

OBJECT-ORIENTED LANGUAGE AND THEORY

0. INTRODUCTION TO COURSE

Nguyen Thi Thu Trang
trangntt@soict.hust.edu.vn



2

Course objectives

- Common knowledge of object-oriented programming languages using a popular programming language Java.
- Basic and elementary concepts and notations of object-oriented theory using Unified Modeling Language (UML).

Programming language/tools

- Modeling language: UML
- Software design tool: Astah
 - Free for students
- Programming language: Java
- IDE: Eclipse
- Version control: Bitbucket



Assessment

- Mid-term score: 40%
 - Hands-on labs and Mini-Project (60:40)
 - Submission Channel: <https://bitbucket.org>
 - Add to your project member: `trangntt-for-student` (trangntt.for.student@gmail.com)
- Final score: 60%
 - Final exam

Reference books

- ***Object-Oriented Programming and Java.*** Danny Poo, Derek Kiong and Swarnalatha Ashok. Springer. 2008.
- ***Effective Java.*** Joshua Bloch. Addison-Wesley, 2008
- ***UML 2 Toolkit.*** Hans-Erik Eriksson and Magnus Penker. Wiley Publishing Inc. URL:
http://www.ges.dc.ufscar.br/posgraduacao/UML_2_Toolkit.pdf.

Course Materials

- Lecture notes for students (pdf): Slides in 4-page handouts
- Assignments, Mini-Project descriptions
- Interaction channels:
 - BKE eLearning system: <https://lms.hust.edu.vn>
 - Microsoft Teams: OOLT.ICT.20192 / OOLT.VN.20192
 - Facebook group:
 - <https://www.facebook.com/groups/oolt.ict.20192/>
 - <https://www.facebook.com/groups/oolt.vn.20192/>

Naming convention for the repository

- Weekly assignment (individual):
 - OOLT.ICT.20192.StudentID.StudentName or
 - OOLT.VN.20192.StudentID.StudentName
- Mini-Project
 - OOLT.ICT.20192-GroupNo
 - OOLT.VN.20192-GroupNo

→ Monitor?

Introduce yourselves

- Full name
- Experience in Computer Science
 - Operating System
 - Programming Languages
 - (Mini-)Projects
 - ...
- Strength / Weakness
- A course you like best / hate
- Desire to study in this course



About Me

OBJECT-ORIENTED LANGUAGE AND THEORY

1. INTRODUCTION TO OBJECT TECHNOLOGY

Nguyen Thi Thu Trang

trangntt@soict.hust.edu.vn

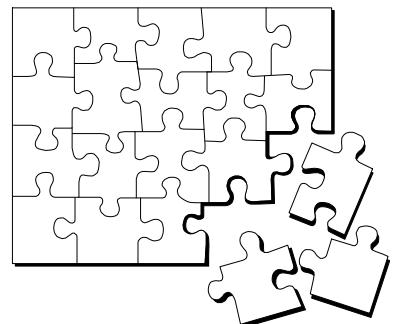


Outline

- ➡ 1. Object-Oriented Technology
- 2. Object and Class
- 3. Java programming language
- 4. Examples and Exercises

1.1 Object Technology

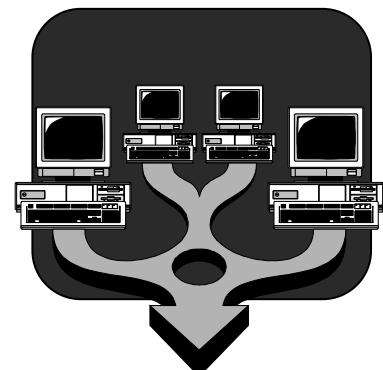
- Object technology is a set of rules (abstraction, encapsulation, polymorphism), instructions to build a software, together with languages, databases and other tools to support these rules.



(Object Technology - A Manager's Guide, Taylor, 1997)

1.2 Where is the object technology used?

- Client/Server Systems and Web development
 - Object technology allows companies to encapsulate information in objects and to distribute its computation/processing via Internet or via a network.



1.2. Where is the object technology used? (2)

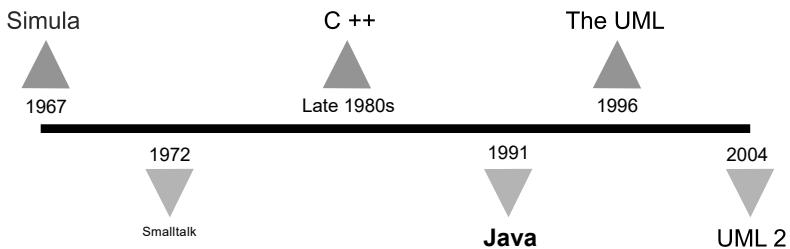
- Mobile development (Android)
- Embedded system
- Real-time systems
 - Object technology allows real-time systems to be developed with higher quality and in a more flexible way
 - Satellite systems
 - Defense systems and space airline ...



The power of the object technology

- Allow re-using source code and architectures
- Reflecting more closely the real world
- More stable, a system change is done in a small part of the system
- More adaptable with changes

Milestones of the object technology



1.3 Evolution of programming languages

- Assembly language
- Structure/Procedure programming languages
 - Pascal, C
- Object programming languages
 - C++, Java, C#.NET, Python...

