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

Class meeting time : \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Mon/Wed 11.00 am to 13.35pm

**CS 2163 Spring 2020 Semester JAVA TEST 1 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 1**. 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 of this document.** Just place your answer below each question in this document.

Can you finish this study guide **without looking at textbook book or using a computer?** You will NOT have a computer or calculator when you are taking the test. After you complete this study guide, you may want to test questions that you are not sued on the computer. If the test result is not correct, you can find out where it goes wrong and then correct it. **Test 1 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 in test 1 is 100.

**Binary and hexadecimal and decimal**

There are many study materials online for binary, hexadecimal and decimal, such as this one:

<http://www.mathsisfun.com/binary-decimal-hexadecimal.html>

You need to understand these three number systems, and how to convert among the three numbering systems, and memorize this three-column table of binary-hex-decimal conversion.

|  |  |  |
| --- | --- | --- |
| Hex Value | Binary Value | Decimal Value |
| 0 | 0000 | 0 |
| 1 | 0001 | 1 |
| 2 | 0010 | 2 |
| 3 | 0011 | 3 |
| 4 | 0100 | 4 |
| 5 | 0101 | 5 |
| 6 | 0110 | 6 |
| 7 | 0111 | 7 |
| 8 | 1000 | 8 |
| 9 | 1001 | 9 |
| A | 1010 | 10 |
| B | 1011 | 11 |
| C | 1100 | 12 |
| D | 1101 | 13 |
| E | 1110 | 14 |
| F | 1111 | 15 |

The table above organizes 4 binary bits into a group, and each 4-bit group has an equivalent hexadecimal letter. Remembering this equivalence is very important: **4-bits of binary equals to 1 hexadecimal letter**.

When you are given a series of binary bits, then you just organize it into multiple 4-bit groups, starting from **right**, and filling bit 0(s) on the **left** if needed, for example:

Binary number: 11011010100 🡪 0110 1101 0100 , where I fill in one bit 0 on the **left** to make every group has four bits. Then I just need to follow the table to convert it into hexadecimal by converting each group into a hexadecimal number, and the result is : 0x 6 D 4.

Each memory location stores one byte of data, which contains 8 bits of binary.

Can you finish the following conversions among decimal, hex, and binary, without using a calculator? Assume all representations are positive integers. Notice that 0x is just the prefix for hexadecimal numbers, and 0x itself does not represent any actual value.

|  |  |  |
| --- | --- | --- |
| Decimal | Binary | Hexadecimal |
| 207 | 1100 1111 | 0xCF |
| 55179 | 1101 0111 1000 1011 | 0x D78B |
| 1133 | 10001101101 | 46D |

What are the possible values a binary bit can hold? How many bits make a byte in the memory?

Binary bit holds 0 or 1. A byte is 8 bits.

Given the fact that : 2 to the power of 10, i.e., 210 = 1024, and it is very close to 1000, which is 10 to the power of 3, i.e., 103, thus we have the table below to explain scale of K, M, G, T in binary and decimal:

|  |  |  |
| --- | --- | --- |
| Binary | Scale | decimal |
| 210 | K , kilo (thousand) | 103 |
| 220 | M, mega (million) | 106 |
| 230 | G, giga (billion) | 109 |
| 240 | T, tera (trillion) | 1012 |

Please have a good understand of the above table. And you are required to answer questions similar to:

What is the scale of 215?

Answer: it is 32K, because 215 = 25 x 210, and notice that 25 is 32, and 210 is K.

Answer these questions regarding 2exponent : what is the scale of 228? 256M

And 236 ?64G And 244 ? 16T

The table below may help you remember the value of 2exponent , with exponent goes from 0 to 10.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| exponent: | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 2exponent | 210 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 |
| value: | 1024 | 512 | 256 | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |

**Java primitive data types**

study textbook Appendix D.1 , which lists 8 different primitive data types, and they are:

**Four** integer types: **byte, short, int, long**

**Two** floating point type or real numbers or fractional numbers: **float, double**

**One** bool type that has the value of either true or false: **boolean**

**One** character type : **char**

for integer types, understand the **size** and **range** of each type

for floating point type, understand their **range** and **precision**, and type conversions between float and double;

for bool type, understand its **value** **range**;

for character type, understand its **\u** representation format, for example, \u0041 stands for character 'A'.

Why is Java type byte having a range from -128 to 127 ?

Answer: a byte has 8 bits, and the highest bit is reserved to represent the sign of the data, with 0 representing non-negative, and 1 representing negative. So, for the non-negative side, the smallest value is 0, with all the remaining 7 bits set to 0; and the highest value is 127, with all the remaining 7 bits set to 1, as indicated below: 26 + 25 + 24 + 23 + 22 + 21 + 20 = 127 . And that is binary value **0111 1111**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Signed bit | 26 | 25 | 24 | 23 | 22 | 21 | 20 |
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Therefore, in the non-negative side, we have 128 values ranging from 0 to 127, then for the negative side, we also must have 128 different values, because for any non-negative value, if you flip the signed-bit(the highest bit) to 1, then you get a negative value. In other words, the total number of non-negative values and the total number of negative values must be the same, since there is a symmetric pair of non-negative value and negative value (the only difference between this pair is the signed bit, with the non-negative value being 0 and its pair in negative side being 1 in the signed bit). Therefore, the negative value must go from -1, -2, -3, and all the way to -128, which concludes a total of 128 negative numbers.

If we look at this problem from another perspective: the total bits in one byte is 8, and only 7 bits can represent the value, thus the data range is from -27 to 27-1, which is -128 to 127.

