Generic Classes and Data Structures

CSE/IT 213

NMT Department of Computer Science and Engineering

"Bad programmers worry about the code. Good programmers worry about data structures and their relationships."

— Linus Torvalds

"Of all the trees we could've hit, we had to get one that hits back."

— J.K. Rowling

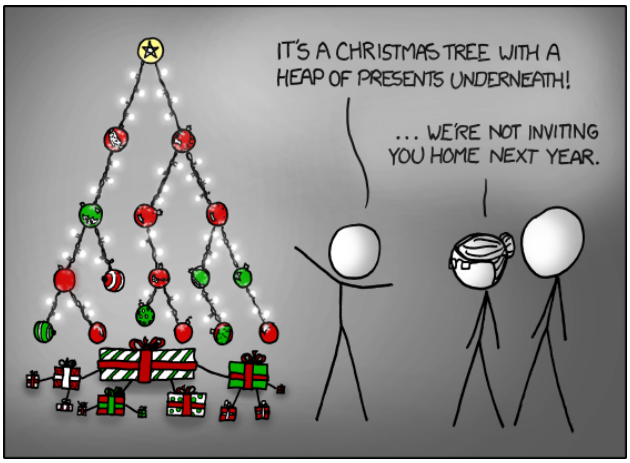


Figure 1: `https://xkcd.com/835/`

Introduction

Generic Classes

You've already seen the Java syntax for instantiating generic classes. For instance, you can use `new ArrayList<String>();` to create an array-list whose elements are of type `String`. The term enclosed between the angle brackets is called a *type parameter*. Type parameters allow you to build new classes from existing ones, by filling in the missing type for a *generic class*.

Generic classes, (also called template classes, in C++), store data with an arbitrary type. We can look at the simplest possible example:

```
1  public class Box<T> {
2      private T contents;
3
4      public Box(T contents) {
5          this.contents = contents;
6      }
7
8      public T unBox() {
9          return contents;
10     }
11 }
```

Figure 2: The `Box` class is a wrapper around any single variable, whose type is set by the parameter `T`

Here, `T` is a placeholder for any type, which will be filled in when a new `Box` variable is declared. It's important to understand that `T` cannot stand in for more than one type at a time. For example, this is not allowed:

```
1  Box<String> ring = new Box<String>("Muhammad Ali");
2  ring = new Box<Integer>(1942);
```

Figure 3: Incorrect use of generic types

When `ring` is first declared on line 1, the type of its `contents` is set to `String`. But the `new Box` on line 2 sets that same type to `Integer`, which makes it different from the original type. In other words, both of the values are `Boxes`, but that doesn't guarantee they are compatible because "`Box`" on its own is not a complete type.

It is also possible for a generic class to have multiple type parameters. For example, we take the concept of a `Box` a step further by storing two values instead of one:

```java
public class Pair<T, S> {
    private T first;
    private S second;

    public Pair(T first, S second) {
        this.first = first;
        this.second = second;
    }

    public T getFirst() {
        return first;
    }

    public S getSecond() {
        return second;
    }

    public Pair<S, T> swap() {
        return new Pair<>(second, first);
    }
}
```

Figure 4: Generic class containing an arbitrary pair of elements

Again, `Pair` by itself would be considered an incomplete type. But when an actual variable is declared:

```
Pair<String, Integer> pear;
```

the type parameters are filled in to define a new class, which gives you a fully fledged type. As you've already seen with array-lists, the most recent version of Java let you leave the type arguments empty on the right hand side of the assignment, since the compiler already knows what classes it needs to fill in.

```java
Pair<String, Integer> pear;
// good:
pear = new Pair("January", 17);
// bad:
pear = new Pair<>(19, 42.08333);
// ugly:
Pair<Integer, Pair<Integer, Pair<Integer, Integer>>> chain =
    new Pair(1, new Pair(2, new Pair(3, 4)));
```

Figure 5: Java knows how to fill in implied type arguments, even in complicated cases (like line 8)

Autoboxing

In Java, *almost* everything is an object. Builtin types are generally defined by classes somewhere in the standard library. However, for performance reasons Java also includes eight *primitive* types. The

primitive data types in Java are `boolean`, `byte`, `char`, `double`, `float`, `int`, `long`, and `short`. These contain no attributes and cannot run any methods, they're simply stored as raw bytes within the program.

