<< Lecture 8 >>

Preamble

- I am trying a new font—is it more readable?
- **Homework due next Wednesday**
- **Midterm 1 in three weeks exactly**
- Lecture:
  - Going over things I forgot to mention
  - Method overloading
  - Problem solving—preamble to **polymorphism**

Special characters

- There are also ways to write *special characters* in Strings and chars
- Special characters start with backslash "\"
- Typical special characters:
  - \\ **: backslash (yes, you need to write it twice)**
  - **n** : new line; ends a line and starts a new one
    - **r** : return carriage, for typewriters and Windows; in Windows, a new line consists of \r\n (two chars)
  - **t** : tabular space
- For *Unicode* values, we can use the \u prefix:
  - **uXXXX** : where XXXX is a number in hexadecimal notation, for example:
    - \u0041 : uppercase A (0x0041 == 65)
    - \u000d : newline (or \n; 13 == 0x000d)
- *Hex* digits are 0, 1, ..., 9, A, B, ..., F
  - F means 15 (base-1, i.e., 16-1, pretty much like 9=10-1)
- '\u0041' == (char) 0x0041 == (char) 65

Method overloading

- It is possible to define several methods with the same name within the same class—*how come!?*
- If the methods differ in the number of parameters or the types of the parameters, then

there is no ambiguity!

- Java automatically picks the method that *best matches* the caller's supplied parameters
- If the type of an parameter does not match any of the supplied types, Java will try casting; the method that matches with the *closest type* will be used

```
class Zup {
  private void print(String x) { System.out.println(x); }

  public void typ(double x) { print("double"); }

  public void typ(String x) { print("String"); }
}

public class Demo{
  public static void main(String[] args) {
    Zup z = new Zup();
    z.typ( 5 );
    z.typ( "hello worldo" );
    z.typ( null );
  }
}
```

- Java is smart enough: 5 goes as double, not int, and null goes as String

```
class Zup {
  private void print(String x) { System.out.println(x); }

  public void typ(String x) { print("String"); }

  public void typ(int[] x)  { print("Integer Array"); }
}

public class Demo{
  public static void main(String[] args) {
    Zup z = new Zup();
    z.typ( "hello worldo" );
    z.typ( null );              // WRONG!
    z.typ( (int[]) null );      // Correct
  }
}
```

- Now, the null is ambiguous—we can resolve this with casting

```
class Zup {
  private void print(String x) { System.out.println(x); }

  public void typ(int x)    { print("Integer"); }

  public void typ(String x) { print("String"); }
}

public class Demo{
  public static void main(String[] args) {
    Zup z = new Zup();
    z.typ( 5.0 );       // WRONG
    z.typ( 5 );         // Correct
    z.typ( "hello worldo" );
  }
}
```

- Java does not like destroying precision all that much—double won't be treated as int, but the converse works

```
class Zup {
  private void print(String x) { System.out.println(x); }

  public void typ(int x, double y)  { print("int, double"); }

  public void typ(double x, int y)  { print("double, int"); }
}

public class Demo{
  public static void main(String[] args) {
    Zup z = new Zup();
    z.typ( 5, 5.0 );
    z.typ( 1.0, 2 );
    z.typ( 3, 3 );      // WRONG
  }
}
```

- But ambiguity comes in more forms; see the above code for an example

```
class Zup {
  private void print(String x) { System.out.println(x); }

  public void typ()  { print("0 args"); }
  public void typ(double x)  { print("1 arg"); }
  public void typ(double x, double y)  { print("2 args"); }
}

public class Demo{
  public static void main(String[] args) {
    Zup z = new Zup();
    z.typ();
    z.typ( 1 );
    z.typ( 2, 2 );
  }
}
```

- Different numbers of parameters do not introduce ambiguity

CONSTRUCTOR OVERLOADING

- In absence of an explicit constructors, classes have a default constructor—but once a constructor is defined, the default constructor cannot be used again
- We can add the empty constructor if we wish; in fact, we can add as many alternative constructors as we desire
- **Constructors can be overloaded**
- *Why?* Because it might be useful to have several ways of initializing an object

