Object-Oriented Programming with Java
Introduction

Quan Thanh Tho, Ph.D.
CSE – HCMUT
qttho@hcmut.edu.vn

Contents

- Object-Orientation
- Objects and Classes
- Overloading
- Inheritance
- Polymorphism
- Generic Programming
- Exception Handling
- Introduction to Design Patterns

Outline

- Java
- First Program in Java
- OOP

What Is Java?

- History
- Characteristics of Java

History

- James Gosling
- Oak
- Java, May 20, 1995, Sun World
- HotJava
  - The first Java-enabled Web browser

Characteristics of Java

- Java is simple
- Java is object-oriented
- Java is distributed
- Java is interpreted
- Java is robust
- Java is secure
- Java is architecture-neutral
- Java is portable
- Java's performance
- Java is multithreaded
- Java is dynamic

JDK Versions

- JDK 1.02 (1995)
- JDK 1.1 (1996)
- Java 2 SDK v 1.2 (a.k.a JDK 1.2, 1998)
- Java 2 SDK v 1.3 (a.k.a JDK 1.2, 2000)

Java IDE Tools

- Inprise JBuilder
- Microsoft Visual J++
- Symantec Café
- Forte by Sun MicroSystems
- IBM Visual Age for Java

Getting Started with Java Programming

- A Simple Java Application
- Compiling Programs
- Executing Applications
- A Simple Java Applet
- Viewing Java Applets
- Applications vs. Applets

A Simple Application

Example 1.1

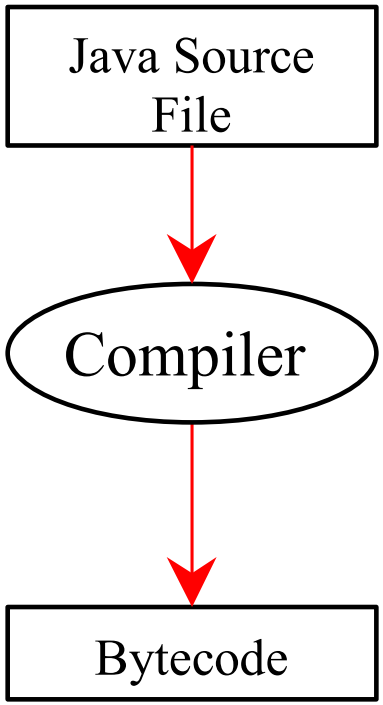
```
//This application program prints Welcome
//to Java!
public class Welcome
{
  public static void main(String[] args)
  {
    System.out.println("Welcome to Java!");
  }
}
```