These small exceptions are sort of a loophole in the object orientation of the language. One consequence of this is that primitive types can't be used as type arguments for generic classes. It's more useful to the type system to be able to assume that `<T>` is a type that extends `Object`, so the primitive types are left out!

The downside is that the primitive types are also some of the most commonly used in programs. It's very likely that at some point you will want to create an `ArrayList<>` of integers. As a workaround, each primitive data type has an "equivalent" class defined in the Java standard library. These are sometimes called the "boxed" types: `Boolean`, `Byte`, `Character`, `Double`, `Float`, `Integer`, `Long`, and `Short`. Each of these comes with some useful methods to perform common operations for its corresponding data type. For example:

```
Integer.parseInt("123");
Boolean.logicalXor(true, false);
Double.isNaN(2 * 1.6e308 * 0);
```

More importantly, the boxed types are compatible with their primitive counterparts. You can assign an `Integer` to and `int` value, or vice versa. This feature is called *autoboxing*, and it lets you get around the problem of using "primitive" types in generic classes:

```
ArrayList<Integer> numbers = new ArrayList();

for (int i = 0; i < 100; i++) {
    numbers.add(i + 1);
}

int fifty = numbers.get(50);
```

Figure 6: Adding `int`s to an array-list of `Integers`

Inner Classes

It's common for data structures to have a recursive underlying structure. In a linked list, for instance, the structure is achieved by creating a node which contains a reference to another node. No single node can be called a "list", since each node probably contains only one data element. The list comes from having a collection of nodes that are neatly organized to form a linear sequence.

When it comes to object oriented design, the concept of a `Node` class might seem a little awkward. If all you want to see is a list, then the `Node` itself isn't an object that you would be particularly interested in. It's just a building block for the `List` object. Situations like this are a great use-case for *inner classes*:

```java
public class List<E> {
    // Nested class, all attributes are visible only within List
    private class Node {
        private E data;
        private Node next;

        private Node(E data, Node next) {
            this.data = data;
            this.next = next;
        }
    }

    // Use a Node to keep track of the head of a linked list
    private Node head;

    public List() {
        this.head = null;
    }

    public void add(E item) {
        Node newHead = new Node(item, this.head);
        head = newHead;
    }

    public E remove() {
        if (head == null) {
            throw new EmptyStackException();
        }

        E item = head.data;
        head = head.next;
        return item;
    }
}
```

Figure 7: Using nested classes to create a linked list

Java allows you to define a new class within the scope of an outer class. This allows you to create new data types that only the outer class has access to. In the above example, we could also move the inner class into it's own file called `Node.java`, but that would make this package's interface more complex than it needs to be. `Node` doesn't need to be shared with the rest of the program, because `List.java` is the only place where that class is actually relevant.

Nested classes are often the best way to achieve the right encapsulation in your code. Even `private` attributes in a nested class are visible to its outer class, which can make smaller sub-structures easier to work with than separate classes. Especially when defining recursive data structures, inner classes are a convenient pattern to be aware of.

JUnit More Annotations and Features

When writing tests that require and rely on an object through its many tests, It is useful to look at the `@Before`, `@After`, `@BeforeClass` and `@AfterClass` annotations.

@Before

The `@Before` annotation tells the JUnit framework to execute the method the annotation corresponds with before every test.

@After

The `@Before` annotation tells the JUnit framework to execute the method the annotation corresponds with After every test.

A scenario that this can be helpful for is when you need to reset a set of data such as an `ArrayList` or display a message to the console.

```
1  ArrayList<Integer> history = new ArrayList<>();
2
3  @Before
4  public void ensureArrayIsEmpty() {
5      System.out.println("ensureArrayIsEmpty() Array has " + history.size() +
       ↪  " elements");
6  }
7
8  @Test
9  public void testCalcHistoryHasTenResults() {
10     // Test objects method here.
11 }
12
13 @After
14 public void clearArrayAfterEveryTest() {
15     System.out.println("clearArrayAfterEveryTest() clearing ArrayList elements!");
16     history.clear();
17 }
```

Figure 8: An example of using `@Before` and `@After` annotations.

