Introduction to OOP (cont.)

Vũ Thị Hồng Nhạn

(vthnhan@vnu.edu.vn)

Dept. of software engineering, FIT

UET, VNU HN

Classes vs. Objects

	Class	Object
What:	A Data Type	A Variable
Where:	Has its own file	Scattered around the project
Naming convention:	CamelCase (starts with an upper case)	camelCase (starts with a lower case)
Examples:	Country	australia
	Book	lordOfTheRings
	Pokemon	pikachu

Contents

- Java programming language
- Classes
- Fields/attributes
- Methods
- Access modifiers
- Constructors

Reference

- *Giáo trình Lập trình HĐT,Chapter* 3, 4

Brief history

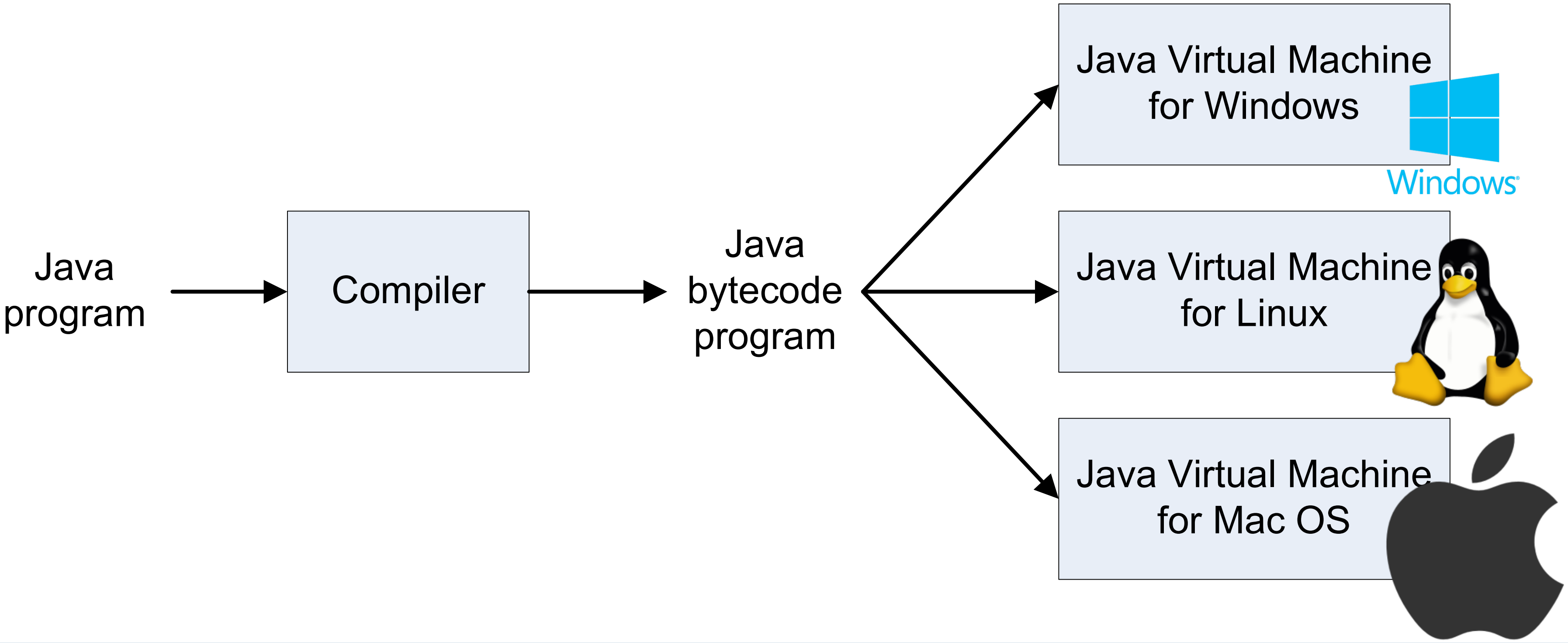
- 1991: developed by Sun Micrsoft as a programming language for embedded environments
  - Oak was the first name of Java
- Java 1.0.2, 1.1
  - "Write ONCE, run ANYWHERE"
  - Slow
  - Used in web applications (applets)
- Java 2 (version 1.2 – 1.4)
  - Fast & more powerful
  - 3 platforms: J2ME, J2SE, J2EE
- Java 5,6,7 (version 1.5…)
  - Much more upgraded!

