**Chapter 2**

Ex. 2.1 –Done

Ex 2.2 - After printing a ticket the machine balance resets to zero

Ex 2.3 - The machine doesn’t seem to track anything; it lets you insert money and prints a ticket when told to. It doesn’t hold onto balances after printing a ticket and doesn’t give change. It also accepts less than the cost of a ticket.

Ex. 2.4 – Done

Ex. 2.5 – The only real difference between the two is the the listed price on the ticket, one says 500 cents the other 200. Besides that they are identical/

Ex. 2.6 – The outer wrappers for “Students” and “LabClass” probable look like

public class Student and public class LabClass both according to the book will have a brief explanation to what each class does and represents.

Ex. 2.7 – Yes the order does matter, public class “ClassName” is correct and class public “ClassName” won’t compile listed error is “<identifier> expected” the class icon in the BlueJ change from a single color to having a crosshatch pattern over it.

Ex. 2.8 – The class will compile without the word public in front of it.

Ex. 2.9 – It doesn’t work without class, displays “class, interface, or enum expected”.

Ex. 2.10 – the fields of the naïve ticket machine are all private integers named “pice, balance, and total”. The constructor in this code is public TicketMachine (int cost). Finally, it has the methods getPrice, getBalance, insertMoney, and printTicket.

Ex. 2.11 – Two differences between the class constructor and the methods are that it starts with a capital letter and it lacks an return type.

Ex. 2.12 – the first one is of type int, the second and third are likely strings.

Ex. 2.13– the names of the following fields are alive, tutor, and game.

Ex. 2.14– I believe the class names in order from exercises 2.12 and 2.13 are Student, Server, Person, and Game.

Ex. 2.15– Done

Ex. 2.16 – A semicolon is always needed at the end of a field declaration.

Ex. 2.17 – private int status;

Ex. 2.18 – the constructor belongs to class Student

Ex. 2.19– The constructor has two formal parameter, string and double. The actual parameters are title (string) and price (double).

Ex. 2.20– Some possible types contained within the class Book’s fields are string, int, and double. Some possible names for those fields are author name, ISBN number, price, number of pages, title, possibly even genre.

Ex. 2.21– public Pet (string petsName)

{

name = petsName ;

}

Ex. 2.22– public Calender (string “month”, int day, int year”)

{

month = “March”;

day = 23;

year = 1861;

}

Ex. 2.23– The structure is near identical between the getPrice and getBalance methods. Apart from name their headers are identical and the only difference in the body is what fields are returned.

Ex. 2.24– the “getBalance” method could be characterized as “How much have I put in the machine?”.

Ex. 2.25– If we change the method name we do NOT change the return statement. If we were to change the return statement to something the method won’t function unless the variable balance was changed to match.

Ex. 2.26– Done

Ex. 2.27– The message given says statement is missing and doesn’t compile.

Ex. 2.28– The biggest difference is the return type. Price returns an int where as printTicket has void meaning it has no return type.

Ex. 2.29– No, neither of these methods have a return type. They don’t have one because they are meant to change and alter things. The void statement in their headers gives away that they don’t need return statements.

Ex. 2.30– Done

Ex. 2.31– We know the method is header is a method header and not for a constructor because it has statement between public and the method name.

Ex. 2.32– Done and saved

Ex. 2.33– I am assuming the field “score” starts at 0

public void increase( int points)

{

score = score + points;

}

Ex. 2.34– Yes the increase method is a mutator. It is a mutator because

Ex. 2.35–

public void discount (int amount)

{

price = cost - amount;

}

Ex. 2.36 – the string will be printed to the screen and will be displayed as My cat has green eyes.

Ex. 2.37 – Done and saved.

Ex. 2.38 – It would no longer print the variable value price but the string “price”.

Ex. 2.39 – it doesn’t compile because it of break points.

Ex. 2.40 – Only if the other ticket machines had their code written differently.

Ex. 2.41 – Done and saved.

Ex. 2.42 – When the method is called on the 1st ticket machine it displays the cost of that machines ticket and the same is true for the second. They are different because both machines are separate instances of the class TicketMachine and both have same methods and variables but the associated values are different so we get the two different prices displayed. Like the example of class Car and how different instances of it could be red or blue. But both are still of the class Car.

Ex. 2.43 – When we create a TicketMachine only asks us to name the ticket machine now and the get price displays the fixed value 1000.

Ex. 2.44 – Done and Saved

Ex. 2.45 – The method “empty” doesn’t need any parameters because it is changing a preexisting one. This method is also a mutator method because it takes one of the stored values and changes it.

