

## Learning Notes for Unit 3 – Part A

### Methods (aka functions):

In your first programming course, you learned how to create a function. Simply put, functions are blocks of code that are removed from the main routine to:

1. Simplify the main routine (i.e. main routine is shortened and reads like a sequence of function calls)
2. Avoid code duplication.

Please refer to the reference section for notes on how functions are composed and how they are accessed.

### Creating Storage Classes:

As seen in the Unit 2, there are a number of data types that can be used in your programs.

Single value storage options include: int, double, long, short, char, float, byte, Boolean

Sometimes you need to collect multiple values together in a single unit. For example, let's assume you want to store five different course marks for a student. In this case, you could use an array of double or integer data values. However, the values MUST be of the same data type.

If the data types to be collected together are of different types, you must create a custom class definition for your collection.

The skeleton of a storage class:

# Creating a storage object in Java

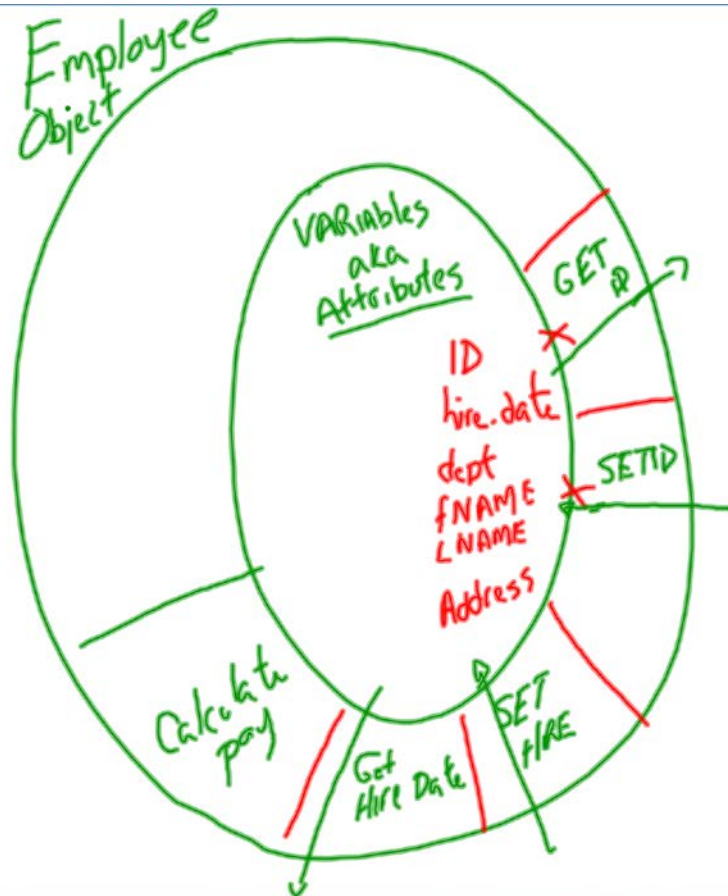
Skeleton:

```
public StorageClassName {  
    // Storage variable List  
    // Constructor(s) → where stored variables  
    //                      are initialized  
  
    // functions that allow you to interact  
    // with stored variables  
}
```

\* NO MAIN!

As you can see in the skeleton above, there is no main function in the class definition. This class contains only the required information to store your collection of information. You would use it as you would use a String data type in a program.

You can envision a software object to behave like an egg (pictured below). Inside the egg, you have a number **attributes** that describe your object. These attributes are surrounded by a hard shell which means you do not have direct access to them. To gain this protection, the keyword “**private**” is used to prevent direct access. At the interface between the outside world and the interior of the egg, you have the shell. The class would be of little to no use to us if there is no way to access what is inside. So we poke some holes in the shell to provide a gateway in and out. These **methods**/functions have public access which means they can be accessed from outside. But the method itself is actually inside the shell. It has direct access to the stored attributes. We can have methods to retrieve each stored value and send it out (**accessors**). We can have methods to send a value into the object and overwrite an attribute (**mutators**).



