CSE 250: Java Refresher

Lecture 2

Aug 28, 2024

Reminders

- AI Quiz due Sun, Sept 8 at 11:59 PM.
  - Your final submission must have a score of 1.0 to pass the class.
  - If you can't submit in autolab, let course staff know ASAP.
- PA 0 due Sun, Sept 8 at 11:59 PM.
  - All you need to do is make sure you have a working environment.
  - If you can't submit in autolab, let course staff know ASAP.

Environment

Programming Environment

- IntelliJ

Platform

- MacOS
- Windows
- Ubuntu Linux

Environment

Programming Environment

- IntelliJ

Platform

- MacOS
- Windows
- Ubuntu Linux

You don't have to use this environment, but we may not be able to help you if you don't.

Why Java?

- **Strongly Typed Language**: The compiler helps you make sure you mean what you say.
- **Compiled Language**: Run anywhere, see the impacts of data layout.
- **You know it**: You learned the basics in 116.

Hello World

```
1  package cse250.examples;
2
3  class MainExample
4  {
5    /**
6     * Main function
7     * @param  args    The arguments to main
8     */
9    public static void main(String[] args)
10   {
11     System.out.println("Hello World");
12   }
13 }
```

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

- `String args[]`

  There is a parameter `args` and its type is `array` of `String`.
- `public`

  The function can be called by anyone.
- `static`

  The function isn't tied to an object (e.g., `MainExample.main(...)`).
- `void`

  The function doesn't return anything.

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

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

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

Javadoc comments start with /**.

```
1  package cse250.examples;
2
3  class MainExample
4  {
5     ...
6  }
```

- All code in java lives in a class.
- Classes are organized into packages.

Exceptions

```
1   public List<String> loadData(String filename)
2   {
3     List<String> ret = new ArrayList<String>();
4     BufferedReader input =
5       new BufferedReader(new FileReader(filename));
6     String line;
7     while( (line = input.readLine) != null )
8     {
9       ret.add(line)
10    }
11    return ret;
12  }
```

Exceptions

```
1  public List<String> loadData(String filename)
2  {
3    List<String> ret = new ArrayList<String>();
4    BufferedReader input =
5      new BufferedReader(new FileReader(filename));
6    String line;
7    while( (line = input.readLine) != null )
8    {
9      ret.add(line)
10   }
11   return ret;
12 }
```

```
error: unreported exception IOException; must be caught
or declared to be thrown
```

What are Exceptions

Something goes horribly wrong!

What do **you** do?

Catching Exceptions

```
1   public List<String> loadData(String filename)
2   {
3     List<String> ret = new ArrayList<String>();
4     try {
5       BufferedReader input =
6         new BufferedReader(new FileReader(filename));
7       String line;
8       while( (line = input.readLine) != null )
9       {
10        ret.add(line)
11      }
12    } catch(IOException e) {
13      // handle the situation: e.g., print an error
14      e.printStackTrace()
15    }
16    return ret;
17  }
```

Passing Along Exceptions

```
1   public List<String> loadData(String filename)
2     throws IOException // Communicate the explosive potential
3   {
4     List<String> ret = new ArrayList<String>();
5     BufferedReader input =
6       new BufferedReader(new FileReader(filename));
7     String line;
8     while( (line = input.readLine) != null )
9     {
10      ret.add(line)
11    }
12    return ret;
13  }
```

Coding Style is Important

```
1     class SHAZboT
2                                   {
3       public static void
4   doThings(String ILikeLlamas[])
5                                  {
6     String AString = "No";
7
8     // This is a for loop
9     for(q : ILikeLlamas) System.out.println(q);
10      }
11            }
```

Coding Style is Important

```
1     class SHAZboT
2                                   {
3       public static void
4   doThings(String ILikeLlamas[])
5                                   {
6     String AString = "No";
7
8     // This is a for loop
9     for(q : ILikeLlamas) System.out.println(q);
10      }
11            }
```

What the heck is going on here?

Naming

- `SHAZboT`
- `doThings`
- `AString`
- `ILikeLlamas`

These are all valid variable names, but not helpful to someone reading your code.

Naming

- `SHAZboT`
- `doThings`
- `AString`
- `ILikeLlamas`

These are all valid variable names, but not helpful to someone reading your code.

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

- `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

```
1   class SHAZboT
2   {
3     public static void doThings(String ILikeLlamas[])
4     {
5       String AString = "No";
6
7       // This is a for loop
8       for(q : ILikeLlamas) System.out.println(q);
9     }
10  }
```

Consistent use of indentation is a big help to readers.

