Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

# Object-Oriented Programming

## Terminology

- Each **object** created in a program is an **instance** of a **class**.
- Each class presents to the outside world a concise and consistent view of the objects that are instances of this class, without going into too much unnecessary detail or giving others access to the inner workings of the objects.
- The class definition typically specifies **instance variables**, also known as **data members**, that the object contains, as well as the **methods**, also known as **member functions**, that the object can execute.

## Goals

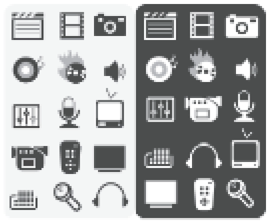
- Robustness
  - We want software to be capable of handling unexpected inputs that are not explicitly defined for its application.
- Adaptability
  - Software needs to be able to evolve over time in response to changing conditions in its environment.
- Reusability
  - The same code should be usable as a component of different systems in various applications.

## Abstract Data Types

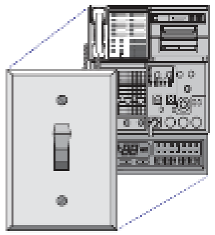
- **Abstraction** is to distill a system to its most fundamental parts.
- Applying the abstraction paradigm to the design of data structures gives rise to **abstract data types** (ADTs).
- An ADT is a model of a data structure that specifies the **type** of data stored, the **operations** supported on them, and the types of parameters of the operations.
- An ADT specifies what each operation does, but not how it does it.
- The collective set of behaviors supported by an ADT is its **public interface**.

# Object-Oriented Design Principles

- Modularity
  - Breaking down a software system into smaller, independent modules that can function, develop, and be tested separately.
- Abstraction
  - Hiding the complex reality while only exposing the necessary parts. It often means providing a simplified interface or model of a more complex system and therefore enable programmers to implement functionalities without the need of understanding the details.
- Encapsulation
  - Closely related to abstraction.
  - Bundling of related data and methods into a single unit or class, and restricting the direct Access.
  - Shields object’s internal state and ensures it remains constant.
  - Prevents the code to be accessed outside the class.


Modularity

Abstraction


Encapsulation

# Interfaces and Abstract Classes

- The main structural element in Java that enforces an application programming interface (API) is an **interface**.
- An interface is a collection of method declarations with no data and no bodies.
  - They don't have constructors, therefore they cannot be directly instantiated.
  - When a class **implements** an interface, it must implement all of the methods declared in the interface.
- An abstract class also cannot be instantiated, but it can define one or more common methods that all implementations of the abstraction will have.
  - (I.N.) For some differences between interface and abstract classes, refer to: https://www.javatpoint.com/difference-between-abstract-class-and-interface
- Interface defines a contract of methods that classes must adhere to when they implement the interface.
  - A single class can implement multiple interfaces.
  - Primarily used to define capabilities.
- Abstract class can provide both unimplemented (abstract) and implemented methods and it may also have fields.
  - Classes can only extend one abstract class.
  - Used to provide shared functionality or default behavior to subclasses.