Adding Attributes to a Class (code):

```
public Employee {  
  
    //list of attributes  
    private int ID;  
    private String HireDate;  
    private String Department;  
    private String FirstName;  
    private String LastName;  
    private String Address;  
  
    //functions that use these attributes  
  
}
```

← intension of egg structure  
on previous slide

↗ private keyword provides  
the barrier against direct  
access

Adding Accessors/Mutators to a Class (code):

```
//functions that use these attributes
public void setID (int temp) { ID = temp; }
public void setHireDate (String temp) { HireDate = temp; }
public void setDepartment (String temp) { Department = temp; }
public void setFirstName (String temp) { FirstName = temp; }
public void setLastName (String temp) { LastName = temp; }
public void setAddress (String temp) { Address = temp; }
public int getID () { return ID; }
public String getHireDate () { return HireDate; }
public String getDepartment () { return Department; }
public String getFirstName () { return FirstName; }
public String getLastName () { return LastName; }
public String getAddress () { return Address; }
```

mutators -> change  
stored value

Accessors -> retrieve  
stored value and  
return it to outside  
application

Seven Steps to create and test storage classes:

Steps to create and test a storage class

- (1) Set up storage class template (no main...)
- (2) Read the problem and determine what variables are required to store.  
Declare those variables at the top of the class body
- (3) Create a get and a set function for each variable  
(accessor and mutator functions)
- (4) Create a default constructor to set default values for our  
attributes
- (5) Create any additional constructors
- (6) Create any additional functions that the problem requires
- (7) Write a Java Application to use and test our storage class

Steps 4 and 5 will be described later in this unit. An example of step 6 is below:


---

```
private String Address;
private int HoursWorked;
private int RateOfPay;

//functions that use these attributes
public void setID (int temp) { ID = temp; }
public void setHireDate (String temp) { HireDate = temp; }
public void setDepartment (String temp) { Department = temp; }
public void setFirstName (String temp) { FirstName = temp; }
public void setLastName (String temp) { LastName = temp; }
public void setAddress (String temp) { Address = temp; }
public void setHours (int temp) { HoursWorked = temp; }
public void setRate (int temp) { RateOfPay = temp; }

public int getID () { return ID; }
public String getHireDate () { return HireDate; }
public String getDepartment () { return Department; }
public String getFirstName () { return FirstName; }
public String getLastName () { return LastName; }
public String getAddress () { return Address; }
public int getHours () { return HoursWorked; }
public int getRate () { return RateOfPay; }

public int NetIncome () { int total=HoursWorked*RateOfPay; return total; }
```



Compile to create the .class byte code.

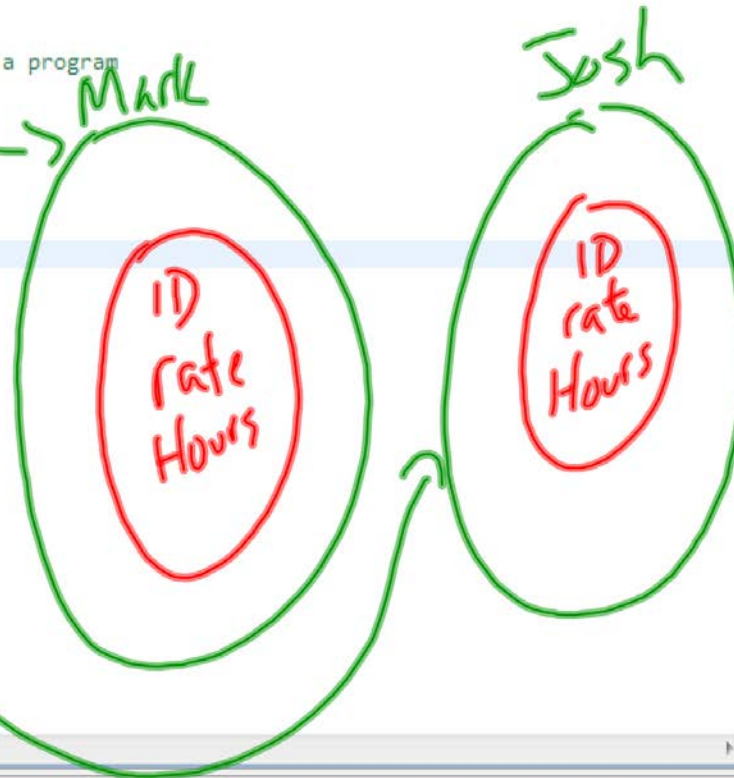
### Using Storage Classes in a Program:

Step 1: Create a new java program with a main function (i.e. will run).

Step 2: Declare an instance of your storage class (your new program must be in the same folder as the storage class's ".class" byte code)

**Employee Mark = new Employee();** creates an instance of the Employee storage class for use in the program

```
public class TestEmployee {  
    public static void main(String[] args) {  
        //Declare an instance of our custom class to use in a program  
        Employee Mark = new Employee();  
        Employee Josh = new Employee();  
    }  
}
```





Use the accessor/mutator methods to interact with the attributes:

```
public class TestEmployee {  
    public static void main(String[] args) {  
        //Declare an instance of our custom class to use in a program  
        Employee Mark = new Employee();  
        Employee Josh = new Employee();  
        Mark.setHours(50);  
        Mark.setRate(11);  
        Josh.setHours(40);  
        Josh.setRate(15);  
    }  
}
```

Mark

Hours=50  
Rate=11

Josh

Hours=40  
Rate=15

problems @ Javadoc Declaration Console

onsoles to display at this time.

