Module 9 - Objects and References

Zybooks

General Notes

Objects and References

- Primitive data types *(such as* `int` *)* are copied into the method formal parameters: **pass by value**
- Arrays have the reference to them passed, not the actual array: **pass by reference**

```
// Example 1
Scanner scnr = new Scanner(System.in);

// Example 2

Random gen;

gen = new Radom();
```

- Example 1
  - The `new` operator calls the `Scanner` class constructor, creates the `Scanner` object and returns the reference to `scnr`.
- Example 2
  1. No object has been instantiated and no memory allocated. gen is assigned `null`.
  2. `new` calls the Random class constructor and the object is created. `new` returns the reference to `gen`.

References

A **reference** is a variable type that refers to an object.

- A reference can be seen as storing the memory address of an object.
- Variables of a class data type (an array types) are reference variables

```
TimeHrMin travelTime;
String firstName;
```

- Both of these variables declare a reference to an object of type `TimeHrMin` or `String`, but with an unknown value.
- The `new` operator allocates memory for an object, then returns a reference to the object's location in memory.

The `this` Implicit Parameter

> Using the `this` keyword

An object's member method is called using the syntax:

```
objectReference.method();
```

- The object reference before the method name is known as an **implicit parameter** of the member method because the compiler converts the call syntax `objectReference.methd(...)` into a method call with the object reference implicitly passed as a parameter:

  ```
  method(objectReference, ...)
  ```

- The implicitly-passed object reference is accessible via the keyword `this` inside the member method.
- A class member can be accessed as `this.classMember`.
  - The `.` is the member access operator.
- `this` is useful for showing that a class member is being accessed, especially for times when a field member and parameter have the same identifier.
  - E.g., The field member `sideLength` and the parameter `sideLength`
  - A field member is a variable inside the class.
  - A class member is a field or method that belongs to the class itself rather
    than an instance of the class.

`this` can also be used in a constructor to invoke a different (overloaded) constructor:

```
public class ElapsedTime {
  private int hours;
  private int minutes;

  // Overloaded constructor definition
  public ElapsedTime(int timeHours, int timeMins) {
    hours   = timeHours;
    minutes = timeMins;
  }

  // Default constructor definition
  public ElapsedTime() {
    this(0, 0);
  }

  // Other methods ...
}
```

- Invoking other constructors is useful when a clas` initialization routine is length and avoids rewriting the same code.

Static Fields and Methods

- `static` indicates a variable is allocated in memory only once during a program's execution.
- Static variables reside in the program's static memory region and have a global scope.
  - They can be accessed from anywhere in the program.
- In a class, a `static field` is a field of the class instead of a field of each class object.
  - Static fields are also called **class variables**, and non-static fields are also called **instance variables**.
  - They are independent of any class object and can be accessed without creating a class object.
  - Static fields are declared and initialized in the class definition.
- A static field is accessed using the field name when in the class, and a public static field is accessed outside the class using dot notation:

```
ClassName.fieldName;
```

```
public class Store {
  // Declare and initialize public static field
  public static int nextId = 101;

  // Define private fields
  private String name;
  private String type;
  private int id;

  public Store(String storeName, String storeType) {
    name = storeName;
    type = storeType;
    id = nextId;

    ++nextId;   // Increment each time a Store object is created
  }

  ...
}

public class NewStores {
  public static void main(String[] args) {
    Store store1 = new Store("Macy's", "Department");
    Store store2 = new Store("Albertsons", "Grocery");
    Store store3 = new Store("Ace", "Hardware");

    System.out.println("Store 1's ID: " + store1.getId()); // Store 1's ID: 101
    System.out.println("Store 2's ID: " + store2.getId()); // Store 2's ID: 102
    System.out.println("Store 3's ID: " + store3.getId()); // Store 3's ID: 103
    System.out.println("Next ID: " + Store.nextId);       // Next ID: 104
  }
}
```

Static Member Methods

A **static member method** is a class method that is independent of class objects.

- Static member methods are typically used to access and mutate private static fields from outside the class.
- Since static methods are independent of class objects, the `this` parameter is not passed to a static member method. So, a static member method can only access a class' static fields.

