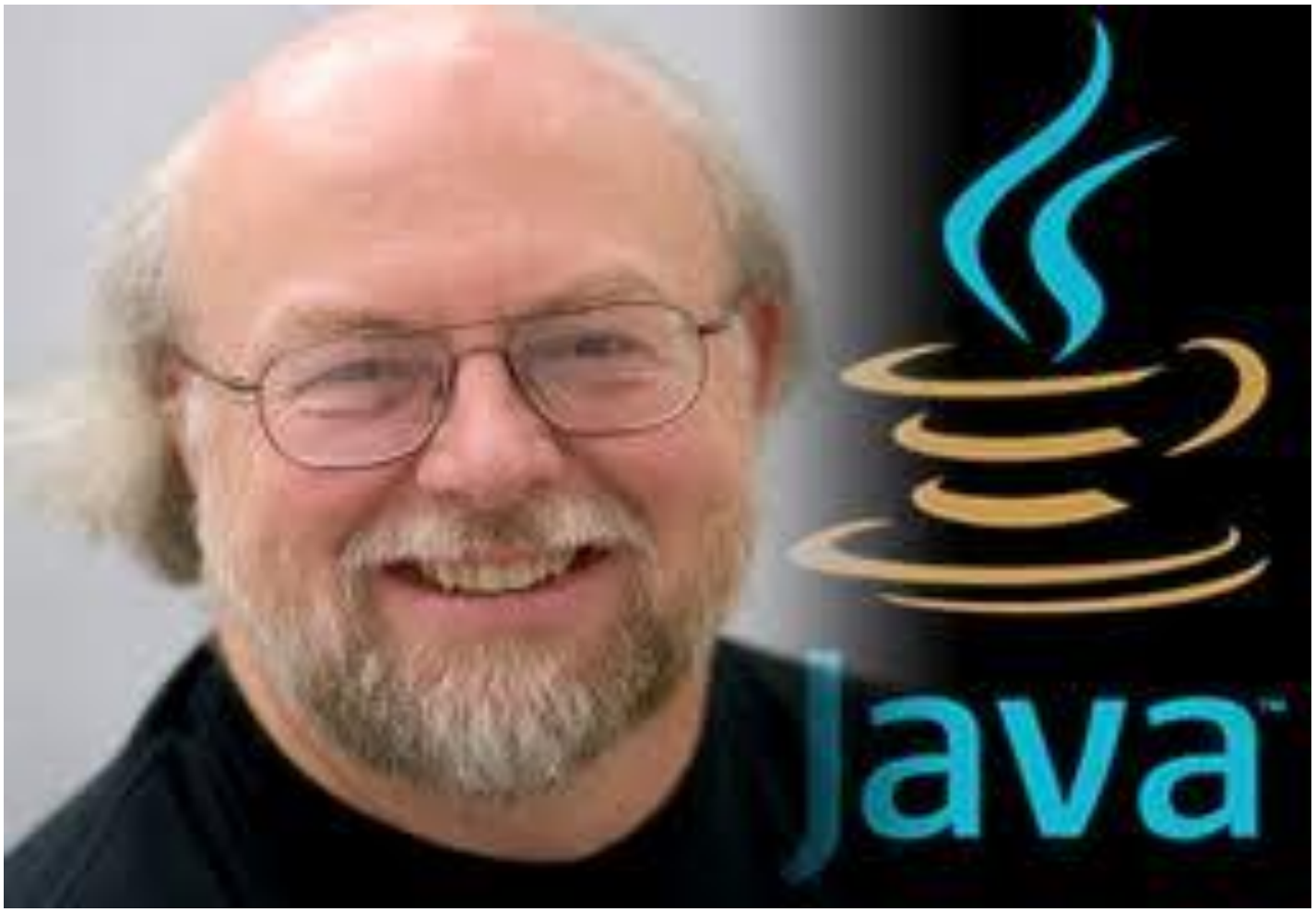
Saladin Minhaaj
Java Mentoring Session - 02
03/12/2013



One misconception about main( ) method

While you start learning programming with Java, you may think that all the classes need main ( ) method to run. Then you figure out that you were wrong! Not all the classes need main( ) method, some of them do. But the question is- which ones? I'll try to explain the deal with main( ) method.