On writing code and problem solving

Readable code

- As mentioned, method overloading exists for convenience
- The idea is to make code more readable; thus, several methods can exist with the same name if they are related to the **same idea**
- Pick **good names** for **methods** and variables (especially **fields**)
- Pick good names for parameters
- Temporary variables are of lesser importance

Commenting and documenting

- A good practice is to always document your code
- Documentation starts with comments
- Java allows in-line and multi-line comments

```
// this is an in-line comment

/*
 * this is a multi-line comment
 */
```

- Comment before (important) methods
- **Don't restate the code!**

```
/* The following method concatenates a series of Strings
 *
 * Parameters:
 *    String[] S : the collection of Strings to be concatenated.
 *                 S will not be modified in any way.
 *
 * Returns: a String with the elements of S concatenated, i.e.,
 *    S[0] + S[1] + …    */

public String concatStrings( String[] S ) {
   // we first check whether S is null
   if ( S == null )  return "";
   // otherwise, we build a solution
   String result = "";
   for(String x : S)
      // we can only concatenate non-null strings
      if (x != null) result = result + x ;
   return result;
}
```

- Perhaps the previous function was too simple for all that commentary, but you can't deny its readability!
- The IDE will highlight comments and code in general to make it more readable

- **Javadoc**
  - The Javadoc tool reads your code and generates a guide to it
  - Good code leads to a good document!
    - @author @version
    - @param @return (@since for updates)

```
/**
 * The following method concatenates a series of Strings
 *
 * @param   S     the collection of Strings to be concatenated.
 *                S will not be modified in any way.
 *
 * @return        a String with the elements of S concatenated,
 *                i.e., S[0] + S[1] + …
 */
```

- The comment above can be used to generate a Javadoc document automatically

LAYING OUT A PLAN AND PSEUDOCODE

- Often times, to write a method, it is good to plan how the method will be coded in terms of **steps**
- One can also think of definitions and meditate about the *meaning* of the method—this often leads to **recursion**
- But back to **steps** (or imperative thinking)
- A detailed plan is called **pseudocode**:
  - pseudocode is much like normal code, but written in lay english language
  - Python was meant to look a lot like pseudocode

Examples of pseudocode:

https://commons.wikimedia.org/wiki/File%3ALatex-algorithm2e-if-else.png

(CC Angela0130)

Data: this text
Result: how to write algorithm with LaTeX2e
initialization;
while *not at end of this document* **do**
  read current;
  if *understand* **then**
    go to next section;
    current section becomes this one;
  else
    go back to the beginning of current section;
  end
end

Algorithm 1: How to write algorithms

https://commons.wikimedia.org/wiki/File%3AMedian_filter_pseudocode.png

(CC Angela0130)

```
Program median filter
for every pixel in the image do
        sort values in the mask
        pick the middle one in the sorted list
        replace the pixel value with median one
end
```

I wrote this one for my last paper.

Algorithm 1 Greedy randomized embedded feature filter

Input: m: bucket size, $\mathcal{H}$: features, $\mathcal{S}$: rows of the data set

1: Randomly partition $\mathcal{H}$ into sets $B_1, ..., B_k$, so that $|B_i| = m$ for every i such that $1 \leq i \leq k-1$.
2: Let $C = B_1$.
3: **for** i = 2 to k **do**
4: Fit logistic regression model to $\mathcal{S}$ projected on $C \cup B_i$
5: For $h \in C \cup B_i$, define $s(h)$ as the number of classification errors introduced when β_h is set to 0 (β_h is the coefficient of the logistic regression model for feature h).
6: Update C to be the m features in $C \cup B_i$ with the highest $s(h)$.
7: **end for**
8: **return** C

<< Lecture 9 will resume from here >>