a. Assembly language

```

;CLEAR SCREEN USING BIOS
CLR: MOV AX,0600H ;SCROLL SCREEN
      MOV BH,30 ;COLOUR
      MOV CX,0000 ;FROM
      MOV DX,184FH ;TO 24,79
      INT 10H ;CALL BIOS;

;INPUTTING OF A STRING
KEY: MOV AH,0AH ;INPUT REQUEST
      LEA DX,BUFFER ;POINT TO BUFFER WHERE STRING STORED
      INT 21H ;CALL DOS
      RET ;RETURN FROM SUBROUTINE TO MAIN PROGRAM;

; DISPLAY STRING TO SCREEN
SCR: MOV AH,09 ;DISPLAY REQUEST
      LEA DX,STRING ;POINT TO STRING
      INT 21H ;CALL DOS
      RET ;RETURN FROM THIS SUBROUTINE;

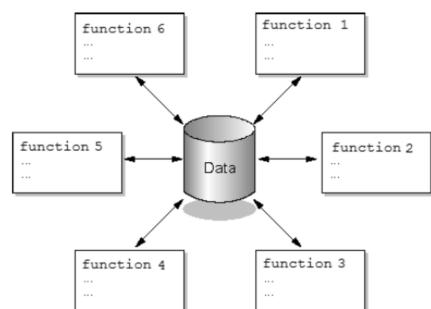
```

Assembly code

- Is a sequence programming language, is very close to machine codes of CPU.
- Hard to remember, to write, especially for complex systems.
- Hard to fix, to maintain.

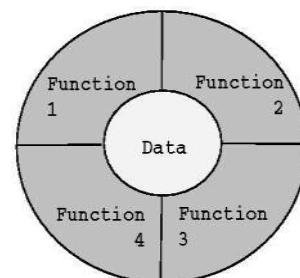
b. Structure/Procedure programming languages

- Build a program based on functions/procedures/sub-programs
- Data and data processing unit (functions) are separate
- Functions are not forced to follow a common rule for accessing data



c. Object programming languages

- Characterizing elements of a problem in form of “đối tượng” (object).
- Object-oriented is a technique to model a system by objects.



Evolution of programming languages

- ***Is the history and evolution of abstraction***
 - Assembly : Abstraction of data type/basic command
 - Structure languages: control abstraction + functional abstraction
 - OO languages: Data abstraction

Reading exercises

- Read and summarize some differences between struture programming and OOP

<http://www.desy.de/gna/html/cc/Tutorial/node3.htm>

What about other programming paradigms?

- Aspect-Oriented Programming
- Functional Programming

Outline

1. Object-Oriented Technology
2. Object and Class
3. Java programming languages
4. Examples and Exercises

16

Alan Kay' concepts

1. All are objects.
2. A software program can be considered as a set of objects interacting with each other
3. An object in a program has its own data and its own memory.
4. An object has all characteristics of its class.
5. All objects of a class have the same behavior



Alan Kay

2.1 Object

- **Object** is the key to understand the object technology
- In a OO system, all are objects



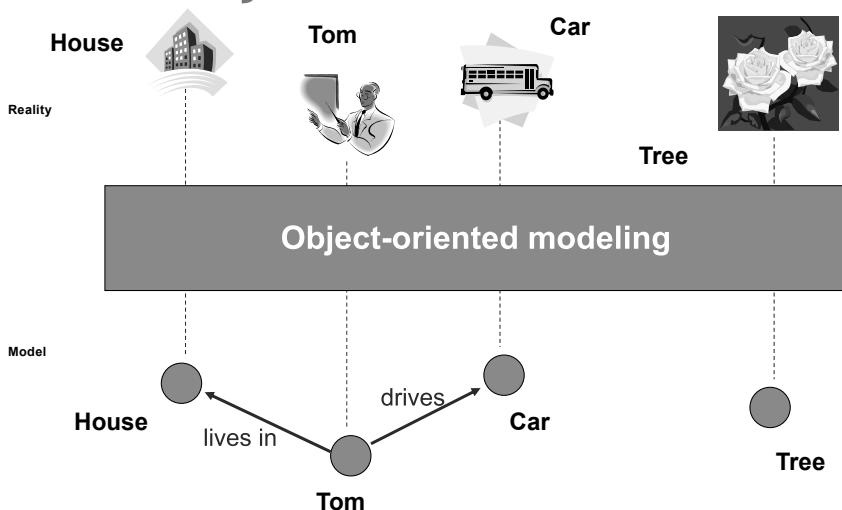
Writing a OO program means to build a model of some parts in the real world

2.1 What is object?

- Objects in the real world
 - For example, a car
- Related to a car:
 - Car information such as: color, speed,...
 - Car activities: moving forward, reversing, stopping,...

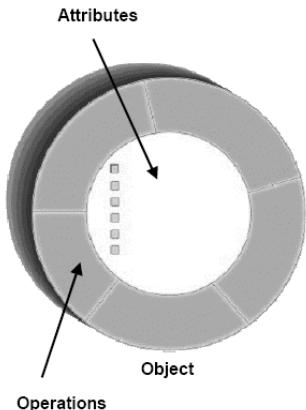


What is object?



What is object?

- Is an entity encapsulated in form of state and behavior.
 - **State** is represented by attributes and relationships.
 - **Behaviour** is represented by operations and methods.



An object has a state

- The state of an object is one of the possible conditions that the object exists.
- The state of an object can change over time



Name: J Clark
Employee ID: 567138
Date Hired: July 25, 1991
Status: Tenured
Discipline: Finance
Maximum Course Load: 3 classes



Professor Clark

State



Dave
Age: 32
Height: 6' 2"



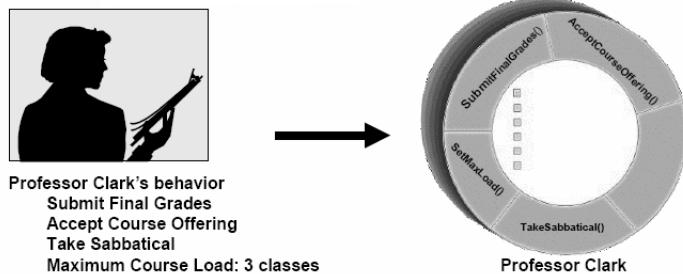
Brett
Age: 35
Height: 5' 10"



Gary
Age: 61
Height: 5' 8"

An object has its behavior

- Behavior determines how an object acts and reacts to requests from other objects.
- Object behavior is represented by the operations that the object can perform.



