CSE 250
Data Structures

Dr. Eric Mikida
epmikida@buffalo.edu
208 Capen Hall

Lec02: Java Refresher

Announcements and Feedback

- Join Piazza! (you should have an invite in your email)
- Academic Integrity Quiz due 2/4 @ 11:59PM **(MUST GET 100%)**
- PA0 due 2/4 @ 11:59PM **(MUST GET 100%)**
- WA1 - Don't worry about it too much yet, we'll get there on Monday

Why Java?

- **Strongly Typed Language:** The compiler helps make sure you mean what you say
- **Compiled Language:** Can run it anywhere, see the impacts of your data structure choice and data layout
- **You know it (hopefully):** You learned the basics in 116

Hello World

```
1  package cse250.examples;
2
3  class MainExample {
4    /**
5     * Main function
6     * @param args  The arguments to main
7     */
8    public static void main(String[] args) {
9      System.out.println("Hello World");
10   }
11 }
```

Hello World

```
1  package cse250.examples;
2
3  class MainExample {
4    ...
5  }
```

- All code in Java lives in a class
  - In general each class will be in it's own .java file
- Classes are organized into packages
  - Think directories…

Hello World

```
1   /**
2    * Main function
3    * @param args  The arguments to main
4    */
```

- Single line comments in Java start with //
- Multi line comments in Java start with /* and end with */
- Javadoc comments start with /**

Hello World

```
1   public static void main(String[] args)
```

- `public` - the function can be called by anyone (instead of `private`)
- `static` - the function isn't tied to a specific object
  - To call this function we would write `MainExample.main(...)`
- `void` - the functions return type (in this case it doesn't return anything)
- `main` - the function name
- `String[] args` - the parameter list
  - In this case, a single parameter with the type array of `String`

Hello World

```
1    System.out.println("Hello World");
```

- `System` refers to `java.lang.System`
- `System.out` is the out field of `System`
- `System.out.println` is a function that prints a line of text
- Semicolons (`;`) are mandatory

Exceptions

```
1  public List<String> loadData(String filename) {
2    List<String> ret = new ArrayList<String>();
3    BufferedReader input =
4        new BufferedReader(new FileReader(filename));
5    String line;
6    while( (line = input.readLine()) != null ) {
7      ret.add(line);
8    }
9    return ret;
10 }
```

Exceptions

```
 1 public List<String> loadData(String filename) {
 2   List<String> ret = new ArrayList<String>();
 3   BufferedReader input =
 4       new BufferedReader(new FileReader(filename));
 5   String line;
 6   while( (line = input.readLine()) != null ) {
 7     ret.add(line);
 8   }
 9   return ret;
10 }
```

```
java: unreported exception java.io.IOException; must be caught or declared to be thrown
```

What are Exceptions

They are a way to catch an error when something goes horribly wrong!

So what do you do?

Catching Exceptions

```
1  public List<String> loadData(String filename) {
2    try {
3      BufferedReader input =
4          new BufferedReader(new FileReader(filename));
5      String line;
6      while ((line = input.readLine()) != null) {
7        ret.add(line);
8      }
9      return ret;
10   } catch(IOException e) {
11     // Handle the exception, ie print out what went wrong
12     e.printStackTrace();
13   }
14 }
```

Catching Exceptions

```
1  public List<String> loadData(String filename) {
2    try {
3      BufferedReader input =
4          new BufferedReader(new FileReader(filename));
5      String line;
6      while ((line = input.readLine()) != null) {
7        ret.add(line);
8      }
9      return ret;
10   } catch(IOException e) {
11     // Handle the exception, ie print out what went wrong
12     e.printStackTrace();
13   }
14 }
```

Try something that isn't guaranteed to work....

Catching Exceptions

```
1  public List<String> loadData(String filename) {
2    try {
3      BufferedReader input =
4          new BufferedReader(new FileReader(filename));
5      String line;
6      while ((line = input.readLine()) != null) {
7        ret.add(line);
8      }
9      return ret;
10   } catch(IOException e) {
11     // Handle the exception, ie print out what went wrong
12     e.printStackTrace();
13   }
14 }
```

…and "catch" the exception in case something goes wrong

Passing Along Exceptions

```
1  public List<String> loadData(String filename)
2    throws IOException // Communicate the explosive potential
3  {
4    BufferedReader input =
5        new BufferedReader(new FileReader(filename));
6    String line;
7    while ((line = input.readLine()) != null) {
8      ret.add(line);
9    }
10   return ret;
11 }
```

Passing Along Exceptions

```
1  public List<String> loadData(String filename)
2    throws IOException // Communicate the explosive potential
3  {
4    BufferedReader input =
5        new BufferedReader(new FileReader(filename));
6    String line;
7    while ((line = input.readLine()) != null) {
8      ret.add(line);
9    }
10   return ret;
11 }
```

If your function does not handle the exception itself, then you need to let the outside world know something might go wrong

Coding Style is IMPORTANT!!

```
1     class neatClass
2  {
3     public static void
4    doSomething(String wowwww)
5        {
6    String weee = "Yes";
7    // this is definitely a for loop
8      for (q : wowwww)
9        System.out.println(q);
10       System.out.println(wee);
11    }
12 }
```

What the heck is going on here!?

