Lecture 8 — The String class

CITS2005 Object Oriented Programming

Department of Computer Science and Software Engineering
University of Western Australia

Contents

- See Chapters 5 and 6 of the textbook
- More about the `String` type
- More about classes

The `String` type

- We have seen many uses of the `String` type
- It stores a sequence of characters (`chars`)
- There are string *literals* e.g., `String hi = "hello!";`
- But `String` is a *class* and thus each instance is an *object*
- This means it is passed by reference and can be constructed using `new`

String construction

```
public class StringConstruction {
    public static void main(String[] args) {
        String s = "Hello";
        String s2 = new String("Hello");
        String s3 = new String(s);
        System.out.println(s + " " + s2 + " " + s3);
    }
}
```

- Various ways to construct a `String`
- A subtle note: `"Hello"` is not really constructed. It is a constant value the compiler knows the make available even before the program runs

Java API

- String comes with many useful methods
- We can look at the various methods and constructors using the Java API
- https://docs.oracle.com/en/java/javase/11/docs/api/index.html
- See the different constructors?
- We will be looking at a subset of useful methods: length(), charAt(), equals(), substring(), toCharArray()

length() and charAt()

- String in Java is analogous to an array of chars
- The length() and charAt() methods let us use String in an array-like way
- length() gets the length (number of characters)
- charAt(i) gets the character at index i
- Note that we will see errors if we index an String (or array) out of bounds

length() and charAt()

```java
public class StringLength {
    public static void main(String[] args) {
        for (int i = 0; i < args[0].length(); i++) {
            System.out.println("The " + i + "th character is " +
                args[0].charAt(i));
        }
    }
}
```

- We can iterate through the characters of a string using `length()` and `charAt()`
- Note that the for-each loop does not work on strings directly

`String` is immutable

- Arrays can be modified by assigning an index e.g., `myArray[x] = 10;`
- `String` only provides `charAt`
- This means we can only read the characters in a string, but not write them
- `String` in Java is *immutable*
- A string cannot be changed once it has been created
- If you want to modify a string, you create a new string with the modification (e.g., concatenation with +)
- How can we modify a specific index?

toCharArray()

```java
public class ModifyString {
    public static void main(String[] args) {
        String s = "Heloo";
        char[] chars = s.toCharArray();
        chars[3] = 'l';
        s = new String(chars);
        System.out.println(s);
    }
}
```

- `s.toCharArray()` returns a *copy* of the string as a character array
- A common pattern is to modify the character array, then create a new `String`
- Note that even though strings are immutable, we can still change which reference a `String` variable points to: `s = new String(chars);`

substring()

```java
public class Substring {
    public static void main(String[] args) {
        String s = "CITS2005";
        String cits = s.substring(0, 4);
        String code = s.substring(4);
        System.out.println(s);
        System.out.println(cits + " -- " + code);
    }
}
```

- `s.substring(0, 4)` returns the substring from indexes 0 up to but excluding 4
- `s.substring(4)` returns the substring from index 4 onwards
- Notice that calling substring always returns a new string and does not modify *s*

equals()

```
public class WrongEquals {
    public static void main(String[] args) {
        String s = "CITS2005";
        String s2 = new String("CITS2005");
        if (s == s2) {
            System.out.println("s == s2");
        } else {
            System.out.println("s != s2");
        }
    }
}
```

- What is wrong here?

equals()

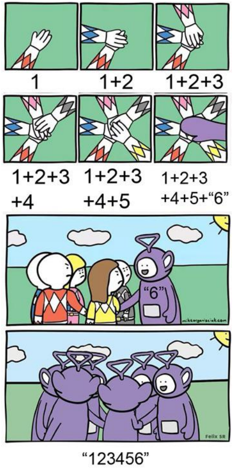
```java
public class WrongEquals {
    public static void main(String[] args) {
        String s = "CITS2005";
        String s2 = new String("CITS2005");
        if (s == s2) {
            System.out.println("s == s2");
        } else {
            System.out.println("s != s2");
        }
    }
}
```

- `String` is a class, so `s` and `s2` are objects
- The `==` operator for objects compares their references, not their contents!
- The `equals()` method is used to compare contents for `String` and many other classes

equals()

```
public class CorrectEquals {
    public static void main(String[] args) {
        String s = "CITS2005";
        String s2 = new String("CITS2005");
        if (s.equals(s2)) {
            System.out.println("s == s2");
        } else {
            System.out.println("s != s2");
        }
    }
}
```

Mid-lecture Break



There is a Java error in this meme. What is it?