An alternative view: there are 8 bits in Java type byte, so there are total 28 (which is 256) different values. Because negative values and non-negatives form symmetric pairs, so there are equal numbers of negative values and non-negative values, that means, negative values account for half of the total values which results in 128 negative values, and non-negative values account for the other half, which results in 128 non-negative values. Since negative value stars from -1, so it must count to -128; since non-negative value start from 0, so it must count to 127.

Similarly, why is Java short type having the range from -32K to 32K-1?

Answer: there are 16 bits in Java type short, and only 15 bits can represent value because the highest bit is reserved to represent the sign (negative or non-negative) of the value, thus the data range from -215 to 215-1, which is -32K to 32K-1, and the exact value is -32768 to 32767.

Similarly, why is Java int type having the range from -2billion to 2billion-1 ?

Please provide the answer here. You don’t need to come up with the exact value, and you only need to come up with the correct scale for the range of the negative side and the range of the non-negative side.

Can you explain why the C++ char type has 256 characters, based on the fact that the length of C++ char type is one byte?

1 byte is 8 bits which is 2⁸ which means 256 characters.

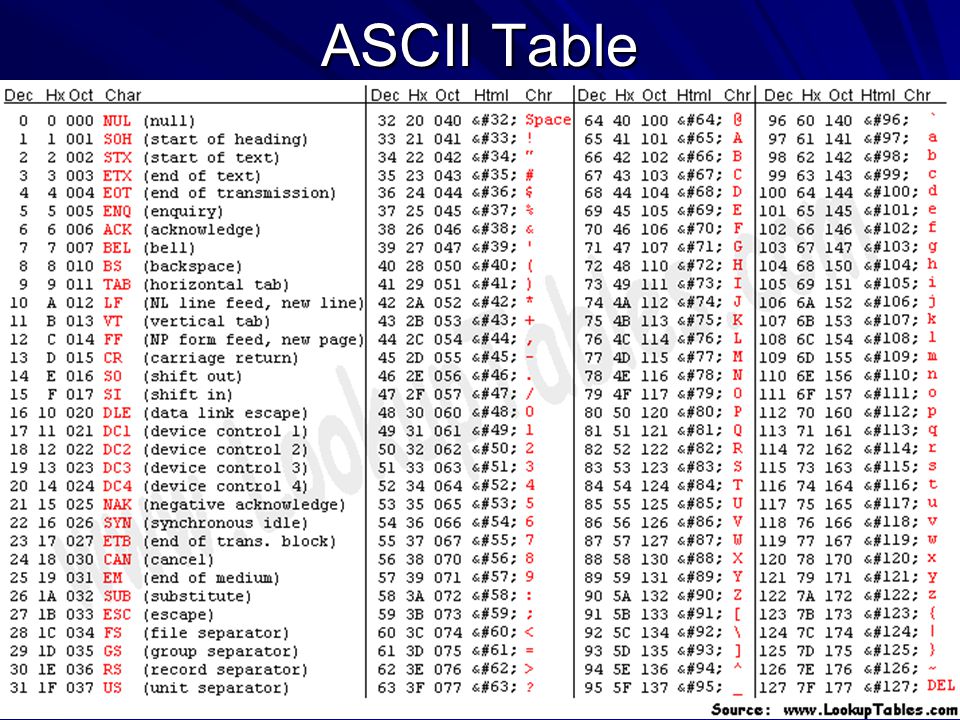
Can you explain why the Java char type has 65536 characters, based on the fact that the length of Java char type is two bytes?

2 bytes is 16 bits which is 2¹⁶ which means 65536 characters.

How is ASCII coding scheme different from Unicode coding scheme? Which scheme does Java adopt? Which scheme supports multilingual?ASCII code has 2⁸ or 256 characters while Unicode has 2¹⁶ or 64k characters. Java adopts Unicode. Unicode is multi-lingual.

When comparing ASCII code with the lower byte of Unicode with, they have the same coding scheme, because Unicode is invented later than ASCII code, and Unicode wants to be backward compatible with ASCII code for the first 256 characters. ASCII code needs only one byte, thus it has 8 bits, which can represent 28= 256 different characters.

Here is a table of the standard ASCII characters set, which contains the first 128 characters in ASCII table with coding value from 0 to 127. For coding value from 128 to 255 (the next 128 character set) is called the extended ASCII characters.



From the above table, you can see that, they divide the 128 standard ASCII characters into four columns, and the visible characters starts from the second column, with the first visible character being the space character, and its coding value is 32 in decimal and 0x20 in hex. For hex number, there is usually a 0x prefix leading the value, which distinguishes hex number from decimal and binary numbers. For each character in the table, we need to be able to read the char itself in red font, and the decimal and hex value representations. For the oct(8-base number) and html coding, we don’t need to worry them in this course.

For the standard ASCII table listed above, you don’t need to remember the exact coding values for each character, but you need to know the following relationships:

* The first column contains the invisible (or unprintable) characters, with coding value from 0 to 31. Coding value from 32 to 126 are visible (printable) characters, and coding value 127 becomes unprintable again (it is the delete key in the keyboard).
* You should notice that, lower case letters follow the alphabet order from character 'a' to character 'z' in the coding scheme of ASCII table, therefore 'a' < 'z', because 97 (coding of 'a') is less than 122 (coding of 'z'). For upper case letters, the same rule applies: 'A' < 'Z'. Also, because lower case letters are coded after the upper case letters in the ASCII table, we have 'a' > 'A', since 97 > 65.
* For numeric letters '0' to '9', they are coded sequentially, therefore '0' < '9'.