Comments

```
1  // This is a for loop
```

Comments should provide information that's not already present in the code. For example:

- Assumptions you're making when writing the code.
- References to relevant documentation/citations.
- Cleaner descriptions of any non-obvious math.
- Explanations of hacks or workarounds; why you're not doing things the "obvious" way.

Brackets

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

vs

```
1  for(q : ILikeLlamas)
2  {
3    System.out.println(q);
4  }
```

Java supports one-line for loops. This is one of the easiest ways to introduce bugs.

Brackets

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

vs

```
1  for(q : ILikeLlamas)
2  {
3    System.out.println(q);
4  }
```

Java supports one-line for loops. This is one of the easiest ways to introduce bugs.

Always use braces.

Imagine you're writing a letter to future-you

Imagine you're writing a letter to future-you

Help future-you (and your TAs/instructors) understand your thought process.

Ways to Succeed

- Never start with code.

Ways to Succeed

- Never start with code.
- What do you have? How is it initially organized?
  - Draw diagrams
  - Try out examples

Ways to Succeed

- Never start with code.
- What do you have? How is it initially organized?
  - Draw diagrams
  - Try out examples
- What do you want? How should it be structured?
  - Draw diagrams
  - Try out examples

Ways to Succeed

- Never start with code.
- What do you have? How is it initially organized?
  - Draw diagrams
  - Try out examples
- What do you want? How should it be structured?
  - Draw diagrams
  - Try out examples
- How do the input and output relate?
  - Connect the diagrams.

Ways to Succeed

- Never start with code.
- What do you have? How is it initially organized?
  - Draw diagrams
  - Try out examples
- What do you want? How should it be structured?
  - Draw diagrams
  - Try out examples
- How do the input and output relate?
  - Connect the diagrams.
- Break the big problem down into smaller ones.
  - If I had "X", I could solve the problem.
  - Separately figure out how to do X.

Ways to Obtain Assistance

- Explain what you've tried.

Ways to Obtain Assistance

- Explain what you've tried.
  - Which test cases fail?
  - What approaches have you tried and what breaks?

Ways to Obtain Assistance

- Explain what you've tried.
  - Which test cases fail?
  - What approaches have you tried and what breaks?
- Explain what you're trying to accomplish and why.
  - Make sure your interlocutor has all the context.

Ways to Obtain Assistance

- Explain what you've tried.
  - Which test cases fail?
  - What approaches have you tried and what breaks?
- Explain what you're trying to accomplish and why.
  - Make sure your interlocutor has all the context.
- Follow coding style guidelines.

If you don't feel comfortable with Java

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

Typical questions:

- **Syntax** (e.g., "How do I break out of a for loop?")
  Ask on Piazza, Office Hours, Recitations!
- **Semantics** (e.g., "How do I insert an item into a linked list?")
  Ask, but help will usually not come in the form of code.

If you don't feel comfortable with Java

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

Typical questions:

- **Syntax** (e.g., "How do I break out of a for loop?")
  Ask on Piazza, Office Hours, Recitations!
- **Semantics** (e.g., "How do I insert an item into a linked list?")
  Ask, but help will usually not come in the form of code.

Most language (syntax) complaints we get are actually about semantics.

Basic Debugging

Demo

Unit Testing

Look for this phrase

`[part of code] should [do a thing]`

Any phrase like this can become a unit test.

Unit Testing

Look for this phrase

`[part of code] should [do a thing]`

Any phrase like this can become a unit test.

A typical unit test

- Set up a minimal input.
- Invoke the code being tested.
- Test the output/program state.

JUnit

```java
package cse250.examples.debugging;

import org.junit.jupiter.api.Test;

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

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

```
1  public class BreakItDownTest {
2    ArrayList<FarmersMarket> data =
3        BreakItDown.readMarkets(/*...*/);
4
5    /*...*/
6  }
```

Test cases are normal class files.

Usually, they live in a separate directory (`test` instead of `src`).

```
1    @Test
2    void shouldCount75BakedGoods()
3            throws IOException
```

A **test case** is any function labeled with the `@Test` annotation.

- The name of the function does not matter
- The return value should be `void`
- The function may throw exceptions.

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

The test case should run the code you want to test, and then call `assert` to confirm that the outputs are correct.

In this case, we already know that there are 75 farmers markets that sell baked goods, so we can check whether the code computes the right value.

Debugging with JUnit

Demo