In code:

```
public class TestEmployee {  
  
    public static void main(String[] args) {  
  
        //Declare an instance of our custom class to use in a program  
  
        Employee Mark = new Employee();  
  
        Employee Josh = new Employee();  
  
        Mark.setHours(50);  
        Mark.setRate(11);  
  
        Josh.setHours(40);  
        Josh.setRate(15);  
  
        int josh_rate = Josh.getRate();  
        int josh_hours = Josh.getHours();  
        System.out.println("Josh earns " + josh_rate + " and he has worked " + josh_hours + " hours");  
  
        int mark_rate = Mark.getRate();  
        int mark_hours = Mark.getHours();  
        System.out.println("Mark earns " + mark_rate + " and he has worked " + mark_hours + " hours");  
  
        int josh_pay = Josh.NetIncome();  
        int mark_pay = Mark.NetIncome();  
  
        System.out.println("Mark earned " + mark_pay + ". Josh earned " + josh_pay + ".");  
    }  
}
```

Problems Javadoc Declaration Console

<terminated> TestEmployee [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (Feb 2, 2012 2:04:10 AM)

Josh earns 15 and he has worked 40 hours

Mark earns 11 and he has worked 50 hours

Mark earned 550. Josh earned 600.

### Constructors:

Constructors are special methods/functions that are called when memory is allocated for an object.

Ex. `Employee Mark = new Employee();` <= correct declaration syntax

The “new” keyword is a command that will allocate enough memory to store the Employee variable. You will note that the allocation is not:

**`Employee Mark = new Employee;` <= incorrect declaration syntax**

The `Employee()` portion of the declaration is actually a calling of the default constructor for the Employee class.

The role of a constructor is to initialize the attributes (because **`private int ID=0;`** is not allowed when declaring the attributes in a class). A constructor call can have nothing in the parentheses as shown above which is a call to the class’s default constructor, or it can contain values which will be sent inside the object to initialize the attributes to those custom values.

Rules for writing a constructor:

## Rules for writing a constructor

- function name has same name as storage class
- no return type, not even void
- Can have 0 parameters in variable list (called default constructor)

or many parameters

- the parameter list must be different in the number of parameters and/or the data types of the parameters

