Object Oriented Thinking

Luis Tarrataca
luis.tarrataca@gmail.com

CEFET-RJ

Introduction

What did the previous chapter talked about? Any ideas?

Introduction

What did the previous chapter talked about? Any ideas?

- Introduced objects and classes;
- How to define classes;
- How to create objects;
- Etc...

What do you think will be the focus of this chapter? Any ideas?

What do you think will be the focus of this chapter? Any ideas?

Focus of this chapter:

- Class design:
  - Understand the advantages of the object-oriented approach.
- Exploring differences between:
  - Procedural programming vs. object-oriented programming.

Lets start

Class Abstraction and Encapsulation

The first thing we need to ask is:

What is class abstraction? Any ideas?

Class Abstraction and Encapsulation

The first thing we need to ask is:

What is class abstraction and encapsulation? Any ideas?

- Class **abstraction** is the separation of class implementation from its use;
- Details of implementation are **encapsulated** and hidden from the user;

Lets see if we can make these concepts clearer...

Class Abstraction

Class abstraction:

- Separates class implementation from how the class is used;
- Creator of a class describes:
  - Functions of the class;
  - How the class can be used;
- **Class contract:**
  - Collection of methods and fields that are accessible;
  - Description of how these members are expected to behave,

Class Encapsulation

Class user does not need to know how the class is implemented:

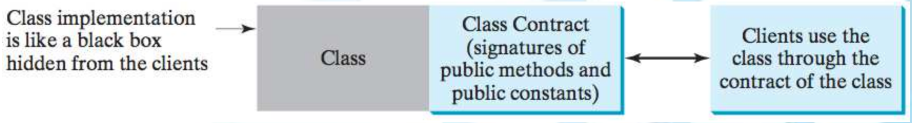

Class implementation is like a black box hidden from the clients
Class
Class Contract (signatures of public methods and public constants)
Clients use the class through the contract of the class

Figure: Class abstraction separates class implementation from the use of the class.(Source: (Liang, 2014))

- Implementation details are encapsulated and hidden from the user.
- This is called class **encapsulation.**

Example 1

Create a Circle object:

- Find the area of the circle:
    - Without knowing how the area is computed

For this reason:

- Class is a.k.a. an **abstract data type (ADT).**

Example 2

Consider computer assembly:

- Many components: CPU, memory, disk, etc.;
- Each component can be viewed as an **object**:
  - With properties and methods.
- Components work together by knowing how:
  - Each component is used and how it interacts with the others.
- No need to know how the components work internally:
  - Internal implementation is **encapsulated** and **hidden**

Example 3

Bank loan:

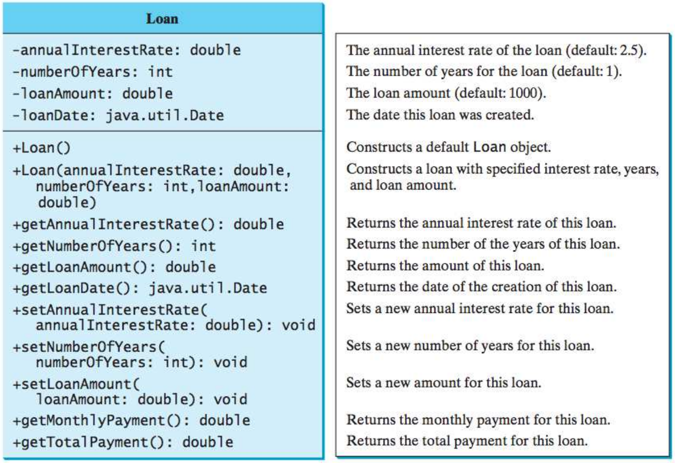
- Can be viewed as an object of a Loan class;
- Attributes:
    - Interest rate, loan amount, and loan period;
- Methods:
    - Compute monthly payment and total payment;
- As a user of the Loan class:
    - No need to know how these methods are implemented.

Exercise

Can you define the UML for the Loan example?

Can you define the Java code for the Loan example?

