Java Platform Standard Edition (SE) API specification: https://docs.oracle.com/javase/8/docs/api/

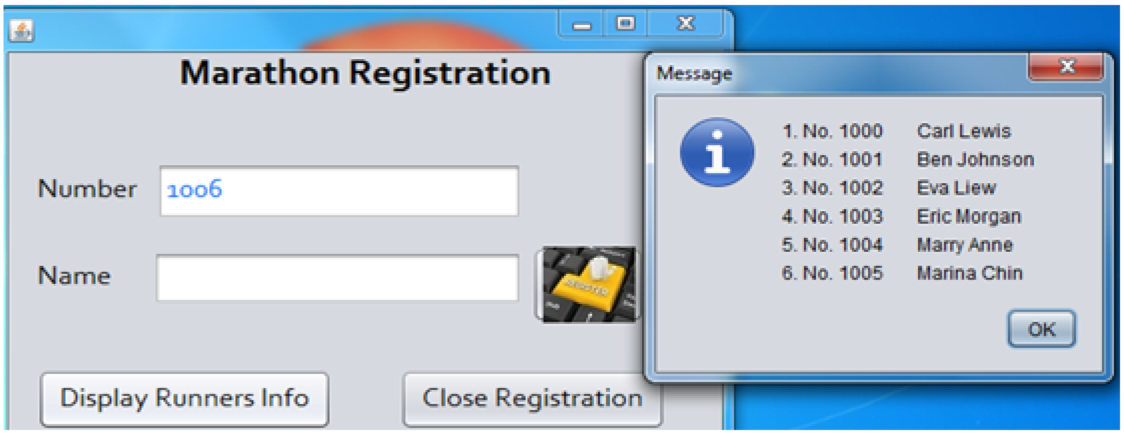
PRACTICAL 1 Applications of ADTs

A. List Applications

1. Your lecturer would like to know the lowest, highest and average of the test scores for this tutorial group. Write a program that would enable your lecturer to do the following using a list:
   - Add the test scores obtained by students from your tutorial group. Assume that the test score is a whole number.
   - Display all the test scores.
   - Find and display the lowest score in the list.
   - Find and display the highest score in the list.
   - Compute and display the average of the scores in the list.

2. For this question, you will use the following Java classes in the **`Chapter1`** NetBeans project's `runningman` folder:
   - `Runner.java`
   - `RecordResults.java`
   - `Registration.java`

   The **`Registration`** class performs registration of marathon runners with the runner number automatically generated. The "Display Runners Info" button will display a pop-up box with the details of the runners who have registered so far (see Figure 1.1).



```
Number 1006
Name
```

```
1. No. 1000   Carl Lewis
2. No. 1001   Ben Johnson
3. No. 1002   Eva Liew
4. No. 1003   Eric Morgan
5. No. 1004   Marry Anne
6. No. 1005   Marina Chin
```

Figure 1.1 Register runner's name and display runner list

The class **`RecordResults.java`** is for recording the runners' numbers as they cross the finishing line. Complete the following methods for the class **`RecordResults`**:

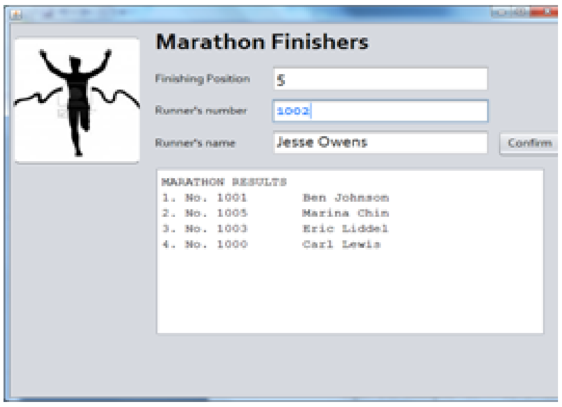
(a) Method **`jtfNumberActionPerformed()`**- refer to Figure 1.2(a) for expected output.

(b) Method **`jtfConfirmActionPerformed()`** - refer to Figure 1.2(b) for expected output.

