Student name: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Instructor: Haining Chen

Class meeting time (e.g., online, or Mon 5:30pm, or Tues/Thur 9:30am, etc):\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**CS 2163 Spring 2020 Semester JAVA TEST 2 STUDY GUIDE**

**To earn 10 bonus points**, answer this study guide and submit the **electronic copy** to the corresponding Moodle drop box **before taking test 2**. I will NOT grade your submitted answers, but I will look at your submission to see if you have answered all the questions or not. **For every question not answered**, **I will be deducting one point from the 10 points.** There is no late submission accepted or makeup for this study guide. **Please DO NOT SUBMIT paper copy.**

You need to place your answer right after or besides each question in this document.

Can you finish this study guide **without looking at textbook book or being on a computer?** You will NOT have computer or textbook access when you are taking the test. After you complete this study guide, you may want to test questions that you are sure on the computer. If the result is not correct, you can discover where it goes wrong and then correct it. **Test 2 is a paper-and-pencil close book test**, and you may want to bring a rubber eraser **A letter-size paper (8.5 by 11 inches) with hand-written notes on both sides is allowed in the test.** This notes paper must be a complete piece of paper, and it cannot be a partial paper with tearing-apart lines. This notes paper needs to be turned in together with your test paper, and you cannot bring the notes out of the testing center. If you turn in the notes paper with tearing-apart lines, the score will be 0 for this test. **Computer/phone/calculator/book is NOT allowed in the test**. The total points for test 2 is 100.

Test 2 covers textbook chapter 5, 6, 7, 8 and 9 including the Moodle homework in each lesson. Test 2 also covers the materials in this study guide document. In addition, you may want to review the OOP principle of “data abstraction” and “encapsulation” in Moodle folder “chap 3” and “chap 4”.

**OOP Coding requirement**

You need to study all the homework submissions from hw #5 to hw #9:

* homework 5 – focus on the encapsulation rule of OOP, and know how to use public getter/setter/effector to access /modify the private data of another class
* homework 6 – focus on the array and sorting algorithm
* homework 7 – focus on the inheritance principle of OOP, and know how to design subclass inherited from superclass
* homework 8 – focus on the polymorphism principle of OOP, and know how to use array and dynamic binding to navigate through an array of parent type but filled with different subclass objects, without having to know the exact subclass type of each array element.
* homework 9 – focus on the use of generic programming (static binding) class such as ArrayList<> class type, or List<> class type. With homework7, 8 and 9 together, we cover all four principles in OOP including all the contexts of polymorphism

In test 2, there will be a similar coding exercise as homework 7. You need to read document “***lesson7-schedule.docx***” Moodle folder “chap 7”, and understand how inheritance works, and what benefit can inheritance bring in class design.

As a practice, work on this coding exercise below to enhance your understanding of inheritance in class design. This exercise provides you an opportunity to practice a coding questions similar to homework 7, so that you can be better prepared when you see the coding question in test #2.

For this coding exercise, you need to create a new Java project in Eclipse, and finish all the coding in Eclipse. Then run and debug your Eclipse project, and make sure it works. Then you can just copy and paste the java source code from Eclipse into the answer area above.

All data are private and all methods are public in each class/interface in the exercise below.

Code a class named **RentalItem**. This class has three private data members: id of int type, description of String type, and processingFee of double type. Provide a default constructor with no parameters, and a constructor with three parameters to initialize its three data members, respectively. Provide a getter method and a setter method for each data member.

|  |
| --- |
| /\*\* RentalItem.java \*/ |

Code a class named **Car** that inherits class **RentalItem**, which has two private data: mileTravelled of int type, and chargesPerMile of double type. Provide a default constructor with no parameter, and a constructor with all parameters to initialize its own private data and the inherited data from its superclass **RentalItem**. Provide a getter method and a setter method for each of its own private data member. Provide a method named *calcRentalFee()* with double as its return type, and it takes no parameter. In this method, the rental fee is calculated by multiplying mileTravelled by chargesPerMile, then adding the multiplication result to processingFee.

