First Name: _____ **Last Name**: _____

McGill ID: _____ **Section**: _____

Faculty of Science
COMP-202B - Introduction to Computing I (Winter 2010) - All Sections
Final Examination

Friday, April 30, 2010
9:00–12:00

Examiners: Milena Scaccia [Section 1]
Mathieu Petitpas [Sections 2 and 3]

Instructions:

- **DO NOT TURN THIS PAGE UNTIL INSTRUCTED**
- This is a **closed book** final examination; notes, slides, textbooks, and other forms of documentation are **not** allowed. However, translation dictionaries (for human languages only) are permitted.
- **Non-programmable calculators** are allowed (though you should not need one).
- **Computers, PDAs, cell phones, and other electronic devices** are **not** allowed.
- Answer **all** questions **on this examination paper** and return it. If you need additional space, use pages 22-23 or the booklets supplied upon request and clearly indicate where each question is continued. **In order to receive full marks for a question, you must show all work** unless otherwise stated.
- This final examination has **26** pages including this cover page, and is printed on both sides of the paper. Pages 24-26 contain information about useful classes and methods.

1	2	3	4	Subtotal
/4	/6	/5	/5	/20

5	6	Subtotal
/15	/10	/25

7	8	9	Subtotal
/25	/20	/10	/55

Total
/100

Section 1 - Short Questions

[4] 1. In one or two sentences, explain each of the following concepts. **BE BRIEF**; overly long answers will be grounds for mark deductions.

(a) Off-by-one error

(b) Local variable

(c) Method overloading

(d) Garbage collection

[6] 2. In one or two sentences, explain the differences between the concepts in each of the following pairs. **BE BRIEF**; overly long answers will be grounds for mark deductions.

(a) An `if` statement and a `while` statement

(b) Classes and objects

(c) Formal parameters and actual parameters

[5] 3. Consider the following method, which returns the number of times the parameter `String word` occurs in the parameter array `wordArray`:

```
public static int countOccurrences(String[] wordArray, String word) {
  int result;

  result = 0;
  for (int i = 0; i < wordArray.length; i++) {
    if (wordArray[i] == word) {
      result = result + 1;
    }
  }
  return result;
}
```

Unfortunately, the above method contains a bug. This bug causes the method to return `0` instead of the expected value of `2` when it is called in the following code fragment:

```
String[] animalism = {
  new String("four"),
  new String("legs"),
  new String("good"),
  new String("two"),
  new String("legs"),
  new String("bad")
};
int count = countOccurrences(animalism, "legs");
System.out.println("Number of occurrences: " + count);
```

Describe what the problem is, and suggest a simple way to fix it. **BE BRIEF**; overly long answers will be grounds for mark deductions.

[5] 4. Consider the following program.

```
public class Fraction {
  private int numerator;
  private int denominator;

  public Fraction(int n, int d) {
    numerator = n;
    denominator = d;
  }

  public Fraction add(Fraction f) {
    return new Fraction(numerator * f.denominator +
      f.numerator * denominator, denominator * f.denominator);
  }

  public Fraction add(int x) {
    return new Fraction(numerator + x * denominator, denominator);
  }

  public Fraction add(int n, int d) {
    return new Fraction(numerator * d + n * denominator,
      denominator * d);
  }

  public String toString() {
    return numerator + "/" + denominator;
  }

  public static void main(String[] args) {
    Fraction f1 = new Fraction(1, 2);
    Fraction f2 = new Fraction(2, 3);
    Fraction f3 = new Fraction(3, 4);

    Fraction[] sums = new Fraction[4];
    sums[0] = f1.add(f3);
    sums[1] = f2.add(f3);
    sums[2] = f1.add(1);
    sums[3] = f2.add(3, 3);

    for (int i = 0; i <= sums.length; i++) {
      System.out.println(sums[i].toString());
    }
  }
}
```

The above program compiles without error; what does it display when it is executed?