Extra challenge

Perform the necessary validation, e.g.:

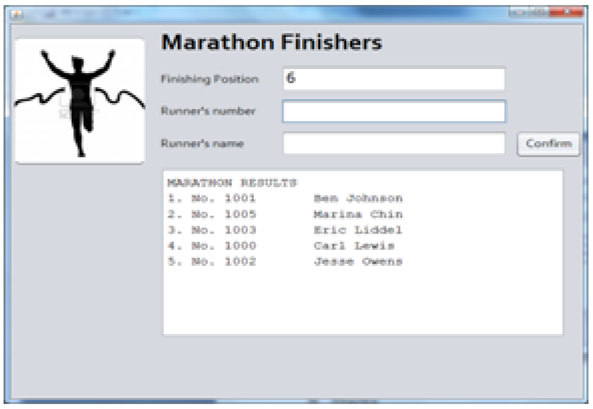
- To check if a runner number that is entered exists
- A runner should only be recorded as crossing the finishing line at most once



Finishing Position: 5
Runner's number: 1002
Runner's name: Jesse Owens

```
MARATHON RESULTS
1. No. 1001        Ben Johnson
2. No. 1005        Marina Chin
3. No. 1003        Eric Liddel
4. No. 1000        Carl Lewis
```

Figure 1.2(a) After**jtfNumberActionPerformed()** runs



Finishing Position: 6

```
MARATHON RESULTS
1. No. 1001        Ben Johnson
2. No. 1005        Marina Chin
3. No. 1003        Eric Liddel
4. No. 1000        Carl Lewis
5. No. 1002        Jesse Owens
```

Figure 1.2(b) After **jtfConfirmActionPerformed()** runs

B. Stack Applications

3. Write a method that uses the *Stack* ADT to evaluate a mathematical expression for proper pairing of brackets, i.e. (), {}, and []. The method returns true if the mathematical expression has balanced pairing of brackets, otherwise returns false.

4. Write a method that uses the *Stack* ADT to evaluate a postfix expression. Assume that the postfix expression to be processed is in the correct form and each operand consists of *a single digit*. For example, "`(6 + 2) / (5 * 3)`" would be entered as "`62+53*/`" by the user.

   Extra challenge
   Create an enhanced version of your solution such that your method is able to handle postfix expressions containing operands consisting of more than 1 digit or even real-valued numbers.

C. Queue Applications

5. A ***palindrome*** is a string of characters (a word, phrase or sentence) that is the same regardless of whether you read it forward or backward – assuming that you ignore spaces, punctuation, and case. Examples of palindromes are "*Race car*" and "*A man, a plan, a canal: Panama!*". Write a program that uses both a *stack* and a *queue* to determine whether a string is a palindrome or not.

6. For this question, you will use the following Java classes in the **`Chapter1`** NetBeans project's `postoffice` folder:
   - **`Customer.java`**
   - **`PostOfficeSim.java`**

   The class **`PostOfficeSim.java`** simulates a post office's single-queue-multicounter environment whereby customers take a number and when their turn comes, their number and the servicing counter will be announced (refer to Figure 1.3 for the GUI).

