
CS2040 Lecture Note #1:

Object Oriented Programming

A paradigm shift:

From procedural to object-oriented model

1. Bank Account : A simple illustration

- Let's look at C implementation of a simple bank account
- **Basic Information:**
 - ***Account Number*** : an integer value
 - ***Balance*** : a double value (should be ≥ 0)
- **Basic operations:**
 - ***Withdrawal***
 - Attempt to withdraw a certain amount from account
 - ***Deposit***
 - Attempt to deposit a certain amount to account
- Using "**struct**" (structure) is the best approach in C

1. Bank Account : C Implementation

```
typedef struct {  
    int acctNum;  
    double balance;  
} BankAcct;
```

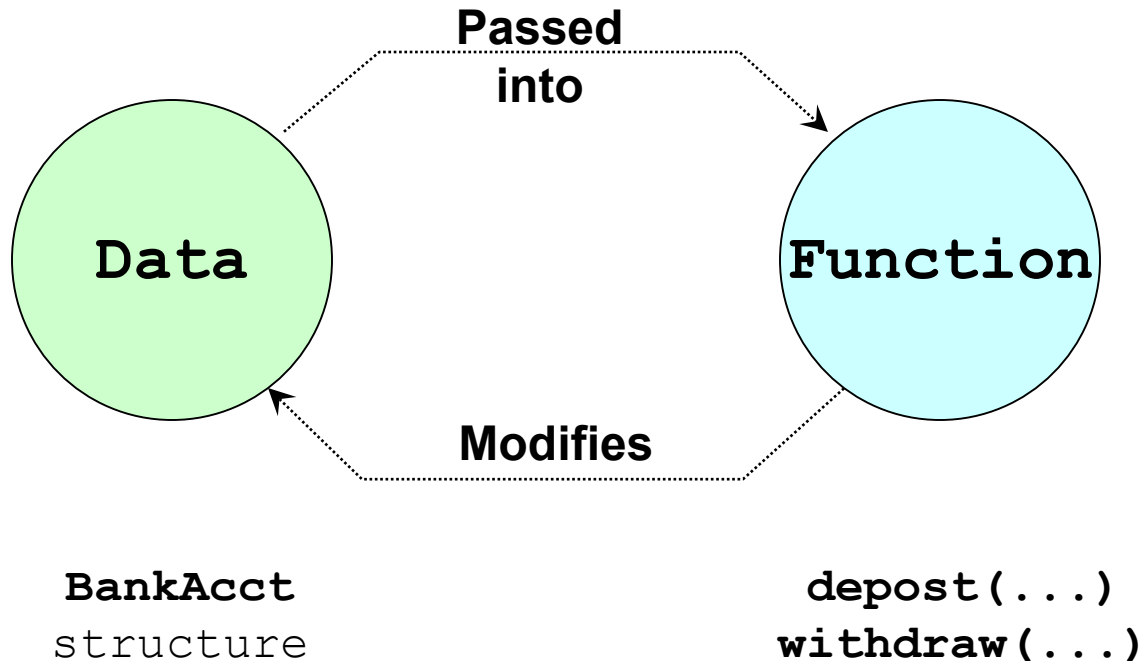
Structure to hold
information for bank
account

```
void initialize(BankAcct* baPtr, int anum)  
{  
    baPtr->acctNum = anum;  
    baPtr->balance = 0;  
}  
  
int withdraw(BankAcct* baPtr, double amount)  
{  
    if (baPtr->balance < amount)  
        return 0;           //indicate failure  
    baPtr->balance -= amount;  
    return 1;               //success  
}  
  
void deposit(BankAcct* baPtr, double amount)  
{  
    ... Code not shown ...  
}
```

Functions to
provide basic
operations

1. Bank Account : C Implementation

- C treats the data (structure) and process (function) as separate entity:



1. Bank Account : Usage Examples

Correct use of
BankAcct and its
operations

```
BankAcct ba1;  
  
initialize(&ba1, 12345);  
deposit(&ba1, 1000.50);  
withdraw(&ba1, 500.00);  
withdraw(&ba1, 600.00);  
...
```

Wrong and
malicious exploits
of BankAcct

```
BankAcct ba1;  
  
deposit(&ba1, 1000.50);  
  
initialize(&ba1, 12345);  
ba1.acctNum = 54321;  
  
ba1.balance = 10000000.00;  
...
```

Forgot to initialize

Account Number
should not change!

