Chapitre 6

Strings, Arrays, ArrayLists, Dates, and Lambdas

Business Training

Content

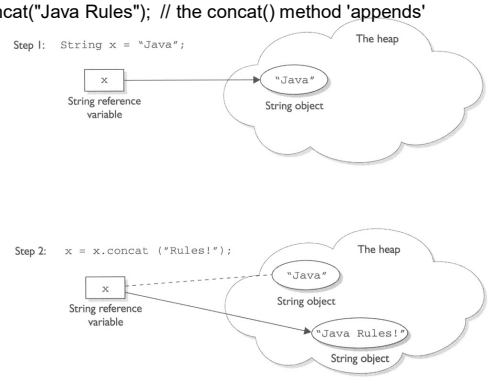
- **Using String and StringBuilder**
  (OCA Objectives 9.2 and 9.1)
- **Working with Calendar Data**
  (OCA Objective 9.3)
- **Using Arrays**
  (OCA Objectives 4.1 and 4.2)
- **Using ArrayLists and Wrappers**
  (OCA Objectives 9.4 and 2.5)
- **Advanced Encapsulation**
  (OCA Objective 6.5)
- **Using Simple Lambdas**
  (OCA Objective 9.5)

Using String and StringBuilder
(OCA Objectives 9.2 and 9.1)

The String Class

- **Strings Are Immutable Objects**
  - Once you have assigned a String a value, that value can never change—it's immutable, frozen solid, won't budge, fini, done.
    - The good news is that while the String object is immutable, its reference variable is not :

```
s = s.concat("Java Rules");  // the concat() method 'appends'
```



Notice in step 2 that there is no valid reference to the "Java" String; that object has been "abandoned," and a new object created.

The String Class

- **Let's Lake this example a little further:**

```
String x = "Java";
x = x.concat(" Rules!");
System.out.println("x = " + x);    // the output is: Java Rules!
x.toLowerCase();                   // no assignment, create a new, abandoned String
System.out.println("x = " + x);    // no assignment, the output is still: x = Java Rules!
x = x.toLowerCase();               // create a new String, assigned to x
System.out.println("x = " + x);    // the assignment causes the output: x = java rules!
```

- **Let's Lake this example a little further:**

```
String s1 = "spring ";
String s2 = s1 + "summer ";
s1.concat("fall ") ;
s2.concat(s1);
s1 += "winter ";
System.out.println(s1 + " " + s2);
```

The result of this code fragment is "spring winter spring summer".

The String Class is Immuable !

- **Important Facts About Strings and Memory**
  - To make Java more memory efficient, the JVM sets aside a special area of memory called the "String constant pool." When the compiler encounters a String literal, it checks the pool to seek if an identical String already exists. If a match is found, the reference to the new literal is directed to the existing String, and no new String literal object is created.
    - Now we can start to see **why making String objects immutable is such a good idea**. If several reference variables refer to the same String without even knowing it, it would be very bad if any of them could change the String's value.
    - what if someone overrides the String class functionality; couldn't that cause problems in the pool?" That's one of the main reasons that the String class is marked **final**.
- **Creating New Strings**
  - **String s = "abc";** // creates one String object and one reference variable.
    - **"abc" will go in the pool and s will refer to it.**
  - **String s = new String("abc");** // creates two objects, and one reference variable
    - **Java will create a new String object in normal (non-pool) memory, and s will refer to it. In addition, the literal "abc" will be placed in the pool.**

Important Methods in the String Class

- **The following methods arc some of the more commonly used methods in the String class, and also the ones that you're most likely to encounter on the exam.**
  - **charAt()** Returns the character located at the specified index
  - **concat()** Appends one String to the end of another ("+" also works)
  - **equalsIgnoreCase()** Determines the equality of two Strings, ignoring case
  - **length()** Returns the number of characters in a String
  - **replace()** Replaces occurrences of a character with a new character
  - **substring()** Returns a part of a String
  - **toLowerCase()** Returns a String with uppercase characters converted
  - **toString()** Returns the value of a String
  - **toUpperCase()** Returns a String with lowercase characters converted
  - **trim()** Removes whitespace from the ends of a String

The StringBuffer and StringBuilder Classes

- **The java.lang.StringBuffer and java.lang.StringBuilder classes should be used when you have to make a lot of modifications to strings of characters.**
- **StringBuffer vs. StringBuilder**
  - The StringBuilder class was added in Java 5. It has exactly the same API as the StringBuffer class, except StringBuilder is not thread safe.
    - In other words, its methods are not synchronized. It will run faster
  - So apart from synchronization, anything we say about StringBuilder's methods holds true for StringBuffer's methods, and vice versa.
- **Using StringBuilder and StringBuffer**

```
StringBuffer sb = new StringBuffer("abc");
sb.append("def");
System.out.println("sb = " + sb);     // output is "sb = abcdef"
```

  - The method calls can be chained to each other :

```
StringBuilder sb = new StringBuilder("abc");
sb.append("def").reverse().insert(3, "---");
System.out.println( sb );            // output is  "fed --- cba"
```