|  |
| --- |
| */\*\* Car.java \*/* |

Code a class named **Tool** that inherits class **RentalItem**, which has two private data: daysRent of int type, and dailyRentalFee of double type. Provide a default constructor with no parameter, and a constructor with all parameters to initialize its own private data and the inherited data from its superclass **RentalItem**. Provide a getter method and a setter method for each of its own private data member. Provide a method named *calcRentalFee()* with double as its return type, and it takes no parameter. In this method, the rental fee is calculated by multiplying daysRent by dailyRentalFee, then adding the multiplication result to processingFee.

|  |
| --- |
| */\*\* Tool.java \*/* |

Code a class named **JohnDoeTest** that has a main method. In the main method, create two objects: one **Car** object, and one **Tool** object. When creating these two objects, you need to use the constructor that has all parameters in class **Car** and **Tool**, respectively. Hardcode reasonable values for the constructors’ parameters, and there is no need to ask user to input data.

Then in the main method, invoke the necessary methods of the Car object, and output its id, description, and rental fee. Do the same thing for the Tool object.

Then the main method ends, and the class ends.

|  |
| --- |
| */\*\* JohnDoeTest.java, and replace JohnDoe with your name\*/* |

Summary: in Modle lesson “chap 7”, the source code in “***inheritanceExample.zip***” illustrates the class-interface design patterns; then in homework 7, you work a class design exercise using inheritance; now in the exercise above, you work on this inheritance topic again. Therefore, when you see a similar coding exercise in test #2, it will be the fourth time that you work on the inheritance class design exercise, and you should have no problem finishing it successfully.

**Java naming conventions:**

* Java naming convention follows CamelCase (or UpperCamelCase ), a naming convention in which a name is formed of multiple words that are joined together as a single word with the first letter of each of the multiple words capitalized, so that each word that makes up the name can easily be read.
* Java also follows lowerCamelCase, in which the first letter of the first word is lowercase of the CamelCase.
* Names that start with lower case letter ( lowerCamelCase ): variable/object/instance name, package name, method (except constructor) name. **Examples**: java.util ; calculateAverage(); studentsPerSession
* Names that start with upper case letter ( UpperCamelCase ): class/interface name, constructor name. **Examples**: class name such as Actor, Scanner, Rocket, and their constructor names; interface name such as Profitable
* Names that have to be all upper-case letters, with underscore connecting adjacent words: static final variables, in other words, constant variables. **Examples**: MAX\_CREDITS\_PER\_SEMESTER; DAILY\_RENTAL\_FEE
* For boolean type variable, it usually start with keyword is, such as isEligible, isRaining.
* All getter methods start with keyword get, except for the getter for boolean variable, which usually starts with is, such as ***public boolean isMacheRunning*** .
* All setter methods start with keyword set.

**The Four Principles in OOP (Objected-Oriented Programming)**

We reviewed the four OOP principles in test 1 already, and we are still covering them in test 2.

For each principle and context of OOP, you need to be able to achieve these three levels of understanding:

1. understand the definitions as listed below
2. on top of that, you need to associate each definition with java source code examples that we have gone through either in textbook java source code examples or in our Moodle zip files’ java source code examples
3. eventually, you need to be able to write your own Java code incorporating these principles and context

In file “chap1-schedule.docx”, we covered the **four principles** of OOP: **data abstraction, encapsulation, inheritance,** and **polymorphism** , and their definitions are listed as below:

* **data abstraction** : real-world modelling by taking out the relevant data out of the real-world objects to form a class.
* **Encapsulation:** private data is encapsulated (protected) by public methods
* **inheritance** : subclass “is-a” superclass
* **polymorphism** with the following context:
* **method overloading** : in the same class, same method name, different method signature
* **method overriding** : the same method name and the same method signature appearing in superclass and subclass, respectively
* **parent reference refers to child object** : a superclass variable can refer to a subclass object
* **dynamic binding** : late-binding, runtime-binding, or, it is the object being referred to that determines which overridden method to invoke, not the variable type that can determine
* **static binding** (generic programming) : the pair angle brackets <Element > enclosing the Element class type, and the class type is determined in the compile-time, which includes the generic type and the element type, therefore, ***ArrayList<String>*** is a different type than ***ArrayList<Student>***. Other names for static binding include early-binding, and compile-time binding.