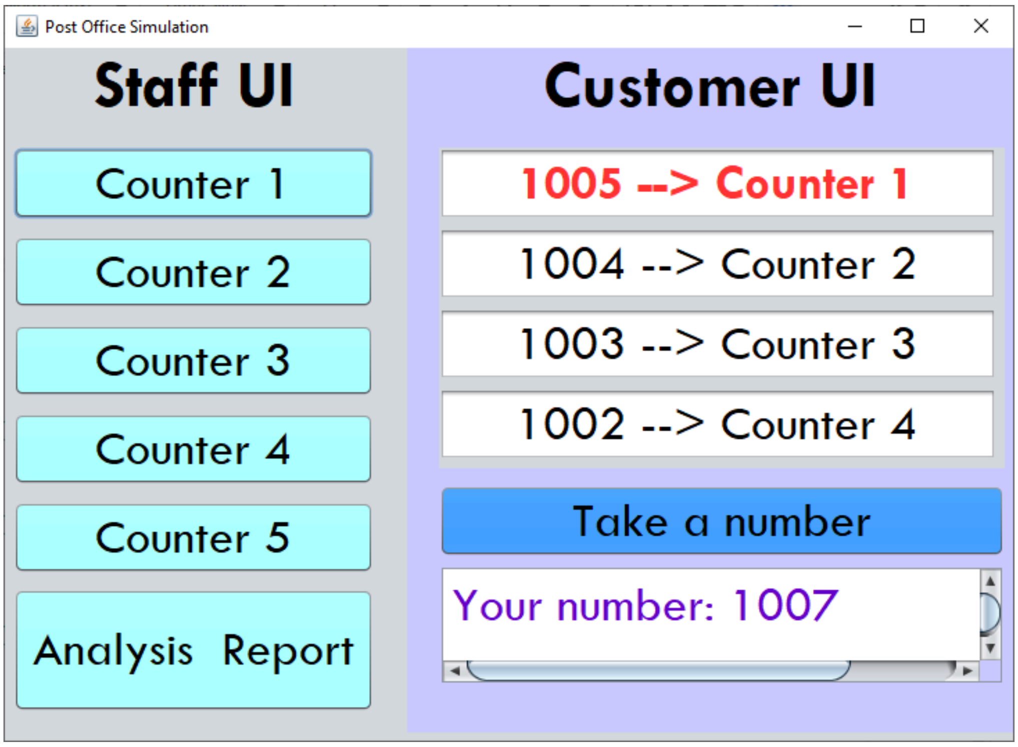


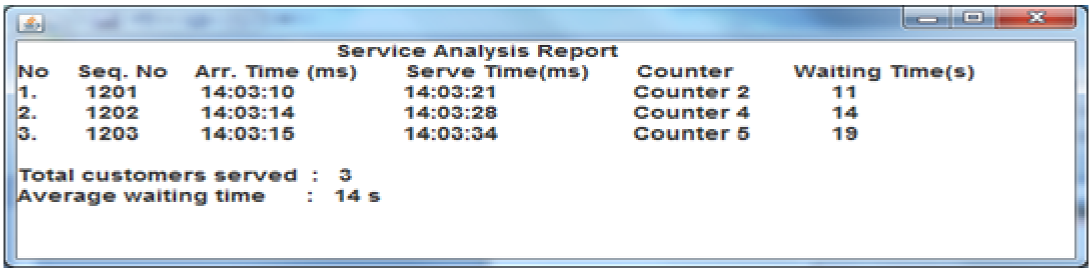
Figure 1.3 GUI for `PostOfficeSim`

Complete the following methods for the class **`PostOfficeSim`**:

a. The event handler for the "Take a number" button. The text area below this button should then display "Your number: …" with the automatically generated next sequence number.

b. **`CounterListener`**'s **`actionPerformed()`**method – to be executed when any of the "Counter *N*" buttons are clicked. Ensure that:
   - The text field in the first row under the Customer UI section will list the next number to be serviced and the assigned counter number.
   - All the remaining text fields below the top row will be updated accordingly.
   - Announce the sequence number to be serviced and the counter number to go to.

c. The event handler for the "Print Analysis" button – to be executed when is clicked. Refer to Figure 1.4 for the expected output.



```
                       Service Analysis Report
No   Seq. No   Arr. Time (ms)    Serve Time(ms)    Counter     Waiting Time(s)
1.    1201      14:03:10          14:03:21         Counter 2       11
2.    1202      14:03:14          14:03:28         Counter 4       14
3.    1203      14:03:15          14:03:34         Counter 5       19

Total customers served  :  3
Average waiting time    :  14 s
```

Figure 1.4 The report that will be printed when the "Analysis Report" button is clicked

PRACTICAL 2 Abstract Data Types (ADTs)

Important: The ADTs in this practical are not **collection ADTs**. Collection ADTs are able to store objects of different types (e.g., the List ADT can be used to store Integers, Strings, Runners, Customers, AudioClips, etc. as illustrated in Practical 1). Therefore, collection ADTs should be defined using generic types to represent the type of objects that will be stored in the collection. In contrast, the Fraction ADT and SquareMatrix ADT covered in Practical can only have int values for their data fields.

Observe that the Java class for the ADT implementation must have at least one appropriate instance variable and appropriate instance methods. If all the variables/methods in the class can/should be replaced with static variables/static methods, then the so-called ADT is not an ADT and the Java class is in effect a utility class.

1. Rational fractions are of the form $\frac{a}{b}$, where a and b are integers and $b \neq 0$.

   a. Write an ADT specification to represent a fraction. Include operations to set and get the numerator and denominator, as well as to perform arithmetic operations (i.e. addition, subtraction, multiplication and division). For the arithmetic operations, the current fraction is the first operand whilst the second operand is passed as a parameter to the operation and the resulting fraction should be returned.