Important Methods in the StringBuffer and StringBuilder Classes

- **charAt()** Returns the character located at the specified index
- **concat()** Appends one string to the end of another (+ also works)
- **equalsIgnoreCase**() Determines the equality of two strings, ignoring case
- **length()** Returns the number of characters in a string
- **replace()** Replaces occurrences of a character with a new character
- **substring()** Returns a part of a string
- **toLowerCase()** Returns a string, with uppercase characters converted to lowercase
- **toString()** Returns the value of a string
- **toUpperCase()** Returns a string, with lowercase characters converted to uppercase
- **trim()** Removes whitespace from both ends of a string

Important Facts About Strings and Memory

- **To make Java more memory efficient, the JVM sets aside a special area of memory called the String constant pool.**
  - When the compiler encounters a String literal, it checks the pool to see if an identical String already exists. If a match is found, the reference to the new literal is directed to the existing String, and no new String literal object is created.
  - If a match is found, the reference to the new literal is directed to the existing String, and no new String literal object is created.
- **Creating New Strings**

Working with Calendar Data
(OCA Objectives 9.3)

Calendar-related classes

- Here's a summary of the five calendar-related classes we'll study, plus an interface that looms large:
  - **java.time.LocalDateTime** This class is used to create immutable objects, each of which represents a specific date and time. Additionally, this class provides methods that can manipulate the values of the date/time objects created and assign them to new immutable objects.
  - **java.time.LocalDate** This class is used to create immutable objects, each of which represents a specific date.
  - **java.time.LocalTime** This class is used to create immutable objects, each of which represents a specific time
  - **java.time.format.DateTimeFormatter** This class is used by the classes just described to format date/time objects for output and to parse input strings and convert them to date/time objects.
  - **java.time.Period** This class is used to create immutable objects that represent a period of time, for example, "one year, two months, and three days
    - If you want to represent chunks of time in increments finer than a day (e.g., hours and minutes), you can use the java.time.Duration class, but Duration is not on the exam.
  - **java.time.temporal.TemporalAmount** This interface is implemented by the Period class.

Immutability and Factories

- There are a couple of recurring themes in the previous definitions.
  - First, notice that most of the calendar-related objects you'll create **are immutable**. Just like String objects!

```
LocalDate date1 = LocalDate.of(2017, 1, 31);
    Period period1 = Period.ofMonths(1);
    System.out.println(date1);
    date1.plus(period1);                    // new value is lost
    System.out.println(date1);
    LocalDate date2 = date1.plus(period1);  // new value is captured
    System.out.println(date2);
```

  - The next thing to notice in the previous code listing is that we never used the keyword new in the code. We didn't directly invoke a constructor.
    - Instead, for all these classes, you invoke a public static method in the class to create a new object. As you go further into your studies of OO design, you'll come across the phrases "**factory pattern**," "**factory methods**," and "**factory classes**".

Formatting Dates and Times

- For the exam, you should know the following two-step process for creating Strings that represent well-formatted calendar objects:
  1. Use formatters and patterns from the HUGE lists provided in the **DateTimeFormatter** class to create a DataTimeFormatter object.
  2. In the LocalDate, LocalDateTime, and LocalTime classes, use the **format()** method with the DateTimeFormatter object as the argument to create a well-formed String—or use the DateTimeFormatter.format() method with a calendar argument to create a well-formed String.

```
import java.time.*;
import java.time.format.*;
public class NiceDates {
  public static void main(String[] args) {
    DateTimeFormatter f1 =
              DateTimeFormatter.ofPattern("MMM dd, yyyy");
    DateTimeFormatter f2 =
              DateTimeFormatter.ofPattern("E MMM dd, yyyy G");
    DateTimeFormatter tf1 =
              DateTimeFormatter.ofPattern("k:m:s A a");

    LocalDate d = LocalDate.now();
    String s = d.format(f1);          // thus proving that the format()
                                      // method makes String objects
    System.out.println(s);
    System.out.println(d.format(f2));

    LocalTime t = LocalTime.now();
    System.out.println(t.format(tf1));
  }
}
```

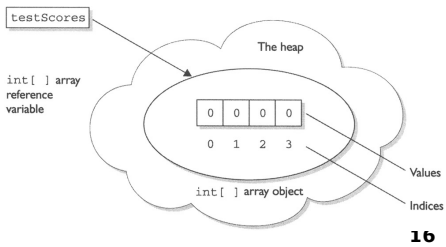
The result is :

```
Jan 14, 2017
Sat Jan 14, 2017 AD
14:17:9 51429958 PM
```

Using Arrays
(OCA Objectives 4.1 and 4.2)

Using Arrays

- **Declaring an Array**
  - Declaring an array of primitives:
    - int[] key; // brackets before name (recommended)
    - int key []; // brackets after name (legal but less readable)
  - Declaring an array of object references:
    - Thread[] threads; // Recommended
    - Thread threads[]; // Legal but less readable
- **Constructing an Array**
  - int [] testScores; // Declares the array of ints
  - testScores = new int[4]; // constructs an array and assigns it testScores variable
- **Initializing an Array**
  - testScores = new int[] {4,7,2};
  - int[] dots = {6,7,8};
  - dots[0]=6; dots[1]=7; dots[2]=8;