Balance should be
changed by authorized
operations only

1. Procedural language: **Characteristics**

- C is a typical **procedural language**
- Characteristics of procedural languages:
 - ❑ View program as a process of transforming data
 - ❑ Data and associated functions are separated
 - Require good programming discipline to ensure good organization in a program
 - ❑ Data is **publicly accessible** to everyone

Object Oriented Programming

Definition and Motivation

2. Object Oriented Languages

■ Main features:

□ **Encapsulation**

- Group data and associated functionalities into a single package
- Hide internal details from outsider

□ **Inheritance**

- A meaningful way of extending current implementation
- Introduce logical relationship between packages

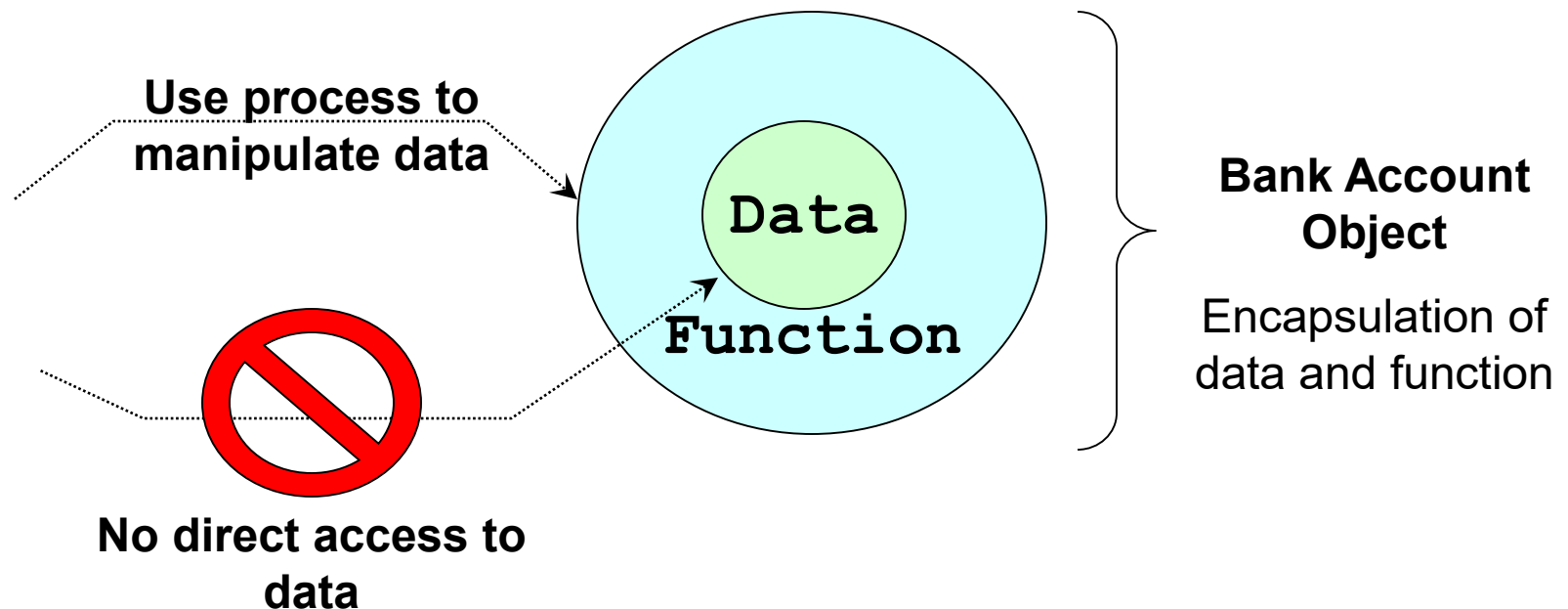
□ **Polymorphism**

- Behavior of the functionality changes according to the actual type of data

- We shall focus on encapsulation for now.

