CS 162FZ: Introduction to Computer Science II

Lecture 16

Final Revision

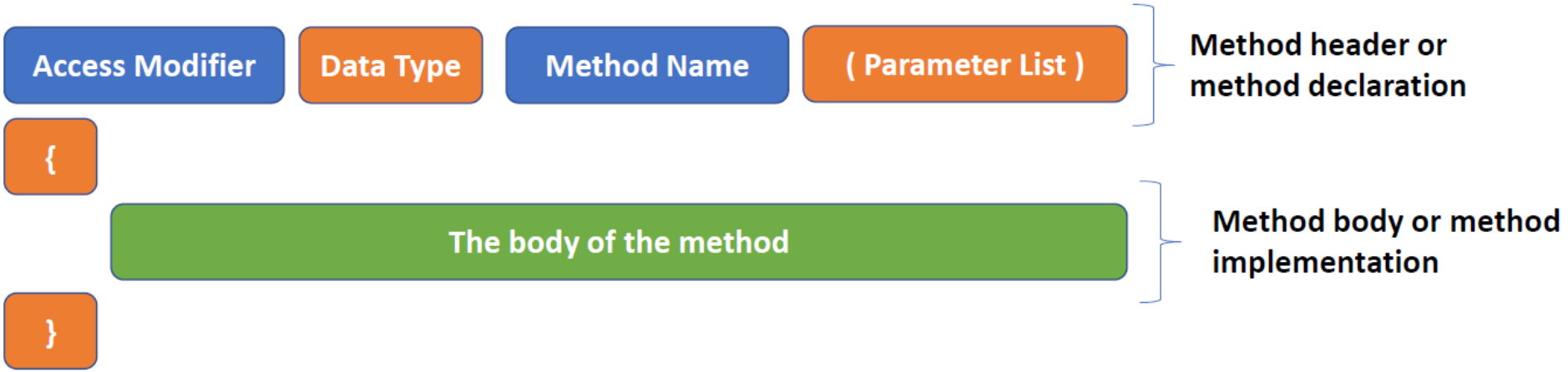
Dr. Chun-Yang Zhang

Methods

- A **method** is a program module that contains a series of statements that carry out a task.
- Methods are often used to define reusable code, organize and simplify coding.
- Methods are sometimes called *Procedures*, *Functions* or *Operations*.

Defining a Method

- A method definition consists of an access modifier (optional), data type of returning value, a method name, a list of parameters and a body.



Access Modifiers

Access modifiers can be public , protected , private or unspecified (no explicit modifier).

- private : The most restricted access modifier. A private method can only be accessed within the class of its definition. It cannot be access from outside of the class.
- Unspecified : If the access level of a method is unspecified, the method can only be accessed within the package. It cannot be accessed from outside the package.
- protected : The access level of a protected methods is within the package and/or outside the package through child class.
- public : No restriction on the access to public

Data Type of Returning Values

- A method may return a value. The data type in the method declaration is thus the data type of the value the method returns.
- To return a value from a method, use the keyword return
- A method may not return a value. The data type in the method declaration is specified using the keyword void
- If a method returns a value, it is called a value returning method (or function). If a method does not return a value, it is often called a void method (or procedure)

Keyword: Static

- Static methods are also known as class method that are declared with a class.
- If you do not declare the method to be static then it must be called by creating an object of the class. This is known as an instance method.
- Static methods are commonly used to perform calculations that are independent of any object that might be defined in a class.

Calling a static method from outside

The required format is:

The class name followed by a dot, followed by the method name

For example to call the method printGreeting() which is contained in the MethodsExample1 class we could use:

```
MethodExample1. printGreeting()
```

Parameter List

- Variables defined in the method header are called formal parameters or simply **parameters**
- Parameters are optional. A method may have one or more parameters or not at all.
- Each parameter in a method must be declared separately, i.e., each parameter in the list must have a data type and a name.
- Multiple parameters are separated using a comma sign “,”

Parameter Names—same names

You can use a same name for the parameters when you call a method (actual parameters) to the manes used in the method signature (formal parameters).

```
public class MethodScope1
{
    // instance variables - replace the example below with your own
    public static void main(String[] args)
    {
     int inValue = 4;
    //Actual parameters
    twoTimes(inValue);


    } //same names
// Formal parameters public static void twoTimes(int inValue)
   public  static void twoTimes(int inValue)
   {

       System.out.println("The result is "+inValue*2);

   }
}
```