   For parts b and c below, write your code in the **`Chapter2`** NetBeans project's **`fraction`** folder.

   b. Translate the ADT specification from part (a) into a Java interface.

   c. Write a class which implements the Java interface from part (b). Suppose a / b and c / d are rational fractions. Arithmetic operations on fractions are defined by the following rules:

$$\frac{a}{b} + \frac{c}{d} = \frac{(ad + bc)}{bd}$$

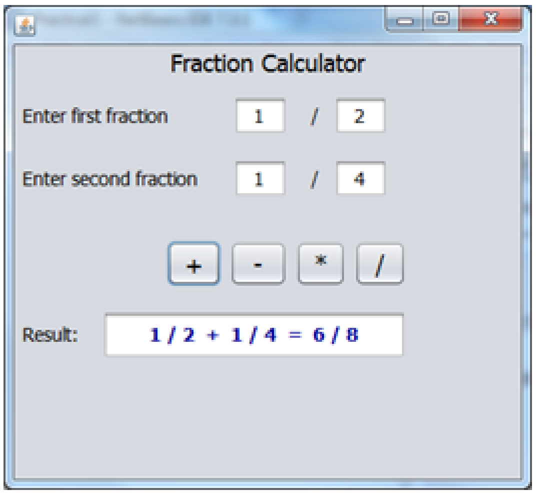
$$\frac{a}{b} - \frac{c}{d} = \frac{(ad - bc)}{bd}$$

$$\frac{a}{b} \times \frac{c}{d} = \frac{ac}{bd}$$

$$\frac{a}{b} \div \frac{c}{d} = \frac{ad}{bc} \text{ where } \frac{c}{d} \neq 0$$

   d. Override the **`toString()`** method so that the fraction's value is returned in the format a / b.

   e. Test your ADT Fraction implementation using the **`FractionCalculator`** class (Figure 3.1) by completing the relevant sections in the class for this application to be fully functional.



```
1 / 2 + 1 / 4 = 6 / 8
```

Figure 3.1 GUI for **`FractionCalculator.java`**

<u>Extra challenge</u>:

Reduce the fraction to the simplest form.

2. Consider an ADT called *SquareMatrix* which represents a two-dimensional array of integers with *n* rows and *n* columns.

   a. Write the ADT specification and include the following operations (parameters are already listed for the first two operations; for the remaining operations you must determine which parameters to use). State any assumptions made.
      - *makeEmpty(m)*, which sets the first *m* rows and columns to zero.
      - *storeValue(i, j, value)*, which stores *value* into the position at row *i*, column *j*. Note that both row and column start from 1.
      - *add*, which adds two matrices together.
      - *copy*, which copies another matrix into this matrix.

   b. Translate the ADT specification from part (a) into a Java interface.

   c. Create a Java class that implements the interface. Override the **`toString()`** method so that the matrix's value is returned in a *n* x *n* layout, for example a 2 x 2 array will appear as:

```
1    2

3    4
```

   d. Create a simple driver program that tests the ADT.

PRACTICAL 3 Efficiency of Algorithms

1. Write a Java program that implements the three algorithms in Chapter 3 (Algorithm A, B and C) and records their average execution time based on 10 iterations for various values of n (e.g., 10, 100, 1000).

   The program should display a table of the running times of each algorithm for various values of n.

PRACTICAL 4 Array Implementations of Collection ADTs

Important: In this practical, observe that **collection ADTs** are able to store objects of different types (e.g., the List ADT can be used to store Integers, Strings, Runners, Customers, AudioClips, etc. as illustrated in Practical 1). Therefore, collection ADTs should be defined using a **generic type** to represent the type of objects that will be stored in the collection. For best practices on writing ADT specifications, as well as Java interfaces and classes for the ADT implementation, please refer to the relevant sections in the Assignment Q&A.

1. Refer to the `adt\ArrayList` class within the `Chapter4` NetBeans project.
   a. Would you be able to add any more entries into the list once you have added the DEFAULT_CAPACITY number of entries?
   b. Discuss how you can rectify the problem identified in part a and then implement the solution.

2. A ***set*** is a collection of *distinct* objects. Typical operations on a set include:
   - add: Add a new element to the set
   - remove: Remove an element from the set
   - checkSubset: Check if another set is a subset of the current set
   - union: Add another set to the current set
   - intersection: Returns a set with elements that are common in both the current set and the given set.
   - isEmpty: Check to see if the set is empty

   a. Write the specification for the Set ADT:
   ADT *Set*
   A Set is ....
   ```
   boolean add(T newElement)
   boolean remove(T anElement)
   boolean checkSubset(Set anotherSet)
   union(Set anotherSet)
   Set intersection(Set anotherSet)
   boolean isEmpty()
   ```

   b. Implement the *Set ADT*. Use array-based implementation. You may include any necessary utility methods.

   c. Write a driver program to test your implementation of the Set ADT.

   d. Modify the Java interface and class from part b such that your implementation provides an *iterator* to the set object. Then, include a method in your driver program that uses the iterator to display all the elements in the set object.