Primitive and Reference Types

Wrapper Classes

> Java Docs:
>
> - Number
> - Character
> - Boolean

Java variables are either:

- **Primitive Type**
  - Variable directly stores the data for that variable ttype, such as:
    - `int`
    - `double`
    - `char`
  - `int numStudents = 20;` declares an `int` that directly stores the data `20`
- **Reference Type**
  - Variable can refer to an instance of a class, known as an object.

Java **wrapper classes** are built-in reference types that augment the primitive types.

The `Integer` data type is a built-in class in Java that augments the `int` primitive type:

```
Integer maxPlayers = 10;
```

- Declares an `Integer` type reference variable named `maxPlayers` that refers to an instance of the `Integer` class, also known as an `Integer` object.

Many of Java's built-in classes, such as Java's Collection library, only work with objects.

- E.g., `ArrayList<Integer> myArray;` or `ArrayList<Double> myArray;`

Commonly Used Wrapper Classes

Reference type	Associated primitive type
`Character`	`char`
`Integer`	`int`
`Double`	`double`
`Boolean`	`boolean`
`Long`	`long`

Memory Allocation for Wrapper Class Objects

A wrapper class object (including `String`) is **immutable**, meaning that the object cannot be changed via methods or variable assignments after object creation.

When the result of an expression is assigned to an `Integer` reference variable, memory for a new `Integer` object with the computed value is allocated, and the reference *(or address)* of this new object is assigned to the reference variable.

- A new memory allocation occurs every time a new value is assigned to an `Integer` variable, and the previous memory location to which the variable referred, remains unmodified

Numeric wrapper classes have `MIN_VALUE` and `MAX_VALUE` fields that state the lowest and highest values that they are capable of accepting.

Comparing Wrapper Class Objects

Expression	Description
`objectVar == objectVar` (also applies to `!=`)	**DO NOT USE**. Compares references to objects, not the value of the objects.
`objectVar == primitiveVar` (also applies to `!=`)	**OK**. Compares value of object to value of primitive variable.
`objectVar == 100` (also applies to `!=`)	**OK**. Compares value of object to literal constant.
`objectVar < objectVar` (also applies to `<=`, `>`, and `>=`)	**OK**. Compares values of objects.
`objectVar < primitiveVar` (also applies to `<=`, `>`, and `>=`)	**OK**. Compares values of object to value of primitive.
`objectVar < 100` (also applies to `<=`, `>`, and `>=`)	**OK**. Compares values of object to literal constant.

- `equals()` and `compareTo()` methods also work for comparing.

`equals()` and `compareTo()` Methods for Wrapper Class Types

- `equals(otherInt)`
  - True if both Integers contain the same value. `otherInt` may be an `Integer` object, int variable, or integer literal.
- `compareTo(otherInt)`
  - Return `0` if the two `Integer` values are equal

- Returns a negative number if the Integer value is less than other Int's value
- Returns a positive number if the Integer value is greater than other Int's value.
- otherInt may be an `Integer` object, `int` variable, or integer literal.

```
Integer num1 = 10;
Integer num2 = 8;
Integer num3 = 10;
int regularInt = 20;

// equals()
num1.equals(num2)             // Evaluates to false
num1.equals(10)               // Evaluates to true
!(num2.equals(regularInt))    // Evaluates to true because 8 != 20

// compareTo()
num1.compareTo(num2)          // Returns value greater than 0, because 10 > 8
num2.compareTo(8)             // Returns 0 because 8 == 8
num1.compareTo(regularInt)    // Returns value less than 0, because 10 < 20
```

Weapper Class Conversions

Autoboxing and Unboxing

- **Autoboxing** is the automatic conversion of primitive types to the corresponding wrapper classes.
- **Unboxing** is the automatic conversion of wrapper class objects to the corresponding primitive types.