WRITE THE PROGRAM'S OUTPUT IN THE SPACE BELOW:

Total marks for Section 1: **20**

Section 2 - Long Questions

[15] 5. Consider the following class:

```
import java.util.Scanner;

public class Money {
  private int dollars;
  private int cents;

  public Money(int d, int c) {
    dollars = d;
    cents = c;
    /* Point 1 */
  }

  public String toString() {
    String result;

    if (cents > 100) {
      int totalCents = dollars * 100 + cents;
      dollars = totalCents / 100;
      cents = totalCents % 100;
      /* Point 2 */
    }
    /* Point 3 */
    result = "$" + dollars + ", " + cents + "c";
    return result;
  }

  public static void main(String[] yourGrandfather) {
    Scanner keyboard = new Scanner(System.in);
    Money amount;
    int c;

    /* Point 4 */
    System.out.print("Enter an amount in cents: ");
    c = keyboard.nextInt();
    amount = new Money(c / 100, c % 100);
    System.out.println("You have: " + amount.toString());
  }
}
```

The above program compiles and runs without error.

Which variables are in scope at each of the points marked by comments in the above class? In other words, if each of these comments were replaced by an actual Java statement, which variables could be used in this statement without a prefix (that is, without having to write the name of a class or other variable in front of the variable name) and without the compiler reporting an unknown variable error?

List each variable that is in scope at each of the points marked by comments in the above class, and indicate whether it is an instance variable, a formal parameter, or a local variable. Note that you will be penalized for every variable that is out of scope at a given point but that you list as being in scope at that point.

(a) Which variables are in scope at the point given by the comment `/* Point 1 */`? For each variable you list, indicate whether it is an instance variable, a formal parameter, or a local variable.

(b) Which variables are in scope at the point given by the comment `/* Point 2 */`? For each variable you list, indicate whether it is an instance variable, a formal parameter, or a local variable.

(c) Which variables are in scope at the point given by the comment `/* Point 3 */`? For each variable you list, indicate whether it is an instance variable, a formal parameter, or a local variable.

(d) Which variables are in scope at the point given by the comment `/* Point 4 */`? For each variable you list, indicate whether it is an instance variable, a formal parameter, or a local variable.

[10] 6. The two classes below compile without error. What is displayed when the `main()` method of class `NightmareMix` is executed? Clearly indicate the program output in the appropriate space on page 13.

You may use the other blank space as a scratch pad to track the state of memory as the program executes, but the contents of this other space will not be graded.

The definition of the `Pair` class:

```
public class Pair {
  private int left;
  private int right;

  public Pair(int l, int r) {
    left = l;
    right = r;
  }

  public void mix(Pair p) {
    left = p.left;
  }

  public String toString() {
    return "[L: " + left + ", R: " + right + "]";
  }
}
```

The definition of the `NightmareMix` class:

```
public class NightmareMix {
  public static void mixInts(int i1, int i2) {
    i1 = i2;
  }

  public static void mixArrayWithInt(int[] a, int i, int v) {
    a[i] = v;
  }

  public static void mixArraysOfInt(int[] a1, int[] a2) {
    a1 = a2;
  }

  public static void mixMatrixWithInt(int[][] m, int i1, int i2, int v) {
    m[i1][i2] = v;
  }

  public static void mixMatrixWithArray(int[][] m, int i, int[] a) {
    m[i] = a;
  }

  public static void mixPairAddresses(Pair p1, Pair p2) {
    p1 = p2;
  }
```

```
  public static void mixArrayWithPairState(Pair[] a, int i, Pair p) {
    a[i].mix(p);
  }

  public static void mixArrayWithPairAddress(Pair[] a, int i, Pair p) {
    a[i] = p;
  }

  public static void mixArraysOfPairs(Pair[] a1, Pair[] a2) {
    a1 = a2;
  }

  public static void main(String[] args) {
    int myV1 = 1;
    int myV2 = 2;
    int[] myA1 = {3, 4};
    int[] myA2 = {5, 6};
    int[][] myM = {{7, 8}, {9, 10}};
    Pair myP1 = new Pair(11, 12);
    Pair myP2 = new Pair(13, 14);
    Pair[] myPA1 = {new Pair(15, 16), new Pair(17, 18)};
    Pair[] myPA2 = {new Pair(19, 20), new Pair(21, 22)};

    mixInts(myV1, myV2);

    mixArrayWithInt(myA1, 0, myV2);
    mixArraysOfInt(myA1, myA2);

    mixMatrixWithInt(myM, 0, 0, myV2);
    mixArraysOfInt(myM[0], myM[1]);
    mixMatrixWithArray(myM, 1, myA2);
    mixArrayWithInt(myA2, 0, myV2);

    mixPairAddresses(myP1, myP2);
    mixPairAddresses(myPA1[0], myPA1[1]);

    mixArrayWithPairState(myPA2, 0, myP1);
    mixArrayWithPairAddress(myPA2, 1, myP2);
    mixArraysOfPairs(myPA1, myPA2);

    System.out.println("1: " + myV1);
    System.out.println("2: {" + myA1[0] + ", " + myA1[1] + "}");
    System.out.println("3: {" + myA2[0] + ", " + myA2[1] + "}");
    System.out.println("4: {{" + myM[0][0] + ", " + myM[0][1] + "}, {" +
      myM[1][0] + ", " + myM[1][1] + "}}");
    System.out.println("5: {" + myPA1[0] + ", " + myPA1[1] + "}");
    System.out.println("6: {" + myPA2[0] + ", " + myPA2[1] + "}");
  }
}
```