   Note: You only need to implement the **`hasNext()`** and **`next()`** methods of the **`java.util.Iterator`** interface. Refer to Table 4.1 for a description of the methods.

Table 4.1 `java.util.Iterator`

```
public interface Iterator <T> {

    /** Task: Detects whether the iterator has completed its traversal
      * and gone beyond the last entry in the collection of data.
      * @return true if the iterator has another entry to return */
    public boolean hasNext();

    /** Task: Retrieves the next entry in the collection and
    * advances the iterator by one position.
    * @return a reference to the next entry in the iteration,
    * if one exists
    * @throws NoSuchElementException if the iterator had reached the
    * end already, that is, if hasNext() is false */
    public T next();

    /** Task: Removes from the collection of data the last entry that
    * next() returned. A subsequent call to next() will behave
    * as it would have before the removal.
    * Precondition: next() has been called, and remove() has not been
    * called since then. The collection has not been altered
    * during the iteration except by calls to this method.
    * @throws IllegalStateException if next() has not been called, or
    * if remove() was called already after the last call to
    * next().
    * @throws UnsupportedOperationException if this iterator does
    * not permit a remove operation. */
    public void remove(); // Optional method

} // end Iterator
```

PRACTICAL 5 Linked Implementations of Collection ADTs

Important: In this practical, observe that **collection ADTs** are able to store objects of different types (e.g., the List ADT can be used to store Integers, Strings, Runners, Customers, AudioClips, etc. as illustrated in Practical 1). Therefore, collection ADTs should be defined using a **generic type** to represent the type of objects that will be stored in the collection. For best practices on writing ADT specifications, as well as Java interfaces and classes for the ADT implementation, please refer to the relevant sections in the Assignment Q&A.

1. Implement the ADT stack by using a linked chain with an external reference to its top node. Use the interface `StackInterface` from Chapter 4.

2. Implement the ADT queue by using a circular linked chain with only an external reference to its last node. Only one external reference – to the last node – is maintained, since the first node is found easily from the last one.)

PRACTICAL 6 Recursion

1. Write a recursive method **`countUp`** that displays the count-up from 1 to n, where n is a positive integer.

   Hint: A recursive call will occur before you display anything.

2. Write a recursive method that takes two integers and finds the greatest common divisor (GCD).

   Complete the Extra Challenge question in Practical 2 Q1 if you have not done so.

   Hint: Use Euclid's Algorithm.

3. Write code to compare the performance of 3 different implementations of the Fibonacci numbers: using recursion, iteration and array.

4. Write a recursive method **`displayBackward`** that writes a given array backward. Consider the last element of the array first.

   You may add your code to the **`Chapter6`** NetBeans project's **`RecursiveDisplayArray`** class.

5. Write a recursive method **`countNodes`** that counts and returns the number of nodes in a chain of linked nodes.

   You may add your code to the **`Chapter6`** NetBeans project's **`SimpleList`** class.

PRACTICAL 7 Sorted Lists

Important: In this practical, observe that **collection ADTs** are able to store objects of different types (e.g., the List ADT can be used to store Integers, Strings, Runners, Customers, AudioClips, etc. as illustrated in Practical 1). Therefore, collection ADTs should be defined using a **generic type** to represent the type of objects that will be stored in the collection. For best practices on writing ADT specifications, as well as Java interfaces and classes for the ADT implementation, please refer to the relevant sections in the Assignment Q&A.

1. Implement the method `remove()` in `Chapter7\adt\SortedArrayList` class.

   Note on generic types to enable comparison:
   As the entries in this ADT must be in sorted order, the objects in the array must be `Comparable`. Thus, the class that the generic type T represents must implement the interface `Comparable`. To ensure this requirement, we write `<T extends Comparable<T>>`
   instead of just `<T>` after the interface name and the class name. We then can use `T` as the data type of the parameters and local variables within the methods. Hence, the interface could begin as follows: `public interface SortedListInterface<T extends Comparable<T>>`
   Within the implementation class for the sorted list, we then use the `compareTo` method to compare an object with the specified object for order.
   For the constructor, you need to allocate an array of `Comparable` objects. Hence, use
   `new Comparable[aSize]` instead of `new Object[aSize]`

2. Consider the class hierarchy in Figure 7.1 which consists of entity classes providing an oversimplified representation of employee levels in a company.

