IT 313 -- Midterm Exam

February 9, 2015

Part A: Multiple Choice Questions. Circle the letter of each correct answer. Give an optional reason or show your work for any answer to get partial credit (up to 4 points) in case you are not sure of your answer.
5 points each.

1. Which method call converts the **`String`** object **`s = "23.45"`** into the double value **`x`**?
   a. **`x = Convert.toDouble(s);`** b. **`x = (double) s;`**
   c. **`x = Double.parseDouble(s);`** d. **`x = s.toDouble( );`**

2. Which method call converts the int value n into the String object t?
   a. **`t = n.toString( );`** b. **`t = String.parseString(n);`**
   c. **`t = (String) n;`** d. **`t = String.valueOf(n);`**

3. Which scanner method reads the next line from the input stream?
   a. **`nextLine`** b. **`read`** c. **`readLine`** d. **`scanLine`**

4. An instance method is called by
   a. **`a class`** b. **`an external file`** c. **`an object`** d. **`another method`**

5. What is the most plausable output?

```
Random r = new Random( );
double m = 0.0;
for(int i = 1; i <= 1000; i++) {
    m = Math.min(m, r.nextDouble( ));
}
System.out.println(m);
```

   a. **`0.0`** b. **`0.001442515350896345`**
   c. **`0.4962678260983467`** d. **`0.9990904130582258`**

6. What is the symbol for redirecting standard err to a file?
   a. **<** b. **>** c. **>>** d. **2>**

7. What is wrong with these statements?

```
ArrayList<String> col = new ArrayList<String>( );
col.add = 5;
```

a. No initial capacity has been specified for the constructor.

b. The value 5 must be inserted into a wrapper class object before adding it to the collection.

c. The collection can only accept String objects (or objects of a class derived from String).

d. No index has been specified for where to add the new item in the collection.

8. What is the ultimate base class of every Java class?

a. **`Base`** b. **`Object`** c. **`Root`** d. **`super`**

9. How does Java indicate inheritance?

a. **`<`** b. **`:`** c. **`super`** d. **`extends`**

10. Which JUnit testing statement checks whether the age instance variable of the **`Person`** object p is 23?

a. **`assert(23, p.age( ));`** b. **`assertEqual(p.getAge( ), 23);`**

c. **`assertEqual(23, p.age( ));`** d. **`assertEquals(23, p.getAge( ));`**

Part B : Predict the Output. Predict the output of this Java code? 10 points.

1. What printed to the computer screen?

```
String items = "102112012", letters = "345123226", output="";
String[ ] cities = {"chicago", "denver", "seattle"};

for(int i = 0; i <= 8; i++) {
// The String instance method call s.substring(start, end);
// returns the substring starting at index start and
// ending with index end-1.
    int h = Integer.parseInt(items.substring(i, i+1));
    int k = Integer.parseInt(letters.substring(i, i+1));
    output += cities[h].charAt(k);
}
System.out.println(output);
```

Part C : Write Methods. Write a static method as specified. Also, write a main method with statements that test the method. Only write and test 1 of these 2 methods. 15 points.

1. Write a method named **`findMaxIndex`** that inputs an array of positive **`int`** values and returns the index of the element that contains the maximum value in the array. You may assume that the array arr has at least one element.

```
public static int findMaxIndex(int[ ] arr) {









}
public static void main(String[ ] args) {




}
```

2. Write a method named **`getMonthAbbrev`** that inputs a zero-based month number (0=January, … , 11=December). The method returns the month abbreviation. For full credit, use this String array:

```
String[ ] abbrevs = {"JAN", "FEB", "MAR", "APR", "MAY", "JUN",
    "JUL", "AUG", "SEP", "OCT", "NOV", "DEC"};
```

For brevity, you can write **`String[ ] abbrevs = {"JAN", ... ,"DEC"};`** in your method. Consider the case of an illegal month number input.

```
public static String getMonthAbbrev(int month) {









}
public static void main(String[ ] args) {




}
```

Part D: Short Essay Questions. Write about 1 to 2 paragraphs for your short essay. For full credit, write in full sentences and paragraphs. Answer only 1 out of 2 questions. 10 points.

