Lecture 00: Introduction

SE313, Software Design and Architecture

Damla Oguz

Course Information

- The course will be given by Damla Oguz.
- The course will be assisted by Sencer Dogan and Erinç Cibil.
- Please enroll to the course at *lectures.yasar.edu.tr*.
- Office hours will be announced.
- Books
  - Head First Design Patterns: A Brain-Friendly Guide, Bert Bates, Kathy Sierra, Eric Freeman, Elisabeth Robson
  - Software Engineering, Ian Sommerville
- Recommendations
  - Design Patterns: Elements of Reusable Object-Oriented Software, Gang of Four
  - Data Structures & Algorithms in Java, M. T. Goodrich, R. Tamassia, M. H. Goldwasser
  - The Java™ Tutorials, https://docs.oracle.com/javase/tutorial/java/index.html
- Attendance will be taken.

Course Information: Evaluation

- Midterm: 35%
- Labs: 20%
- Final examination: 45%
- Plagiarism is forbidden.

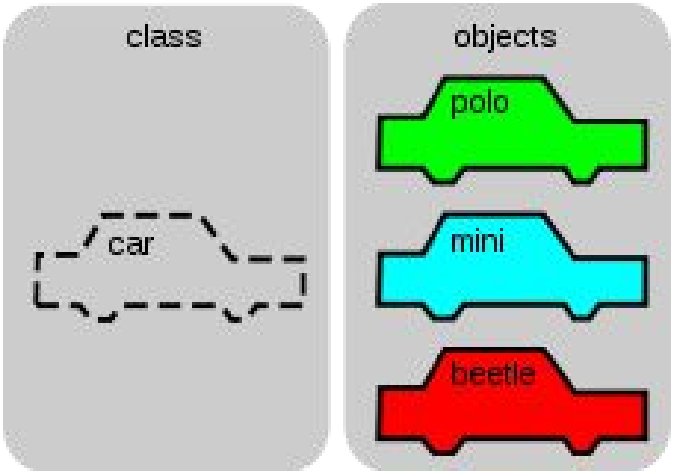
Review of Object-Oriented Programming Concepts

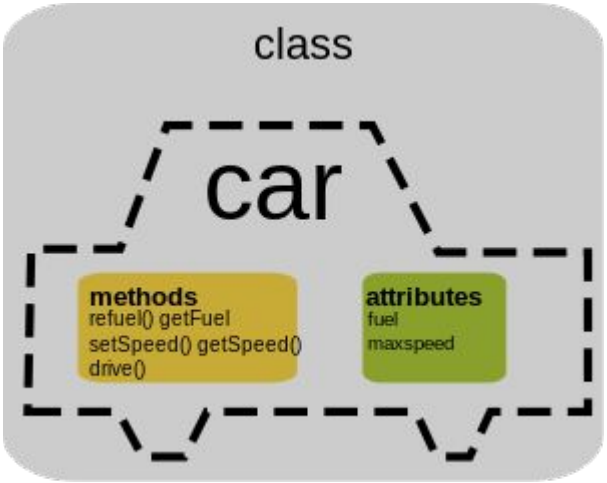
In this lecture, we will review:

- Object-Oriented (OO) Design Goals
- OO Principles
- Inheritance
- Polymorphism
- Interfaces
- Abstract Classes

Object-Oriented Programming Concepts: A Review

Object-oriented programming is a type of programming paradigm based on the concept of **classes** and instances of classes called **objects**.