@BeforeClass

Using the `@Before` annotation may not always be the best approach for setting up objects for your tests. Lets say you need to write multiple tests that need to share an object or massive/computationally expensive setup (such as logging into a database). Using `@Before` will provide a large overhead before each test is executed. Therefore, it is useful to utilize the `@BeforeClass` annotation since it tells the framework to run the method only once and before any of the tests methods. It is also worth noting that this can negatively impact the independence of tests, but it is a necessary operation sometimes.

@AfterClass

This annotation is similar to the `@After` annotation but this annotation is executed after all tests have executed. This is used to clean up any external expensive resources you allocated in your `@BeforeClass` annotation method such as closing a database connection.

Note: Methods with the `@BeforeClass`/`@AfterClass` need to be static.

```
private static DatabaseConnection database;

@BeforeClass
public static void login() {
    /* Init the database before testing */
    System.out.println("login() setting up database");
    private String url = "jdbc:postgresql:...";
    private String username = "...";
    private String password = "...";
    database = ...
}

@Test
public void testSumWithDBValues() {
    // Your test here
}

@AfterClass
public static void logout() {
    System.out.println("logout() logging out of database.");
    database.logout();
}
```

Figure 9: An example of using `@BeforeClass` and `@AfterClass` annotations.

@Test(expected = Exception ...)

If we want to verify an object created throws the correct exception, we can utilize the optional "`expected`" parameter inside of `@Test`. If we want to verify `ArrayList` throws the correction exception for trying to access an index of an empty array, we can write:

```
@Test(expected = IndexOutOfBoundsException.class)
public void shouldThrowIOOBException() {
  new ArrayList<Object>().get(0);
}
```

Figure 10: An example of expecting exceptions.

It is worth noting that the optional "`expected`" parameter should be used sparingly. This is because this test will pass if ANY object returns the defined exception. Better ways of handling and under-

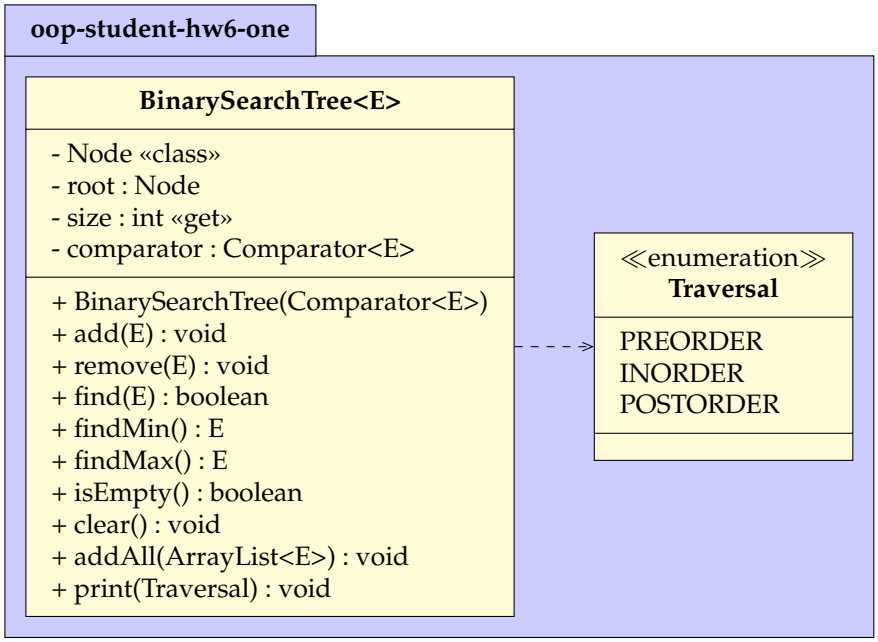
standing the state of the object before throwing the exception can be found on Junit GitHub [1] or official documentation [2].

[1] The JUnit GitHub documentation `https://github.com/junit-team/junit4/wiki/Exception-testing`.
[2] The JUnit official documentation `https://junit.org/junit4`.

Problems

Problem 1: Binary Search Tree

For this problem, you will write a Binary Search Tree to store a sorted collection of Books. Your class should follow this template:



Here, `Node` is an inner class which is managed by the outer `BinarySearchTree` class. It can contain its own and methods to help the outer class insert, search for, and remove elements from the tree:

```
1  private class Node {
2      private E data;
3      private Node left;
4      private Node right;
5
6      private Node(E data, Node left, Node right);
7      private Node add(E data);
8      private Node find(E data);
9      private Node remove(E data, Node parent);
10     ...
11 }
```

Figure 11: Sketch of what the inner `Node` class might look like

`BinarySearchTree` has one constructor. It creates an empty tree (`root = null`, `size = 0`), and sets up a comparator which will be used for searching the tree.

