

# Designing Classes

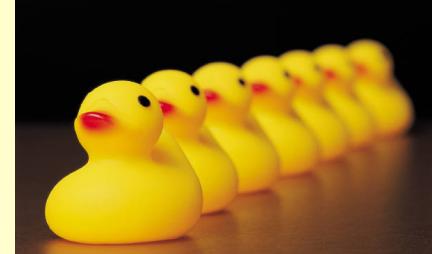
CSC 2014 – Java Bootcamp

Dr. Mary-Angela Papalaskari  
Department of Computing Sciences  
Villanova University

# Where do objects come from?



# Where do objects come from?



***Good question!***

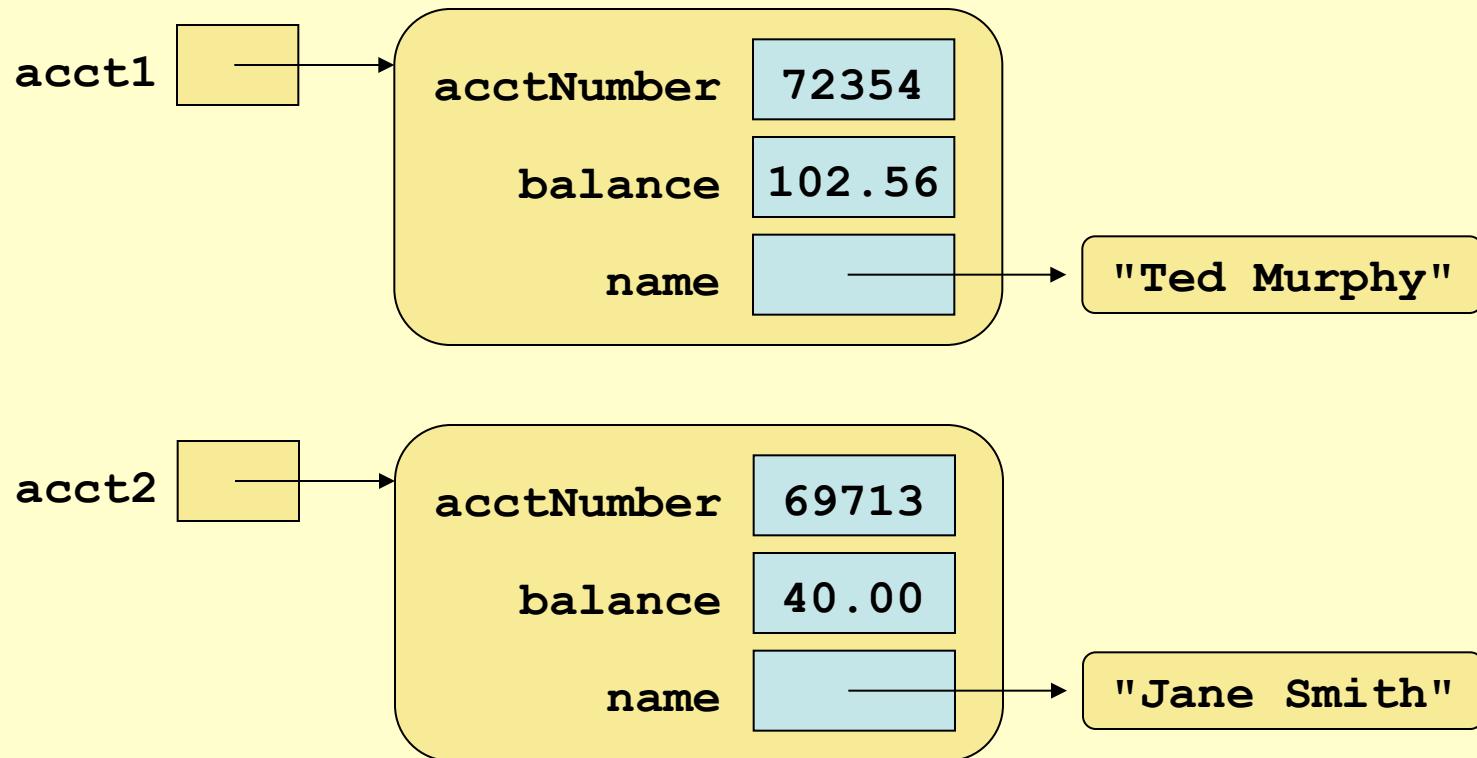
***We will learn how to create a class that defines a new datatype, i.e., a new type of objects***

**We need to learn:**

1. What is the structure of a class definition?
2. How to specify what happens when an object is instantiated (i.e., when the **new** operator is used)?
3. How do we define the methods that can be invoked through objects of this class?

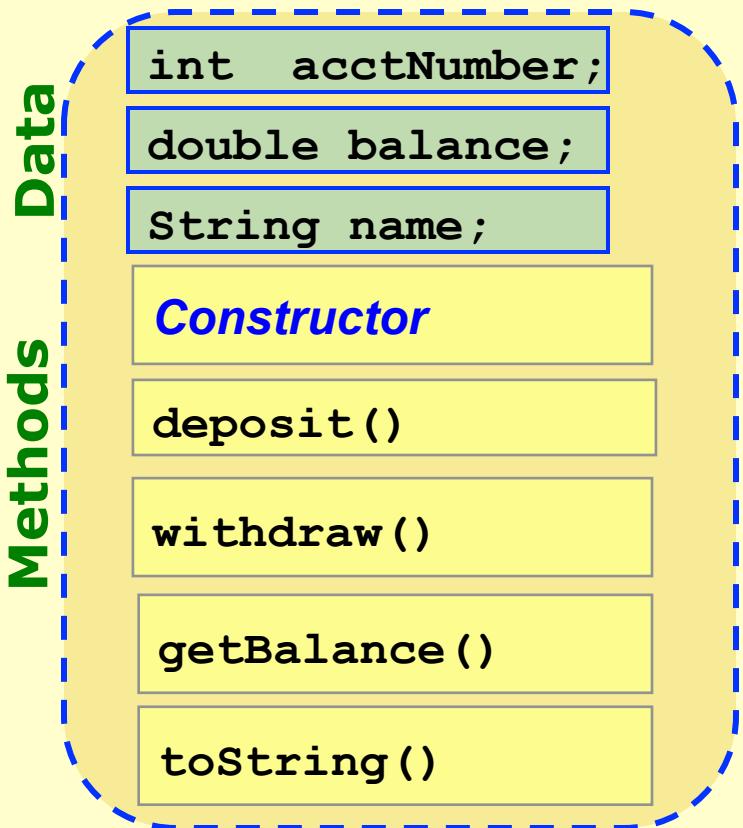
# Example: Account datatype

- represents a generic bank account

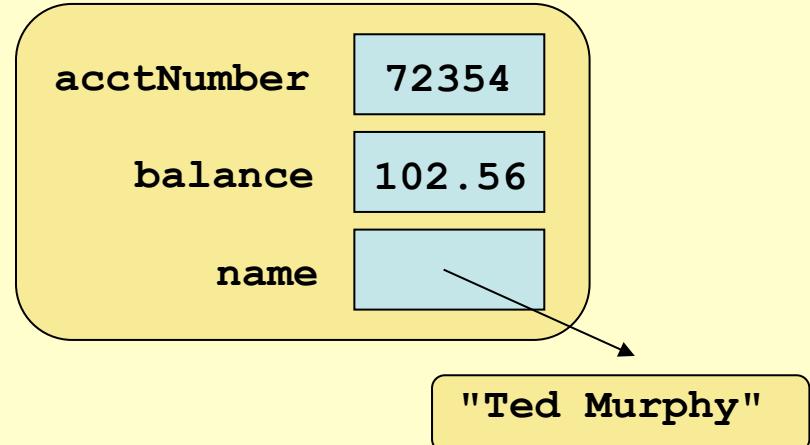


# 1. Structure of class definition

Account class



Account object



- The class is the blueprint  
Classes define DATA and METHODS  
i.e., a datatype
- The object:
  - is like the house built from the blueprint
  - is an instance of the class
  - has its own data space & shares methods defined for this datatype

## 2. Object instantiation

Creating Objects – old example:

- We have already seen something like this:

```
Scanner scan = new Scanner (System.in);
```



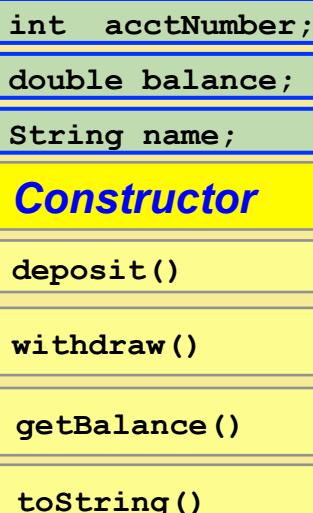
This invokes the **Scanner** ***constructor***, which is  
a special method that sets up the object

## 2. Object instantiation

Creating Objects – our newly defined **Account** class:

```
Account acct1 = new Account ("Ted Murphy", 72354, 102.56);
```

Invokes the **Account *constructor***, which is  
a special method that sets up the object