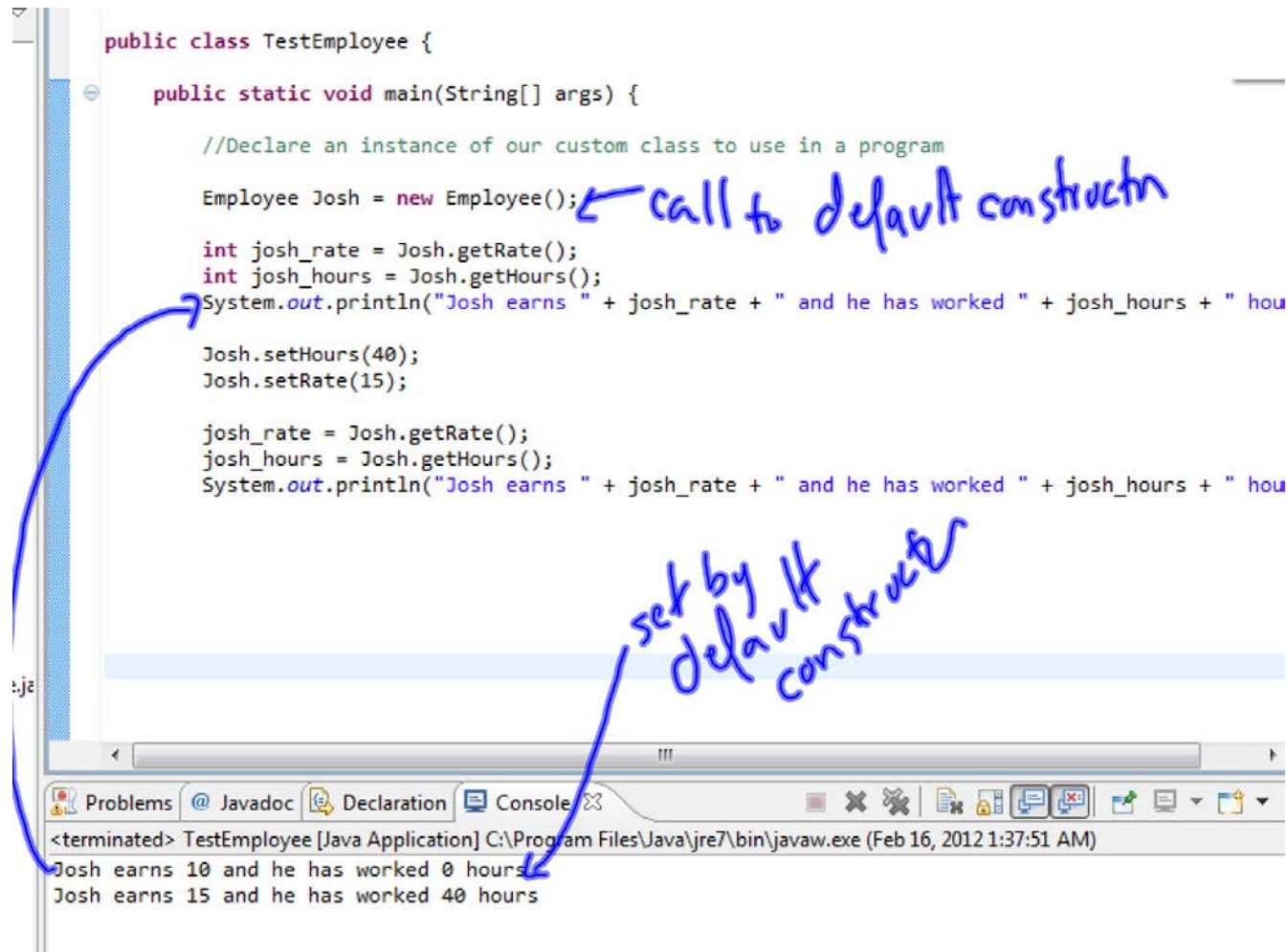
**AMBIGUITY!**

Setting a Default Constructor for your class (code):

```
public class Employee {  
    //list of attributes  
    private int ID;  
    private String HireDate;  
    private String Department;  
    private String FirstName;  
    private String LastName;  
    private String Address;  
    private int HoursWorked;  
    private int RateOfPay;  
  
    //Default Constructor  
    public Employee () {  
        ID = 123;           HireDate = "1970-01-01";  
        Department = "None";  FirstName = "J";  
        LastName = "Doe";     Address = "nowhereville";  
        HoursWorked=0;        RateOfPay=10;  
    }  
}
```

initial values set  
to these until you  
change them with  
mutator calls  
(set functions)

Calling a default constructor (code):



```
public class TestEmployee {  
    public static void main(String[] args) {  
        //Declare an instance of our custom class to use in a program  
        Employee Josh = new Employee();  
        int josh_rate = Josh.getRate();  
        int josh_hours = Josh.getHours();  
        System.out.println("Josh earns " + josh_rate + " and he has worked " + josh_hours + " hou  
        Josh.setHours(40);  
        Josh.setRate(15);  
  
        josh_rate = Josh.getRate();  
        josh_hours = Josh.getHours();  
        System.out.println("Josh earns " + josh_rate + " and he has worked " + josh_hours + " hou
```

Josh earns 10 and he has worked 0 hours  
Josh earns 15 and he has worked 40 hours

*call to default constructn*

*set by default constructn*



Every storage class requires at least A default constructor to be defined. In limited circumstances, the compiler will automatically generate one for you. However, things will break if the class is used for inheritance.

### Secondary Constructors:

You can have multiple constructors defined within your class. Each definition must differ in EITHER the number of parameters within the parentheses OR the data types of the parameters must be sufficiently different that the compiler can tell which one is being called.

```
Employee Brandon = new Employee(123, "2012-02-15", "Java", "B", "MacDonald",  
                                "Funky Town", 10, 10);
```

Q looks for constructor  
with 8 fields

### Secondary Constructor definition (code):


From above: constructor with 8 fields:

```
public Employee (int temp1, String temp2, String temp3, String temp4, String temp5, String temp6, int temp7, int temp8) {  
  
    ID=temp1;           HireDate=temp2;       Department=temp3;  
  
    FirstName=temp4;     LastName=temp5;    Address=temp6;  
  
    HoursWorked=temp7;   RateOfPay=temp8;  
  
}
```



### Creating a Display function:

Defining a display function is a good way to display the state of the attributes with a single function call.



```
public void Display() {  
    System.out.println("ID:" + ID + ", FirstName: " + FirstName +", LastName:" + LastName +  
        "Address:" + Address + ", Department: " + Department +  
        ", HireDate:" + HireDate +", HoursWorked:" + HoursWorked +  
        ", RateOfPay:" + RateOfPay);  
}
```

Mark.Display() would display the current values of the attributes in the Mark object.