1. What is code refactoring? What are some of the ways that Eclipse can help you refactor Java source code?

2. What is a JAR file? For what are JAR files used? How are they created?

Part E: The TaxiService Project

1. Correct the errors in this TaxiService project class. Mark the corrections on these pages. Do not recopy the code. There are about 5 errors in each of the Vehicle, Taxi, and Main classes. There are no errors in the Person class. 15 points.

```
==== Source code file Person.java ==========================
// There are no errors in this Person class.
public class Person {
    private String name;
    private char gender;
    private int age;
    public String getName( ) {
        return name;
    }
    public Person(String theName, char theGender, int theAge) {
        name = theName;
        gender = theGender;
        age = theAge;
    }
    @Override
    public String toString( ) {
        return String.format("%s %c %d", name, gender, age);
         }
     }

==== Source code file Vehicle.java ==========================
// There are about 5 errors in this Vehicle class.  Correct them.
private class Vehicle {
    private String vin, color, driver;

    public Vehicle(String theVin, String theColor,
        Person theDriver) {
            vin = theVin;
            color = theColor;
            driver = theDriver;
    }

    Override
    public String toString( ) {
        return String.format("VIN: %s Color: %s Driver: %d",
            vin, color, driver.getName);
    }
```

```
==== Source code file Taxi.java ============================
// There are about 5 errors in this Vehicle class. Correct them.
public class Taxi < Vehicle {
    private String pickupLocation;
    private String dropoffLocation;
    private Person passenger;

    public void Taxi(String theVin, String theColor,
        Person theDriver) {
        super(theVin, theColor, theDriver);
        pickupLocation = "";
        dropoffLocation = "";
        passenger = null;
    }

    public void pickup(Person thePassenger,
        String theDropoffLocation)
        Person passenger = the_Passenger;
        dropoffLocation = theDropoffLocation;
    }

    @Override
    public String toString( ) {
        String passengerName = "";
        Person p = getPassenger( );
        if p != null {
            passengerName = p.getName( );
        }
        else
        {
            passengerName = "No passenger";
        }
        return super.toString( ) + "\n" +
            String.format("Pickup Location: %s\n" +
                "Passenger Name: %s\n",
                "Dropoff Location: %s", pickupLocation,
                passengerName, dropoffLocation);
        }
    }
}
```

```
==== Source code file Main.java ============================
// There are about 5 errors in this main class.  Correct them.
public class Main {
    public static String main(String args)
        Taxi t1 = new Taxi("abc34567", "yellow",
            Person.new("Bruno", 'M', 45));
        t1.setPickupLocation("Art Institute");
        t1.pickup(new Person("Julie", 'F', 29),
            "Union Station");
        System.println(t1);
    }
}
```

2. Write the source code for the getter **`getPickupLocation`** and the setter **`setPickupLocation`** for the **`pickupLocation`** instance variable in the **`Taxi`** class. 10 points.

3. Write a Junit test class that tests the getter getPickupLocation and the setter setPickupLocation that you wrote in the preceding problem. 10 points.

```
import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.Test;

public class JUnitTest {



    @Before
    public void setUp() {









    }

    @Test
    public void test() {









    }
}
```

Part F: Use Methods from File Class from Java Class Library.

1. Select at least one constructor and at least four methods from the Java File class in the Java Class Library. See the printed abridged documentation for the File class on pages 10 through 12. Write a main method that calls each of the four methods that you select? For full credit, predict the output. 10 points.

Class File – Abridged Documentation from the Java Class Library

Package: java.io java.io.File has base class java.lang.Object
java.io.File

All Implemented Interfaces: Serializable, Comparable<File>

```
public class File
extends Object
implements Serializable, Comparable<File>
```

An abstract representation of file and directory pathnames.

User interfaces and operating systems use system-dependent *pathname strings* to name files and directories. This class presents an abstract, system-independent view of hierarchical pathnames. An *abstract pathname* has two components:

1. An optional system-dependent *prefix* string, such as a disk-drive specifier, `"/"` for the UNIX root directory, or `"\\\\"` for a Microsoft Windows UNC pathname, and
2. A sequence of zero or more string *names*.