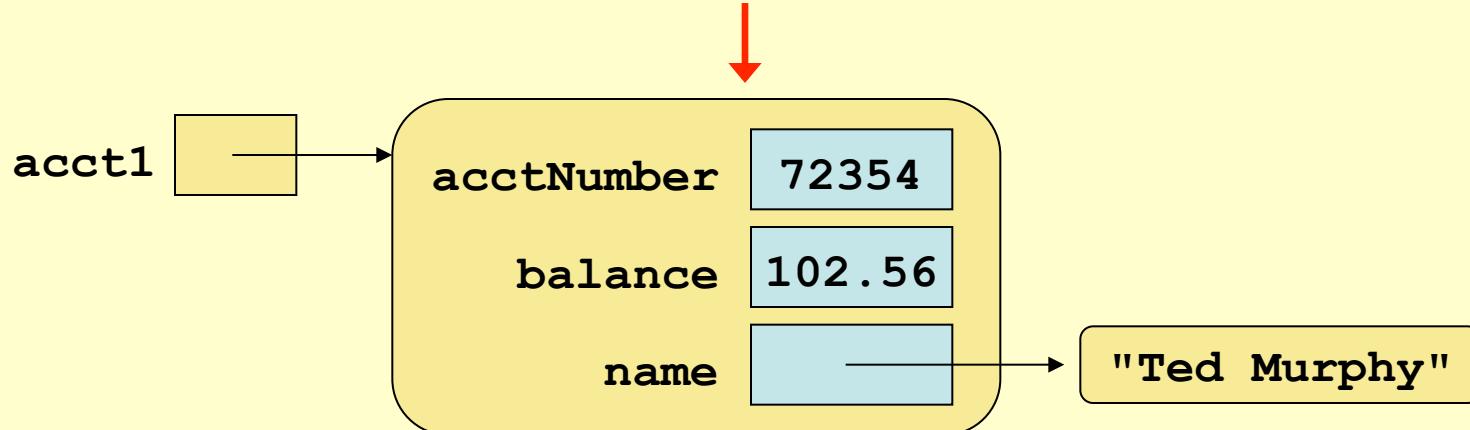


## 2. Object instantiation

Creating Objects – our newly defined **Account** class:

```
Account acct1 = new Account ("Ted Murphy", 72354, 102.56);
```

A new **Account** object is created!



# 3. Method invocation

- As we have seen, once an object has been created, we can use the ***dot operator*** to invoke its methods:

```
ans = scan.nextLine();
```

```
numChars = title.length();
```

# 3. Method invocation

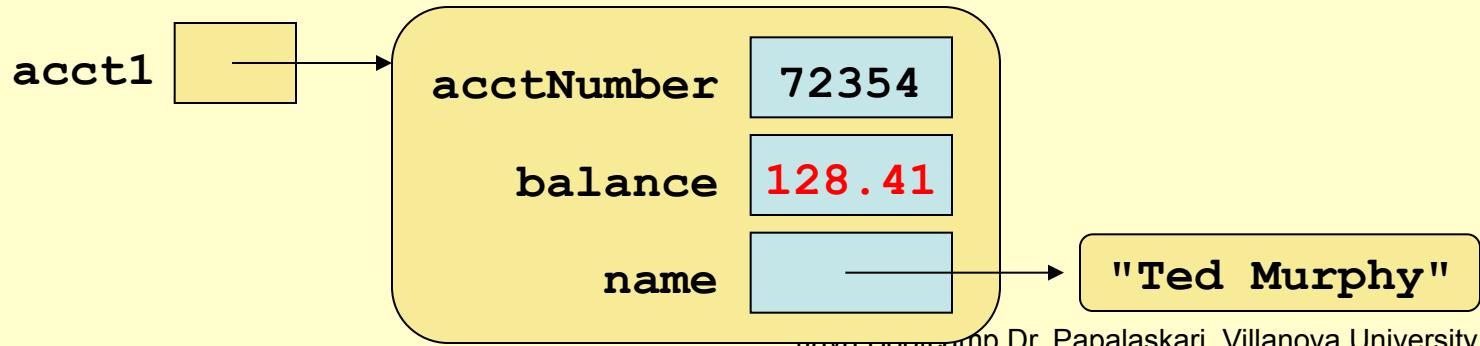
- As we have seen, once an object has been created, we can use the **dot operator** to invoke its methods:

```
ans = scan.nextLine();
```

```
numChars = title.length();
```

```
amount = acct1.getBalance();
```

```
acct1.deposit(25.85);
```



# Writing Classes

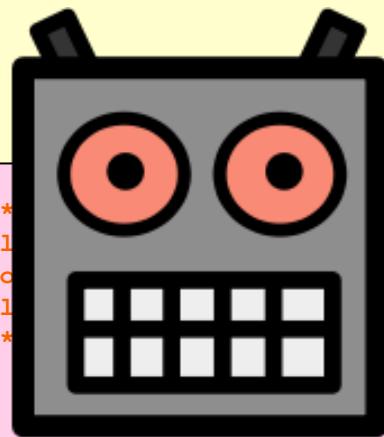
- So far, we've been using predefined classes from the Java API. Now we will learn to write our own classes.
  - class definitions
  - constructors
  - instance data
  - method declaration and parameter passing
  - encapsulation and Java modifiers
  - more about creating graphical objects (next week)

# Datatype / Client (also referred to as “ slave / driver” classes)

```
//*****  
// Account.java  
// Represents a bank  
//*****  
import java.text.NumberFormat;  
  
public class Account {  
    int acctNumber;  
    double balance;  
    String name;  
  
    //-----  
    // Sets up the account by defining its owner's name, account  
    // number, and initial balance.  
    //-----  
  
    public Account (String x, int y, double z)  
    {  
        name = x;  
        acctNumber = y;  
        balance = z;  
    }  
  
    int acctNumber;  
    double balance;  
    String name;  
  
    Constructor  
    deposit()  
    withdraw()  
    getBalance()  
    toString()  
}
```

## Account Datatype

```
*****  
// Transactions.java      Author: MA Papalaskari  
// (based on Java How  
// Demonstrates the creation and use of multiple  
*****
```



## Transactions Client

```
ns  
  
nk accounts and requests various services.  
-----  
main (String[] args)  
  
new Account ("Ted Murphy", 72354, 102.56);  
new Account ("Jane Smith", 69713, 40.00);  
new Account ("Edward Demsey", 93757, 759.32);  
n (acct1);  
n (acct2);  
n (acct3);  
6.85);  
50, 2.50);  
  
n ();  
n (acct1);  
n (acct2);  
n (acct3);
```

```
*****  
// Account.java          Author: Lewis/Loftus  
//                      Simplified code by MA Papalaskari  
// Represents a bank account with methods deposit and withdraw.  
*****
```

```
import java.text.NumberFormat;
```

```
public class Account
```

```
{
```

```
    int acctNumber;  
    double balance;  
    String name;
```

```
-----  
// Sets up the account by defining its owner's name, account  
// number, and initial balance.  
//
```

```
public Account (String x, int y, double z)
```

```
{
```

```
    name = x;  
    acctNumber = y;  
    balance = z;
```

```
}
```

```
-----  
// Deposits the specified amount x into the account.
```

```
-----
```

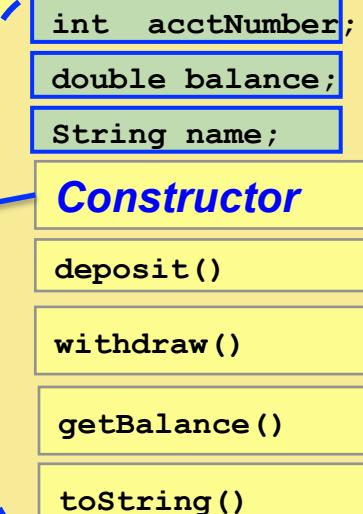
```
public void deposit (double x)
```

```
{
```

```
    balance = balance + x;
```

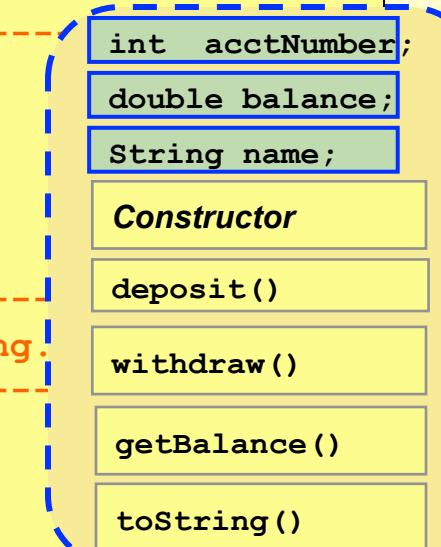
```
}
```

```
continue
```



continue

```
//-----  
// Withdraws the specified amount from the account and applies  
// the fee.  
//-----  
public void withdraw (double x, double fee)  
{  
    balance = balance - x - fee;  
}  
  
//-----  
// Returns the current balance of the account.  
//-----  
public double getBalance ()  
{  
    return balance;  
}  
  
//-----  
// Returns a one-line description of the account as a string.  
//-----  
public String toString ()  
{  
    NumberFormat fmt = NumberFormat.getCurrencyInstance ();  
    return (acctNumber + "\t" + name + "\t" + fmt.format(balance));  
}
```



```

//*****
// Transactions.java          Author: MA Papalaskari
//                               (based on Lewis/Loftus example)
// Demonstrates the creation and use of multiple Account objects.
//*****

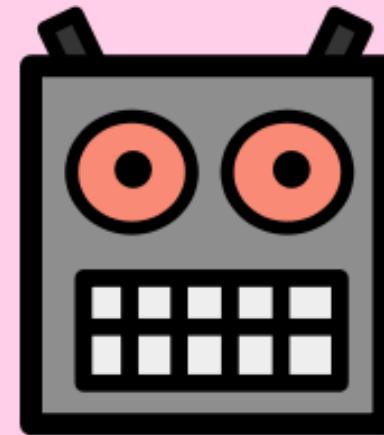

public class Transactions
{
    //-----
    // Creates some bank accounts and requests various services.
    //-----
    public static void main (String[] args)
    {
        Account acct1 = new Account ("Ted Murphy", 72354, 102.56);
        Account acct2 = new Account ("Jane Smith", 69713, 40.00);
        Account acct3 = new Account ("Edward Demsey", 93757, 759.32);

        System.out.println (acct1);
        System.out.println (acct2);
        System.out.println (acct3);

        acct1.deposit (25.85);
        acct1.withdraw (60, 2.50);

        System.out.println ();
        System.out.println (acct1);
        System.out.println (acct2);
        System.out.println (acct3);
    }
}