Passing Variables

When we are passing variables between methods, the variables are passed by value. For example, when a primitive data type is passed into a method a copy of the value is sent to the method.

```
public class PassingByValue {
    public static void main (String args[]) {
        int x = 2;
        System.out.println("The value of x before timesTwo is " + x);
        x = timesTwo(5);
        System.out.println("The value of x after timesTwo is " + x);
        }
    public static int timesTwo (int x) {
        System.out.println("The value of x in timesTwo is " + x);
        int result = x * 2;
        return result;
        }
}
```

Note: 5 is passed to timesTwo() method and stored in a variable (also called x).

Passing Primitive data type- example

```
/** * The following code demonstrates pass by value for primitives in Java. **/
 public class PassByValuePrimitiveType {
    public static void main (String args []) {
        int x = 10;
        System.out.println("About to call changeX(), the value of x is " + x);
        //QUESTION 1: What value is x after this method call? Why?
        changeX(x);
        System.out.println("Back from changeX(), the value of x is " + x);
        System.out.println("\n*****************************************\n");
        /* The following code uses explicit re-assignment to change the value of x * in main */
        System.out.println("Calling changeX2() and assigning the returned value to x "
                + "(current value of x is " + x + ")");
        //Question 2: What value is x after this method call? Why?
        x = changeX2(x);
        System.out.println("Back from changeX2()new value of x is " + x);
        System.out.println("\n*****************************************\n");
        }
    /** * This method changes the value of the formal int parameter x */
    public static void changeX(int x) {
        x = 17;
        System.out.println("The value of x in changeX() is " + x);
        }

    /** * This method changes the value of the formal int parameter x and * returns same */
    public static int changeX2(int y) {
        y = 17;
        System.out.println("The value of x in changeX2() is " + y);
        return y;
    }
}
```

Passing Reference Types

When passing reference types of type object (e.g. arrays, Strings, etc.), java passes the memory address of that object to the method rather than passing the actual value.

When we create an array as follows:

```
int a[] = {1, 2, 3, 4}
```

a piece of memory is set aside that is referenced by the variable a. Here, a will store the memory address of the first value in the array.

It is important to understand that what a forma parameter object is used, it modifies the actual object state unless you reassign the object. That is, change the address the variable is pointing to.

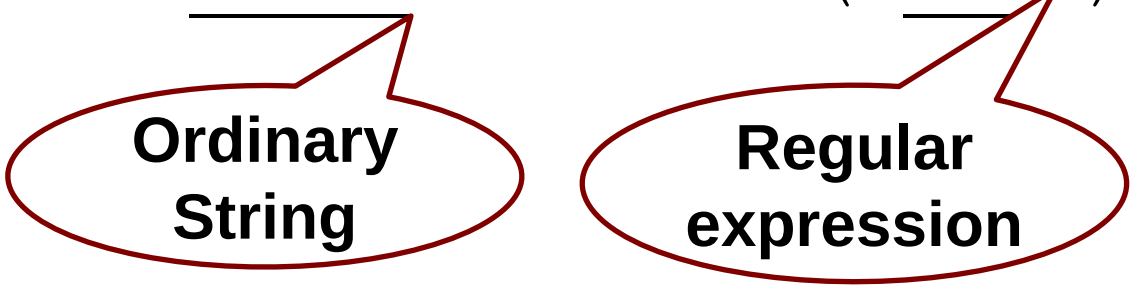
What is a regular expression?

- A regular expression is a special sequence of characters that helps you match or find other strings or sets of strings, using a specialized syntax held in a pattern.

- They can be used to search, edit, or manipulate text and data.

String.matches(String regex)

- Regular expressions are used by both the
  - `String` class using `matches()` method, and the
  - `Pattern` & `Matcher` classes
- `boolean bool = "abbbb".matches("ab*")`



- The general format for the matches method is

"Problem-string" .matches("RegEx")

RegEx as a String Template

- We can think of the regular expression grammar as a ***template*** against which to match strings
- Typically, we either
  - find something that matches a template from a large volume of data
  - Ensure that a specific datum matches a template
    - Searching the internet, text of books, textual records …
  - Looking for valid phone numbers, student ID, names, variables ...