`add()` takes an element and inserts it into the tree. When the new element compares less than or equal to the element in the root node, it should be inserted under the `left` node; and when it

compares greater than the root element it should be inserted under the `right` node. This can be done recursively until either `left` or `right` is `null`, at which point the new node can fill in the empty space. Adding an element also increments the `size` counter.

`remove()` finds the node in the tree containing the given element. Once this node is found, there are three cases to consider:

1. If the node has no children, then it can simply be unlinked from its parent, (`parent.left = null` or `parent.right = null`, depending on its position in the tree).
2. If the node has only one child, then it should be moved up the tree to take the place of its parent, (`parent.left = child`, etc.).
3. If the node has two children, then remove the maximum element from the left subtree, and move it up to replace the position of current node.

Removing an element also decrements the `size` counter.

`find()` takes a possible element, and does a binary search to find it in the tree. If another element compares equal to the given one, return `true`, otherwise return `false`.

`findMin()` returns the element in the left-most node of the tree, and `findMax()` returns the right-most element. When the tree is empty, both methods return `null`.

`isEmpty()` returns `true` if the tree contains no elements, and `false` if it contains at least one.

`clear()` removes every element from the tree, and resets the size to 0. Note that Java is garbage collected, so you don't need to traverse the tree to delete each node. Simply resetting `root` to `null` will allow the JVM to clean up the memory for you.

`addAll()` simply calls `add()` for each element in an array-list.

Finally, `print()` traverses the tree and prints every element on a separate line. This function takes an instance of `Traversal`, which tells you *how* to traverse the tree; either pre-order, in-order, or post-order. Rougly speaking:

- Pre-order means `visit self, go left, go right`
- In-order means `go left, visit self, go right`
- Post-order means `go left, go right, visit self`

You can use `private` helper functions to help traverse the nodes in the tree recursively.

Library

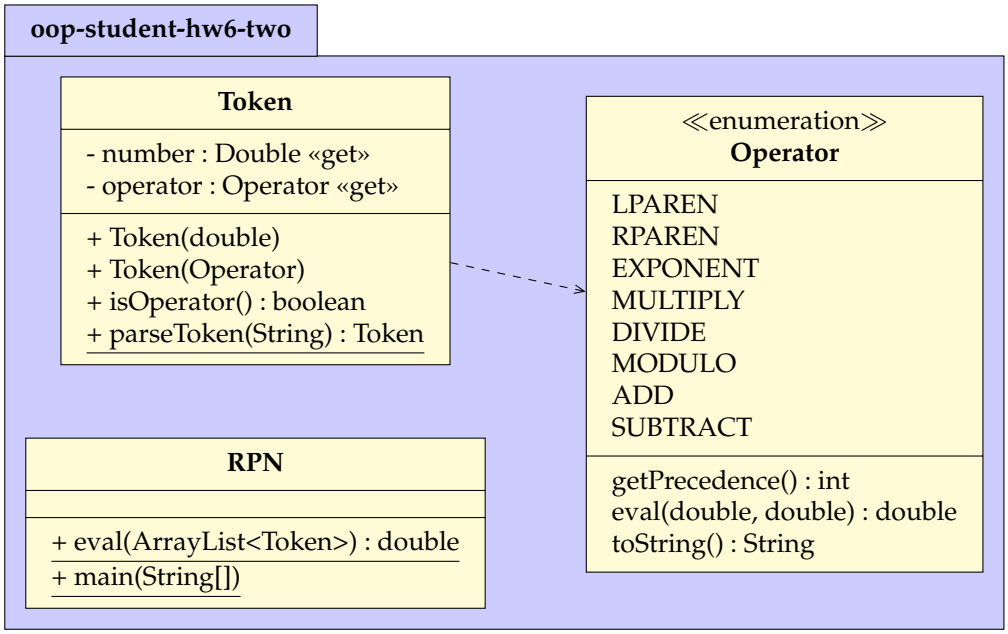
Finally, add a class called `Library.java` to sort a collection of Books. You can include your code from `Book.java` and `Author.java` in homework 5.