```



```

//*****Transactions.java***** Author: MA Papalaskari
// (based on Lewis/Loftus example)
// Demonstrates the creation and use of multiple Account objects.
//*****Transactions.java***** Author: MA Papalaskari
// (based on Lewis/Loftus example)
// Demonstrates the creation and use of multiple Account objects.

public class Transactions
{
    //-----
    // Creates some bank accounts and requests various services.
    //-----
    public static void main (String[] args)
    {
        Account acct1 = new Account ("Ted Murphy", 72354, 102.56);
        Account acct2 = new Account ("Jane Smith", 69713, 40.00);
        Account acct3 = new Account ("Edward Demsey", 93757, 759.32);

        System.out.println (acct1);
        System.out.println (acct2);
        System.out.println (acct3);

        acct1.deposit (25.85);
        acct1.withdraw (60, 2.50);

        System.out.println ();
        System.out.println (acct1);
        System.out.println (acct2);
        System.out.println (acct3);
    }
}

```

## Sample Run

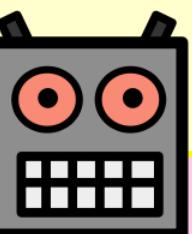
72354	Ted Murphy	\$102.56
69713	Jane Smith	\$40.00
93757	Edward Demsey	\$759.32
72354	Ted Murphy	\$65.91
69713	Jane Smith	\$40.00
93757	Edward Demsey	\$759.32

# Transactions class:

## Creating Account objects

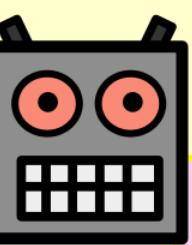
**Transactions** class

```
Account acct1 = new Account ("Ted Murphy", 72354, 102.56);
```



# Transactions class:

## Creating Account objects



Transactions class

```
Account acct1 = new Account ("Ted Murphy", 72354, 102.56);
```

int acctNumber;  
double balance;  
String name;

**Constructor**

deposit()

withdraw()

getBalance()

toString()

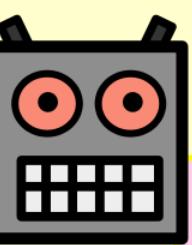
Account class

```
public Account (String x, int y, double z)  
{  
    name = x;  
    acctNumber = y;  
    balance = z;  
}
```

**Constructor method**

# Transactions class:

## Creating Account objects



Transactions class

```
Account acct1 = new Account ("Ted Murphy", 72354, 102.56);
```

int acctNumber;  
double balance;  
String name;

**Constructor**

deposit()

withdraw()

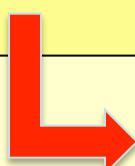
getBalance()

toString()

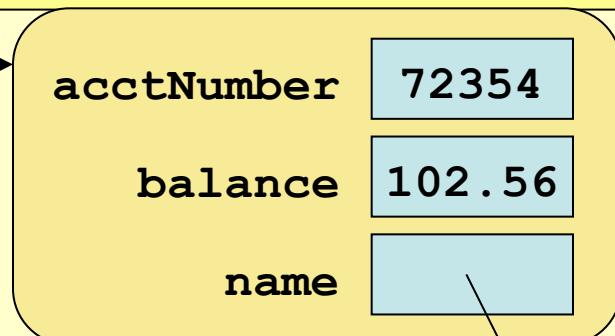
Account class

```
public Account (String x, int y, double z)  
{  
    name = x;  
    acctNumber = y;  
    balance = z;  
}
```

**Constructor method**



acct1



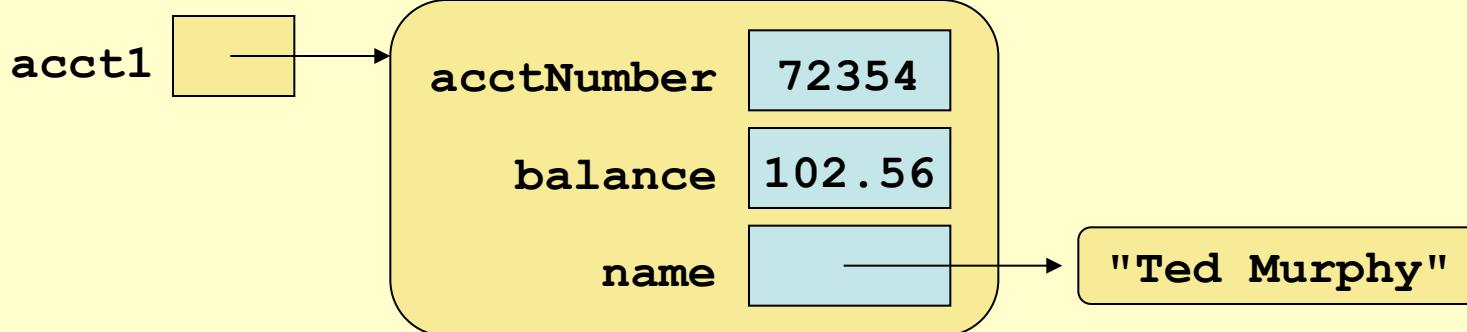
"Ted Murphy"

# Transactions class:

## Creating more Account objects

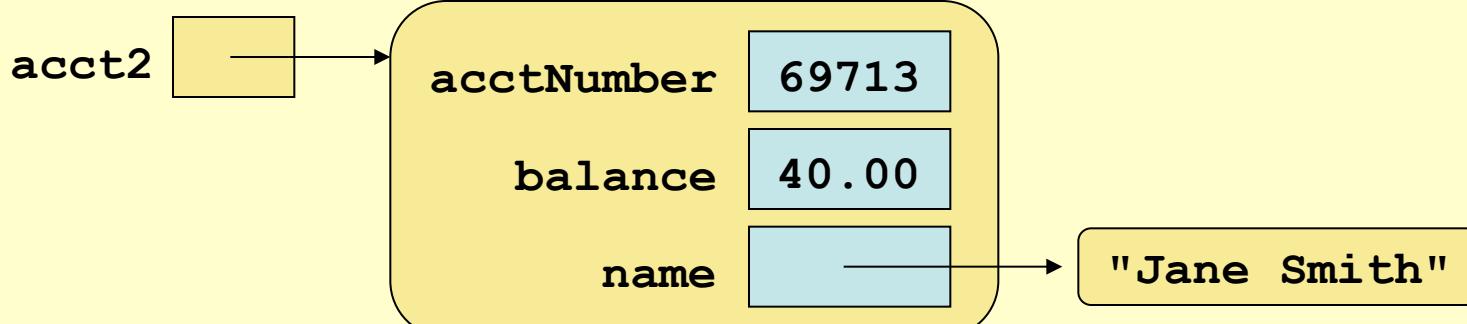
Transactions class

```
Account acct1 = new Account ("Ted Murphy", 72354, 102.56);
```



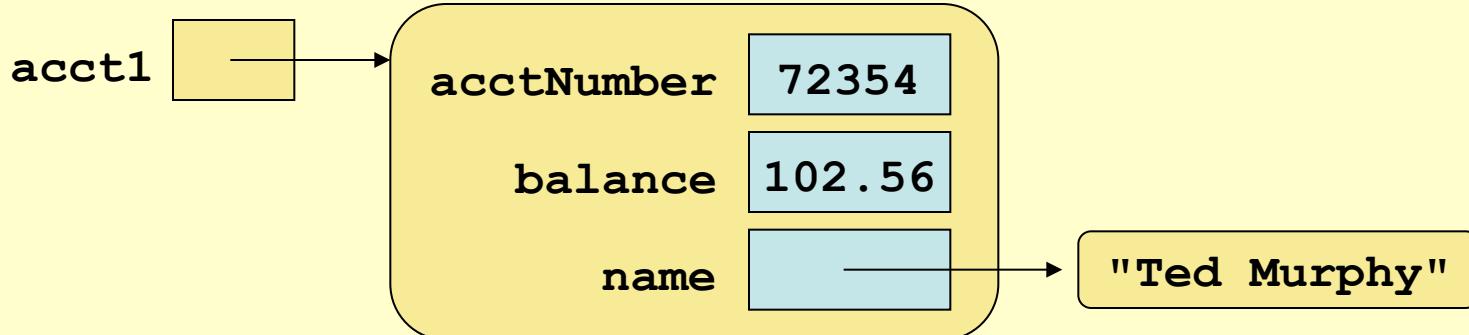
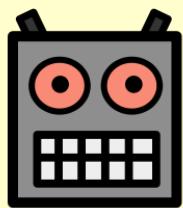
Transactions class

```
Account acct2 = new Account ("Jane Smith", 69713, 40.00);
```



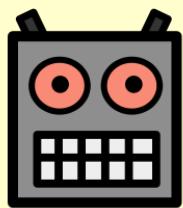
# Account class: Using methods

```
acct1.deposit (25.85);
```