Common Autoboxing Scenarios

```
// Assign primitive type to wrapper class variable
Double floorArea = 20.25;   // Autoboxing of 20.25 to a Double
Integer calcResult;

calcResult = 5 / 2;        // Autoboxing of expression result to Integer

int num1 = 23;
Integer num2 = num1;       // Autoboxing of num1 to Integer

// Pass primitive type to a method with wrapper class parameter
public void setRate(Double rate) {
  // ...
}

setRate(50.2);             // Autoboxing of 50.2 to Double

double newRate = 97.2;
setRate(newRate);          // Autoboxing of newRate to Double
```

Common Unboxing Scenarios

```
// Assign wrapper class variable to primitive type
Double num1 = 3.14;
Character letter1 = 'A';

double num2 = num1; // Unboxing of Double to double
char letter2 = letter1; // Unboxing of Character to char

// Pass wrapper class variable to a method with primitive type parameter
public void setInitial(char letter) {
  // ...
}

Character userInitial = 'Z';
setInitial(userInitial);          // Unboxing of userInitial to a char

// Combine wrapper class and primitive types in expression
Double currTemp = 95.2;
double tempDiff = 100.0 - currTemp; // Unboxing of currTemp to double

Integer numItems = 11;

if (numItems % 2 == 0) {          // Unboxing of numItems to int
  // ...
}
```

Converting Wrapper Class Objects to Primitive Types

```
Integer num1 = 14;
Double num2 = 6.7643;
Double num3 = 5.6e12;

// ============intValue()============
// Returns the value of the wrapper class object as a primitive int value,
// type casting if necessary.

num2.intValue()     // Returns 6

// ============doubleValue()============
// Returns the value of the wrapper class object as a primitive double value,
// type casting if necessary.

num1.doubleValue()   // Returns 14.0

// ============longValue()============
// Returns the value of the wrapper class object as a primitive long value, type
// casting if necessary.

num3.longValue()     // Returns 5600000000000
```

- There are also the `charValue()` and `booleanValue()` methods.

Converting *To* and *From* Strings

```
Integer num1 = 10;
Double num2 = 3.14;
String str1 = "32";
int regularInt = 20;

// ====================toString()=======================
// Returns a String containing the decimal representation of the value
// contained by the wrapper class object.

num1.toString() // Returns "10"
num2.toString() // Returns "3.14"

// ============Integer.toString(someInteger)=============
// Returns a String containing the decimal representation of the value of
// someInteger. someInteger may be an Integer object, a int variable, or an
// integer literal. This static method is also available for the other wrapper
// classes (e.g.,  Double.toString(someDouble)).

Integer.toString(num1)            // Returns "10"
Integer.toString(regularInt)      // Returns "20"
Integer.toString(3)               // Returns "3"

// ============Integer.parseInt(someInteger)=============
// Parses someString and returns an int representing the value encoded by
// someString. This static method is also available for the other wrapper
// classes (e.g., Double.parseDouble(someString)), returning the corresponding
// primitive type.

Integer.parseInt(str1)            // Returns int value 32
Integer.parseInt("2001")          // Returns int value 2001

// ============Integer.valueOf(someInteger)==============
// Parses someString and returns a new Integer object with the value encoded by
// someString. This static method is also available for the other wrapper
// classes (e.g., Double.valueOf(someString)), returning a new object of the
// corresponding type.

Integer.valueOf(str1)             // Returns Integer object with value 32
Integer.valueOf("2001")           // Returns Integer object with value 2001
```

```
// ============Integer.toBinaryString(someInteger)=======
// Returns a String containing the binary representation of someInteger.
// someInteger may be an Integer object, a int variable, or an integer literal.
// This static method is also available for the Long classes (e.g.,
// Long.toBinaryString(someLong)).

Integer.toBinaryString(num1)      // Returns "1010"
Integer.toBinaryString(regularInt) // Returns "10100"
Integer.toBinaryString(7)         // Return "111"
```

ArrayList

An **ArrayList** is an ordered list of reference type items that comes with Java.

- **ArrayList** objects can only hold **objects**.
- Each item in an `ArrayList` is known as an **element**.
- Import them with `import java.util.ArrayList;`
- An `ArrayList` can grow and shrink, and has many more methods and features.