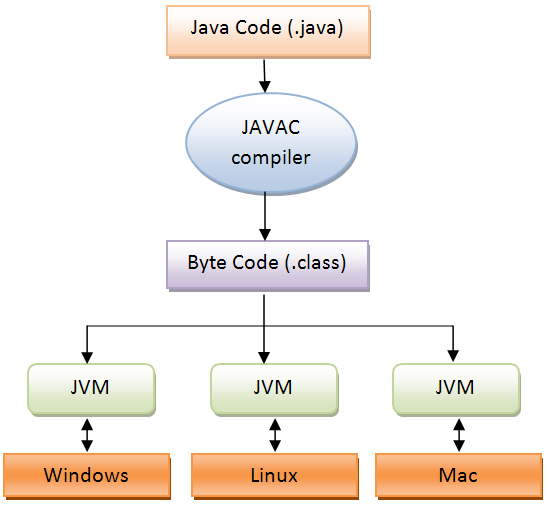
In Unicode, because there are two bytes, they can accommodate 64K different characters, which include the alphabets of all languages in the world, compared to the 256 characters in ASCII, which can only accommodate the English language.

**Java compiler and bytecode file format**

Read the last picture in file “CompileAndRunTheFirstJavaProgram.docx”, and you can see that, a java source code file named HelloWorld.java, needs to be compiled, and the generated file after compilation is named HelloWorld.class.

The .class file has the format called bytecode.

The bytecode format is a platform independent format, which means that the bytecode file can be interpreted into executable code in different operating systems, such as Windows, Linux, and Mac OS.



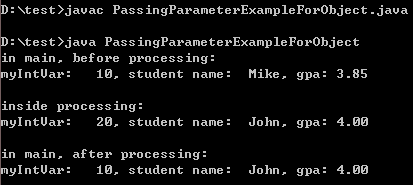
The compiler program in Java is javac.exe.

The interpreter program in Java is java.exe, and it is also called Java Virtual Machine (JVM) or Java Runtime Environment (JRE).

The compiler javac.exe is part of the JDK (Java Development Kit), which is the downloaded installation file for Java, as explained in “InstallingJDK.docx”.

In the picture above, you can see that, the same bytecode format file (the .class file), can be interpreted and run in different operating systems. That is the principle of “write once, run everywhere” in Java. In other words, it is called cross-platform or architecture-neutral.

Take a look at the last picture in file “MemoryManagementForObject.docx”:



Then answer the following question:

Given a Java source code file ComputeTax.java , what is the command we use in command prompt, in order to compile this Java program? To compile: javac ComputeTax.java . After compilation, what is the command we use in command prompt, in order to run this Java program? To run: java ComputeTax

What is the input and output of a Java compiler? Input: .java file. Output: .class file.

What do these abbreviations JDK, JRE, and JVM stand for?JDK..Java Development Kit.

JRE..Java Runtime Environment. JVM..Java Virtual Machine.

Is bytecode the format for .java source code file, or for .class file? .class file

Hint: Java is cross-platform, or architecture-neutral, because bytecode is designed to be machine-independent, so that bytecode can run on different operating systems and different CPUs in the java virtual machine, which means, in other words, cross-platform or architecture-neutral.

In a java source code file Welcome.java, we have the following code:

public class TryItOut{

public void Main(){

System.out.println(Try it out);

}

}

public class Welcome{

public static void main(String[] args){

System.out.println(“Try it out”);

}

}

So what is wrong with the above code? Please modify it to make it can compile and run, and assume you cannot change the name of the file “Welcome.java”.

Hint: the file name has to be the same as the public class name in a java source code.

**Conditional statement using “if --- else” or “if --- else if --- else” statement**

A standard “if --- else” statement example is in textbook page 68.

A standard “if --- else if --- else” statement example is as below. What is wrong with the code below? How to you correct the mistake? (Hint: it is a logical error).

if( Greenfoot.getMicLevel() > 5)

{

move(3);

}

else if (Greenfoot.getMicLevel() > 10)

{

move(7);

}

else

{

move(-5);

}

This program will never run else if (Greenfoot.getMicLevel() > 10) because every time the processor finds the condition if( Greenfoot.getMicLevel() > 5) , it executes it and exits. The correction will be

if( Greenfoot.getMicLevel() > 5 && Greenfoot.getMicLevel() <=10)

{

move(3);

}

else if (Greenfoot.getMicLevel() > 10)

{

move(7);

}

else

{

move(-5);

}

if( Greenfoot.getMicLevel() >10

{

move(7);

}

else if (Greenfoot.getMicLevel()<3)

{

move(3);

}

else

{

move(-5);

}

The code indentation in the above multiple branch “if-else if” is correct. The error is not in the code indentation.