Regular Expression Grammar

- **Sequence** (*and*) is the 1st character followed by 2nd character…
  - abc // exact match
  - Abc // case sensitive
- **Alternatives** (*or*) are enclosed in []
  - ca[bdn] // cab, cad, can
    //NOT car, cat...
  - Diarm[au]id //alternative spellings
  - [Dd]ean //possible captials

More RegEx

- **Not [^]**
  - ca[^brt] can and cad, but **not** car, cat or cab

Note ^ can be used to match the beginning of a line.

- **Ranges**
  - [a-z] any lower case letter
  - [A-Z] any capital letter
  - [0-9] any digit
  - [a-z&&[^xyz]] a-z but **Not** x, y or z

Quantifiers

- * **zero or more** times (Kleene *)
  - ab* a ab abb abbbb
  - [ab]* aa aabbababba aabbaba abba bbba
- + **One or more** times
  - ab+ ab abb abbbb
  - [0-9]+ any sequence of >1 digits
  - [A-Z]+ any sequence of >1 capitals
- ? **Zero or once** **(An Optional character)**
  - Colou?r Colour Color
  - rea?d red read, but **not** reed

Counted Items

- Counted number of items
  - x{3} // xxx only
- At least number of items
  - x{3,} /// xxx xxxx xxxxxxxx *etc* *3 or more*
- Between 2 and 4 instances of
  - x{2,4} xx, xxx and xxxx only
  - .[a-z]{2,4} Top level of email address (.ie .com .info)

Wild Character = “ . “

- The dot **.** is the wild character which allows for **any** character in a string except the new line character

For example:

re.d // matches *read reed rezd*

Special Characters

- Special characters are occasionally used within “complicated” strings
- How do we get a String that contains the ” character?
- We use backslash so that the following character is not treated in the usual way
  - “the quote \” mark “ matches *the quote “ mark*
- So how do we get a backslash in a string?
  - “one \\ character” **Matches** *one backslash \ character*
  - “two backslash \\\\ characters”**Matches** *two backslash \\ characters*

Special Characters & Strings

- Write down the Java code that creates the following Strings
- **What Java sees What i must type**
- “ab”c” “ab\”c”
- “ab””c” “ab\”\”c”
- “ab\c” “ab\\c”
- “ab\\c” “ab\\\\c”

Special Characters in RegEx

- Remember the **.** will match any character
- Note: the backslash **** character is a special character. It means, **do Not** treat the following character in the normal way
- \. The full stop character
- \b word boundary
- \s white space (space or tab)
- \\ the backslash \ character
- \t the Tab character
- \d the digits
- \w the word characters

```
a "word" is a nonempty sequence of alphanumeric characters
and underscores
```

Regular Expression Syntax

Subexpression	Matches
^	Matches the beginning of the line.
$	Matches the end of the line.
.	Matches any single character except newline. Using **m** option allows it to match the newline as well.
[...]	Matches any single character in brackets.
[^...]	Matches any single character not in brackets.
\A	Beginning of the entire string.
\z	End of the entire string.
\Z	End of the entire string except allowable final line terminator.
re*	Matches 0 or more occurrences of the preceding expression.
re+	Matches 1 or more of the previous thing.
re?	Matches 0 or 1 occurrence of the preceding expression.
re{ n}	Matches exactly n number of occurrences of the preceding expression.

Regular Expression Syntax

re{ n,}	Matches n or more occurrences of the preceding expression.
re{ n, m}	Matches at least n and at most m occurrences of the preceding expression.
a| b	Matches either a or b.
(re)	Groups regular expressions and remembers the matched text.
(?: re)	Groups regular expressions without remembering the matched text.
(?> re)	Matches the independent pattern without backtracking.
\w	Matches the word characters.
\W	Matches the nonword characters.
\s	Matches the whitespace. Equivalent to [\t\n\r\f].
\S	Matches the nonwhitespace.
\d	Matches the digits. Equivalent to [0-9].
\D	Matches the nondigits.

Regular Expression Syntax

\A	Matches the beginning of the string.
\Z	Matches the end of the string. If a newline exists, it matches just before newline.
\z	Matches the end of the string.
\G	Matches the point where the last match finished.
\n	Back-reference to capture group number "n".
\b	Matches the word boundaries when outside the brackets. Matches the backspace (0x08) when inside the brackets.
\B	Matches the nonword boundaries.
\n, \t, etc.	Matches newlines, carriage returns, tabs, etc.
\Q	Escape (quote) all characters up to \E.
\E	Ends quoting begun with \Q.

