

Week 5 - Class relationships, String class and 1D arrays

Week 5 - Description

This week we'll look at how to use classes to break a large problem into smaller problems.

We'll also start our discussion of class relationships. We'll touch on inheritance (IS-A relationship), which we'll come back to later, and we'll cover Aggregation, which is a HAS-A relationship where one class contains an object of another class.

We'll also dig deeper into classes and talk in general about classes and methods - such as the types of classes and methods we need for various purposes.

Then, we'll add to our arsenal by discussing some common methods of the String class and how they are used. This also serves to enhance our knowledge of using the Java API.

This week also deals with a programming structure called an array, which allows us to create several variables of the same data type, give them a common name, and process each variable (called an array element) in the same way, usually with a loop. We'll look at one dimensional arrays of primitive types this week - how to create and process them.

We will also look at special type of zip files used to hold .class files.

Classes for problem breakup

One of the reasons Object Oriented programming works is that it allows us to take a complex problem and break it up into smaller, more manageable pieces -> classes!

Usually there are two types of relationships among classes: IS-A and HAS-A. The upward arrows indicate inheritance (is-a relationship), the diamonds indicate an aggregation relationship (has-a relationship):

If we code all the logic for any application in main, there would be a **LOT** of code, and it would be repetitive and difficult to understand. By separating it into classes, and using inheritance and aggregation, we've been able to:

- Reuse a lot of the code in the classes higher on the hierarchy
- Separate the code into logical pieces so it's easier to understand
- Allow more people to code the project simultaneously.

Class Relationships – Aggregation

Classes in a program have various types of **relationships** to each other. The most common one is **Inheritance**, where one class inherits data and methods from another in an "IS-A" relationship. We'll come back to this one later on, and discuss it in detail. For

Week 5 - Class relationships, String class and 1D arrays

now, we'll cover another common relationship -- Aggregation. **Aggregation** is a "HAS-A" relationship, where one class contains objects of another class.

Aggregation

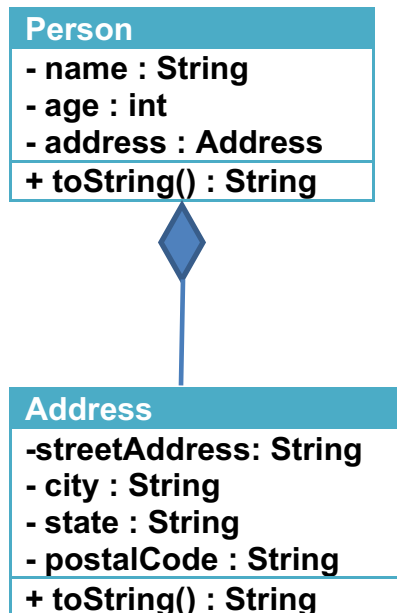
An aggregate object is one which contains other objects. You may also hear this called composition or association (there are distinctions between these, and you have covered it in first semester). A common example here is a Car, which **has an** Engine, several Doors, and a Driver.

Whereas the test for inheritance is "IS-A" (such as a Circle IS-A GeometricObject or a Cat IS-A Mammal), the test for aggregation is "HAS-A" - to see if there is a whole/part relationship between two classes. A synonym for this is "part-of".

In UML, an aggregation relationship is shown with a diamond. For our purposes, we won't worry about whether it is filled or not. For example:



Here's another example with a Person object that contains address objects:



To create this in your Person class (I'm assuming the Address class is coded elsewhere), you would code:

```
public class Person {  
    private String name;  
    private Address homeAddress;  
    ...  
}
```

Types of classes and methods

Types of Classes

So far, we've used Java API classes and discussed designing and building **classes of our own to model objects** in our applications.

