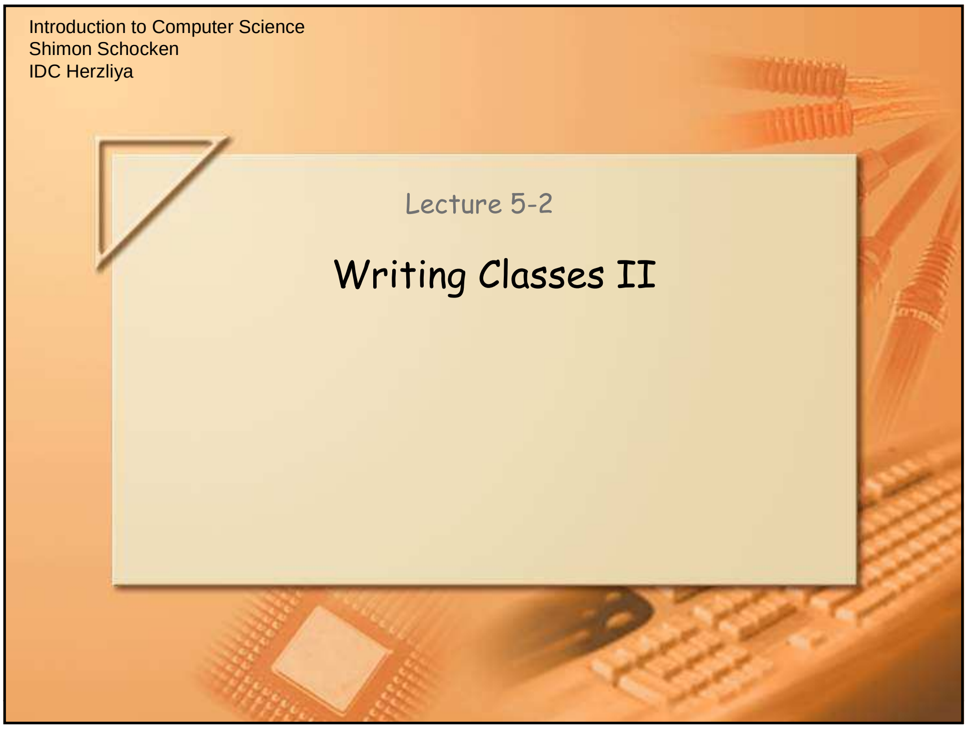
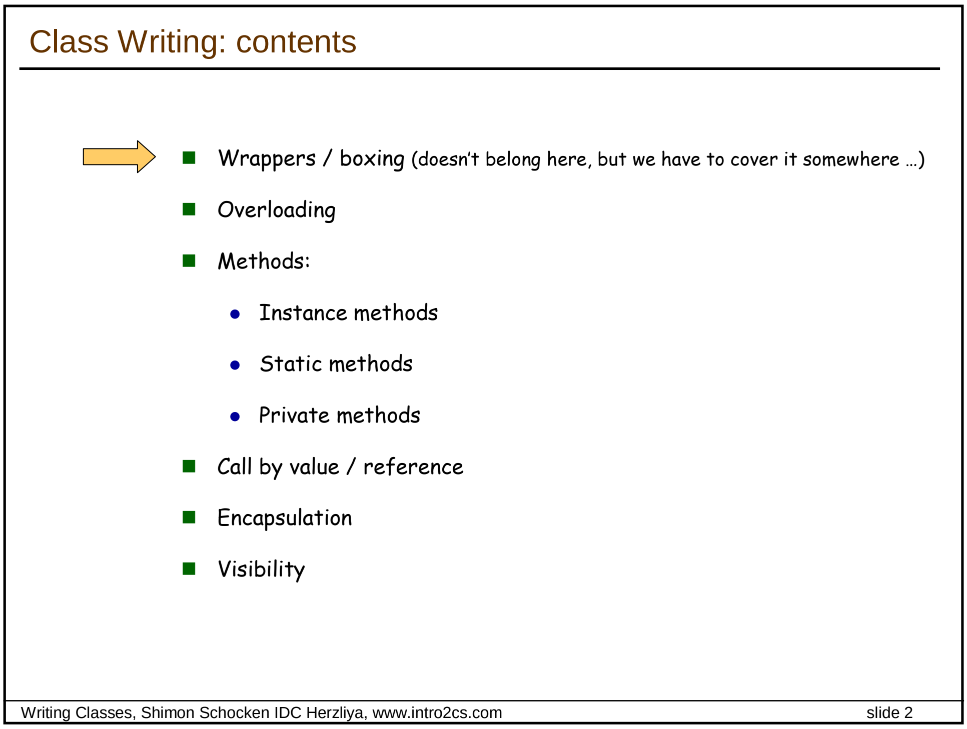


Introduction to Computer Science
Shimon Schocken
IDC Herzliya

Lecture 5-2

Writing Classes II



Class Writing: contents

- Wrappers / boxing (doesn't belong here, but we have to cover it somewhere …)
- Overloading
- Methods:
  - Instance methods
  - Static methods
  - Private methods
- Call by value / reference
- Encapsulation
- Visibility

Wrapper classes

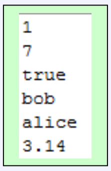
- Values of primitive types, say the `int` value `25`, are *literals*
- In some situations, it is necessary to treat, say, `25`, as an *object*
- For this purpose, Java provides a class called `Integer`
- There are nine such wrapper classes, designed to give object representations of the corresponding primitive types:

Java's primitive types:	Corresponding wrapper classes:
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`
`boolean`	`Boolean`
`Char`	`Character`
`void`	`Void`

- Why do we need this headache?
- Because in some situations you simply cannot use primitive values.
  For example, some collection classes are designed to contain objects only.