UML diagram serves as the **contract** for the Loan class:



Loan	
-annualInterestRate: double	The annual interest rate of the loan (default: 2.5).
-numberOfYears: int	The number of years for the loan (default: 1).
-loanAmount: double	The loan amount (default: 1000).
-loanDate: java.util.Date	The date this loan was created.
+Loan()	Constructs a default Loan object.
+Loan(annualInterestRate: double, numberOfYears: int, loanAmount: double)	Constructs a loan with specified interest rate, years, and loan amount.
+getAnnualInterestRate(): double	Returns the annual interest rate of this loan.
+getNumberOfYears(): int	Returns the number of the years of this loan.
+getLoanAmount(): double	Returns the amount of this loan.
+getLoanDate(): java.util.Date	Returns the date of the creation of this loan.
+setAnnualInterestRate(annualInterestRate: double): void	Sets a new annual interest rate for this loan.
+setNumberOfYears(numberOfYears: int): void	Sets a new number of years for this loan.
+setLoanAmount(loanAmount: double): void	Sets a new amount for this loan.
+getMonthlyPayment(): double	Returns the monthly payment for this loan.
+getTotalPayment(): double	Returns the total payment for this loan.

Figure: The Loan class models the properties and behaviors of loans.(Source: (Liang, 2014))

```
public class Loan {
    private double annualInterestRate;
    private int numberOfYears;
    private double loanAmount;
    private java. util .Date loanDate;

    /** Default constructor */
    public Loan() {
        this ( 2.5, 1, 1000);
    }

    /** Construct a loan with specified annual interest rate,
     number of years, and loan amount */
     public Loan( double annualInterestRate, int numberOfYears, double loanAmount ){
        this .annualInterestRate = annualInterestRate;
        this .numberOfYears = numberOfYears;
        this .loanAmount = loanAmount;
        this .loanDate = new java.util .Date();
     }

     /** Return annualInterestRate */
     public double getAnnualInterestRate() { return annualInterestRate; }

     /** Set a new annualInterestRate */
    public void setAnnualInterestRate(double annualInterestRate) { this .annualInterestRate = annualInterestRate; }

    /** Return numberOfYears */
    public int getNumberOfYears() { return numberOfYears ;}

     /** Set a new numberOfYears */
     public void setNumberOfYears(int numberOfYears) { this.numberOfYears = numberOfYears; }

    ...
...
    /** Return loanAmount */
    public double getLoanAmount() { return loanAmount; }

    /** Set a new loanAmount */
    public void setLoanAmount(double loanAmount) { this.loanAmount = loanAmount; }

     /** Find monthly payment */
    public double getMonthlyPayment() {
        double monthlyInterestRate = annualInterestRate / 1200;
        double monthlyPayment = loanAmount * monthlyInterestRate / (1 −
(1 / Math.pow(1 + monthlyInterestRate, numberOfYears * 12)));

        return monthlyPayment;
    }

    /** Find total payment */
    public double getTotalPayment() {
        double totalPayment = getMonthlyPayment() * numberOfYears * 12; return totalPayment;
    }

    /** Return loan date */
    public java. util .Date getLoanDate() { return loanDate; }
}
```

Class **definition** $\neq$ Class **use**:

- Class definition:
    - UML Diagram + Java code for class definition
- Class use:
    - Object instantiation of the class defined

Thinking in Objects

How many of you have used the C-programming language?

Thinking in Objects

How many of you have used the C-programming language?

What is the programming paradigm of C-language? Any ideas?

Thinking in Objects

How many of you have used the C-programming language?

What is the programming paradigm of C-language? Any ideas?

- Procedural programming

Thinking in Objects

How many of you have used the C-programming language?

What is the programming paradigm of C-language? Any ideas?

- Procedural programming

What is the procedural programming paradigm? Any ideas?

- Based on the concept of the procedure call, a.k.a.:
  - Procedures, routines, subroutines, or functions;
- Contain a series of computational steps to be carried out.
- **Focus** on designing methods;

What is the name of the programming paradigm taught in this class? Any ideas?

What is the name of the programming paradigm taught in this class? Any ideas?

- Object Oriented Programming ;)

Object-oriented paradigm couples:

- Data and methods together into object;
- Focuses on objects and operations on objects:
    - Combines power of the procedural paradigm;
    - With an added dimension:
        - Integrates data with operations into objects.
- Classes provide more flexibility / modularity for building reusable software.

So what are the main differences between procedural and O.O.P? Any ideas?

Procedural vs. O.O.P

Procedural programming:

- Data and operations on the data are separate;
- Requires passing data to methods

Object-oriented programming (1/2):

- Groups data and related operations in an object;
- Solves many of the problems of procedural programming;
- Mirrors the real world:
    - All objects are associated with both attributes and activities.