Behavior



An object has an unique identity

- Each object has its own unique identity, although two objects may share the same state (attributes and relationships)

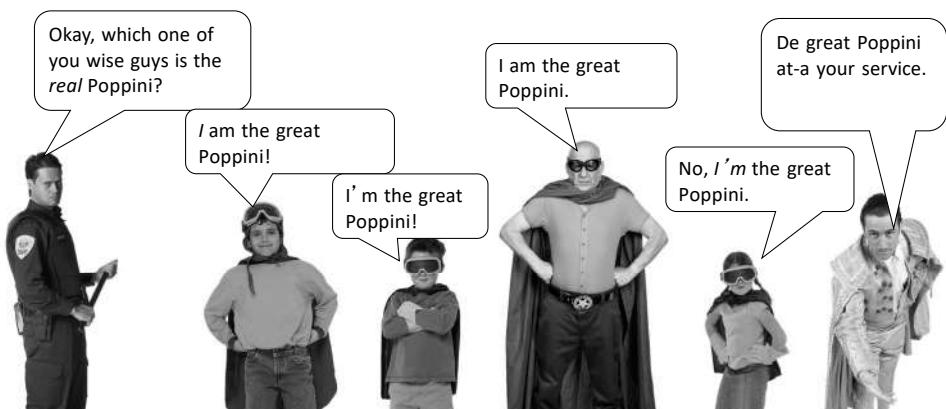


Professor "J Clark"
teaches Biology

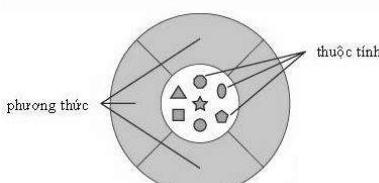


Professor "J Clark"
teaches Biology

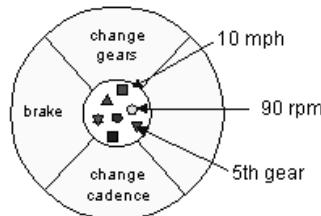
ID



Object



Software object



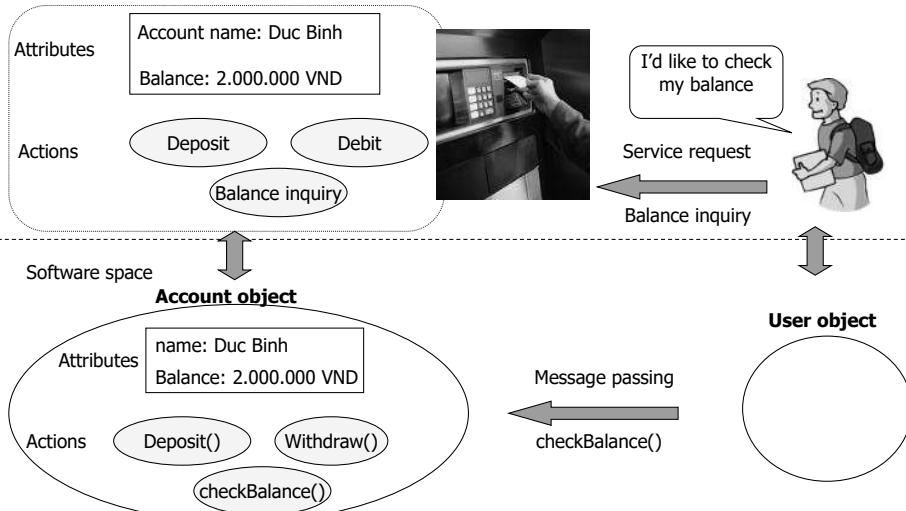
Software object Xe Đạp

Object is an entity encapsulating **attributes** và các **methods** involving.

Attributes are defined by a specific value called **representation attributes**. A specific object is called a **representation**.

Software objects and a real-life problem

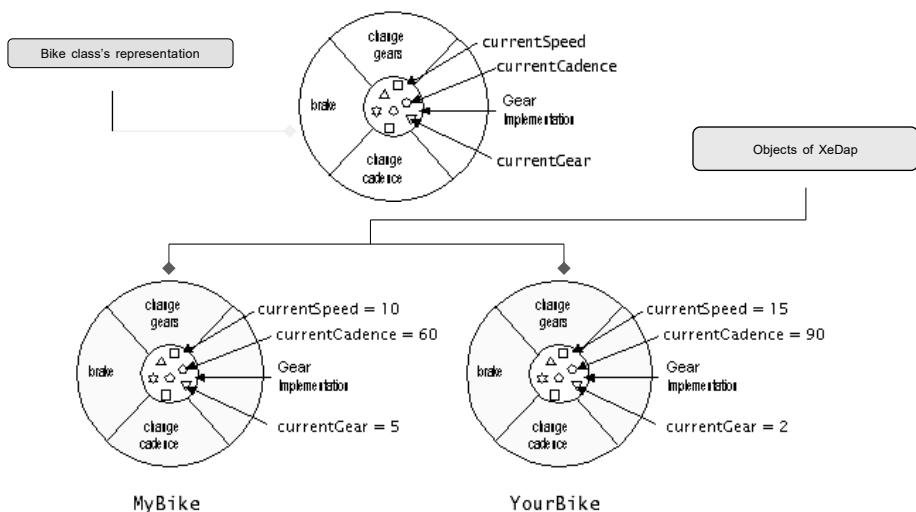
Problem of bank account management – ATM ther – electronic payment



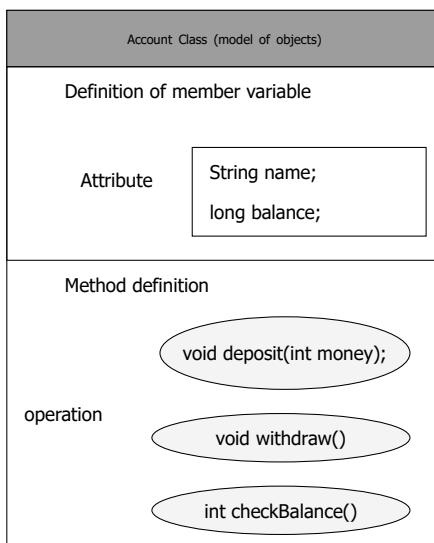
2.2 Class

- A class is a blueprint or prototype for all the objects of a same type
 - Example: class Bike is a common blueprint for many bike objects that are created
- A class defines common attributes and methods for all the objects of some type
- An object is a detailed representation of a class.
 - Example: a bike object is a representation of the class Bicycle
- Each object can have different attribute's representation
 - Example: a bike can be at the 5th gear while another bike can be at the 3rd gear.