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

//Creating space for 3 Integers
valsList.add(31);
valsList.add(41);
valsList.add(59);

System.out.println(valsList.get(1));

// Setting the value of existing elements
valsList.set(1, 119);

System.out.println(valsList.get(1));
```

Common ArrayList Methods

Method	Description	Example

add()	add(element)
	Create space for and add the element at the end of the list.
		// List originally empty
		valsList.add(31); // List now: 31
		valsList.add(41); // List now 31 41

get()	get(index)
	Returns the element at the specified list location known as the **Index**.
	Indices start at 0.
		// List originally: 31 41 59. Assume x is an int.
		x = valsList.get(0); // Assigns 31 to x
		x = valsList.get(1); // Assigns 41
		x = valsList.get(2); // Assigns 59
		x = valsList.get(3); // Error: No such element

set()	set(index, element) Replaces the element at the specified position in this list with the specified element.	// List originally: 31 41 59 valsList.set(1, 119); // List now 31 119 59

size()	size()
	Returns the number of list elements.
		// List originally: 31 41 59. Assume x is an int.
		x = valsList.size(); // Assigns x with 3

Acceessing ArrayList Elements

```
import java.util.ArrayList;
import java.util.Scanner;

public class MostPopularOS {
  public static void main(String[] args) {
    Scanner scnr = new Scanner(System.in);
    ArrayList<String> operatingSystems = new ArrayList<String>();
    int nthOS;     // User input, Nth most popular OS

    // Source: StatCounter.com, 2018
    operatingSystems.add("Windows 10");
    operatingSystems.add("Windows 7");
    operatingSystems.add("Mac OS X");
    operatingSystems.add("Windows 8");
    operatingSystems.add("Windows XP");
    operatingSystems.add("Linux");
    operatingSystems.add("Chrome OS");
    operatingSystems.add("Other");

    System.out.println("Enter N (1-8): ");
    nthOS = scnr.nextInt();

    if ((nthOS >= 1) && (nthOS <= 8)) {
      System.out.print("The " + getNumberSuffix(nthOS) + " most popular OS is ");
      System.out.println(operatingSystems.get(nthOS - 1));
    }
  }

  private static String getNumberSuffix(int number) {
    String[] firstThree = { "st", "nd", "rd" };
    if (number >= 1 && number <= 3) {
      return number + firstThree[number - 1];
    }
    return number + "th";
  }
}
```

Iterating Through ArrayLists

```
import java.util.ArrayList;
import java.util.Scanner;

public class ArrayListAverage {
  public static void main(String[] args) {
    final int NUM_ELEMENTS = 8;
    Scanner scnr = new Scanner(System.in);
    ArrayList<Double> userNums = new ArrayList<Double>(); // User numbers
    Double sumVal;
    Double averageVal;
    int i;

    // Get user numbers and add to userNums
    System.out.println("Enter " + NUM_ELEMENTS + " numbers...");
    for (i = 0; i < NUM_ELEMENTS; ++i) {
      System.out.print("Number " + (i + 1) + ": ");
      userNums.add(scnr.nextDouble());
    }

    // Determine average value
    sumVal = 0.0;
    for (i = 0; i < userNums.size(); ++i) {
      sumVal = sumVal + userNums.get(i); // Calculate sum of all numbers
    }
    averageVal = sumVal / userNums.size(); // Calculate average

    System.out.println("Average: " + averageVal);
  }
}
```

Java Collections Framework

> Collections
>
> from Oracle's Java tutorial.

An ArrayList is one of several **Collections** supported by Java for keeping groups of items.

Other collections include:

- **LinkedList**
- **Set**
- **Queue**
- **Map**
- And many more

A programmer selects the collection whose features best suit the desired task.

For example, an `ArrayList` can efficiently access elements at any valid index but inserts are expensive, whereas a `LinkedList` supports efficient inserts but access requires iterating through elements.

- So a program that will do many accesses and few inserts might use an `ArrayList`.

ArrayList ADT

List Interface and ArrayList ADT

- The **Java Collection Framework** (or **JCF**) defines interfaces and classes for common ADTs known as collections in Java.
- A **Collection** represents a generic group of objects known as elements.
- Java supports several different Collections, including:
  - **List**
  - **Queue**
  - **Map**
  - Etc.
  - Refer to Introduction to Collection Interfaces and Java Collections Framework overview from Oracle's Java documentation for detailed information on each Collection type.
- Each Collection type is an interface that declares the methods accessible to programmers.

The **List** interface is one of the most commonly used Collection types as it represents an ordered group of elements

- I.e., a sequence.

- Both an `ArrayList` and `LinkedList` are ADTs implementing the List interface.
- Although both `ArrayList` and `LinkedList` implement a List, a programmer should select the implementation that is appropriate for the intended task.
    - For example, an ArrayList offers faster positional access
        - E.g., `myArrayList.get(2)`
    - While a `LinkedList` offers faster element insertion and removal.

The `ArrayList` type is an ADT implemented as a class (actually as a generic class that supports different types such as `ArrayList<Integer>` or `ArrayList<String>`)

For the commonly-used public member methods below, assume an `ArrayList` declared as:

```
ArrayList<T> arrayList = new ArrayList<T>();
ArrayList<Integer> teamNums = new ArrayList<Integer>();
```

- where T represents the ArrayList's type, such as:
- Assume `ArrayList teamNums` has existing Integer elements of `5, 9, 23, 11, 14` .