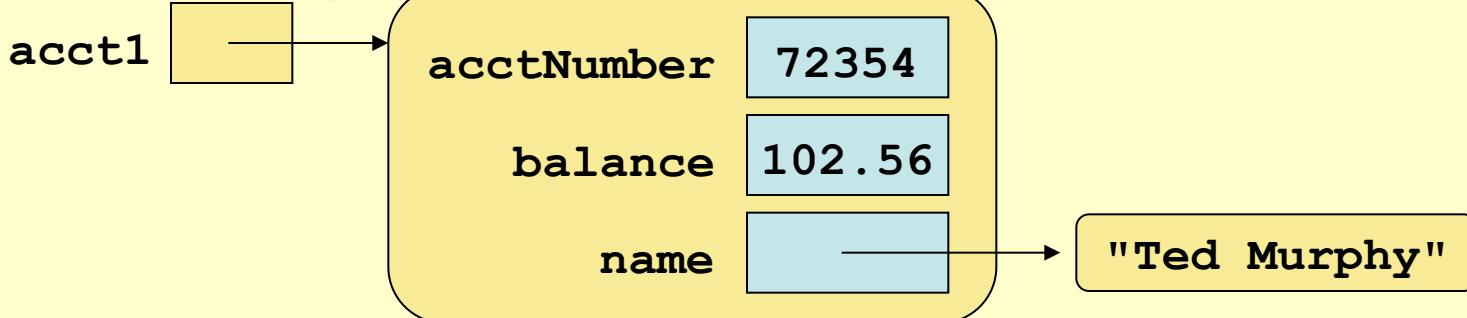


# Account class: Using methods

```
acct1.deposit (25.85);
```

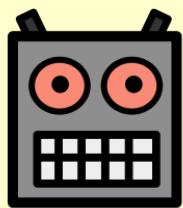


```
-----  
// Deposits the specified amount into the account.  
-----  
public void deposit (double x)  
{  
    balance = balance + x;  
}
```



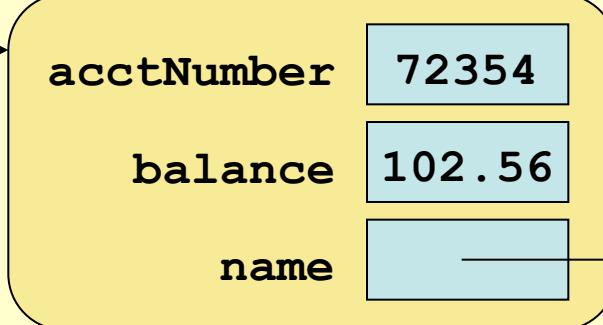
# Account class: Using methods

```
acct1.deposit (25.85);
```



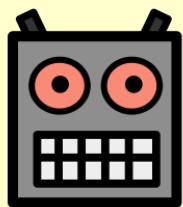
```
/*
 * Deposits the specified amount into the account.
 */
public void deposit (double x)
{
    balance = balance + x;
}
```

acct1

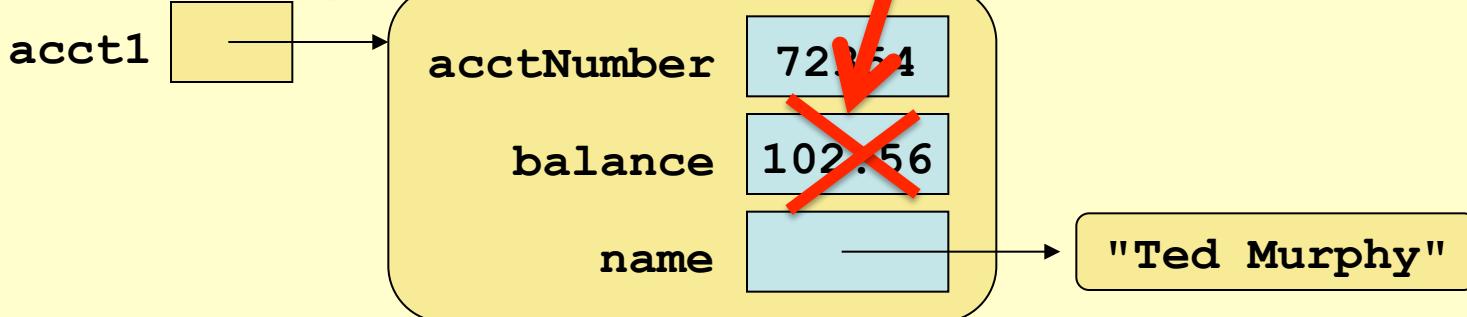


# Account class: Using methods

```
acct1.deposit (25.85);
```

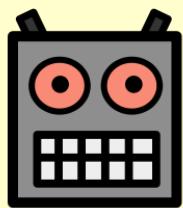


```
-----  
// Deposits the specified amount into the account.  
-----  
public void deposit (double x)  
{  
    balance = balance + x;  
}
```

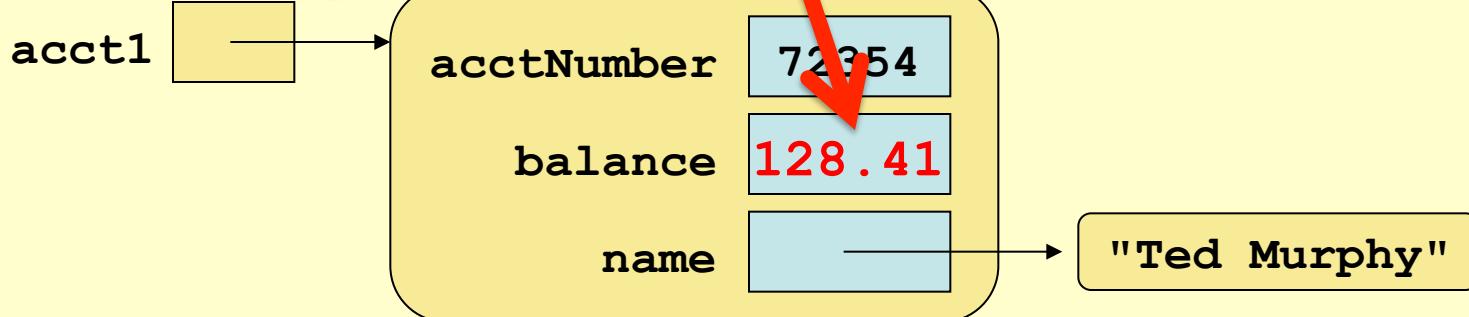


# Account class: Using methods

```
acct1.deposit (25.85);
```

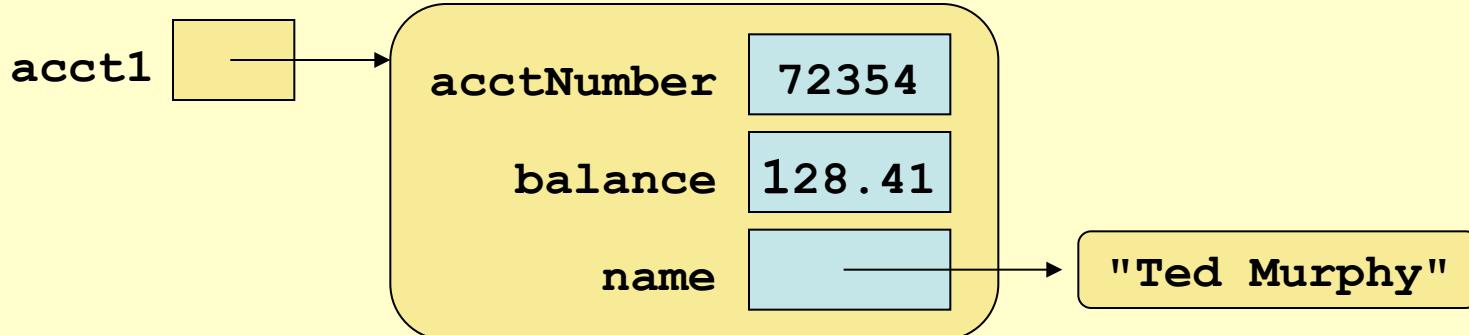
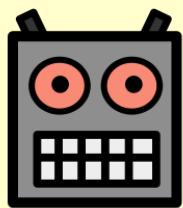


```
-----  
// Deposits the specified amount into the account.  
-----  
public void deposit (double x)  
{  
    balance = balance + x;  
}
```



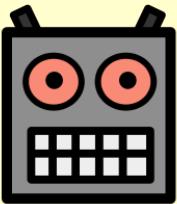
# Account class: Another Example

```
acct1.withdraw (60, 2.50);
```

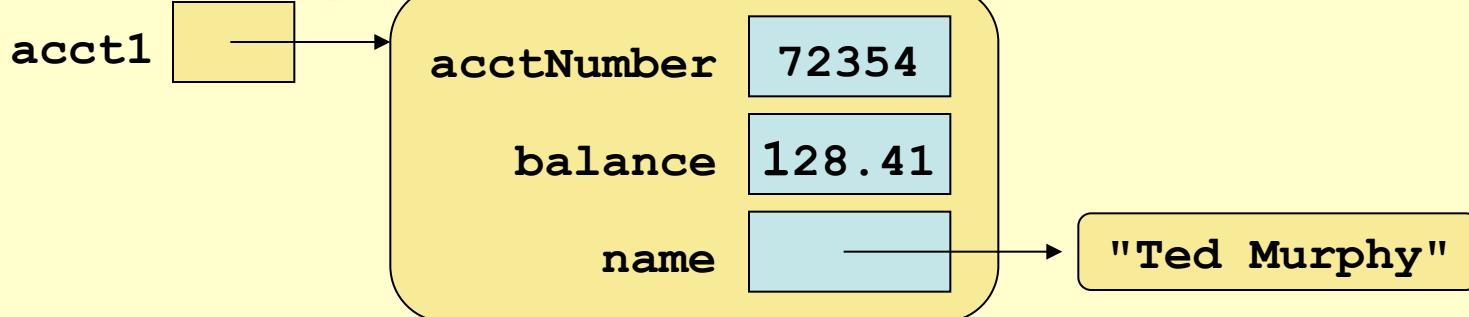


# Account class: Another Example

```
acct1.withdraw (60, 2.50);
```

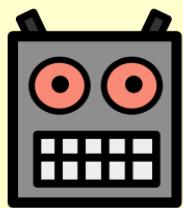


```
-----  
// Withdraws the specified amount from the  
// account and applies the fee.  
-----  
public void withdraw (double x, double fee)  
{  
    balance = balance - x - fee;  
}
```