Procedural vs. O.O.P

Object-oriented programming (2/2):

- Improves software reusability. Why?
- Makes programs easier to develop and easier to maintain. Why?
- Java program can be viewed as a collection of cooperating objects.

Procedural vs. O.O.P

Object-oriented programming (2/2):

- Improves software reusability. Why?
    - The same class can be reused in the code;
- Makes programs easier to develop and easier to maintain. Why?
    - Complexity can be added to preexisting classes:
        - Through **class relationships**
- Java program can be viewed as a collection of cooperating objects.

Class Relationships

From all the previous concepts:

What is the main philosophy of O.O.P?

Class Relationships

From all the previous concepts:

What is the main philosophy of O.O.P?

- Everything is an object ;)
- Doing so requires the ability to design classes;

To design classes, we need to explore relationships among classes:

- Association;
- Aggregation;
- Composition;
- Inheritance.

Lets have a look into these:

- Inheritance relationship will be introduced in the next chapter.

Association

Association:

- General binary relationship describing an activity between two classes

Lets look at a specific example.

Example (1/2)

A student may take any number of courses:

- Association between the Student class and the Course class;

A faculty member may teach at most three courses course:

- Association between the Faculty class and the Course class.

A course:

- May have from five to sixty students;
- Is taught by only one faculty member

How can the following relationships be represented through UML?

Example (2/2)

How can the following relationships be represented through UML?

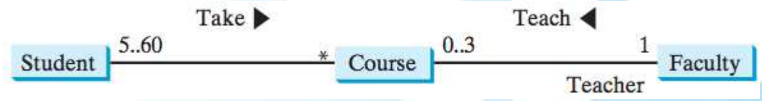


Figure: Student may take any number of courses. A faculty member may teach at most three courses. A course may have from five to sixty students, and a course is taught by only one faculty member. (Source: (Liang, 2014))

- Association is illustrated by a solid line between two classes:
  - Optional labels describing relationship (*e.g.:* ``Take'' and ``Teach'');
  - Optional black triangle indicating direction of the relationship
- Each class of an association may specify a **multiplicity**:
  - Number / interval specifying how many instances exist;
  - Character * means an unlimited number of objects,

Exercise

So the question now is:

How can these associations be implemented in Java? Any ideas?

Exercise

So the question now is:

How can these associations be implemented in Java? Any ideas?

- Using attributes and methods ;)

```
public class Student{

    private Course() courseList ;

    public void addCourse( Course s){
        ...
    }

}
```

```
public class Course {
    private Student() classList ;
    private Faculty faculty ;
    public void addStudent( Student s) {
        ... }
    public void setFaculty ( Faculty
        faculty ) { ... }

}
```

```
public class Faculty {
    private Course() courseList ;

    public void addCourse( Course c) {
        ... }

}
```

Aggregation and Composition

Aggregation:

- Special form of association representing ownership between two objects:
  - Owner object / class is called **aggregating object / class**;
  - Subject object / class is called **aggregated object / class**;
- Models *has-a* relationships;

Aggregation and Composition

Composition:

- If an object is exclusively owned by an aggregating object;

Can you thing of any examples illustrating $\neq$'s between aggregation and composition?

Can you thing of any examples illustrating $\neq$'s between aggregation and composition?

- Example: ``student has a name'':
    - Composition between Student and Name;
- Example: ``student has an address'':
    - Aggregation, since address can be shared between multiple students;

Example

How can the previous relationships be modelled in UML?

Example

How can the previous relationships be modelled in UML?

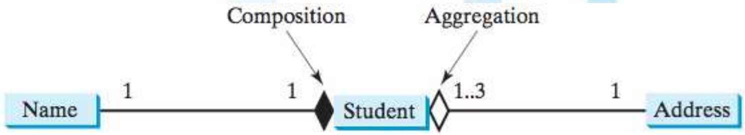


Figure: Each student has a name and an address. (Source: (Liang, 2014))

- Each student has one address;
- Each address can be shared by up to 3 students;
- Filled diamond denotes composition;
- Empty diamond denotes aggregation;

How can composition / aggregation be represented in Java? Any ideas?

How can composition / aggregation be represented in Java? Any ideas?

