CS 1027b - Computer Science Fundamentals II
Assignment 4
Due ~~Friday July 23~~ at 11:59 PM

1 Overview

In this assignment, you will create a game, similar to TyperShark[1]. In TyperShark, sharks continuously approach your character, each of which has a specific word displayed above it. To prevent the sharks from attacking you, you must quickly type the words associated with them, at which time they disappear.

In our version of this game – *TyperRacer* – sharks will be replaced with obstacles on the road as you cruise the streets. Each obstacle will have a word above it, and you can either choose to type the words to remove the obstacles, or simply dodge them.

2 Learning Outcomes

This assignment will provide you with experience in:

- Creating and using tries
- Implementing interfaces and employing polymorphism
- Using the public API of a set of classes provided to you
- Creating custom exception classes and handling exceptions
- Writing recursive algorithms
- Javadoc commenting and good Java coding style
- Getting marks for playing games

Part I

Trie Data Structure

3 Introducing the Trie

In the first part of the assignment, you will create a class which implements a *trie* data structure. A trie is a tree structure that allows one to efficiently store and retrieve a list of words[2], as well as determine whether or not a given prefix exists in the trie.

This will be useful to us as we will need a data structure in which to store the words that appear above our obstacles on the screen. For example, suppose that we have four words on the screen: *Alice*, *Albert*, *Joe*, and *John*. In TyperRacer, we must type the word hovering above an obstacle to remove it from the road. Suppose that we begin by typing `Al`. We require an efficient data structure that can tell us if `Al` is a prefix of any of the words currently on screen. If it is, then we should be allowed to continue typing. If it is not, then we should display an error message.

Figure 1 shows the structure of a trie containing the names of our four words. A trie always has an empty root node, from which descend one or more child nodes. Notice how the trie in Figure 1 efficiently stores the words, minimizing duplication where possible. For instance, `Alice` and `Albert` share the prefix `Al`, and so this prefix is not duplicated in the trie. Instead, the suffixes `ice` and `bert` descend from the single `Al` branch.

[1] http://www.popcap.com/games/free/typershark
[2] In fact, the name *trie* comes from re*trie*val.

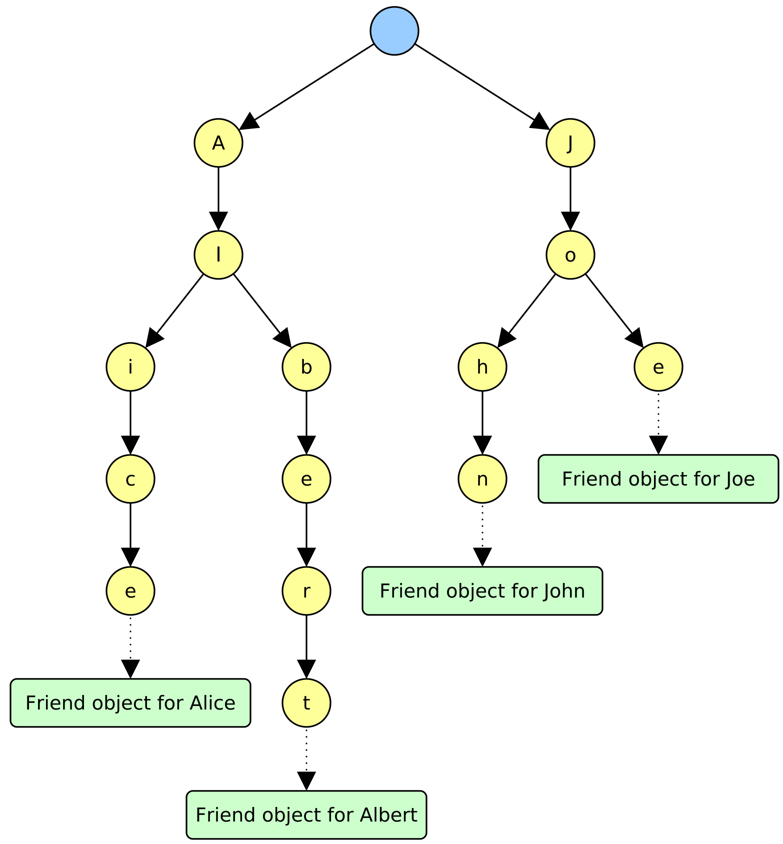

Figure 1: Sample trie

Notice as well that the leaves of our trie store references to some sort of objects related to the words stored in the trie. This will be useful later when you integrate your trie into the provided game code. After all, when the user types the word above a given obstacle in the trie, we wish to have a way to remove that obstacle from the screen. We will therefore store game objects in the leaves, which will contain a method for removing a given object from the screen. As such, we will make our trie class *generic*, so that any possible object type could be stored in the leaf nodes.

You will be provided with the interfaces `TrieADT` and `TrieNodeADT`, as well as the `SmartArray` class (described later). It will be up to you to implement the `TrieADT` and `TrieNode` interfaces. A UML diagram showing the classes involved in Part 1 is shown in Figure 2.