Utility Classes

Sometimes, there are methods you need to build which do not fit into any particular class. You might have methods which:

- Are not related to the data in any particular class
- Are widely used in many classes throughout the program
- Perform some generally useful function that many classes will need

When you run into a situation like this, it is best to create a **utility class** to hold these methods. A utility class is simply a class created to contain useful methods which do not fit into any other class. In the Java libraries, we see an example of this with the Math class which contains a collection of methods to perform useful operations like square root and absolute value. The Math class satisfies the definition of a utility class since these math methods do not fit into any other class and they are widely applicable to a wide range of software that needs to use math.

Often, string handling is a good choice for a utility class. Many programs do a lot of string manipulation and the methods for this are used throughout the program and do not fit into any particular class. The solution is to create a class just to hold the string handling methods. Let's say we need a method to extract the extension from a file name and a method which will look at someone's name and return their middle initial, if they have one. We could combine these methods into a string utility class like this:

```
public class StringUtil {  
  
    public static String getFileExtension(String filename) {  
        String result = "";  
        int posn = filename.lastIndexOf('.');  
        if(posn >= 0) result = filename.substring(posn + 1);  
        return result;  
    }  
}
```

Week 5 - Class relationships, String class and 1D arrays

```
public static String getMiddleInitial(String name) {  
    // code to find middle initial  
}  
}
```

I have made the methods in the StringUtil class static. Since the class contains no data, there is no reason to actually create an instance of it. In reality, this class is just a container for methods. By making all the methods static, they can be easily accessed from other classes.

Logic Classes

Most of the classes you build represent some object in the real world – a bank account, an online shopping cart, a cell phone, etc. However, classes modeling real-world objects and utility classes do not make complete programs. A complete program requires the addition of classes to tie together all of the classes you already built. We've called these tester classes.

Types of Methods

We've also discussed different types of methods:

- get and set methods (aka accessors and modifiers) to control access to the data in the class.
- service methods that manipulate class data and provide a service within the class, for example, the getArea method in Circle class.
- static utility methods - as mentioned above.

Strings

A char variable stores **one** character. A different type, String, is used to store a sequence of characters. Notice that String starts with a capital letter. This is because it is not one of the 8 java primitive types, but it's a class (a reference type). The class is in java.lang, so we don't need to import it.

Remember that the **new** operation is used to initialize objects. The **String** type is an exception to the rule, it can be initialized either using the same syntax as a primitive type, **or** using the standard syntax for declaring objects.

We can declare variables of type String and assign values to them using the new operation:

```
String message = new String();  
message = "Hello World";
```

Or, we can declare them in the same way we do with primitive types:

Week 5 - Class relationships, String class and 1D arrays

```
String message = "Hello World";
```

Note that a String literal is enclosed in double quotation marks, as opposed to a char literal, which is in single quotes.

A String can be one character long, but it is still different from a char, and you cannot assign a String value to a char variable, or vice versa:

```
String message = "A"; // OK
char letter = 'A'; // OK
message = letter; // not allowed
letter = message; // not allowed
```

Two strings can be combined to create a third string. This is called *concatenation*, and the symbol for it is the plus sign: +. (Java knows enough to treat + as concatenation when strings are involved, and as addition when numbers are involved):

```
String string1 = "Hello, ";
String string2 = "World!";
String string3 = string1 + string2;
```

This creates a String saying "Hello, World!" and assigns it to the variable string3.

We can use the shorthand operator with Strings if we want to assign the concatenated string back into the original one:

```
String string1 = "Hello";
string1 += " World";
```

If we use the + operator with one operand of type String and another operand of a different data type, then both operands will be converted to Strings, and they will be concatenated to create a new String:

```
String string1 = "Number: ";
int number = 12345;
String string2 = string1 + number;
```

This creates a new String: "Number: 12345" and assigns it to the variable string2.

String Methods and the Java API

We're going to look at various methods for manipulating strings.