Introduction

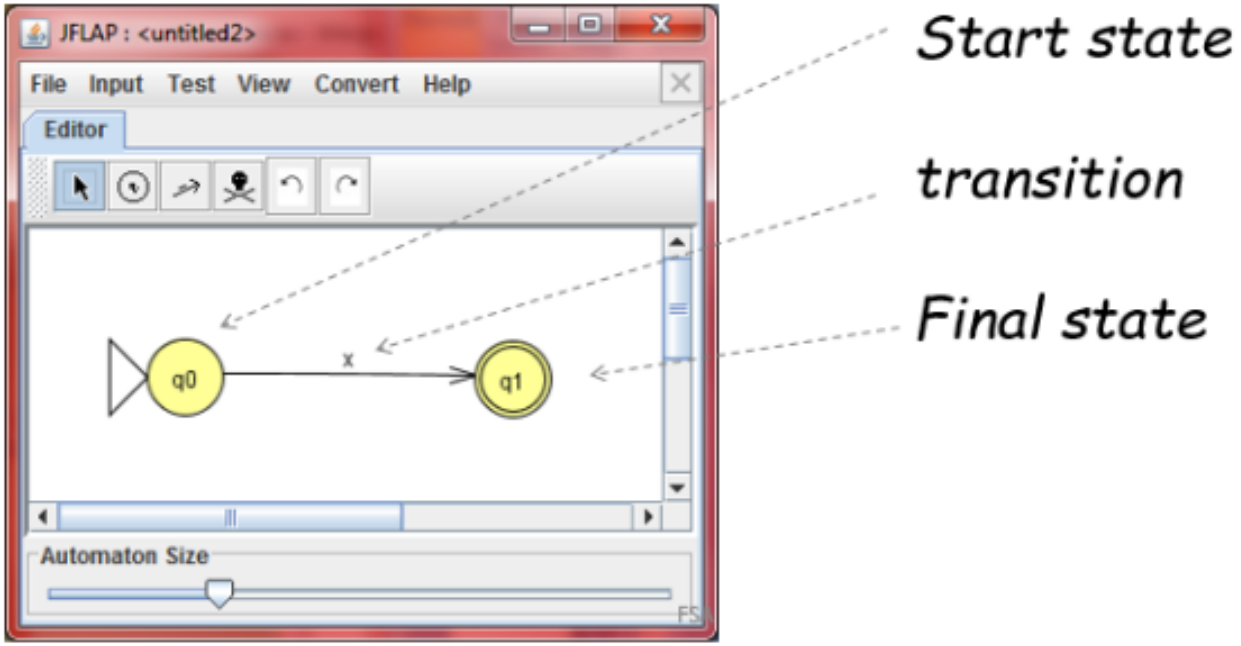
Finite State Machines (FSMs) offer a different perspective on regular expressions. A FSM describes the behaviour of a Regular Expression (RegEx). Each FSM is composed of the following 5-tuple (Q, Σ, δ, q0, F) where:

- *Q:* a finite set of states
- Σ*:* a finite alphabet set
- *δ*: Q X Σ → Q: transition function - a set of maps from states and inputs into states
- q0: an initial state (q0 ∈ Q)
- F: a set of final/accepting states (F ⊆ Q)

FSMs are abstract machines that can only be in one of a finite number of states (Q) at any given time. The FSM can transition from one state to another in response to some external inputs.

JFLAP

- JFLAP is a tool which allows us to model the operations of FSMs. You can download it from http://www.cs.duke.edu/csed/jflap/
- Here is a simple JFLAP representation of the language composed of a single x:



Components of a Class

- There are a number of components that most classes share in common. These are:

  1. A class name.

- This name should be meaningful and represent the function/purpose of the class. For example – if you are creating a class to represent a car then the name of the class should be `Car`. The normal naming convention is that class names should start with an **uppercase letter and be a noun.**

  2. Attributes, also known as **instance variables.**

- These are variables that will be used inside the class to hold values such as the make/model of the car, how many doors it has, etc.

Components of a Class

3. Constructor(s).

- There should always be a default constructor. This type of constructor has zero input parameters. If there is none provided the JVM will automatically provide one.
- There will usually be another constructor with 1 or more with parameters.