Structure of a java program

- A set of object classes
- Usually each class is a source code file named **the same as** the class name
  - Increase the independence
  - Easy to modify, save compilation time

Biên dịch

- Java source code is compiled into **bytecode**
- Bytecode is platform independent
- Bytecode is executed by JVM (**J**ava **V**irtual **M**achine)


Java program
Compiler
Java bytecode program
Java Virtual Machine for Windows
Windows
Java Virtual Machine for Linux
Java Virtual Machine for Mac OS

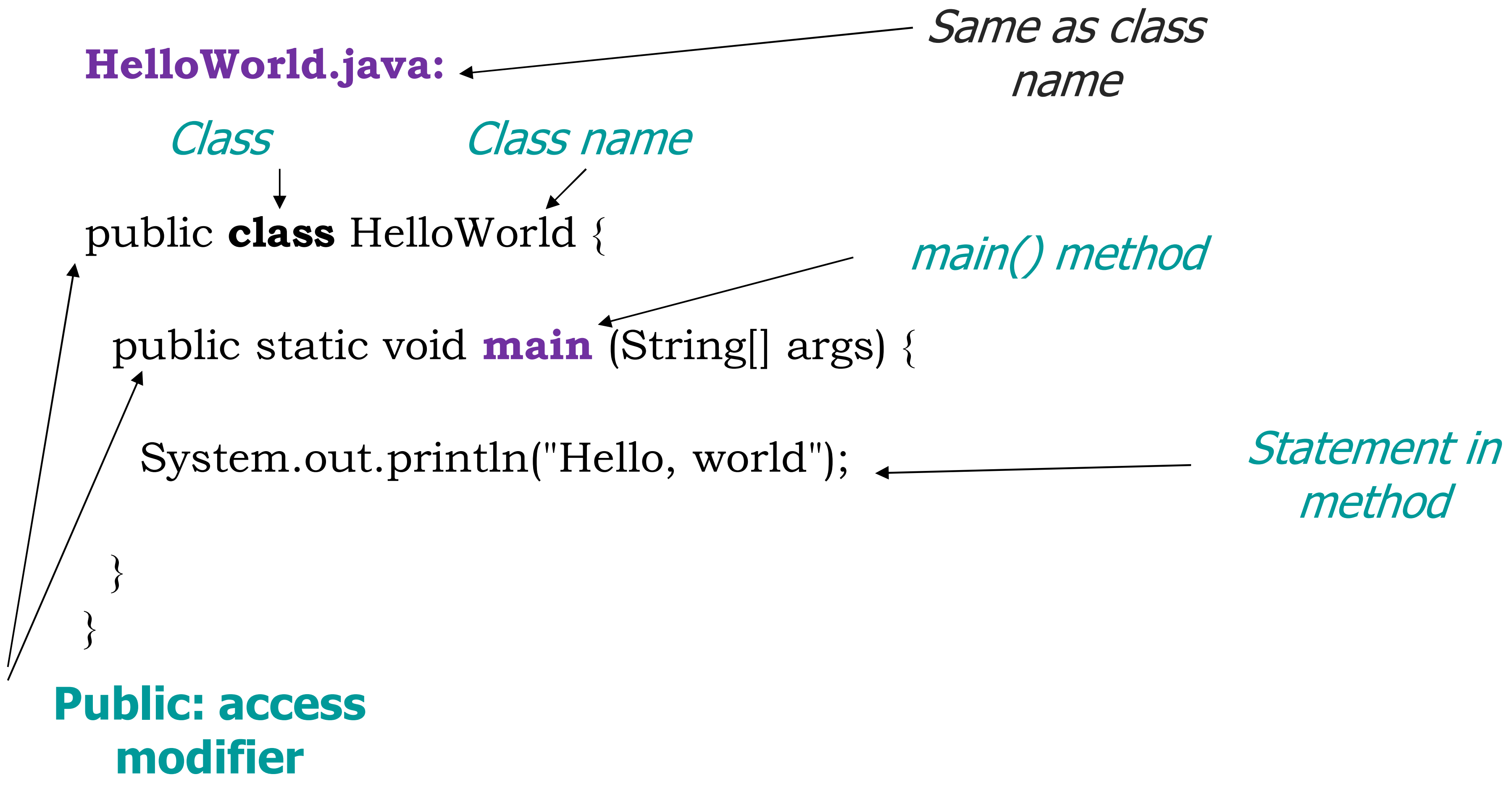
JVM

- JVM is platform dependent (hardware, OS)
- Ensure java program (bytecode) can execute on different platforms (i.e. platform independent)
- Guarantee security
- Usually implemented as a software
  - JRE (Java Runtime Environment)
- Java platform: JVM + APIs

Java applications

- Desktop applications – Java standard edition
- Distributed application, host application – Java Enterprise Edition
- Mobile applications
- Card applications

Example



HelloWorld.java:

Same as class name

Class

Class name

```
public class HelloWorld {

  public static void main (String[] args) {

    System.out.println("Hello, world");

  }
}
```

main() method

Statement in method

Public: access modifier

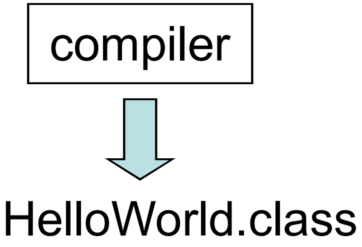
Compile & run

- Compile HelloWorld.java

  javac HelloWorld.java
- Run

  java HelloWorld

```
public class HelloWorld {
  public static void main (String[] args)
  {
    System.out.println("Hello, world");
  }
}
```


compiler

HelloWorld.class

```
%> javac HelloWorld.java
%> java HelloWorld
Hello, world
```

More than two classes

2 classes in different files

TestGreeting.java:

```
public class TestGreeting {
  public static void main(String[] args) {
    Greeting gr = new Greeting();
    gr.greet();
  }
}
```

Greeting.java

```
public class Greeting {
  public void greet() {
    System.out.print("Hi there!");
  }
}
```

Compile & run

- Compile

  javac TestGreeting.java

  `Greeting.java` automatically translated

