Calculate the following:

```
(int) 16.7 / 3
```

a) 3
b) 1.7
c) 5.567
d) 5
e) illegal operation

Solution:
d)

Calculate the following:

`2.25 * 3 % 4`

```
a) 6
b) 2.75
c) 1
d) 6.75
e) illegal operation
```

```
Solution:
b)
```

Calculate the following:

```
2 * 8 % 9 - 11 + 72 / 9
```

a) 6
b) 5.5
c) 4
d) 22
e) illegal operation

Solution:
c)

STUDENT OUTLINE

Lesson 6 – Defining and Using Classes

INTRODUCTION: The previous lessons have discussed how to use objects and their methods. But all the objects have been created using a class from one of the Java libraries or curriculum supplied classes. This lesson discusses how to define your own classes and objects.

The key topics for this lesson are:

A. Designing a Class
B. Determining Object Behavior
C. Instance Variables
D. Implementing Methods
E. Constructors
F. Using Classes

VOCABULARY:

ATTRIBUTES	ACCESS SPECIFIER
BEHAVIORS	CONSTRUCTOR
ENCAPSULATION	INSTANCE VARIABLE
METHOD CALLS	OVERLOADED
REFERENCE	

DISCUSSION: A. Designing a Class

NOUNS & VERBS

1. One of the advantages of object-oriented design is it allows a programmer to create a new abstract data type that is reusable in other situations

2. When designing a new data type, two components must be identified - attributes and behaviors.

A.K.A. INSTANCE VARIABLES

3. Consider the icons used in computer operating systems. The attributes that describe the icon are things like a graphic pattern, colors, size, name, and its position on the screen. Some of its behaviors include changing color and moving its position.

4. The attributes of an object are the nouns that describe that object. For example, in our checking account example below, the attributes are "current balance" and "account number". These will become the private data members of a class.

5. The behaviors of an object are the verbs that denote the actions of that object or what it does. For example, in our checking account example below, behaviors are "accept a deposit", "process a check", etc. These will become the member methods of a class. In a Java program, behaviors of an object are described by methods.

(PLANNING FOR WHAT CODE TO WRITE)

B. Determining Object Behavior

1. In this section, you will learn how to create a simple class that describes the behavior of a bank account. Before you start programming, you need to understand how the objects of your class behave. Operations that can be carried out with a checking account could consist of:

MODEL THESE WITH CLASS METHODS

- Accept a deposit
- Withdraw from the account
- Get the current balance

INSTANCE (OBJECT)

2. In Java, these ~~operations~~ ACTIONS are expressed as *method calls*. For example, assume we have an object `checking` of type `CheckingAccount`. The methods that invoke the required behaviors

CLASS (BLUEPRINT)

WITH THESE IDEAS WE ANTICIPATE MAKING METHOD CALLS LIKE THESE

```
checking.deposit(1000)
checking.withdraw(250)
System.out.println("Balance: " + checking.getBalance());
```

are represented by the set of methods

```
deposit
withdraw
getBalance
```

HELPS US TO DECIDE ON METHOD IDENTIFIERS

These methods form the behavior of the `CheckingAccount` class. The behavior is the complete list of the methods that you can apply to objects of a given class. An object of type `CheckingAccount` can be viewed as a "black box" that can carry out its methods.

ALL OF THIS IS LIKELY TO BE WRITTEN IN METHOD main() NOT IN CheckingAccount CLASS

3. To construct objects of the `CheckingAccount` class, it is necessary to declare an object variable

```
CheckingAccount checking;
```

DECLARATION

Object variables such as `checking` are *references* to objects. Instead of holding an object itself, a reference variable holds the information necessary to find the object in memory.

4. This object variable `checking` does not refer to any object at all. An attempt to invoke a method on this variable would cause the compiler to generate an error indicating that the variable had not been initialized. To initialize the variable, it is necessary to create a new `CheckingAccount` object using the **new** operator

```
checking = new CheckingAccount();
```

INSTANTIATION

This call creates a new object and returns a reference to the newly created object. To use an object, you must assign that reference to an object variable.