Example of wrapper classes in action: `ArrayList`

```
import java.util.ArrayList;

public class WrapperDemo {
   public static void main(String[] args) {
        ArrayList a = new ArrayList();
        a.add(new Integer(1));
        a.add(new Integer(7));
        a.add(new Boolean(true));
        a.add("bob");
        a.add("alice");
        a.add(new Double(3.14));
        for (Object obj : a)
            System.out.println(obj);
   }
}
```

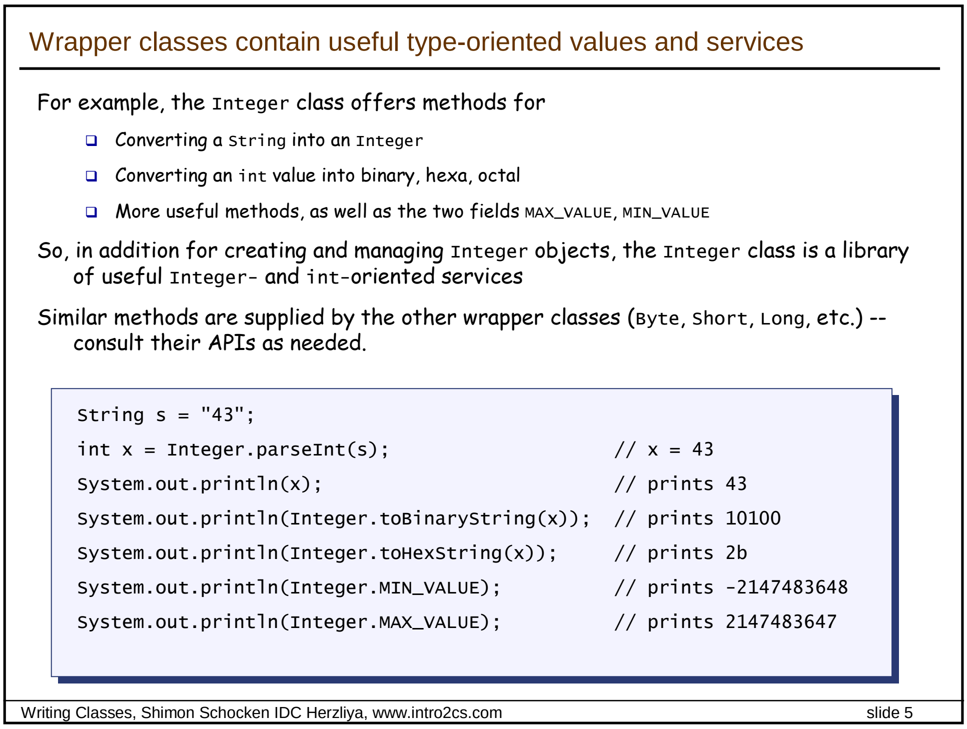


```
1
7
true
bob
alice
3.14
```

ArrayList is like a growable array that can accommodate any object type

The elements of ArrayList must be objects.

- In Java, every object is an `Object` type;
  that's why the `for` loop above works fine
- The loop is an example of *polymorphic processing*,
  to be discussed later in the course.



Wrapper classes contain useful type-oriented values and services

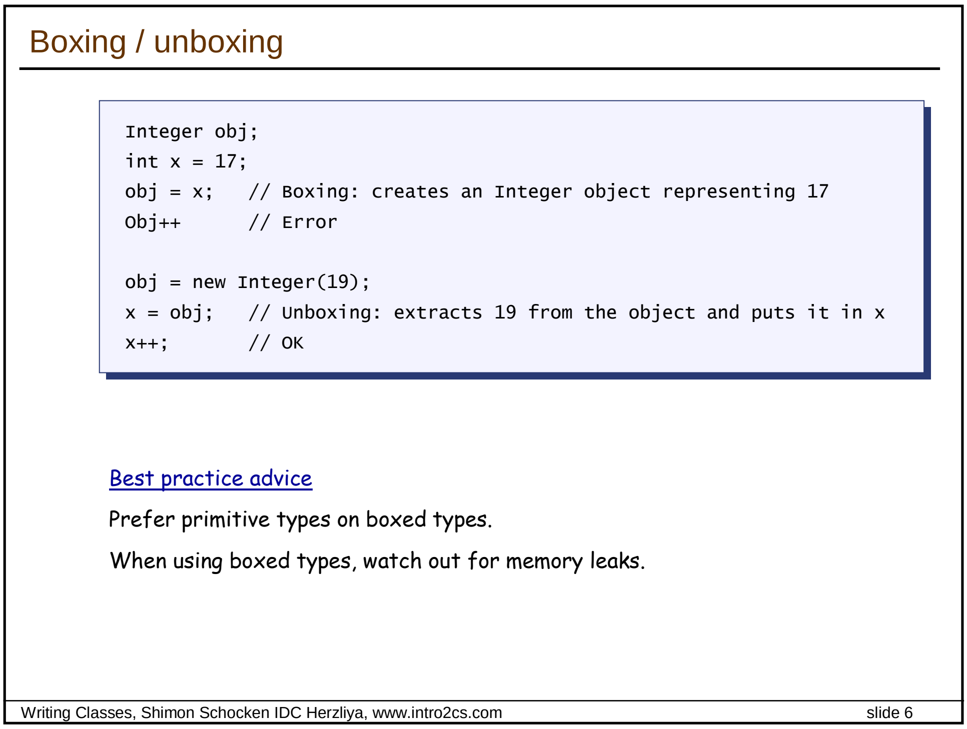
For example, the `Integer` class offers methods for

- Converting a `String` into an `Integer`
- Converting an `int` value into binary, hexa, octal
- More useful methods, as well as the two fields `MAX_VALUE`, `MIN_VALUE`

