CS1020 Lecture Note #2: **Object Oriented Programming**

A paradigm shift:

From procedural to object-oriented model

Lecture Note #2: **OOP**

- **Objectives:**
  - Understand major features of OOP
  - Able to use object oriented modeling to formulate solution
- **References:**
  - Chapter 2
    - Section 2.2: pages 119 to 130
    - Section 2.3: pages 131 to 150
  - Scanner class:
    - Chapter 1, Section 1.7, pages 74 to 75
  - String
    - Chapter 1, Section 1.5: pages 59 to 64

Lecture Overview

1. Review of Procedural Programming Model used in C
2. Introduction to Object Oriented Programming (**OOP**)
3. OOP Features in Java
4. Object Oriented Modeling
5. Predefined Java Classes

1. Programming Model

- All programming languages like C, C++, Java etc have an underlying **programming model**
  - Also known as **programming paradigms**
- **Programming Model** tells you:
  - How to organize the information and processes needed for a solution (program)
  - Allows/facilitates a certain way of thinking about the solution
  - Analogy : it is the "***world view***" of the language
- Various programming paradigms:
  - **Procedural**: C, Pascal
  - **Object Oriented**: Java, C++
  - **Functional:** Scheme, LISP
  - others

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;
```

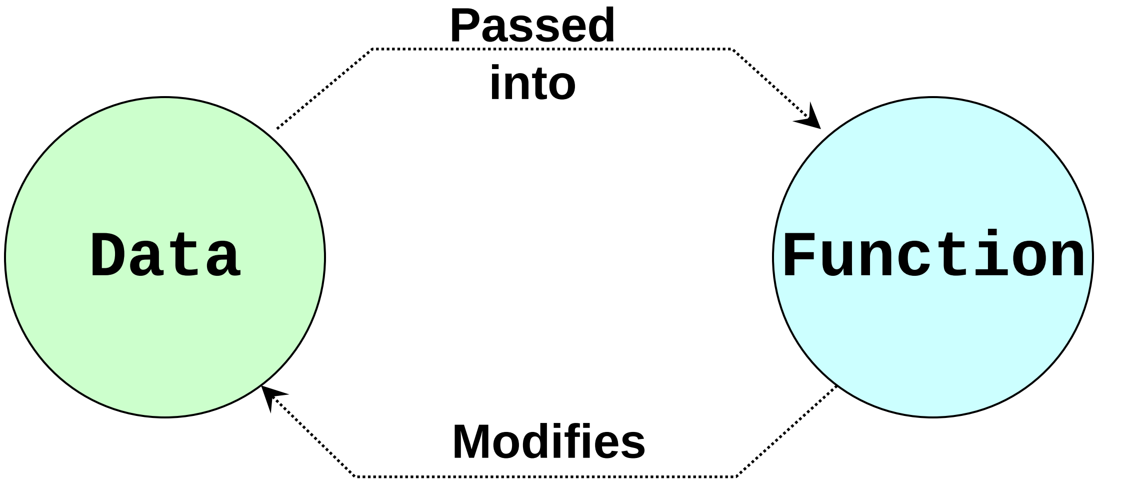
Structure to hold information for bank account

```
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:



`BankAcct` `structure`

`depost(...)`
`withdraw(...)`

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

1. Procedural language: **Summary**

- **Advantages:**
  - Closely resemble the execution model of computer
    - Efficient in execution and allows low level optimization
  - Less overhead when designing

- **Disadvantages**:
  - Harder to understand
    - Logical relation between data and functions is not clear
  - Hard to maintain
    - Requires self-imposed good programming discipline
  - Hard to extend / expand
    - e.g. How to introduce a new type of bank account?
      - Without affecting the current implementation
      - Without recoding the common stuff