USE THE SPACE BELOW AS A SCRATCH PAD TO TRACK THE STATE OF THE VARIABLES IN MEMORY (THE CONTENTS OF THIS SPACE **WILL NOT** BE GRADED):

CLEARLY INDICATE THE PROGRAM'S OUTPUT IN THE SPACE BELOW (THE CONTENTS OF THIS SPACE **WILL** BE GRADED):

Section 3 - Programming Questions

7. Role-playing video games (RPGs) form a loosely defined genre of video games in which the player controls a small number of game characters. These characters form a group of adventurers whose purpose is to fulfill one or many quests.

   A common feature of RPGs is that the group of adventurers can collect various items and keep them for future use; such items are added to the player's inventory. In this question, you will write classes which could be used to manage a player's inventory in an RPG.

[8] **Part 1**:

Write a class called `Item`. An `Item` object represents an entire category of items carried by a group of adventurers, and each `Item` object has the following attributes:

- The description of the items in this category (a `String`)
- The quantity of items in this category that the group of adventurers is currently carrying (an `int`).

For example, suppose the group of adventurers is carrying 50 Heal Potions and 10 Life Potions. The 50 Heal Potions would be represented by a single `Item` object; the value of the description attribute of this `Item` object would be the `String` `"Heal Potion"`, and the value of the quantity attribute of this `Item` object would be `50`. The 10 Life Potions would also be represented by a single `Item` object, which is different from the `Item` object representing the Heal Potions. The value of the description attribute of this second `Item` object would be `"Life Potion"`, and the value of the quantity attribute of this second Item object would be `10`.

The `Item` class **MUST** provide the following **INSTANCE** methods; note that in the method descriptions below, the phrase "this `Item`" means "the `Item` on which the method in question is called."

- A constructor, which takes as its only parameter a `String` representing the description attribute of the newly-created `Item`. This constructor sets the quantity attribute of the newly-created `Item` to `1`.
- A method called `getDescription()`, which takes no parameters and returns a `String` representing the value of the description attribute of this `Item`.
- A method called `getQuantity()`, which takes no parameters and returns an `int` representing the value of the quantity attribute of this `Item`.
- A method called `setQuantity()`, which takes a value of type `int` as its only parameter and returns nothing. This method changes the value of the quantity attribute of this `Item` so that its new value is equal to the `int` value this method accepts as parameter; this happens regardless of what the value of the `int` parameter is, even if it represents a nonsensical value.

Note that you **MUST** respect proper encapsulation practices; that is, the attributes and methods of your class **MUST** be declared using the proper visibility modifiers.

WRITE YOUR `Item` CLASS IN THE SPACE BELOW:

[17] **Part 2**:

Write a class called `Inventory`, which represents the inventory of a group of adventurers. An `Inventory` object has only one attribute: a collection of items, which specifies all the types of items that a group of adventurers is currently carrying along with the number of items of each type that the group is carrying. `Inventory` objects **MUST** keep track of the items being carried by adventurers using a **PLAIN ARRAY** of `Items`; you **MUST NOT** use `ArrayLists` (or any other class which is part of the the Java Collection Framework such as `LinkedList`) within the `Inventory` class under **ANY** circumstances. Each position in the array contains either:

- The address of an `Item` object in memory; an `Item` is considered to be in the `Inventory` if and only if its address in memory is stored in the array
- `null`

Each `Item` object in the inventory **MUST** appear exactly once in this array; in other words, there **MUST NOT** be any duplicate elements in the array under **ANY** circumstances, whether these duplicates are deep copies or aliases.

The `Inventory` class **MUST** provide the following **INSTANCE** methods; note that an actual implementation used in an RPG would provide additional methods, in particular methods to add an item to the `Inventory`. Also note that in the method descriptions below, the phrase "this `Inventory`" means "the `Inventory` on which the method in question is called."