The first name in an abstract pathname may be a directory name or, in the case of Microsoft Windows UNC pathnames, a hostname. Each subsequent name in an abstract pathname denotes a directory; the last name may denote either a directory or a file. The *empty* abstract pathname has no prefix and an empty name sequence.

The conversion of a pathname string to or from an abstract pathname is inherently system-dependent. When an abstract pathname is converted into a pathname string, each name is separated from the next by a single copy of the default *separator character*. The default name-separator character is defined by the system property `file.separator`, and is made available in the public static fields `separator` and `separatorChar` of this class. When a pathname string is converted into an abstract pathname, the names within it may be separated by the default name-separator character or by any other name-separator character that is supported by the underlying system.

A pathname, whether abstract or in string form, may be either *absolute* or *relative*. An absolute pathname is complete in that no other information is required in order to locate the file that it denotes. A relative pathname, in contrast, must be interpreted in terms of information taken from some other pathname. By default the classes in the `java.io` package always resolve relative pathnames against the current user directory. This directory is named by the system property `user.dir`, and is typically the directory in which the Java virtual machine was invoked.

The *parent* of an abstract pathname may be obtained by invoking the `getParent()` method of this class and consists of the pathname's prefix and each name in the pathname's name sequence except for the last. Each directory's absolute pathname is an ancestor of any `File` object with an absolute abstract pathname which begins with the directory's absolute pathname. For example, the directory denoted by the abstract pathname `"/usr"` is an ancestor of the directory denoted by the pathname `"/usr/local/bin"`.

The prefix concept is used to handle root directories on UNIX platforms, and drive specifiers, root directories and UNC pathnames on Microsoft Windows platforms, as follows:

- For UNIX platforms, the prefix of an absolute pathname is always `"/"`. Relative pathnames have no prefix. The abstract pathname denoting the root directory has the prefix `"/"` and an empty name sequence.
- For Microsoft Windows platforms, the prefix of a pathname that contains a drive specifier consists of the drive letter followed by `":"` and possibly followed by `"\\"` if the pathname is absolute. The prefix of a UNC pathname is `"\\\\"`; the hostname and the share name are the first two names in the name sequence. A relative pathname that does not specify a drive has no prefix.

Instances of this class may or may not denote an actual file-system object such as a file or a directory. If it does denote such an object then that object resides in a *partition*. A partition is an operating system-specific portion of storage for a file system. A single storage device (e.g. a physical disk-drive, flash memory, CD-ROM) may contain multiple partitions. The object, if any, will reside on the partition named by some ancestor of the absolute form of this pathname.

A file system may implement restrictions to certain operations on the actual file-system object, such as reading, writing, and executing. These restrictions are collectively known as *access permissions*. The file system may have multiple sets of access permissions on a single object. For example, one set may apply to the object's *owner*, and another may apply to all other users. The access permissions on an object may cause some methods in this class to fail.

Instances of the `File` class are immutable; that is, once created, the abstract pathname represented by a `File` object will never change.

Interoperability with `java.nio.file` package

The `java.nio.file` package defines interfaces and classes for the Java virtual machine to access files, file attributes, and file systems. This API may be used to overcome many of the limitations of the `java.io.File` class. The `toPath` method may be used to obtain a `Path` that uses the abstract path represented by a `File` object to locate a file. The resulting `Path` may be used with the `Files` class to provide more efficient and extensive access to additional file operations, file attributes, and I/O exceptions to help diagnose errors when an operation on a file fails.

Since: JDK1.0 **See Also:** Serialized Form

Field Summary

Modifier and Type	Field and Description
Static String	pathSeparator The system-dependent path-separator character, represented as a string for convenience.
Static char	pathSeparatorChar The system-dependent path-separator character.

Constructor Summary

Constructor and Description

`File(String pathname)`
Creates a new `File` instance by converting the given pathname string into an abstract pathname

`File(URI uri)`
Creates a new `File` instance by converting the given `file:` URI into an abstract pathname.