A.K.A. REFERENCE VARIABLE

A CLASS WHICH CONTAINS THE main() METHOD WILL CALL A CLIENT OR DRIVER CLASS

⊕ EXTRA ⊕

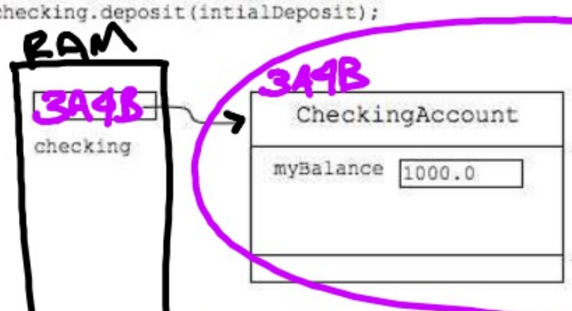
5. We will implement (that is, create and write the code for) the CheckingAccount object so that the account has an initial balance of 1000.0 dollars.

```
// open a new account
double initialDeposit = 1000.0;
CheckingAccount checking = new CheckingAccount();

// set initial balance to 1000.0
checking.deposit(intialDeposit);
```

RAM STACK

DETAILS ARE HIDDEN BY JAVA LANGUAGE



HEAP

Figure 6-1. Creating a New Object

KNOWN AS ENCAPSULATION A.K.A. INFORMATION HIDING

6. Objects of the CheckingAccount class can be used to carry out meaningful tasks without knowing how the CheckingAccount objects store their data or how the CheckingAccount methods do their work. This is an important aspect of object-oriented programming.

STEP 2 IN CLASS DESIGN

7. Once we understand how to use objects of the CheckingAccount class, it is possible to design a Java class that implements its behaviors. To describe object behavior, you first need to implement a class, and then implement methods within that class.

```
public class CheckingAccount
{
  // CheckingAccount data

  // CheckingAccount constructors

  // CheckingAccount methods
}
```

NOUNS

VERBS

FROM PG 2

Next we implement the three methods that have already been identified:

```
deposit
withdraw
getBalance
```

METHOD SIGNATURES

```
public class CheckingAccount
{
  // CheckingAccount data

  // CheckingAccount constructors

  public void deposit( double amount )
  {
    // method implementation
  }
  public void withdraw( double amount )
  {
    // method implementation
  }
  public double getBalance()
  {
    // method implementation
  }
}
```

DECLARES A LOCAL DATA SPACE

THIS METHOD IS PUBLICLY AVAILABLE!

8. A *method signature* consists of the following parts:

access_specifier return_type *method_name* (*arguments*)

a. An ***access_specifier*** (such as **public**). The access specifier controls which other methods can call this method. Most methods should be declared as public so all other methods in your program can call them.

b. The ***return_type*** of the method (such as **double** or **void**). The return type is the type of the value that the method computes. For example, in the CheckingAccount class, the getBalance method returns the current account balance, which is a floating-point number, so its return type is **double**. The deposit and withdraw methods don't return any value. To indicate that a method does not return a value, you use the special type **void**.

(*) NOMENCLATURE

c. The *method_name* (such as deposit).

PASS A PARAMETER * (CALL)

ACCEPT AN ARGUMENT (DEFINITION)

d. A list of the *arguments* of the method. The arguments are the input received by the method. The deposit and withdraw methods each have one argument, the amount of money to deposit or withdraw. The type of argument, such as **double**, and identifier for each argument, such as amount, must be specified. If a method has no arguments, like getBalance, it is still necessary to supply a pair of parentheses () behind the method name.

SPACE FROM { TO }

9. Once the method signature has been specified, the implementation of the method must be supplied in a block that is delimited by delimiters {...}.

   The CheckingAccount methods will be implemented later in Section D.

C. Instance Variables

NOUNS

1. Each object must store its current *state*. The state is the set of values that describe the object and that influence how an object reacts to method calls. In the case of our checking account objects, the state is the current balance and an account identifier.

2. Each object stores its state in one or more *instance variables*.

FROM ELSEWHERE, I CANT SAY checking.myBalance=0;

```
public class CheckingAccount
{
  ...
  private double myBalance;
  private String myAccountNumber;

  // CheckingAccount methods
}
```

PROTECTION FROM RUDENESS.