2. Bank Account: OO Implementation

- A conceptual view of equivalent object oriented implementation for the Bank Account



2. OO language: Characteristics

- Characteristics of OO languages:
 - View program as a collection of **objects**
 - Computation is performed through interaction of objects
 - Each object has a set of capabilities (functionalities) and information (data)
 - Capabilities are generally exposed to the public
 - Data are generally kept within the object
- Analogy:
 - Watching a DVD movie in the real world
 - DVD and DVD players are objects with distinct capabilities
 - Interaction between them allows a DVD movie to be played by a DVD player

Encapsulation

*Separating data (attributes) and
functions (methods)*

3.1 Encapsulation in Java: Classes

- In Java, a logical grouping of **data** + **processes** = **class**
 - A **class** is a user defined **data type**
 - Variables of a class are called **objects (instances)**
- A class contains:
 - **Data**: each object has an independent copy
 - **Functions**: process to manipulate data in an object
- **Terminology:**
 - **Data of a class :**
 - member data (**attributes**)
 - **Functions of a class:**
 - member functions (**methods**)

3.2 Accessibility

- Attributes and methods in a class can have different level of **accessibilities** (visibilities)

public

- Anyone can access
- Usually intended for methods only

private

- Can be assessed by the same class
- Recommended for all attributes

protected

- Can be assessed of the same class or its child classes can access AND
- Can be assessed by the classes in the same **Java package (not covered)**
- Recommended for attributes/methods that are common in a “family”

[None]

- Only accessible to classes in the same **Java package (not covered)**
- Known as the **package private visibility**

3.3 Bank Account: Java Implementation

```
class BankAcct {  
  
    private int _acctNum;  
    private double _balance;  
  
    public boolean withdraw(double amount) {  
        if (_balance < amount)  
            return false;  
        _balance -= amount;  
        return true;  
    }  
  
    public void deposit(double amount) {  
        if (amount <= 0)  
            return;  
        _balance += amount;  
    }  
}
```

Good coding habits:

- Separate attributes and methods
- Use "_" or myXXXX to denote attributes

TestBankAcct.java

3.4 Constructors

- Each class has one or more specialized methods known as **constructor**
 - Called when an object is created
 - Useful for initializing the attributes of an object
- **Default constructor**
 - Take in no parameter
 - Automatically provided by the compiler **if programmer does not define any constructor method**
 - Initialize all attributes to 0
- **Non-default constructor**
 - Can take in parameter
 - Can have multiple different constructors

3.4 Constructors: Example

```
class BankAcct {  
  
    private int _acctNum;  
    private double _balance;  
  
    public BankAcct() {  
        //initialize all attributes to 0  
    }  
  
    public BankAcct(int aNum, double bal) {  
        //initialize attributes with user provided values  
        _acctNum = aNum;  
        _balance = bal;  
    }  
  
    //Other methods not shown  
  
}
```

Syntax Note:

- Constructor has NO return type.
- Constructor has the same name as the class

TestBankAcct.java

3.5 Accessors and Mutators

- A method can also be called
 - an **accessor** if it accesses (retrieves) the value of an object's attribute
 - a **mutator** if it mutates (modifies) the value of an object's attribute
- Are the `withdraw()` and `deposit()` methods in slide 14 accessors or mutators?

3.6 Class and Object

- The class declaration defines a **new data type**
 - No actual variables are allocated!
- To have an instance of a class:
 - Create (instantiate) **object**
 - Variable that refers to an object is known as **reference** in Java
- The distinction between class and object
 - Similar to *structure declaration* and *structure variable* in C
 - Analogy: **class** == blueprint/template, **object** == actual house
- To access **public** attribute or method of an object
 - Use the “.” dot operator (Similar to structure access in C)

3.7 Bank Account: Example usage

```
class BankAcct { ..... } //not shown
```

```
class TestBankAcct {  
    public static void main(String[] args) {  
        BankAcct ba1 = new BankAcct();  
        BankAcct ba2 = new BankAcct(1234, 99.99);
```

```
        ...
```

```
        ba1.deposit(1000);
```

```
        ba2.withdraw(500.25);
```

```
        // Accessibility restricts access, the following  
        // statements will result in compilation error
```

```
        ba1._acctNum = 555555;  
        ba1._balance += 12345.99;
```

```
    }
```

```
}
```

Syntax Note:

- "new" keyword creates an object
- One of the constructors is used

Compilation error!