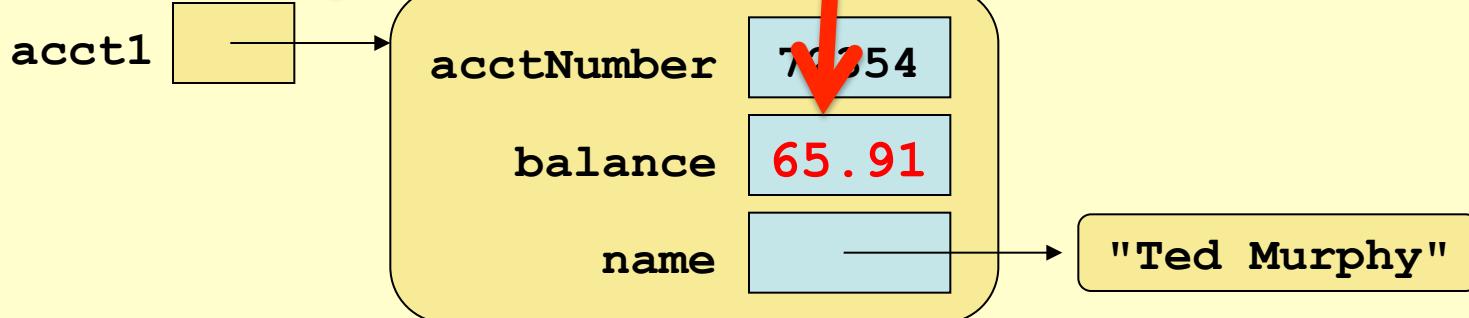


# Account class: Another Example

```
acct1.withdraw (60, 2.50);
```

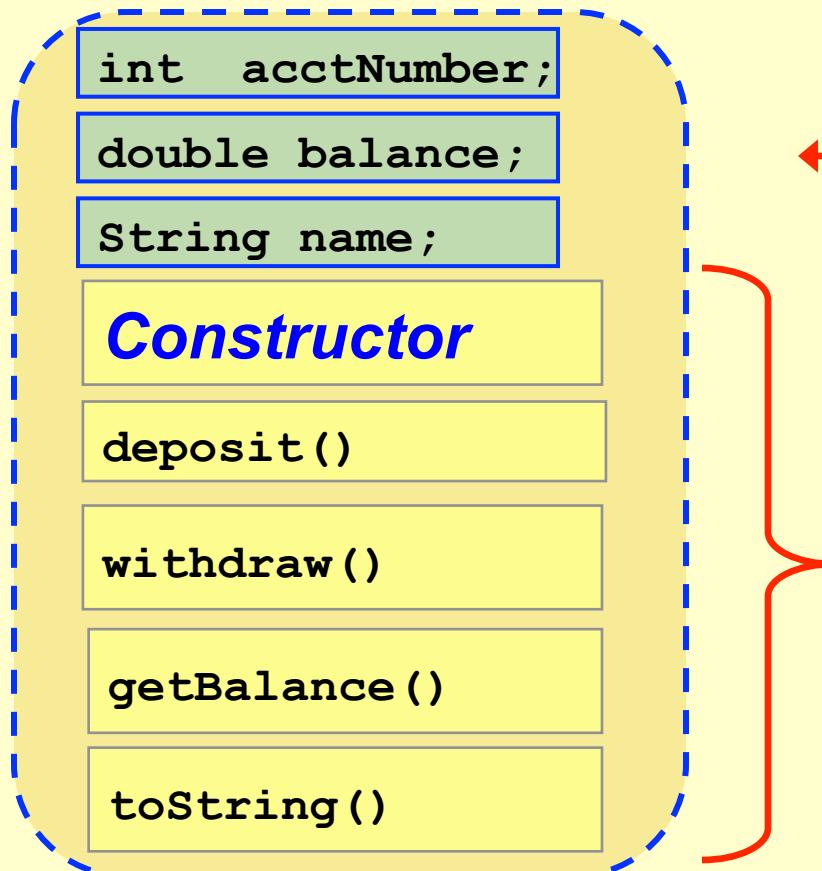


```
-----  
// Withdraws the specified amount from the account  
// and applies the fee.  
-----  
public void withdraw (double x, double fee)  
{  
    balance = balance - x - fee;  
}
```



# Class definitions

- A class can contain data declarations and method declarations

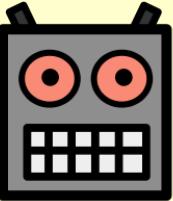


**Data declarations**  
(also called **fields**)

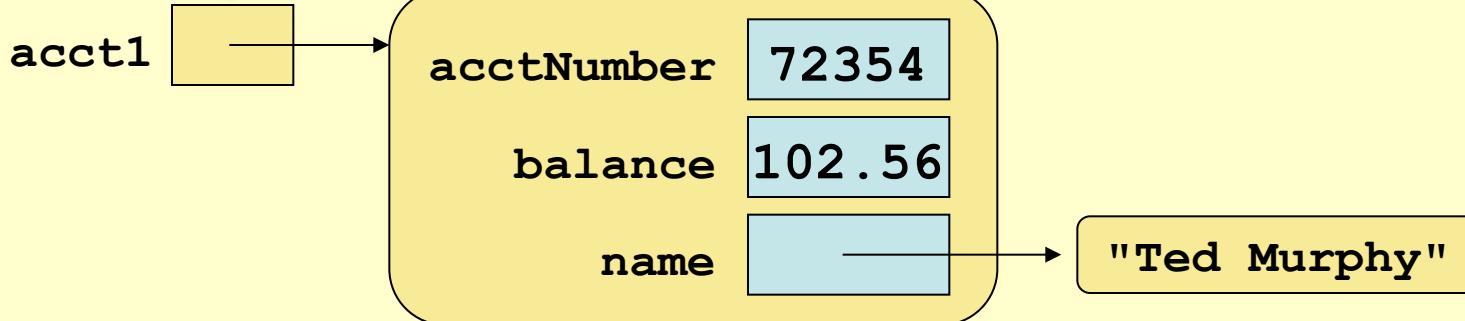
**Method declarations**  
(note: the constructor is  
also a method)

# **toString() method**

```
System.out.println(acct1.toString());
```



```
public String toString ()
{
    NumberFormat fmt = NumberFormat.getCurrencyInstance();
    return (acctNumber +"\t"+ name +"\t"+ fmt.format(balance))
}
```

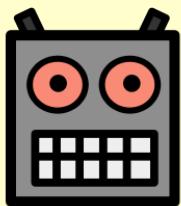


can be omitted!

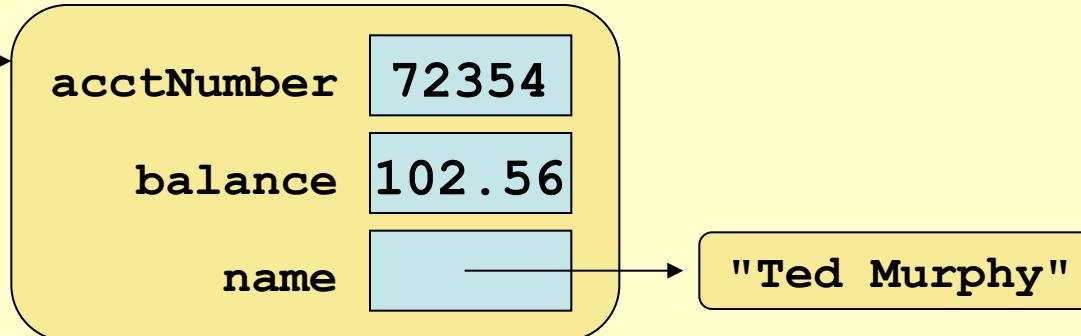
# toString() method

```
System.out.println(acct1.toString());
```

```
public String toString ()
{
    NumberFormat fmt = NumberFormat.getCurrencyInstance();
    return (acctNumber +"\t"+ name +"\t"+ fmt.format(balance))
}
```



