

5. Defining Your Own Classes

Part1 (2/2)

3 Sep 2015

Objectives

- Define a class with custom constructor.
- Distinguish private and public methods.
- Distinguish private and public data members.
- Understand the advantage of information hiding.
- Define class constants.
- Differentiate the local and instance variables.

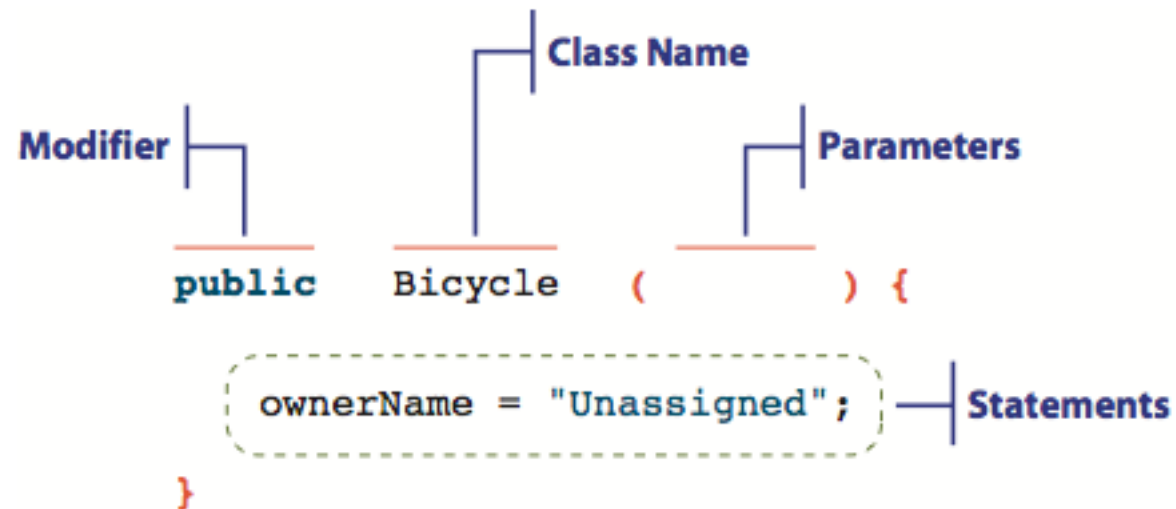
Constructors

- Constructor is a special method that is executed when a new instance of the class is created, that is, when the 'new' operator is called. It follows the general syntax

```
public <class name> ( <parameters> ) {  
    <statements>  
}
```

Constructors

- The components in the general syntax in Bicycle() constructor:

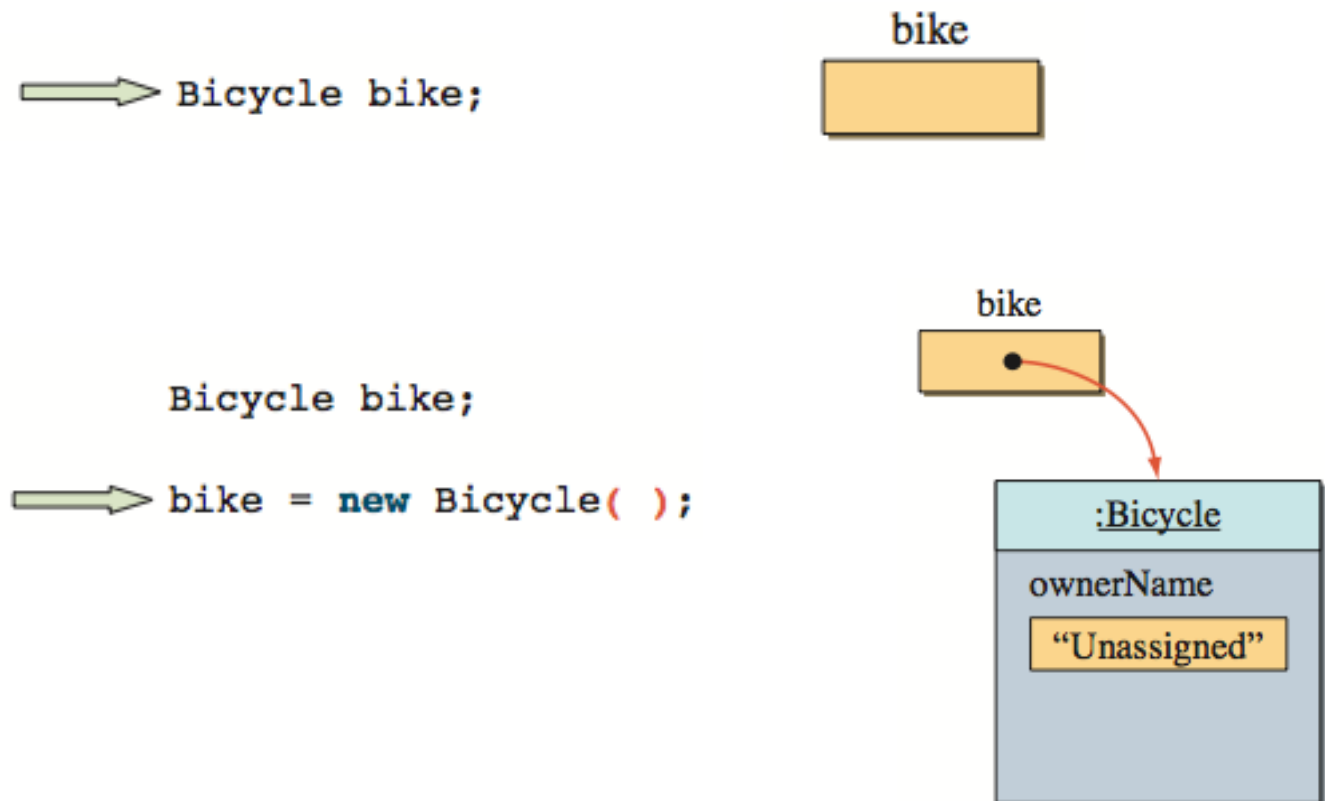


Constructors

- Notice that a constructor does not have a return type and, consequently, will never include a return statement.
- The modifier of a constructor does not have to be public, but non-public constructors are rarely used.
- The purpose of the constructor is to initialize the data member of new instance.

Constructors

- A sequence of state-of-memory diagrams after executing the constructor



Constructors

- Account Class

```
class Account {  
    // Data Members  
    private String ownerName;  
    private double balance;  
    //Constructor  
    public Account( ) {  
        ownerName = "Unassigned";  
        balance = 0.0;  
    }  
}
```

Constructors

- Account Class (cont.)

```
//Adds the passed amount to the balance
public void add(double amt) {
    balance = balance + amt;
}

//Deducts the passed amount from the balance
public void deduct(double amt) {
    balance = balance - amt;
}
```


Constructors

- Account Class (cont.)

```
//Returns the current balance of this account
public double getCurrentBalance( ) {
    return balance;
}

//Returns the name of this account's owner
public String getOwnerName( ) {
    return ownerName;
}
```

Constructors

- Account Class (cont.)

```
//Sets the initial balance of this account
public void setInitialBalance(double bal) {
    balance = bal;
}

//Assigns the name of this account's owner
public void setOwnerName(String name) {
    ownerName = name;
}
}
```

Constructors

- It is logically inconsistent to initialize the starting balance more than once. There is no such Java language feature that puts constraints on the number of times the `setInitialBalance()` method can be called.

```
Account acct;  
acct = new Account( );  
  
acct.setInitialBalance(500);  
acct.setInitialBalance(300);
```

Constructors

- This problem can also be solved by the new version of constructor that sets the initial balance to a specified amount.

```
public Account(double startingBalance) {  
    ownerName = "Unassigned";  
    balance = startingBalance;  
}
```

Constructors

- After the old constructor is replaced by this new constructor, we must create an instance by passing one argument when calling the new operator and now we can delete setInitialBalance() method.

```
Account acct;  
acct = new Account(500.00);
```

Constructors

- After the old constructor is replaced, we can no longer create an instance by writing codes below because there is no matching constructor anymore.

```
Account acct;  
acct = new Account( );
```

Constructors

- Instead of this one-parameter constructor, we can define a constructor that accepts the name of the owner also.

```
public Account(String name, double startingBalance) {  
    ownerName = name;  
    balance = startingBalance;  
}
```

Constructors

- With this two-parameter constructor, here's how we create an Account object:

```
Account acct;  
acct = new Account("John Smith", 500.00);
```


Constructors

- From the three different constructors possible for the Account class, we have selected the two-parameter constructor to include in the class.
- It is possible to include all three constructors in the definition of the Account class.
- We are now ready to list the complete definition. Here's the second version of the Account class, we will use AccountVer2 to avoid confusion.