3.8 Problem: Print Account Information

- At this point, the **BankAcct** class has some usage problems:
 - Cannot access the account number and balance from outside the class
- Modify the class such that:
 - We can print out the account number and balance as an outsider
 - Many solutions!
 - Don't jump for any answers
 - Good solution should follow the encapsulation rule

3.8 Solution: Print Account Information (1/2)

- We can add a simple `print()` method to the class

```
class BankAcct {  
  
    //Other methods and attributes not shown  
  
    public void print() {  
        System.out.println("Account Number: " + _acctNum);  
        System.out.printf("Balance: $%.2f\n", _balance);  
    }  
}
```

3.8 Solution: Print Account Information (2/2)

- Better OOP practice
 - Provide accessors for the object's attributes

```
class BankAcct {  
    //Other methods and attributes not shown  
  
    public int getAcct() {  
        return _acctNum;  
    }  
  
    public double getBal() {  
        return _balance;  
    }  
  
    public void print() {  
        System.out.println("Account Number: " + getAcct());  
        System.out.printf("Balance: $%.2f\n", getBal());  
    }  
}
```

TestBankAcct2.java

3.9 Object Reference Data Type

- In Java, all non-primitive data type variables are object references
 - An object reference **works like a C pointer**

```
class BankAcct { ..... } //not shown
```

```
class TestBankAcct {  
    public static void main(String[] args) {
```

```
        BankAcct ba1 = new BankAcct();  
        BankAcct ba2;
```

ba1 has a balance of 0

```
        ba2 = ba1;  
        ba1.deposit(1000);
```

```
        ba2.print();
```

Is ba2 changed?

```
    }  
}
```

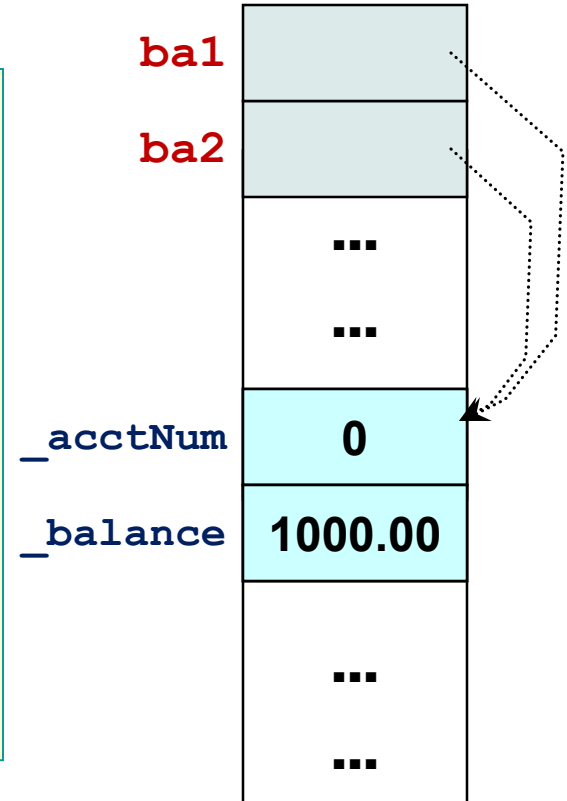
3.9 Object Reference: Memory Snapshot

```
class BankAcct { ..... } //not shown

class TestBankAcct {
    public static void main( String[] args ) {
        BankAcct ba1 = new BankAcct();
        BankAcct ba2;

        ba2 = ba1;
        ba1.deposit(1000);

        ba2.print();
    }
}
```



- Before the "**ba2 = ba1**" assignment:
 - ❑ **ba2** is a **NULL reference**
 - ❑ Results in runtime error if you attempt to access it

3.10 Instance Method vs Static Method

- Methods in the **BankAcct** class are known as **instance method**:
 - You need an object reference of the right type to invoke these methods
 - These methods have access to the attributes in the object automatically
- Different from **static/class method** covered earlier
 - Static methods have no access to object attributes
 - i.e. there is no additional data other than the parameter
 - Similar to function in C
 - Distinguished by the modifier "**static**" in front of the method return type