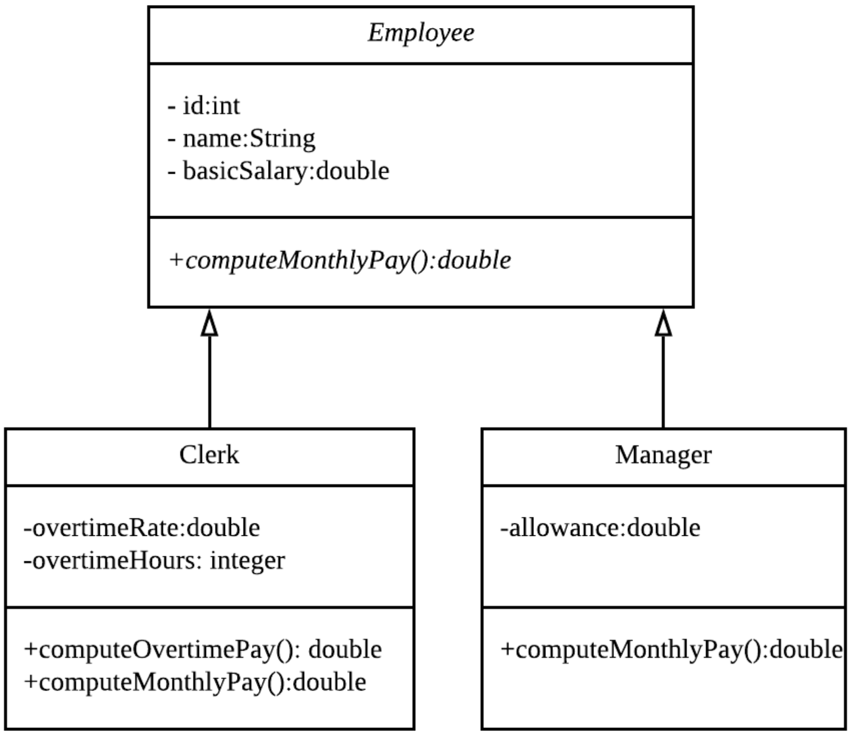


Figure 7.1 Class diagram for employee levels

Q2 (continued)

a. Implement the entity classes shown in Figure 7.1 as follows:
   - The `Employee` class implements the `Comparable` interface based on **`id`** . The `Clerk` and `Manager` inherits the **`compareTo`** method from the superclass `Employee`.
   - In each class, include constructors, setters, getters, and the `toString` method.
   - Override the `equals` method in the `Employee` class. Note that the `Clerk` and `Manager` inherit the `equals` method from the `Employee` class
   - Implement the additional methods in the classes as follows:
     - A clerk's overtime pay is the overtime hours multiplied by the overtime rate.
     - A clerk's monthly pay is the basic salary plus overtime pay.
     - A manager's monthly pay is the basic salary plus allowance

b. Write a client program that creates a sorted list of employees with entries comprising manager and clerk objects using the entity classes from part a.

3. Implement the method **`remove()`** in **`Chapter7\adt\SortedLinkedList`** class.

4. Sorted linked list with iterator.

a. Create a linked implementation of the ADT sorted list which provides an *iterator*. Define the iterator class as an inner class of the ADT.

b. Write a client program that creates a sorted list of employees with entries comprising manager and clerk objects using the entity classes from part a.

c. Use the iterator to display the following monthly payroll report:

```
EmpID      Employee Name   Salary (RM)
2222       Col. Sanders       5900.00
3333       Tony Fey           7127.00
5555       Jack Bauer        10088.88
7777       Lee Hom            7976.55
8888       John Doe           2018.00

Total employees: 5
Total payroll for the month: RM 33110.43
```

PRACTICAL 8 Algorithms for Searching & Sorting

A. Algorithms for Searching

1. When searching a sorted array sequentially, you can ascertain that a given item does not appear in the array without searching the entire array. Modify the method **`contains`** in the `Chapter8\adt\SortedArrayList` class such that it takes advantage of this observation when searching a sorted array sequentially.