Method Summary

Modifier and Type	Method and Description
`boolean`	**`canRead()`** Tests whether the application can read the file denoted by this abstract pathname. Returns: true if and only if the file specified by this abstract pathname exists *and* can be read by the application; false otherwise. Throws: SecurityException - If a security manager exists and its checkRead method denies read access to file.
`boolean`	**`canWrite()`** Tests whether the application can modify the file denoted by this abstract pathname. Returns: true if and only if the file system actually contains a file denoted by this abstract pathname and the the application is allowed to write to the file; false otherwise. Throws: SecurityException - If a security manager exists and its checkWrite method denies write access to file.
`boolean`	**`createNewFile() throws IOException`** Atomically creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist. Returns: true if the named file does not exist and was successfully created; false if the named file exists. Throws: IOException if an I/O error occurred. SecurityException - If a security manager exists and its checkWrite method denies write access to file.

boolean **delete()**

Deletes the file or directory denoted by this abstract pathname. If this pathname denotes a directory, then the directory must be empty in order to be deleted. Note that the Files class defines the delete to throw an IOException when a file cannot be deleted. This is useful for error reporting and to diagnose why a file cannot be deleted.

Returns: true if and only if the file or directory is successfully deleted; false otherwise.

Throws:

boolean **exists()**

Tests whether the file or directory denoted by this abstract pathname exists.

Returns: true if and only if the file or directory denoted by this abstract pathname exists; false otherwise.

Throws: SecurityException - If a security manager exists and its checkRead methhod denies read access to the file or directory.

String **getAbsolutePath()**

Returns the absolute pathname string of this abstract pathname. If this abstract pathname is already absolute, then the pathname string is simply returned as if by the getPath() method. If this abstract pathname is the empty abstract pathname then the pathname string of the current user directory, which is named by the system property user.dir, is returned. Otherwise this pathname is resolved in a system-dependent way. On UNIX systems, a relative pathname is made absolute by resolving it against the current user directory. On Microsoft Windows systems, a relative pathname is made absolute by resolving it against the current directory of the drive named by the pathname, if any; if not, it is resolved against the current user directory.

Returns: The absolute pathname string denoting the same file or directory as this abstract pathname.

Throws: SecurityException - If a required system property value cannot be accessed.

boolean **getName()**

Returns the name of the file or directory denoted by this abstract pathname.

Returns: The name of the file or directory denoted by this abstract pathname, or the empty string if this pathname's name sequence is empty

boolean **isAbsolute()**

Tests whether this abstract pathname is absolute. The definition of absolute pathname is system dependent. On UNIX systems, a pathname is absolute if its prefix is "/". On Microsoft Windows systems, a pathname is absolute if its prefix is a drive specifier followed by "\\", or if its prefix is "\\\\".

Returns: true if this abstract pathname is absolute, false otherwise.

String **isDirectory()**

Tests whether the file denoted by this abstract pathname is a directory. Where it is required to distinguish an I/O exception from the case that the file is not a directory, or where several attributes of the same file are required at the same time, then the Files.readAttributes method may be used.

Returns: true if and only if the file denoted by this abstract pathname exists *and* is a directory; false otherwise.

Throws: SecurityException - If a security manager exists and its checkread method denies read access to file.

long **length()**

Returns the length of the file denoted by this abstract pathname. The return value is unspecified if this pathname denotes a directory. Where it is required to distinguish an I/O exception from the case that 0L is returned, or where several attributes of the same file are required at the same time, then the Files.readAttributes method may be used.

Return: The length, in bytes, of the file denoted by this abstract pathname, or 0L if the file does not exist. Some operating systems may return 0L for pathnames denoting system-dependent entities such as devices or pipes.

Throws: SecurityException - If a security manager exists and its checkRead method denies read access to file.

boolean **mkdir()**

Creates the directory named by this abstract pathname.

Return: true if and only if the directory was created; false otherwise.

Throws: SecurityException - If a security manager exists and its checkWrite method does not permit the named directory to be created.