Constructors

- AccountVer2 Class

```
class AccountVer2 {  
    // Data Members  
    private String ownerName;  
  
    private double balance;  
  
    //Constructor  
    public AccountVer2(String name, double startingBalance) {  
        ownerName = name;  
        balance = startingBalance;  
    }  
}
```

Constructors

- AccountVer2 Class (cont.)

```
//Adds the passed amount to the balance
public void add(double amt) {
    balance = balance + amt;
}
```

```
//Deducts the passed amount from the balance
public void deduct(double amt) {
    balance = balance - amt;
}
```

```
//Returns the current balance of this account
public double getCurrentBalance( ) {
    return balance;
}
```

Constructors

- AccountVer2 Class (cont.)

```
//Returns the name of this account's owner
public String getOwnerName( ) {

    return ownerName;
}

//Assigns the name of this account's owner
public void setOwnerName(String name) {

    ownerName = name;
}
}
```

Constructors

- Ex. Element Class

```
class Element {  
    //Data Members  
    private String name;  
    private int    number;  
    private String symbol;  
    private double mass;  
    private int    period;  
    private int    group;
```

Constructors

- Ex. Element Class (cont.)

```
//Constructor
public Element (String elementName, int elementNumber,
                String elementSymbol, double elementMass,
                int elementPeriod, int elementGroup) {

    name    = elementName;
    number  = elementNumber;
    symbol   = elementSymbol;
    mass     = elementMass;
    period   = elementPeriod;
    group    = elementGroup;
}
```

Constructors

- Ex. Element Class (cont.)
 - The following sample code creates three Element objects:

```
Element e1, e2, e3;  
  
e1 = new Element ("Hydrogen", 1, "H", 1.008, 1, 1);  
e2 = new Element ("Gold", 79, "Au", 197.0, 6, 11);  
e3 = new Element ("Oxygen", 8, "O", 16.0, 2, 16);
```

Default Constructor

- We strongly recommend to include constructors to classes. However, it is not a requirement to define a constructor in a class. If no constructor is defined, then the Java compiler will automatically include a 'default constructor' or 'empty constructor'.

Default Constructor

- For example, if we omit a constructor from the Bicycle class, a default constructor will be added to the class by the compiler to ensure its instances can be created.

```
public Bicycle( ) {  
}
```

Default Constructor

- Once we define our own constructor, no default constructor is added. This means that once the constructor, such as code below is added to the Account class

```
public Account(String name, double startingBalance ) {  
    ownerName = name;  
    balance = startingBalance;  
}
```

Default Constructor

we will no longer be able to create a Account object anymore by executing code below because no matching constructor can be found in the class.

```
Account acct;  
acct = new Account( );
```

Information Hiding and Visibility Modifiers

- The modifiers 'public' and 'private' designate the accessibility, or visibility, of data members and methods.
- From the object-oriented design standpoint, we recommend that you always designate the 'data members as private' and 'methods as public'.

Information Hiding and Visibility Modifiers

- Consider a robot as an example. Behaviors of robot such as moving forward, turning, stopping, and changing speed come to mind easily.
- When we define a class, Robot class, we will include public methods such as `move()`, `turn()`, `stop()`, and `changeSpeed()`. These methods are declared public so the programmers who use a Robot class can call these methods from their programs.

Information Hiding and Visibility Modifiers

- There are some private methods because they are internal details that need to be hidden from the programmers who use a Robot class.
- When we call its move() method. We do not care what's going on inside. This is called 'information hiding'. We say the mobile robot 'encapsulates' the internal workings.

Information Hiding and Visibility Modifiers

- Ex. We modify the Account class so the add() and deduct() methods call the private method adjust(). The adjust() method is hidden from the programmers

```
class AccountVer4 {  
    ...  
  
    //Adds the passed amount to the balance  
    public void add(double amt) {  
        adjust(amt);  
    }  
}
```

Information Hiding and Visibility Modifiers

-

```
//Deducts the passed amount from the balance
public void deduct(double amt) {
    adjust( -(amt+FEE) );
}
...

//Adjusts the account balance
private void adjust(double adjustAmt) {
    balance = balance + adjustAmt;
}
}
```


Information Hiding and Visibility Modifiers

- Why declaring data members public is considered a bad design, let's consider the AccountVer2 class. Suppose its data member balance is declared as public:

```
class AccountVer2 {  
    public double balance;  
    //the rest is the same  
}
```