Source

Run

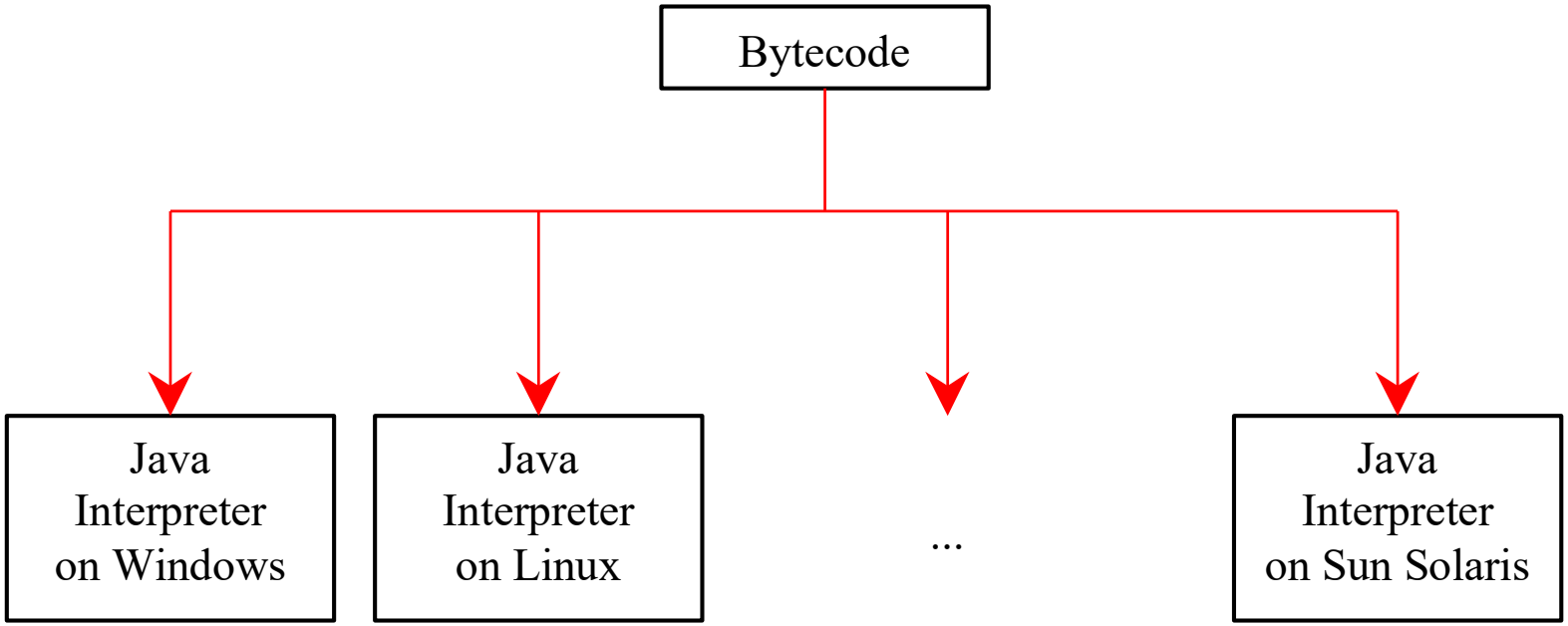
Compiling Programs

- On command line
  - `javac file.java`



Executing Applications

- On command line
  - `java classname`



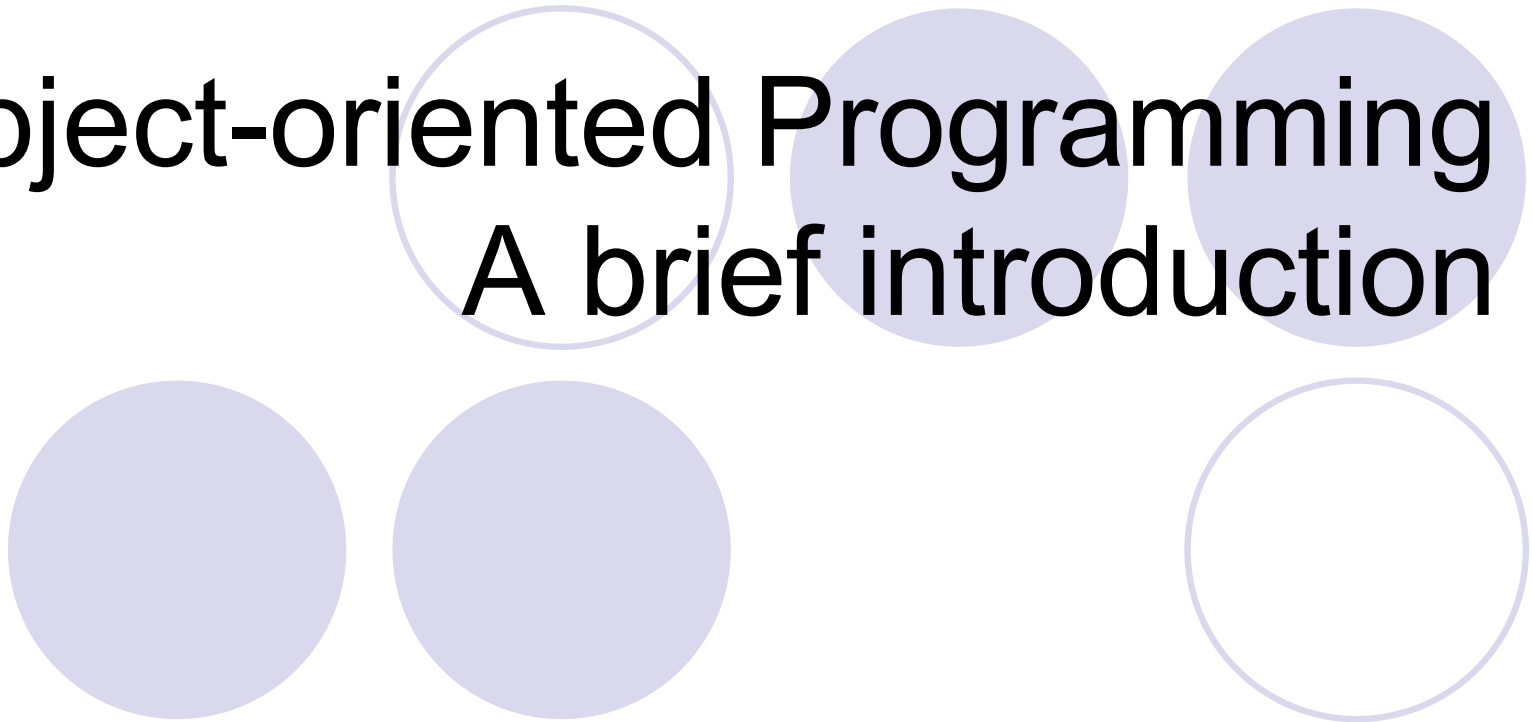
Example

```
javac Welcome.java

java Welcome

output:...
```

Object-oriented Programming
A brief introduction



Challenges of Software Development

- Complexity:
- Longevity and Evolution
- High User Expectations

1) Complexity

• The software systems are very large and complex.