4 Trie Nodes

We will represent our trie as a structure of linked `TrieNode` objects. Figure 3 shows the structure of a `TrieNode`. The following attributes are stored within a `TrieNode`:

- `parent` - A reference to the node's parent node
- `character` - The character stored in the node
- `data` - If the node is a leaf, a reference to some sort of data object (remember: `Trie` and `TrieNode` are generic classes, so we can store any kind of data in the leaves)
- `childCount` - The number of children of the current node
- `children` - A `SmartArray` of `TrieNode` objects representing the node's children

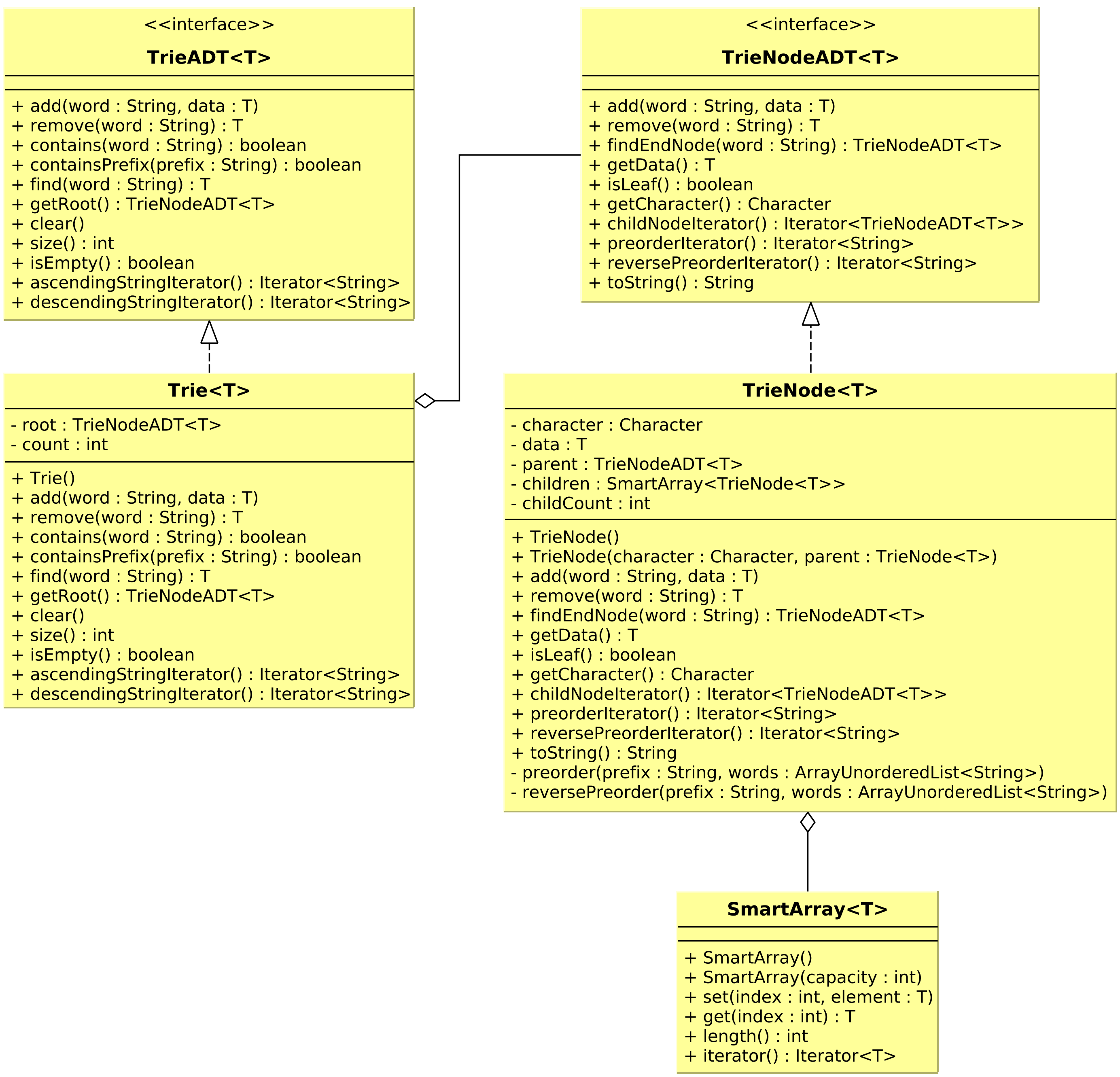

Figure 2: UML class diagram for Part 1

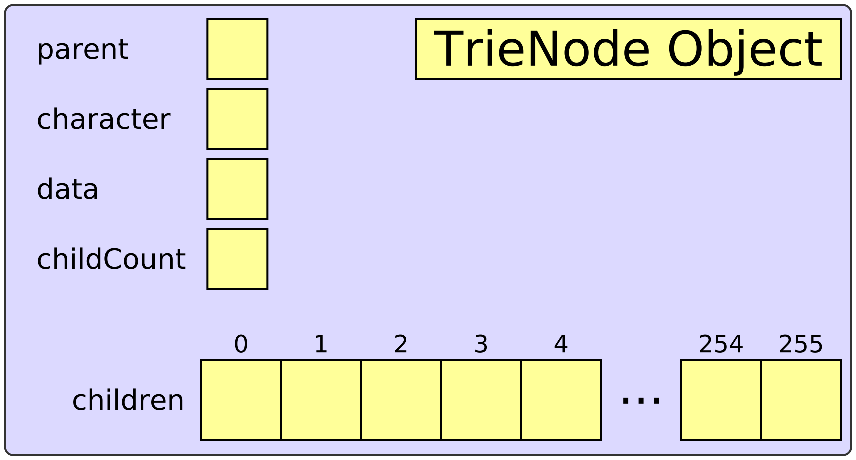

Figure 3: Visualizing a `TrieNode` object

Figure 5 shows a trie storing the words `Ale` and `All`, and storing `Integer` objects at the leaves. This trie might have been created using code similar to that shown in Figure 6. Observe that each node in the trie in Figure 5 stores a character, with the exception of the root node. Additionally, each node stores a reference to its parent node, again excluding the root node. Finally, each node stores a count of its children, and leaf nodes store integers in their `data` instance variables, since the trie was declared to be a trie storing integer data in Figure 6.