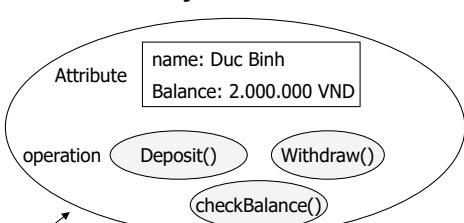
Example: Bike class



Class and Object

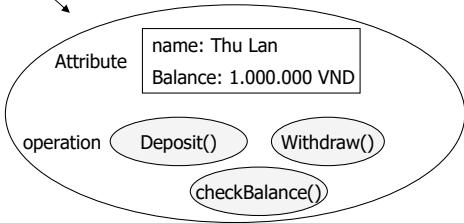


Account object of Mr Duc Binh



INSTANTIATE

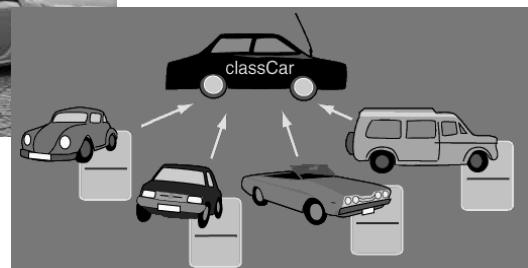
Account object of Mrs Thu Lan



Class and Object



Blueprint/prototype

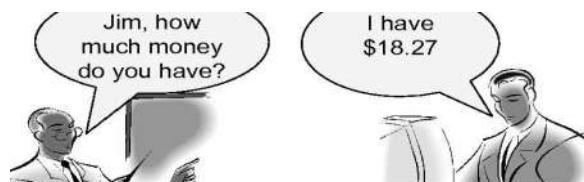


Quick question

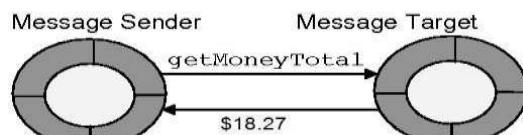
- Given the Amazon online shopping system. Provide some examples about class and object in this system?
- The same question for HUST Student Information System?

2.3 Interactions between objects

- Communication between objects in the real world



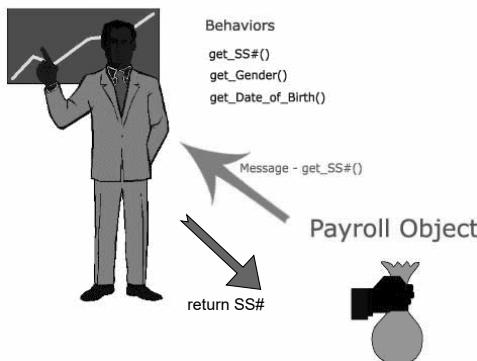
- Objects and their interactions in programming
 - Objects communicate to each other by message passing



Message passing

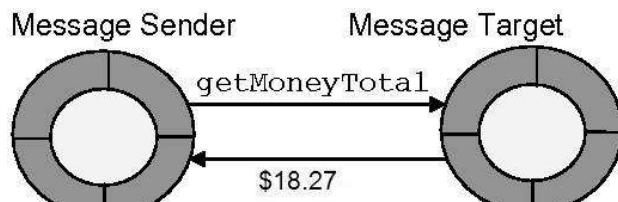
- A program (built via OOP) is a set of objects exchanging messages between them

Employee Object



2.4 Structure-Oriented vs. OO?

- Structure-Oriented:
 - data structures + algorithms = Program
- Object-Oriented:
 - objects + messages = Program



Procedural-oriented vs. Object-oriented

- Procedural Programming:
 - Main components are procedures, functions
 - Data is independent with procedures

- Object-oriented programming
 - Main components are objects
 - Data is associated to function (method) in an object
 - Each data structure has methods executing on it

Examples of class and object in some OOP languages

Class declaration: each class is, by default, an extension of **Object** (can be omitted)

Class constructor: initialises the various fields

Class method: retrieves and/or modifies the state of the class

```
public class Time extends Object {
    private int hour;
    private int minute;
    private int second;

    public Time () {
        setTime(0, 0, 0);
    }

    public void setTime (int h, int m, int s) {
        hour = ( ( h >= 0 && h < 24 ) ? h : 0 );
        minute = ( ( m >= 0 && m < 60 ) ? m : 0 );
        second = ( ( s >= 0 && s < 60 ) ? s : 0 );
    }
}
```

Class fields: **private** means they can not be accessed from outside the class

Java: Program and object

```
public class Test {  
  
    public static void main (String args[]) {  
        Time time = new Time();  
  
        time.hour = 7;  
        time.minute = 15;  
        time.second = 30;  
    }  
}  
  
Test.java:6: hour has private access in Time  
        time.hour = 7;  
               ^  
Time.java:7: minute has private access in Time  
        time.minute = 15;  
               ^  
Time.java:8: second has private access in Time  
        time.second = 30;  
               ^  
3 errors
```