3.11 What is “**this**” reference?

- A common confusion:
 - How does the method “know” which is the “object” it is currently communicating with? (as there could be many objects created from that class)
- Whenever a method is called,
 - a **reference to the calling object** is set automatically
 - Given the name “**this**” in Java, meaning “*this particular object*”
- All attributes/methods are then accessed implicitly through this reference

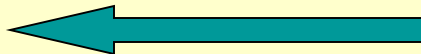
3.11 Object : What is “**this**” (1/2)

```
class BankAcct {  
    //... other code not shown ...  
    public int withdraw(double amount)  
    {  
        if (_balance < amount)  
            return 0;  
        _balance -= amount;  
        return 1;  
    }  
}
```

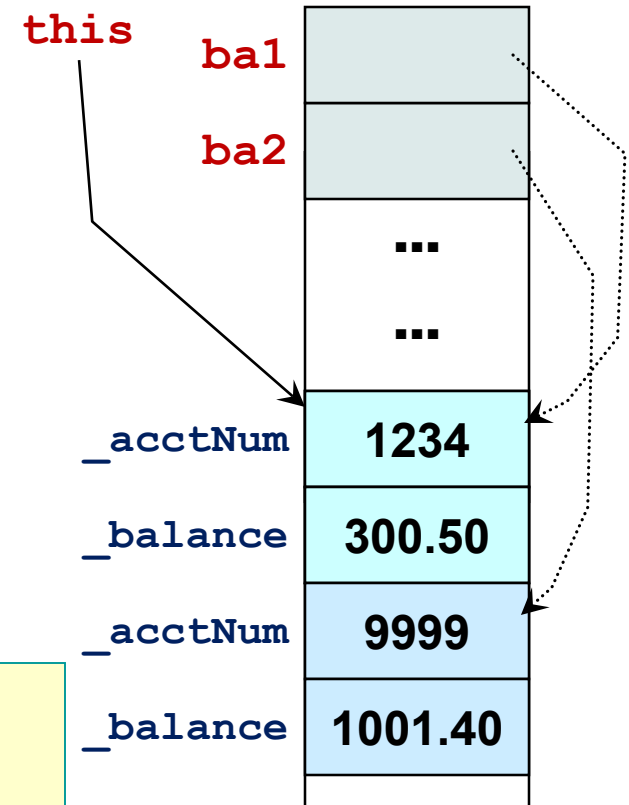
//Code fragment only

```
BankAcct ba1 = new BankAcct(1234, 300.50);  
BankAcct ba2 = new BankAcct(9999, 1001.40);
```

```
ba1.withdraw(100.00);  
ba2.withdraw(100.00);
```



after the 1st **withdraw()** method



3.11 Object : What is “this” (2/2)

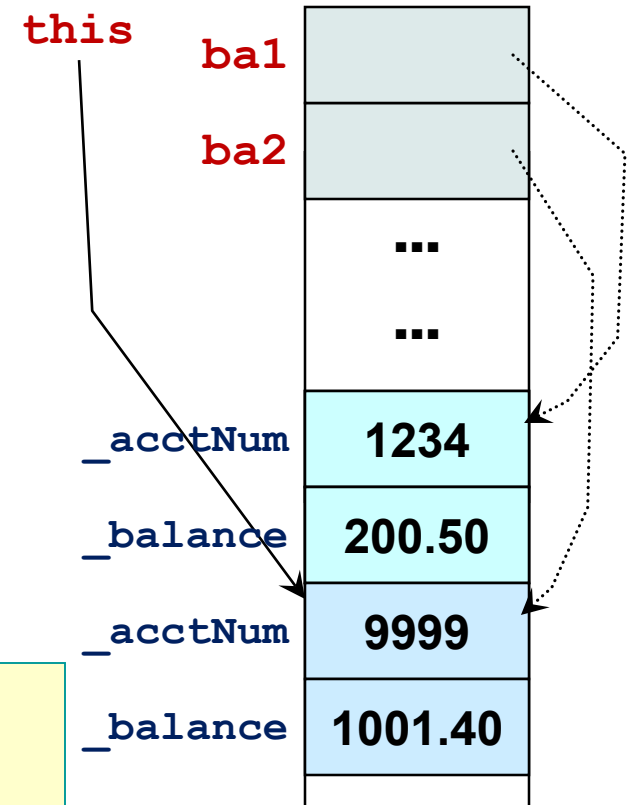
```
class BankAcct {  
    //... other code not shown ...  
    public int withdraw(double amount)  
    {  
        if (_balance < amount)  
            return 0;  
        _balance -= amount;  
        return 1;  
    }  
}
```