In `main()`, create three empty `BinarySearchTrees`, `authors`, `titles`, and `years`. Instantiate each tree with a different comparator, to sort books by author, title, and year respectively. To each tree, add all of the books listed in `library.txt`. Then use the methods of your tree class to print the following information to the console:

- Print the `titles` tree, pre-order
- Print the `years` tree, in-order
- Print the `authors` tree, post-order
- Print the most recently published book in the library
- Remove oldest book from the `authors` tree. Does it contain any books by Miguel de Cervantes? What about William Shakespeare?

Problem 2: RPN Calculator

For this problem, you will write a simple calculator to evaluate an expression in reverse Polish notation. You will need to write three short classes:



Operator

Write an `enum` class, `Operator.java`. The class should differentiate four different math operators, with the following precedence:

Operator	Precedence
`"("` `")"`	3
`"^"`	2
`"*"` `"/"` `"%"`	1
`"+"` `"-"`	0

You can add a private attribute to the `enum` to keep track of the precedence. Add a `getPrecedence()` method so each operator returns the correct value.

Also add a method, `eval()`, so each operator can compute the result of applying that operator to two `double`s. For example, `MULTIPLY.eval(a, b)` should return `a * b`, `DIVIDE.eval(a, b)` should return `a / b`, etc. Within the method you can use `switch (this) { ... }` to determine which instance of `Operator` the current object is. For `LPAREN` and `RPAREN`, `eval()` can simply return `Double.NaN`.

Finally, add a `toString()` to print each operator using the one-character strings shown in the table above.

Token

Write the class `Token.java` to represent either a number or an operator. The class has two constructors:

1. Takes a `double` to set the `number`, and sets the `operator` to `null`.
2. Takes an `Operator` to set the `operator`, and sets the `number` to `null` (a `Double` can be set to `null`, but the primitive type `double` cannot!).

The method `isOperator()` should return `true` if `operator` is not `null`, and `false` otherwise.

Finally, the static method `parseToken()` should take a `String` and convert it to a `Token`:

- If the string is `"("`, `")"`, `"^"`, `"*"`, `"/"`, `"%"`, `"+"`, or `"-"`, create a token containing the corresponding operator.
- Otherwise, if the string can by parsed by `Double.parseDouble()`, create a token containing that number.
- If `Double.parseDouble()` throws a `NumberFormatException`, then catch it and return `null` to indicate the token could not be parsed.

RPN

Write a main class, `RPN.java`, that can read a math expression in reverse Polish notation and calculate the solution. To get started, you can `import java.util.Stack` [3] so you can use the `Stack` class provided by the Java standard library.

First, write a method called `eval()` that takes an array-list of `Tokens` and returns a `double`. You calculate the result using this algorithm:

1. Create an empty stack of `Doubles`

[3] The `Stack` class is a part of the Java collections framework, which is usually a good place to start when looking for data organization tools. A summary of the data structures available in the collections framework can be found here: `http://docs.oracle.com/javase/8/docs/technotes/guides/collections/reference.html`

2. For each token in the list:
   (a) If the token is **not** an operator, push the number onto the stack
   (b) Otherwise pop two values off the stack, (first b, then a), then push the result of `operator.eval(a, b)` back onto the stack

At the end of this process the stack should contain **exactly 1** number – if not, then the input was not a valid RPN expression. The method can also fail if the stack is empty before you try to call `pop()`. In either case, the method should throw an exception:

```
throw new ArithmeticException("Invalid RPN Expression!");
```

If the calculation succeeds, pop the last number off the stack and return it.

Add a `main()` method that uses a while loop to prompt the user for RPN expressions, and prints the calculated results. The input should be a space-separated line of numbers and operators (with no parenthesis). Split the line into words, and convert the array of strings into an array-list of tokens that you can pass to `eval()`.

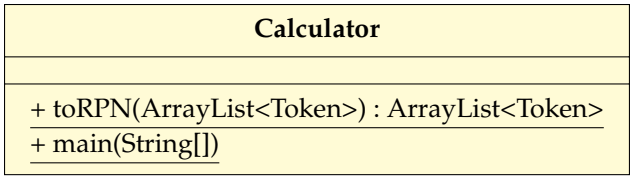
If `eval()` throws an exception, catch it and print an error message, then prompt again for the next expression. Keep scanning until you hit `EOF` (the user entered `<Ctrl>-D`), then exit.