Outline

1. Object-Oriented Technology
2. Object and Class
3. Java programming languages
4. Examples and Exercises

3.1 What is Java?

- Java is a object-oriented programming language developped by Sun Microsystems, and now bought by Oracle
- Java is a popular programming language
 - Initially used for building control processor applications inside the electronics consumer devices such as cell phones, microwaves ...
 - Initially used in 1995



Green Team and James Gosling
(the leader)



J2SE (Java 2 Platform Standard Edition)

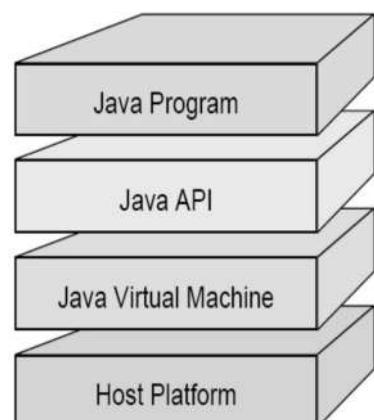
- <http://java.sun.com/j2se>
- Java 2 Runtime Environment, Standard Edition (J2RE):
 - Executable Environment or JRE provides Java APIs, Java Virtual Machine (JVM) and other necessary components to run applets and applications written in Java.
- Java 2 Software Development Kit, Standard Edition (J2SDK)
 - Super set of JRE, and contains everything in the JRE, additional tools such as compilers and the debugger need to develop applets and applications.

J2EE (Java 2 Platform Enterprise Edition)

- <http://java.sun.com/j2ee>
- Service-Oriented Architecture (SOA) và Web services
- Web Applications
 - Servlet/JSP
 - JSF...
- Enterprise Applications
 - EJB
 - JavaMail...
 - ...

3.2 Java platform

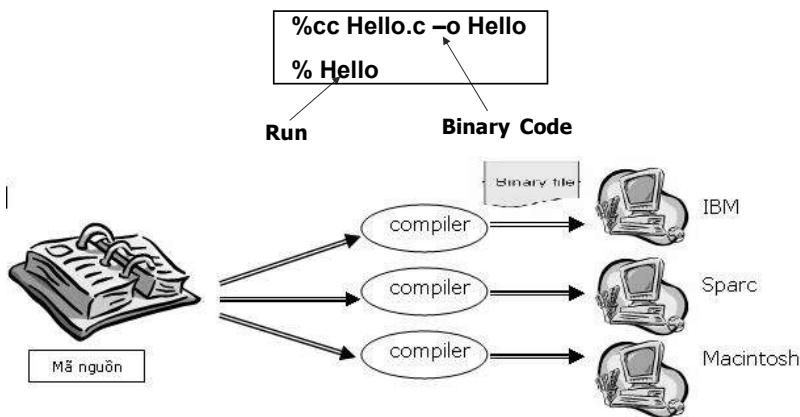
- Platform is environment for development of deployment.
- Java platform can be run on all OSs
 - Other platforms depend on hardware
 - Java platform provides:
 - Java Virtual Machine (JVM).
 - Application Programming Interface (API).



3.3. Compiling model of Java

- a. Classical compiling model:

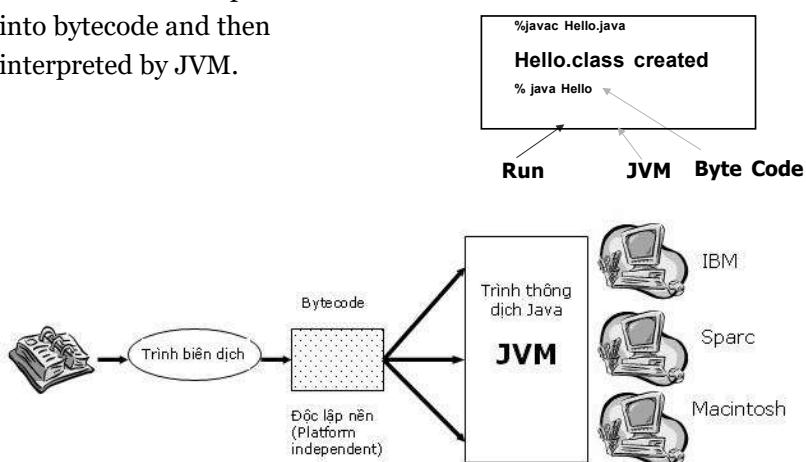
- Source code is compiled into binary code.



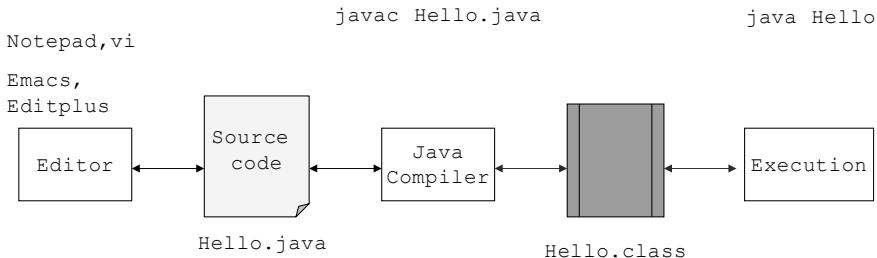
3.3. Compiling model of Java (2)

- b. Compiling model of Java:

- Source code is compiled into bytecode and then interpreted by JVM.



Making procedure of Java Application



3.3. Compiling Model of Java (3)

- Java Virtual Machine:
 - JVM is the heart of Java language
 - Bring the feature “Write once, run everywhere”
 - Provides environment to execute instructions:
 - Load file .class
 - Manage memory
 - Garbage collections
 - The Interpreter “**Just In Time - JIT**”
 - Transform bytecode to machine code for each type of CPU.