```
// ====================get()=======================
// Returns element at specified index.
T get(int index)

// Example
x = teamNums.get(3);              // Assigns element 3's value 11 to x

// ====================set()=======================
// Replaces element at specified index with newElement. Returns element
// previously at specified index.
T set(int index, T newElement)

// Example
teamNums.set(0, 25);              // Assigns 25 to element 0

x = teamNums.set(3, 88);          // Assigns 88 to element 3.
                                  // Assigns previous element's
                                  // value of 11 to x.

// ====================size()======================
// Returns the number of elements in the ArrayList.
int size()

// Example
if (teamNums.size() > 0) {        // Size is 5 so condition is true
  ...
}

// ====================isEmpty()====================
// Returns true if the ArrayList does not contain any elements. Otherwise,
// returns false.
boolean isEmpty()

// Example
if (teamNums.isEmpty()) {         // Size is 5 so condition is false
  ...
}

// ====================clear()======================
// Removes all elements from the ArrayList.
void clear()
```

```
// Example
teamNums.clear();                // ArrayList now has no elements
System.out.println(teamNums.size()); // Prints 0

// ====================add()=======================
// Adds newElement to the end of the ArrayList. ArrayList's size is increased
// by one.
boolean add(T newElement)

// Adds newElement to the ArrayList at the specified index. Elements at that
// specified index and higher are shifted over to make room. ArrayList's size
// is increased by one.
void add(int index, T newElement)

// Example
// Assume ArrayList is empty
teamNums.add(77);                // ArrayList is: 77
teamNums.add(88);                // ArrayList is: 77, 88
System.out.println(teamNums.size()); // Prints 2
teamNums.add(0, 23);             // ArrayList is: 23, 77, 88
teamNums.add(2, 34);             // ArrayList is: 23, 77, 34, 88
System.out.println(teamNums.size()); // Prints 4

// ====================remove()====================
// Removes the first occurrence of an element which refers to the same object
// as existingElement. Elements from higher positions are shifted back to fill
// gap. ArrayList size is decreased by one. Return true if specified element
// was found and removed.
boolean remove(T existingElement)

// Removes element at specified index. Elements from higher positions are
// shifted back to fill gap. ArrayList size is decreased by one. Returns
// reference to element removed from ArrayList.
T remove(int index)

// Example
// Assume ArrayList is: 23, 77, 34, 88
teamNums.remove(1);              // ArrayList is: 23, 34, 88
System.out.println(teamNums.size()); // Prints 3
```

Example of `ArrayList` Member Methods

```
import java.util.ArrayList;
import java.util.Scanner;

public class PlayerManager {
  // Adds playerNum to end of ArrayList
  public static void addPlayer (ArrayList<Integer> players, int playerNum) {
    players.add(playerNum);
  }

  // Deletes playerNum from ArrayList
  public static void deletePlayer (ArrayList<Integer> players, int playerNum) {
    int i;
    boolean found;

    // Search for playerNum in ArrayList
    found = false;
    i = 0;

    while ( (!found) && (i < players.size()) ) {
      if (players.get(i).equals(playerNum)) {
        players.remove(i); // Remove
        found = true;
      }

      ++i;
    }
  }

  // Prints player numbers currently in ArrayList
  public static void printPlayers(ArrayList<Integer> players) {
    int i;

    for (i = 0; i < players.size(); ++i) {
      System.out.println(" " + players.get(i));
    }
  }

  // Maintains ArrayList of player numbers
  public static void main(String [] args) {
    Scanner scnr = new Scanner(System.in);
```

```
    ArrayList<Integer> players = new ArrayList<Integer>();
    String userInput;
    int playerNum;

    userInput = "-";