- A constructor, which takes a value of type `int` as its only parameter, and initializes the newly-created `Inventory` so that the number of item types it can store is equal to the value of the parameter. The number of item types which can be stored in the `Inventory` is fixed during instantiation, and can never change during the lifetime of the `Inventory` object.
- A method called `getCapacity()`, which takes no parameters and returns a value of type `int` representing the capacity of this `Inventory` (that is, the number of different item types which can be stored in this `Inventory`).
- A method called `getItem()`, which takes a value of type `int` as its only parameter, and returns the `Item` stored at the corresponding position in the array holding the `Items` stored in this `Inventory`; this method returns `null` if the element at the position specified by the parameter value is `null`, or if the parameter represents an invalid index in this array.
- A method called `consumeOne()`, which takes as its only parameter a `String` representing the description of an item, and returns a value of type `boolean`. The method searches the `Inventory` for an `Item` whose description is equal to the parameter `String`, with the comparison being performed in a case-**INSENSITIVE** manner. If no such `Item` exists in the inventory, the method returns `false`, and the state of the `Inventory` does not change. On the other hand, if such an `Item` exists in the inventory, the method returns `true` and modifies the state of the `Inventory` as follows:
  - If the quantity attribute of the `Item` is greater than or equal to `2`, then its value is reduced by `1`.
  - If the quantity attribute of the `Item` is equal to `1`, then the `Item` is removed from the `Inventory`. Note that removing an `Item` can be done by simply assigning `null` to the array position that contains it.

  The state of the array before and after calling this method can consist of addresses of `Item` objects interspersed with `null` elements. For example, in an `Inventory` whose capacity is 20, it is possible that the array contains the addresses of `Item` objects at positions `3` and `17`,

while the other array position contains `null`. This implies that the method does **NOT** have to "compact" the array so that the addresses of all the actual `Item` objects appear at the beginning of the array while all the `null` elements appear at the end.
You **MAY** assume that the `String` parameter is not `null`; in other words, your method does not have to handle cases in which the `String` parameter is indeed `null`.

Additional notes:

- You **MUST** respect proper encapsulation practices; that is, the attributes and methods of your class **MUST** be declared using the proper visibility modifiers.
- You **MUST** call the methods you were asked to write in the previous part to retrieve the description and quantity attributes of an `Item` object, or to change the quantity attribute of an `Item` object. You **MAY** assume that the `Item` class has been implemented correctly, even if you did not successfully complete the previous part.

WRITE YOUR `Inventory` CLASS IN THE SPACE BELOW:

[20] 8. Write a class called `WordSet` representing a set of `String` objects which can contain an arbitrary number of `String`s, limited only by the memory available to the Java Virtual Machine. A set is a collection of elements; however, unlike a list, a set cannot contain duplicate elements. Therefore, as its name implies, a `WordSet` object cannot contain duplicate `String`s; that is, if one attempts to add a `String` to a `WordSet`, and the `WordSet` already contains an element which is equal to this `String`, the attempt will fail and the state of the `WordSet` will not change.

All `String` comparisons performed in the methods defined in the `WordSet` class are done in a case-**SENSITIVE** manner. Note that in the method descriptions below, the phrase "this `WordSet`" means "the `WordSet` on which the method in question is called."

Your `WordSet` class **MUST** provide the following **INSTANCE** methods:

- A constructor, which takes no parameters and initializes the new `WordSet` object so that it contains no elements.
- A method called `add()`, which takes as its only parameter a `String` and returns a value of type `boolean`. If this `WordSet` already contains a `String` which is equal to the parameter `String`, or the parameter `String` is `null`, then the state of this `WordSet` does not change, and the method returns `false`; otherwise, the parameter `String` is added to this `WordSet`, and the method returns `true`.
- A method called `remove()`, which takes as its only parameter a `String` and returns a value of type `boolean`. If this `WordSet` contains a `String` which is equal to the parameter `String`, this `String` is removed from the `WordSet` and the method returns `true`. On the other hand, if the `String` parameter is `null`, or this `WordSet` does not contain a `String` which is equal to the parameter `String`, then the state of this `WordSet` does not change, and the method returns `false`.
- A method called `getSize()`, which takes no parameters and returns the number of `String`s currently stored in this `WordSet`. This method **MUST NOT** change the state of this `WordSet`.
- A method called `isMember()`, which takes as its only parameter a `String` and returns a value of type `boolean` which is `true` if this `WordSet` contains the parameter `String`, `false` otherwise. If the parameter `String` is `null`, this method also returns `false`. This method **MUST NOT** change the state of this `WordSet`.
- A method called `computeDifference()`, which takes as its only parameter a `WordSet` and returns a new `WordSet` representing the difference between this `WordSet` and the parameter `WordSet`; in other words, the `WordSet` returned by this method contains all the `String`s that are in this `WordSet`, but not in the parameter `WordSet`. If the parameter `WordSet` contains all the elements of this `WordSet` or this `WordSet` contains no elements, then this method returns a new `WordSet` containing no elements. Finally, if the parameter `WordSet` is `null` or contains no `String`s, then this method returns a new `WordSet` which contains all the elements of this `WordSet`. This method **MUST NOT** change the state of this `WordSet` or the parameter `WordSet`.
- A method called `toString()`, which takes no parameters and returns a `String` which is the textual representation of this `WordSet`. This textual representation consists of the concatenation of the following elements:
  - The `String "["`
  - The text representation of each element in the `WordSet`; each pair of adjacent elements is separated by the `String ", "`, but this `String` **MUST NOT** appear before the first element nor after the last element
  - The `String "]"`

The order in which the elements of this `WordSet` appear in the textual representation generated by the `toString()` method does not matter. If the `WordSet` contains no `String`s, then the textual representation returned by this method is `"[]"`.

Note that you **MUST** respect proper encapsulation practices; that is, the attributes and methods of your class **MUST** be declared using the proper visibility modifiers.

HINT: Use an `ArrayList` (of `String`) to keep track of the `String` stored in a `WordSet` object. Is there a method defined in the `ArrayList` class which determines whether an element is already stored in an `ArrayList`, and which will do a case-sensitive comparison when invoked on an `ArrayList` of `String`s? See the pages 24- 26 of this examination for information about potentially useful methods defined by the `ArrayList` class.

WRITE YOUR `WordSet` CLASS IN THE SPACE BELOW:

[10] 9. Write a method with the following header:

```
public static void replace(String oldFile, String newFile,
  char oldChar, char newChar) throws IOException
```

This method opens the file whose name is given by the `String` parameter `oldFile` (the input file) for reading, and opens the file whose name is given by the `String` parameter `newFile` (the output file) for writing. It then reads every line in the input file. For each line, it creates a new line identical to the original line, except for the fact that all occurrences of `char oldChar` in the original `String` have been replaced by occurrences of `char newChar` in the new line. When the new line has been properly created, it writes it to the output file. Once all the lines in the input file have been read and a corresponding line has been written to the output file, the method closes both files.

Additional notes:

- You **MAY** assume that none of the parameters this method accepts are `null`; in other words, your method does not have to handle the case where one or more of the parameters are `null`.
- You **MAY** assume that all the IO-related classes have already been imported properly.
- Finally, because the `replace()` method is specified to throw an exception object of type `IOException`, you **MAY** assume that no I/O-related problems will occur while reading from the input file or writing to the output file. In other words, your method **DOES NOT** have to check that no such problems occur, and **DOES NOT** have to manage IOExceptions using `try-catch` blocks.

WRITE YOUR `replace()` METHOD IN THE SPACE BELOW:

YOUR `replace()` METHOD CONTINUED:

Total marks for Section 3: **55**

Total marks: **100**

USE THIS PAGE IF YOU NEED ADDITIONAL SPACE. CLEARLY INDICATE WHICH QUESTION(S) YOU ARE ANSWERING HERE.

USE THIS PAGE IF YOU NEED MORE ADDITIONAL SPACE. CLEARLY INDICATE WHICH QUESTION(S) YOU ARE ANSWERING HERE.