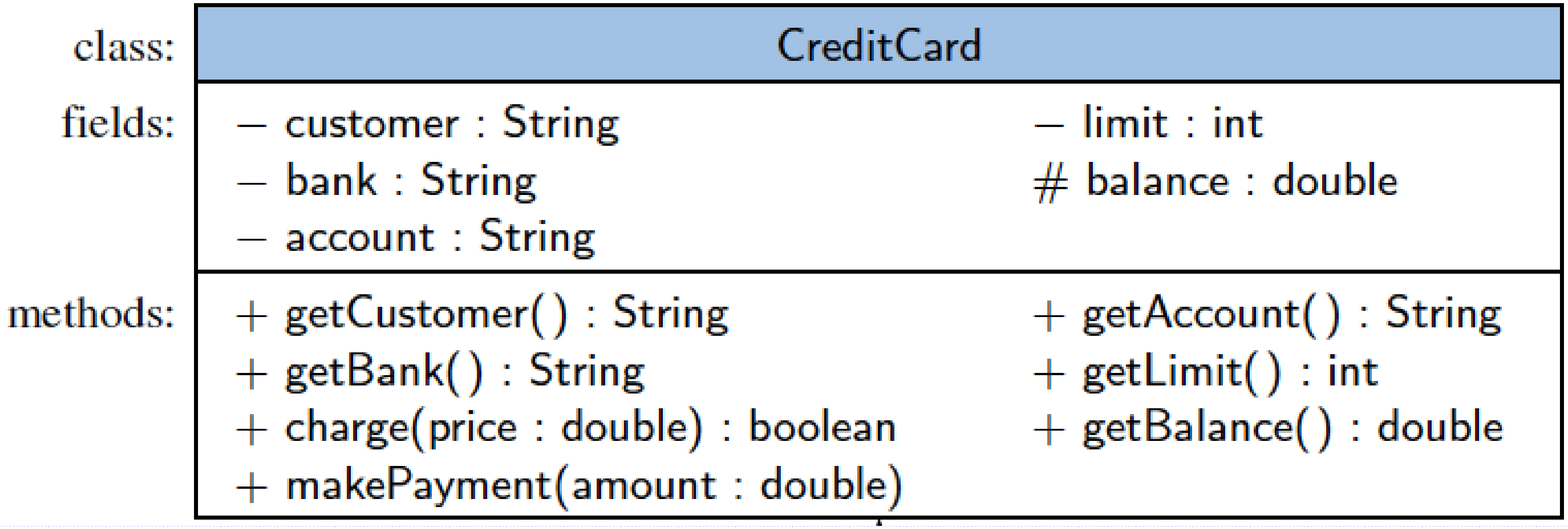
## Object-Oriented Software Design

- **Responsibilities**: Divide the work into different actors, each with a different responsibility.
- **Independence**: Define the work for each class to be as independent from other classes as possible.
- **Behaviors**: Define the behaviors for each class carefully and precisely, so that the consequences of each action performed by a class will be well understood by other classes that interact with it.

# Unified Modeling Language (UML)

A **class diagram** has three portions.

1. The name of the class
2. The recommended instance variables
3. The recommended methods of the class.



| | | |
|---|---|---|
| class: | CreditCard | |
| fields: | − customer : String<br>− bank : String<br>− account : String | − limit : int<br># balance : double |
| methods: | + getCustomer( ) : String<br>+ getBank( ) : String<br>+ charge(price : double) : boolean<br>+ makePayment(amount : double) | + getAccount( ) : String<br>+ getLimit( ) : int<br>+ getBalance( ) : double |

## Class Definitions

- A class serves as the primary means for abstraction in object-oriented programming.
- In Java, every variable is either a base type or is a reference to an instance of some class.
- A class provides a set of behaviors in the form of member functions (also known as **methods**), with implementations that belong to all its instances.
- A class also serves as a blueprint for its instances, effectively determining the way that state information for each instance is represented in the form of **attributes** (also known as **fields**, **instance variables**, or **data members**).

## Constructors

- A user can create an instance of a class by using the **new** operator with a method that has the same name as the class.
- Such a method, known as a **constructor**, has as its responsibility is to establish the state of a newly object with appropriate initial values for its instance variables.

## Inheritance

- A mechanism for a modular and hierarchical organization is **inheritance**.
- This allows a new class to be defined based upon an existing class as the starting point.
- 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.
- There are two ways in which a subclass can differentiate itself from its superclass:
  - A subclass may specialize an existing behavior by providing a new implementation that overrides an existing method.
  - A subclass may also extend its superclass by providing brand new methods.