•No individual can comprehend every detail of the system.

•The system must be broken down into manageable parts.

•Methodologies, techniques, and tools that work well for small systems developed by individuals are not effective for large systems developed by teams.

2) Longevity and Evolution

• Because of economic, political, and other constraints,
software systems are often in service for very long periods of time.

• During their lifetimes, software must constantly evolve to accommodate changes in users' needs and environments.

• Making changes to software systems is a difficult task.

2) Longevity and Evolution

- Furthermore, maintenance not only is costly and time-consuming, but also usually degrades the quality of the system being maintained.
- The maintenance cost of the software system over its lifetime is far greater than its initial development cost.

3) High User Expectations

- *In the past*: Computers were mainly used in universities, research institutions, and large corporations.

The majority of software systems users were engineers and scientists, who had the technical skills to handle the glitches they might encounter while using the system.

-

3) High User Expectations

• *Today*: Computers are used in homes, schools, and business of all sizes, and are used for pleasure as well as for work.

Majority of software users are nontechnical, ordinary people.

Computer software products are considered as consumer products. The products are expected to perform as household appliance. Software systems are expected to be “bug-free”,

but such perfection is next to impossible.

Challenges of Software Development

The challenges faced by software development are:

- to find effective solution to control complexities of software systems;
- to manage the longevity and evolution of software systems;
- to deliver software systems with higher reliability and usability.

Two Paradigms of Programming

- *Program = Code + Data*

  Program can be organized around Data or around Code

- Two paradigms of programming:

  1) Process-oriented model: around Code (acting on data)

  Program is defined by sequence of codes

  Data structures + Algorithms = Program

  Problem complexity: <25,000-100.000 lines of code

  2) Object-oriented model: around Data (Objects)

  Objects + Messages = Program

  Data-controlling access to code

OOP five rules of Alan Kay

- Everything is an *object*.
- A program is a bunch of *objects* telling each other what to do by sending *messages*.
- Each *object* has its own memory made up of other *objects*.
- Every *object* has a type (*class*).
- All *objects* of a particular type (*class*) can receive the same *messages*.

Class

- A *class* defines the abstract characteristics of a thing (object), including the thing's characteristics (its attributes) and the thing's behaviors (the things it can do or methods).
- For example, the class Dog would consist of traits shared by all dogs, such as breed and fur color (characteristics), and the ability to bark (behavior).
- Classes provide modularity and structure in an object-oriented program.

Objects

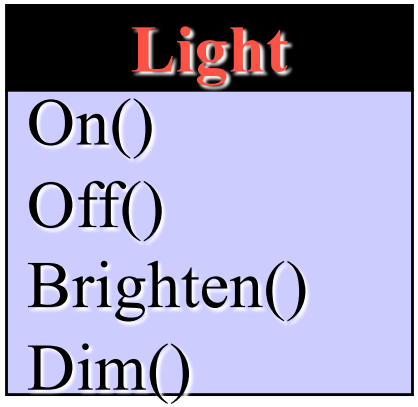
- OOP model is to work in the problem space
- Objects can be described in terms of:
  - Their attributes;
  - Their behaviors
- Object's attributes and behaviors are encapsulated together a data type (class).
- Objects are independent entities.
- Objects respond to messages.

Message passing

- *Message* passing is the process by which an object sends data to another object or asks the other object to invoke a method.
- Also known to some programming languages as *interfacing*.

Objects Interfaces

- An object has to have an interface.
- The interface will
  - Provide a way to let other objects to communicate with it.
  - Hide the details of the object implementation.
- An example: Light bulb and Car.



Class Dog and Object Lassie

- Object is a particular instance of a *class*.
- The class of Dog defines all possible dogs by listing the characteristics (data, states) and behaviors (methods) they can have.
- The object Lassie is one particular dog, with particular versions of the characteristics. A Dog has fur, Lassie has brown-and-white fur.
- In programmer jargon, the *object* Lassie is an instance of the Dog *class*.

Methods

- An object's abilities. Lassie, being a Dog, has the ability to bark. So bark() is one of Lassie's methods. She may have other methods as well, for example sit() or eat().
- Within the program, using a method should only affect one particular object; all Dogs can bark, but you need one particular dog, Lassie, to do the barking.