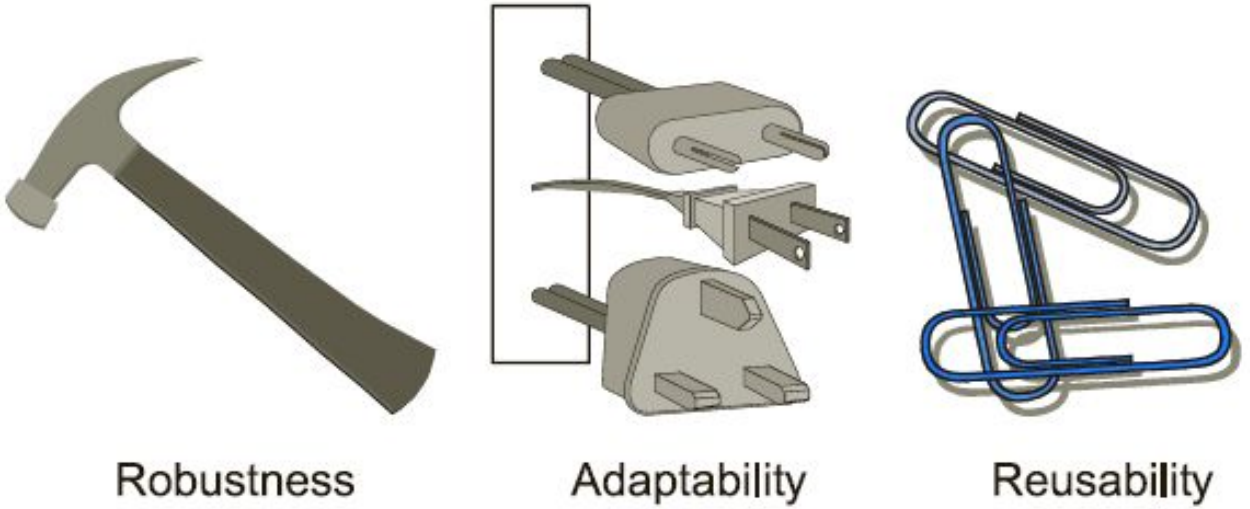


https://en.wikibooks.org/wiki/A-level_Computing_2009/AQA/Problem_Solving,_Programming,_Operating_Systems,_Databases_and_Networking/Programming_Concepts/Object-oriented_programming_(OOP)

Object-Oriented Programming Concepts: A Review

- The main “**actors**” in the object-oriented paradigm are called **objects.**
- Each **object** is an **instance** of a **class**.
- Each **class** presents to the outside world a concise and consistent view of the objects without going into too much unnecessary detail or giving others access to the inner workings of the objects.
- The class definition typically specifies the **data fields** (also known as **instance variables**), that an object contains, as well as the **methods** (**operations**) that an object can execute.

Object-Oriented Design Goals



Object-Oriented Design Goals: Robustness

- Every good programmer wants to develop software that is correct, which means that a program produces the right output for all the anticipated inputs in the program's application.
- In addition, we want software to be **robust**, that is, capable of handling unexpected inputs that are not explicitly defined for its application.
    - For example, if a program is expecting a positive integer and instead is given a negative integer, then the program should be able to recover gracefully from this error.
- In life-critical applications, where a software error can lead to injury or loss of life, software that is not robust be deadly.

Object-Oriented Design Goals: Adaptability

- Software needs to be able to evolve over time in response to changing user needs, desires and environment.
- Thus, another important goal of quality software is that it achieves **adaptability** (also called **evolvability**).
- Related to this concept is **portability**, which is the ability of software to run with minimal change on different hardware and operating system platforms.

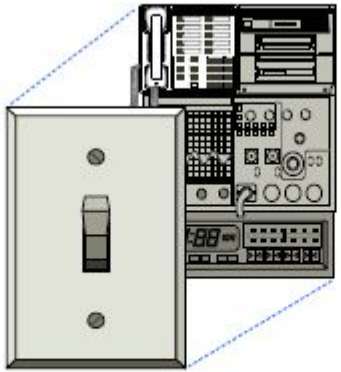
Object-Oriented Design Goals: Reusability

- The same code should be usable as a component of different systems in various applications.
- Developing quality software can be an expensive enterprise, and its cost can be offset somewhat if the software is designed in a way that makes it easily **reusable** in future applications.

Object-Oriented Design Principles

- Object-oriented design principles intend to facilitate the goals we have just discussed.

Abstraction



Encapsulation

Modularity

Object-Oriented Design Principles: Abstraction

- The notion of **abstraction** is to distill a complicated system down to its most fundamental parts.
- Typically, describing the parts of a system involves naming them and explaining their functionality.
- Applying the abstraction paradigm to the design of data structures gives rise to **abstract data types** (ADTs).
    - An ADT is a mathematical model of a data structure that specifies the **type of data stored,** the **operations supported** on them, and the types of **parameters of the operations**.
    - An ADT specifies **what** each operation does, but **not how** it does it.