    System.out.println("Commands: 'a' add, 'd' delete,");
    System.out.println("'p' print, 'q' quit: ");

    while (!userInput.equals("q")) {
      System.out.print("Command: ");
      userInput = scnr.next();

      if (userInput.equals("a")) {
        System.out.print(" Player number: ");
        playerNum = scnr.nextInt();

        addPlayer(players, playerNum);
      }
      if (userInput.equals("d")) {
        System.out.print(" Player number: ");
        playerNum = scnr.nextInt();

        deletePlayer(players, playerNum);
      }
      else if (userInput.equals("p")) {
        printPlayers(players);
      }
    }

  }
}
```

Further Readings

- Oracle's Java String class specification
- Oracle's Java ArrayList class specification
- Oracle's Java LinkedList class specification
- Introduction to Collection Interfaces from Oracle's Java tutorials
- Introduction to List Implementations from Oracle's Java tutorials

Classes and ArrayLists

Classes ArrayLists are commonly used together. An example Reviews program:

Review.java

```
public class Review {
  private int rating = -1;
  private String comment = "NoComment";

  public void setRatingAndComment(int revRating, String revComment) {
    rating = revRating;
    comment = revComment;
  }
  public int getRating() { return rating; }
  public String getComment() { return comment; }
}
```

Reviews.java

```
import java.util.ArrayList;
import java.util.Scanner;

public class Reviews {
  private ArrayList<Review> reviewList = new ArrayList<Review>();

  public void inputReviews(Scanner scnr) {
    Review currReview;
    int currRating;
    String currComment;

    currRating = scnr.nextInt();
    while (currRating >= 0) {
      currReview = new Review();
      currComment = scnr.nextLine(); // Gets rest of line
      currReview.setRatingAndComment(currRating, currComment);
      reviewList.add(currReview);
      currRating = scnr.nextInt();
    }
  }

  public void printCommentsForRating(int currRating) {
    Review currReview;
    int i;

    for (i = 0; i < reviewList.size(); ++i) {
      currReview = reviewList.get(i);
      if (currRating == currReview.getRating()) {
        System.out.println(currReview.getComment());
      }
    }
  }

  public int getAverageRating() {
    int ratingsSum;
    int i;

    ratingsSum = 0;
    for (i = 0; i < reviewList.size(); ++i) {
      ratingsSum += reviewList.get(i).getRating();
    }
```

```
    return (ratingsSum / reviewList.size());
  }
}
```

ReviewSystem.java

```
import java.util.ArrayList;
import java.util.Scanner;

public class ReviewSystem {

  public static void main(String [] args) {
    Scanner scnr = new Scanner(System.in);
    Reviews allReviews = new Reviews();
    String currName;
    int currRating;

    System.out.println("Type rating + comments. To end: -1");
    allReviews.inputReviews(scnr);

    System.out.println("\nAverage rating: ");
    System.out.println(allReviews.getAverageRating());

    // Output all comments for given rating
    System.out.println("\nType rating. To end: -1");
    currRating = scnr.nextInt();
    while (currRating != -1) {
      allReviews.printCommentsForRating(currRating);
      currRating = scnr.nextInt();
    }
  }
}
```

Ouput

```
Type rating + comments. To end: -1
5 Great place!
5 Loved the food.
2 Pretty bad service.
4 New owners are nice.
2 Yuk!!!
4 What a gem.
-1

Type rating. To end: -1
5
 Great place!
 Loved the food.
1
4
 New owners are nice.
 What a gem.
-1
```

Using Reviews in the Restaurant Class

It's common to use classes within classes. The following uses the previously created Reviews classes:

Restaurant.java

```
import java.util.Scanner;

// Review and Reviews classes omitted from the figure

public class Restaurant {
   private String name;
   private Reviews reviews = new Reviews();

   public void setName(String restaurantName) {
      name = restaurantName;
   }

   public void readAllReviews(Scanner scnr) {
      System.out.println("Type ratings + comments. To end: -1");
      reviews.inputReviews(scnr);
   }

   public void printCommentsByRating() {
      int i;

      System.out.println("Comments for each rating level: ");
      for (i = 1; i <= 5; ++i) {
         System.out.println(i + ":");
         reviews.printCommentsForRating(i);
      }
   }
}
```

RestaurantReviews.java