|  |  |  |
| --- | --- | --- |
| Write an **if** statement that prints out **student** or **non-student** based on a boolean variable **isStudent**  Hint: you can use System.out.println () method for outputting the result.  **boolean** isStudent = **true**;  **if**(isStudent){  System.***out***.println("student");  }  **else** {  System.***out***.println("non-student");  } | | |
| Write an **if** statement that tests a variable that prints out “**give a raise”** if the variable **workerEvaluation** (which can hold 1-5) is greater than 3 **AND** if the variable **yearsWithCompany** is greater than 2. Use logical operator && in the boolean expression.  **if**(workEvaluation >3 && yearsWithCompany>2){  System.***out***.println("give a raise");  } | | |
| Write a multiple **if** statement that tests a variable that prints out **give a raise** if the variable **workerEvaluation** (which can hold 1-5) is greater than 3 **OR** if the variable **yearsWithCompany** is greater than 2. Use logical operator || in the boolean expression.  **if**(workEvaluation >3 || yearsWithCompany>2){  System.***out***.println("give a raise");  } | | |
| Write a multiple-branch “**if -- else if ”** statement that tests the same variable **average** and assigns (A,B,C,D or F) to char type variable **grade**, based on average. (>=90, >=80, >=70, >=60, or < 60)  **public** **class** HelloWorld {  **public** **static** **void** main(String[] args) {  *method2*(98);  // end of method 1  }  **public** **static** **void** method2(**double** average) {  **if** (average >= 90) {  System.***out***.println("grade A");  }  **else** **if** (average >= 80) {  System.***out***.println("grade B");  } **else** **if** (average >= 70) {  System.***out***.println("grade C");  } **else** **if** (average >= 60) {  System.***out***.println("grade D");  } **else** {  System.***out***.println("grade F");  }  }  } | | |
| Correct the code on the left column, and provide the correct one on the right column. | | |
| **boolean x = 1;** | boolean x =true; | |
| **double y == 53.4l** | double y = 53.41; | |
| **if( x > 5 )**  **y ++;**  **z++;**  **else**  **y--;** | if(x>5)  {  y++;  z++;  }  else  {  y--;  } | |
| **if y < 3**  **x =+ z; // add z to x**  **else**  **x -= z; // subtract z from x** | if(y<3)  {  x+=z;  }  else  {  x -=z;  } | |
| **if(x=y) {**  **System.out.println(They are equal!);**  **}** | **if(x==y) {**  **System.out.println(“They are equal!”);**  **}** | |
| Decide whether the following “if statement” is true or false | | |
| **int x=5,y=6,z=10;**    **if(x<y && y>z)**  **false** | | **int x=5,y=6,z=10;**    **if(x<y || y>z)**  **true** |
| **int x=5,y=6,z=3;**  **if(x<y && y>z)**  true | | **int x=5,y=2,z=10;**  **if(x<y || y>z)**  **false** |
| **int x=5,y=6,z=3;**  **if(x!=y)**  **true** | | **int x=5,y=6,z=3;**    **if(!(y > z && x > y))**  true |

What is the printout of the following statement? If the statement cannot compile, please indicate.

System.out.println('a' == 'a'); true

System.out.println('a' >= 'A'); true

System.out.println('A' >= 'a'); false

System.out.println('a' != 'b'); true

System.out.println('a' == "a"); cannot compile

What are the printout results of the code below in the left column and in the right column, respectively? Assume the variable ***number*** is 8. What are the printout results if the variable ***number*** is 23 ?

|  |  |
| --- | --- |
| if (***number*** %2 == 0){  System.out.println(***number*** + " is even ");  }  System.out.println(***number*** + " is odd ");  For variable number = 8  8 is even  8 is odd  For variable number = 23  23 is odd | if (***number*** %2 == 0){  System.out.println(***number*** + " is even ");  } else {  System.out.println(***number*** + " is odd ");  }  For variable number = 8  8 is even  For variable number = 23  23 is odd |

Write an if statement that increases variable pay by 3% if score is greater than 90.

if(score >90){

double pay+=pay\*3\*0.01;

}

Write an if statement that increases variable pay by 3% if score is greater than 90, otherwise increases pay by 1%.

if(score >90)

{

pay+=pay\*3\*0.01;

}

else

{

pay+=pay\*1\*0.01;

}

Search the random method definition in online java class Math documentation:

[https://docs.oracle.com/javase/13/docs/api/java/lang/Math.html](https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html)

, then answer this question: which of the following is a possible output from invoking Math.random()?

1.25 3.04 5 0.06 19.2 99.5 0.28 1 0.99 100

Possible outputs: 0.06, 0.28, 0.99

Given two integers a and b, assuming a < b, and the formula to generate a random integer r between a and b (both a and b are inclusive) is as below. The equivalent boolean expression is: r>=a && r <=b

int r = a + (int)( Math.random() \* (b – a + 1) ) , can you explain why? The formula r = a + (int)( Math.random() \* (b – a + 1) generates random numbers between a and b inclusive which can also be expressed using the expression r>=a&&r<=b which also is a way of writing that the value of r will start from a inclusive, include all the values in between a and b , plus the value b itself is inclusive.

Using java Math.random() method, how to generate a random integer between 1 and 99? And Between 1 and 100? And between 0 and 99? And between 0 and 100? All numbers are inclusive.

import java. util. Random

Generate 0 to 99 -- double random = Math.random() \* 99

Generate 1 to 100 -- double random = Math.random() \* 99 + 1;

Another method is ***Grennfoot.getRandomNumber(int limit)*** , which generates an int type variable greater or equal to 0 and less than parameter ***limit***, what is the range of this expression below?

***10 + 7 \* Grennfoot.getRandomNumber(6)***

in Greenfoot java source code, if an import statement is like this:

***import greenfoot.Actor;***

, then the lowercase **greenfoot** is the package name, and the uppercase **Actor** is the class name.

Similarly, in Eclipse java project, if an import statement is like this:

***import java.util.Scanner;***

, then ***java.util*** is the package name starting with lowercase letters, and ***Scanner*** is the class name starting with uppercase letter.

**Code indentation and alignment**

|  |  |
| --- | --- |
| Provide in the right column with corrected indention and alignment for the code in the left column. Now can you see how important it is to OBSERVE the code indentation and alignment rule!!!!! | |
| import java.util.Scanner;  public class IncorrectIndentIsUnacceptable{  public static void  main(String[] args)  {int hours = 0;  double salary =0.0,rate=5.00;  Scanner input = new Scanner(System.in);  System.out.print("Enter hours worked(0-40): ");  hours = input.nextInt();  while(hours<0 || hours>40){  System.out.print("Re-enter hours worked(0-40): ");  hours = input.nextInt();  }salary = hours \* rate;  System.out.printf("Salary: $%5.2f\n",salary);  }} | **import** java.util.Scanner;  **public** **class** IncorrectIndentIsUnacceptable {  **public** **static** **void** main(String[] args) {  **int** hours = 0;  **double** salary = 0.0, rate = 5.00;  Scanner input = **new** Scanner(System.***in***);  System.***out***.print("Enter hours worked(0-40): ");  hours = input.nextInt();  **while** (hours < 0 || hours > 40) {  System.***out***.print("Re-enter hours worked(0-40): “);  hours = input.nextInt();  }  salary = hours \* rate;  System.***out***.printf("Salary: $%5.2f\n", salary);  }  } |

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

Please review the materials in this section. There isn’t any question for you to answer in this section.