Naming

These are all valid variable names…

- `neatClass`
- `doSomething`
- `wowwww`
- `weee`

But are not helpful to anyone reading your code (including you)

Use variable names that summarize the variable's role or contents

Naming

Use variable names that summarize the variable's role or contents

- `username`: a string containing a users login name
- `nextNode`: a pointer to the next node in a linked list
- `data`: the contents of an ArrayList
- `leftChild`: a pointer to the left child of a BST

Indentation/Spacing

```
1  class neatClass {
2    public static void doSomething(String wowwww) {
3      String weee = "Yes";
4      // this is definitely a for loop
5      for (q : wowwww) System.out.println(q);
6      System.out.println(wee);
7    }
8  }
```

Consistent spacing helps the reader more quickly understand the structure of the code

Comments

```
1 // this is definitely a for loop
```

This comment doesn't actually tell us anything useful (we can clearly see that what follows is a for loop…)

Comments should provide info that's **not** already present in the code

- Assumptions you have made when writing the code
- References to documentation/citations
- Clean descriptions of any non-obvious math
- The reasoning behind the chosen solution (especially if it is not the "obvious" way)

Brackets/Braces

```
1 for (q : wowwww) System.out.println(q);
```

Java supports one-line for loops. This is a really nifty and easy way to…introduce bugs into your code.

ALWAYS USE BRACES!

Ways to Succeed when Coding

- **NEVER** start with code
- What do you have to start with? How is it organized?
  - Draw pictures
  - Try examples on paper
- What do you want the result to be? How should it be organized?
  - DRAW MORE PICTURES/EXAMPLES
- Now figure out how the given input and desired output relate
  - Connect your drawings/diagrams
- Break down bigger problems into smaller ones as needed

Ways to Obtain Assistance

- Explain what you've tried
  - Which test cases fail (and if you don't have test cases, make them!)
  - What approaches have you tried and how do they break
- Explain **what** it is you want to accomplish, and **why** you want to
  - Make sure we have all the context
- Follow coding style guidelines!

If you don't feel comfortable with Java...

Remember: Don't start with coding, you should already have plenty of pictures/examples/ideas before coding

If you bring us (mostly working) pseudocode, the course staff will happily help you translate it to Java

If you don't feel comfortable with Java...

Typical Questions:

- **Syntax Questions** (eg: How do I break out of a for loop?)
  - Ask on Piazza, Office hours, etc
  - We can give a very direct answer (ie: you can use the **`break`** keyword)
- **Semantics Questions** (eg: How do I insert an item into a linked list?)
  - Still ask the question!
  - ...but the answer will generally not involve code

Many of the "syntax" questions we get are actually about semantics

Basic Debugging

Live Demo

Unit Testing

- When we write code we make a lot of assumptions
  - Often statements of the form [piece of code] should [do a thing]
  - The computer does not know about these assumptions…unless…

Unit Testing

- Tests allow us to encode our assumptions in a way that the computer can understand **and** automatically check
- Phrases like "[piece of code] should [do a thing]" can become a unit test
- A typical unit test will:
  - Set up a *minimal* input
  - Invoke the code you want to be tested
  - Test the output/program state to make sure it matches your assumptions

JUnit

```
1  package cse250.examples.debugging;
2
3  import org.junit.jupiter.api.Test;
4
5  public class BreakItDownTest {
6    ArrayList<FarmersMarket> data =
7        BreakItDown.readMarkets(/*...*/);
8
9    @Test
10   void shouldCount75BakedGoods() throws IOException {
11     int count = BreakItDown.countTheBakedGoods(data);
12     assert (count == 75);
13   }
14 }
```

JUnit

```
1  package cse250.examples.debugging;
2
3  import org.junit.jupiter.api.Test;
4
5  public class BreakItDownTest {
6    ArrayList<FarmersMarket> data =
7        BreakItDown.readMarkets(/*...*/);
8
9    @Test
10   void shouldCount75BakedGoods() throws IOException {
11     int count = BreakItDown.countTheBakedGoods(data);
12     assert (count == 75);
13   }
14 }
```

Import the junit package so you can use its functionality

JUnit

```
1 public class BreakItDownTest {
2   ...
3 }
```

- Test cases go in normal class files
- Usually they will be in a separate directory (test instead of src)

JUnit

```
1  @Test
2  void shouldCount75BakedGoods() throws IOException {
3    int count = BreakItDown.countTheBakedGoods(data);
4    assert (count == 75);
5  }
```

- Test cases are *any* normal function, labeled with the @Test annotation
  - Function name does not matter (should still follow good coding style)
  - The return type should be void
  - The function *may* throw exceptions

JUnit

```
1 assert (count == 75);
```

- Your tests should include one or more assertions
  - This is how you encode your assumptions
  - Usually you will use them to check the output of whatever code your test just executed

JUnit

Live Demo

JUnit Advice

- Keep individual test cases (and their inputs) small
  - Try to focus on tests that just test ONE of your functions
  - Tests that test multiple functions working together are still important, but not that useful if you don't have the small ones working first
- If you are stuck, describe your code out loud
  - If you ever find yourself saying: "well this part should…", make sure you have a test that confirms that
- At first, try not to think about implementation details
- Write plenty of your own tests, **don't just rely on ours**