Object Oriented Programming

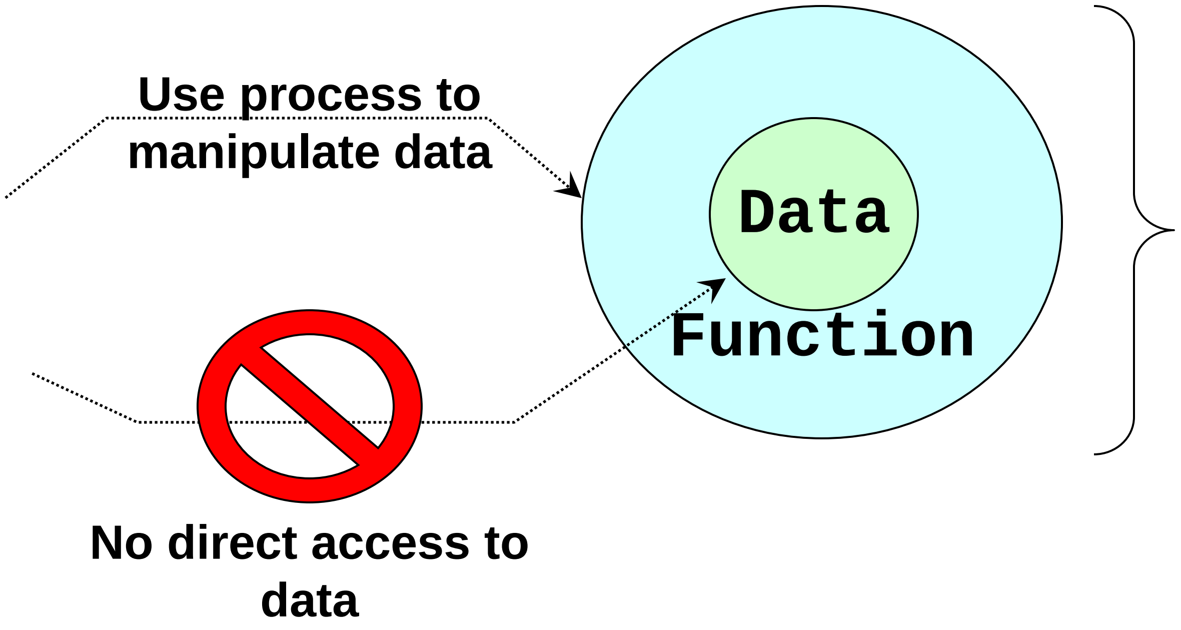
Definition and Motivation

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



Bank Account Object

Encapsulation of data and function

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

2. OO language: **Summary**

- **Advantages:**
  - Easier to design as it closely resemble the real world
  - Easier to maintain:
    - Modularity is enforced
    - Extensible

- **Disadvantages**:
  - Less efficient in execution
    - Further removed from low level execution
  - Program is usually longer with high design overhead

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 19 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

TestBankAcct2.java

```
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());
  }
}
```

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**

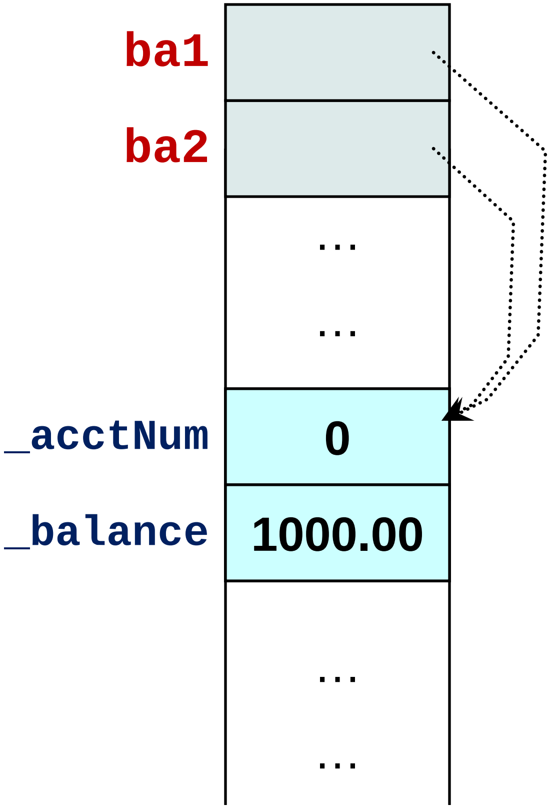
```
class BankAcct { …… }  //not shown

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

    ba2 = ba1;
    ba1.deposit(1000);

    ba2.print();
  }
}
```

ba1 has a balance of 0

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 (week 1, slide 34: class **`Factorial`**)
  - 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?