2. Write a new array implementation of the `Chapter8\adt\SortedArrayList` class as follows:
   - Provide an *iterative* (for loop) version of the binary search algorithm that will *return the index* of the target entry in the array. If the array does not contain the target entry, the method should return the index of the insertion point as a negative value.
   - Implement the `contains`, `add` and `remove` methods such that they invoke the binary search method to determine the index of the given entry in the array.

.

B. Algorithms for Sorting

3. Implement the iterative selection sort algorithm as a static method in the NetBeans project **`Chapter8`**'s **`SortArray`** class.

4. Implement the iterative insertion sort algorithm (including `insertInOrder`) as Java methods.

5. Implement the recursive selection sort algorithm as a private method using the following method header

```
private static <T extends Comparable<T>>
      void selectionSort(T[] a, int first, int last)
```

   and include the following public method to invoke the recursive method:

```
public static <T extends Comparable<T>>
      void selectionSort (T[] a, int n)
```

PRACTICAL 9 Binary Trees

Important: In this practical, observe that **collection ADTs** are able to store objects of different types (e.g., the List ADT can be used to store Integers, Strings, Runners, Customers, AudioClips, etc. as illustrated in Practical 1). Therefore, collection ADTs should be defined using a **generic type** to represent the type of objects that will be stored in the collection. For best practices on writing ADT specifications, as well as Java interfaces and classes for the ADT implementation, please refer to the relevant sections in the Assignment Q&A.

1. Write a program that takes a postfix expression and produces a binary expression tree. You can assume that the postfix expression is a string that has only binary operators and single-letter operands. After building the expression tree, your program should display the postfix form of the expression.

   Hint:
   Use a stack of binary subtrees to store the subexpressions. Write your client program in the `Chapter9` NetBeans project and use the `adt\BinaryTree` class to declare and create the stack object as follows:
   ```
   Stack<BinaryTree<String>> stack = new Stack<>();
   ```

2. Implement the preorder and postorder traversals of the binary search tree in the `Chapter9\adt\BinarySearchTree` class.

3. Add the following method to the `Chapter9\adt\BinarySearchTree` class to find and return the smallest entry in the binary search tree:
   ```
   public T getSmallestValue()
   ```
   As is often the case with recursive algorithms, implement the actual search as a private method **`findMinNode()`** that the public method **`getSmallestValue()`** invokes. The private recursive method **`findMinNode()`** should find and return the node with the smallest entry in the binary search tree. Use the following method header:
   ```
   private Node findMinNode(Node node)
   ```

Extra challenge

4. A guessing game is a game where the player thinks of something and the program has to guess what it is by asking the player questions that have a yes or no answer. A binary decision tree that grows as the game progresses may be used for this application.

   a. Given the source code for **`DecisionTreeInterface.java`**, create the class(es) for a guessing game. Your game distinguishes between different animals. The user thinks of an animal and the program asks a sequence of questions until it can guess the animal.

      For example, Figure 7.1 below shows the initial decision tree and Figure 7.2 shows a possible dialog where the program wins the game. Your program should learn from the user: if the program makes an incorrect guess, it asks the user to enter a new question that can distinguish between the correct animal and the program's incorrect guess. The decision tree should be updated with this new question. With this new information (i.e. the actual animal in the player's mind and the new question), we augment the tree:
      - we replace the contents of the leaf that contained the wrong answer – "*an Elephant*" in this case – with the new question provided by the player, and
      - we give the leaf 2 children: the left child (the NO answer) is the former leaf ("*an*

Elephant") and the right child (the YES answer) contains the player's answer.

Figure 7.3 shows a possible dialog where the program gives an incorrect guess and Figure 4 shows the subsequent augmented tree.

b. Add code to store the game decision tree to a binary file and retrieve it for future game sessions.
Hint: You need to ensure that your classes for the node and decision tree implements `java.io.Serializable`.