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

* **data abstraction**
* **encapsulation**
* **inheritance**
* **polymorphism** with the following context:
* **method overloading**
* **method overriding**
* **parent reference refers to child object**
* **dynamic binding**
* **static binding** (generic programming)

We cover **“data abstraction” and “encapsulation”** in the PowerPoint slide “OOP-BasicIntro.ppt”.

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.

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

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" \o "class in greenfoot)** 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 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 loster, 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).

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.

**Stack and method invocation**

Read document “MemoryManagementForObject.docx”, and study the three accompany example java source code. Understand the operation of stack, and how method invocation works, and pass primitive data vs passing object variable in method calling.

When power is off, does the content in the memory device RAM still being kept?

The RAM does not store data when the power is off.

Is the content is the storage device such as a hard disk still being kept when power if off?

Hard disk stores data when power is off and when power is on.

Read document “MemoryManagementForObject.docx”, and explain what are the functionalities for memory address and memory content, respectively?

The processor reads data from the memory address .The memory address is data storage location which has a unique address.

Hint: given an analogy of a hotel compared to memory, the hotel room number can be compared to memory address, and the guest staying in the hotel room can be compared to the memory content.

Stack has the property of First In Last Out (FILO), or Last In First Out (LIFO). For example, in file “PassingParameterExampleForObject.java”, the main method is the first to be pushed into the stack, and the last to be pop out of the stack, while the process method is the last to be pushed into the stack, and the first to be pop out of the stack. In other words, main method runs first and finish last, while the process method (called by main) runs second but finish first (finish earlier than main).

Draw a simple graph showing the methods in stack for each of the check point in the table below, when a calling sequences is given as: main calls m1 then calls m2; m1 calls m3; m2 and m3 does not call any other method. You don’t need to worry about the grammar syntax of each method, since these methods are not necessarily written in Java.

Check point 1: after main is executing, before main calls m1;

|  |
| --- |
| stack  stack top |
|  |
|  |
| main  stack bottom |

Check point 2: after main calls m1, m1 not calling m3 yet;

|  |
| --- |
| stack  stack top |
|  |
| m1 |
| main  stack bottom |

Check point 3: after m1 calls m3, m3 not finished yet;

|  |
| --- |
| stack  stack top |
| m3 |
| m1 |
| main  stack bottom |

Check point 4: after m1 calls m3, m3 already finished;

|  |
| --- |
| stack  stack top |
|  |
| m1 |
| main  stack bottom |

Check point 5: after main finishes calling m1, before main calls m2;

|  |
| --- |
| stack  stack top |
|  |
|  |
| main  stack bottom |

Check point 6: after main calls m2, m2 not finished yet.

|  |
| --- |
| stack  stack top |
|  |
| M2 |
| main  stack bottom |

Check point 7: after main calls m2, m2 already finished.

|  |
| --- |
| stack  stack top |
|  |
|  |
| main  stack bottom |

Check point 8: after main itself finishes execution.

|  |
| --- |
| stack  stack top |
|  |
|  |
| stack bottom |

|  |  |  |  |
| --- | --- | --- | --- |
| void main(){  ….  m1();  m2();  …..  } | void m1(){  …..  m3()  …..  } | void m2(){  ……  } | void m3(){  ……  } |

The … sign in the above table indicate “here goes some code”. The parameters of each method are omitted to save space, and the access modifier are also omitted in each method. Assume the above method invocations are valid. The syntax of the above methods are not necessarily in Java.

A sample graph for **check point 3** above will be as below:

|  |
| --- |
| stack  stack top |
| m3 |
| m1 |
| main  stack bottom |

A sample graph for **check point 4** above will be as below: (the stack of m3 has been pop out)

|  |
| --- |
| stack  stack top |
| m1 |
| main  stack bottom |

Now you need to draw of the stack diagram for the rest of the scenarios. You can start with copy and paste the drawings above, and then modify accordingly. Similar questions will appear in the exam.

**Method definition and invocation**

|  |  |
| --- | --- |
| Implement a method that is passed two double type numbers, prints out their total and doesn’t return anything  **public** **class** Test {  **public** **static** **void** main(String[] args) {  *method2*(3.5, 4.1);    }  **public** **static** **void** method2(double n, double m) {  double c = n + m;  System.***out***.println(c);  }  } | |
| Write a statement to invoke the method described above  *method2*(3, 4); | |
| Write a method that isn’t passed anything, asks the user to input two integer numbers, adds them and returns the total.  Hint: search the online java document for class Scanner, and learn how to use the nextInt for integer input in Scanner class. For double type input, you need to use nextDouble method.  **import** java.util.Scanner;  **public** **class** HelloWorld {  **public** **static** **void** main(String[] args) {  *method2*();  // end of method 1  }  **public** **static** **void** method2() {  Scanner input = **new** Scanner(System.***in***);  System.***out***.print("Enter first integer :");  **int** firstInteger = input.nextInt();  System.***out***.print("Enter second integer :");  **int** secondInteger = input.nextInt();  System.***out***.println(firstInteger + secondInteger);  }  } | |
| Write a statement to invoke the method described above  method2(); | |
| What is the definition for “method signature”?  This includes the method name and the data type for example  calSum(int,int)  Does the method signature include return type?No | |
| Assume a method accepts an int variable named age as formal parameter, and this method only accept age value that is between 20 (inclusive) and 30(exclusive). Write a range checking code block to prevent the method from accepting an out-of-range value for variable age, and if the range checking fails, this method should return immediately. Only write down this piece of range checking code.  age <= 20 || age >= 30 | |
| **public void calcSalary(hours, payRate)**  **{**  **int sal;**  **sal = hours \* payRate;**  **return sal;**  **}** | **public int calcSalary(int hours, int payRate)**  **{**  **int sal;**  **sal = hours \* payRate;**  **return sal;**  **}** |