We cover **data abstraction and encapsulation** in file“OOP-BasicIntro.ppt” in Moodle “chap 2” folder. Data abstraction is to build a class from real world entity, i.e., real world modeling, by abstracting the characteristics of a read world entity into private class data and public class methods of a class, and these data and methods are relevant to the application of the class.

Encapsulation is to hide the private data with public methods of the class, so that the user does not see the private data directly. The public method serves as a bridge to connect the private data of a class to the user of this class.

We cover **examples of inheritance** in every Greenfoot scenario. For example, class MyWorld inherits from class World, and class Wombat inherits from class Actor, etc. We also cover inheritance in homework 7.

Notice that in inheritance, the parent class is called superclass, and the child class is called subclass. In the UML class diagram, the subclass has a solid line with a **triangle shape (not an arrow shape ^)** at the top, pointing to the superclass, as indicated below.

|  |  |  |
| --- | --- | --- |
| Superclass  is-a or is-an  Subclass | Example 1:  Actor  is-an  Wombat | Example 2:  Person  is-a  Staff |

The inheritance relationship shows the “is-a” relationship, such as: a wombat “is-an” actor. The “is-a” only goes one way, and we **CANNOT** say: An actor “is-a” wombat”.

We cover **method overloading** with several examples, such as the overloaded constructors in class Asteroid (in chapter 1’s scenario asteroids1), or the overloaded method setImage() in class Actor:

**void** [**setImage**](http://www.greenfoot.org/files/javadoc/greenfoot/Actor.html#setImage(greenfoot.GreenfootImage))([**GreenfootImage**](http://www.greenfoot.org/files/javadoc/greenfoot/GreenfootImage.html) image)

**void** [**setImage**](http://www.greenfoot.org/files/javadoc/greenfoot/Actor.html#setImage(java.lang.String))(java.lang.String filename)

You can access all the java document of Greenfoot API via this link: <http://www.greenfoot.org/files/javadoc/>

We cover **method overriding** with examples like: a subclass of Actor such as Crab overrides the act() method in Actor, so that Crab class has its own act() method. So does the Lobster class who also overrides the act() method in its superclass Actor, and so does class Worm.

We cover **parent reference (parent variable) refers to child object** in Greenfoot textbook chapter 4’s example code CrabWorld.java (in scenario “little-crab-5”). Look at the code below:

Crab crab = new Crab();

addObject(crab, 231, 203);

Worm worm = new Worm();

addObject(worm, 445, 137);

Lobster lobster = new Lobster();

addObject(lobster, 334, 65);

And the method definition of addObject in class World is:

**void** [**addObject**](http://www.greenfoot.org/files/javadoc/greenfoot/World.html#addObject(greenfoot.Actor, int, int))([**Actor**](http://www.greenfoot.org/files/javadoc/greenfoot/Actor.html) object, int x, int y)

From the line of code above, notice that the first formal parameter of method addObject() is of type Actor, and the three different actual parameters plugged in are crab, worm, and lobster, which are of type Carb, Worm, and Lobster, respectively. These three class types, Crab, Worm and Lobster, are all subclasses of the superclass Actor, thus we can see that: the parent referent (of type Actor) refers to the child object (of type Crab, Worm, and Lobster).

However, **siblings cannot refer to each other**. For example, a variable of class Crab type CANNOT refer to a Worn object, and a variable of class Worm type CANNOT refer to a Crab object, either.

We cover dynamic binding in file “exampleOfClassInterfaceDesign.zip” from Moodle folder “chap 8”.

We cover static binding in file “***dynamicBinding-staticBinding.zip***” from Moodle folder “chap 7”. We also cover it in Greetfoot “chap 7” project “autumn -2”, with the use of generic type List<Leaf>.

**In summary, we have covered all four principles of OOP including all 5 contexts of polymorphism, principle, with our source code examples from Greenfoot textbook, or from additional zip files in Moodle folders.**

Method signature is defined as method name and the formal parameter list of this method. In Java, method signature does not include method return type. In the formal parameter list, the number of parameter matters, and the type of each parameter matters, and the sequence of the parameters matters. In Java, method signature does not include method return type, and the textbook page 9’s definition of method signature had it wrong.