- Run

  java TestGreeting

```
%> javac TestGreeting.java
%> java TestGreeting
Hi there!
```

JDK – Java Development Kit

- Java application development environment
- Main components
  - **javac** compiler, converts source code into Java bytecode
  - **java** interpreter and application loader
  - **javadoc** documentation generator, automatically generates documentation from source code comments
  - **jdb** debugger
  - …

main() Method

- In Java, everything has to be in a class
- When executing a program, we execute a class
  - Load class and execute **main()** method
  - Class must have **main()** method

Define a class

- **Syntax**

```
[public] class class_name {
...
}
```

- E.g.,

```
class MyDate {
        ....
}
```

Constructors

- Default constructor
- Parameterized constructor
- Copy constructor
- Accessing constructor
- Multiple constructors & self-reference

Constructors

- Constructors are special types of methods that are responsible for **creating** & **initializing** an object of that class
- Constructor is very much like creating a method, except that
  - Constructors don't have any return types
  - Constructors have the same **name** as **the class** itself
- They can take input parameters like a normal method
- **Multiple constructors** are allowed

Default constructor

- is one that does **not takes** any **input parameters**
- it's **optional,** which means **if you don't** create *a default constructor* Java will automatically assume there's one by default that doesn't really do anything
  - it is called **empty constructor**

- E.g., a class **Game** defined as follows…

```
class Game {

}

Game o = new Game();
```

Default constructor…

- However, if the class has **fields** that **need to be initialized** before the object can be used, then you should create one that does so

- E.g.,

```
class Game {
        int score;
        //default constructor
        Game(){  score=0; //initialize the score; or you can let it empty    }
}
```