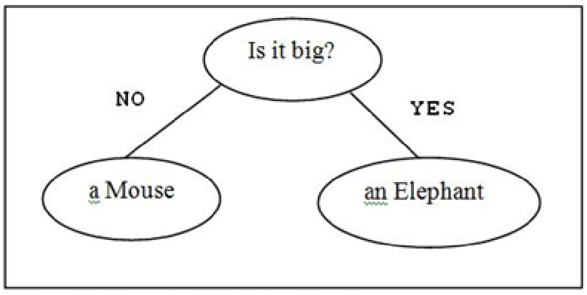
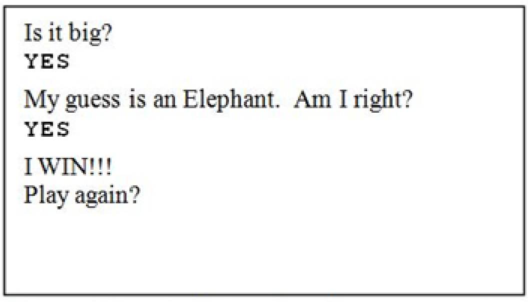
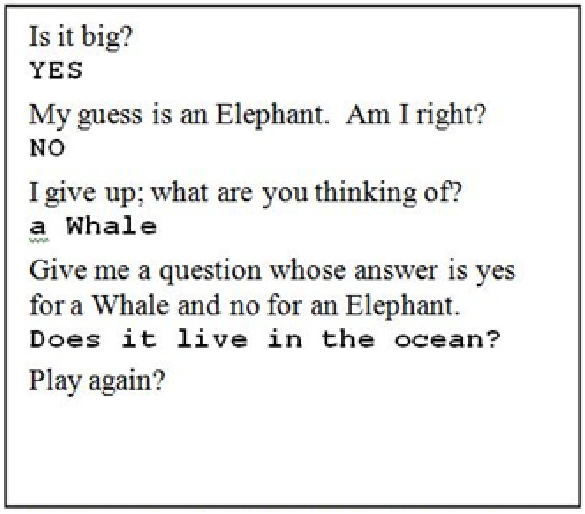


Figure 7.1 Initial decision tree example



```
Is it big?
YES
My guess is an Elephant.  Am I right?
YES
I WIN!!!
Play again?
```

Figure 7.2 Sample dialog for correct guess



```
Is it big?
YES
My guess is an Elephant.  Am I right?
NO
I give up; what are you thinking of?
a Whale
Give me a question whose answer is yes
for a Whale and no for an Elephant.
Does it live in the ocean?
Play again?
```

Figure 7.3 Sample dialog for incorrect guess

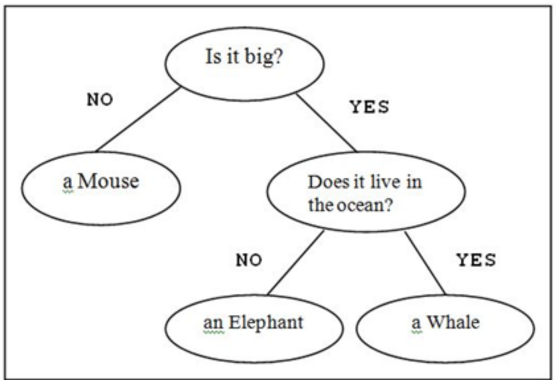


Figure 7.4

PRACTICAL 10 Hashing

Important: In this practical, observe that **collection ADTs** are able to store objects of different types (e.g., the List ADT can be used to store Integers, Strings, Runners, Customers, AudioClips, etc. as illustrated in Practical 1). Therefore, collection ADTs should be defined using **generic types** to represent the type of objects that will be stored in the collection. For best practices on writing ADT specifications, as well as Java interfaces and classes for the ADT implementation, please refer to the relevant sections in the Assignment Q&A.

1. Recall the various techniques for generating hash codes for a string that was covered in the lecture. Override the hashCode method for the **`Name`** class provided in the `Chapter10\`**`entity`** package such that the hash code will produce distinct hash codes for different `Name` objects.

2. Modify the **`Student`** class in the `Chapter10\`**`entity`** package such that the **`id`** is type **`long`** and comprises a 16-digit number. Then, modify the implementation of the method **`hashCode`** such that the *folding technique* is applied as follows:
   - break the key into groups of digits
   - then combine the groups using addition

3. Rewrite the **`HashedDictionary`** class such that the collision-resolution scheme used is *open addressing with double hashing*. Use the following primary and secondary hash functions:
   H_1(key) = key % tableSize
   H_2(key) = 7 – key % 7

Extra challenge

4. Enhance your solution for Practical 1 Question 4 (which is regarding the evaluation of a given postfix expression) such that the operands can be
   - consisting of more than 1-digit (e.g. ***3.14152***)
   - Variables (e.g. ***rate***), or
   - A mix of alphanumeric tokens, e.g. ***5500 rate ****

   You may assume that a space is used as a delimiter between tokens.

   Hint: Use a dictionary to store each variable and its corresponding value.