In the same class, a method signature uniquely identifies this method within the entire class; therefore, it is not allowed to have two methods appearing in the same class, if these two methods have the same method signature.

Method overloading is defined as two or more methods in the same class share the same name, but they must have different formal parameter list, which means either the number of parameters, or the type of parameters, or the sequence of parameters must be different.

Method overriding is defined as a method appearing in the superclass and its subclass at the same time, and both appearances have the same method name and parameter list, i.e., the same signature.

Answer the following questions:

What is the definition of method overloading?

Do the overloaded methods share the same method name?

Do the overloaded methods share the same method signature?

Can the overloaded methods share the same formal parameter list?

Can the overloaded methods have the same return type?

Can the overloaded methods have different return type?

Given the two method definitions:

public static double m (double x, double y) // version 1

public static double m (int x, double y) // version 2

, please tell which method is invoked for :

double z = m(4.5, 7.9); // version \_\_\_ is invoked

double z = m(4, 7.9); // version \_\_\_ is invoked

double z = m(4, 7); // version \_\_\_ is invoked

For now, the only type conversion you need to know is the automatic type conversion from int to double, if the actual parameter is int, while the formal parameter is double. But there is no automatic type conversion from double to int.

Assume we have this method defined in a class: **public double getBalance(int accountNumber)**, and assume that methods below are defined in the same class.

Question: are the following method allowed or not allowed?

public double getBalance(int socialSecurityNumber)

public String getBalance(int accountNumber)

public double getBalance(String accountName)

public double getBalance(boolean accountStatus)

What is wrong with the following code? The method body is omitted deliberately to save space.

public class Test {

public static void calcScore (int x){

}

public static void calcScore (int y){

}

}

The inheritance relationship implies “is-a” relationship, for example, when class HourlyWorker inherits from class Employee, we can say that an HourlyWorker person “is-an” Employee, but we cannot say that an Employee person is an HourlyWorker because an HourlyWorker object has hourly rate and number of hours worked as its data members, but an Employee object does not have these two data properties. Please refer to source code ***Employee.java*** and ***HourlyWorker.java*** in file “***exampleOfClassInterfaceDesign.zip***”.

When class A’s private data is of type class B, then we can say that class A “has-a” class B. Please do not confuse the “is-a” relationship with “has-a” relationship. An example of “has-a” relationship is as below:

public class Department {

private String departmentName;

private Employee secretary;

}

, and we can say that Department “has-an” String type id, but we CANNOT say that String has a Department. It is Department “contains” a String, not the other way around.

And obviously, their relationship here is not “is-a”, because we cannot say: a department “is-an” id, nor can we say: an id “is-a” department.

Another word for “has-a” relationship is “containership”. We can say: Department contains an Employee.

In inheritance, the subclass is a more detailed example of a superclass, and superclass is a more general model of a subclass. For example, Orange class is a more detailed example of Fruit class, and Fruit class is a more general example of Orange class. That is why in UML class diagram, the line that connects the superclass and subclass is named as “Generalization” in the “strong inheritance” scenario, because superclass is a more general example of subclass. In the “weak inheritance” case, the UML class diagram use dotted line instead of solid line, and the name in Visual Paradigm is “Realization”, because a subclass of an interface realizes the abstract method defined in the interface.

**Loop, Array and ArrayLisy**

Review textbook Appendix D in page 268 and page 269 for the format of these loops: for loop, while loop, and do-while loop, and for-each loop. You also need to know how to convert among these loops.

Given this while loop below, please convert it to a for loop, and a do-while loop, respectively.

int i = 1;

while(i < 100){

System.out.printf("%4d\n", i);

i += 2;

}

Given this loop below, what is wrong with it? Please place this loop in a main method of your test code in Eclipse, and run the code, what do you see from the output of the code? Can the code terminate or not? Please explain your observation.

int i = 1;

while(i < 100)

System.out.printf("%4d\n", i);

i += 2;

For an array, you need to know that in Java, there is a constant variable associated with each array, and it is: arrayName.length , which has to be determined when you initialize an array variable, like this:

int[] scores = new int[size];

, where the size as an integer type variable must be initialized before this new statement. And the valid index of an array goes from the lowest index 0 to the highest index of (array.length - 1). For example, an array of size 4, the highest index is 3, not 4, and the lowest index is 0, and an index cannot go negative.