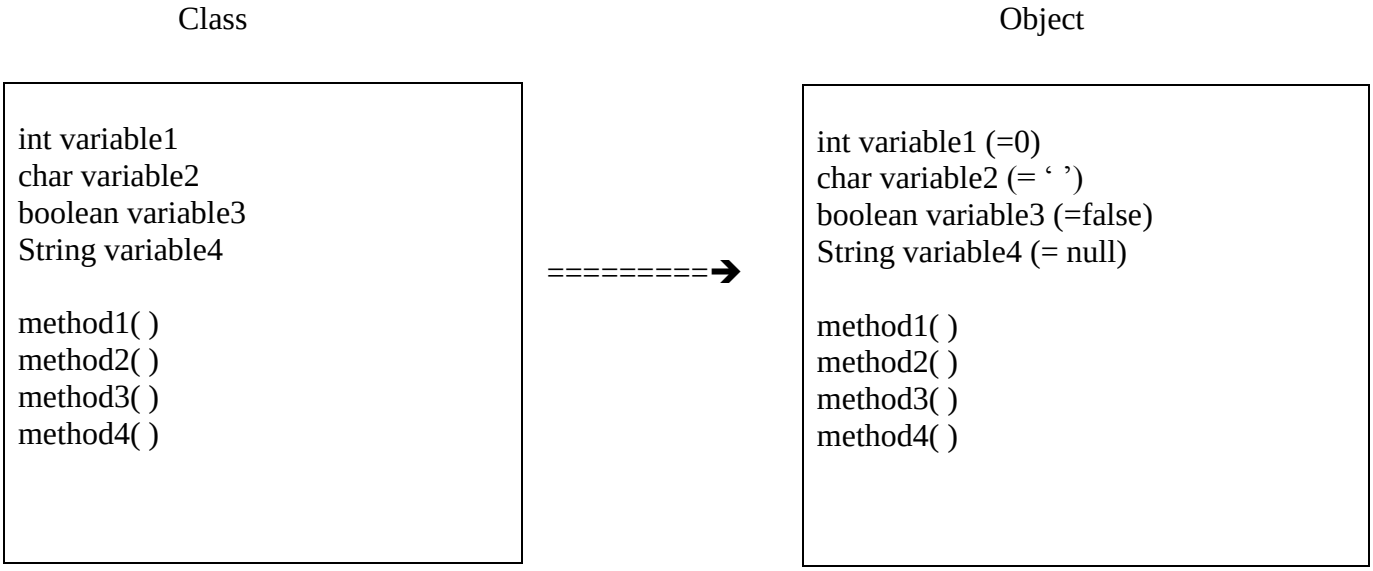
We must use main( ) method to run our Java program. It's mandatory; we shouldn't have any doubt about that(at least right now, while learning Core Java). When we'll start learning Selenium, we'll use testing frameworks ("what the heck are those!" We'll explore later, bare with me now) to run our Selenium tests (which are actually Java programs), without using main( ) method. But in core Java programming, we CANNOT avoid main( ) method.

Now come to the relation between main( ) method and class. In one Java program, we need one main( ) method. We can write this main( ) method under any class. But we should avoid this practice. Since main( ) method is very important in running a program, we should create a separate class just to write this main( ) method. Some programmers call this class "Driver class", since this class drives the entire program. Other synonyms are "Main class" and "Test class".

In summary, not all the classes need main( ) method, only one class in a program needs main( ) method.

Constructors

I said in the mentoring session that, whenever we create an object from a class, that object gets copies of all the variables and methods of that class



If we don’t assign any value to those variables inside the class, after instantiating, Java will assign default values to the copies of object’s variables. We can assign values ourselves. Last session, I mentioned 2 ways (Check page 9). There is another way to assign values to the object’s own variables, which is more efficient than those 2 previously mentioned—by using Constructors.

Constructors are special kind of methods, whose only function is to initialize an object i.e. initializing the object’s own variables. Let’s get back to the real life scenario from last session- “Saladin Computer Services, LLC”.

We will create a Computer class, and declare 4 variables:

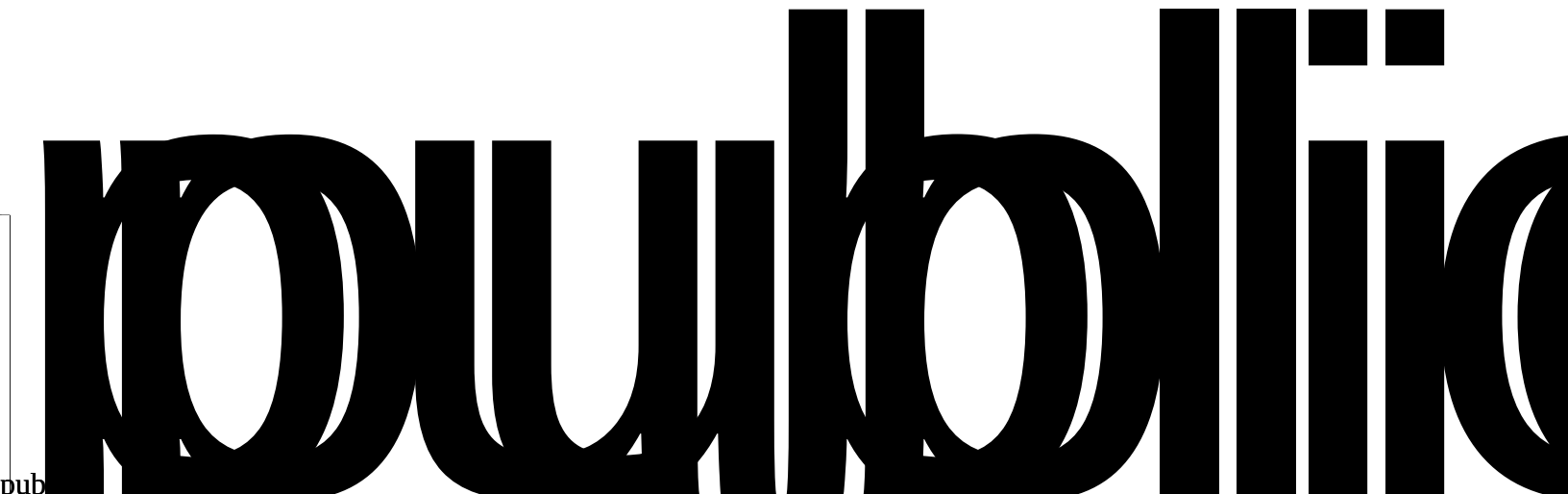
```
public class Computer{

int ram;
String processor;
char series;
boolean windows;

}
```

Now, if I want to choose the values of all of the object's variables while instantiating, I can create a constructor like this:

publicpublicpublicpublicpublicpublicpublic

```
public Computer(int ram, String processor, char series, boolean windows){
```

[redacted]



Principles of Object Oriented Programming

All the Object oriented programming languages (OOP) follow several principles. Often programmers' opinions vary regarding the number, definition and scope of the principles, but there are 4 OOP principles that all the programmers and programming gurus agree upon. In the 2nd mentoring session of Java class, I talked about these 4 principles and how Java applies those principles in its syntax, and how we can use those principles for efficient programming.

These 4 principles are:

1. Encapsulation
2. Inheritance
3. Polymorphism
4. Abstraction

Encapsulation:

In programming languages (not only Java), the concept of Encapsulation refers to two related but distinct principles:

1. Data Hiding: mechanism for restricting access to some of the object's components
2. Data Bundling: mechanism that facilitates the bundling of data with the methods operating on that data

[Data Hiding is a standard term, where Data Bundling is my term, don't use this term in the interview !!!] Some programmers use first principle, some both while defining Encapsulation. In Java, data hiding is achieved by Access Specifiers, data bundling is achieved by Classes. I already talked about these two mechanisms in the last mentoring session, and by the time you read this note, you must me comfortable with Access specifiers and Classes. Therefore, I'll skip talking about Encapsulation in detail.

Inheritance:

The process by which one object acquires the properties of another object is called Inheritance. It supports the concept of hierarchical classification. Suppose, I want to make a HPComputer class in my program. HPComputer will contain all the variables and methods defined in Computer class, and some additional variables and methods.

```
public class HPComputer{

int ram;
String processor;
char series;
boolean windows;
String model;

}
```

Notice that I have declared one extra variable, model, which is unique to HPComputer class.

Now, instead of creating HPComputer class from scratch, I can simply make a link between Computer and HPComputer class, so that HPComputer class will receive all the variables and methods from Computer class. In this way, we can reuse Computer class as many times as we want. In Java, this process is called Inheritance. HPComputer class will inherit all the variables and methods from Computer class. The syntax is:

```
public class HPComputer extends Computer{
```

Here, "extends" is a keyword. We use this keyword to indicate Inheritance. In Java, one class can inherit at most one class. Java doesn't support multiple inheritance, unlike some other languages. The class that we are inheriting from is called Parent class or Base class or Super class. The class that inherits from Parent class is called Child class or Derived class or Sub class. In our example, Computer is a super class, and HPComputer is a subclass.

After HPComputer inherits all the members of Computer class, we no longer need to declare all the variables and methods, we just need to declare any additional member, or any modified member.

We can add any number of variables or methods with ease, but we must be careful in modifying members. We should avoid modifying variables. After inheritance, if you declare the same variable in the subclass, that variable will hide the functionality of the same name variable in the super class. This process is called Variable Shadowing.

```
public class HPComputer extends Computer{

int ram;

}
```

In this case, we declared an integer type variable, ram, in HPComputer class, even after extending Computer class. This ram variable will shadow the Computer class's ram variable. Variable Shadowing is a poor programming practice. We SHOULD avoid this in our programs. We can modify the methods easily, and we should do this. This process of modifying methods will take us to the next topic- Polymorphism.