So, in addition for creating and managing `Integer` objects, the `Integer` class is a library of useful `Integer`- and `int`-oriented services

Similar methods are supplied by the other wrapper classes (`Byte`, `Short`, `Long`, etc.) -- consult their APIs as needed.

```
String s = "43";
int x = Integer.parseInt(s);                     // x = 43
System.out.println(x);                           // prints 43
System.out.println(Integer.toBinaryString(x));  // prints 10100
System.out.println(Integer.toHexString(x));     // prints 2b
System.out.println(Integer.MIN_VALUE);          // prints -2147483648
System.out.println(Integer.MAX_VALUE);          // prints 2147483647
```



Boxing / unboxing

```
Integer obj;
int x = 17;
obj = x;   // Boxing: creates an Integer object representing 17
Obj++      // Error

obj = new Integer(19);
x = obj;   // Unboxing: extracts 19 from the object and puts it in x
x++;       // OK
```

Best practice advice

Prefer primitive types on boxed types.

When using boxed types, watch out for memory leaks.

Class Writing: contents

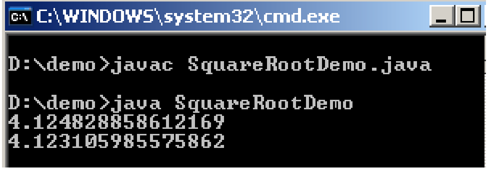
- Wrappers / boxing
- Overloading
- Methods:
  - Instance methods
  - Static methods
  - Private methods
- Call by value / reference
- Encapsulation
- Visibility

Method overloading

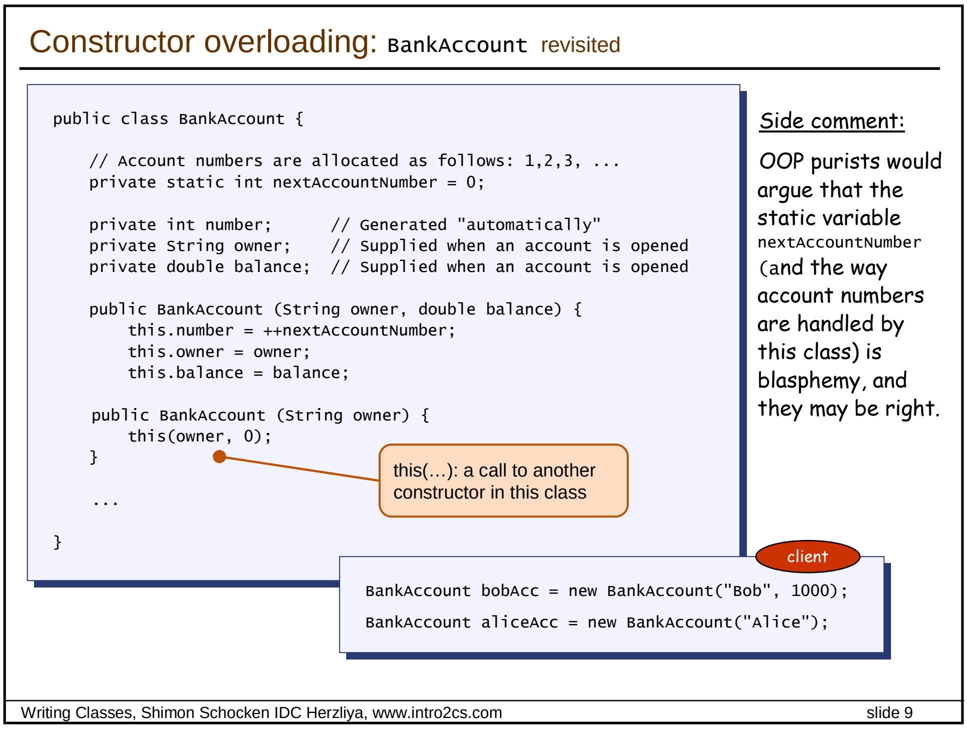
```
public class SquareRootDemo {

public static void main(String args[]) {
  System.out.println(sqrt(17));
  System.out.println(sqrt(17, 0.0001));
}

static double sqrt(double x) {
  return sqrt(x,0.1);
}

static double sqrt(double x, double precision) {
  double root = x / 2;
  while (Math.abs((root * root) - x) > precision) {
    root = (root + (x / root)) / 2;
  }
  return root;
}
}
```

Two methods, same name

- Method signature: method name, parameter names, parameter types
- Java allows defining different methods with the same name, provided that they have different signatures
- When the caller invokes a method, the compiler determines which method to invoke according to the arguments passed by the caller
- Advantages: Promotes shorter, fewer, and readable method names.



```
D:\demo>javac SquareRootDemo.java

D:\demo>java SquareRootDemo
4.124828858612169
4.123105985575862
```



Constructor overloading: `BankAccount` revisited

```
public class BankAccount {

    // Account numbers are allocated as follows: 1,2,3, ...
    private static int nextAccountNumber = 0;

    private int number;      // Generated "automatically"
    private String owner;    // Supplied when an account is opened
    private double balance;  // Supplied when an account is opened

    public BankAccount (String owner, double balance) {
        this.number = ++nextAccountNumber;
        this.owner = owner;
        this.balance = balance;

    public BankAccount (String owner) {
        this(owner, 0);
    }

    ...

}
```

