STRINGS REITERATION

OUTLINE

1. STRING
2. STRINGBUILDER
3. EQUALITY

1) STRING

- A string is basically a sequence of characters.
  - **`String name = "Max"`**
  - As we've learned this is an example of a reference type.
- We also learned that reference types are created using the `new` keyword.
- Wait a minute …
- Something is missing from the previous example:
- It doesn't have `new` in it!
- In Java, these two snippets both create a String:
  - `String name = "Max";`
  - `String name = new String("Max");`
- Both give us a reference variable of type name pointing to the String object "Max".
- They are subtly different, as we'll see in the section "String Pool," later.
- **For now, just remember that the String class is special and doesn't need to be instantiated with `new`.**

1) STRING
1.1) CONCATENATION

- Until now we've learned how to add numbers.
- 1 + 2 is clearly 3.
- But what is "1" + "2“ ?. It's actually "12" because Java combines the two String objects.
- Placing one String before the other String and combining them together is called string **concatenation**.
- There aren't a lot of rules to know for this, but we have to know them well:
  1. If both operands are numeric, + means numeric addition.
  2. If either operand is a String, + means concatenation.
  3. The expression is evaluated left to right.

```
System.out.println(1 + 2); // 3
System.out.println("a" + "b"); // ab
System.out.println("a" + "b" + 3); // ab3
System.out.println(1 + 2 + "c"); // 3c
```

- The first example uses the first rule. Both operands are numbers, so we use normal addition.
- The second example is simple string concatenation, described in the second rule. The quotes for the String are only used in code, they aren't outputted.
- The third example combines both the second and third rules. Since we start on the left, Java figures out what "a" + "b" evaluates to. You already know that one: it's "ab". Then Java looks at the remaining expression of "ab" + 3. The second rule tells us to concatenate since one of the operands is a String.

1) STRING
1.1) CONCATENATION

- In the fourth example, we start with the third rule, which tells us to consider 1 + 2.
- Both operands are numeric, so the first rule tells us the answer is 3.
- Then we have 3 + "c", which uses the second rule to give us "3c".
- Another example:

```
int three = 3;
String four = "4";
System.out.println(1 + 2 + three + four);
```

- In this example, we start with the third rule, which tells us to consider 1 + 2.
- The first rule gives us 3.
- Next we have 3 + **three**.
- Since **three** is of type **int**, we still use the first rule, giving us 6.
- Next we have 6 + **four**.
- Since **four** is of type String, we switch to the second rule and get a final answer of "64".
- There is only one more thing to know about concatenation, but it is an easy one.
- In this example, we just have to remember what += does.
- `s += "2"` means the same thing as `s = s + "2"`.

```
String s = "1"; // s currently holds "1"
s += "2"; // s currently holds "12"
s += 3; // s currently holds "123"
System.out.println(s); // 123
```

1) STRING
1.2) IMMUTABILITY

- Once a String object is created, **it is not allowed to change**.
- It cannot be made larger or smaller, and we cannot change one of the characters inside it.
- We can think of a string as a storage box we have perfectly full and whose sides can't bulge.
- There's no way to add objects, nor can we replace objects without disturbing the entire arrangement.
- The trade-off for the optimal packing is zero flexibility.
- **Mutable** is another word for changeable.
- Immutable is the opposite - an object that can't be changed once it's created.
- We need to know that **String is immutable**.

```
class Mutable {
    private String s;
    // Setter makes it mutable
    public void setS(String newS) {
        s = newS;
    }
    public String getS() {
```

```
final class Immutable {
    private String s = "test";
    public String getS() {
        return s;
```

- Immutable only has a getter.
- There's no way to change the value of **s** once it's set.
- Mutable has a setter as well.
- This allows the reference s to change to point to a different String later.

1) STRING
1.2) IMMUTABILITY

- We learned that + is used to do String concatenation in Java.
- There's another way, which isn't used much on real projects but is great for tricking people on the exam.
- Let's see what this code prints.
  - The answer is "12"
  - The trick is **to not forget that the String class is immutable.**

```
String s1 = "1";
String s2 = s1.concat("2");
s2.concat("3");
System.out.println(s2);
```

1) STRING
1.3) THE STRING POOL

- Since strings are everywhere in Java, they use up a lot of memory.
- In some production applications, they can use up 25%-40% of the memory in the entire program.
- Java realizes that many strings repeat in the program and solves this issue by reusing common ones.
- The string pool, also known as the intern pool, is a location in the Java virtual machine (JVM) that collects all these strings.
- The string pool contains literal values that appear in our program.
- For example, “Hello” is a literal and therefore goes into the string pool.
- `myObject.toString()` is a string but not a literal, so it does not go into the string pool.

- **Strings not in the string pool are garbage collected just like any other object.**
- Remember back when we said these two lines are subtly different ?
  - `String name = "Max";`
  - `String name = new String("Max");`
- The first one says to use the string pool normally.
- The second says “No, JVM. I really don’t want you to use the string pool. Please create a new object for me even though it is less efficient.”