Parameterized constructor

- A constructor can also take input parameters
- e.g., assume that some games starts with a positive score value and not just 0, that means we need another constructor that takes an integer parameter as an input, and uses it to initialize the score variable

```
class Game {
        int score;
        //default constructor
        Game(){
            score=0;//initialize the score
        }
        Game(int startingScore){
            score=startingScore;
        }
    }
```

Parameterized constructor…

- **However….**

```
class Game {
        int score;
        //default constructor
        Game(int startingScore){
            score=startingScore;
        }
    }
```

```
Game g1 = new Game ();//error
Game d2 = new Game(10);
```

Constructors

Accessing constructor

- Unlike normal methods, constructors **cannot** be called using the dot **'.'** modifier, instead *every time* you create **an object variable** of a class type *the appropriate* ***constructor is called***

- To create an object of a certain class, we use the **new keyword** followed by the constructor we want to use

- E.g.
  - Game *01* = **new** Game();
    - this will create an object called *01* using the default constructor
  - Game *02*= **new** Game(200)
    - this calls the 2nd constructor

Constructors

Accessing constructor…

- If you **don't** initialize an object using the **new** keyword, then its value will be set to something called **null**
  - Game *o* = **null;**
- **null object** means "**empty**" **object**
  - an object has **no** **fields** or **methods**
- In some case, you want to set an object to **null** to indicate that *such object is* ***invalid*** *or* ***yet to be set***

Why multiple constructors

- **WHY** still need to keep the **default constructor** now that we have another constructor that can create, say a game object with ***any starting score*** *value* (including **0**)?

- It’s considered a good practice to always include a default constructor that initializes **all the fields** with values that correspond to typical scenarios

- Then, you can add extra **parameterized constructors** that allow more customization when dealing with **less common cases**

Self reference

- Sometimes you need *to refer to an object* **within** *one of its methods* or *constructors,* to do so we use the keyword **this**
- The most common reason for using **this** keyword is because **a field** has the **same** name as **a parameter** in the method or constructor
- e.g., a Position class is defined as

```
class Position {
        int row=0;
        int column=0;
        Position(int r,int c){
            row=r; column=c;
        }
    }
```

Self reference…

- *A more readable way* would be use **the same name** for the constructor parameters which means we need to use the this keyword to separate the fields and the parameters

- e.g., a **Position class** is defined as

```
class Position {
        int row=0;
        int column=0;
        Position(int row, int column){
            this.row=row;  this.column=column;
        }
    }
```

Example

Contact manager

```
class Contact{
        String name;
        String email;
        String phoneNumber;
}
```

- **All fields, no methods,** since a contact object itself won’t be “doing” much attention
- Next, create **the class** that store an array of contacts and is in charge of adding or searching for contacts

```
class ContactsManager{
        Contact[] myFriends;
        int friendCount;
        …
}
```

Example

Contact manager…

```
class ContactsManager{
    Contact[] myFriends;
    int friendCount;
    ContactsManager(){
        FriendCount=0;
        myFriends = new Contact[100];
    }
    ….
}
```

- The friendCount starts from **0** and will increment every time we add a new contact later

Example of Contact manager...

Class methods

- The method addContact() will add a Contact object to **the Contact array myFriends**
  - Takes a Contact object as an input parameter
  - Use friendCount value to fill **that slot** in the array with the contact that was passed into the method

```
void addContact(Contact contact){
        myFriend[friendCount]=contact
        friendCount++;
}
```

Example of Contact manager…

Class methods

- Now, add another method searchContact () that will search through the array using a name String and return a Contact object once a match is found

```
Contact  searchContact(String searchName){
        for(int i=0;i<friendCount;i++)
                if(myFriend[i].name.equals(searchName))
                        return myFriend[i]
        return null;
}
```

run the program

```
class Main{
    public static void main(String [] args){
        ContactManager myContactManager= new ContactManager();

        Contact friendMinh=new Contact();
        friendMinh.name="Minh"; friendMinh.phoneNumber="01287761990"
        myContactManager.addContact(friendMinh);
        //...add some more contacts;
        Contact found=myContactManager.searchContact("Minh");
        System.out.println(found.phoneNumber)
    }
}
```