3.4. Features of Java

- Java is designed to be:
 - A powerful programming language, full of OO features and completely OO.
 - Easy to learn, syntax is similar to C++
 - Platform independance
 - Support the development of applications in network environment
 - Ideal for Web application

3.4. Features of Java (2)

- Powerful
 - Class library: Hundreds of classes already written with utility functions.
 - Java uses pointer model without accessing directly to the memory; memory can not be over-written.
- Object-Oriented
 - Java supports software development by using OO
 - Software built in Java includes classes and objects

3.4. Features of Java (3)

- Simple
 - Keywords
 - Java has 50 keywords
 - Compared to Cobol VB that have hundreds of keywords
- Network capable
 - Java supports the development of distributed applications
 - Some applications of Java are designed in order to be accessed via Web browser.

3.4. Features of Java (3)

- Java has 50 key words
 - assert (New in 1.5) enum (New in 1.5)

abstract	boolean	break	byte
case	catch	char	class
const	continue	default	do
double	else	extends	final
finally	float	For	goto
If	implements	import	instanceof
int	interface	long	native
new	package	private	protected
public	return	short	static
strictfp	super	switch	synchronized
this	throw	throws	transient
try	void	volatile	while

3.4. Features of Java (5)

- Multi-threaded
 - Allows a program to run more than one task at the same time.
- Portable
 - Programs can be written once and run on different platforms
 - Based on compiler/interpreter model
(WORE – Write Once, Run Everywhere)

3.4. Features of Java (6)

- Development Environment
 - Java Development Kit
 - Free on Sun Website: java.sun.com
 - Including: Compiler, JVM and existing classes
 - Integrated Development Environments (IDEs): Providing:
 - Complex Text Editors
 - Debugging Tools
 - Graphics Development Tools

3.5. Applications in Java

- Application
 - Do not need to run on browsers
 - Can call functions through commands or option menu (GUI)
 - main() method is the starting point of the program execution
- Applet
 - GUI application running on browser in the client side.
 - Can be viewed by appletviewer or embedded in Web browser with JVM installed.

3.5. Applications in Java (2)

- Web application
 - Create dynamic content on Server instead of on browsers.
 - Used in Server application
 - Servlet: manage requests from browsers and send the responses back
 - JavaServer Page (JSP): HTML pages with embedded Java code.

Outline

1. Object-Oriented Technology
2. Object and Class
3. Java programming languages
4. Examples and Exercises

Example 1 - HelloWorld

```
// HelloWorld.java  
// Chuong trinh hien thi dong chu "Hello World"  
public class HelloWorld {  
/* Phuong thuc main se duoc goi dau tien  
trong bat cu ung dung Java nao*/  
public static void main(String args[]){  
    System.out.println( "Hello World!" );  
} // ket thuc phuong thuc main  
} // ket thuc lop HelloWorld
```



Example 1 (Cont.)

- Comment
 - In one line: Starts with //
 - In multiple lines: /* ... */
- Java distinguish between lowercase and uppercase
- Keywords in Java:
 - class: Class definition
 - public: Access permission
- Class name containing main function must have the same name with the file .java.

Installing and Running Java application

- Step 1: Install jdk, install environment variables (if using cmd)
- Step 2: Install Eclipse or Netbean IDE
- Step 3: Coding
- Step 4: Compile
 - cmd: javac HelloWorld.java
 - Eclipse/Netbean: Build automatically (Look at Console to see syntax errors if any)/F11 (Project) or F9 (File)
- Step 5: Run program
 - cmd: java HelloWorld
 - Eclipse/Netbean: Run as Java application (Alt+Shift+X+J)/F6 (Project) or Shift-F6 (File)

Environment Variables

- PATH = %PATH%;C:\Program Files\Java\jdkx.x\bin
- JAVA_HOME=C:\Program Files\Java\jdkx.x
- CLASSPATH = C:\Program
Files\Java\jdkx.x\lib;.;C:\Program
Files\Java\jdkx.x\include

Example 2 - GUI

```
import javax.swing.JOptionPane;
public class FirstDialog{
    public static void main(String[] args) {
        JOptionPane.showMessageDialog(null,
            "Xin chao ban!");
        System.exit(0);
    }
}
```



Example 3 - Data Input/Output

```
import javax.swing.JOptionPane;
public class HelloNameDialog{
    public static void main(String[] args){
        String result;
        result = JOptionPane.showInputDialog("Hay nhap ten ban:");
        JOptionPane.showMessageDialog(null,
            "Xin chao " + result + "!");
        System.exit(0);
    }
}
```



Example of Class and Object in Java

Class declaration: each class is, by default, an extension of **Object** (can be omitted)

Class constructor: initialises the various fields

Class method: retrieves and/or modifies the state of the class

```
public class Time extends Object {
```

```
private int hour;
private int minute;
private int second;
```

```
public Time () {
    setTime(0, 0, 0);
}
```

```
public void setTime (int h, int m, int s) {
    hour = ( ( h >= 0 && h < 24 ) ? h : 0 );
    minute = ( ( m >= 0 && m < 60 ) ? m : 0 );
    second = ( ( s >= 0 && s < 60 ) ? s : 0 );
}
```

Class fields: **private** means they can not be accessed from outside the class

Java: Program and Objects

```
public class Test {  
  
    public static void main (String args[]) {  
        Time time = new Time();  
  
        time.hour = 7;  
        time.minute = 15;  
        time.second = 30;  
    }  
}  
  
Test.java:6: hour has private access in Time  
    time.hour = 7;  
           ^  
Time.java:7: minute has private access in Time  
    time.minute = 15;  
           ^  
Time.java:8: second has private access in Time  
    time.second = 30;  
           ^  
3 errors
```

1

OBJECT-ORIENTED LANGUAGE AND THEORY

1-2. VERSION CONTROL

Nguyen Thi Thu Trang
trangntt@soict.hust.edu.vn



Outline

- 1. Introduction**
2. Models
3. Vocabulary
4. Tools

Why version control? (1/2)

- Scenario 1:
 - Your program is working
 - You change “just one thing”
 - Your program breaks
 - You change it back
 - Your program is still broken--*why?*
- Has this ever happened to you?