ADTs in Java

- In Java, an ADT can be expressed by an **interface**, which is simply **a list of method declarations**, where each method has an **empty body**.
- An ADT is realized by a **class** in Java which defines the data being stored and the operations supported by the objects that are instances of the class.
- Unlike interfaces, **classes** specify **how** the operations are performed in the body of each method.
- A **Java class** is said to **implement an interface** if its methods include all the methods declared in the interface, thus providing a body for them.

Object-Oriented Design Principles: Encapsulation

- **Encapsulation** states that different components of a software system should not reveal the internal details of their respective implementations.
- One of the main advantages of encapsulation is that it gives one programmer freedom to implement the details of a component.
- Encapsulation yields **robustness** and **adaptability**.
  - Allows the implementation details of parts of a program to change without adversely affecting other parts.
  - Makes it easier to fix bugs or add new functionality with relatively local changes to a component.

Object-Oriented Design Principles: Modularity

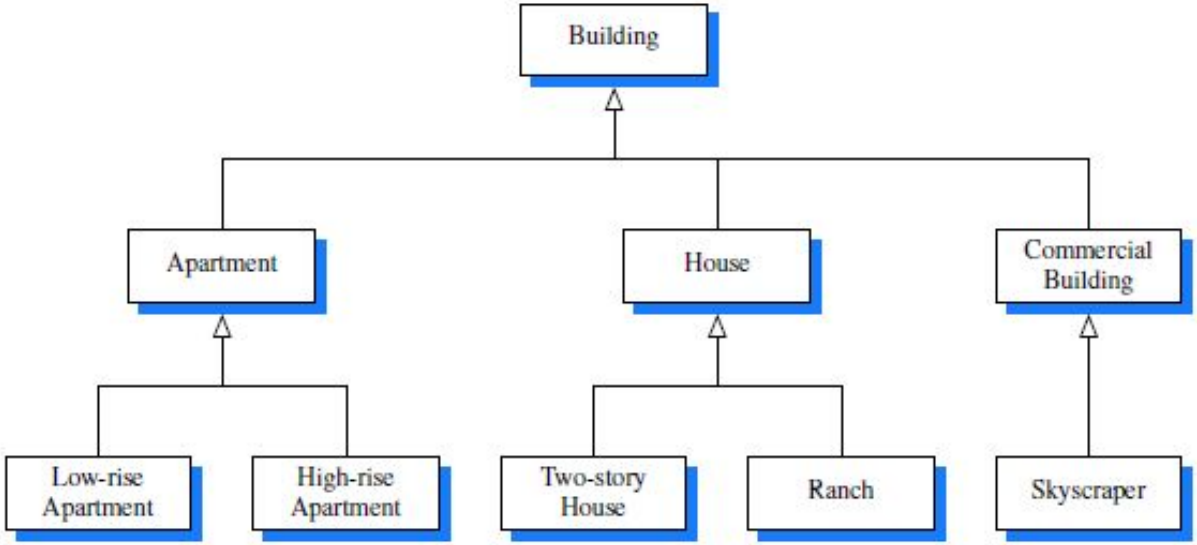
- Modern software systems typically consist of several different components that must interact correctly in order for the entire system to work properly.
- Keeping these interactions straight requires that these different components be well organized.
- **Modularity** refers to an **organizing principle** in which different components of a software system are divided into **separate functional units**.

Inheritance

- A natural way to organize various structural components of a software package is in a **hierarchical** fashion, which groups similar abstract definitions together in a level-by-level manner that goes from **specific** to more **general** as one traverses up in the hierarchy.

Inheritance

- A house **is a** building and a ranch **is a** house.



Inheritance

- A hierarchical design is useful in software development, as common functionality can be grouped at the most general level, thereby promoting reuse of code, while differentiated behaviors can be viewed as extensions of the general case.
- In object-oriented programming, the mechanism for a **modular** and **hierarchical organization** is a technique known as **inheritance**.