In java file “PassingParameterExampleForObject.java”, when invoking method process(),

the actual parameters are: myIntVar , studentA

the formal parameters are: myInt, oneStudent

the pass-by-vale principle in Java means: the actual parameter copies its value to its corresponding formal parameter, and this is a one-way value passing from actual to formal parameter.

the method header of method ***process*** is:

public static void process(int myInt, Student oneStudent)

Now please look at sample source code “TestSum.java” listed below:

public class TestSum {

public static void main(String[] args){

int a = 10;

int b = 20;

int total = calcSum(a, b);

System.out.println("the total is " + total);

}

public static int calcSum(int x, int y){

int result;

result = x + y;

return result;

}

}

, then answer the questions below :

What are the names of the formal parameters for method calcSum? int x and int y

What are the names of the actual parameters for method calcSum? int a and int b

Do you find formal parameters in method definition or in method invocation? Method defination

Do you find actual parameters in method definition or in method invocation? Method invocation

What is the method signature of method calcSum? calcSum(int, int)

What is the method header of method calcSum? public static int calcSum(int x, int y)

Does the method signature include the return type? No

Does the method header include the return type? Yes

If the method signature in the invocation does not match the method signature in the definition, will the program compile? No

Can the formal parameters have the same name as the actual parameters? Yes, but it is not a good coding practice.

In the above code, what is the first method pushed into the stack, main method or calcSum method? Main method

In the above code, what is the first method pop out of the stack, main method or calcSum method?calcSum

The questions above are related to class TestSum as listed above.

In java method invocation, are the values passed from actual parameters to formal parameters, or from formal parameters to actual parameters? Values are passed from actual parameters to formal parameters.

When the value of primitive type variables is passed, will the change of the formal parameters affect the actual parameters? Hint: the value passing from actual parameter to formal parameter is one-way passing.

No

Evaluate the following methods calls:

|  |  |  |
| --- | --- | --- |
| Math.sqrt(49) | Math.pow(2, 5) | Math.max(Math.min(4, 6), 5) |
| 7 | 32 | 5 |

For the table below, identify the program errors on the left column, then correct the code on the right:

|  |  |
| --- | --- |
| public class Test{  public static method1(int n , m){  n += m;  method2(3.4);  } // end of method 1  public static method2(int n){  if(n > 0){  return 1;  }  else if(n == 0){  return 0;  }  else if(n < 0){  return -1;  }  } // end of method 2  } // end of class | **public** **class** Test {  **public** **static** **void** main(String[] args) {  *method2*(3);  // end of method 1  }  **public** **static** **int** method2(**int** n) {  **if** (n > 0) {  **return** 1;  } **else** **if** (n == 0) {  **return** 0;  } **else** **if** (n < 0) {  **return** -1;  }  **return** n;  } // end of method 2  } // end of class |
| public class TestPrint{  public static void main (String[] args){  nPrintln(5, "welcome to Java!");  } // end of main  public static nPrintln (String msg, int n){  for(int i=0; i<n; i++){  System.out.println(msg);  }  } // end of nPrintln  } // end of class | **public** **class** TestPrint {  **public** **static** **void** main(String[] args) {  *nPrintln*(5, "welcome to Java!");  } // end of main  **public** **static** **void** nPrintln(**int** n, String msg) {  **for** (**int** i = 0; i < n; i++) {  System.***out***.println(msg);  }  } // end of nPrintln  } // end of class |

**Object creation and method invocation**

Understand the syntax of creating a new object. It is also called “initialization” of an object.

**ClassName objectName = new ConstructorName( actual parameter list …);**

For example :

Student studentA = new Student("Mike", 3.85);

Lobster myLobster = new Lobster();

Constructor is a special method of a class, because:

* constructor is only used to construct (in other words, build, or create) objects (instances) of a class because of the above reason, constructor is always associated with Java keyword ***new***
* constructor does not have any return type, not even ***void***. If you ever put Java keyword ***void*** in front of a constructor, that is wrong
* constructor must have the same name of the class, thus constructor is the only type of method that starts with upper case letter in the method name. In comparison, method name, object name, variable name, and package name must start with lower case letter

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

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

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

Remember, we do NOT have this pattern below, because private data member cannot be accessed directly without going through the public method. Again, it is because of the OOP encapsulation principle!

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

Read the following table, and in the first row, are the various terms we use to describe data in a class, and in the second row, are the terms for method in a class. Each term has its own perspectives in describing the encapsulation principle, and hopefully this table will give you a full spectrum of the meaning of encapsulation principle.

|  |  |
| --- | --- |
| data | attributes, fields, states, properties, characteristics |
| methods | functions, operations, handlers, contracts, bridges, behaviors, manipulators, actions |

Why in the above table, bridge is used to describe method? Because method connects the private data on one hand, and method itself is public so it connects user on the other hand. In other words, user can manipulate the private data through the public method. Public methods serve as bridges via which user can reach private data indirectly.