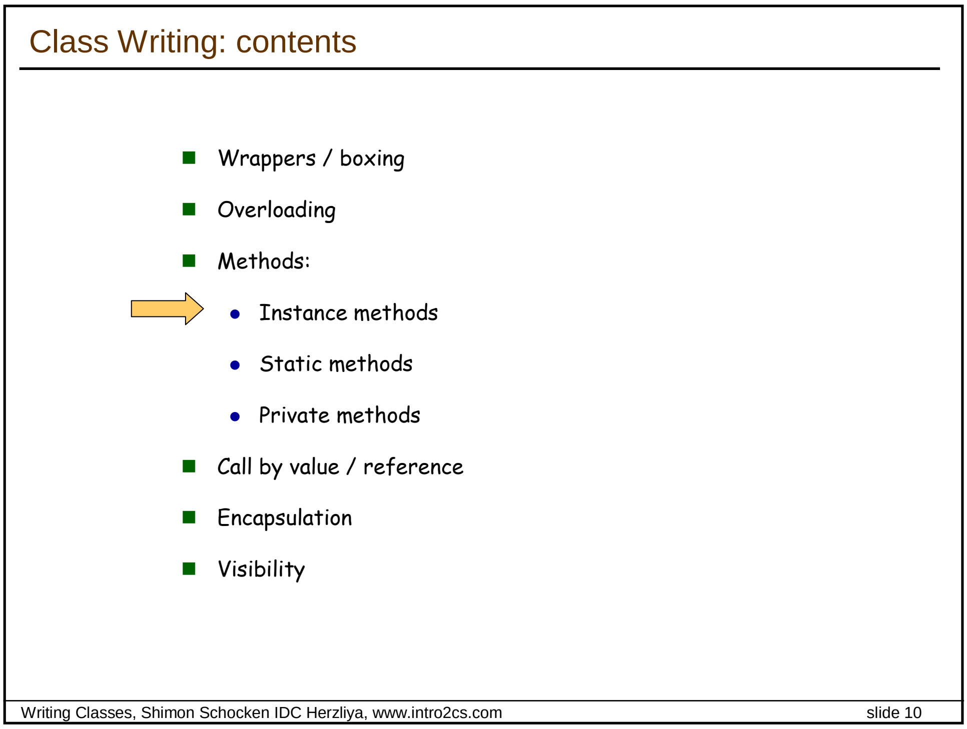
this(…): a call to another constructor in this class

client

```
BankAccount bobAcc = new BankAccount("Bob", 1000);
BankAccount aliceAcc = new BankAccount("Alice");
```

Side comment:

OOP purists would argue that the static variable `nextAccountNumber` (and the way account numbers are handled by this class) is blasphemy, and they may be right.



Class Writing: contents

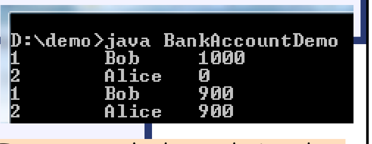
- Wrappers / boxing
- Overloading
- Methods:
  - Instance methods
  - Static methods
  - Private methods
- Call by value / reference
- Encapsulation
- Visibility

Instance methods

```
public class BankAccount {

    private static int nextAccountNumber = 0;

    private int number;
    private String owner;
    private double balance;

   // Constructors (previous slide) come here

    public void deposit (double amount) {
      balance = balance + amount;
    }

    public void withdraw (double amount) {
      balance = balance - amount;
    }

    public int getNumber() { return number; }
    public String getOwner() { return owner; }
    public double getBalance() { return balance; }

    public String toString () {
        return (number + "\t" + owner +
                "\t" + (int) balance);
    }
}
```

client

```
public class BankAccountDemo {
  public static void main (String args[]) {
    BankAccount bobAcc = new BankAccount("Bob", 1000);
    BankAccount aliceAcc = new BankAccount("Alice");

     System.out.println(bobAcc);
     System.out.println(aliceAcc);

     aliceAcc.deposit(900);
     bobAcc.withdraw(100);

     System.out.println(bobAcc);
     System.out.println(aliceAcc);
  }
}
```



```
D:\demo>java BankAccountDemo
1        Bob      1000
2        Alice    0
1        Bob      900
2        Alice    900
```

- Instance methods are designed to operate on the current object
- Instance variables implement the object's data
- Instance methods implement abstract object behaviors

Static methods

```
public class BankAccount {

    private int number;
    private String owner;
    private double balance;

    // Instead of writing the instance method:
    public void deposit (double amount) {
      balance = balance + amount;
    }

    // We could have written the static method:
    public static void deposit (BankAccount acct, double amount) {
      acct.balance = acct.balance + amount;
    }

    // More methods
}
```

client

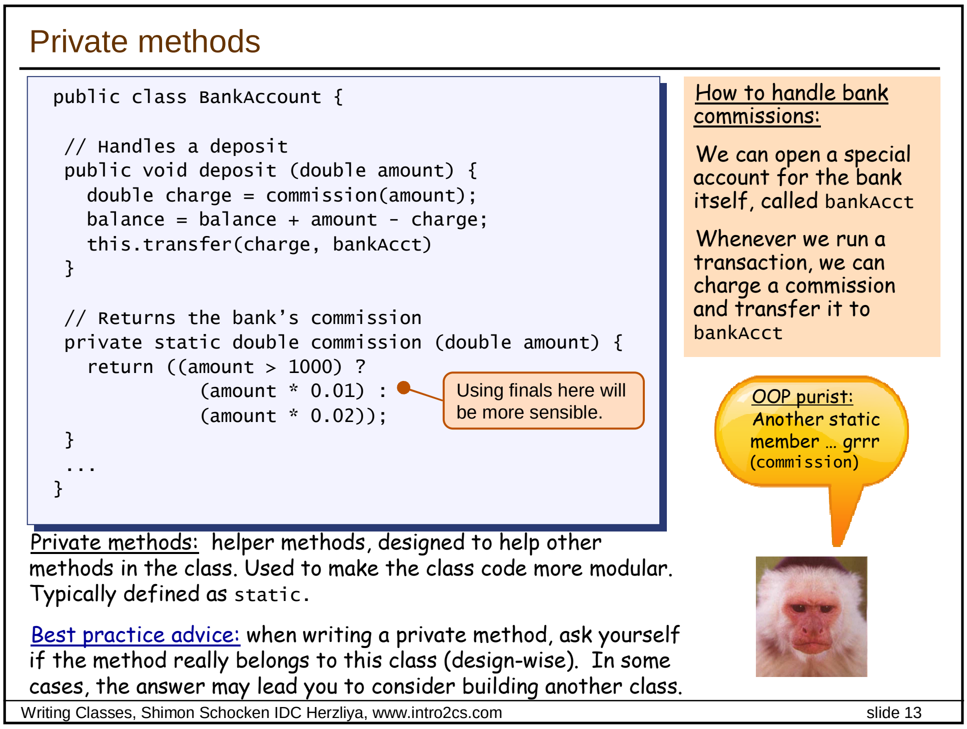
```
public class BankAccountDemo {
    ...
    // instead of calling:
    aliceAcc.deposit(900);

    // we would call:
    BankAccount.deposit(aliceAcct, 900)
    ...
}
```