Each node stores its children in a `SmartArray` object, which is a simple structure that acts much like an array, but automatically expands itself if an index is passed to its `get` or `set` methods that is past the current bounds of the array. You have been provided with this class. Recall from assignment 1 that each character in a string of text is represented by a unique integer called an *ASCII code*. For instance, the letter `A` is represented by the ASCII code `65`. In Java, we can obtain the ASCII code of a given character simply by casting it to an integer, as shown in Figure 4.

```java
char c = 'A';
int asciiCode = (int)c;   // Stores 97 in asciiCode
```

Figure 4: Converting a character to its ASCII code

For simplicity, we will store each child in our `children` array at the index corresponding to its ASCII code. For instance, in Figure 5, the root has one child – the node representing `A`. Because `A` is represented by the ASCII code `65`, the root stores this node in its `children` array at index `65`. Similarly, the node representing `A` has one child – the node representing `l`. The letter `l` is represented by ASCII code `108`, so this node is stored at index `108`. A full ASCII table is available at `http://www.asciitable.com`. Our `children` arrays will be of length `256`, allowing us to store any ASCII character, including letters, numbers, and symbols.

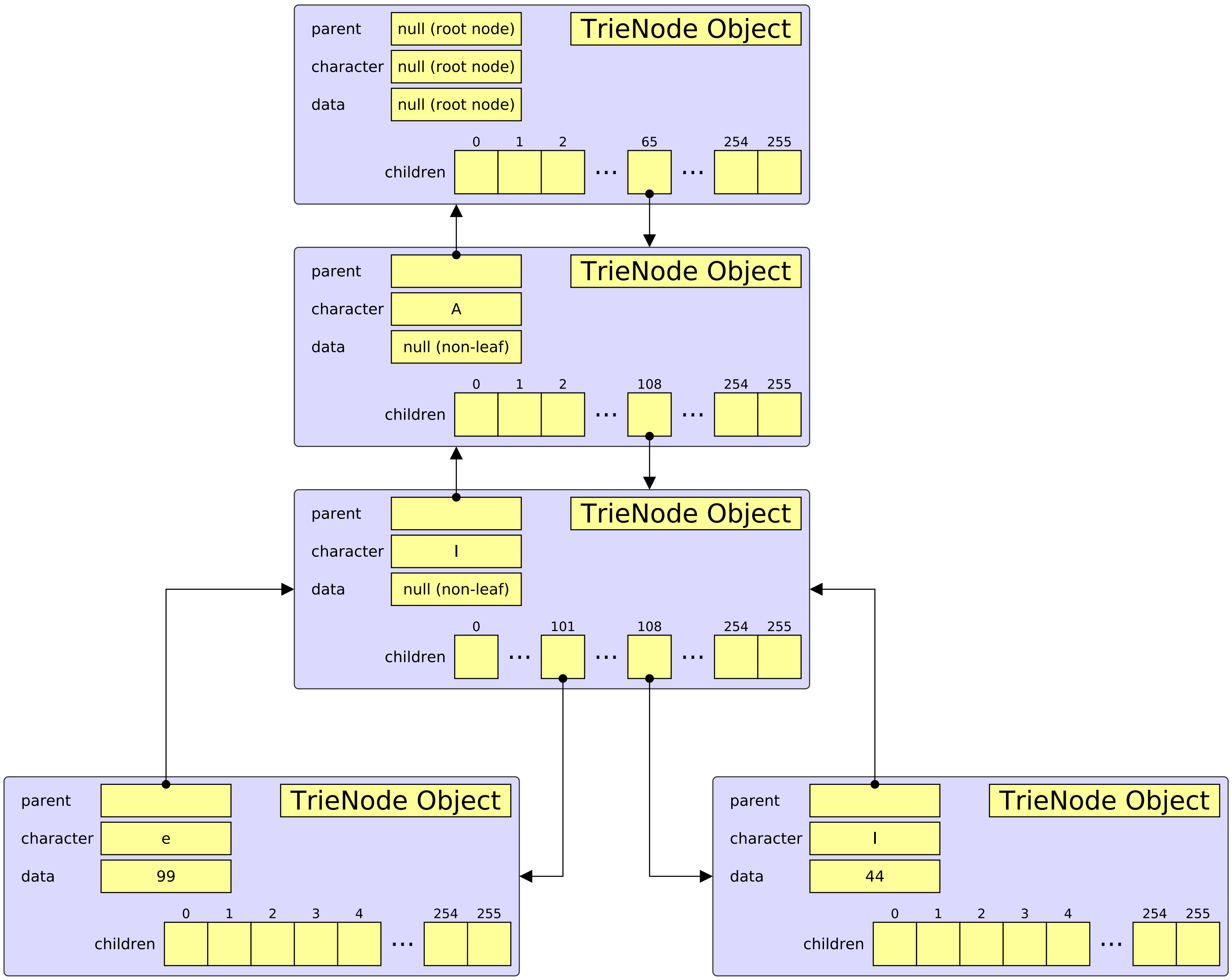

Figure 5: Trie storing the words `Ale` and `All`

```
Trie<Integer> trie = new Trie<Integer>();
trie.add("Ale", 99);  // Add Ale to the trie, and store 99 at its leaf
trie.add("All", 44);  // Add All to the trie, and store 44 at its leaf
```

Figure 6: Code to create the trie shown in Figure 5

5 Adding a Word to a Trie

As with the other tree structures that we have studied, many operations on a trie have simple and elegant recursive solutions. Let's take a look at an example of how we might recursively add the word `John` to a trie. Suppose we create a trie storing `Person` objects at its leaves, as shown in Figure 7.

```
Trie<Person> trie = new Trie<Person>();
Person p = new Person("John","Smith");

// Add the word "John" to the trie, storing the Person object referenced by p at its leaf
trie.add("John", p);
```

Figure 7: Adding `John` to an empty trie of `Person` objects

Recall from Figure 2 that the `TrieNode` class has a method `add` which takes a word and a data object to store at the leaf of the newly added branch. Our `add` method in the `Trie` class can simply call this method on the root node of the trie, passing in the parameters that were passed to it. Figure 8 shows the process of recursively adding `John` to an empty trie. The steps taken in the diagram are described below.