- Aggregation relationship is represented as an attribute in the aggregating class.

```java
public class Name {
    ...

}
```

Listing 1: Aggregated class

```java
public class Student {
    private Name name;
    private Address address;
    ...
}
```

Listing 2: Aggregating class

```java
public class Address {
    ...

}
```

Listing 3: Aggregated class

Aggregation may exist between objects of the same class:

- Example: person may have a supervisor.

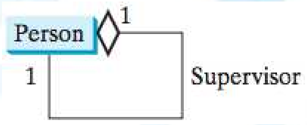


Figure: A person may have a supervisor. (Source: (Liang, 2014))

How can the previous relationship be modelled in Java? Any ideas?

How can the previous relationship be modelled in Java? Any ideas?

```
public class Person {
    // The type for the data is the class itself
    private Person supervisor;
    ...
}
```

What if a person can have multiple supervisors? Any ideas?

What if a person can have multiple supervisors? Any ideas?

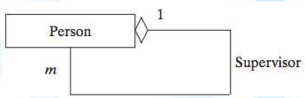


Figure: A person can have several supervisor. (Source: (Liang, 2014))

```
public class Person{
    ...
    private Person()  supervisors ;
}
```

Case Study: Designing the **Course** Class

Suppose you need to process course information:

- Each course has a name and has students enrolled;
- You should be able to add/drop a student to/from the course;
- Course can be created using constructor Course(String courseName);
- Students can be:
    - Added to the course through addStudent(String student);
    - Dropped from the course through dropStudent(String student);
    - Returned through getStudents();

What is the UML diagram for the previous domain?

What is the respective Java code for the previous domain?

Lets answer these individually =)

What is the UML diagram for the previous domain?

What is the UML diagram for the previous domain?

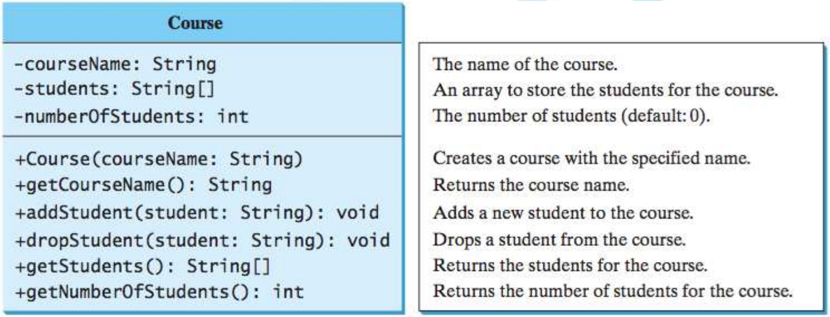

Course
-courseName: String
-students: String[]
-numberOfStudents: int
+Course(courseName: String)
+getCourseName(): String
+addStudent(student: String): void
+dropStudent(student: String): void
+getStudents(): String[]
+getNumberOfStudents(): int
The name of the course.
An array to store the students for the course.
The number of students (default: 0).
Creates a course with the specified name.
Returns the course name.
Adds a new student to the course.
Drops a student from the course.
Returns the students for the course.
Returns the number of students for the course.

Figure: The Course class models the courses. (Source: (Liang, 2014))

What is the respective Java code for the previous domain?

```
public class Course {
    private String courseName;
    private String () students = new String (100);
    private int numberOfStudents;

    public Course(String courseName) {
        this .courseName = courseName;
    }

    public void addStudent(String student){
        students (numberOfStudents) = student;
        numberOfStudents++;
    }

    public String () getStudents(){
        return students;
    }

    public int getNumberOfStudents(){
        return numberOfStudents;
    }

    public String getCourseName(){
        return courseName;
    }

    public void dropStudent(String student){
        ...
    }
```

Case Study: Designing a Class for Stacks

Develop a stack that holds data in a LIFO:

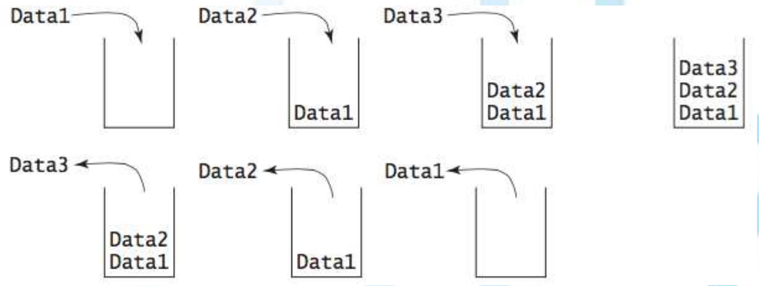


Figure: Stack holds data in a last-in, first-out fashion(Source: (Liang, 2014))

What is the UML diagram for the previous domain?