Every instance method can be re-written as a static method (passing the object as an argument)

But, this will defeat the whole purpose of object-oriented programming!

Best practice advice:

- When it's natural to work with objects, use instance methods
- Don't mix instance and static methods in the same class.



Private methods

```
public class BankAccount {

  // Handles a deposit
  public void deposit (double amount) {
    double charge = commission(amount);
    balance = balance + amount - charge;
    this.transfer(charge, bankAcct)
  }

  // Returns the bank's commission
  private static double commission (double amount) {
    return ((amount > 1000) ?
                (amount * 0.01) :
                (amount * 0.02));
  }
  ...
}
```

Using finals here will be more sensible.

How to handle bank commissions:

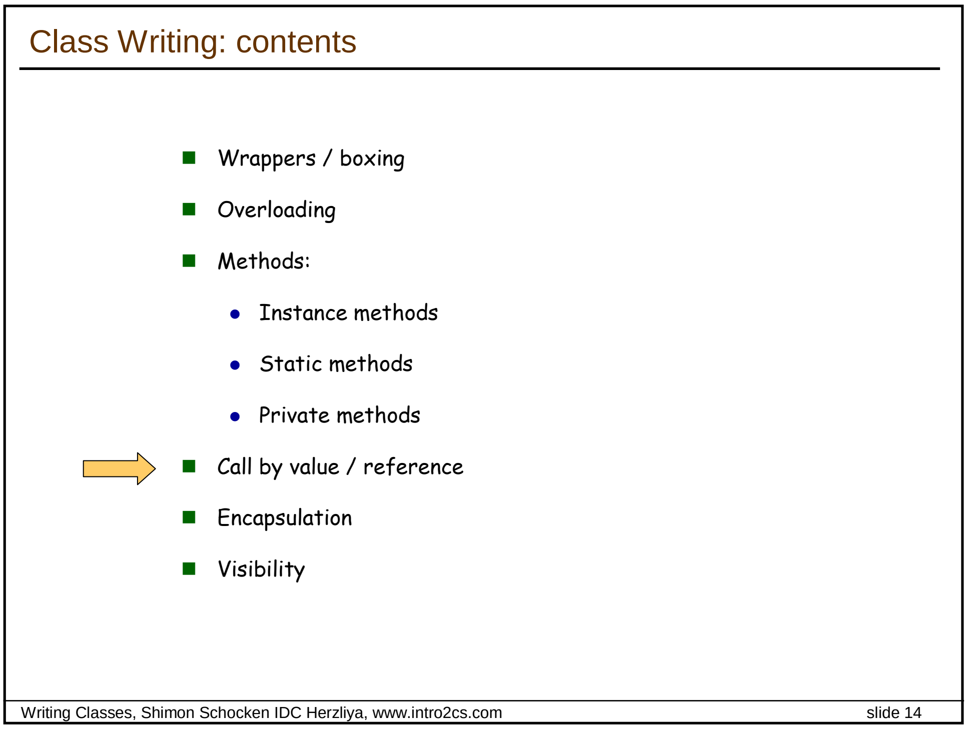
We can open a special account for the bank itself, called `bankAcct`

Whenever we run a transaction, we can charge a commission and transfer it to `bankAcct`

OOP purist: Another static member … grrr (`commission`)

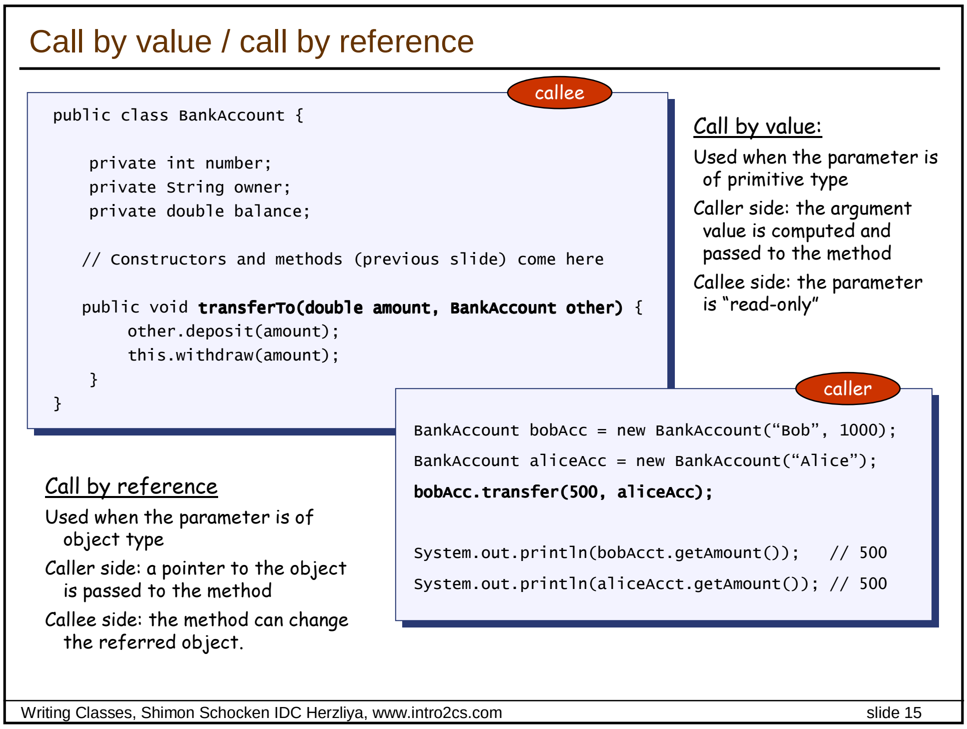
Private methods: helper methods, designed to help other methods in the class. Used to make the class code more modular. Typically defined as `static`.