1. The user calls the `add` method in the `Trie` class, passing in the string `John`, and a `Person` object to be stored at the leaf node of the word
2. The `add` method in `Trie` simply calls the `add` method on the root node of the trie, passing in the parameters it received from the user.
3. The root node examines the first character of the string to be added, and determines that it does not have a child node representing `J`. It creates one, stores it in its `children` array, and then calls `add` on the new child node, passing in `ohn` and the data to be stored at the leaf.
4. Node `J` creates a new child node representing `o`, stores it, and calls `add` on the new child, passing in `hn` and the data to be stored at the leaf.
5. Node `o` creates a new child node representing `h`, stores it, and calls `add` on the new child, passing in `n` and the data to be stored at the leaf.
6. Node `h` creates a new child node representing `n`, stores it, and calls `add` on the new child, passing in the empty string, and the data to be stored at the leaf.
7. Node `n` receives the empty string, so it determines that it is the leaf node in the string. It therefore stores the data passed to it (the `Person` object) and returns. Based on this series of recursive calls, what is the base case?

It turns out that two other recursive methods used in the `TrieNode` class – `remove` and `findEndNode` – both follow the same formula described above, although different actions are taken at each node depending on the algorithm.

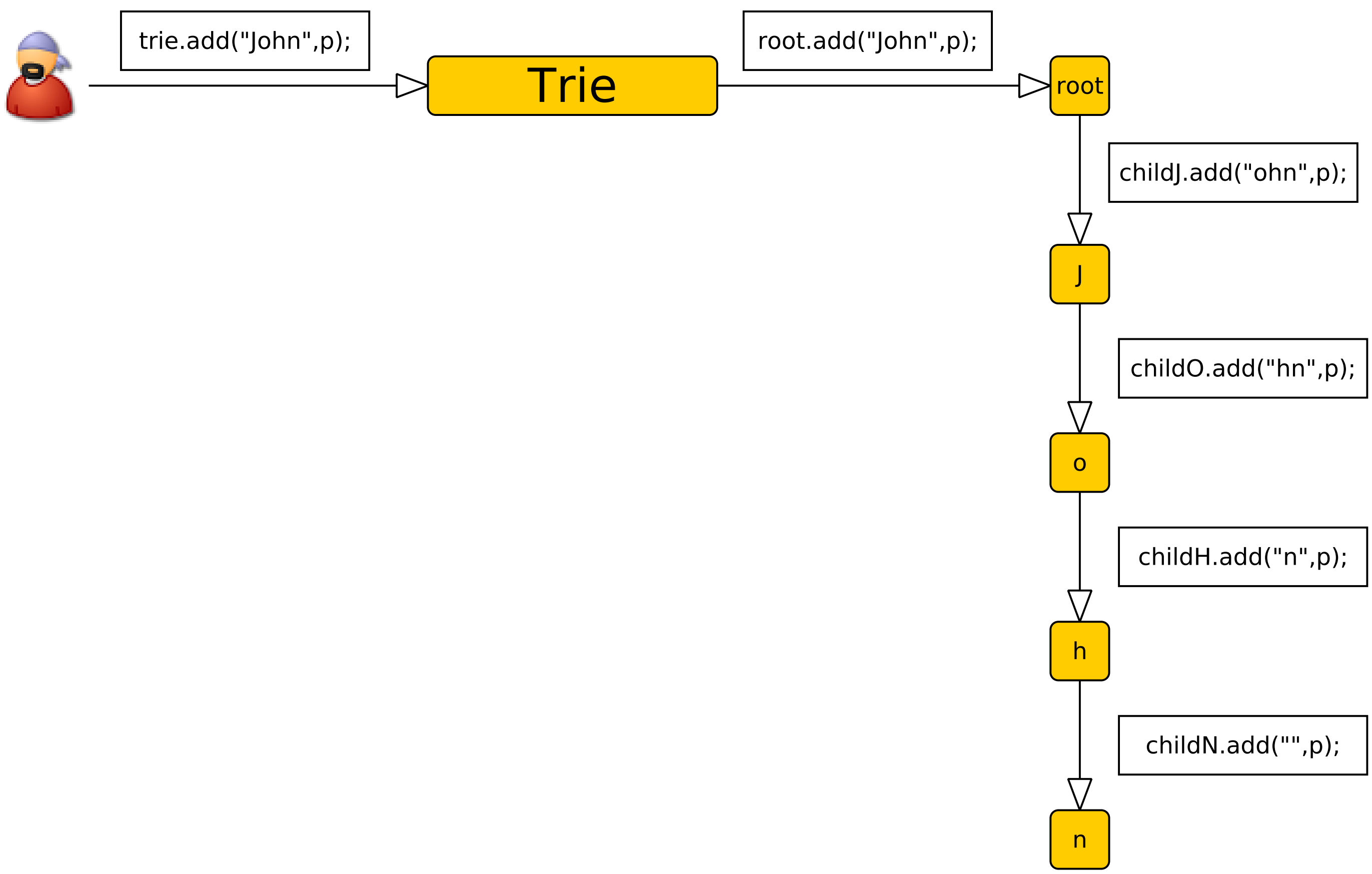

Figure 8: Adding `John` to an empty trie

6 Classes To Implement

6.1 `TrieNode`

The `TrieNode` class must implement the `TrieNodeADT` interface and must be generic. You must add all methods and instance variables shown in Figure 2. The methods are described below, but you must obtain their signatures from the UML diagram.