//Code fragment only

```
BankAcct ba1 = new BankAcct(1234, 300.50);  
BankAcct ba2 = new BankAcct(9999, 1001.40);
```

```
ba1.withdraw(100.00);  
ba2.withdraw(100.00);
```

after the 2nd **withdraw()** method



3.12 Service class and Client class (1/2)

- Preceding examples (**TestBankAcct.java** and **TestBankAcct2.java**)
 - The classes **BankAcct** and **TestBankAcct** (or **TestBankAcct2**) are in one Java file
 - Multiple classes may reside in a single Java file, provided there is only one **main()** method in the file.
 - **BankAcct** is the **service class**, while **TestBankAcct** (or **TestBankAcct2**) is the **client class** (also called **driver class**), which contains the **main()** method. The client is an application of the service class.
- Better design:
 - Put the service class and client class into separate files.
 - Example: **BankAcct.java** and **TestBankAcct3.java**
 - We can then write as many application programs (client classes) as necessary to use the service class.

3.12 Service class and Client class (2/2)

```
class BankAcct {  
    ...  
}
```

TestBankAcct.java

```
class TestBankAcct {  
    public static void main(String[] args) {  
        ...  
    }  
}
```

```
class BankAcct {  
    ...  
}
```

BankAcct.java

```
class TestBankAcct3 {  
    public static void main(String[] args) {  
        ...  
    }  
}
```

TestBankAcct3.java

```
javac BankAcct.java  
javac TestBankAcct3.java  
java TestBankAcct3
```

4. Predefined Java Classes

Introducing the Application
Programming Interface (API)

4.1 The API

- There are many predefined Java classes
 - Scanner
 - String
 - Math
 - and many more...
- Check out the API documentation
 - <http://docs.oracle.com/javase/7/docs/api/>

*Very
important!*

4.3 The String class (1/4)

■ The `String` class

- ❑ `import java.lang.String;` (which is default)

Java™ Platform
Standard Ed. 7

All Classes

Packages

- java.applet
- java.awt
- java.awt.color
- java.awt.datatransfer
- java.awt.dnd
- java.awt.event
- java.io
- java.lang
- java.math
- java.net
- java.nio
- java.rmi
- java.security
- java.sql
- java.util
- java.util.concurrent
- java.util.logging
- javax.swing
- javax.xml

Method Summary

Modifier and Type	Method and Description
char	charAt (int index) Returns the char value at the specified index.
int	codePointAt (int index) Returns the character (Unicode code point) at the specified index.
int	codePointBefore (int index) Returns the character (Unicode code point) before the specified index.
int	codePointCount (int beginIndex, int endIndex) Returns the number of Unicode code points in the specified text range of this String.
int	compareTo (String anotherString) Compares two strings lexicographically.
int	compareToIgnoreCase (String str) Compares two strings lexicographically, ignoring case differences.
String	concat (String str) Concatenates the specified string to the end of this string.
boolean	contains (CharSequence s) Returns true if and only if this string contains the specified sequence of char values.
boolean	contentEquals (CharSequence cs)

4.3 The String class (2/4)

TestString.java

```
class TestString {
    public static void main(String[] args) {
        String text = "I'm studying CS2040.";
        //or String text = new String("I'm studying CS2040.");
        //We'll explain the difference next time.
        System.out.println("text: " + text);
        System.out.println("text.length() = " + text.length());
        System.out.println("text.substring(5,8) = " +
                           text.substring(5,8));

        System.out.println("text.indexOf(\"in\") = " +
                           text.indexOf("in"));

        String newText = text + "How about you?";
        System.out.print("newText: " + newText);
        if (text.equals(newText))
            System.out.println("text and newText are equal.");
        else
            System.out.println("text and newText are not equal.");
    }
}
```

Why are there 2
backslashes \ here?

4.3 The `String` class (4/4)

- A `String` object is **immutable**:
 - Any method that modifies the `String` object actually constructs a new `String` object with the updated information.