Why version control? (2/2)

- Your program worked well enough yesterday
- You made a lot of improvements last night...
 - ...but you haven't gotten them to work yet
- You need to turn in your program *now*
- Has this ever happened to you?

Version control for teams (1/2)

- Scenario:
 - You change one part of a program--it works
 - Your co-worker changes another part--it works
 - You put them together--it doesn't work
 - Some change in one part must have broken something in the other part
 - What were all the changes?

Version control for teams (2/2)

- Scenario:
 - You make a number of improvements to a class
 - Your co-worker makes a number of different improvements to the same class
- How can you merge these changes?

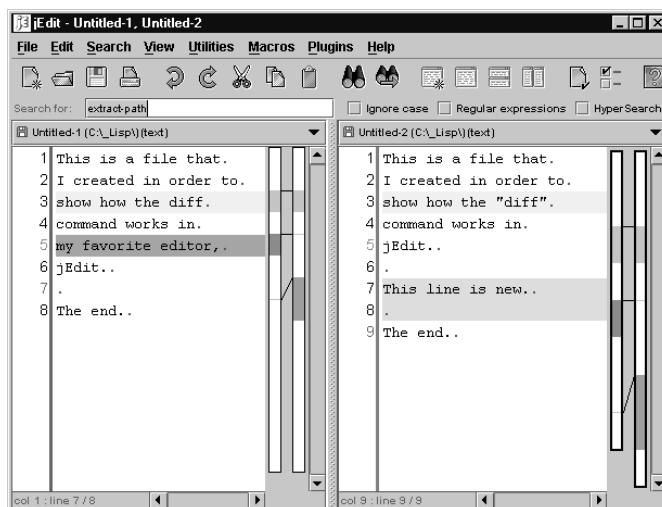
Tools: diff

- There are a number of tools that help you spot changes (differences) between two files
- Tools include `diff`, `rcsdiff`, `jDiff`, etc.
- Of course, they won't help unless you kept a copy of the older version
- Differencing tools are useful for finding a *small* number of differences in a *few* files

Tools: jDiff

- jDiff is a plugin for the jEdit editor
- Advantages:
 - Everything is color coded
 - Uses synchronized scrolling
 - It's inside an editor--you can make changes directly
- Disadvantages:
 - Not stand-alone, but must be used within jDiff
 - Just a diff tool, not a complete solution

Tools: jDiff



Version control systems

- A version control system (often called a source code control system) does these things:
 - Keeps multiple (older and newer) versions of everything (not just source code)
 - Requests comments regarding every change
 - Allows “check in” and “check out” of files so you know which files someone else is working on
 - Displays differences between versions