```
import java.util.ArrayList;
import java.util.Scanner;

public class RestaurantReviews {

  public static void main (String [] args) {
    Scanner scnr = new Scanner(System.in);
    Restaurant ourPlace = new Restaurant();
    String currName;

    System.out.println("Type restaurant name: ");
    currName = scnr.nextLine();
    ourPlace.setName(currName);
    System.out.println();

    ourPlace.readAllReviews(scnr);
    System.out.println();

    ourPlace.printCommentsByRating();
  }
}
```

Output

```
Type restaurant name:
Maria's Healthy Food

Type ratings + comments. To end: -1
5 Great place!
5 Loved the food.
2 Pretty bad service.
4 New owners are nice.
2 Yuk!!!
4 What a gem.
-1

Comments for each rating level:
1:
2:
 Pretty bad service.
 Yuk!!!
3:
4:
 New owners are nice.
 What a gem.
5:
 Great place!
 Loved the food.
```

Java Documentation for Classes

- The **Javadoc** tool parses source code along with specially formatted comments to generate documentation.
- The documentation generated by Javadoc is known as an **API** for classes and class members.
    - **API** is short for **Application Programming Interface**.
- The specially formatted comments for Javadoc are called **Doc comments**, which are multi-line comments consisting of all text enclosed between the `/**` and `*/` characters.
- Importantly, Doc comments are distinguished by the opening characters `/**`, which include two asterisks.

- A doc comment associated with a class is written immediately before the class definition.
- `@author` and `@version` are two block tags used in the tag section of a Doc comment to specify the class` author and version number.

Examples of Javadoc Comments To Document Classes

ElapsedTime.java

```java
/**
 * A class representing an elapsed time measurement
 * in hours and minutes.
 * @author Mary Adams
 * @version 1.0
 */
public class ElapsedTime {
  /**
   * The hours portion of the time
   */
  private int hours;

  /**
   * The minutes portion of the time
   */
  private int minutes;

  /**
   * Constructor initializing hours to timeHours and
   * minutes to timeMins.
   * @param timeHours hours portion of time
   * @param timeMins minutes portion of time
   */
  public ElapsedTime(int timeHours, int timeMins) {
    hours   = timeHours;
    minutes = timeMins;
  }

  /**
   * Default constructor initializing all fields to 0.
   */
  public ElapsedTime() {
    hours = 0;
    minutes = 0;
  }

  /**
   * Prints the time represented by an ElapsedTime
```

```
     * object in hours and minutes.
     */
    public void printTime() {
        System.out.print(hours + " hour(s) " + minutes + " minute(s)");
    }

    /**
     * Sets the time to timeHours:timeMins.
     * @param timeHours hours portion of time
     * @param timeMins minutes portion of time
    */
    public void setTime(int timeHours, int timeMins) {
        hours = timeHours;
        minutes = timeMins;
    }

    /**
     * Returns the total time in minutes.
     * @return an int value representing the elapsed time in minutes.
     */
    public int getTimeMinutes() {
        return ((hours * 60) + minutes);
    }
}
```

TimeDifference.java

```java
import java.util.Scanner;

/**
 * This program calculates the difference between two
 * user-entered times. This class contains the
 * program's main() method and is not meant to be instantiated.
 * @author Mary Adams
 * @version 1.0
 */
public class TimeDifference {
  /**
   * Asks for two times, creating an ElapsedTime object for each,
   * and uses ElapsedTime's getTimeMinutes() method to properly
   * calculate the difference between both times.
   * @param args command-line arguments
   * @see ElapsedTime#getTimeMinutes()
   */
  public static void main(String[] args) {
    Scanner scnr = new Scanner(System.in);
    int timeDiff;     // Stores time difference
    int userHours;
    int userMins;
    ElapsedTime startTime = new ElapsedTime(); // Starting time
    ElapsedTime endTime = new ElapsedTime(); // Ending time

    // Read starting time in hours and minutes
    System.out.print("Enter starting time (hrs mins): ");
    userHours = scnr.nextInt();
    userMins = scnr.nextInt();
    startTime.setTime(userHours, userMins);