Information Hiding and Visibility Modifiers

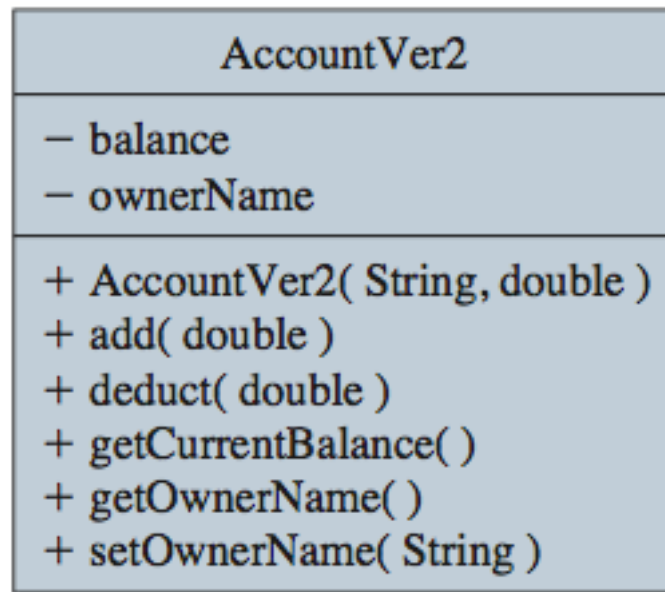
If this were the class definition, we could not prohibit programmers from writing code such as

```
AccountVer2 myAcct;  
myAcct = new AccountVer2("John Smith", 300.00);  
myAcct.balance = 670.00;
```

This breaks the AccountVer2 class because the 'balance' can be modified directly by the programmers.

Information Hiding and Visibility Modifiers

- We use the plus symbol '+' for public and the minus symbol '-' for private in class diagram.



Class Constants

- In this section we will show how a class constant is declared. A class constant will be shared by all methods of the class.
- Using 'final' to declare constant and 'static' to declare class data member
- Let's define another version of the Account class (the actual name will be AccountVer3). This time we will charge a fixed fee whenever a deduction is made.

Class Constants

- AccountVer3 Class


```
class AccountVer3 {
```

```
    // Data Members
```

```
    private static final double FEE = 0.50;
```

```
    private String ownerName;
```

```
    private double balance;
```



Class constant
declaration

Class Constants

- AccountVer3 Class (cont.)

```
//Constructor
public AccountVer3(String name, double startingBalance) {

    ownerName = name;
    balance = startingBalance;
}

//Deducts the passed amount from the balance
public void deduct(double amt) {
    balance = balance - amt - FEE;
}
```



Fee is charged
every time

Class Constants

- DeductionWithFee Class (Main Class)

```
import java.text.*;

class DeductionWithFee {
    //This sample program deducts money three times
    //from the account

    public static void main(String[] args) {
        DecimalFormat df = new DecimalFormat("0.00");
        AccountVer3 acct;
```

Class Constants

- DeductionWithFee Class (Main Class) (cont.)

```
acct = new AccountVer3("Carl Smith", 50.00);

acct.deduct(10);
acct.deduct(10);
acct.deduct(10);
System.out.println("Owner: " + acct.getOwnerName());
System.out.println("Bal   : $"
                    + df.format(acct.getCurrentBalance()));
    }
}
```


Class Constants

- DeductionWithFee Class (Main Class) (cont.)
 - Output:

```
Owner: Carl Smith  
Bal : $18.50
```

Class Constants

- The modifier 'final' designates that the identifier FEE is a constant, and the modifier 'static' designates that it is a class constant.
- The reserved word 'static' is used to declare class components, such as class variables and class methods.

```
private static final double FEE = 0.50;
```

Class Constants

- This declaration is not an error, but it is inefficient. If FEE is declared without the static modifier, then it is an instance constant. This means every instance of the class will have its own copy of the same value.

```
class AccountVer3 {  
    private final double FEE = 0.50;  
    //the rest is the same  
}
```