SUMMARY OF JAVA STANDARD LIBRARY METHODS FOR SELECTED CLASSES

- `String` (package `java.lang`) Methods:
  - `public String(char[] value)`: Allocates a new `String` so that it represents the sequence of characters currently contained in the character array `value`.
  - `public int length()`: Returns the length of this `String`.
  - `public char charAt(int index)`: Returns the `char` value at the specified `index`.
  - `public char[] toCharArray()`: Converts this `String` to a new character array.
  - `public boolean equals(Object anObject)`: Compares this `String` to `anObject`.
  - `public boolean equalsIgnoreCase(String anotherString)`: Compares, ignoring case considerations, this `String` to `anotherString`.
  - `public int compareTo(String anotherString)`: Compares this `String` to `anotherString` lexicographically; returns a negative value if this `String` occurs before `anotherString`, a positive value if this `String` occurs after `anotherString`, and `0` if both `String`s are equal.
  - `public int compareToIgnoreCase(String anotherString)`: Compares, ignoring case considerations, this `String` to `anotherString` lexicographically; returns a negative value if this `String` occurs before `anotherString`, a positive value if this `String` occurs after `anotherString`, and `0` if both `String`s are equal.
  - `public int indexOf(int ch)`: Returns the index within this `String` of the first occurrence of character `ch`, `-1` if it does not occur.
  - `public int indexOf(int ch, int fromIndex)`: Returns the index within this `String` of the first occurrence of character `ch`, starting the search at position `fromIndex`; returns `-1` if `ch` does not occur in this `String`.
  - `public int indexOf(String str)`: Returns the index within this `String` of the first occurrence of substring `str`, `-1` if it does not occur.
  - `public int indexOf(String str, int fromIndex)`: Returns the index within this `String` of the first occurrence of substring `str`, starting at position `fromIndex`; returns `-1` if `str` does not occur in this `String`.
  - `public String substring(int beginIndex)`: Returns a new `String` which is a substring of this `String`, composed of the characters starting at position `beginIndex` (inclusive).
  - `public String substring(int beginIndex, int endIndex)`: Returns a new `String` that is a substring of this `String`, composed of the characters starting at position `beginIndex` (inclusive), and ending at position `endIndex` (exclusive).
  - `public String replace(char oldChar, char newChar)`: Returns a new `String` resulting from replacing all occurrences of `oldChar` in this `String` with `newChar`.
  - `public String toLowerCase()`: Returns a new `String` consisting of all the characters in this `String` converted to lower case.
  - `public String toUpperCase()`: Returns a new `String` consisting of all the characters in this `String` converted to upper case.
  - `public String trim()`: Returns a copy of this `String`, with leading and trailing whitespace omitted.
- `Scanner` (package `java.util`) Methods:
  - `public Scanner(File source) throws java.io.FileNotFoundException`: Constructs a new `Scanner` that produces values scanned from the specified file.
  - `public Scanner(InputStream source)`: Constructs a new `Scanner` that produces values scanned from the specified input stream.
  - `public Scanner(String source)`: Constructs a new `Scanner` that produces values scanned from the specified `String`.
  - `public void close()`: Closes this `Scanner`.
  - `public boolean hasNext()`: Returns `true` if this `Scanner` has another token in its input.
  - `public boolean hasNextDouble()`: Returns `true` if the next token in this `Scanner`'s input can be interpreted as a `double` value using the `nextDouble()` method.
  - `public boolean hasNextInt()`: Returns `true` if the next token in this `Scanner`'s input can be interpreted as an `int` value using the `nextInt()` method.
  - `public boolean hasNextLine()`: Returns `true` if there is another line in the input of this `Scanner`

  - `public boolean hasNextLong()`: Returns `true` if the next token in this `Scanner`'s input can be interpreted as a `long` value using the `nextLong()` method.
  - `public String next()`: Finds and returns the next complete token from this `Scanner`.
  - `public double nextDouble()`: Scans the next token of the input as a `double`.
  - `public int nextInt()`: Scans the next token of the input as an `int`.
  - `public String nextLine()`: Advances this `Scanner` past the current line and returns the input read.
  - `public int nextLong()`: Scans the next token of the input as an `long`.

- `PrintStream` (package `java.io`) Methods:

  - `public PrintStream(File file) throws java.io.FileNotFoundException`: Creates a new `PrintStream` which writes to the specified `File`.
  - `public PrintStream(String fileName) throws java.io.FileNotFoundException`: Initializes a new `PrintStream` which writes to the file with the specified `fileName`.
  - `public void close()`: Closes the stream.
  - `public void print(boolean b)`: Prints `boolean` value `b`.
  - `public void print(char c)`: Prints `char` value `c`.
  - `public void print(char[] s)`: Prints the array of `char` `s`.
  - `public void print(double d)`: Prints `double` value `d`.
  - `public void print(int i)`: Prints `int` value `i`.
  - `public void print(Object o)`: Prints `Object` `o`.
  - `public void print(String s)`: Prints `String` `s`.
  - `public void println()`: Terminates the current line by writing the line separator string.
  - `public void println(boolean b)`: Prints `boolean` value `b` and then terminates the line.
  - `public void println(char c)`: Prints `char` value `c` and then terminates the line.
  - `public void println(char[] s)`: Prints array of `char` `s` and then terminates the line.
  - `public void println(double d)`: Prints `double` value `d` and then terminates the line.
  - `public void println(int i)`: Prints `int` value `i` and then terminates the line.
  - `public void println(Object o)`: Prints `Object` `o` and then terminates the line.
  - `public void println(String s)`: Prints `String` `s` and then terminates the line.

  Note that the `PrintWriter` class defines the same methods and constructors (except for the fact that the constructors are called `PrintWriter` instead of `PrintStream`).

- `Math` (package `java.lang`) Methods:

  - `public static double pow(double a, double b)`: Returns the value of `a` raised to the power of `b`.
  - `public static double sqrt(double a)`: Returns the correctly rounded positive square root of `double` value `a`.
  - `public static double random()`: Returns a `double` value with a positive sign, greater than or equal to `0.0` and less than `1.0`.
  - `public static double sin(double a)`: Returns the trigonometric sine of angle `a`, where `a` is in radians.
  - `public static double cos(double a)`: Returns the trigonometric cosine of angle `a`, where `a` is in radians.
  - `public static double tan(double a)`: Returns the trigonometric tangent of angle `a`, where `a` is in radians.
  - `public static double toDegrees(double angrad)`: Converts angle `angrad` measured in radians to an approximately equivalent angle measured in degrees.
  - `public static double toRadians(double angdeg)`: Converts angle `angdeg` measured in degrees to an approximately equivalent angle measured in radians.
  - `public static double exp(double a)`: Returns Euler's number e raised to the power of `double` value `a`.
  - `public static double log(double a)`: Returns the natural logarithm (base e) of `double` value `a`.
  - `public static double log10(double a)`: Returns the base 10 logarithm of `double` value `a`.

- `Character` (package `java.lang`) Methods:

  - `public static boolean isDigit(char ch)`: Determines if character `ch` is a digit.
  - `public static int digit(char ch, int radix)`: Returns the numeric value of character `ch` in the radix `radix`, `-1` if `ch` does not represent a digit.
  - `public static char forDigit(int digit, int radix)`: Returns the character representation of `digit` in the radix `radix`.
  - `public static boolean isLetter(char ch)`: Determines if character `ch` is a letter.
  - `public static boolean isLowerCase(char ch)`: Determines if character `ch` is a lowercase character.
  - `public static boolean isUpperCase(char ch)`: Determines if character `ch` is an uppercase character.
  - `public static boolean isWhitespace(char ch)`: Determines if character `ch` is white space according to Java.
  - `public static char toLowerCase(char ch)`: Converts character `ch` to lowercase.
  - `public static char toUpperCase(char ch)`: Converts character `ch` to uppercase.

- `ArrayList<E>` (package `java.util`) Methods:

  - `public ArrayList<E>()`: Creates a new empty `ArrayList` which contains elements of type `E`.
  - `public int size()`: Returns the number of elements in this list.
  - `public boolean isEmpty()`: Returns `true` if this list contains no elements.
  - `public boolean contains(Object o)`: Returns `true` if this list contains element `o`; comparisons are performed using the `equals()` method on `o`.
  - `public int indexOf(Object o)`: Returns the index of the first occurrence of element `o` in this list, or `-1` if this list does not contain this element; comparisons are performed using the `equals()` method on `o`.
  - `public E get(int index)`: Returns the element at position `index` in this list.
  - `public E set(int index, E element)`: Replaces the element at the position `index` in this list with the specified `element`.
  - `public boolean add(E e)`: Appends the specified `element` to the end of this list.
  - `public void add(int index, E element)`: Inserts the specified element at the position `index` in this list.
  - `public E remove(int index)`: Removes the element at position `index` in this list.
  - `public boolean remove(Object o)`: Removes the first occurrence of the specified element `o` from this list, if it is present; comparisons are performed using the `equals()` method on `o`.
  - `public void clear()`: Removes all of the elements from this list.

- `File` (package `java.io`) Methods:

  - `public File(String pathname)`: Creates a `File` representing the file at the given `pathname`.