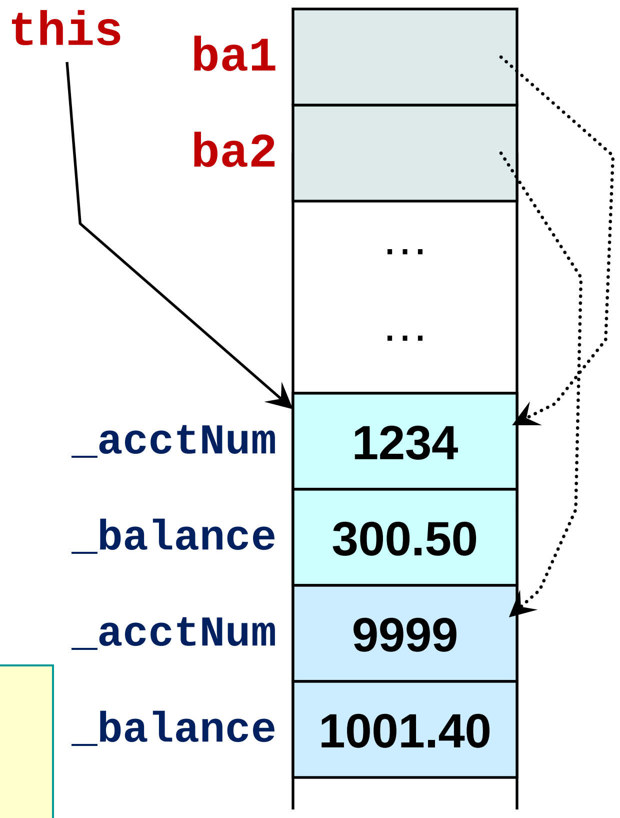
- A common confusion:
  - How does the method “know” which is the “object” it is currently communicating with? (as there could be many objects created from that class)
- Whenever a method is called,
  - a **reference to the calling object** is set automatically
  - Given the name **“this”** in Java, 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;
  }
}
```

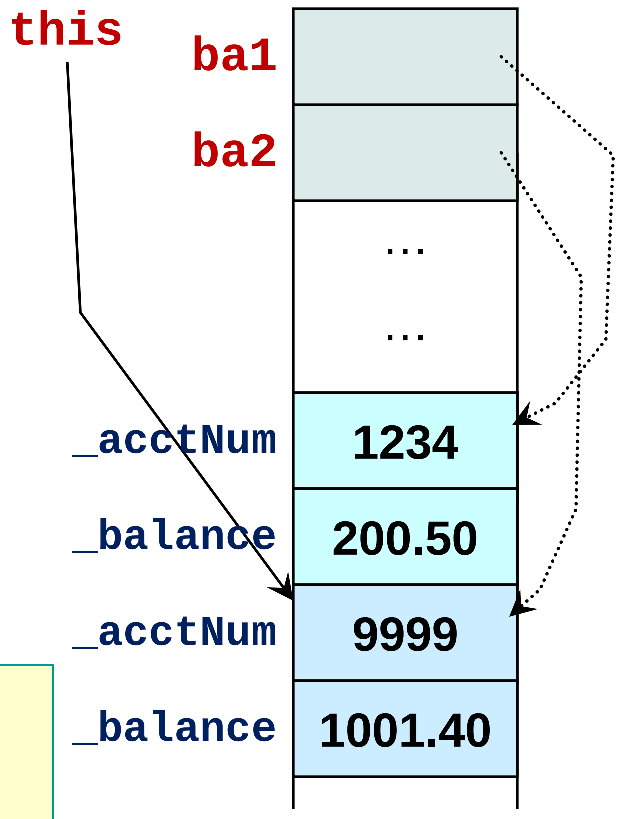
```
//Code fragment only

 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;
  }
}
```



this

ba1

ba2

...

...

_acctNum 1234

_balance 200.50

_acctNum 9999

_balance 1001.40

```
//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)

TestBankAcct.java

```
class BankAcct {
  ...
}

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
```