3. An instance variable declaration consists of the following parts:

   access_specifier type *variable_name*

   a. An ***access_specifier*** (such as **private**). Instance variables are generally declared with the access specifier **private**. That means they can be accessed only by methods of the *same class*, not by any other method. In particular, the balance variable can be accessed only by the deposit, withdraw, and getBalance methods.
   b. The ***type*** of the variable (such as **double**).
   c. The *variable_name* (such as myBalance).

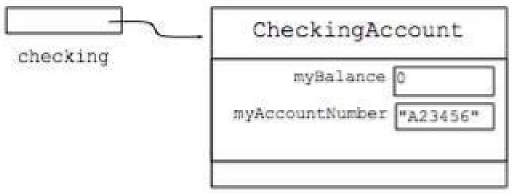


Figure 6-2. Instance Variables

4. If instance variables are declared private, then all data access must occur through the public methods. This means that the instance variables of an object are effectively hidden from the programmer who only uses a class. They are available only to the programmer who implements (defines) the class, that is, the one who writes or revises the methods. The process of hiding data is called *encapsulation*. Although it is possible in Java to define instance variables as **public** (leave them un-encapsulated), it is very uncommon in practice. We will always make instance variables **private** in this curriculum guide.

IF I SEE THIS IN YOUR LABS....

5. For example, because the `myBalance` instance variable is **private**, it cannot be accessed in other code:

IN THE CLIENT

```
double balance = checking.mybalance;  // compiler ERROR!
```

However, the public `getBalance` method to inquire about the balance can be called:

```
double balance = checking.getBalance();  // OK
```

DEFINING

D. Implementing Methods

1. An implementation must be provided for every method of the class. The implementation for three methods of the `CheckingAccount` class is given below.

```
public class CheckingAccount
{
  private double myBalance;
  private String myAccountNumber;

  public double getBalance()
  {
    return myBalance;
  }

  public void deposit(double amount)
  {
    myBalance =  myBalance + amount;
  }

  public void withdraw(double amount)
  {
    myBalance =  myBalance - amount;
  }
}
```

AN ACCESSOR METHOD

THESE ARE PAIRED

AN ARGUMENT IS LIKE A DECLARATION FOR THIS BLOCK ONLY

THESE ARE MODIFIER METHODS

2. The implementation of the methods is straightforward. When some amount of money is deposited or withdrawn, the balance increases or decreases by that amount.

3. The `getBalance` method simply *returns* the current balance. A **return** statement obtains the value of a variable and exits the method immediately. The return value becomes the value of the method call expression. The syntax of a **return** statement is:

```
return expression;
```

or

DONT DO THIS

```
return;  // Exits the method without bringing back a value
```

E. Constructors

1. The final requirement to implement the `CheckingAccount` class is to define a *constructor*, whose purpose is to initialize the values of instance variables of an object.

MUST MATCH

DEFAULT INITIALIZATIONS

```
public class CheckingAccount
{
  ...
  public CheckingAccount()  // constructor
  {
    myBalance = 0;
    myAccountNumber = "NEW";
  }
  ...
}
```

ACTUALLY DEFINING THE DEFAULT CONSTRUCTOR

2. Constructors always have the same name as their class. Similar to methods, constructors are generally declared as public to enable any code in a program to construct new objects of the class. Unlike methods, constructors ~~do not~~ NEVER have return types.

3. Constructors are always invoked together with the **new** operator:

```
new CheckingAccount();
```

(IN THE CLIENT)

The new operator allocates memory for the objects, and the constructor initializes it. The value of the new operator is the reference to the newly allocated and constructed object.

In most cases, you want to declare and store a reference to an object in an object variable as follows:

YOU MAY NOW DO THIS!!

```
CheckingAccount checking = new CheckingAccount();
```

4. If you do not initialize an instance variable that is a number, it is initialized automatically to zero. Even though, initialization is handled automatically for instance variables, it's a matter of good style to initialize all instance variables explicitly.

DO THIS!!