6.1.1 Methods to Implement

- Default constructor that initializes all instance variables to sensible defaults
- Constructor that takes a `Character` (not a `char`) and a parent node, and initializes all other instance variables to sensible defaults
- `add` method
  - Implement the algorithm described in section 5
  - Note that a node stores the data object passed to it **only** if it is a leaf node
  - At each node, you may need to create a new child node if it does not already exist
  - Be sure to update the appropriate instance variables to reflect the new state of the node after a child has been added
- `remove` method
  - Removes the specified word from the trie. Follow the same basic algorithm as in section 5
  - Returns the data stored at the leaf node of the removed word branch
  - Be sure to update the appropriate instance variables to reflect the new state of the node after a child has been removed
- `findEndNode` method
  - Returns the last node in the branch identified by the specified word
  - Note that this may or may not be a leaf node since the user might be searching for a prefix
  - For instance, if we call `findEndNode("A")` on the root node of the trie in Figure 5, the method would return the node representing `A`
  - If we call `findEndNode("Ale")` on the root node of the trie in Figure 5, the method would return the node represented `e`
  - If the word does not exist in the trie, returns `null`
  - Follow the same basic algorithm as in section 5
- `getData` method
  - Accessor for the `data` instance variable
- `isLeaf` method
  - Accessor indicating whether or not the node is a leaf
  - You do **not** need a new instance variable for this – you can determine whether or not the node is a leaf based on your existing instance variables
- `getCharacter`
  - Accessor for the character stored by the node. Note the return type in the UML diagram

- preorderIterator()
  - Returns an iterator over all words stored in the trie
  - This method should make use of the recursive preorder method
  - This is similar to the iterator methods seen in the LinkedBinaryTree class
  - Note that a preorder traversal of the trie will return the words of the trie in ascending lexographical order
- preorder method
  - Takes a String prefix and an ArrayUnorderedList of String objects to which we will add the words stored in the trie
  - Recursively populates the list passed to it with all the words stored in the trie
  - Similar in spirit to the traversal methods seen in the LinkedBinaryTree class
  - If we reach a leaf, we add the word to the list and return
  - Otherwise, the current node adds its character to the prefix
  - It then iterates from left to right over its children, recursively calling preorder on them, passing in the new prefix, and the word list
  - See Figure 9 on page 10 for an example of how preorderIterator and preorder work together
- reversePreorderIterator method
  - Same as preorderIterator, except this method must use the reversePreorder method
  - Note that a reverse preorder traversal of the trie will return the words of the trie in descending lexographical order
- reversePreorder method
  - Same as preorder, except we iterate right-to-left over our children, and we recursively call reversePreorder
  - Note that this is still a preorder traversal – whether it is *root-left-right*, or *root-right-left* – it is still a preorder traversal
- childNodeIterator method
  - Returns an Iterator over the node's children
  - Note that the Iterator **must store** TrieNodeADT references and not TrieNode references
- toString method
  - Returns the character represented by the node as a String
  - If the node stores no character (i.e. the root), returns null

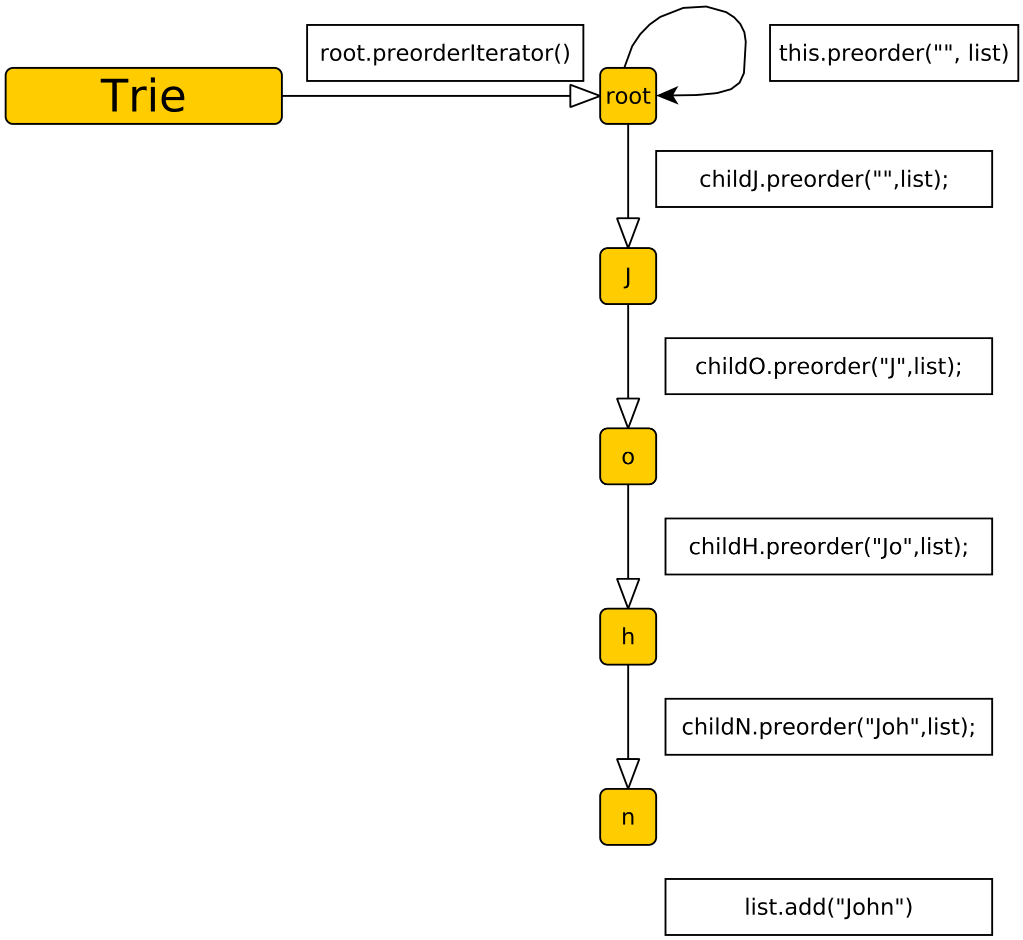

Figure 9: Visualizing the recursive `preorder` traversal of a trie

6.2 Exception Classes

Create three `Exception` classes:

- `StringExistsException`
- `StringNotFoundException`
- `InvalidStringException`

Each class should have a constructor which takes an error message and passes it to its superclass.