2) STRINGBUILDER

- A small program can create a lot of String objects very quickly.
- For example, let's look at the following code.

```
String alpha = "";
for (char current = 'a'; current <= 'z'; current++)
    alpha += current;
System.out.println(alpha);
```

- The empty String on line 10 is instantiated, and then line 12 appends an "a".
- However, **because the String object is immutable**, a new String object is assigned to alpha and the "" object becomes eligible for garbage collection.
- The next time through the loop, alpha is assigned a new String object, "ab", and the "a" object becomes eligible for garbage collection.
- The next iteration assigns alpha to "abc" and the "ab" object becomes eligible for garbage collection, and so on.
- This sequence of events continues, and after 26 iterations through the loop, a total of 27 literals are created, most of which are immediately eligible for garbage collection.
- This is very inefficient.
- Luckily, Java has a solution.
- The StringBuilder class creates a String without storing all those interim String values.
- **Unlike the String class, StringBuilder is not immutable.**

2) STRINGBUILDER

- On line 6, a new StringBuilder object is instantiated.
- The call to append() on line 8 adds a character to the StringBuilder object each time through the for loop and appends the value of current to the end of alpha.
- This code reuses the same StringBuilder without creating an interim String each time.

```
6 StringBuilder alpha = new StringBuilder();
7   for (char current = 'a'; current <= 'z'; current++)
8       alpha.append(current);
9 System.out.println(alpha);
```

2) STRINGBUILDER
2.1) MUTABILITY AND CHAINING

- When we chained String method calls, the result was a new String with the answer.
- Chaining StringBuilder objects doesn't work this way.
- Instead, the StringBuilder **changes its own state and returns a reference to itself !**
- Line 7 adds text to the end of **sb**.
- It also returns a reference to **sb**, which is ignored.
- Line 8 also adds text to the end of **sb** and returns a reference to **sb**.
- This time the reference is stored in **same** which means **sb** and **same** point to the exact same object and would print out the same value.

```
6 StringBuilder sb = new StringBuilder("start");
7 sb.append("+middle"); // sb = "start+middle"
8 StringBuilder same = sb.append("+end"); // "start+middle+end"
```

2) STRINGBUILDER
2.1) MUTABILITY AND CHAINING

- Let’s see what this example prints.
- There’s only one StringBuilder object here.
- We know that because **`new StringBuilder()`** was called only once.
- On line 7, there are two variables referring to that object, which has a value of "abcde".
- On line 8, those two variables are still referring to that same object, which now has a value of "abcdefg".
- Incidentally, the assignment back to **`b`** does absolutely nothing.
- **`b`** is already pointing to that StringBuilder.
- They both are print abcdefg.

```
6  StringBuilder a = new StringBuilder("abc");
7  StringBuilder b = a.append("de");
8  b = b.append("f").append("g");
9  System.out.println("a=" + a);
10 System.out.println("b=" + b);
```

2) STRINGBUILDER
2.2) CREATING A STRINGBUILDER

- There are three ways to construct a StringBuilder:
  - `StringBuilder sb1 = new StringBuilder();`
  - `StringBuilder sb2 = new StringBuilder("animal");`
  - `StringBuilder sb3 = new StringBuilder(10);`
- The first says to create a StringBuilder containing an empty sequence of characters and assign sb1 to point to it.
- The second says to create a StringBuilder containing a specific value and assign sb2 to point to it.
- For the first two, it tells Java to manage the implementation details.
- The final example tells Java that we have some idea of how big the eventual value will be and would like the StringBuilder to reserve a certain number of slots for characters.

2) STRINGBUILDER
2.2) CREATING A STRINGBUILDER

- The behind-the-scenes process of how objects are stored may help us better understand and remember StringBuilder.
- Size is the number of characters currently in the sequence, and capacity is the number of characters the sequence can currently hold.
- Since a String is immutable, the size and capacity are the same.
- The number of characters appearing in the String is both the size and capacity.
- For StringBuilder, Java knows the size is likely to change as the object is used.
- When StringBuilder is constructed, it may start at the default capacity (which happens to be 16) or one of the programmer's choosing.

- In the example, we request a capacity of 5.
- At this point, the size is 0 since no characters have been added yet, but we have space for 5.
- **`StringBuilder sb = new StringBuilder(5);`**

0	1	2	3	4

- Next we add four characters. At this point, the size is 4 since four slots are taken. The capacity is still 5.
- **`sb.append("anim");`**

a	n	i	m	
0	1	2	3	4

2) STRINGBUILDER
2.2) CREATING A STRINGBUILDER

- Then we add three more characters.
- The size is now 7 since we have used up seven slots.
- Because the capacity wasn't large enough to store seven characters, Java automatically increased it for us.
- **`sb.append("als");`**

a	n	i	m	a	l	s		
0	1	2	3	4	5	6	7	…

3) EQUALITY

- In the previous chapters we learned how to use == to compare numbers.
- == is also used to check if object references refer to the same object.

```
StringBuilder one = new StringBuilder();
StringBuilder two = new StringBuilder();
StringBuilder three = one.append("a");
System.out.println(one == two); // false
System.out.println(one == three); // true
```

- Since this example isn't dealing with primitives, we know to look for whether the references are referring to the same object.
- **one** and **two** are both completely separate StringBuilders, giving us two objects.
- Therefore, the first print statement gives us false.
- **three** is more interesting.
- Remember how StringBuilder methods like to return the current reference for chaining?
- This means **one** and **three** both point to the same object and the second print statement gives us **true**.

3) EQUALITY

- Let's now visit the more complex and confusing scenario, String equality, made so in part because of the way the JVM reuses String literals:

```
String x = "Hello World";
String y = "Hello World";
System.out.println(x == y); // true
```

- Remember that Strings are immutable and literals are pooled.
- The JVM created only one literal in memory.
- x and y both point to the same location in memory; therefore, the statement outputs `true`.

- We can even force the issue by creating a new String:

```
String x = new String("Hello World");
String y = "Hello World";
System.out.println(x == y); // false
```

- Since we have specifically requested a different String object, the pooled value isn't shared.