    // Read ending time in hours and minutes
    System.out.print("Enter ending time (hrs mins): ");
    userHours = scnr.nextInt();
    userMins = scnr.nextInt();
    endTime.setTime(userHours, userMins);

    // Calculate time difference by converting both times to minutes
    timeDiff = endTime.getTimeMinutes() - startTime.getTimeMinutes();

    System.out.println("Time difference is " + timeDiff + " minutes");
```

```
  }
}
```

Doc comments may be used to describe a class's fields as well. Unlike classes or methods, a field's Doc comment is not typically associated with specific block tags. However, generic block tags, such as `@see` and others described by the Javadoc specification, may be used to provide more information.

For example, the `main()` method's Doc comment uses the `@see` block tag to refer to ElapsedTime's `getTimeMinutes()` method, as in `@see ElapsedTime#getTimeMinutes()`.

- Note that when referring to a method, the `@see` block tag requires the programmer to precede the method name with the class name followed by the `#` character.

Common Block Tags Used in Javadoc Comments

Block tag	Compatibility	Description
`@author`	classes	Used to specify an author.
`@version`	classes	Used to specify a version number.
`@param`	methods, constructors	Used to describe a parameter.
`@return`	methods	Used to describe the value or object returned by the method.
`@see`	all	Used to refer reader to relevant websites or class members.

Additional Notes on API Documentation

- Private class members are not included by default in the API documentation generated by the Javadoc tool.
- API documentation is meant to provide a summary of all functionality available to external applications interfacing with the described classes. Thus, private

class members, which cannot be accessed by other classes, are not typically included in the documentation.
- The resulting API documentation for the above classes need only include information that enables their use by other programmers.
    - If a programmer needs to document a class's complete structure, the Javadoc tool can be executed with the `-private` flag, as in `javadoc -private -d destination class1.java class2.java`, to enable the documentation of private class members.

Further Reading

- The Javadoc specification from Oracle's Java documentation
- How to write Javadoc comments from Oracle's Java documentation
- How to run the Javadoc tool from Oracle's Java documentation

Parameters of Reference Types

Methods With Reference Variables as Parameters

A **reference variable** is a variable that points to, or refers to, an object or array. Internally, a reference variable stores a reference, or the memory location, of the object to which it refers.

- A programmer can only access the data or functionality provided by objects through the use of reference variables.

Methods With Wrapper Class Parameters

Instances of wrapper classes, such as `Integer` and `Double`, and the `String` class are defined as **immutable**, meaning that a programmer cannot modify the object's contents after initialization; new objects must be created instead.

Methods With User-Declared Reference Variables As Parameters

A programmer-defined object is passed to a method by passing a reference (i.e., memory location) to the object.

Using Packages

Built-in Java Packages

- A **package** is a grouping of related types, classes, interfaces, and subpackage.
- The types, classes, and interfaces in a package are called **package members**.

Common Java Packages

Package	Sample package members	Description
`java.lang`	`String`, `Integer`, `Double`, `Math`	Contains fundamental Java classes. Automatically imported by Java.
java.util	Collection, ArrayList, LinkedList, Scanner	Contains the Java collections framework classes and miscellaneous utility classes.
`java.io`	`File`, `InputStream`, `OutputStream`	Contains classes for system input and output.
javax.swing	JFrame, JTextField, JButton	Contains classes for building graphical user interfaces.

Using Package Members in a Program

A programmer can use a package member using one of the following methods:

- **Using a package member's fully qualified name**
  - A class' **fully qualified name** is the concatenation of the package name with the class name using a period. Ex: `java.util.Scanner` is the fully qualified name for the `Scanner` class in the `java.util` package.
- **Using an import statement to import the package member**
  - An **import statement** imports a package member into a file to enable use of the package member directly, without having to use the package member's fullyqualified name. Ex: `import java.util.Scanner;` imports the `Scanner` class into a file and allows a programmer to use `Scanner` instead of `java.util.Scanner`.

- **Using an import statement to import every member in a package**
  - A programmer import all members of a package by using the **wildcard** character `*` instead of a package member name. Ex: `import java.util.*;` imports all classes in the java.util package and allows a programmer use package members such as `Scanner` and `ArrayList` directly.

Further Reading

- Creating and using packages
  from Oracle's Java documentation
- Java 12 API Specification
  from Oracle's Java documentation