6.3 `Trie`

The `Trie` class must implement the `TrieADT` interface and must be generic. You must add all methods and instance variables shown in Figure 2. The methods are described below, but you must obtain their signatures from the UML diagram.

6.3.1 Methods to Implement

- Default constructor that initializes a new empty trie (i.e. with a root node)
- `getRoot` method
  - Accessor for the root node. This should return a `TrieNodeADT<T>`
- `clear` method
  - Removes all nodes, leaving the empty trie (i.e. with a root node)
- `add` method
  - Adds the word to the trie
  - If the word passed is empty, null, or contains a character with an ASCII code greater than 255, an `InvalidStringException` should be thrown
  - If the trie already contains the word **as a prefix** (this is important: use your `containsPrefix` method, and not your `contains` method), a `StringExistsException` should be thrown
- `remove` method
  - Removes the word from the trie and returns the data stored at its leaf node
  - If the word passed is empty, null, or contains a character with an ASCII code greater than 255, an `InvalidStringException` should be thrown
  - If the trie does not contain the word, a `StringNotFoundException` should be thrown
- `contains` method
  - Returns a Boolean value indicating whether or not the trie contains the specified word
  - For instance, for the trie in Figure 5, this method would return `false` for `Al`, but `true` for `Ale`
- `containsPrefix` method
  - Returns a Boolean value indicating whether or not the trie contains the specified prefix
  - For instance, for the trie in Figure 5, this method would return `true` for `Al`, `true` for `Ale` (a word is a prefix of itself), but `false` for `At`
- `find` method

  - Finds the given word in the trie and returns the data stored at its leaf
  - If the specified word is not found, a `StringNotFoundException` should be thrown

- `size` method
  - Returns the number of words stored in the trie

- `isEmpty` method
  - Whether or not the trie is empty (contains no words – it will still have a root node even if it is empty)

- `ascendingStringIterator` method
  - Returns an iterator over the strings stored in the trie in ascending lexographical order

- `descendingStringIterator` method
  - Returns an iterator over the strings stored in the trie in descending lexographical order

7 Testing Part 1

Before integrating your trie structure, you will want to make sure that it is fully tested and working. To help you with this, a visualization tool has been provided for your use. This tool allows you to add and remove words from your trie, all the while displaying your trie graphically. In addition, the tool allows you to check if words and prefixes are contained within your trie, and also tests your ascending and descending iterators. This tool – particularly the trie visualization – should help you to determine if any problems are occurring in the algorithms you write for part 1.

To use the tool, you will need to download `TrieVisualizer` and `jung.jar` from the assignment 4 web page. You will also need to add `jung.jar` as a library on the Java build path in Eclipse, ~~as you did in Lab 5 (exercise 3) / assignment 2~~.

Once you have the code compiling, you may run it and you should see a display similar to that shown in Figure 10.