3.13 Quiz

TestBankAcct4.java

```
class TestBankAcct4 {
  public static void transfer(BankAcct fromAcct,
                  BankAcct toAcct, double amt) {
     fromAcct.withdraw(amt);
     toAcct.deposit(amt);
  }

  public static void main(String[] args) {
     BankAcct ba1 = new BankAcct(1, 234.56);
     BankAcct ba2 = new BankAcct(2, 1000.0);

     transfer(ba1, ba2, 200.50);

     ba1.print();
     ba2.print();
  }
}
```

What is the output?

```
Account Number: 1
Balance: $34.06
Account Number: 2
Balance: $1200.50
```

4. Object Oriented Modeling

How to approach problem in OO way

4. Problem Solving Approach: Review

- With procedural programming languages, we usually approach a problem in the following steps:
  1. Identify all information (data) known at the beginning
  2. Identify the desired end result (data)
  3. Figure out the necessary steps to transform (1) into (2)
  4. From (3), modularize the steps into separate functions
  5. Implement the functions in an incremental fashion

4. OO Problem Solving Approach

- With object oriented languages, the approach is slight different:
  1. Identify objects involved in the problem
     1. Identify the **capability** (functionality) of the objects
     2. Identify the **information** (data) kept by the objects
  2. Deduce classes from (1)
     - Generalize the objects found to design the classes
  3. Identify relationship between classes
     - Use the "**is-a**" and "**has-a**" rules to help
     - "**is-a**": Potential class hierarchy
     - "**has-a**": Association between separate classes
  4. Implement the classes in incremental fashion
     - Implement method by method

Man is-a Human

Man has-a Name

4.1 Design Principles

- Here are a few guidelines to good program design:
  1. Abstraction and Information Hiding
  2. Coherence and Coupling
  3. Top-down Design

- A brief overview for these principles are provided:
  - Actual applications will be highlighted in subsequent lectures

4.2 Abstraction

- **Abstraction:**
  - The process of isolating implementation details and extracting only **essential property** from an entity
  - Concentrate on "**what** can be done" but not "**how** to do it"

- For a class, concentrates on the **functionalities** (capabilities):
  - Give specification of the public methods first
  - Specify what a method does, but not how to do it

4.3 Information Hiding

- **Information Hiding:**
  - Only expose necessary information to outsider
  - Internal details should be "hidden":
    - Protected from outside influence

- In programming term:
  - Most (if not all) of the object attributes should be declared as private visibility
  - Refrain from providing methods that access and modify important attributes