Information on the Java library classes is given in a document called the Application Programming Interface (API) specification, available online at:

<https://docs.oracle.com/en/java/javase/20/docs/api/java.base/java/lang/String.html>

The lower left frame of this web page gives you a list of all of the Java library classes. If you scroll down to String and click on it, the right-hand frame of the web page will show you information about that class. The API provides much more information than you need to know to use this class and its methods, so don't be intimidated by the length of the document, and don't feel obliged to read and understand everything in the API before you use a class and its methods.

Search for the String class. Near the top of the page there is text that says:

java.lang
Class String

[java.lang.Object](#)
└─ [java.lang.String](#)

"java.lang" is the package that this class belongs to.

Scrolling down the API page for String, we come to a section called "Method Summary". This provides information on all of the public methods of class String. The methods are listed in alphabetical order. The summary indicates whether or not the method is static, its return type, its name, its parameter list, and a one-sentence description of what the method does. By clicking on the method name, you will go to another part of the same page called "Method Detail". For some methods, this section provides only a small amount of additional information; for others it provides a great deal.

For example, String has a method called length. The Method Summary section of the API says:

| | |
|-----|--|
| int | length() Returns the length of this string. |
|-----|--|

If you click on the link, you are directed to the Method Detail section:

Method Detail

length

public int `length()`

Returns the length of this string. The length is equal to the number of [Unicode code units](#) in the string.

Specified by:

[length](#) in interface [CharSequence](#)

Returns:

the length of the sequence of characters represented by this object.

Week 5 - Class relationships, String class and 1D arrays

The API tells you that the method `length()` has a return type of `int`. The method has no parameters, so it takes no arguments when we call it. Remember, though, that parentheses are still required in the method call.

Armed with this information, we can now write a program that uses this method:

```
/* Snippet to demonstrate the use of String method length.
The snippet assumes we already have an input string s0 that contains data.
The code finds the length of the string, and outputs the result. */
```

```
int length0;
length0 = s0.length();
System.out.println(s0 + " has length " + length0);
```

Note that `length()` is a **method** (When we get to arrays, you'll see that an array object has a **variable** called `length`!)

The following snippet demonstrates some more string methods:

```
Assume we have a string s0 that contains data */
```

```
System.out.println(s0 + " starts with " + s0.charAt(0));
System.out.println(s0 + " ends with " + s0.charAt(s0.length() - 1));
System.out.println("Upper case: " + s0.toUpperCase());
System.out.println("Lower case: " + s0.toLowerCase());

/* Quotes added to trimmed string to show where it starts and ends */
System.out.println("Trimmed: \"" + s0.trim() + "\"");
System.out.println("Trimmed string starts with: " + s0.trim().charAt(0));
```

The API specifications for the `charAt` method say

`char` [charAt](#)(int index)
Returns the character at the specified index.

Here's the output assuming `s0` contains the string `Harry`:

```
Harry starts with H
Harry ends with y
Upper case: HARRY
Lower case: harry
Trimmed: "Harry"
Trimmed string starts with: H
```

The `charAt()` method is not static, so it must be called on an object of class `String`, not on the class as a whole. It returns a `char`. It takes one argument, an `int` that specifies which character of the string to return. The characters of a string are numbered from

Week 5 - Class relationships, String class and 1D arrays

zero to one less than the length of the string. For example, in the string "Hello", the character 'H' has number or index 0, 'e' has index 1, 'l' has index 2, 'l' has index 3, and 'o' has index 4. The length of the string is 5, but the maximum index is one less than this, or 4.

Assume we have two strings s0 and s1 that are already declared and contain a string of some sort:

```
System.out.println("Comparing s0 and s1 with equals gives " + s0.equals(s1));

System.out.println("Comparing s0 and s1 with equalsIgnoreCase gives "
    + s0.equalsIgnoreCase(s1));
```

Boolean [equals\(Object](#) anObject)
 Compares this string to the specified object.

boolean [equalsIgnoreCase\(String](#) anotherString)
 Compares this String to another String, ignoring case considerations.