Best practice advice: when writing a private method, ask yourself if the method really belongs to this class (design-wise). In some cases, the answer may lead you to consider building another class.



Class Writing: contents

- Wrappers / boxing
- Overloading
- Methods:
  - Instance methods
  - Static methods
  - Private methods
- ➡ Call by value / reference
- Encapsulation
- Visibility



Call by value / call by reference

callee

```
public class BankAccount {

    private int number;
    private String owner;
    private double balance;

   // Constructors and methods (previous slide) come here

   public void transferTo(double amount, BankAccount other) {
        other.deposit(amount);
        this.withdraw(amount);
    }
}
```

Call by value:

Used when the parameter is of primitive type

Caller side: the argument value is computed and passed to the method

Callee side: the parameter is "read-only"

caller

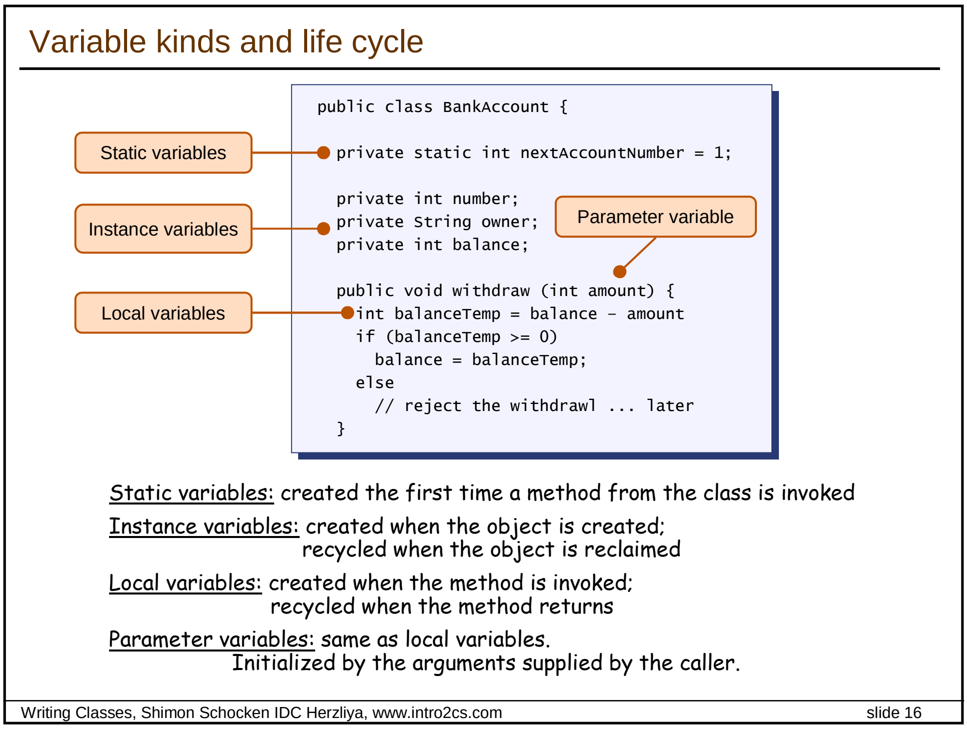
```
BankAccount bobAcc = new BankAccount("Bob", 1000);
BankAccount aliceAcc = new BankAccount("Alice");
bobAcc.transfer(500, aliceAcc);


System.out.println(bobAcct.getAmount());   // 500
System.out.println(aliceAcct.getAmount()); // 500
```

Call by reference

Used when the parameter is of object type

Caller side: a pointer to the object is passed to the method

Callee side: the method can change the referred object.