Please read the document “MemoryManagementForArray.docx”, and understand that array name is a reference type of variable, and the content of the array is in the heap, with the beginning memory address of the loop content in the heap being saved in the array name.

You need to understand the difference between post-incremental and pre-incremental operator.

Post-incremental operator: a++

Pre-incremental operator: ++a

For post-incremental, the value of a is used first, and then the adding 1 to variable ***a*** is done later.

For pre-incremental, the value of a is used later, after the adding 1 to variable ***a*** is done first.

For code like this:

int a = 7;

int b = a++;

System.out.printf("a is %d, b is %d\n", a, b);

, the output result will be “a is 8, b is 7”

For code like this:

int a = 7;

int b = ++a;

System.out.printf("a is %d, b is %d\n", a, b);

, the output result will be “a is 8, b is 8”.

For code like this:

int a = 7;

System.out.printf("a is %d\n", a++);

System.out.printf("a is %d\n", a);

What are the two lines of output results?

Do a trace on the for loop below, and show each iteration of the loop in the tracing table until the loop finishes. Each row in the table corresponds to one iteration of the for loop.

int j=8, k=9;

for(int i = 0; i < 5; i++){

k = k - j++ \* i;

}

The tracing table is here:

|  |  |  |  |
| --- | --- | --- | --- |
| **i** | **i<4** | **j** | **k** |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

You can place the code above into an Eclipse project, and add some printf method to output the value of i, j, and k, in each iteration, in order to verify your manual tracing result in the tracing table.

Given the main method below, fill in the missing line that invokes a method named *calcAverage*, and this method takes the double type array prices as the only actual parameter.

public static void main (String[] args)

{

double[] prices = {15.20, 8.59, 12.47};

double average;

// finish the code below to invoke method *calcAverage* which takes array pricesdefined above as the only actual parameter.

average = \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_;

System.out.printf ("average is %8.2f: " , average );

}

Given the main method above, implement a public method named *calcAverage* in the same class of the main method. This method is called by the main method above. This method *calcAverage* takes a double type array as the only formal parameter, and this method returns a double type. You need to first provide the correct method header below. In the method header line, before the method name *calcAverage*, you need to fill in three words above the three underscores, with one word per underscore. Inside the parenthesis after the method name, you need to fill in the type (occupying the 1st underscore) and the name of the formal parameter (occupying the 2nd underscore) of this method. Then in the method body, you need to first sum up all elements using a regular **for** loop *(*not a *for each* loop*)*. After you obtain the sum, then calculate the average by dividing the sum by the number of elements in the array. Finally, return the average. You can define local variables inside the method body as needed.

\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_ calcAverage ( \_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_ )

{

} // end of method body for method calcAverage;

Embedded loop should be studied together with two-dimensional arrays. Please read the source code example file “TwoDimArrayDemo.java”, and learn how to use embedded loops to navigate a two-dimensional array.

Given a two-dimensional array named ***scores*** with double type elements, write the embedded loop to navigate this array and multiply each element in the array by 10, and put the embedded loop inside a method named **increase** , and the only parameter of this method is a double type two-dimensional array. This method **increase** should have void as its return type.

Write the one line of code inside main to invoke the **increase** method mentioned above. Do you need to receive the return value of **increase** with a local variable, why or why not?

Given a two-dimensional array named ***scores*** with double type elements, write the embedded loop to navigate this array and summarize all items in this array, and put the embedded loop inside a method named **sum** , and the only parameter of this method **sum** is a double type two-dimensional array. This method **sum** should return the summarized total, therefor its return type should be double.

Write the one line of code inside main to invoke the **sum** method mentioned above, and receive the return value with a local variable. What type should this local variable be? Then use a second line of code to output the return value to the console. Assume the actual parameter (the double type two-dimensional array) you plug in for method ***increase*** has been initialized.