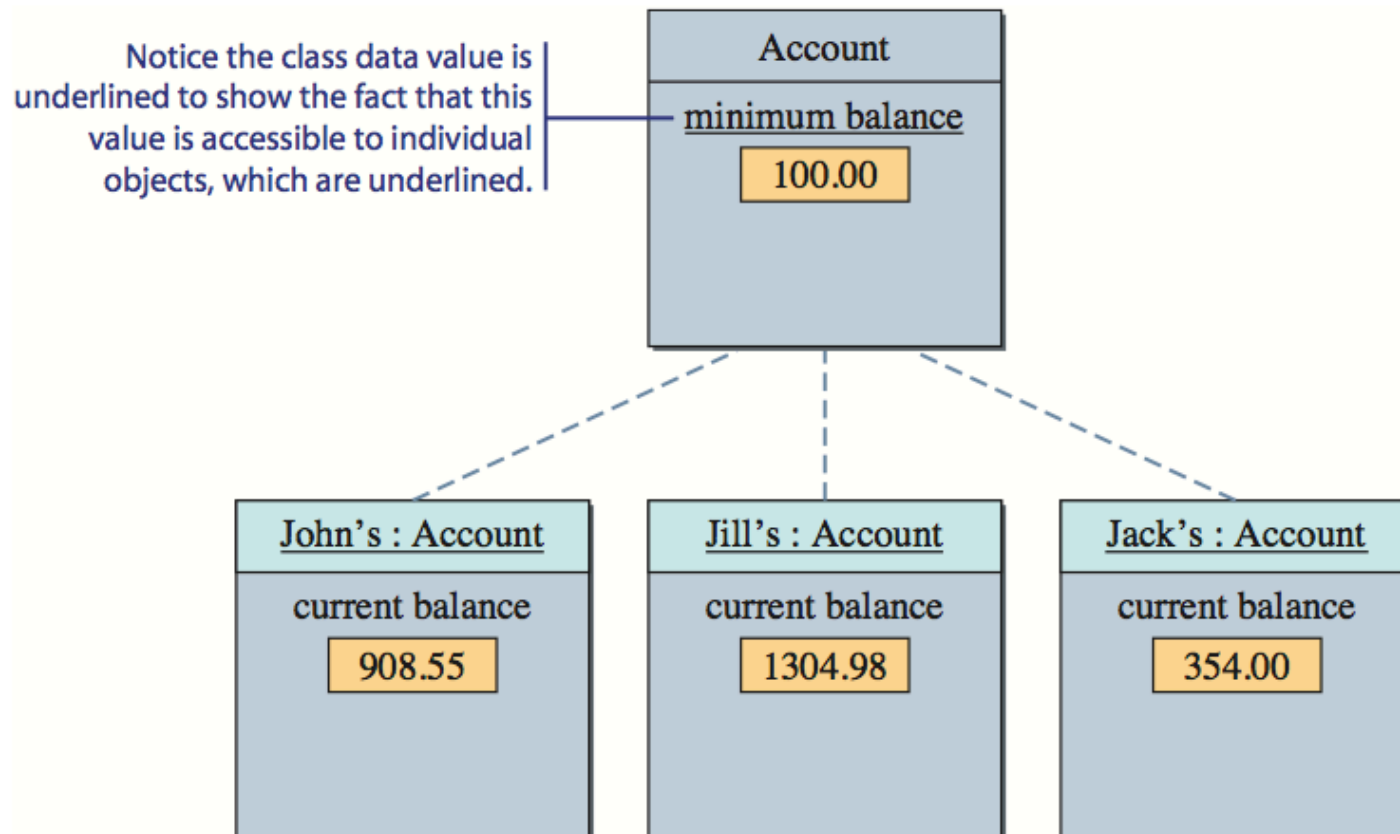
Class Constants

- From week1 slide: Information shared by all instances instance should be 'static' data member
 - Non-static Data Member (minimum balance)

| <u>John's : Account</u> | <u>Jill's : Account</u> | <u>Jack's : Account</u> |
|---------------------------|----------------------------|---------------------------|
| current balance 908.55 | current balance 1304.98 | current balance 354.00 |
| minimum balance 100.00 | minimum balance 100.00 | minimum balance 100.00 |

Class Constants

- Static Data Member (minimum balance)



Class Constants

- Ex. Die Class

```
import java.util.Random;
class Die {

    //Data Members

    //the largest number on a die
    private static final int MAX_NUMBER = 6;

    //the smallest number on a die
    private static final int MIN_NUMBER = 1;

    //To represent a die that is not yet rolled
    private static final int NO_NUMBER = 0;
```

Class Constants

- Ex. Die Class (cont.)

```
private int number;  
  
private Random random;  
  
//Constructor  
public Die() {  
    random = new Random();  
  
    number = NO_NUMBER;  
}
```

Class Constants

- Ex. Die Class (cont.)

```
//Rolls the die
public void roll( ) {
    number = random.nextInt (MAX_NUMBER - MIN_NUMBER + 1) + MIN_NUMBER;
}

//Returns the number on this die
public int getNumber( ) {
    return number;
}
}
```