Another picture I want you to remember about encapsulation is as below:

Encapsulating layer:

public methods

Encapsulated content:

private data

The user of an encapsulated class can only access its public methods directly

In the above graph, you can see that the user of the class can only access the shell of the class, i.e., the public methods of the class; and the public method serve as the encapsulating layer (protecting layer) of the private data (the data that have been encapsulated by the public methods). Because the data are encapsulated, the user of the class cannot touch the data directly, and the user can only user the public methods as bridges, so that it can access the data indirectly.

Therefore, the benefit of encapsulation includes:

* it protects the integrity of the data
* user can only modify the data through the method members of the class
* user doesn’t need to worry about the implementation details of the class method members
* user cannot modify the private data without accessing the public method of the class

Study file “**MemoryManagementForObject.docx**”, and we summarize the location of different types of data/variables in memory:

* non-static data member of a class (must be private to limit access): locates in heap, inside an objects’ body
* formal parameter of a method: locates in stack of this method
* local variable of a method: locates in stack of this method

Refer to Figure 3 in document “MemoryManagementForObject.docx”. Draw a similar picture for this statement:

Student applicant = new Student ("Brad Smith", 3.5);

|  |
| --- |
|  |
| Name: “Brad Smith”  Gpa:3.5 |
|  |
| 0x465TG568BJK |
| Main |

Follow the picture you draw above, answer the questions below:

Where does the student variable *applicant* locate in the memory, stack or heap? Stack

Where does the student data such as name and gpa locate in the memory, stack or heap? Heap

When modifications are made in method process to change the values of the primitive type formal parameter and the class type formal parameter, which modification will carry back to the main method?

Modification to class type formal parameter will carry back.

Explain the difference between these two assignment statements:

a = b; // assume a, b are both int type variables, and they were initialized earlier.

studentA = studentB; // assume they are objects of Student type, and they were initialized earlier.

Hint: you can refer to page 56 and 57 of textbook.

a=b; assigns a value while studentA = studentB; assigns a reference to an object.

Read the PowerPoint slides “OOP-BasicIntro.ppt”, and understand the analogy below:

private data of class is compared to the magic stuff under the hood of a car

**public** method of class is compared to all handlers that you can manipulate when you sit on the driver’s seat, such as gear, steering wheel, brake, gas pedal, etc.

Refer to class Asteroid.java in greenfoot project “asteroids1” in chapter 1, when working on the following two questions:

Using one line of statement, create an instance of class Asteroid named ***as1*** instantiate with the default constructor that has no parameter.

Asteroid as1 = Asteroid();

Using one line of statement, create an instance of class Asteroid named ***as2*** instantiated with the constructor that has one parameter. Plug in a numeric value of suitable type as the parameter.

Asteroid as2 = Asteroid(7);

The two usages of java keyword ***this***:

* to invoke an overloaded constructor in the same class.
* server as the self-reference to the current object, and it can be used to refer to an instance data in the heap, or to invoke an instance method

The two usages of java keyword ***super***:

* to invoke a constructor in the superclass class.
* to invoke an **overridden** method from the superclass.

In class Asteroid.java, we can see the two usages of this, and the first usage of super:

public Asteroid()

{

this(64);

}

/\*\*

\* Create an asteroid with a given size, random movement direction and

\* default speed.

\*/

public Asteroid(int size)

{

this(size, new Vector(Greenfoot.getRandomNumber(360), 2));

}

/\*\*

\* Create an asteroid with a given size size, direction, and speed.

\*/

private Asteroid(int size, Vector speed)

{

super(speed);

this**.**setSize(size);

}

/\*\*

\* Set the size of this asteroid. Note that stability is directly

\* related to size. Smaller asteroids are less stable.

\*/

public void setSize(int size)

{

this**.**size = size;

stability = size;

GreenfootImage image = getImage();

image.scale(size, size);

}

For the second usage of keyword super can be found in this link:

<https://docs.oracle.com/javase/tutorial/java/IandI/super.html>

where the statement super.printMethod(); invoked inside subclass’s printMethod() actually calls the overridden method in its superclass.

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;
* Names that start with upper case letter ( UpperCamelCase ): class/interface name, constructor name.
* Names that have to be all upper-case letters, with underscore connecting adjacent words: static final variables, in other words, constant variables.

Here is a list of zip files you need to study:

|  |  |
| --- | --- |
| Moodle folder | Zip file name |
| Chap 1 | No zip file |
| Chap 2 | OOPIntroPPT-JeepSourceCode.zip |
| Chap 3 | MemoryManagementForObject.zip |
| Chap 4 | UML-classDiagram.zip  bonusHomework-CircleClass.zip |
| Test 1 review Lesson | No zip file |

Test 1 has two coding questions that require you to write the complete code for classes. These two coding questions in test 1 are similar to the two coding questions below.

**Coding question #1:**

Code a Java class named Test, and in the main method of this class, you need to get from the console input (the Scanner class) for the number of students and the number of classes, then you need to invoke a method to obtain the average number of students per class. Then print out the result to the screen.

Still inside class Test, implement a method that is to be invoked by the main method. This method will be passed the number of students and the number of classes, and then it will calculate and return the average number of students per class. Use suitable names for the method and its formal parameters.