- If you go ahead and **run this program,** and **see what appear**

Copy constructor

- Besides two types of constructors introduced, a class object can be initialized with another previously created object of the same class

```
 public class Game {
     private int score;
     public Game() {score=0;}

     public Game(Game g) {
         score = g.score;
     }
 }
```

Access modifiers

- public vs. private fields
- public vs. private methods
- public vs. private classes

Access modifiers

- Think of it as if you're loading photos to the cloud
  - some of them you'd like to make **public** and share with others
  - while other photos are more of a personal nature and you'd like to keep them **private**
- In java, a field or method can be labeled as public or private

```
class Account{
        public string name;
        private String password;
        public boolean login(){
                return checkPasswrord(password)
        }
}
```

- **Public** field or method can be accessed by other classes

Access modifiers

Fields (public or private)

- Depending on the purpose of the field you'd label **it** as **public** or **private** simply add the modifier just before the field type when declaring it
- E,g.,

```
class Book{
        private String title
        private String author
        public Book(String title, String author){
                this.title = title; this.author=author;
        }
}
```

- All fields are **private** and initialized in **a constructor**
- This guarantee that **once a book object has been created**, the **title** and **author** will never change!

Access modifiers

Fields (public or private)…

- if we want to keep track of **whether a Book is being borrowed or not**, we can add a **public** boolean field to do so

```
 class Book{
        private String title
        private String author
        public boolean  isBorrowed;
        public Book(String title, String author){
               this.title = title; this.author=author;
        }
 }
```

- We can do book.isBorrowed = true anywhere in the project
- However, it’s still risky, we may end up mistakenly setting the boolean to true when we only mean to check if it is true or false

Fields (public or private)...

- A better design would be **to declare** that field **as private**
- & Create **public methods** that return the value of such hidden field and **public** methods to **set or change its value**

```
class Book{
        private String title
        private String author
        private boolean  isBorrowed;
        public Book(String title, String author){
                this.title = title; this.author=author;
        }
        public void setTitle(String title){ this.title = title;} //setter
        public String getTitle(){ return title;}                 //getter
        ....
}
```

Access modifiers

Fields (public or private)…

```
class Book{
        private String title
        private String author
        private boolean  isBorrowed;
        public Book(String title, String author){
                this.title = title; this.author=author;
        }
        ….
        public void borrowBook(){ isBorrowed=true;}
        ….
}
```

Access modifiers

Fields (public or private)…

```
class Book{
        ….
        public void returnBook(){ isBorrowed=false;
        }
        public boolean isBookBorrowed(){ return isBorrowed;
        }
}
```

Methods (public vs private)

- **Private** methods are usually known as **helper methods**
  - since they can only be **seen** and **called** by the **same class**
  - used to **organize your code** and keep it simple and more readable
- **Public** methods are the **actual actions** that the class can perform
  - and the **rest of the program** can **see and call**

Methods (public vs private)

```
class Person{
        private String userName;
        private String SSN;
        private string getID(){return SSN + “-” + userName;}
        public getUserName(){return userName;}
        public boolean isSamePerson(Person p){
                if(p.getID().equals(this.getId()) return true;
                else return false;
        }
}
```

- Method **getID()** was set to **private** so that **no other class** can know the social security number of any person
  - can use it internally only to compare this person with another person
- 2 public methods can be called by **any other class**!

public classes

- Classes can be labeled **public** or **private**
- if you don’t use any label, it will **default** to something called “package public”
  - that means, you’ve labeled them **public** but only to the classes that are in the **same** package/folder

Conclusion

- Always try to declare all fields as **private**
- Create a **constructor** that accepts those private fields as inputs
- Create **a public method** that **set** each private field, this way you will know when you are changing a field
  - these methods are called **setters**
- Create a public method that returns each private field, so you can read the value without mistakenly changing it
  - these methods are called **getters**
- Set all your classes to **public**