MyString class

- How would we go about implementing our own `String` class?
- Note that the `String` class in Java is built-in
- And all literals `"like this"` are automatically strings
- However, imagine this was not true for the sake of an exercise
- We will implement some of the methods we have just seen: `length()`, `charAt()`, `equals()`, `substring()`
- We will also implement a `concatenate()` method that works similarly to +
- First, lets consider the outline of the class

MyString class outline

```
public class MyString {
    private char[] chars;

    public MyString(char[] chars) {
    }

    public char charAt(int index) {
    }

    public int length() {
    }

    public boolean equals(MyString s) {
    }

    public MyString substring(int start, int end) {
    }

    public MyString concatenate(MyString s) {
    }
}
```

`private` keyword

- `private char[] chars;`
- The `private` keyword is new
- We are saying that the `chars` *field* can only be accessed by this class
- The `public` keyword means that something can be seen by every class
- If another class tried to do `myStringInstance.chars`, then it would result in an error!

private example

```
public class MyStringExample {
    public static void main(String[] args) {
        MyString s = new MyString("Hello".toCharArray());
        s.chars[0] = 'J';
        for (int i = 0; i < s.length(); i++)
            System.out.println(s.charAt(i));
    }
}
```

- `MyStringExample.java:4: error: chars has private access in MyString`

Data Hiding

- This achieves *data hiding*.
- A user of the `MyString` class does not know how the characters are stored!
- They only see the `public` methods
- This is part of *encapsulation*
- Related code and data are bundled together as methods and fields in a class. This allows fine control over which parts are exposed
- Data hiding and encapsulation achieve *abstraction*
- A user of `MyString` (or even `String`) does not need to know how the methods are implemented, or what the fields are. They only need to know what they do and how to call them
- This is crucial to maintaining a complex code base with many people. It makes it possible for each person to work on their own classes without needing to know the details of all the other classes

Abstraction

- Abstraction is important
- Building an aeroplane would be nearly impossible if an engineer needed to know how every component worked
- However, they use abstraction: they only need to know what an engine does and where to plug it in
- A user of `MyString` (or even `String`) does not need to know how the methods are implemented, or what the fields are. They only need to know what the public methods do and how to call them
- This is crucial to maintaining a complex code base with many people. It makes it possible for each person to work on their own classes without needing to know how all the other parts of the code work

Constructor

```
public MyString(char[] chars) {
    this.chars = new char[chars.length];
    for (int i = 0; i < chars.length; i++) {
        this.chars[i] = chars[i];
    }
}
```

- Let's start with the constructor
- Recall that constructors are used when we create a new instance
- e.g., `new MyString(...)`
- Note the use of `this.chars` to avoid name collision
- Note that we store a copy instead of a reference: encapsulation!

charAt()

```java
public char charAt(int index) {
    return chars[index];
}
```

- This is called a *getter*
- These methods allow read-only access to data
- Enforces data hiding

length()

```
public int length() {
    return chars.length;
}
```

- This is another getter function that provides read-only access to chars

equals()

```java
public boolean equals(MyString s) {
    if (length() != s.length())
        return false;
    for (int i = 0; i < length(); i++) {
        if (charAt(i) != s.charAt(i))
            return false;
    }
    return true;
}
```

- Note that we expect an object as the parameter. The type is `MyString`
- Checks the length, then checks all the characters
- Composed by using the methods we have already written
- Note the usage of `return` as an early exit

substring()

```
public MyString substring(int start, int end) {
    char[] newChars = new char[end - start];
    for (int i = start; i < end; i++) {
        newChars[i - start] = chars[i];
    }
    return new MyString(newChars);
}
```

- Notice how this method returns an object (the return type is `MyString`)
- Also, it does not modify `this`
- We use some neat index arithmetic: `i - start` and `end - start`

concatenate()

```
public MyString concatenate(MyString s) {
    char[] newChars = new char[chars.length + s.length()];
    for (int i = 0; i < chars.length; i++)
        newChars[i] = chars[i];
    for (int i = 0; i < s.length(); i++)
        newChars[chars.length + i] = s.charAt(i);
    return new MyString(newChars);
}
```

- Again, `MyString` is immutable
- This method creates a new object containing the concatenation

MyString example

```java
public class MyStringExample2 {
    public static void main(String[] args) {
        MyString s = new MyString("Hello".toCharArray());
        s = s.concatenate(new MyString("World".toCharArray()));
        s = s.substring(3, 6);
        if (s.equals(new MyString("loW".toCharArray())))
            System.out.println("Success");
        else
            System.out.println("Failure");
        for (int i = 0; i < s.length(); i++)
            System.out.println(s.charAt(i));
    }
}
```

- Let's see an example usage of MyString