In this line of code : Student[] students = new Student[5];

, students is the array name, which is a reference variable pointing to the array content in the heap. In the array content, there are 5 spaces reserved in the heap, and each space will be used to store a reference to a Student object. However, with only the new statement above, **there is no Student object being created (constructed, or instantiated) yet**, because there is no constructor of the Student class being invoked yet.

In order to initialize the array content, we need the following statements:

students [0] = new Studnt("Mike", 3.4);

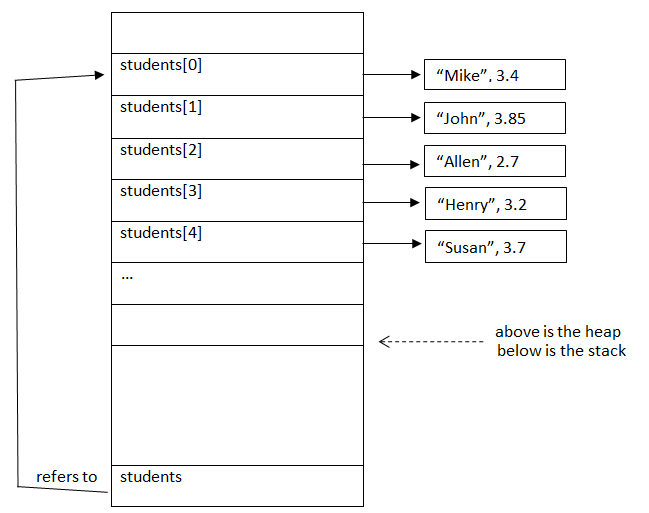
students [1] = new Studnt("John", 3.85);

students [2] = new Studnt("Allen", 2.7);

students [3] = new Studnt("Henry", 3.2);

students [4] = new Studnt("Susan", 3.7);

, so after the above 5 new statements, the memory looks like this:



For the for-each loop, read the comparison of for-each loop and a regular for loop in file “chap7-schedule.docx”, and understand what are the things that a regular for loop can do, but a for-each loop cannot do.

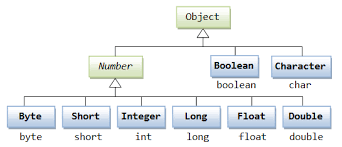
For a generic class, such as ArrayList, we need to plug in an element type to make it a concrete data type, such as ArrayList<String>, or ArrayList<Student>.

Look at line 26 and 27 in file “ArrayListDemo.java”, which can be found after you unzip file “dynamicBinding-staticBinding.zip” in Moodle folder “chap 7”.

Look at the API of ArrayList<>, the first two constructors are like this:

|  |  |
| --- | --- |
| [**ArrayList**](https://docs.oracle.com/javase/9/docs/api/java/util/ArrayList.html#ArrayList--)**<ElementType>** ​() | Constructs an empty list with an initial capacity of ten. |
| [**ArrayList**](https://docs.oracle.com/javase/9/docs/api/java/util/ArrayList.html#ArrayList-int-)**<ElementType>** (int initialCapacity) | Constructs an empty list with the specified initial capacity. |

For generic class like ArrayList, the class name is not complete until the ElementType class is filled in. For example, ArrayList is not a complete class name, while ArrayList<Integer> is, ArrayList<Student>, ArrayList<BankAccount> is, and ArrayList<Car> is. For primitive types, since they cannot be plugged into the generic class type like ArrayList as ElementType, the solution in Java is to define the wrapper class for each primitive type, so that the wrapper class can be plugged in as ElementType in ArrayList class. A picture of wrapper class and primitive type is shown as below:



An example of using the first default constructor of ArrayList that has no formal parameter is as below:

ArrayList< BankAccount > listA = new ArrayList< BankAccount > ();

, and pay attention that, it is a pair of parenthesis after >, the angle bracket symbol. The initial size is 10 for the array list created, according to the description of the default constructor.

An example of using the second default constructor of ArrayList that has a formal parameter is as below:

ArrayList< BankAccount > listB = new ArrayList< BankAccount > (7);

, and pay attention that the parameter surrounded by the pair of parenthesis is 7, which means the initial size if 7 for the array list created.

