CSE 250

Data Structures

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

Lec 02: Java Refresher

Announcements and Feedback

- Make sure you are on Piazza
- Academic Integrity Quiz due 2/1 @ 11:59PM **(MUST GET 100%)**
- PA0 due 2/1 @ 11:59PM **(MUST GET 100%)**
  - See Piazza post @11 if you are having issues accepting the assignment
- AutoLab rosters are being updated ~nightly

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

```java
package cse250.examples;

class MainExample {
  /**
   * Main function
   * @param args  The arguments to main
   */
  public static void main(String[] args) {
    System.out.println("Hello World");
  }
}
```

Hello World

```java
package cse250.examples;

class MainExample {
  ...
}
```

- 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

```
/**
 * Main function
 * @param args  The arguments to main
 */
```

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

Hello World

```java
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

```
    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

Coding Style is IMPORTANT!!

```
    class neatClass
{
    public static void
  doSomething(String wowwww)
      {
  String weee = "Yes";
  // this is definitely a for loop
      for (char q : wowwww)
        System.out.println(q);
        System.out.println(wee);
      }
  }
```

What the heck is going on here!?

Coding Style - Indentation

```
    class neatClass
{
    public static void
  doSomething(String wowwww)
      {
  String weee = "Yes";
  // this is definitely a for loop
      for (char q : wowwww)
        System.out.println(q);
        System.out.println(wee);
      }
  }
```

What the heck is going on here!?

Where does this function end?

What is in this for loop?

Coding Style - Indentation

```java
class neatClass {
    public static void doSomething(String wowwww) {
        String weee = "Yes";
        // this is definitely a for loop
        for (char q : wowwww)
            System.out.println(q);
        System.out.println(wee);
    }
}
```

Consistent indentation helps convey code structure at a glance!

Coding Style - Indentation

```java
class neatClass {
    public static void doSomething(String wowwww) {
        String weee = "Yes";
        // this is definitely a for loop
        for (char q : wowwww)
            System.out.println(q);
        System.out.println(wee);
    }
}
```

Java doesn't use indentation to determine the body of the loop...

Consistent indentation helps convey code structure at a glance!
...but it has no semantic meaning in Java

Coding Style - Indentation

```java
class neatClass {
    public static void doSomething(String wowwww) {
        String weee = "Yes";
        // this is definitely a for loop
        for (char q : wowwww) {
            System.out.println(q);
        }
        System.out.println(wee);
    }
}
```

Always use braces...it saves you from a lot of annoying errors later

Coding Style - Naming

Use variable names that summarize the variable's role or contents, ie:

- **`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

** also make sure the names stay up to date as you change your code...*

Coding Style - Comments

```
// 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)

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

But what if* it doesn't work…?

** no matter how good you are…there <u>will</u> be a time where it doesn't work*

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

Live Demo

JUnit Advice (see also Piazza @ 8)

- 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**

Combining Unit Testing and Debugging

- Once you have a failing test, if you don't know why...run the debugger!
  - You can step through the code and see what it is **ACTUALLY** doing

When you lose points on AutoLab your first instinct should be:

"I need to write more tests"

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

Exceptions

```java
public List<String> loadData(String filename) {
  List<String> ret = new ArrayList<String>();
  BufferedReader input =
      new BufferedReader(new FileReader(filename));
  String line;
  while( (line = input.readLine()) != null ) {
    ret.add(line);
  }
  return ret;
}
```

Exceptions

```java
public List<String> loadData(String filename) {
  List<String> ret = new ArrayList<String>();
  BufferedReader input =
      new BufferedReader(new FileReader(filename));
  String line;
  while( (line = input.readLine()) != null ) {
    ret.add(line);
  }
  return ret;
}
```

```
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

```java
public List<String> loadData(String filename) {
  try {
    BufferedReader input =
        new BufferedReader(new FileReader(filename));
    String line;
    while ((line = input.readLine()) != null) {
      ret.add(line);
    }
    return ret;
  } catch(IOException e) {
    // Handle the exception, ie print out what went wrong
    e.printStackTrace();
  }
}
```

Catching Exceptions

```java
public List<String> loadData(String filename) {
  try {
    BufferedReader input =
        new BufferedReader(new FileReader(filename));
    String line;
    while ((line = input.readLine()) != null) {
      ret.add(line);
    }
    return ret;
  } catch(IOException e) {
    // Handle the exception, ie print out what went wrong
    e.printStackTrace();
  }
}
```

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

Catching Exceptions

```java
public List<String> loadData(String filename) {
  try {
    BufferedReader input =
        new BufferedReader(new FileReader(filename));
    String line;
    while ((line = input.readLine()) != null) {
      ret.add(line);
    }
    return ret;
  } catch(IOException e) {
    // Handle the exception, ie print out what went wrong
    e.printStackTrace();
  }
}
```

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

Passing Along Exceptions

```java
public List<String> loadData(String filename)
  throws IOException // Communicate the explosive potential
{
  BufferedReader input =
      new BufferedReader(new FileReader(filename));
  String line;
  while ((line = input.readLine()) != null) {
    ret.add(line);
  }
  return ret;
}
```

Passing Along Exceptions

```java
public List<String> loadData(String filename)
  throws IOException // Communicate the explosive potential
{
  BufferedReader input =
      new BufferedReader(new FileReader(filename));
  String line;
  while ((line = input.readLine()) != null) {
    ret.add(line);
  }
  return ret;
}
```

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

JUnit

```java
package cse250.examples.debugging;

import org.junit.Test;
import static org.junit.Assert.*;

public class BreakItDownTest {
  ArrayList<FarmersMarket> data = BreakItDown.readMarkets(/*...*/ );

  @Test
  void shouldCount75BakedGoods() throws IOException {
    int count = BreakItDown.countTheBakedGoods(data);
    assertEquals("Incorrect number of baked goods counted", 75, count);
  }
}
```

JUnit

```java
package cse250.examples.debugging;

import org.junit.Test;
import static org.junit.Assert.*;

public class BreakItDownTest {
  ArrayList<FarmersMarket> data = BreakItDown.readMarkets(/*...*/ );

  @Test
  void shouldCount75BakedGoods() throws IOException {
    int count = BreakItDown.countTheBakedGoods(data);
    assertEquals("Incorrect number of baked goods counted", 75, count);
  }
}
```

Import the junit package so you can use its functionality

JUnit

```java
public class BreakItDownTest {
  ...
}
```

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

JUnit

```java
@Test
void shouldCount75BakedGoods() throws IOException {
  int count = BreakItDown.countTheBakedGoods(data);
  assertEquals("Incorrect number of baked goods counted", 75, count);
}
```

- 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

```java
assertEquals("Incorrect number of baked goods counted", 75, count);
```

Your tests should include one or more assertions

- This is how you encode your assumptions
- Use them to check:
  - The output of functions you just called
  - The state of data structures after function calls
  - That exceptions are thrown when expected
- The message is optional **but highly suggested!**