Inheritance

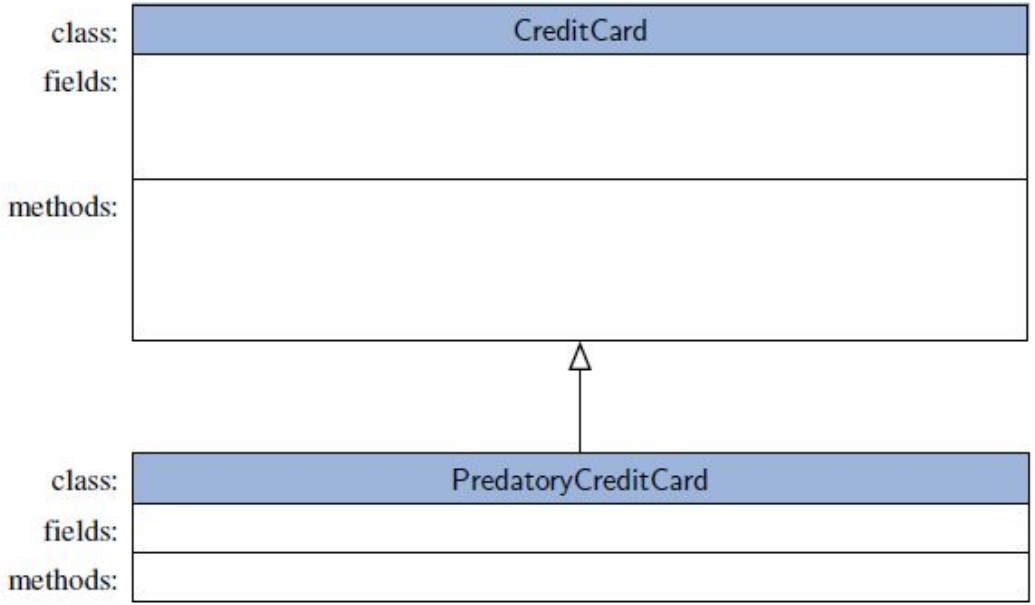
- Inheritance allows a new class to be defined based upon an existing class as the starting point.
- In object-oriented terminology, the **existing class** is typically described as the **base class**, **parent class**, or **superclass**, while the **newly defined class** is known as the **subclass** or **child class**.
- We say that the **subclass extends the superclass**.

Inheritance

- When inheritance is used, the subclass automatically inherits, as its starting point, all methods from the superclass (other than constructors).
- The subclass can differentiate itself from its superclass in two ways.
  - It may **augment** the superclass by adding new fields and new methods.
  - It may also specialize existing behaviors by providing a new implementation that **overrides** an existing method.

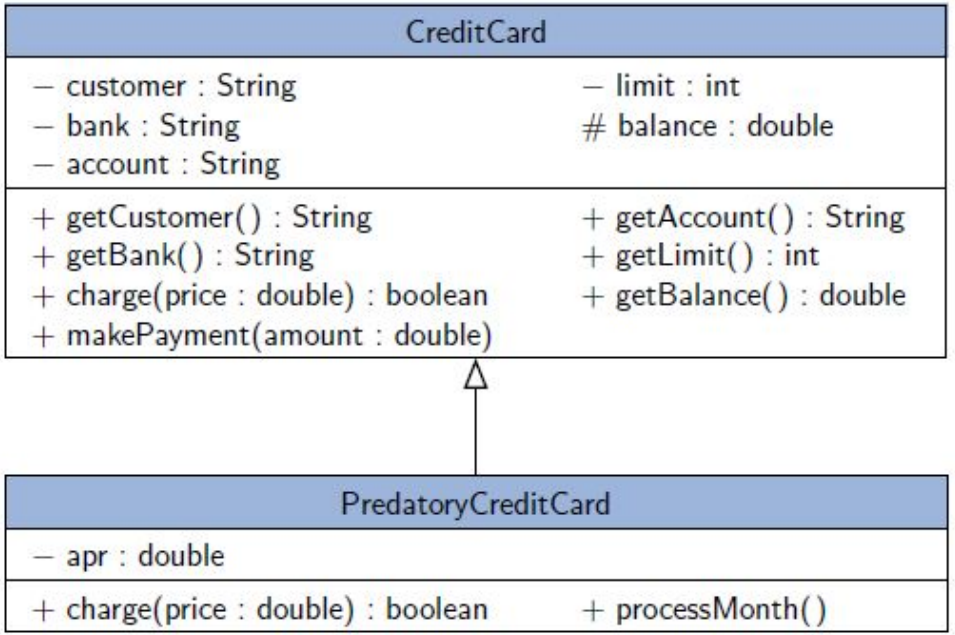
Class Inheritance Diagram in UML

- **Base class / parent class / superclass:** CreditCard
- **Subclass / child class:** PredatoryCreditCard