What is the respective Java code for the previous domain?

Lets answer these individually =)

What is the UML diagram for the previous domain?

What is the UML diagram for the previous domain?

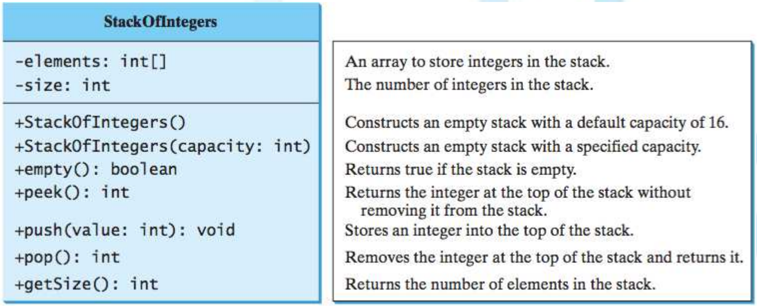

StackOfIntegers
-elements: int[]
-size: int
+StackOfIntegers()
+StackOfIntegers(capacity: int)
+empty(): boolean
+peek(): int
+push(value: int): void
+pop(): int
+getSize(): int
An array to store integers in the stack.
The number of integers in the stack.
Constructs an empty stack with a default capacity of 16.
Constructs an empty stack with a specified capacity.
Returns true if the stack is empty.
Returns the integer at the top of the stack without removing it from the stack.
Stores an integer into the top of the stack.
Removes the integer at the top of the stack and returns it.
Returns the number of elements in the stack.

Figure: StackOfIntegers class encapsulates the stack storage and provides the operations for manipulating the stack. (Source: (Liang, 2014))

What is the respective Java code for the previous domain?

```
public class StackOfIntegers{
    private int () elements,
    private int size ;
    public static final int DEFAULT_CAPACITY = 16;

    public StackOfIntegers(){
        this ( DEFAULT_CAPACITY ),
    }

    public StackOfIntegers( int capacity ){
        elements = new int( capacity );
    }

    public void push( int value ){
        if ( size >= elements.length ){
            int () temp = new int( elements.length ∗ 2 );
            System.arraycopy( elements, 0, temp, 0, elements.length );
            elements = temp;
        }
        elemetns( size ++ ) = value;
    }

    public int pop(){ return elements( ——size ); }

    public int peek(){ return elements( size — 1 ); }

    public boolean empty(){ return size == 0; }

    public int getSize (){ return size ; }
}
```

The String Class

A String object is immutable:

- Contents cannot be changed once string is created;
- Multitude of useful methods:
  - charAt(index) returns character at specified index;
  - length() returns string size;
  - substring method returns a substring;
  - indexOf and lastIndexOf methods return first or last index of a character;

The String class has:

- 15 constructors;
- More than 40 methods for manipulating strings.

Are the students supposed to know each and every one of these?

The String class has:

- 15 constructors;
- More than 40 methods for manipulating strings.

Are the students supposed to know each and every one of these?

- Yes!

The String class has:

- 15 constructors;
- More than 40 methods for manipulating strings.

Are the students supposed to know each and every one of these?

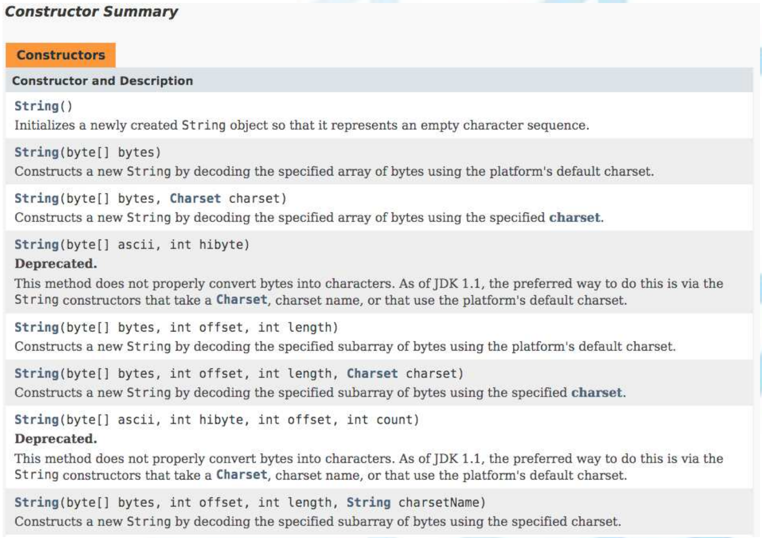
- Yes! ;)

How can we find out about the constructors / methods available for String? Any ideas?