Class Constants

- Ex. Die Class (cont.)
 - Main Class

```
class RollDice {  
  
    //Simulates the rolling of three dice  
    public static void main(String[] args) {  
  
        Die one, two, three;  
  
        one    = new Die( );  
        two    = new Die( );  
        three  = new Die( );  
    }  
}
```

Class Constants

- Ex. Die Class (cont.)

```
one.roll();
two.roll();
three.roll();

System.out.println("Results are " + one.getNumber( ) + " " +
                    two.getNumber( ) + " " +
                    three.getNumber( ) );
}
```

Class Constants

- Ex. Die Class (cont.)
 - Output:

```
Results are 3 6 5
```

Public Constants

- We stated that data members should be declared 'private'. But we may want to declare class constants as 'public' because a constant is “read only”, so it is illegal to assign new value to constant data member.

```
class AccountVer3 {  
    public static final double FEE = 0.50;  
    ...  
}
```

Public Constants

- In main() method can then access this information directly as

```
System.out.println("Fee charged per deduction is $ "  
                    + AccountVer3.FEE);
```

- Class data members are accessed by the syntax

```
<class name> . <class data members>
```

Public Constants

- Ex. In java.lang.Math, there are 2 constants

```
public final class Math {  
    ...  
    public static final double E = 2.7182818284590452354;  
    public static final double PI = 3.14159265358979323846;  
    ...  
}
```

```
public static void main(String[] args) {  
    double radius = 10;  
    double area = Math.PI * Math.pow(radius, 2);  
    System.out.println("area = " + area);  
}
```

Local Variables

- In this section we will talk about scope of local variables.
- Consider the deduct method of the Account class:

```
public void deduct(double amt) {  
    balance = balance - amt;  
}
```

Local Variables

We can rewrite the method, using a local variable, as follows:

```
public void deduct(double amt) {  
    double newBalance;  
    newBalance = balance - amt;  
    balance = newBalance;  
}
```



This is a **local variable**

Local Variables

- The variable 'newBalance' is called a 'local variable'. They are declared within the method declaration and used for temporary purposes, such as storing intermediate results of a computation.

Local Variables


- While the 'data members' of a class are accessible from all instance methods of the class, 'local variables' are accessible only from the method in which they are declared, and they are available only while the method is being executed.

Local Variables


- It is acceptable to use the same identifier for a local variable and data member, but it is not advisable.

```
class Sample {  
    private int number;  
    ...  
    public void doSomething( ) {  
        int number;  
        number = 15;  
    }  
    ...  
}
```

This changes the value of the local variable, not the instance variable.



The same identifier is used for both the local variable and the instance variable.



Calling Methods of the Same Class

- We called a method of some object (eg. from main() method), we used dot notation, such as acct.deduct(12). Just as we can call a method of another object using:

`objectName.methodName(<parameter>)`

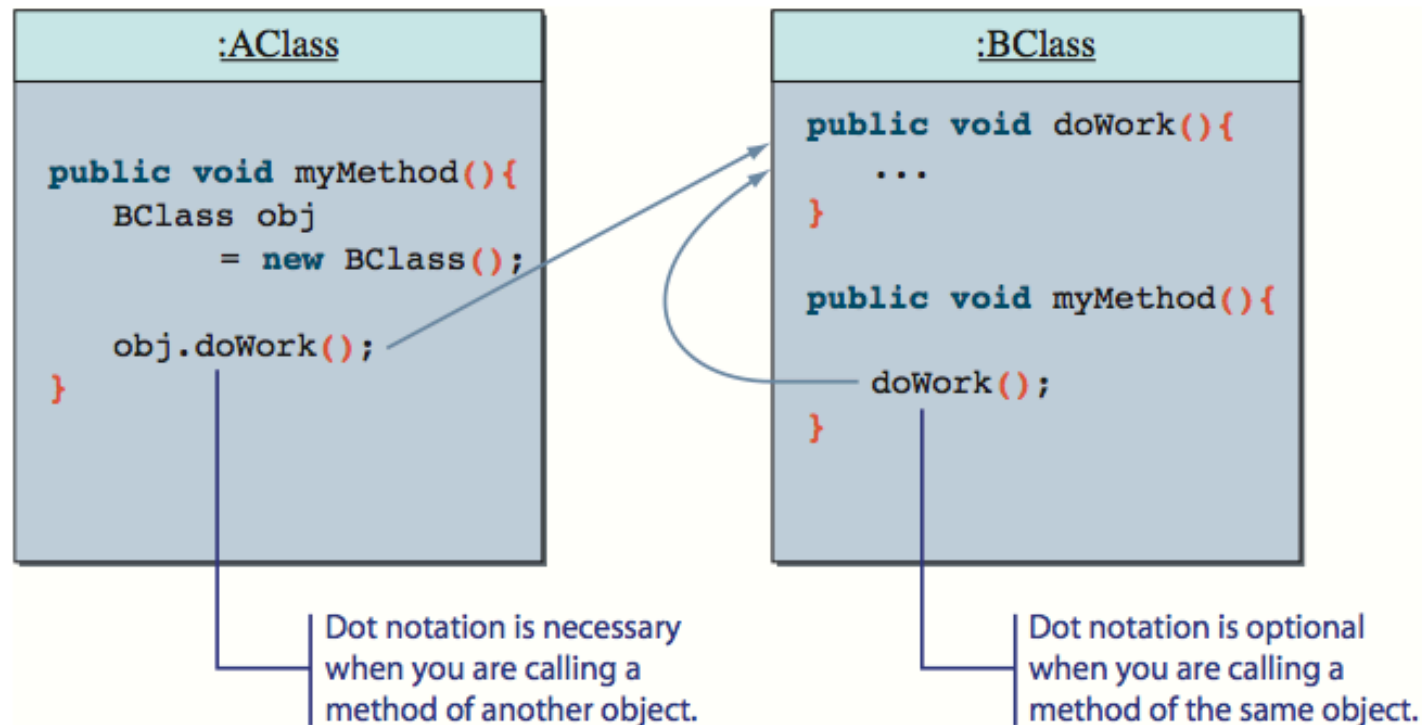
Calling Methods of the Same Class

- it is possible to call a method from a method of the same object using:

`methodName(<parameter>)`

Calling Methods of the Same Class

- The difference between calling a method belonging to the same object and a method belonging to a different object.



Calling Methods of the Same Class

- Ex. We modify the AccountVer3 class so the add() and deduct() methods call the private method adjust().

```
class AccountVer4 {  
    ...  
  
    //Adds the passed amount to the balance  
    public void add(double amt) {  
        adjust(amt);  
    }  
}
```

Calling Methods of the Same Class

-

```
//Deducts the passed amount from the balance
public void deduct(double amt) {
    adjust( -(amt+FEE) );
}
...

//Adjusts the account balance
private void adjust(double adjustAmt) {
    balance = balance + adjustAmt;
}
}
```


Calling Methods of the Same Class

In the modified class, we redefine the two methods so they call the common private method `adjust()`. We pass positive amount for the `add()` and pass negative amount for `deduct()` method

```
public void add(double amt) {  
    adjust(amt);  
}
```



Notice there is no dot notation. This is calling another method that belongs to the same class.

Calling Methods of the Same Class

- Ex. In the original Die class, when a new instance was created, we set its number to NO_NUMBER. we'll redefine the class so a die gets rolled when it is first created. The trick here is to call the roll method from the constructor.

Calling Methods of the Same Class

```
import java.util.Random;
class DieVer2 {

    //Data Members

    //the largest number on a die
    private static final int MAX_NUMBER = 6;

    //the smallest number on a die
    private static final int MIN_NUMBER = 1;

    private int number;

    private Random random;
```

Calling Methods of the Same Class

•

```
//Constructor
public DieVer2( ) {
    random = new Random( ) ;

    roll();
}

//Rolls the die
public void roll( ) {
    number = random.nextInt(MAX_NUMBER - MIN_NUMBER + 1) + MIN_NUMBER;
}

//Returns the number on this die
public int getNumber( ) {
    return number;
}
}
```

Changing Any Class to a Main Class

- In this section, we will show you a simple way to make any class (such as Bicycle) runnable. It is possible to define the `main()` method to a class so the class can run by itself.
- There are a number of advantages in doing this. First, for testing. Second, for show how to use the classes.