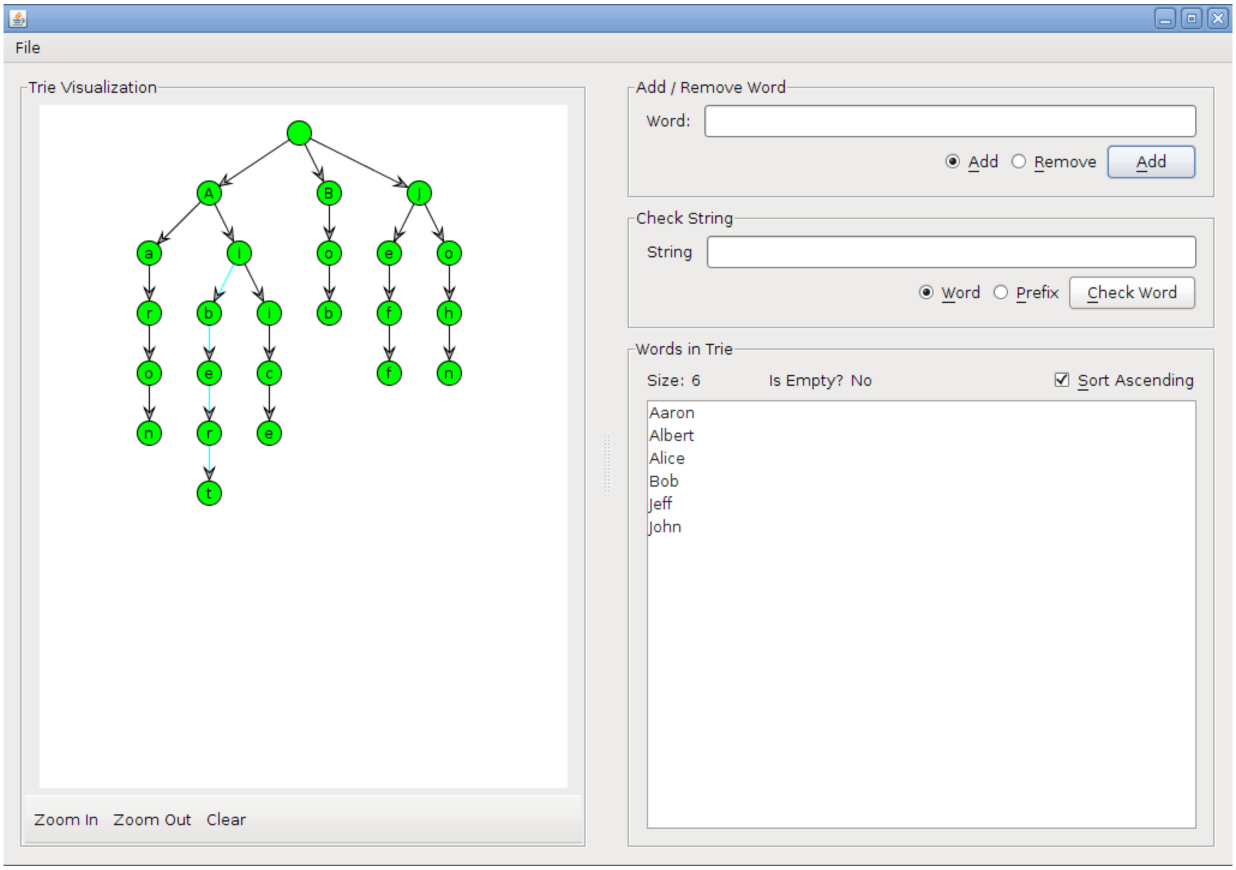

Figure 10: Trie visualizer

Part II

Integrating your Trie into TyperRacer

Now it's time for the fun part – let's create a game! In this part, you will integrate your trie into TyperRacer and you'll get to have some fun testing your assignment. Figures 11 and 12 show sample screenshots from the finished game.

Figure 11: Game splash screen

Figure 12: Game screenshot

Observe in Figure 12 that the current word being typed by the user is displayed in the *HUD* (heads-up display – the bar at the top of the screen). If the user types a prefix found above any obstacle currently in the level, then the current word being displayed should be shown in the HUD. If the user types a prefix that does not exist in your word trie, then the current word being typed should be cleared and the user has to start typing the word all over again. This is why we needed a method in our `Trie` class to check if a given prefix exists in a trie!

8 Dependencies

The TyperRacer game uses the Java 3D library. To set this library up within Eclipse, follow the instructions below. This tutorial assumes that you are using Windows, but is similar for Linux and Mac users.

1. Download the appropriate Java 3D package for your operating system from the assignment web page
2. Unzip the downloaded package
3. Within the downloaded package, there is a file `j3d-jre.zip`. Unzip this file to a directory (and remember where you unzipped it). This tutorial will use `C:\java3d` as an example
4. Open Eclipse and create a new project (or use your existing assignment 4 project)
5. Right-click on the project in the **Package Explorer** and select **Properties**
6. In the left pane, select **Java Build Path**
7. In the right pane, select the **Libraries** tab and click the **Add External JARs...** button
8. Browse to the location where you extracted `j3d-jre.zip` and select each of `vecmath.jar`, `j3dutils.jar`, and `j3dcore.jar`. These can be found within the `lib\ext` subdirectory (e.g. `C:\java3d\lib\ext`). Click **OK**
9. Add the `images.jar` file that you downloaded from the assignment web site as well
10. In the left pane, select **Run/Debug Settings**
11. In the right pane, delete any existing launch configurations that are listed for the project
12. Click the **New...** button
13. Select **Java Application** and click **OK**
14. In the **Edit Configuration** dialog that appears, select the **Main** tab
15. In the **Main Class** field, enter `TyperRacer`
16. Select the **Arguments** tab
17. In the **VM Arguments** field, enter the following: `-Djava.library.path=PATH_TO_BIN`
    Replace `PATH_TO_BIN` with the path to the `bin` subdirectory of the zip file you extracted in step 3, e.g. `-Djava.library.path=C:\java3d\bin`
18. Click **OK** twice

After following these instructions, you should be all set to work with Java 3D.

9 Game Rules

This section will detail the rules of TyperRacer, which you will need to enforce in the `TyperRacer` class that you will write in the next section.

9.1 Adding Obstacles to the Scene

- All obstacles will be assigned random words stored in an `ArrayIndexedList` in your `TyperRacer` class. These words will be loaded from the provided dictionary file
- Initially, 10 obstacles should be added to the scene
- After a threshold number of frames have elapsed, a new obstacle should be added to the scene. The threshold should start at 200 frames (i.e. initially, after 200 frames have elapsed, you will add a new obstacle to the scene). The elapsed frame count will be updated (and the threshold will be checked) in the `tick` method
- When the current level increases, all existing obstacles will remain in the new level, but you must also add ℓ new obstacles, where ℓ is the level number

9.2 Pause / Resume

- When the user presses the spacebar, either pause or resume the game
- The user should not be allowed to move while the game is paused
- The user should not be able to type any words while the game is paused
- See the Javadoc for the `HUD` and `GameWindow` classes and look for methods related to pausing and resuming the game

9.3 Movement

- Adding a negative number to the user's x-coordinate will cause the avatar to move to the left. Similarly, adding a positive number will cause the avatar to move to the right
- When the user presses the left arrow key, move to the left by `0.1` units
- When the user presses the right arrow key, move to the right by `0.1` units
- The user's x-coordinate should always remain within the range `[-2.75, 2.75]`
- The user should never be allowed to move while the game is paused, or when the game is over
- See the Javadoc for the `GameWindow` class to find methods related to getting and setting the user's x-coordinate

9.4 Player Health

- Each time a collision occurs, the player's health should be decremented by `5` units. See the Javadoc for the `HUD` class to learn how to effect this change
- When the player's health reaches `0`, the game is over
- See the Javadoc for the `HUD` and `GameWindow` classes and look for methods related to ending the game

