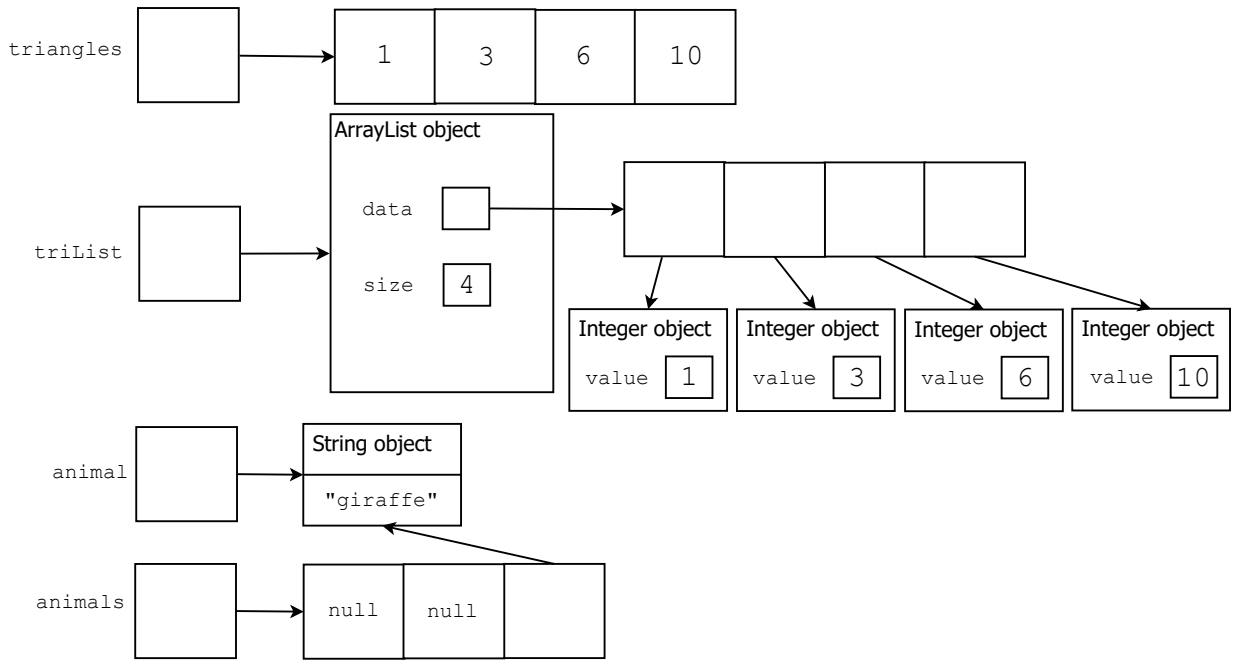
b) overwrite the value at index 1 with 17? `numsList.set(1, 17);`

c) retrieve the number of elements in the list? `numsList.size();`

Model 3 Memory Diagrams with Reference Types

```
int[] triangles = {1, 3, 6, 10};
ArrayList<Integer> triList = new ArrayList<>();
for (Integer tri : triangles) triList.add(tri);
String animal = "giraffe";
String[] animals = new String[3];
animals[animals.length - 1] = animal;
```

In box-and-pointer memory diagrams, we use arrows for reference types (i.e., objects). Each object instance gets its own box, which contains its data (a.k.a. ***fields***, or ***instance variables***).



Questions (15 min) **Start time:**

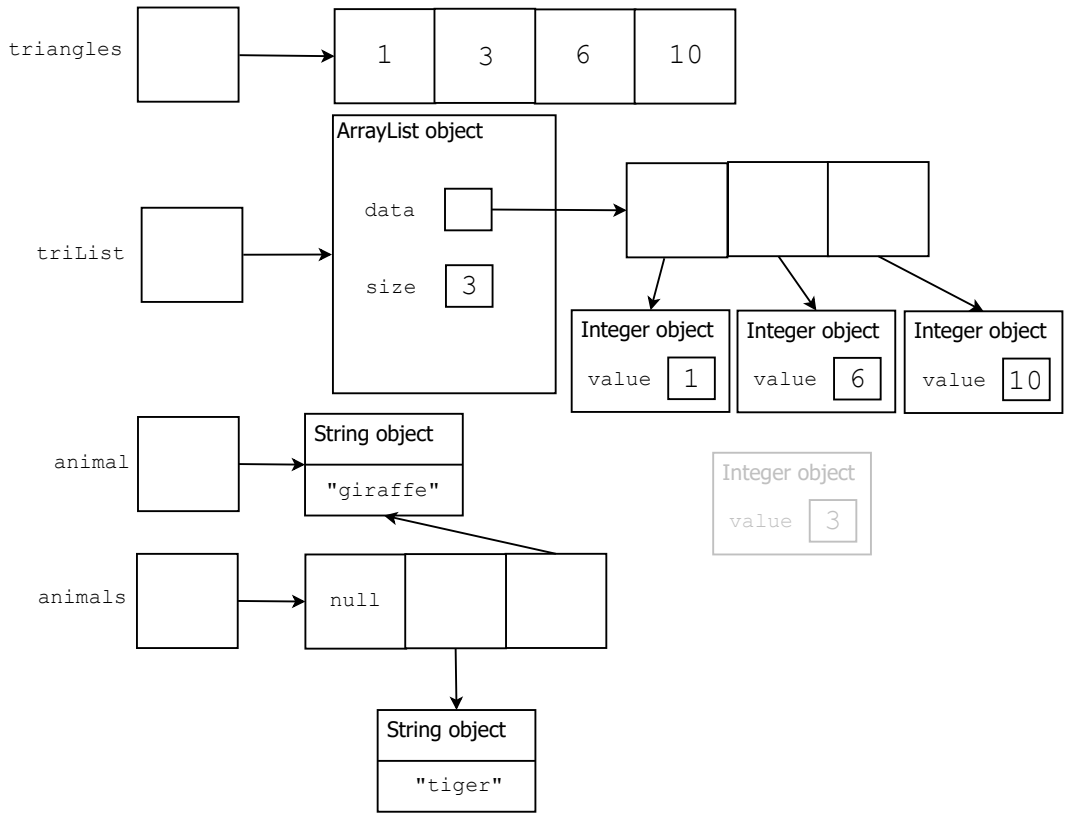
15. What differences do you notice between the memory diagrams for `triangles` and `triList`?

The array stores primitive values directly in adjacent memory locations. The ArrayList instead stores references to `Integer` objects, which contain the actual data. (The ArrayList object also stores its current size to help the `size()` method run fast.)

16. What is the default value for reference types in arrays and ArrayLists?

`null`, representing "nothing", the lack of a value

17. Suppose we add the lines `animals[1] = "tiger";` and `triList.remove(1);` at the end. Redraw the memory diagrams accordingly.



18. (Optional) The above step should leave you with an Integer object box floating off on its own. Hence, memory is still currently allocated to store this Integer, but there are no longer any references to it! Java will eventually clean this up in a process called ***garbage collection***. Search the web to learn more about garbage collection in Java, or start with this video.