Three Principles of OOP (PIE)

- Encapsulation
- Inheritance
- Polymorphism

1) Encapsulation principle

- Encapsulation is a process of binding or wrapping the *data* and the *codes* that operates on the data into a single entity.
- Binding *data* and *functionality* (*methods*) together produces self contained units which are more maintainable, reusable and deployable.
- Implementation hiding: encapsulation is as a protective wrapper that prevents *data* from being arbitrarily accessed by other code defined outside the wrapper.
- The collective term for *data* and *operations (methods)* bundled together with access restrictions is a *class*.

1) Encapsulation principle

- For a programming unit to be truly effective, the barrier between *interface* and *implementation* must be absolute.
- Breaks up the playing field into:
  - Class creators (those who create new data types)
  - Client programmers (those who use these types for their own tasks. They only deal with the interfaces.)
  - The goal of the class creator is to build a class that exposes only what's necessary to the client programmer and keeps everything else hidden.

1) Encapsulation principle

- Two reasons for controlling the access:

  * To keep client programmers' hands off portions they shouldn't touch.

  * To allow the library designer to change the internal workings of the class without worrying about how it will affect the client programmer.

2) Inheritance principle

- Inheritance involves building upon an existing class so that additional or more-specialised functionality is added.
- *Subclasses* are more specialized versions of a class, which inherit attributes and behaviors from their *superclasses*,
  and can introduce their own.
- Inheritance should be used to capture hierarchical relationships integral to the problem being modelled.

2) Inheritance principle

- Object-oriented programming languages permit you to base a new class definition on a class already defined.
- The base class is called a superclass; the new class is its subclass.
- The subclass definition specifies only how it differs from the superclass; everything else is taken to be the same.
- Nothing is copied from superclass to subclass. Instead, the two classes are connected so that the subclass inherits all the methods and instance variables of its superclass.

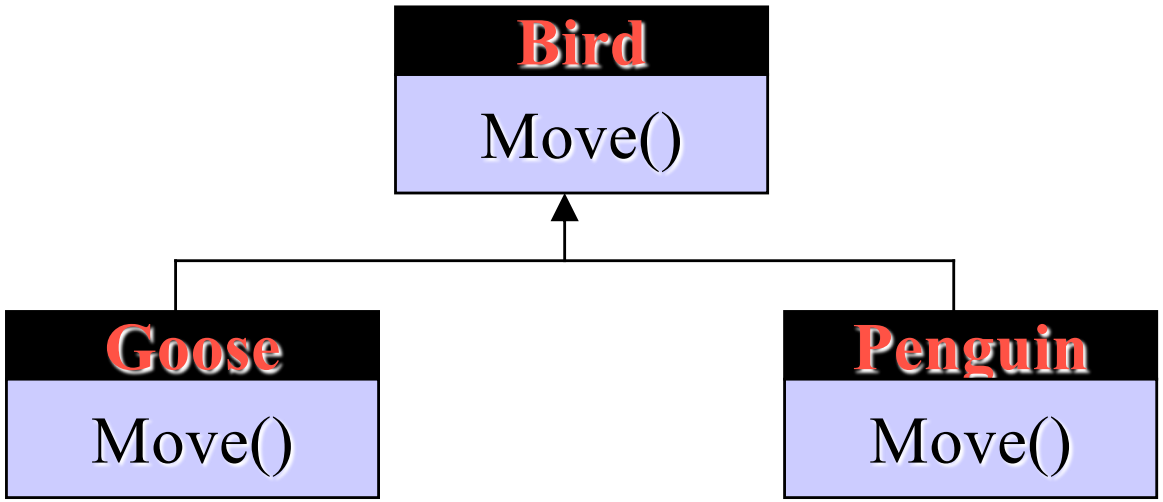
3) Inheritance principle

- For example, the class Dog might have sub-classes called Collie, Chihuahua, and GoldenRetriever. In this case, Lassie would be an instance of the Collie subclass.
- Suppose the Dog class defines a method called bark() and a property called furColor. Each of its sub-classes (Collie, Chihuahua, and GoldenRetriever) will inherit these members, the programmer only needs to write the code for them once.
- In fact, inheritance is an "*is-a*" relationship: Lassie is a Collie.

  A Collie is a Dog. Thus, Lassie inherits the members of both Collies and Dogs.

3) Polymorphism principle

- The word originates from the Greek language, and means ‘many shapes’.
- Within object oriented languages it is used to denote *one name* referring to *several different methods*
- Polymorphism allows you to make the derived types behave differently from their base type.



3) Polymorphism principle

- This ability of different objects to respond, each in its own way, to identical messages is called polymorphism.
- The main benefit of polymorphism is that it simplifies the programming interface. It permits conventions to be established that can be reused in class after class.
- Instead of inventing a new name for each new function you add to a program, the same names can be reused. The programming interface can be described as a set of abstract behaviors, quite apart from the classes that implement them, receive messages from them, and perhaps make the different objects composing the class interact with each other.