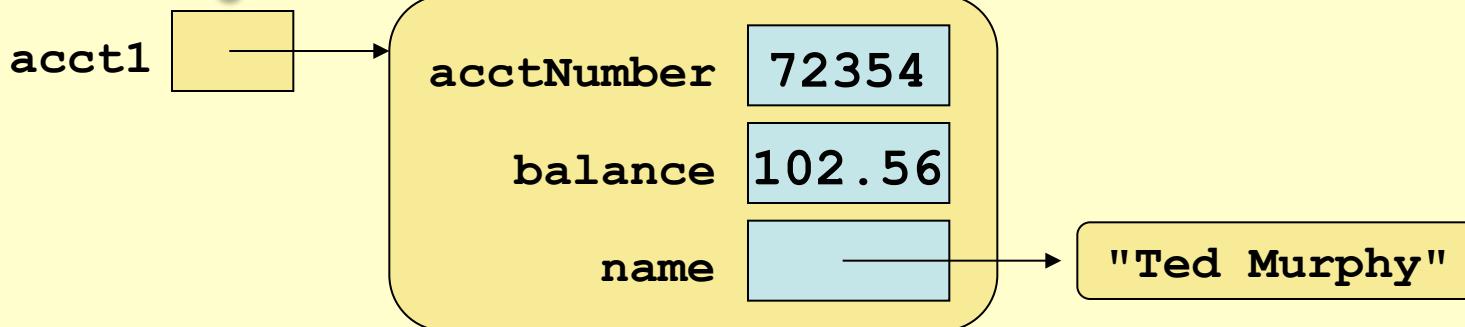
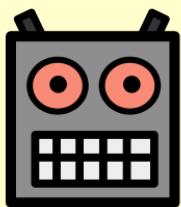
acct1



# *getBalance () method*

```
double amount = acct1.getBalance();  
// Note: this code is not used in Transactions.java
```

```
public double getBalance ()  
{  
    return balance;  
}
```



# Bank Account Example

- There are some improvements that can be made to the **Account** class
- The design of some methods could also be more robust, such as verifying that the **amount** parameter to the **withdraw()** method is positive
- Some of these improvements are in the book examples
- [Account.java](#), [Transactions.java](#) (simplified versions)
- [Account.java](#), [Transactions.java](#) (book versions)

# Examples of datatypes (Classes)

Class	Attributes	Operations
Student	Name Address Major Grade point average	Set address Set major Compute grade point average
Rectangle	Length Width Color	Set length Set width Set color
Aquarium	Material Length Width Height	Set material Set length Set width Set height Compute volume Compute filled weight
Flight	Airline Flight number Origin city Destination city Current status	Set airline Set flight number Determine status
Employee	Name Department Title Salary	Set department Set title Set salary Compute wages Compute bonus Compute taxes

## Another Example: RollingDice.java

```
*****  
// RollingDice.java          Author: Lewis/Loftus  
// Demonstrates the creation and use of a user-defined class.  
*****  
public class RollingDice  
{  
    //-----  
    // Creates two Die objects and rolls them several times.  
    //-----  
    public static void main (String[] args)  
    {  
        Die die1, die2;  
        int sum;  
        die1 = new Die();  
        die2 = new Die();  
  
        die1.roll();  
        die2.roll();  
        System.out.println ("Die One: " + die1 + ", Die Two: " + die2);  
  
        die1.roll();  
        die2.setFaceValue(4);  
        System.out.println ("Die One: " + die1 + ", Die Two: " + die2);  
  
        sum = die1.getFaceValue() + die2.getFaceValue();  
        System.out.println ("Sum: " + sum);  
  
        sum = die1.roll() + die2.roll();  
        System.out.println ("Die One: " + die1 + ", Die Two: " + die2);  
        System.out.println ("New sum: " + sum);  
    }  
}
```

### Sample Run

```
Die One: 5, Die Two: 2  
Die One: 1, Die Two: 4  
Sum: 5  
Die One: 4, Die Two: 2  
New sum: 6
```

```
*****  
// Die.java      Author: Lewis/Loftus  
//  
// Represents one die (singular of dice) with faces showing values  
// between 1 and 6.  
*****  
  
public class Die  
{  
    private final int MAX = 6; // maximum face value  
  
    private int faceValue; // current value showing on the die  
  
    //-----  
    // Constructor: Sets the initial face value.  
    //-----  
    public Die()  
    {  
        faceValue = 1;  
    }  
  
    //-----  
    // Rolls the die and returns the result.  
    //-----  
    public int roll()  
    {  
        faceValue = (int)(Math.random() * MAX) + 1;  
        return faceValue;  
    }  
continue
```

**continue**

```
//-----
// Face value mutator.
//-----
public void setFaceValue (int value)
{
    faceValue = value;
}

//-----
// Face value accessor.
//-----
public int getFaceValue()
{
    return faceValue;
}

//-----
// Returns a string representation of this die.
//-----
public String toString()
{
    String result = Integer.toString(faceValue);

    return result;
}
```

# Next: Focus on Method definition

- parameters
- return type
- return statement

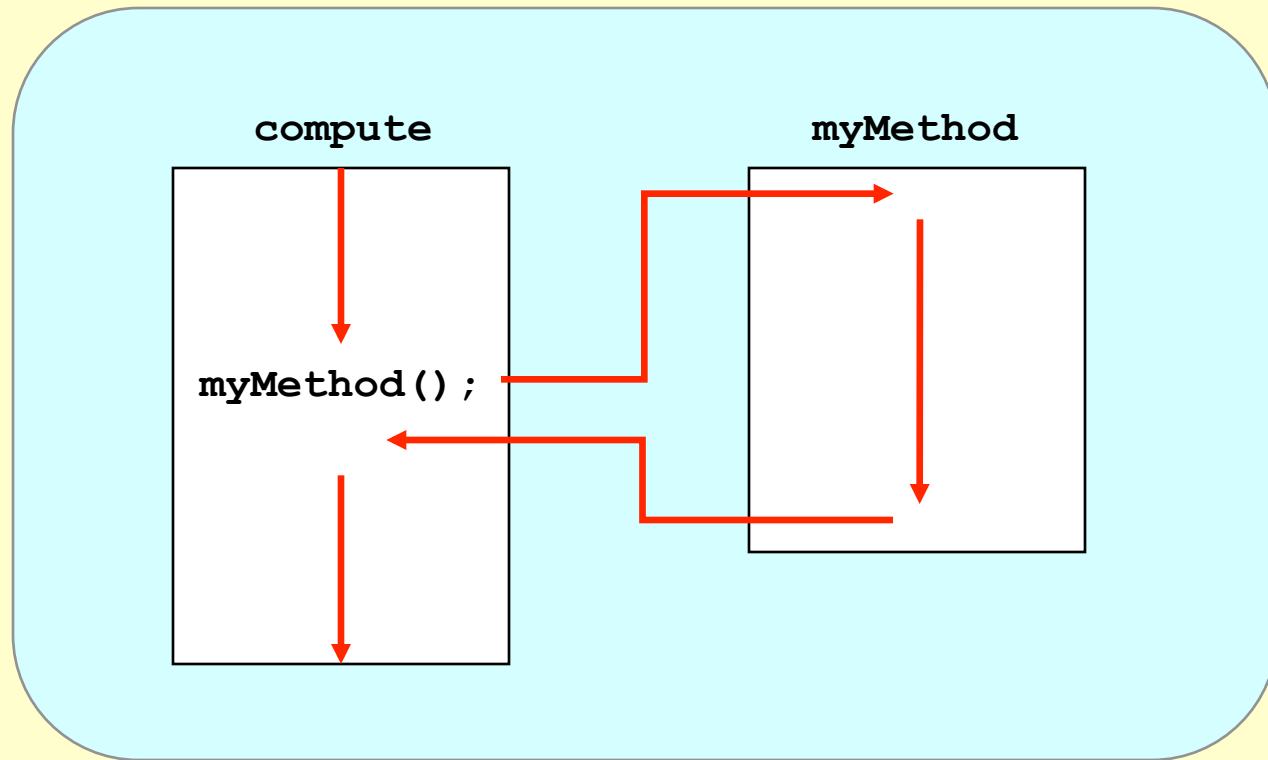
```
char ch = obj.calc (start, 2, "ABCDE");
```

```
char calc (int num1, int num2, String message)
```

```
{  
    int sum = num1 + num2;  
    char result = message.charAt (sum);  
  
    return result;  
}
```