Example output:

```
RPN Expression> 3 4 +
>>> 7
RPN Expression> 3 4 + 5 *
>>> 35
RPN Expression> 3 4 + * 5
Error! Invalid Expression!
RPN Expression> 3 4 - 5 /
>>> -0.2
RPN Expression> <Ctrl>-D
```

Problem 3: Calculator

RPN expressions are particularly easy to evaluate with computer programs, but are generally more difficult for humans to read and write. For this problem, you will write a general purpose calculator by translating normal (infix notation) expressions into reverse Polish notation. You can do this by implementing Dijkstra's shunting-yard algorithm [4].



[4] For more details, see: `https://en.wikipedia.org/wiki/Shunting-yard_algorithm`

Write a class called `Calculator.java` containing two static methods. The first, `toRPN()`, takes an array-list of `Tokens` and returns a second list. It will implement the following algorithm:

1. Create an empty list of `Tokens` for the output expression
2. Create an empty stack of `Operators`
3. For each `Token` in the input expression:
   (a) If the token is a number, add it to the end of the output expression
   (b) If the token is an operator, (other than `LPAREN` or `RPAREN`), then look at the operator stack:
      i. While the top of the stack has an operator (not including parentheses) whose precedence is *greater than* that of the token, or if the two tokens are both `EXPONENT` [5], pop the operator from the stack and add it to end of the output expression
      ii. When done looping, push the current token onto the operator stack
   (c) If the token is `LPAREN`, push it onto the operator stack
   (d) If the token is `RPAREN`, go to the operator stack:
      i. While the operator at the top of the stack is *not* an `LPAREN`, pop an operator off the stack and add it to the end of the output expression
      ii. When you reach the `LPAREN`, pop it from the stack, but do not add it to the output expression
      iii. If you reach the bottom of the stack without finding an `LPAREN`, the expression contains mismatched parentheses. Throw an `ArithmeticException`.
4. When done reading the input, pop each of the remaining operators from the stack and add them to the end of the output expression

The `main()` method works the same as in `RPN.java`. Prompt the user for a expressions, and convert the words on each line into an array-list of `Token`. Evaluate the expression by using `toRPN()` to convert the tokens to RPN format, then use `RPN.eval()` to compute the final result. If the parsing or evaluation steps fail, print an error message and continue. Keep scanning for input until the user sends an `EOF`.

Example output:

```
Expression> 3 + 4
>>> 7
Expression> (3 + 4) * 5
>>> 35
Expression> (3 + 4 * 5
Error! Invalid Expression!
Expression> (3 - 4) / 5
>>> -0.2
Expression> <Ctrl>-D
```

[5] `EXPONENT` is a special case because it's the only right-associative operator.

JUnit Tests

Write a test class for your RPN class. Inside the test class, declare a static `ArrayList` of Token objects you wrote `ArrayList<Token>`. Your test class should include:

1. A `static` method with the `@BeforeClass` annotation. This method should initialize your global `ArrayList` and should be empty.
2. A `static` method with the `@AfterClass` annotation. This method should print a message stating the end of the test class to the console and clear your `ArrayList`.
3. `@Test` methods that test your addition, subtraction, multiplication and division of your rpn calculator.
   - In each test method be sure to clear your `ArrayList` first and then add your tokens to the list to test the appropriate functionality of your calculator. For example, if you want to test 3 + 4 (in rpn it is 3 4 +) in one of your tests:

     ```
     1 arrayListName.clear();
     2 arrayListName.add(Token.parseToken("3"));
     3 arrayListName.add(Token.parseToken("4"));
     4 arrayListName.add(Token.parseToken("+"));
     ```

     When asserting your tests, be sure to call on the `RPN.eval()` method.
4. A test with the optional "`expected`" parameter defined with `ArithmeticException`. Invoke an `ArithmeticException` in your method due the result on your stack not being exactly 1 number.

Submission

Make sure that you have Javadoc style comments for every class and method in your source code, as described in Homework 0. Document any unresolved bugs in the Javadoc comments for each of your classes. When you are satisfied that your code is complete, create a TAR file containing all of the source code for this assignment called:

`cse213_<firstname>_<lastname>_hw6.tar.gz`

Upload your submission to Canvas before the due date.