Ex. 2.46 – The balance remains the same when an error occurs. When I enter zero the program should display the same message asking for a positive amount rather than: 0 and when I tested this it did. This is because it is written amount is greater than 0 not >= to. Making it so zero gets the error message.

Ex. 2.47 – If I change the test to >=0 then 0 will be an acceptable value to enter however if entered it would add 0 to the current balance and the balance wouldn’t change.

Ex. 2.48 – Done and saved.

Ex. 2.49 – A Boolean field controlled the visibility of objects in the figures project. This method was perfect for visibility because there are only two possibilities. The shape is visible (true) then the shape is drawn or it’s not (false) and the circle is erased.

Ex. 2.50 – the differences in the print ticket is that for the total collected in the improved machine only adds the price of one ticket and subtracts that from the balance instead of taking the entire balance into the total. In short when a ticket is printed it only takes and out the price of one ticket and makes sure everything else reflects that.

Ex. 2.51 – If the else portion of the code is removed tickets still can’t be printed if the balance isn’t greater than the ticket price. However, an error message will not be printed.

Ex. 2.52 – No with the improved ticket machine it is impossible to print a ticket if you haven’t put at least the price of the ticket. So as long as the test statements aren’t altered it will be impossible to get a negative balance in this machine.

Ex. 2.53 – Done

Ex. 2.54 – saving = price \* discount;

Ex. 2.55 – mean = total / count;

Ex. 2.56 – if ( price > budget) {

System.out.println(" Too expensive! ");

}

Else {

System.out.println(" Just right. ”);

}

Ex. 2.57 – if ( price > budget) {

System.out.println(" Too expensive! ");

System.out.println("Your budget is " + budget + “dollars” );

}

Else {

System.out.println(" Just right. ”);

}

Ex. 2.58 – This won’t work because the balance has been set to zero so you are garunteed a refund of 0 cents every time. We can test and find this is true by removing int amountToRefund amountToRefund= balance and finally change the return to balance. Once compiled and called we will see that it doesn’t work.

Ex. 2.59 – The return statement must be last or the compiler can’t find it.

Ex. 2.60 – int should not be infront of price as price is a field and defined as and int outside the constructor.

Ex. 2.61 – Done and saved.

Ex. 2.62 – Done and saved.

Ex. 2.63 – Done and saved. I had to add a new field to store the original ticket price and one issue I found with it is if you keep saying “yes” it stacks the discount.

Ex. 2.64 – The name of this method is getCode and the return type is a string.

Ex. 2.65 – the name of this method is setCredits and its parameter is creditValue and has type int.

Ex. 2.66 – public class Person

{

}

Ex. 2.67 – private String name;

private int age;

private String code;

private int credits;

Ex. 2.68 – public Module( String moduleCode)

{

Code = String moduleCode;

}

Ex. 2.69 – public Person ( String myName, int myAge)

{

Name = myName;

Age = myAge;

}

Ex. 2.70 – in this example the return type should be int not void.

Ex. 2.71 – public String getName( )

{

Return name;

}

Ex. 2.72 – public void setAge( int newAge)

{

Age = newAge;

}

Ex. 2.73 – public void printDetails()

{

System.out.println (“ The name of this person is ” + name + “.”);

}

Ex. 2.74 –

Student1:

Student

Name Benjamin Jonson

ID number 738321

Ex. 2.75 – This student login would be Henr557

Ex. 2.76 – If a student only had a 3 character name and we used the getLoginName method we would get an error message\theat says exception occurred and that the “sting index out of range”. This error occurs because a fourth character in string name doesn’t exist and is therefore impossible to use to build a login.

Ex. 2.77 – Done and saved.

Ex. 2.78 – Done and Saved

Ex. 2.79– Done, the results were 102, catfish, cat9, 12cat, cat39, f, and an error message as 8 was out of the strings range. The only thing that surprised me was that is combound the sting and an int into a string.

Ex. 2.80 – The first call the balance is zero and the second returned a balance of 500 because we used the inserMoney method.

Ex. 2.81 – 500 should be returned and it was.

Ex. 2.82 – If we add another 500 to t1 t2 should return a balance of 1000 because we made t2 = to t1 so the two will always be the same.

Ex. 2.83 – Done and saved.

Ex. 2.84 – Done and saved.

Ex. 2.85 – Done and saved.

Ex. 2.86 – Once create a book object is immutable. Once it is created we are unable to further edit the instance.

Ex. 2.87 – Done and saved.

Ex. 2.88 – Done and saved.

Ex. 2.89 – Done and saved.

Ex. 2.90 – Done and saved.

Ex. 2.91 – Done and saved.

Ex. 2.92 – Done and saved.

Ex. 2.93 – Done and saved.

Ex. 2.94 – Done and saved.