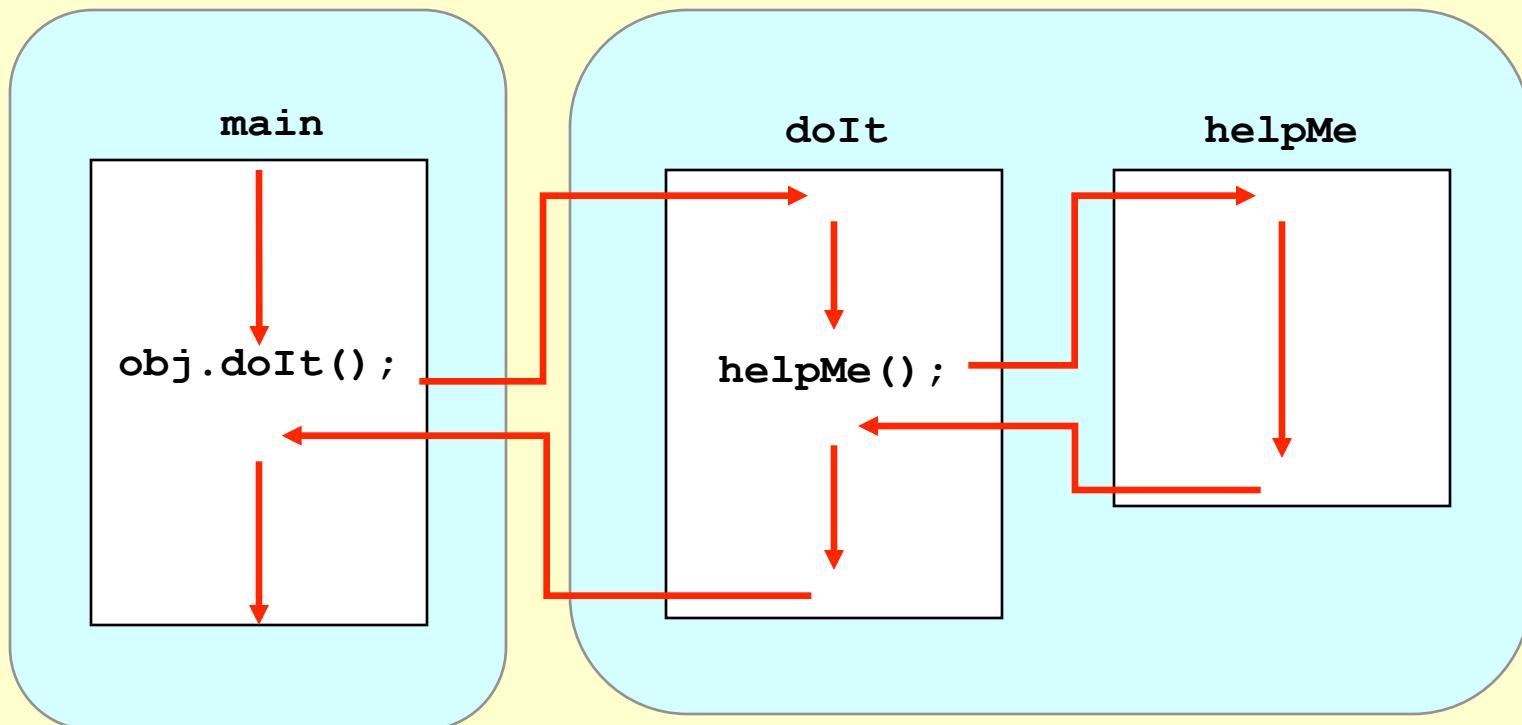
# Method Control Flow

- If the called method is in the same class, only the method name is needed



# Method Control Flow

- The called method is often part of another class or object



- Thus the dot operator is an addressing mechanism. Note that it can also be used to access an object's or class's data directly, for example
  - `acct1.name`
  - `Color.black`
- more on this later (encapsulation)

# Invoking methods within the same class

- An object's method may access any of its other methods directly. Eg:

```
public void addInterest(double rate)
{
    deposit (rate*balance);
}
```

# Invoking methods within the same class

- The main method can do this too!

```
public static void main (String[] args)
{
    printManyGreetings(5);
}
```

- **assumes** `printManyGreetings()` **is defined** in the same class
- A convenient way to test methods

# Invoking methods within the same class

- The main method can do this too!

```
public static void main (String[] args)
{
    printManyGreetings(5);
}
```

- assumes `printManyGreetings()` is defined in the same class

**NOTE:** Since `main()` is **static**,  
`printManyGreetings()` must also be **static**

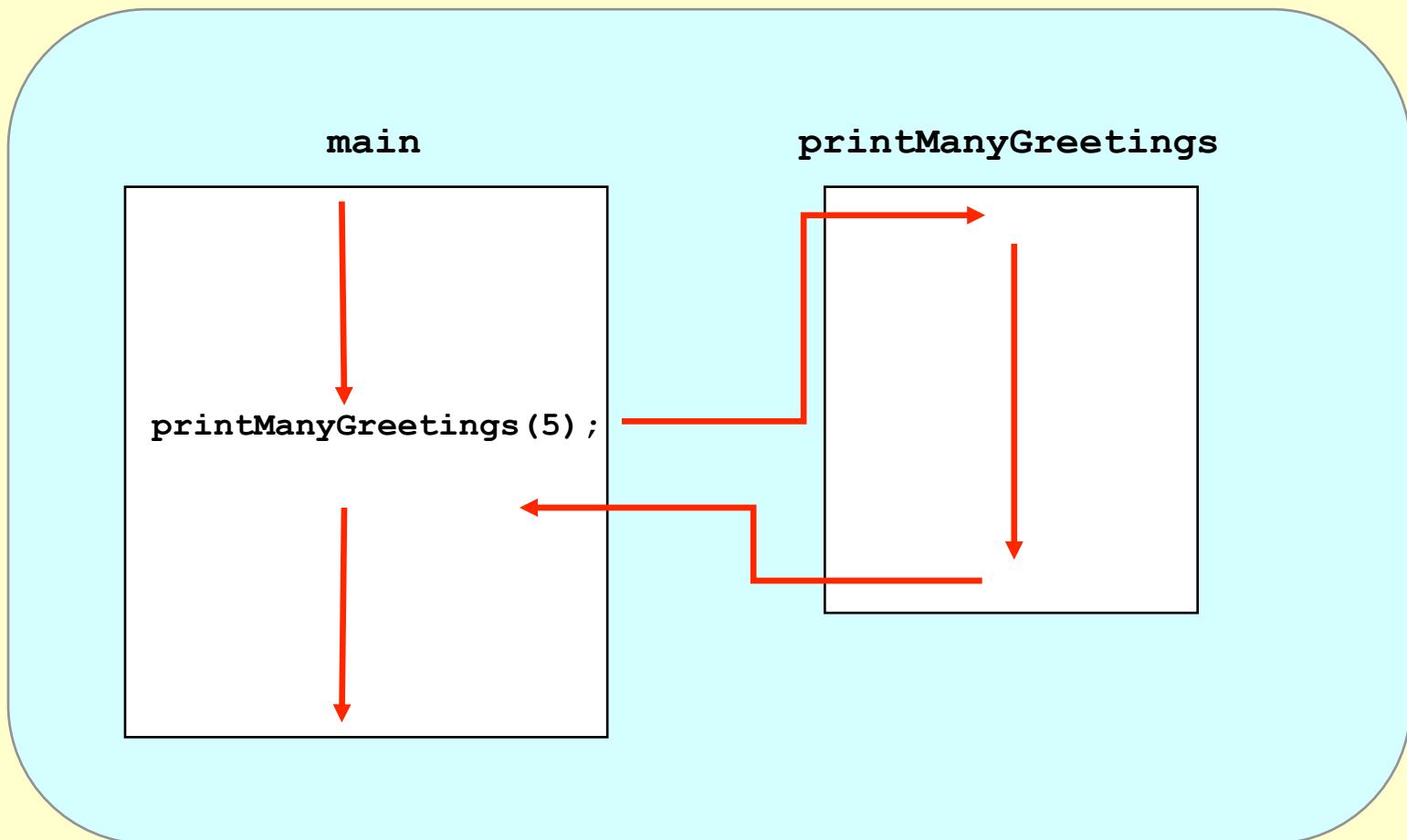
# Implementing printManyGreetings () method:

## Invoking the method:

```
printManyGreetings(5);
```

# Method Control Flow revisited

- Recall, if the called method is in the same class, only the method name is needed



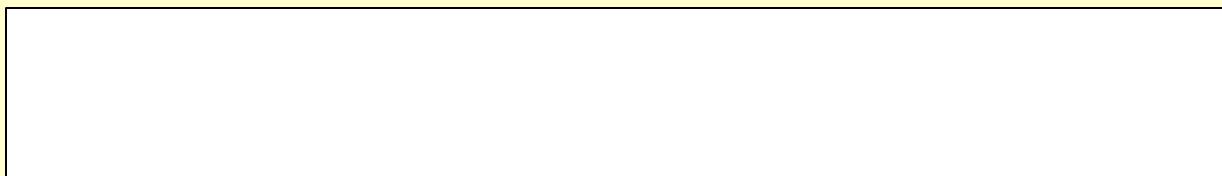
# More Method Examples:

- Write a method with two `double` parameters `a` and `b` that computes and returns the sum of squares of its two parameters (i.e.,  $a^2 + b^2$ ).

How do we invoke the method to compute & print:  $(14.8)^2 + (37.65)^2$  ?

# More Method Examples:

- Write a method with one `int` parameter `num`, that returns a String composed of “Happy Birthday” `num` times
- How do we invoke the method to print “happy birthday” 4 times?

A large, empty rectangular box with a thin black border, designed to look like a text input field for writing code.

# Getting to know classes so far

- Using predefined classes from the Java API.
- Defining classes for our own datatypes.

## **datatypes:**

- Account
- Die
- Shoe
- Person

## **Clients (*Driver classes*):**

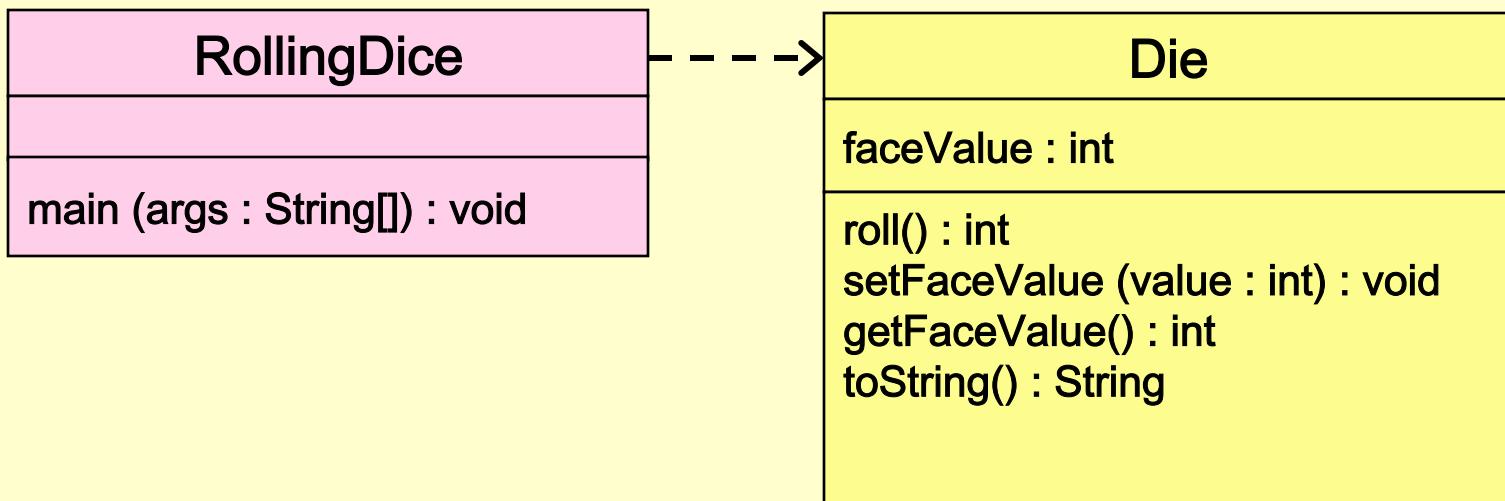
- Transactions, OnePercent
- RollingDice
- YouVeGotShoes (Project)
- PeopleBeingPeople (Lab)