Here's the output with s0 = "Harry" and s1 = "Bill":

```
Comparing s0 and s1 with equals gives false
Comparing s0 and s1 with equalsIgnoreCase gives false
```

And here's the output with s0 = "Harry" and s1 = "harry":

```
Comparing s0 and s1 with equals gives false
Comparing s0 and s1 with equalsIgnoreCase gives true
```

It is not a syntax error to write

```
if (s0 == s1)
```

but this operator compares the addresses stored in the variables s0 and s1 to test whether they both refer to the same object. If they refer to two different string objects that have the same contents, the == comparison will return false.

Usually, we don't want to test whether two string variables refer to the same object; we want to test whether two string objects have the same content. We do this not with == but with the equals method, as shown above. This method is case sensitive; there is also an equalsIgnoreCase method.

Both of these methods have a return type of Boolean.

String toLowerCase(): Converts all of the characters in this String to lower case.

String toUpperCase(): Converts all of the characters in this String to upper case.

The above methods convert "Hello" to "hello" and "HELLO", respectively. **Note that they do not alter the contents of the original string object on which they are**

Week 5 - Class relationships, String class and 1D arrays

called. This object continues to exist, unchanged. The methods create a **new** string object and return it.

In fact, once you have created a string object, you can never change its contents. String objects are *immutable*. You can, however, assign different string objects to the same string variable at different times in a program. Consider the two code fragments below:

```
String s1 = "Hello";  
String s2 = s1.toUpperCase();
```

After this code executes, s1 refers to the string "Hello" and s2 to the string "HELLO".

```
String s3 = "Good-bye";  
s3 = s3.toUpperCase();
```

After this code executes, s3 refers to the string "GOOD-BYE". We no longer have any variable referring to the string "Good-bye", so it has become garbage.

String trim(): Returns a copy of the string, with leading and trailing whitespace omitted. For example, it converts " Hello World " to "Hello World" by removing the spaces before the first non-blank character and after the last non-blank character. It does not affect whitespace in the middle of a string.

Often we want to know more than this; we want to know which string comes first in alphabetical order. We can find this using the method.

int compareTo (String anotherString) : Compares two strings lexicographically.

s0.compareTo(s1) returns an int value that is negative if s0 precedes s1, zero if the two strings are equal, and positive if s0 follows s1. The exact numerical value of the return value does have some significance, but usually we only want to know if it is negative, zero or positive.

To use compareTo, we need to understand how this method decides which of two strings precedes the other. It begins by comparing the first character of each string. If they are different, then it compares the Unicode (ASCII) code for these characters, and the one with the smaller value precedes the other. For example, "Bad" precedes "Cad", because 'B' has ASCII code 66 while 'C' has ASCII code 67.

All upper case letters have smaller ASCII code than all lower case letters, so compareTo says that "Zebra" precedes "aardvark", since 'Z' has a smaller ASCII code than 'a'. There is a compareToIgnoreCase method that compares two strings as if both were all lower case.

If both strings have the same first character, the compareTo method compares the second character of each string. If they are different, it decides which string precedes the other, as above. If they are the same, it goes on to the next character, and so on, until it finds a pair of characters that are different, or until it reaches the end of one of the strings. If it reaches the end of one string but not the other, then the shorter string precedes the longer one. For example, "Hell" precedes "Hello". If it reaches the end of

both strings at the same time without finding any pairs of characters that are different, then the strings must be equal, so the method returns zero.

Arrays

An array is a data structure for collecting and organizing variables or objects of the same type into a group.

The individual items in an array are its elements. Arrays are often processed using a for loop which processes each element in turn. A variable called an index is used to indicate which element the code is referring to. The size (number of elements) of an array is specified when it is declared, and the size doesn't change during the execution of the program (there are other more flexible data structures we'll learn about next semester).

Arrays are objects

Java arrays are objects, even if they are arrays of primitive types like integers.