Polymorphism:

Polymorphism is a feature that allows one interface to be used for a general class of actions. In Java, we can apply polymorphism in two ways:

1. Method Overloading
2. Method Overriding

1. Method Overloading: Method overloading means changing the parameter list of the method while keeping the same name. Suppose, we have declared a method in Computer class to print the receipt for customer.

```
public void printReceipt(){
          System.out.println("Saladin Computer Services, LLC");
          System.out.println("Thank you for your purchase");
          System.out.println("Total price = $800");
     }
```

This method has an empty parameter list. Now, some customer will want to add their names in their receipts. In that case, we can add one parameter to this method.

```
public void printReceipt(String name){
          System.out.println("Saladin Computer Services, LLC");
          System.out.println("Thank you for your purchase Mr./Ms." + name);
          System.out.println("Total price = $800");
     }
```

Some other customers will want to add date in their receipt. We have to add one more parameter then.

```
public void printReceipt(String name, int date){
          System.out.println("Saladin Computer Services, LLC");
          System.out.println("Thank you for your purchase Mr./Ms." + name);
          System.out.println("Today's date: " + date);
          System.out.println("Total price = $800");
     }
```

This process of changing the parameter list of the same method is called Method Overloading. One point to notice that, Method Overloading depends only on method's name and parameter list. It doesn't depend on the access specifier or return type. While overloading, we MUST use the same name and different parameter list. If we don't do that, it'll be a completely new method, which we don't want to. We can change the access specifier or return type. Method overloading can be used in same class or any subclass.

2. Method Overriding: Method Overriding means changing the implementation (body) of a method in any subclass, while keeping the same name and parameter list. Method overriding MUST be done in a subclass, not in the same class where it's declared. Be careful!

```
public class HPComputer extends Computer{

	String model;

	@Override
	public void printReceipt(String name, int date){
		System.out.println("Saladin Computer Services, LLC");
		System.out.println("Thank you for your purchase Mr./Ms." + name);
		System.out.println("Today's date: " + date);
		System.out.println("Total price = $800");
		System.out.println("Have a wonderful day!!!");
	}
```

Here, in HPComputer class, we have overridden printReceipt(String name, int date) method. We've added two extra lines. Look at the @Override annotation. We should use this annotation while overriding method, because in case we make any error while overriding, Eclipse will give a warning sign.

Now, suppose I want to make a HP tablet. So, in my Java program, I have to create a HPTablet class. I will create this class by extending the HPComputer class, and I'll override the printReceipt( ) method again.

```
public class HPTablet extends HPComputer{

   public void printReceipt(String name, int date){
       System.out.println("Thank you for your HP tablet purchase Mr./Mrs."+ name);
       System.out.println("Today is " + date);
        System.out.println("Total price: $700");
       System.out.println("Come back again !!!");
     }
```

You can ask, since we are changing the body of the printReceipt( ) method of the Computer class anyway, so why do we need to write the body of the Computer class. This question will take us to the next topic- Abstraction.

Abstraction:

Abstraction is the process that allows only showing the essential features of the object to the end user, hiding the non-essential details. In our example, we don't need to write the body of the printreceipt( ) method in Computer class, we can simply declare that method, with "abstract" keyword.

```
public abstract void printReceipt(String name, int date);
```

In Java, abstraction means implementation hiding. More generally, incomplete. So, an abstract method means an incomplete method. Therefore, after extending the class, we MUST complete the method i.e. write the body of the method.

If a class has at least one abstract method, we must declare the class as abstract. An abstract class means an incomplete class. We cannot create an object out of an abstract class. At first, we have to create another class by extending the abstract class, write the body of all the abstract methods, and only then we can instantiate the class. In our case, we have to write as follows.

```
public abstract class Computer {
```

Interface

If we want to make all the methods of a class abstract, we can use a Java Interface. In simple English, interface means point of contact. In Java, the basic idea is same. An interface defines only the necessary functionalities of an object. An interface contains only abstract methods, and variables (which are implicitly static and final). In our program, we can create an interface named ComputingMachine.

```
public interface ComputingMachine {

	public int addNumber();
	public int subtractNumber();
	public void displayResult();

}
```

This interface specifies all the necessary functionalities of a computing machine. We can use this interface in our Computer class.

```
public abstract class Computer implements ComputingMachine{
```

To use an interface in any class, we use the keyword “implements”. We can implement as many interface as we want.

Please contact me if you have any question or comment:

Saladin Minhaaj
akms.minhaaj@gmail.com
(646) 420-2509