**Next:** Focus on method definition/invocation

# UML Class Diagrams

UML = Unified Modelling Language

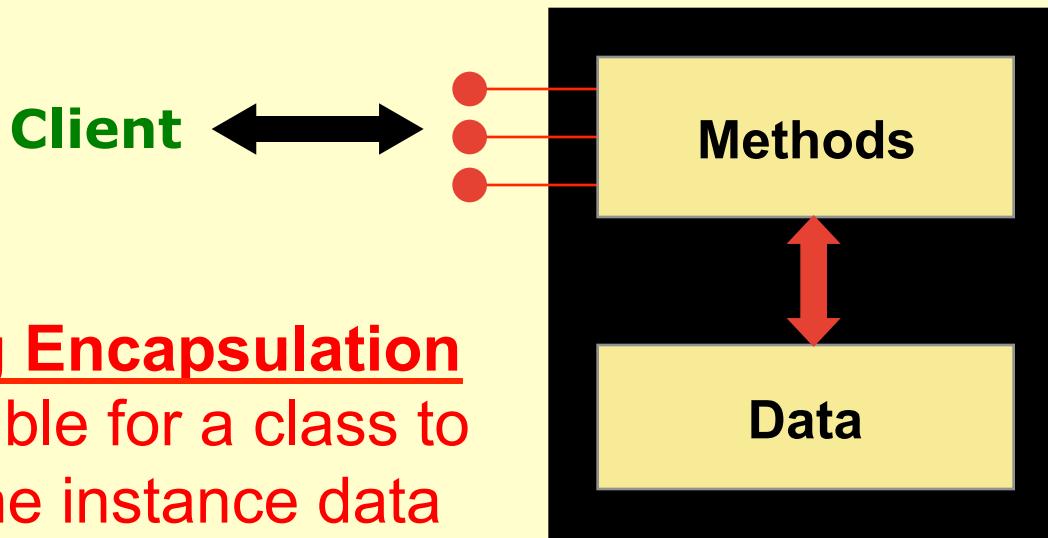
- Example: A UML class diagram for the RollingDice program:



# UML class diagram for Transactions program?

# Encapsulation

- An encapsulated object can be thought of as a *black box* -- its inner workings are hidden from the client
- The client invokes the interface methods which in turn manage the instance data

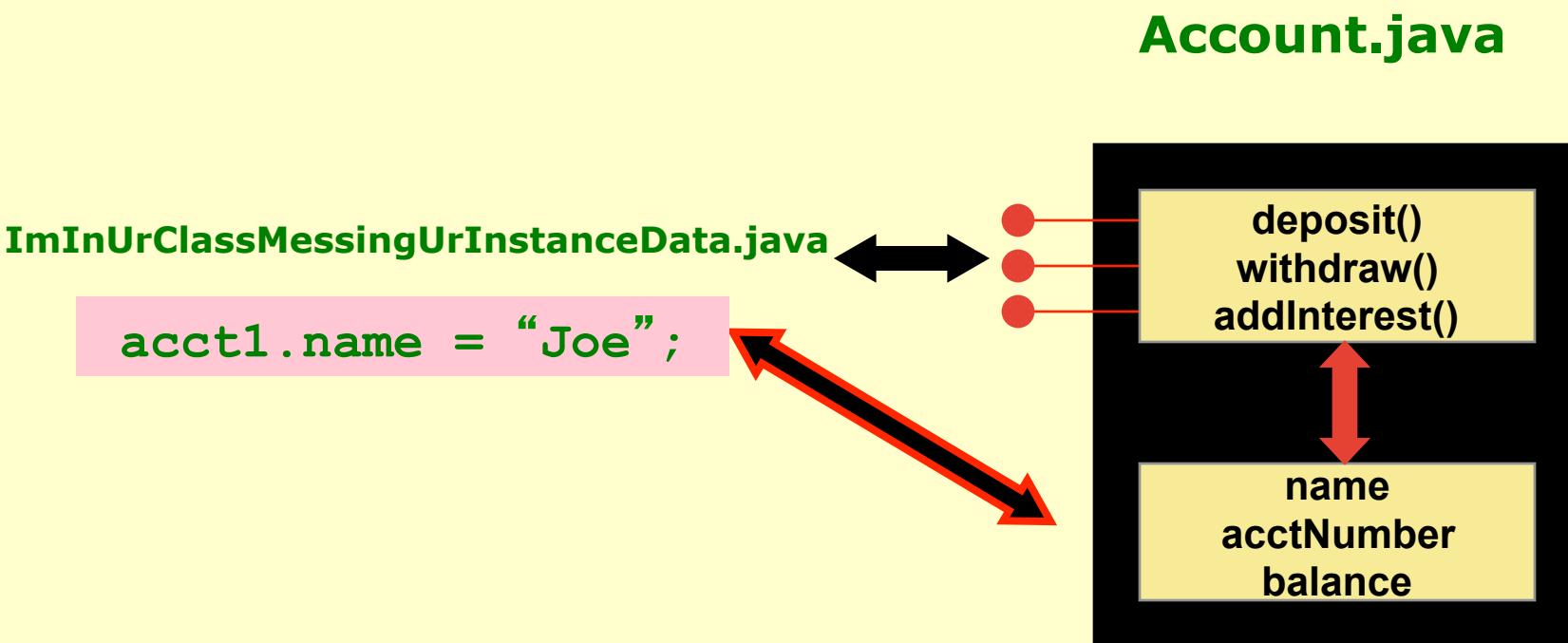


## Violating Encapsulation

It is possible for a class to access the instance data of another class directly

# Violating Encapsulation - **WRONG**

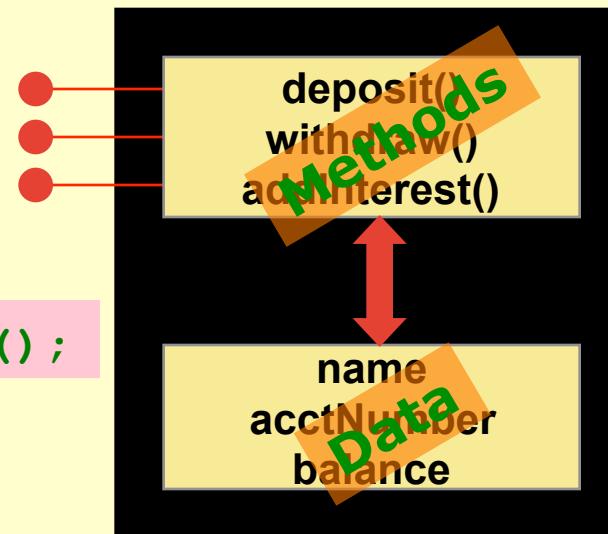
- It is possible for a class to access the instance data of another class directly – ***but it's not a good idea!***
- See [Account.java](#)
- See [ImInUrClassMessingUrInstanceIdData.java](#)



# Use Accessors & Mutators - **RIGHT**

- Indirect access through methods
- accessors and mutators (“getters” and “setters”)
- Usually named getX() and setX()

**Account.java**



**Transactions.java** ←→

**Example**

```
int x1 = acct1.getBalance();
```

# Visibility Modifiers

- In Java, we enforce encapsulation through the appropriate use of *visibility modifiers*:
  - **public** – can be referenced from other classes
  - **private** – can be referenced only within that class:
  - **protected** – involves inheritance (discussed later)
- Data declared without a visibility modifier have *default visibility* and can be referenced by any class in the same package
- An overview of all Java modifiers is presented in Appendix E

# Violating Encapsulation experiment

- Revisit your solution for the [Account Class Exercise](#)
- Add some code to the OnePercent.java class to modify the value of an instance variable, eg:

```
acct1.name = "Joe";
```

- This should work as expected
- Now modify Account.java – insert the modifier **private** in front of that variable declaration:

```
private String name;
```

- Re-compile the Account class and run your program again. Note the error you get.

# public constants are ok

Example: The Account class can have a constant for the interest rate:

```
public final double RATE = 0.015;
```

A client (eg, OnePercent.java) can access this constant directly:

```
System.out.print ("Interest rate = " + acct1.RATE);
```

# public constants are ok

Example: The Account class can have a constant for the interest rate:

Usually, constants are declared **static**

```
public final static double RATE = 0.015;
```

A driver class (eg, OnePercent.java) can access this constant directly **without creating an object**:

```
System.out.print ("Interest rate = " + acct1.RATE);
```

```
System.out.print ("Interest rate = " + Account.RATE);
```

# Visibility Modifiers – the RULES

	public	private
Variables	<b>NO</b> (but OK for public constants)	<b>Yes</b>
Methods	<b>Yes</b>	<b>Yes, for support methods only</b>