For generic class like ArrayList, the capacity of the list can be expanded as needed, and here is how it works: take listA above as an example whose initial size is 10, so when can use this method below:

for(int x=0; x<15; x++) //the incremental argument is empty.

{

listA.add(new BankAccount(x, (x+1)\*10.0));

}

, you can see that, a total of 15 BankAccount objects were added into the list. When the initial capacity of 10 is reached, what happen behind the scene is that, JVM will allocate double the size of the initial capacity and allocate a space in the heap that can hold 20 objects, and then the initial 10 objects’ references are copied from the original space in heap to the newly allocated space in heap that has 20 seats for the objects, and then the remaining 5 objects will be created and placed in the new seats. The original place of 10-seats space in heap will be released, and listA now points to the new placed that has 20 seats. If you keep added more objects into the list and the capacity of 20 seats is not enough, then it will double the size of the array again by allocating a new space in heap that has 40 seats for objects, and then copy-and-paste the first 20 elements from the previous 20-seats to the 40-seats space in heap, and then release the 20-seats space, and let listA points to the 40-seat space in heap. This process repeats itself when it is necessary to accommodate more objects than its current ArrayList capacity.

Now you should understand that for listB declared in the code above and inserted with objects in the code below, it can still accommodate 15 objects, even though its initial capacity is 7. Here is how it works: its size is expanded from 7 to 14, then to 28 to hold 15 objects.

for(int x=0; x<15; x++) //the incremental argument is empty.

{

listB.add(new BankAccount(x, (x+1)\*10.0) );

}

For the above discussion, you can see one fundamental difference between ArrayList and regular array is that: ArrayList can expand its capacity when needed, but the size of array is fixed when this array is declared, and the array cannot expand, such as:

BankAccount [] accounts = new BankAccount[8];

, and the array size is 8, and the array size cannot be modified after the array declaration line.

**String class type and String object input**

Read online api of String class: <https://docs.oracle.com/javase/8/docs/api/java/lang/String.html>

, and learn how to use these String methods:

public char charAt(int index)

public int compareTo([String](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html) anotherString)

public boolean equals([String](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html) anotherString)

public int indexOf(int ch)

public int length()

public [String](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html) substring(int beginIndex)

public static [String](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html) valueOf(int i)

public static [String](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html) valueOf(double d)

|  |
| --- |
| Given a String variable ***bookTitle*** that has the value of “Introduction to Java”, use a regular for loop to output the individual character of this String variable, with one character per line. |

Can we use ***if ( s1 = = s2 )*** to compare whether the two String object s1 and s2 has the same string content? If not, then how to compare the content of two String objects ?

What are the method return values of the following method invocations? Assume String s1 = "Forrest";

s1.length()

s1.indexOf('r')

s1.substring(2)

String.valueOf(14)

String.valueOf(14.5)

For String object input, what is the difference between next() method and nextLine() method in class Scanner? If the user input is “John Doe”, with a space separating the first and last name, what is the result being reading in with these two methods: next() and nextLine(), respectively ?

This link below gives an detail explanation of how to format output using the System.out.printf() method, and we need to know how to use these three format specifiers: %s, %f, and %d, and they are for String, double, and integer types, respectively.

<http://alvinalexander.com/programming/printf-format-cheat-sheet>

We also need to know that method String.format() has the same formal parameter list as method System.out.printf(), so if you know how to use one, you know the other. The only difference them is that: String.format() send the formatted result to a String object, while System.out.printf() method send the formatted result to the output console on the screen.

The number of format specifiers in printf() or in format() must match the number of output variables, and each format specifier must also match the corresponding output item’s data type.

For example, if you have three format specifiers, but you have only two output variables, the code will not compile.

Also, if you use a %f format specifier to format a integer variable, the code will not compile either.

What are the errors in the code below?

System.format("%9s %12d %5.2f %6d ", 12.6, "Computer", 57, "Technology" );

The toString() method is a method that will be invoke by an object, if this object is an output item in method printf, println, or print, under System.out, regardless the class type of this object. Therefore, if you want to customize the output content of an object, you need to provide your own implementation of the toString() method in the class. The original toString() method is in class Object, and if your class provide your own version of toString(), that is called overriding the toString() method.