5. Many classes have more than one constructor. For example, you can supply a second constructor for the `CheckingAccount` class that sets the `myBalance` and `accountNumber` instance variables to initial values, which are the parameters of the constructor:

```
public class CheckingAccount
{
  ...

  public CheckingAccount() // constructor defines values
  {
    myBalance = 0;
    myAccountNumber = "NEW";
  }

  public CheckingAccount(double initialBalance, String acctNum)
   // contructor gets values elsewhere
  {
    myBalance = initialBalance;
    myAccountNumber = acctNum;
  }
  ...
}
```

THE CLIENT SUPPLIES

The second constructor is used if ~~you supply~~ a number and a string as construction parameters.

```
CheckingAccount checking = new CheckingAccount(5000, "A123");
```

THE COMPILER CHOOSES WHICH ONE IS CORRECT TO USE. THIS CHOOSING IS KNOWN AS EARLY BINDING

6. Note that in the above example there are two constructors of the same name. Whenever you have multiple methods (or constructors) with the same name, the name is said to be *overloaded*. The compiler figures out which one to call by looking at the arguments and difference in signaute of each method.

For example, if you construct a new `checkingAccount` object with

```
CheckingAccount checking = new CheckingAccount();
```

then the compiler picks the first constructor. If you construct an object with

```
CheckingAccount checking = new CheckingAccount(5000, "A123");
```

then the compiler picks the second constructor.

7. The implementation of the CheckingAccount class is complete and is given below:

SAVED INTO 1 .java FILE NAMED

```
public class CheckingAccount
{
  private double myBalance;                    //INSTANCE VARIABLES
  private String myAccountNumber;

  public CheckingAccount()
  {
    myBalance = 0;                             // <<CONSTRUCTORS>>
    myAccountNumber = "NEW";
  }

  public CheckingAccount(double initialBalance, String acctNum)
  {
    myBalance = initialBalance;
    myAccountNumber = acctNum;
  }

  public double getBalance()                   // <<ACCESSOR>>
  {
    return myBalance;
  }

  public void deposit(double amount)
  {
    myBalance =  myBalance + amount;           // <<MODIFIERS>>
  }

  public void withdraw( double amount )
  {
    myBalance =  myBalance - amount;
  }
}
```

F. Using Classes

1. Using the CheckingAccount class is best demonstrated by writing a program that solves a specific problem. We want to study the following scenario:

> A interest bearing checking account is created with a balance of $1,000. For two years in a row, add 2.5% interest, compounded annually. How much money is in the account after two years?

2. Two classes are required: the `CheckingAccount` class that was developed in the preceding sections, and a second class called `CheckingTester`. The main method of the `CheckingTester` class constructs a `CheckingAccount` object, adds the interest twice, then prints out the balance.

OUR CLIENT (IN ITS OWN .java FILE)

```
class CheckingTester
{
  public static void main(String[] args)
  {
    CheckingAccount checking =
        new CheckingAccount(1000,"A123");

    final double INTEREST_RATE = 2.5;
    double interest;

    interest = checking.getBalance() * INTEREST_RATE / 100;
    checking.deposit(interest);

    System.out.println("Balance after year 1 is $"
        + checking.getBalance());

    interest = checking.getBalance() * INTEREST_RATE / 100;
    checking.deposit(interest);

    System.out.println("Balance after year 2 is $"
        + checking.getBalance());
  }
}
```

3. The classes can be distributed over multiple files or kept together in a single file. If kept together, the class with the `main` method must be declared as **`public`**. The **`public`** attribute cannot be specified for any other class in the same file since a Java source file can contain only one **`public`** class.

4. Care must be taken to ensure that the name of the file matches the name of the public class. For example, a single file containing both the `CheckingAccount` class and the `CheckingTester` class must be contained in a file called `CheckingTester.java`, not `CheckingAccount.java`.

SUMMARY/ REVIEW: The topics in this lesson are critical in your study of computer science. The concepts of abstraction and object-oriented programming (OOP) will continue to be developed in future lessons. Before you solve the lab exercise, you are encouraged to play with the `CheckingAccount` class and implement objects using all the behaviors of the class.

ASSIGNMENT: Lab Exercise, L.A.6.1, *MPG*
Lab Exercise, L.A.6.2, *Rectangle*