Declaring array objects

```
dataType[] variableName;
```

The square brackets indicate that this is an array variable. Specific examples are:

```
int[] studentMarks;
```

```
String[] employeeNames;
```

`studentMarks` is an object like those we declare when working with classes. Note that its data type is **not** `int`. It is `int[]` (read as “array of ints”). `int` and `int[]` are two different data types, and we cannot use the variable `studentMarks` in the same way we would use an `int` variable. The value we will store in this variable is the address of an array object. So far, however, we have not yet **created** the array object, all we've done is **declare** it.

Creating the array object

We create the array object by writing an expression like the following:

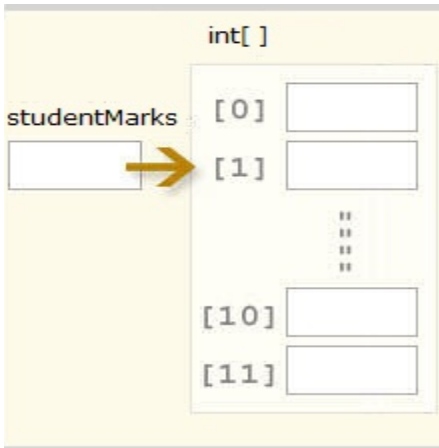
```
new int[12];
```

This allocates space in memory for an array object. Within this object, there are 12 variables, each of type `int`. These are called the *elements* of the array.

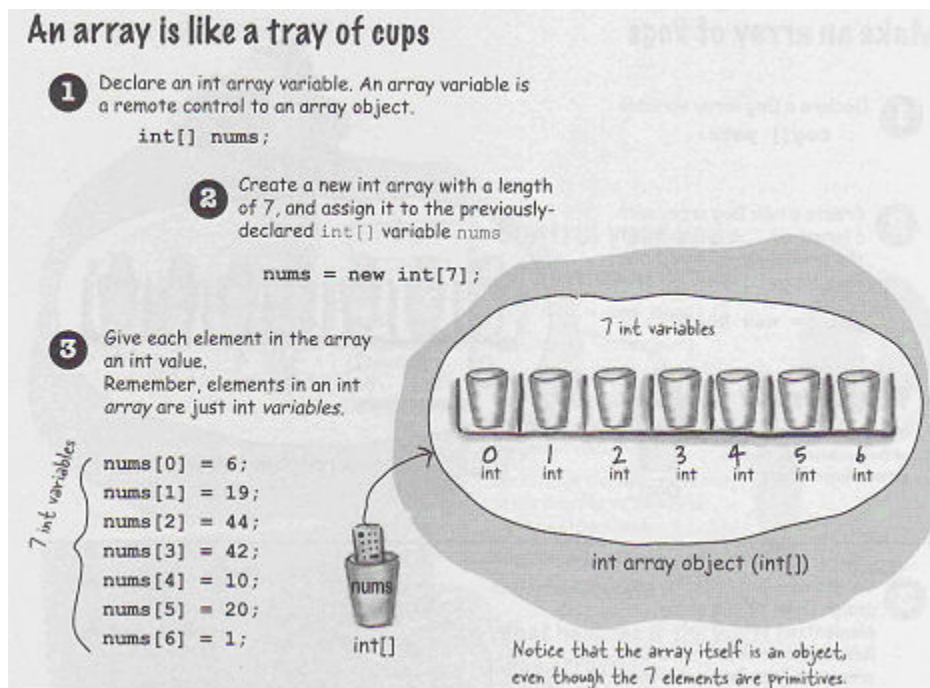
In order to use this array object later in our program, we need to assign it to a variable when we create it, so we would put the above expression on the right-hand side of an assignment statement:

```
int[] studentMarks = new int[12];
```

Week 5 - Class relationships, String class and 1D arrays



Here is another way to look at arrays:



Array elements

We refer to each of the array elements by the variable name, followed by square brackets containing an integer, for example `studentMarks[2]`. The integer in the brackets is called the *index* (or the *subscript*) of the array element. In Java, array indices always begin at 0 and go up to **one less** than the array length. In our example, there are 12 elements in our array, and the array indices range from 0 to 11 inclusive, so the array elements are `studentMarks[0]`, `studentMarks[1]`, `studentMarks[2]`, ... through `studentMarks[11]`.