Changing Any Class to a Main Class

- Ex. We can define the main method to the Bicycle class.

```
class Bicycle {  
    // Data Member  
    private String ownerName;  
  
    //Returns the name of this bicycle's owner  
    public String getOwnerName( ) {  
        return ownerName;  
    }  
  
    //Assigns the name of this bicycle's owner  
    public void setOwnerName(String name) {  
        ownerName = name;  
    }  
}
```

Changing Any Class to a Main Class

-

```
//The main method that shows a sample
//use of the Bicycle class
public static void main(String[] args) {

    Bicycle myBike;

    myBike = new Bicycle( );
    myBike.setOwnerName("Jon Java");

    System.out.println(myBike.getOwnerName() +
                        "owns a bicycle");
}
}
```

Changing Any Class to a Main Class

- Ex. We can define the main method to the Dice class and run it using NetBeans.

```
public class Dice {  
    //data members  
    private static final int MAX_NUMBER = 1;  
    private static final int MIN_NUMBER = 6;  
    private Random random;  
  
    public Dice(){  
        random = new Random();  
    }  
}
```

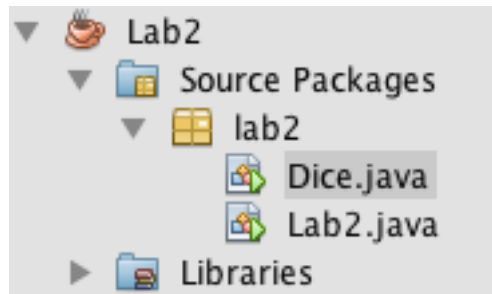

Changing Any Class to a Main Class

-

```
public static void main(String[] args) {  
    Dice d = new Dice();  
    System.out.println(d.roll());  
}  
  
public int roll( ) {  
    int number = random.nextInt (MAX_NUMBER - MIN_NUMBER + 1)  
                + MIN_NUMBER;  
    return number;  
}  
}
```

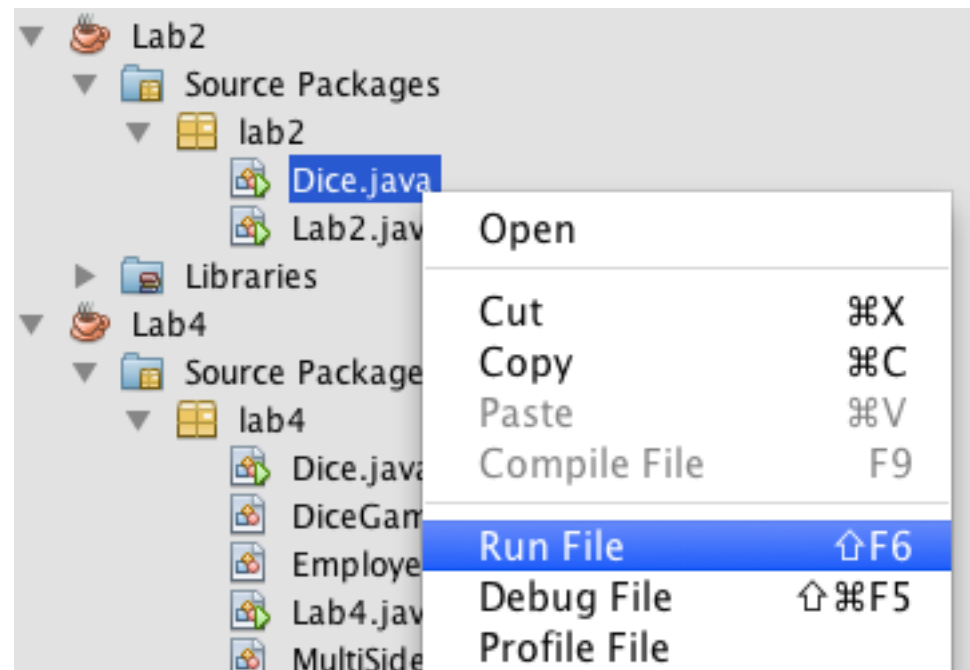
Changing Any Class to a Main Class

Runnable files will show file icon with small green play button



Changing Any Class to a Main Class

We can run by right click at this file then choose 'Run File' from pop-up menu



Summary

- A constructor is a special method that is executed when a new object is created. Its purpose is to initialize the object into a valid state.
- Memory space for local variables is allocated when a method is called and deallocated when the method terminates.
- Public methods define the behavior of an object.

Summary

- Private methods and data members are considered internal details of the class.
- Components (data members and methods) of a class with the visibility modifier 'private' cannot be accessed by the other classes.
- Components of a class with the visibility modifier 'public' can be accessed by the other classes.

Summary

- Dot notation is not used when you call a method from another method of the same class.
- Any class can be set as the main class of a program by adding the `main()` method to it.

Reference

- C. Thomas Wu, An Introduction to Object-Oriented Programming with Java, 5th Edition
 - Chapter 4: Defining Your Own Classes-Part 1

Question?