Handout 8

Defining classes

Class is a type of a non-primitive value (called object). Class definition provides a template for creating objects of that class.

- Each object has **attributes** (data) and **operations** (behaviors) associated with it.
    - An object's **attributes** are its **instance variables**.
    - An object's **operations** are its **instance methods**.

UML (Unified Modeling Language)

`Class Name`

`Attributes`
`(a.k.a. instance variables,`
`fields or properties)`

`Method Names`

Example: The Employee class Two `Employee` objects

CLASS Employee	**employee1**	**employee2**
String name double payRate double hours	name: **Deborah Stone** payRate: 1**7. 45** hours: 1**0**	name: **George Clark** payRate: 2**2.00** hours: **2**

getName()
getPayRate()
getUnpaidHours()
incrementHoursBy(double hrs)
amtDue()
printEmployeeCheck(String filename)
	getName()
	getPayRate()
	getUnpaidHours()
	incrementHoursBy(hrs)
	amtDue()
	printEmployeeCheck(String filename)
		getName()
		getPayRate()
		getUnpaidHours()
		incrementHoursBy(double hrs)
		amtDue()
		printEmployeeCheck(String filename)

The instance methods would return the values **based on <u>the</u> calling object's data**, e.g.

employee1.getPayRate()
 would return 17.45
employee2.getPayRate()
 would return 22.0

Example: **BankAcoount class**

```
/* A class definition consisting of two instance vars
 */
public class BankAccount {
        // instance variables
      public double balance;
      public String accountID;
}
```

Class that uses BankAccount

```
/* *  Example: create an object, access its public instance variable
 */
public class BankAccountDemo {

      public static void main(String[] args) {

            // create an object of class BankAccount
            BankAccount myAccount = new BankAccount();
            BankAccount joeAccount = new BankAccount();

            myAccount.balance = 500;
            myAccount.accountID  = "T3849-55";

            joeAccount.balance = 300;
            joeAccount.accountID = "HGF-4565";

            System.out.println(myAccount.balance);
            System.out.println(joeAccount.balance);
      }
}
```

Instantiation: Creation of an object from a class.

- Create new objects of a class:
  - Use the keyword `new` followed by the name of the class. `BankAccount myAccount = new BankAccount();`

`BankAccount()` – is called **the default constructor**.

Instance variables

Are accessible (i.e. can be used) in every instance method of the same class.

Public versus private instance variables:

Public instance variables can be accessed directly by name the outside of instance methods of their class, as shown:

```
myAccount.balance = 500;
```

Private instance variables **cannot** be accessed from outside of instance methods of their class.

Proper design: Define **data attributes** as **private variables**

- Only methods of that class can access those attributes.
- Protects the data values from users of that class
- Only the programmer can control how they can be modified.

Define **operations** on that class as **public methods -** available to users of the class.

What is **`this`**?

- The calling object (i.e. the object that invoked the instance method) is passed to the instance method as an **implicit parameter**. Keyword **`this`** is used within an instance method to refer to the **calling object.**
- Keyword **`this`** can be omitted whenever there are no local variables or parameters with the name that match the name of an instance variable, but is often used for clarity to distinguish instance variables from other variables.

Encapsulation, or *information hiding*:

- Ability to hide data and behaviors within an object so that only that object can access and change them. For `BankAccount` class:
  - **instance variable**: `balance` declared **private**
  - `declared` **`public :`**
    - **accessor methods**: `getBalance , getAccountID`
    - **mutator method**: `deposit, setAccountID`

```java
/* A class definition consisting of a single instance var */
public class BankAccount {

	  // instance variable
	private double balance;
	private String accountID;

		// mutator method deposit
	public void deposit(double amount) {
		/* increment balance by amount passed to method */
		if (amount > 0)
			this.balance += amount;
	}

	// accessor method getBalance
	public double getBalance() {
		/* pass back account balance  */
		return this.balance;
	}

	// accessor method getAccountID
	public String getAccountID() {
		/* pass back account balance  */
		return this.accountID;
	}

	// mutator method  setAccountID
	public void setAccountID( String id) {
		/* pass back account balance  */
		if (id.length() != 0 )
			this.accountID = id;
	}  }
```

```
/* *  Example: create an object, access its public instance variable
 */
public class BankAccountDemo {

      public static void main(String[] args) {

            BankAccount myAccount = new BankAccount();


            // The foll. would not work because balance is private
            // myAccount.balance = 500;
            // myAccount.accountID = "ABC-345"
            // Instead, must use the mutator methods to set balancea and
accid
            myAccount.setAccountID("ABC-345");
            myAccount.deposit(500.0);

            //System.out.println(myAccount.balance); doesn't work
            // Instead, use the accessor methods
            System.out.println(myAccount.getAccountID());
            System.out.println(myAccount.getBalance());

      }
}
```

Practice Problem:

1. Add instance method withdraw to the BankAccount class. The method should be passed the amount of withdrawal as a parameter. If the amount is less or equal to the balance of the calling object, then the balance instance variable should be decreased by the amount of withdrawal. Otherwise, the method should leave the balance variable unchanged and should display a message stating there are no sufficient funds for the withdrawal.
2. Test the developed method by adding appropriate calls to the BankAccountDemo's main method.

The `toString()` method:

- All classes include a **`toString()`** method
    - ♦ returns a string representation of the class.
    - ♦ Defined in Java's `Object` class

- If the programmer doesn't provide one, then Java will provide:
    - ♦ The name of the object's class and that object's address in memory.

Example: if add the following code to the `UseBankAccount` class:

```
System.out.println(myAccount);
```

Something like the following will print:

```
BankAccount@7307ec83
```

User-defined `toString()` methods:

- ♦ Define a `toString()` method for each class.   The value returned must be a String.

```
public String toString() {
        return "Account #" + this.accountID +
                    " has balance " + this.balance;
}
```

- Use the `println()` method with an object reference to invoke the `toString()` method for that object.

Practice Problem:

3. Implement Employee class depicted in the diagram on the first page. It should include
    - Private instance variables to represent employee's name (`name`), hourly pay rate (`hourlyPayRate`) and number of hours for which the pay is due (`hours`).
    - Public instance methods to
        - setName(String empName)
        - setPayRate(double payRate)
        - toString()
        - Accessor methods for all three variables
        - incrementHoursBy(double hrs)
            - that will increase the employee's number of hours by the value passed as a parameter, in case the parameter is a positive value
        - amtDue()
            - that will calculate and return the amount due to the employee for the hours worked
        - printEmployeeCheck(String filename)
            - that will create a file NAME-CHECK.txt with the following content

              ```
              ********************************************
              Pay to:    EMPLOYEE NAME
              Amount: AMOUNT DUE
              ********************************************
              ```

            - set the instance variable `hours` to 0.

    Test Employee class, **as you develop it:**
    - create two Employee type objects and initialize them with data using mutator methods
    - call all developed methods to verify their correct behavior