Variable kinds and life cycle

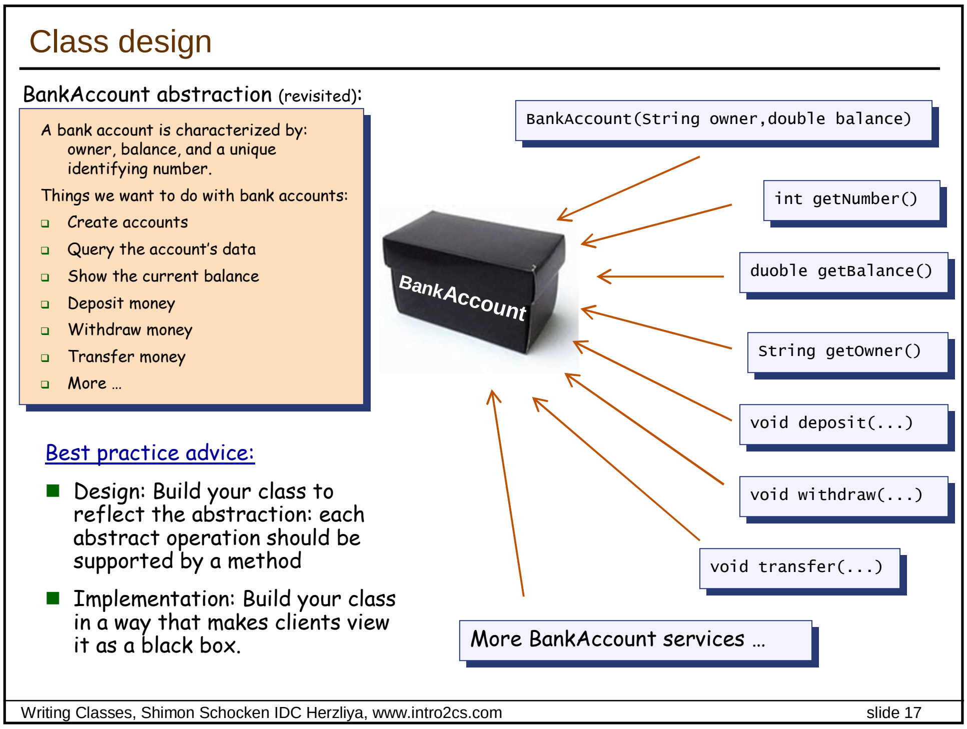
```
public class BankAccount {

  private static int nextAccountNumber = 1;

  private int number;
  private String owner;
  private int balance;

  public void withdraw (int amount) {
    int balanceTemp = balance - amount
    if (balanceTemp >= 0)
      balance = balanceTemp;
    else
      // reject the withdrawl ... later
  }
```

Static variables: created the first time a method from the class is invoked

Instance variables: created when the object is created;
recycled when the object is reclaimed

Local variables: created when the method is invoked;
recycled when the method returns

Parameter variables: same as local variables.
Initialized by the arguments supplied by the caller.



Class design

BankAccount abstraction (revisited):

A bank account is characterized by: owner, balance, and a unique identifying number.

Things we want to do with bank accounts:

- Create accounts
- Query the account's data
- Show the current balance
- Deposit money
- Withdraw money
- Transfer money
- More …

`BankAccount(String owner,double balance)`

`int getNumber()`

`duoble getBalance()`

`String getOwner()`

`void deposit(...)`

`void withdraw(...)`

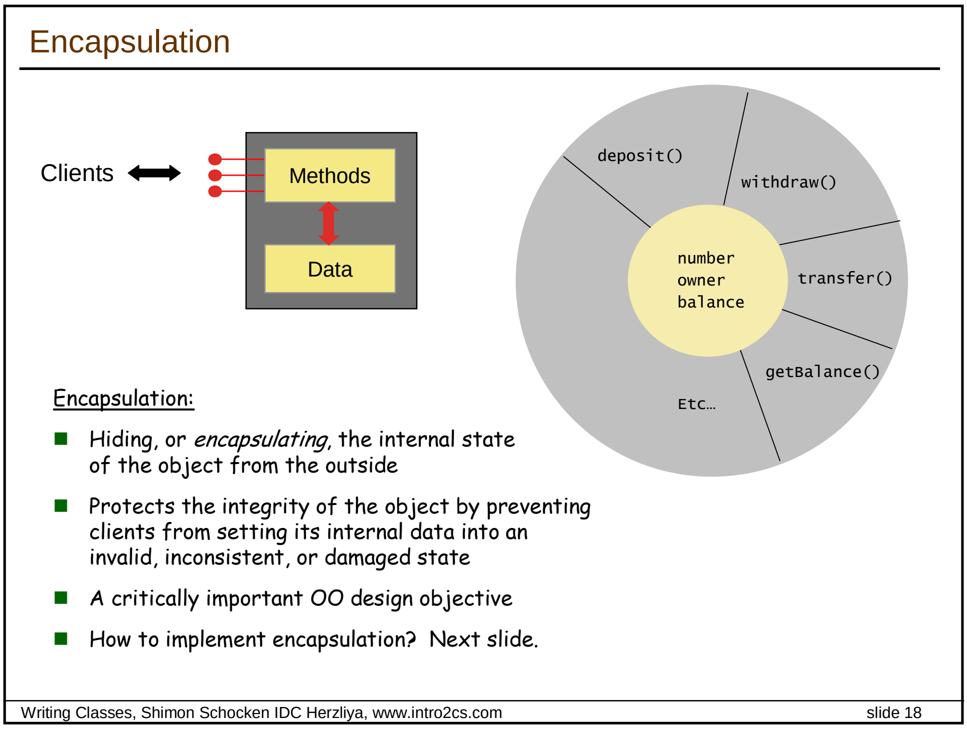
`void transfer(...)`

More BankAccount services …

Best practice advice:

- Design: Build your class to reflect the abstraction: each abstract operation should be supported by a method
- Implementation: Build your class in a way that makes clients view it as a black box.