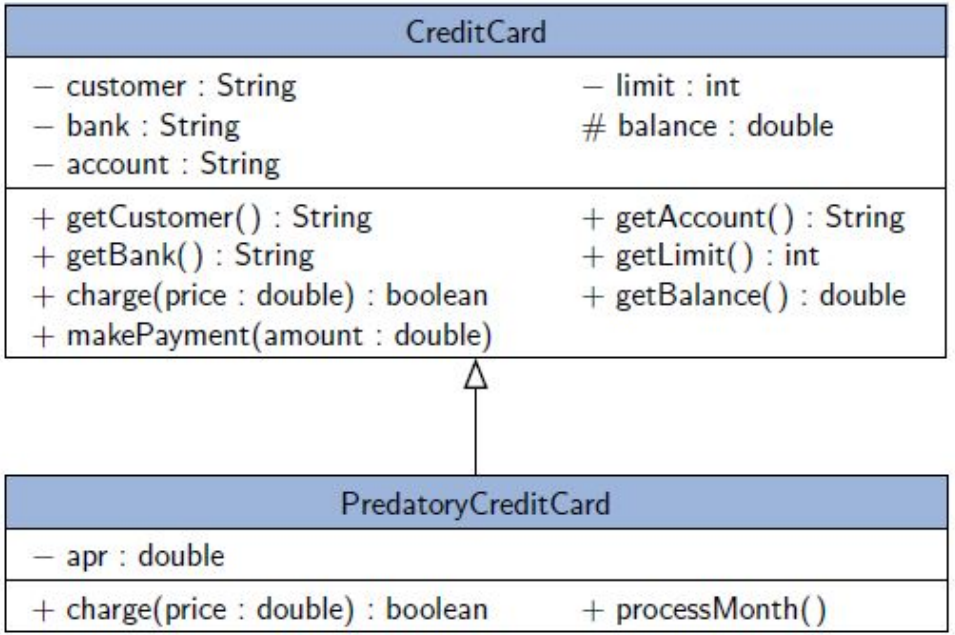
Class Inheritance Diagram in UML

- The new class will differ from the original in two ways:
  - (1) if an attempted charge is rejected because it would have exceeded the credit limit, a $5 fee will be charged,
  - (2) there will be a mechanism for assessing a monthly interest charge on the outstanding balance, using an annual percentage rate (apr) specified as a constructor parameter.



Class Inheritance Diagram in UML

- The PredatoryCreditCard class **augments** the original CreditCard class,
  - adding a new instance **variable** named **apr** to store the annual percentage rate, and
  - adding a new **method** named **processMonth** that will assess interest charges.
- It also specializes its superclass by overriding the original **charge** method.



Polymorphism

- The word **polymorphism** literally means “**many forms**”.
- Subclasses of a class can define their own unique behaviors and yet share some of the same functionality of the parent class.
- Consider the base class Shape. This class has a member function called double Area() { }
- You specialize the Shape class and extend Square, Circle and Rectangle classes.
- Each have their own specialized data fields, methods, etc., but also the Area() method.
- Java selects the appropriate method to call during runtime.

Interfaces

- An interface is a collection of method declarations with **no data** and **no bodies**.
  - The methods of an interface are always empty; they are simply method signatures.
  - Interfaces do not have constructors and they cannot be directly instantiated.
- When a class implements an interface, it must implement all of the methods declared in the interface.

Interfaces

- Suppose that we want to create an inventory of antiques we own, categorized as objects of various types and with various properties. We might wish to identify some of our objects as sellable, in which case they could implement the Sellable **interface**.
- We can then define a concrete **class**, Photograph that **implements** the Sellable **interface**, indicating that we would be willing to sell any of our Photograph objects.
    - This class defines an object that implements each of the methods of the Sellable interface, as required.
    - It adds a method, isColor, which is specialized for Photograph objects.