Code it in Eclipse and debug and run it to make sure it is correct. Then paste the source code here. Remember, you need to provide one class with two methods in this class, but nothing more than that. In the exam, there will be a very similar question like this one. Refer to file ***HouseApplicationSample.java***, which can be found after you unzip file “UML-classDiagram.zip” in Moodle folder “chap 4”.

|  |
| --- |
| **import** java.util.Scanner;  **public** **class** Test {  **public** **static** **void** main(String [] args) {    Scanner input = **new** Scanner(System.***in***);    System.***out***.print("Enter the total number of students: ");  **int** numberOfStudents = input.nextInt();    System.***out***.print("Enter the total number of students: ");  **int** numberOfClasses = input.nextInt();    System.***out***.printf("Class average is :%d",*calcClassAverage*(numberOfStudents, numberOfClasses));    }    **public** **static** **int** calcClassAverage(**int** students, **int** classesCount) {    **return** students/classesCount;  }  } |

**Coding question #2:**

The second coding question is a similar coding requirement as homework 4. Refer to step 6 in file “chap4-schedule.docx”, which has detail explanations of encapsulation principle of OOP.

|  |
| --- |
| Use data abstraction (real-world modeling) to come up with 3 instance variables and 2 effectors for a new data type – the class **HospitalPatient.** You have the freedom to design the instance variables and effectors.  instance variables   1. **private** **double** dailyBedCharge;. 2. **private** **int** numberOfDaysAdmitted;.; 3. **private** **double** costOfMedicine;   effectors  1 **public** **void** setCostOfMedicine(**double** costOfMedicine)  2 **public** **double** calcDiscountedPrice(**double** copayDiscount) |
|  |

|  |
| --- |
| **Write the code** for class **HospitalPatient**, according to the requirement above. Include the default constructors with no argument, and the constructor with all arguments. Provide a getter and setter for each individual instance variable. Provide the two effectors that you design above. All instance data are private and all methods are public. Observe the coding conventions, including proper indent. No user input is needed.  **public** **class** HospitalPatient {    **private** **double** dailyBedCharge;  **private** **int** numberOfDaysAdmitted;  **private** **double** costOfMedicine;    **public** HospitalPatient() {  **this**(34.62, 6, 121.08);  }    **public** HospitalPatient(**double** dailyBedCharge, **int** numberOfDaysAdmitted, **double** costOfMedicine) {    **this**.dailyBedCharge = dailyBedCharge;  **this**.numberOfDaysAdmitted = numberOfDaysAdmitted;  **this**.costOfMedicine =costOfMedicine;  }    **public** **double** getDailyBedCharge() {  **return** dailyBedCharge;  }    **public** **int** getNumberOfDaysAdmitted() {  **return** numberOfDaysAdmitted;  }    **public** **double** getCostOfMedicine() {  **return** costOfMedicine;  }      **public** **void** setDailyBedCharge(**double** dailyBedCharge) {  **this**.dailyBedCharge = dailyBedCharge;  }    **public** **void** setNumberOfDaysAdmitted(**int** numberOfDaysAdmitted) {  **this**.numberOfDaysAdmitted = numberOfDaysAdmitted;  }    **public** **void** setCostOfMedicine(**double** costOfMedicine) {  **this**.costOfMedicine = costOfMedicine;  }    **public** **double** calcTotalCost() {  **double** totalCost = (dailyBedCharge \* numberOfDaysAdmitted) + costOfMedicine;  **return** totalCost;  }    **public** **double** calcDiscountedPrice(**double** copayDiscount) {  **double** discountedPrice = ((dailyBedCharge \* numberOfDaysAdmitted) + costOfMedicine)  - (((dailyBedCharge \* numberOfDaysAdmitted) + costOfMedicine) \*  (copayDiscount \* 0.01));  **return** discountedPrice;  }      } |
| **Write the code** for an application class that has a main method, and in the main method, creates two instances of class **HospitalPatient**, one with default constructor and one with the other constructor. Then ask user to input necessary variables, and then use the setters to set the input variables into the object that you created with default constructor. Then call the effectors of the **HospitalPatient** and prints out some information about the patients. You can name the class JohnDoeTest, where JohnDoe should be replaced with your name. Observe coding conventions, including proper indent. You can use Scanner class here to get input parameters from user, in order to initialize the object created by the default constructor (the constructor that has no parameter). Only in this application class, you need to obtain user input.  **import** java.util.Scanner;  **public** **class** GeorgeGichukiTest {  **public** **static** **void** main(String[] args) {  Scanner input = **new** Scanner(System.***in***);  **double** dailyRate =45.90;  **int** stayedDays = 10;  **double** drugCharges = 12.69;  HospitalPatient babuHospitalPatient = **new** HospitalPatient(dailyRate, stayedDays,drugCharges);    HospitalPatient maraHospitalPatient = **new** HospitalPatient();    System.***out***.print("Enter the daily hospital bed fees :");  **double** bedFees = input.nextDouble();    System.***out***.print("Enter the number of days stayed at the hospital :");  **int** hospitalStay = input.nextInt();    System.***out***.print("Enter the total drug charges: ");  **double** medCharge = input.nextDouble();    maraHospitalPatient.setDailyBedCharge(bedFees);  maraHospitalPatient.setNumberOfDaysAdmitted(hospitalStay);  maraHospitalPatient.setCostOfMedicine(medCharge);    System.***out***.print("Enter the copay insurance discount: ");    **double** insuranceDisc = input.nextDouble();    **double** priceTotalB = babuHospitalPatient.calcTotalCost();  **double** discPriceB = babuHospitalPatient.calcDiscountedPrice(insuranceDisc);  **double** priceTotalM = maraHospitalPatient.calcTotalCost();  **double** discPriceM = maraHospitalPatient.calcDiscountedPrice(insuranceDisc);    System.***out***.printf("\nTotal bill of the first patient before "  + "discount%7.2f\n", priceTotalB);    System.***out***.printf("Insurance discount%6.2f%%\nTotal to pay "  + "after discount%7.2f\n", insuranceDisc, discPriceB);    System.***out***.printf("\nTotal bill of the second patient before "  + "discount%7.2f\n", priceTotalM);    System.***out***.printf("Insurance discount%6.2f%%\nTotal to pay "  + "after discount%7.2f\n", insuranceDisc, discPriceM);    }  } |