Using ArrayLists and Wrappers
(OCA Objectives 9.4 and 2.5)

When to Use ArrayLists

- **We've already talked about arrays. Arrays seem useful and pretty darned flexible. So why do we need more functionality than arrays provide? Consider these two situations:**
  - You need to be able to increase and decrease the size of your list of things.
  - The order of things in your list is important and might change.
- **Technically speaking, ArrayLists hold only object references, not actual objects and not primitives. If you see code like this,**

  myArrayList.add(7);
  - What's really happening is the int is being autoboxed (converted) into an Integer object and then added to the *ArrayList*.
- **Note that ArrayLists can have duplicates**

ArrayList Methods in Action

- **The following methods are some of the more commonly used methods in the ArrayList class**
  - add(element) Adds this element to the end of the ArrayList
  - add(index, element) Adds this element at the index point and shifts the remaining elements back (for example, what was at index is now at index+1)
  - clear() Removes all the elements from the ArrayList
  - **boolean contains(element)** Returns whether the element is in the list
  - Object get(index) Returns the Object located at index
  - int indexOf(Object) Returns the (int) location of the element or -1 if the Object is not found
  - remove(index) Removes the element at that index and shifts later elements toward the beginning one space
  - remove(Object) Removes the first occurrence of the Object and shifts later elements toward the beginning one space
  - int size() Returns the number of elements in the ArrayList

Boxing, ==, and equals()

- Two objects are equal if they are of the same type and have the same value.
  - It shouldn't be surprising that

```
Integer i1 = 1000;
Integer i2 = 1000;
if(i1 != i2) System.out.println("different objects");
if(i1.equals(i2)) System.out.println("meaningfully equal");
```

  - produces the output

```
different objects
meaningfully equal
```

  - It's just two wrapper objects happen to have the same value. Because they have the same int value, the equals() method considers them to be "meaningfully equivalent" and, therefore, returns true. How about this one?

```
Integer i3 = 10;
Integer i4 = 10;
if(i3 == i4) System.out.println("same object");
if(i3.equals(i4)) System.out.println("meaningfully equal");
```

  - This example produces the output:

```
same object
meaningfully equal
```

The Java 7 "Diamond" Syntax

- **Declaring type-safe ArrayLists like this:**

```
ArrayList<String> stuff = new ArrayList<String>();
ArrayList<Dog> myDogs = new ArrayList<Dog>();
```

- **Notice that the type parameters are duplicated in these declarations. As of Java 7, these declarations could be simplified to:**

```
ArrayList<String> stuff = new ArrayList<>();
ArrayList<Dog> myDogs = new ArrayList<>();
```

  - Notice that in the simpler Java 7 declarations, the right side of the declaration included the two characters "<>," which together make a diamond shape

Advanced Encapsulation
(OCA Objectives 6.5)

Encapsulation for Reference Variables

- When encapsulating a mutable object like a **StringBuilder**, or an array, or an **ArrayList**, if you want to let outside classes have a copy of the object, you must actually copy the object and return a reference variable to the object that is a copy

```
class Special {
  private StringBuilder s = new StringBuilder("bob");  // our special data
  StringBuilder getName() { return s; }
  void printName() { System.out.println(s); }          // verify our special
                                                       // data
}
public class TestSpecial {
  public static void main(String[] args) {
    Special sp = new Special();
    StringBuilder s2 = sp.getName();
    s2.append("fred");
    sp.printName();
  }
}
```

  - When we run the code, we get this:

    Bobfred

  - If all you do is return a copy of the original object's reference variable, you DO NOT have encapsulation.

Using Simple Lambdas
(OCA Objectives 9.5)

Using Simple Lambdas

- The creators of the OCA 8 exam felt that, in general, lambdas and streams are topics more appropriate for the OCP 8 exam, but they wanted OCA 8 candidates to get an introduction, perhaps to whet their appetite…
- Lambda Syntax Example :
  - Here's the multipurpose **dogQuerier()** method:

```
static ArrayList<Dog> dogQuerier(ArrayList<Dog> dogList, Predicate<Dog> expr) {
    // do an "on the fly" query
    ArrayList<Dog> result1 = new ArrayList<>();
    for(Dog d: dogList)
      if( expr.test(d) )              // the key moment!
        result1.add(d);
    return result1;
  }
```

  - So far this looks like good-old Java; it's when we invoke dogQuerier() that the syntax gets interesting:

```
dogQuerier(dogs, d -> d.getAge() < 9);
```

    - When we say **d -> d.getAge() < 9** THAT is the lambda expression. The d represents the argument, and then the code must return a boolean.
    - We have to admit that lambdas are a bit tricky to learn. Again, we expect we've left you with some unanswered questions, but we think Oracle did a reasonable job of slicing out a piece of the lambda puzzle to start with.