Although the variable `studentMarks` is not of type `int`, each of the elements `studentMarks[0]`, `studentMarks[1]`, **are** of type `int`, and can be used in exactly the same

Week 5 - Class relationships, String class and 1D arrays

way as we use any int variable. We can use them on the left hand side of an assignment operator:

```
studentMarks[0] = 1234;
```

on the right hand side of an assignment:

```
studentMarks[1] = studentMarks[0];
```

in an arithmetic expression:

```
studentMarks[2] = studentMarks[1] + 5;
```

in a boolean comparison:

```
if (studentMarks[2] > 1500)
```

as a method argument:

```
System.out.println(studentMarks[2]);
```

and anywhere else that you can use an int variable.

Initializing array elements

We've already declared fields in our classes, and we know that they are initialized automatically. Numeric types (byte, short, int, long, double) are initialized to zero. Type boolean is initialized to false. Type char is initialized to the character with ASCII code zero (not a printable character). Class types (objects) are initialized to null. Array elements are also examples of fields, so they are automatically initialized based on their data types.

If you want to initialize array elements to specific values, rather than to their default values, you can create an array object and initialize its elements in one step by putting the values in a list inside curly braces, as shown below:

```
int[] squares = {0, 1, 4, 9, 16, 25, 36, 49, 64};
```

```
char[] vowels = {'a', 'e', 'i', 'o', 'u'};
```

The first line of code above creates an array of ints of length 9 and initializes its elements to the values given in the list. So, for example, `squares[5]` has the value 25. Similarly, the second line of code above creates an array of chars of length 5 and initializes its elements. You can only use this syntax if you declare a variable, and create and initialize the array in the same statement. If you try to write:

```
int[] squares;
```

```
squares = {0, 1, 4, 9, 16, 25, 36, 49, 64}; //the program will give you a compile error.
```


Creating JAR files

Netbeans let us create a special type of zip files called as JAR file. This is used to hold .class files. To create JAR file –

- Create a new project.



Week 5 - Class relationships, String class and 1D arrays

- Select Java Class Library.
- Write code statements.
- Click the hammer  to build the jar.

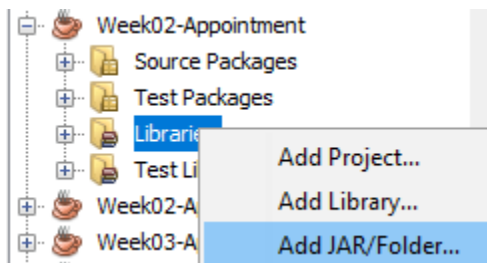
- The jar file goes in project's dist folder.

Some jar files act as libraries of reusable classes. These classes will be imported into other projects and used. They were never intended to be executed on their own, so they do not need a class with main in it. The project which uses them will supply main method.

Some jar files contain an entire program. You can package an entire program in a jar. In this case, one class contains main method. This creates a jar file which can be executed and provides a handy way to package and distribute a program.

Using a Jar in a project

Right click project libraries, click Add JAR/Folder and select your Jar or folder where your .class files are.



Object's equals method

The Object class is the ultimate ancestor. Every class in Java extends Object. However, you never have to write:

```
public class BankAccount extends Object
```

The equals method in the Object class tests whether one object is considered equal to another. As implemented in the Object class, it determines whether two object references are identical.

Any class which will be compared should override equals method. This method says objects are equal if at same memory address. Like toString(), equals() should be overridden to compare values in data members.