9.5 Typing Words

- When the user types a letter, it should be added to the current word in the HUD
- If the current word is not a prefix that exists in the trie, then the current word should be cleared
- If the current word is a prefix in the trie, but does not exist as a word, then simply leave the new current word in the HUD
- If the current word is a word in the trie, then clear the word in the HUD, remove the word from the trie, and remove its associated `Obstacle` from the game
- Each time an obstacle is removed, the word count in the HUD should be incremented
- See the Javadoc for the `Obstacle` class for details on removing an obstacle from the game

9.6 Changing Levels

- If the user's `z`-coordinate is greater than or equal to `185`, the level should be increased. See the Javadoc for the `HUD` class to learn how to effect this change, and see the `FrameEvent` class for details on how to get the user's `z`-coordinate
- When the level is increased, add the appropriate number of obstacles to the level as indicated in section 9.1.
- When the level is increased, the frames threshold should be decremented by 10 (but should never go lower than 50)
- When the level is increased, all objects (buildings, road, sky) in the game should be repainted to make it seem as though the user has entered a new level. See the Javadoc for the `GameWindow` class for details

10 `TyperRacer` class

In this part, you will finish the game code that was provided to you. You will need to download the Part 2 code from the assignment web site before beginning.

10.1 Instance Variables

You will require (at least) the following instance variables:

- A `Trie` capable of storing `Obstacle` objects at its leaf nodes. This will store all obstacles currently in the level, along with the words associated with them
- An `ArrayIndexedList` capable of storing `String` objects. This will store all words loaded from the dictionary file provided to you
- The number of frames elapsed since you last received a frame elapsed event
- The threshold number of frames that must elapse before new obstacles will be randomly added to the level

10.2 Methods to Implement

- A constructor that makes a call to `super`, and then proceeds to initialize all instance variables, and load the dictionary (this should be done in a helper method). Set the frames threshold initially to 200
- `public void gameReady()`
  This method will be called by the superclass after it has finished setting up the game. You should initialize the state of the HUD appropriately (see the Javadoc for the `GameWindow` class to find out how to get a reference to the HUD, and then see the `HUD` class itself in the Javadoc). You should also add 10 obstacles to the game and then call the `startGame` method in the superclass. Each obstacle should be added by a helper method that you write
- A helper method to add an obstacle to the level. This would get a random word from the word list, pass the word to the appropriate superclass method to add an obstacle to the scene (see the Javadoc for `GameWindow`) and then add the word and obstacle to your trie
- `public void keyPressed(KeyEvent e)`
  This method is called when the user presses a key (such as pressing an array, or typing the letters of a word). This method **must** use a `switch` statement and should take the following actions:
  - If the spacebar is pressed (key code is `KeyEvent.VK_SPACE`), pause/resume the game. You will need to pause/resume both the HUD, and call the appropriate superclass method to pause/resume the game itself
  - If the user presses the left arrow (i.e. the key code in the passed `KeyEvent` object is equal to `KeyEvent.VK_LEFT`), move the player to the left according to the game rules
  - If the user presses the right arrow (key code is `KeyEvent.VK_RIGHT`), move the player to the right according to the game rules
  - Otherwise:
    * Append the letter pressed to the current word displayed in the HUD
    * Check if the trie contains the current word as a prefix. If not, clear the current word and return
    * Check if the trie contains the current word as a word. If so, remove the associated object from the trie and remove it from the scene (see the Javadoc for the `Obstacle` class). Then, clear the current word and increment the word count in the HUD

- `public void collisionOccurred()`
  Called when the player collides with a game object. Decrement the user's health in the HUD according to the game rules. If the user's health reaches `0`, call the appropriate methods in the HUD and the superclass to enter the *game over* state

- `public void tick(FrameEvent event)`
  Called every so often by the game code. This method should:

  - Add the elapsed frames to your elapsed frames instance variable (the number of elapsed frames is provided to you in the `event` parameter passed to this method)
  - If the number of frames elapsed has exceeded or met the frames threshold, then add a new obstacle to the scene, and set frames elapsed back to 0
  - Check the user's `z`-coordinate and take any next level actions needed as discussed in the game rules

11 Non-Functional Specifications

1. Commenting
   - You must use Javadoc comments for each class and method that you create or modify
   - **All variable declarations** must be commented
   - There should be block comments within the code
2. Use Java conventions and good Java programming techniques, for example:
   - Meaningful variable names
   - Conventions for variable names and constant names
   - Readability: indentation, white space, consistency
   - Constants instead of "magic numbers"
   - Appropriate control structures (`if`, `for`, `while`, etc.)
   - Note that it is considered to be poor programming style to use a `break` in a `for` statement. Use a `while` or `do-while` instead
3. Use good object-oriented and top-down design techniques
   - Encapsulation (class should contain the data and the operations on that data)
   - Information hiding (implementation details should be hidden within the class)
   - Modularity (each method should perform exactly one task)
   - Inheritance (common functionality in multiple classes should be abstracted into superclasses to minimize code duplication)
4. Remember that assignments are to be done individually and must be your own work. If you use any code that is not your own, you must ask permission of your instructor, and, if granted, you must cite the source. Use of code that is not your own must be kept to a minimum

12 Testing

Remember: Your goal in testing is to *break* your program. Try entering invalid input. Try entering no input. If you can crash your program or otherwise cause it to produce undesired results, then so can your TA, so make sure you test your program thoroughly and fix any bugs that you find. Use exception handling where appropriate to prevent your program from crashing.

To get an idea of how your program might work, see the **Sample Output** link posted on the assignment web site.

13 Submission Details

See the course web site for information on:

- Submission instructions
- Clarifications and answers to frequently-asked questions