Writing Classes, Shimon Schocken IDC Herzliya, www.intro2cs.com slide 17

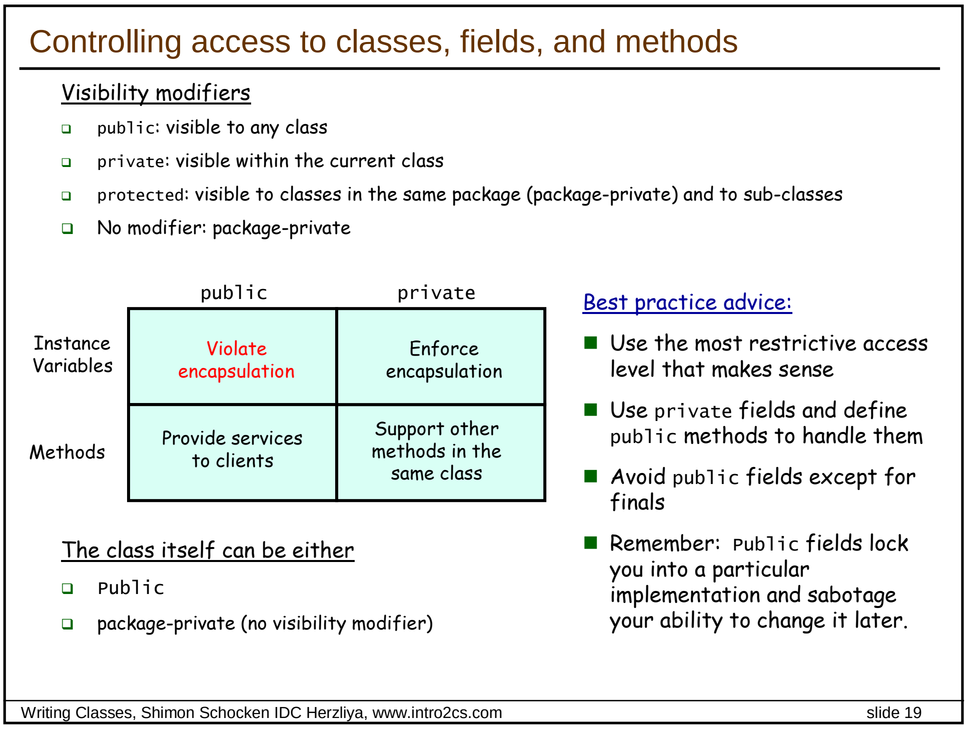


Encapsulation

Encapsulation:

- Hiding, or *encapsulating*, the internal state of the object from the outside
- Protects the integrity of the object by preventing clients from setting its internal data into an invalid, inconsistent, or damaged state
- A critically important OO design objective
- How to implement encapsulation? Next slide.

Writing Classes, Shimon Schocken IDC Herzliya, www.intro2cs.com slide 18



Controlling access to classes, fields, and methods

Visibility modifiers

- public: visible to any class
- private: visible within the current class
- protected: visible to classes in the same package (package-private) and to sub-classes
- No modifier: package-private

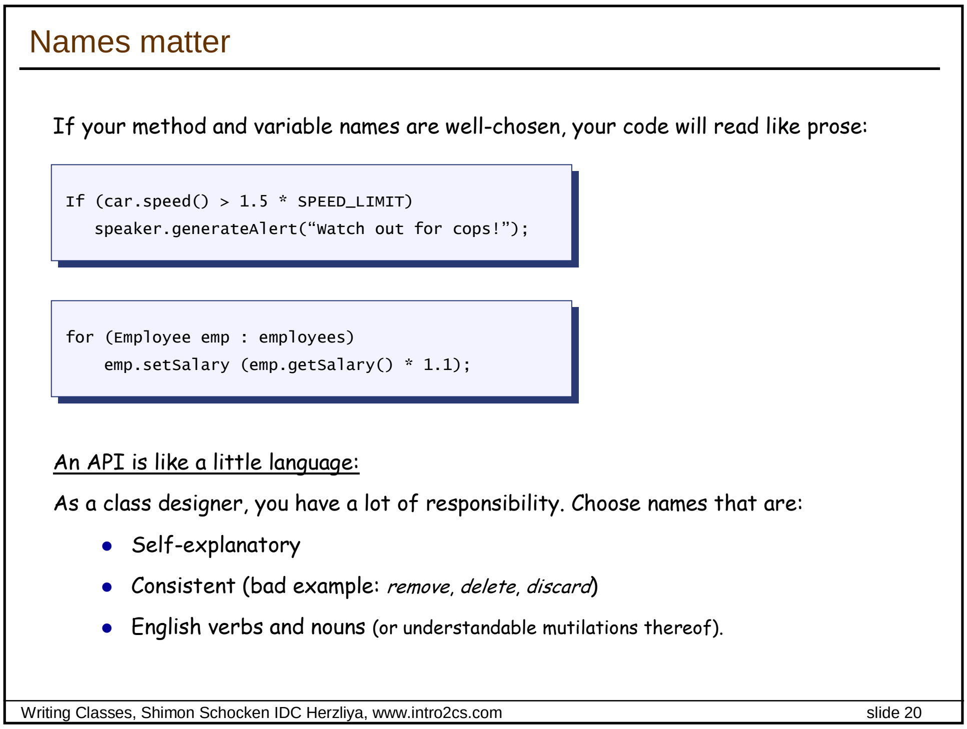
	public	private
Instance Variables	Violate encapsulation	Enforce encapsulation
Methods	Provide services to clients	Support other methods in the same class

The class itself can be either

- Public
- package-private (no visibility modifier)

Best practice advice:

- Use the most restrictive access level that makes sense
- Use private fields and define public methods to handle them
- Avoid public fields except for finals
- Remember: Public fields lock you into a particular implementation and sabotage your ability to change it later.



Names matter

If your method and variable names are well-chosen, your code will read like prose:

```
If (car.speed() > 1.5 * SPEED_LIMIT)
   speaker.generateAlert(“Watch out for cops!”);
```

```
for (Employee emp : employees)
    emp.setSalary (emp.getSalary() * 1.1);
```

An API is like a little language:

As a class designer, you have a lot of responsibility. Choose names that are:

- Self-explanatory
- Consistent (bad example: *remove, delete, discard*)
- English verbs and nouns (or understandable mutilations thereof).