Please refer to the reference section for slideshow notes and source code on how a storage class is created and how they are accessed in a program.

## Learning Notes for Unit 3 – Part B

### Overloading a Method:

As shown in Part A of this unit (in the section regarding multiple constructors), you can have multiple functions with the same name with each demonstrating different behaviors.

For example:

#### **Method A Defined:**

```
public void setAllValues () {  
    ID=0;           HireDate="";      Department="";  
    FirstName="";   LastName="";      Address="";  
    HoursWorked=0;  RateOfPay=0;  
}
```

Method A called in a program would look like:

**Mark. setAllValues ();**

**Method B Defined:**

```
public void setAllValues (int temp1, String temp2, String temp3, String temp4, String temp5, String temp6, int temp7, int temp8) {  
  
    ID=temp1;           HireDate=temp2;       Department=temp3;  
    FirstName=temp4;    LastName=temp5;      Address=temp6;  
    HoursWorked=temp7;  RateOfPay=temp8  
  
}
```

Method B called in a program would look like:

**Mark. setAllValues (123, "2012-02-22", "Java", "Me", "You", "Somewhere", 10, 15);**

Having two functions with the same names, but performing different tasks is a demonstration of polymorphism. The compiler decides which method is being called based on the way the method is being called.

In this case, if you call the version with the 8 values, it knows which one. Likewise if you leave the parentheses empty, it knows which one to call. Things get tricky when you have two functions with the same name and the same number of parameters.

**Method C Defined:**

```
public void setAllValues (int temp1, String temp2) {  
  
    ID=temp1;           HireDate=temp2;       Department="";  
    FirstName="";       LastName="";          Address="";  
    HoursWorked=0;      RateOfPay=0;  
  
}
```

**Method D Defined:**

```
public void setAllValues (String temp1, String temp2) {  
  
    ID=0;                HireDate=temp1;    Department=temp2;  
    FirstName="";        LastName="";      Address="";  
    HoursWorked=0;       RateOfPay=0;  
  
}
```

In this case, the compiler can tell which the difference between the methods because the types of the parameters are sufficiently different.

**Method E Defined:**

```
public void setAllValues (String temp1, String temp2) {  
  
    ID=0;                HireDate="";      Department="";  
    FirstName= temp1;    LastName= temp2;  Address="";  
    HoursWorked=0;       RateOfPay=0;  
  
}
```

In this case, the compiler cannot tell the difference between methods D and E because the types of the parameters are not sufficiently different. The compiler will throw an “ambiguous” error message.

## Learning Notes for Unit 3 – Part C

As mentioned earlier in this unit, arrays are collections of values using a single variable name to reference.

### Declaring and Initializing Arrays:

Arrays can be declared in one of two ways:

- (1) Declare the array, initialize later

```
int [ ] myArray = new int [10];  
double [ ] myDoubleArray = new double [15];
```

After declaration, you would either set the default values one at a time using the array subscripts...

```
myArray[0] = 0; myArray[1]=5; etc...
```

or if all values are initialized to the same value, you can use a for loop to initialize

```
for (int i=0; i < myArray.length; i++) {  
    myArray[ i ] = 0;  
}
```

- (2) Declare the array and initialize in one step

```
int [ ] myArray = {10, 20, 30, 40, 50}; <= declares an array with 5 elements
```

### Arrays of Objects:

As demonstrated above, you indicate what data type is to be stored in your array. You can even make an array of a custom storage class like you saw created earlier in this unit.

To create an array of the Employee object used earlier:

```
Employee [ ] myEmployees = new Employee [10];
```

This would create an array that is capable of storing information about 10 employees. Before you can put values into each element, you must loop through the array and call the constructor for each element.

```
for (int i=0; i < 10; i++ ) {  
    myEmployees[ i ] = new Employee( );  
}
```

To access the attributes stored within each element, you would use the following code:

```
myEmployees[0].setID(50);
```

This will go to position 0 of the array and use the mutator method for ID. An ID value of 50 will be stored in this array position to be accessed later.

```
String empFirstName = myEmployees[9].getFirstName();
```

This will go to the last position of the array and use the accessor method for the FirstName stored at this position and return the value to the program to be used.