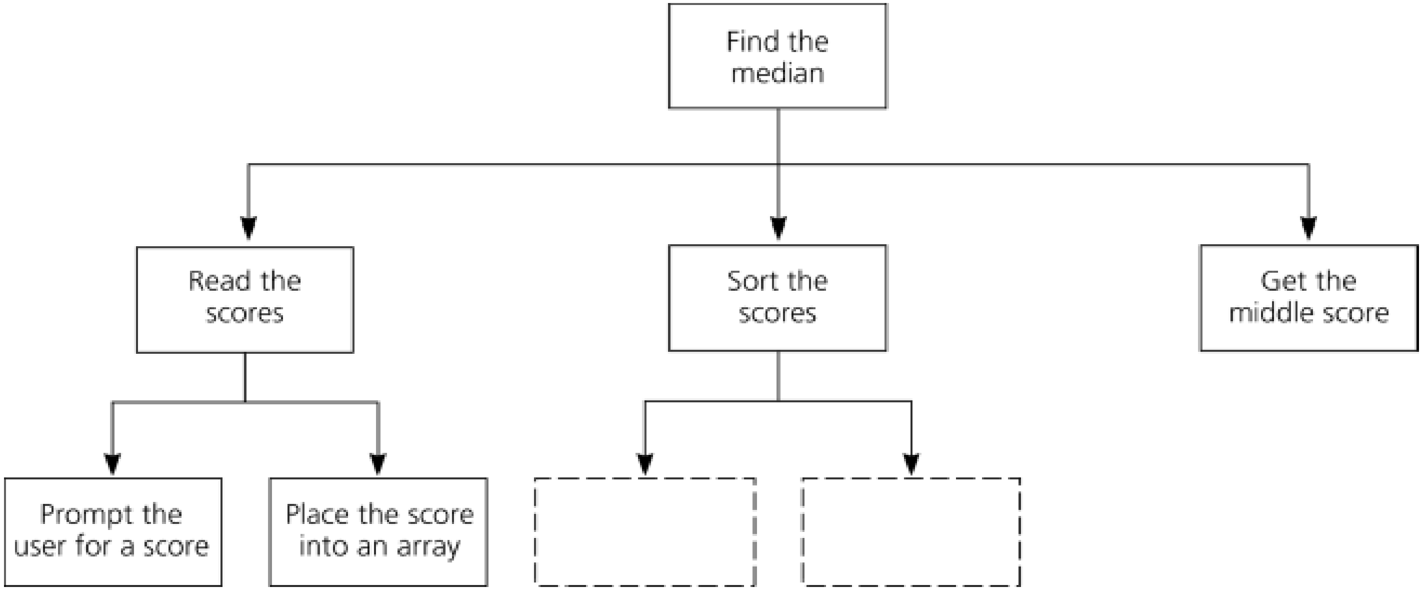
4.4 Coherence and Coupling

- **Coherence:**
  - A class should be about a **single entity only**
  - There should be a clear logical grouping of all the functionalities
- **Coupling:**
  - The interdependent relationship between classes
  - Two highly coupled classes results in:
    - Changes in one class will have a great impact on the other
- Coupling is **unavoidable** when you have independent components that work together:
  - Restrict the coupling to the absolute necessary

4.5 Top-down Design

- **Top-Down Design:**
  - Break down a task into successively more detailed subtasks
  - Also known as **functional decomposition**

- Example (find median):



5. Predefined Java Classes

Introducing the Application Programming Interface (API)