4. Getters methods (also known as Accessors).

- These are methods defined by the user to return the values of the attributes in a class.

5. Setters (also known as Mutators).

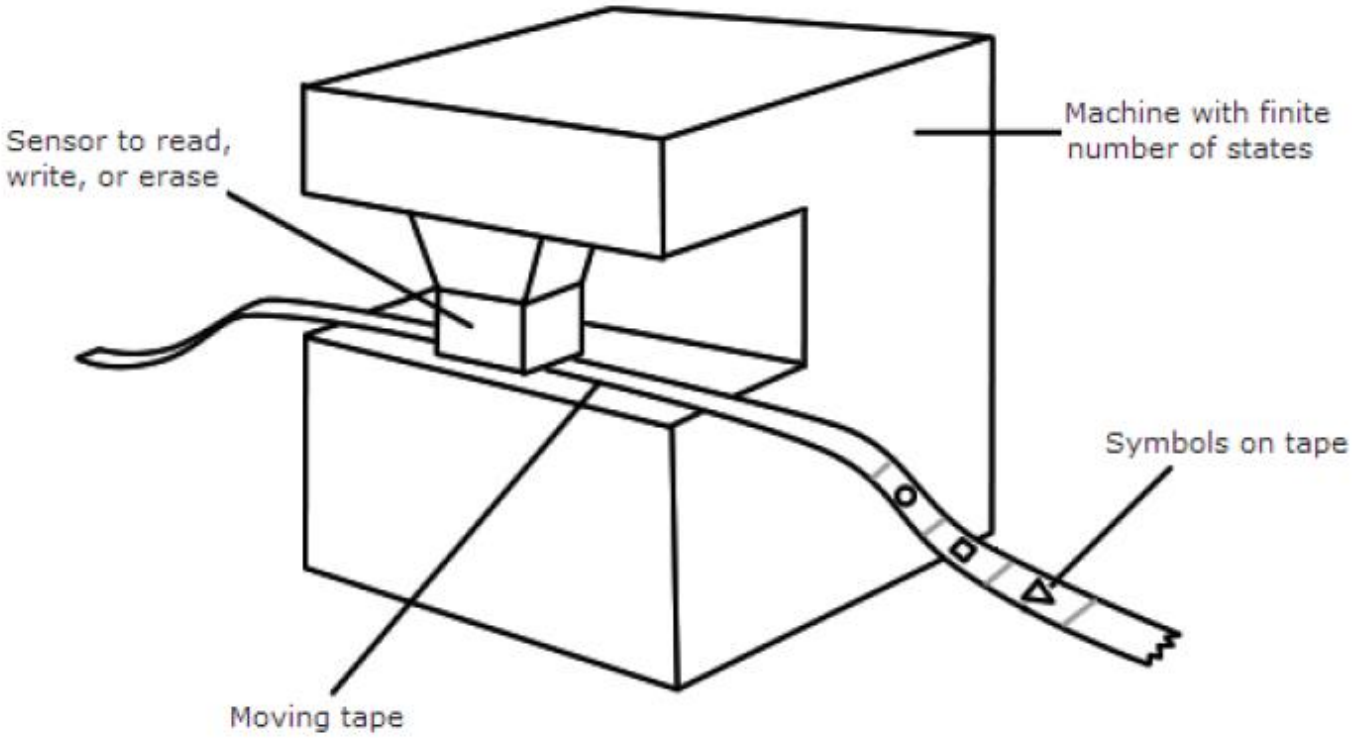
- These are methods which change the value of attributes in a class

6. Functionality in the form of methods.

Inheritance: Summary

- Java allows us to reuse class definitions and **extend** functionality by allowing one class (the child or subclass) to **inherit** from another class (the parent or superclass) - This is known as **inheritance**
- All attributes and methods of the superclass are available in each subclass when inheritance is used
- The keyword `extends` allows one class to inherit from another
- In order to call the super class constructor we use the keyword `super()`
- We can override the behaviour of a super class method and add our own functionality on top of whatever is currently provided.

Example of Turing Machine



Overview of a Turing Machine

There are just **six types of fundamental operation** that a Turing machine performs in the course of a computation. These are to:

- **read the symbol** that the head is currently over
- **write a symbol** on the square the head is currently over it will need to **clear** the symbol currently here, if any
- **move the tape left one position**
- **move the tape right one position**
- **change state**
- **halt**