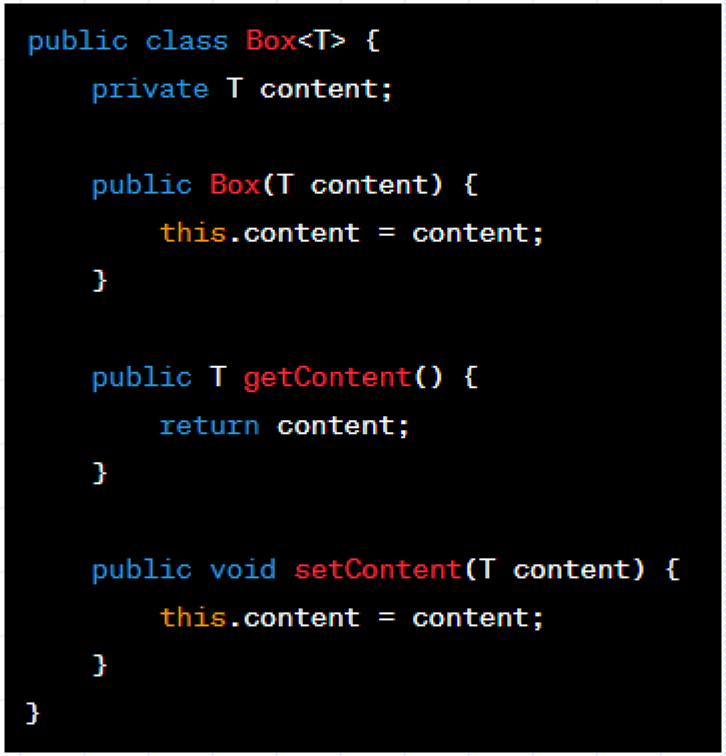
## Generics

- Java includes support for writing generic classes and methods that can operate on a variety of data types while often avoiding the need for explicit casts.
- The generics framework allows us to define a class in terms of a set of formal type parameters, which can then be used as the declared type for variables, parameters, and return values within the class definition.
- Those formal type parameters are later specified when using the generic class as a type elsewhere in a program.

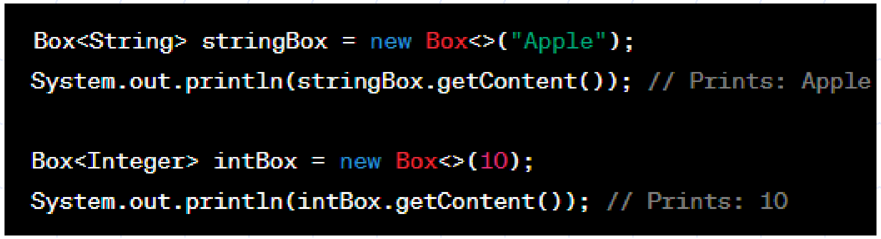
## Generics

- Helps the programmer to reuse the code for whatever type he/she wishes.
  - Let’s say you wrote a generic method for sorting an array of objects.
  - It allows the programmer to use the same method for Integer arrays, Double arrays and even String arrays.

## Generics

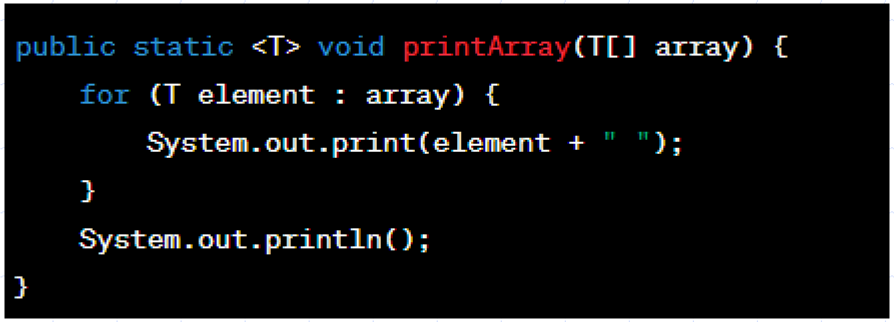


```java
public class Box<T> {
    private T content;

    public Box(T content) {
        this.content = content;
    }

    public T getContent() {
        return content;
    }

    public void setContent(T content) {
        this.content = content;
    }
}
```

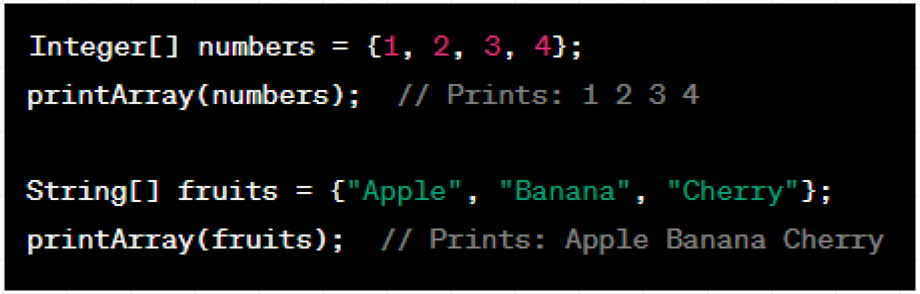


```java
Box<String> stringBox = new Box<>("Apple");
System.out.println(stringBox.getContent()); // Prints: Apple

Box<Integer> intBox = new Box<>(10);
System.out.println(intBox.getContent()); // Prints: 10
```

## Generics



```java
public static <T> void printArray(T[] array) {
    for (T element : array) {
        System.out.print(element + " ");
    }
    System.out.println();
}
```



```java
Integer[] numbers = {1, 2, 3, 4};
printArray(numbers);  // Prints: 1 2 3 4

String[] fruits = {"Apple", "Banana", "Cherry"};
printArray(fruits);  // Prints: Apple Banana Cherry
```