How can we find out about the constructors / methods available for String? Any ideas?

Through the Java API Documentation:

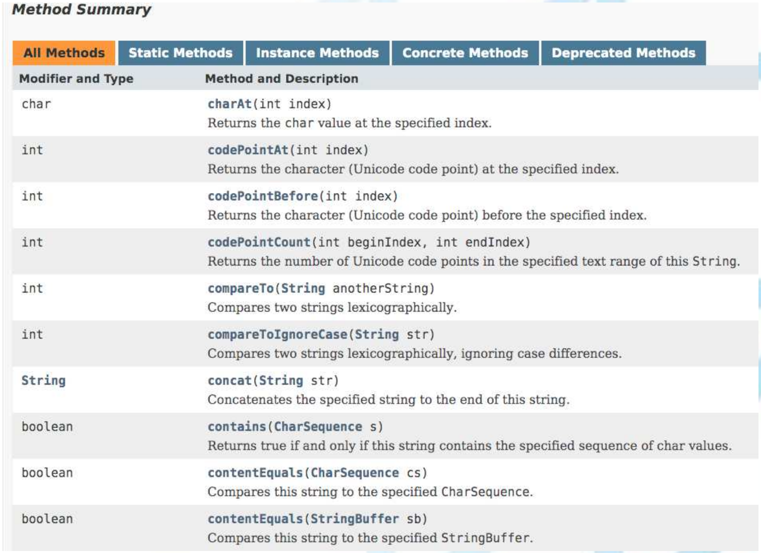
- Google ``javadoc 8´´ ;)
- Contains detailed documentation for the Java API specification:
    - Thousands of classes;



Constructor Summary

Constructors

Constructor and Description
`String()` Initializes a newly created `String` object so that it represents an empty character sequence.
`String(byte[] bytes)` Constructs a new `String` by decoding the specified array of bytes using the platform's default charset.
`String(byte[] bytes, Charset charset)` Constructs a new `String` by decoding the specified array of bytes using the specified **charset**.
`String(byte[] ascii, int hibyte)` **Deprecated.** This method does not properly convert bytes into characters. As of JDK 1.1, the preferred way to do this is via the `String` constructors that take a `Charset`, charset name, or that use the platform's default charset.
`String(byte[] bytes, int offset, int length)` Constructs a new `String` by decoding the specified subarray of bytes using the platform's default charset.
`String(byte[] bytes, int offset, int length, Charset charset)` Constructs a new `String` by decoding the specified subarray of bytes using the specified **charset**.
`String(byte[] ascii, int hibyte, int offset, int count)` **Deprecated.** This method does not properly convert bytes into characters. As of JDK 1.1, the preferred way to do this is via the `String` constructors that take a `Charset`, charset name, or that use the platform's default charset.
`String(byte[] bytes, int offset, int length, String charsetName)` Constructs a new `String` by decoding the specified subarray of bytes using the specified charset.



Method Summary

All Methods | Static Methods | Instance Methods | Concrete Methods | Deprecated Methods

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)` Compares this string to the specified `CharSequence`.
boolean	`contentEquals(StringBuffer sb)` Compares this string to the specified `StringBuffer`.

Important notice:

- Java API Specification manuals should always be open:
  - When developing code useful to have documentation on-hand =)
- There are a lot of other important Java classes already defined:
  - Date
  - Time
  - StringBuilder, StringBuffer;
  - LinkedList, HashMap, TreeMap (RB Tree), etc...
- All with appropriate examples that will help you =)

Where to focus your study

After this class you should be able to:

- Apply class abstraction to develop software;
- Explore $\neq$'s between procedural and OO paradigm;
- Express relationships between classes;
- Design programs using the object-oriented paradigm

References I

Liang, Y. (2014).

Introduction to Java Programming.

Pearson Education.