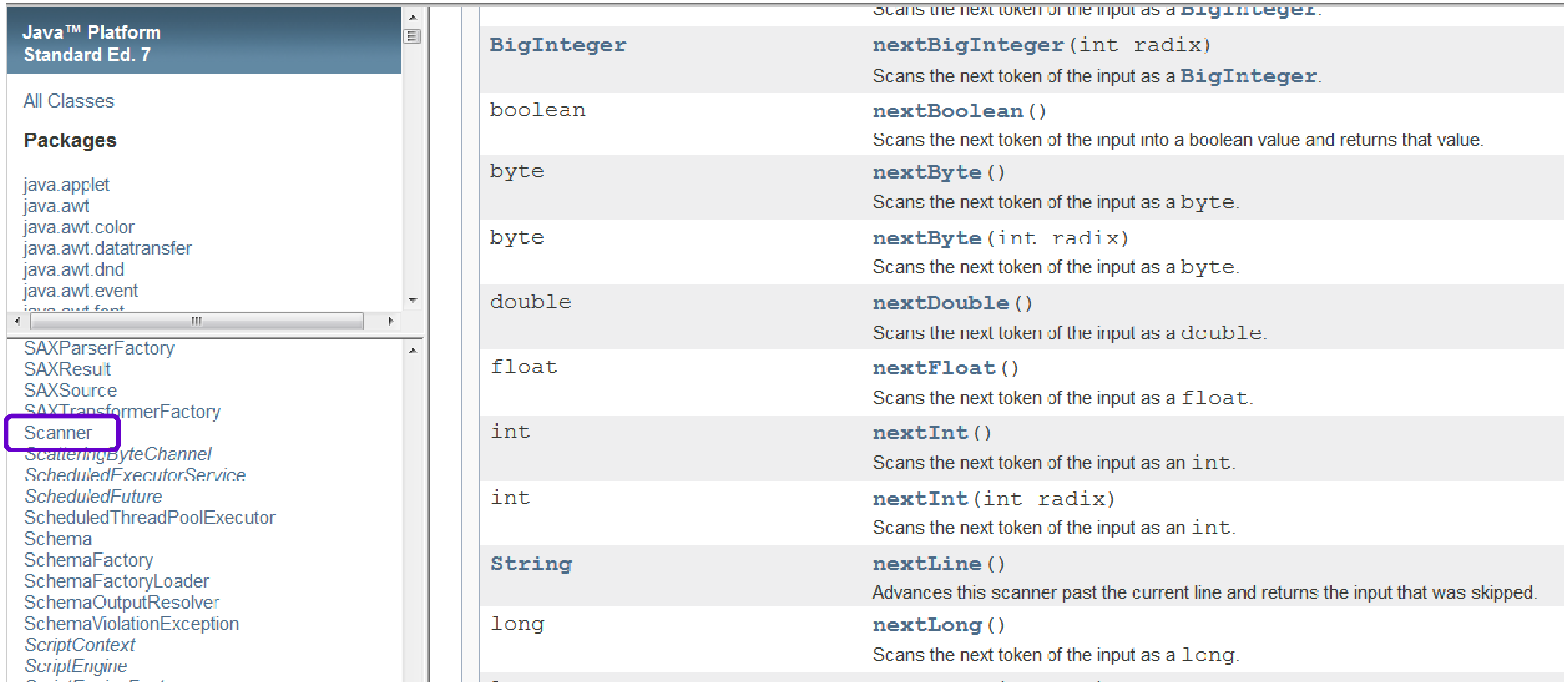
5.1 The API

- 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!

5.2 The **Scanner** class (1/3)

- From the API documentation:



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.awt.font

SAXParserFactory
SAXResult
SAXSource
SAXTransformerFactory
Scanner
ScatteringByteChannel
ScheduledExecutorService
ScheduledFuture
ScheduledThreadPoolExecutor
Schema
SchemaFactory
SchemaFactoryLoader
SchemaOutputResolver
SchemaViolationException
ScriptContext
ScriptEngine

	Scans the next token of the input as a `BigInteger`.
`BigInteger`	**`nextBigInteger`**`(int radix)` Scans the next token of the input as a `BigInteger`.
`boolean`	**`nextBoolean`**`()` Scans the next token of the input into a boolean value and returns that value.
`byte`	**`nextByte`**`()` Scans the next token of the input as a `byte`.
`byte`	**`nextByte`**`(int radix)` Scans the next token of the input as a `byte`.
`double`	**`nextDouble`**`()` Scans the next token of the input as a `double`.
`float`	**`nextFloat`**`()` Scans the next token of the input as a `float`.
`int`	**`nextInt`**`()` Scans the next token of the input as an `int`.
`int`	**`nextInt`**`(int radix)` Scans the next token of the input as an `int`.
`String`	**`nextLine`**`()` Advances this scanner past the current line and returns the input that was skipped.
`long`	**`nextLong`**`()` Scans the next token of the input as a `long`.

5.2 The **Scanner** class (2/3)

- In Lecture 1

TemperatureInteractive.java

```
import java.util.Scanner;

class TemperatureInteractive {

  public static void main(String[] args) {

     double fahrenheit, celcius;
     Scanner myScanner = new Scanner(System.in);

     System.out.print("Enter temperature in Fahrenheit: ");
     fahrenheit = myScanner.nextDouble();

     celcius = (5.0 / 9) * (fahrenheit – 32);
     System.out.println("Celcius: " + celcius);

  }

}
```