**Conditional operator**

Conditional operator can be used to replace an if-else statement, and it is also called ternary operator, because it needs three parameters. For example,

( profit <= 0 ) ? bonus = 0 : bonus = profit \* 0.1;

, and the above line of conditional operator has the same logic as the if-else statement below:

if ( profit <= 0 ){

bonus = 0;

}else {

bonus = profit \* 0.1;

}

**UML class diagram**

In the UML class diagram, static data and static method have to be marked as “Class Scope”, resulting in an underscore for static data and method. Also, we know that the + sign (plus sign) in UML class diagram means public, and - sign (minus sign) means private.

**Java type conversion:**

Given x and y’s definitions below, what is the value of z in the table below?

|  |  |
| --- | --- |
| int x = 3;  int y = 5;  double z = x / y; | int x = 3;  int y = 5;  double z = (double) x / y; |

**Patterns to invoke methods in Java: static vs non-static method:**

The patterns for invoking static and non-static methods are summarized as below:

**ClassName**.***publicStaticMethodName***(parameter list …);

**objectName**.***public*NonStatic*MethodName***(parameter list…);

In addition, we have this pattern to access a public static final variable defined of a class:

**ClassName**.***publicStaticFinalVariableName***

, and an example is :

Math.PI

, where Math is a class in java package java.lang, and PI is a public static final variable defined in class MATH.

Also, this pattern below is wrong, because it violates the encapsulation principle of OOP.

**objectName**.nonStaticPrivateVariableName; // this statement is WRONG!!!!

**Search and sorting**

Study the WORD document and sample source code in file “search-and-sorting.zip” in Moodle folder “chap 6”, and answer the following questions:

What are the Big-O notations for these search algorithms: binary search and linear search ? Which is faster?

What are the Big-O notation for these sorting algorithms: bubble, insertion, selection, and quick sort? Which one is faster?

Here is a list of zip files you need to study. In each zip file, it may include java source code files and WORD documents, and you need to study both java source code files and WORD documents.

|  |  |
| --- | --- |
| Moodle folder | Zip file name |
| Chap 5 | No zip file |
| Chap 6 | MemoryManagementForArray.zip  search-and-sort.zip  TwoDimArrayDemo.zip |
| Chap 7 | inheritanceExample.zip  dynamicBinding-staticBinding.zip |
| Chap 8 | exampleOfClassInterfaceDesign.zip |
| Chap 9 | No zip file |
| Test 2 review Lesson | TraceCodeExample.zip |

Do a trace on the code below. In the method binarySearch below: variable **key** holds the value **27**, and

variable **list** is a reference to an array with these values **{12, 25, 36, 39, 43, 65, 78, 86, 99, 108, 121}**.

public static int binarySearch(int[] list, int key) {

int lowIndex = 0;

int highIndex = list.length - 1;

while (highIndex >= lowIndex) {

int midIndex = (lowIndex + highIndex) / 2;

if (key < list[midIndex]){

highIndex = midIndex - 1;

}

else if (key > list[midIndex]){

lowIndex = midIndex + 1;

}

else if (key == list[midIndex]){

return midIndex;

}

} // end of while loop

return -1;

} // end of binary search method

Each row in the table below corresponds to one iteration of the while loop in the method above. You can add or remove rows according to the actual number of iterations. The first row’s information corresponds to the first iteration, and its value has been filled. You need to continue for the following rows.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **key** | **lowIndex** | **highIndex** | **highIndex>=lowIndex** | **midIndex** | **key==list[midIndex]** | **key<list[midIndex]** |
| 27 | 0 | 10 | true | 5 | false | true |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |

Given the key value and array content listed above, what is the return value of the binary search method?

Re-do the code tracing, and this time, the key is changed to 80, as indicated by the table below.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **key** | **lowIndex** | **highIndex** | **highIndex>=lowIndex** | **midIndex** | **key==list[midIndex]** | **key<list[midIndex]** |
| 80 | 0 | 10 | true | 5 | false | false |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |

How does binary search algorithm work? *( use simple words as if to explain to a non-technical person)* \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_