Example: Sellable Interface

```
1   /** Interface for objects that can be sold. */
2   public interface Sellable {
3
4     /** Returns a description of the object. */
5     public String description( );
6
7     /** Returns the list price in cents. */
8     public int listPrice( );
9
10    /** Returns the lowest price in cents we will accept. */
11    public int lowestPrice( );
12  }
```

Example: Photograph Class

```
1  /** Class for photographs that can be sold. */
2  public class Photograph implements Sellable {
3    private String descript;                          // description of this photo
4    private int price;                                // the price we are setting
5    private boolean color;                            // true if photo is in color
6
7    public Photograph(String desc, int p, boolean c) {    // constructor
8      descript = desc;
9      price = p;
10     color = c;
11   }
12
13   public String description( ) { return descript; }
14   public int listPrice( ) { return price; }
15   public int lowestPrice( ) { return price/2; }
16   public boolean isColor( ) { return color; }
17 }
```

Example: Transport Interface

- Another kind of object in our collection might be something we could transport. For such objects, we define the Transport interface.
- Example: Transport Interface

```
1  /** Interface for objects that can be transported. */
2  public interface Transportable {
3    /** Returns the weight in grams. */
4    public int weight( );
5    /** Returns whether the object is hazardous. */
6    public boolean isHazardous( );
7  }
```

Interfaces

- We could then define the class BoxedItem for miscellaneous antiques that we can sell, pack, and ship.
- Thus, the class BoxedItem **implements** the methods of the Sellable **interface** and the Transportable **interface**, while also adding specialized methods to set an insured value for a boxed shipment and to set the dimensions of a box for shipment.

Example: BoxedItem Class

```
1  /** Class for objects that can be sold, packed, and shipped. */
2  public class BoxedItem implements Sellable, Transportable {
3    private String descript;          // description of this item
4    private int price;                // list price in cents
5    private int weight;               // weight in grams
6    private boolean haz;              // true if object is hazardous
7    private int height=0;             // box height in centimeters
8    private int width=0;              // box width in centimeters
9    private int depth=0;              // box depth in centimeters
10   /** Constructor */
11   public BoxedItem(String desc, int p, int w, boolean h) {
12     descript = desc;
13     price = p;
14     weight = w;
15     haz = h;
16   }
17   public String description( ) { return descript; }
18   public int listPrice( ) { return price; }
19   public int lowestPrice( ) { return price/2;       }
20   public int weight( ) { return weight; }
21   public boolean isHazardous( ) { return haz; }
22   public int insuredValue( ) { return price*2; }
23   public void setBox(int h, int w, int d) {
24     height = h;
25     width = w;
26     depth = d;
27   }
28 }
```

Multiple Inheritance

- The ability of **extending from more than one type** is known as multiple inheritance.
- In Java, **multiple inheritance** is allowed **for interfaces** but **not for classes**.
- Interfaces do not define fields or method bodies, yet classes typically do.

Abstract Classes

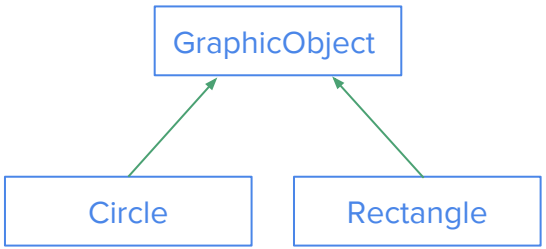
- In Java, an abstract class serves a role somewhat between that of a traditional class and that of an interface.
- Like an interface, an abstract class may define signatures for one or more methods without providing an implementation of those method bodies; such methods are known as abstract methods.
- However, **unlike an interface**, **an abstract class** may define **one or more fields** and **any number of methods with implementation** (so-called concrete methods).
- An abstract class may also extend another class and be extended by further subclasses.

Abstract Classes

- An abstract class may not be instantiated, that is, no object can be created directly from an abstract class.
- In a sense, it remains an incomplete class.
- A subclass of an abstract class must provide an implementation for the abstract methods of its superclass, or else remain abstract.
- The use of abstract classes in Java is limited to single inheritance, so a class may have at most one superclass, whether concrete or abstract.

Abstract Class Example

```
abstract class GraphicObject {
    int x, y;
    ...
    void moveTo(int newX, int newY) {
        ...
    }
    abstract void draw();
    abstract void resize();
}
```




```
class Circle extends GraphicObject {
    void draw() {
        ...
    }
    void resize() {
        ...
    }
}
class Rectangle extends GraphicObject {
    void draw() {
        ...
    }
    void resize() {
        ...
    }
}
```

https://docs.oracle.com/javase/tutorial/java/landl/abstract.html

References

- Goodrich M. T., Tamassia R., Goldwasser M. H.; Data Structures and Algorithms in C++, Wiley, 2014.
- The Java™ Tutorials, https://docs.oracle.com/javase/tutorial/java/index.html.