5.2 The **Scanner** class (3/3)

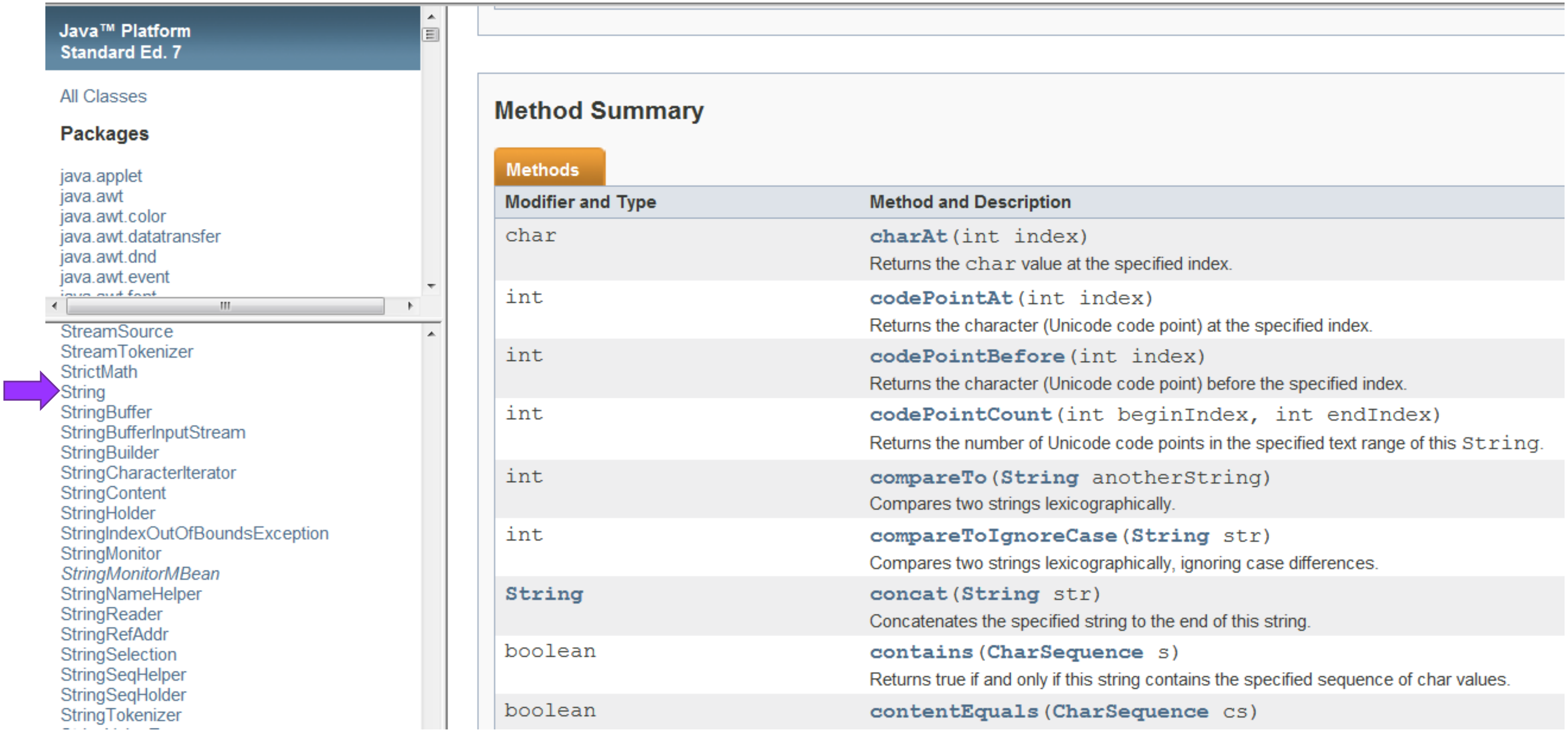
- In Lecture 1

ApproximatePI.java

```
import java.util.*; // using * in import statement

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

     int i, nTerms, sign = 1, denom = 1;
     double PI = 0;

     Scanner myScanner = new Scanner(System.in);

     System.out.print("Enter number of terms: ");
     nTerms = myScanner.nextInt();

     for (i = 0; i < nTerms; i++) {
          PI += 4.0 / denom * sign;
          sign *= -1;
          denom += 2;
     }
     System.out.printf("PI = %.6f\n", PI);
  }
}
```

5.3 The String class (1/4)

- The **String** class
  - import java.lang.String; (which is default)



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)

5.3 The **String** class (2/4)

TestString.java

```java
class TestString {
  public static void main(String[] args) {
    String text = "I'm studying CS1020.";
    //or String text = new String("I'm studying CS1020.");
    //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?

5.3 The **String** class (3/4)

Outputs

Explanations

```
text: I'm studying CS1020.
```

```
text.length() = 20
```

```
text.substring(5,8) = tud
```

```
text.indexOf("in") = 9
```

```
newText: I'm studying CS1020.How about you?
```

```
text and newText are not equal.
```

5.3 The **String** class (4/4)

- **`length()`**, **`substring()`**, **`indexOf()`**, **`equals()`** are just some of the methods in **`String`** class. Refer to the API for more.
- A **`String`** object is immutable:
  - Any method that modifies the **`String`** object actually constructs a new **`String`** object with the updated information.

Summary

Java Elements

```
Object Oriented Features:
    - Encapsulation
        class and object
        attribute and method
Object Oriented Modeling:
    - The 4 steps approach
Design Principles:
    - Abstraction and Information Hiding
    - Coherence and Coupling
    - Top-down design
Using Predefined Class
    - API
    - The Scanner class
    - The String class
```