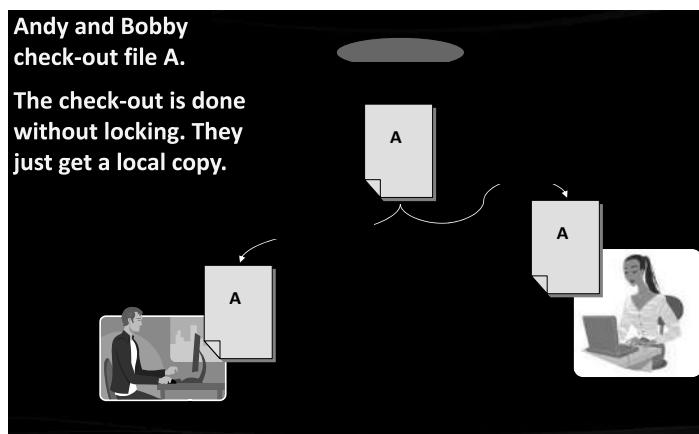
Outline

1. Introduction
- 2. Versioning Models**
3. Vocabulary
4. Tools

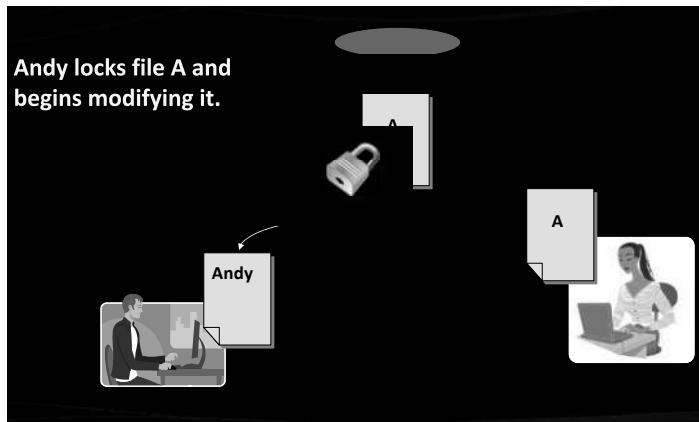
2. Versioning Models

- Lock-Modify-Unlock
- Copy-Modify-Merge
- Distributed Version Control

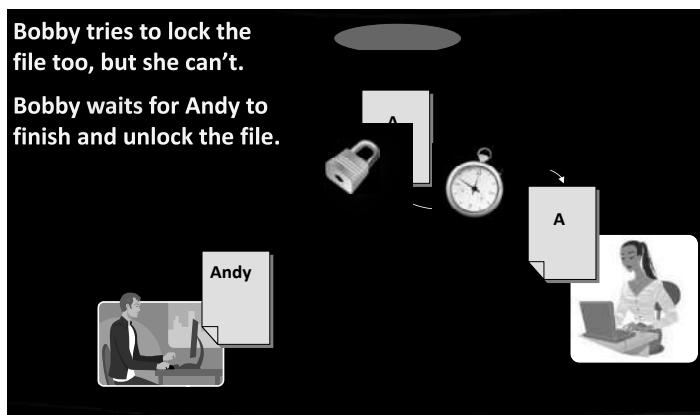
2.1. The Lock-Modify-Unlock Model



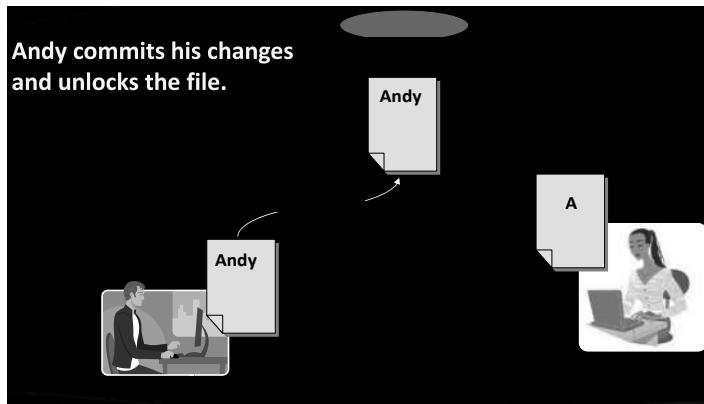
2.1. The Lock-Modify-Unlock Model (2)



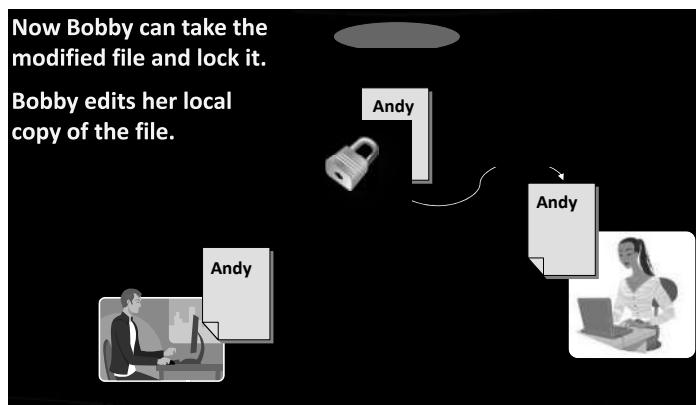
2.1. The Lock-Modify-Unlock Model (3)



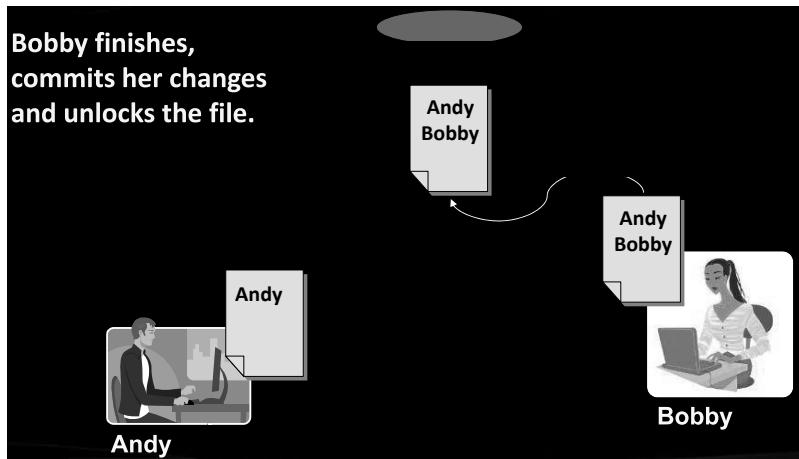
2.1. The Lock-Modify-Unlock Model (4)



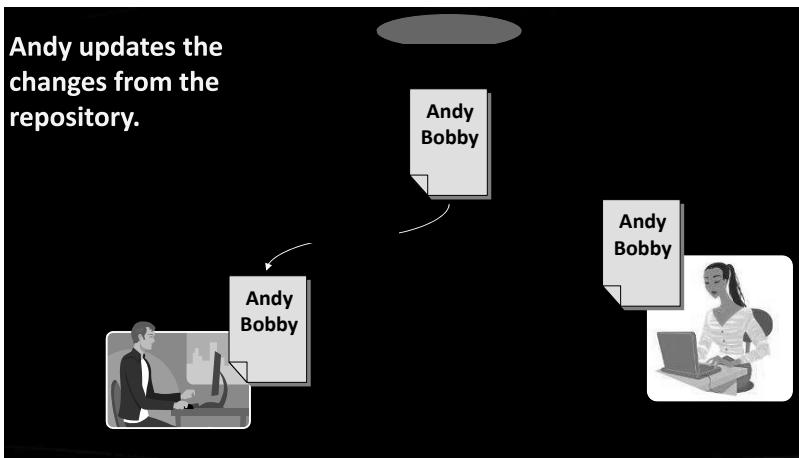
2.1. The Lock-Modify-Unlock Model (5)



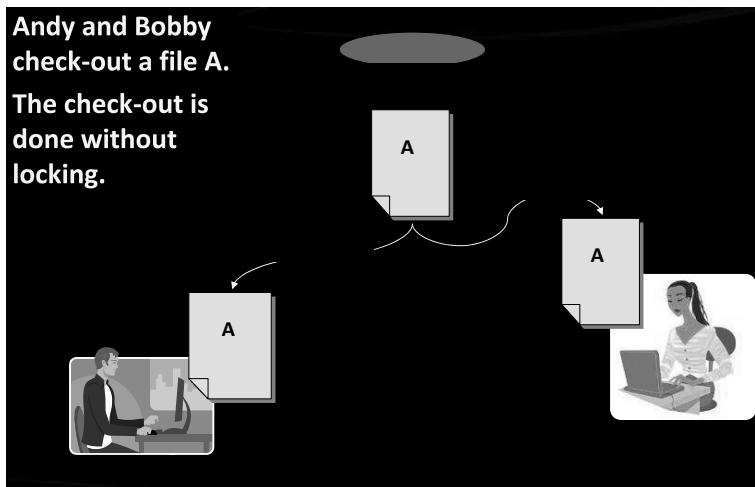
2.1. The Lock-Modify-Unlock Model (6)



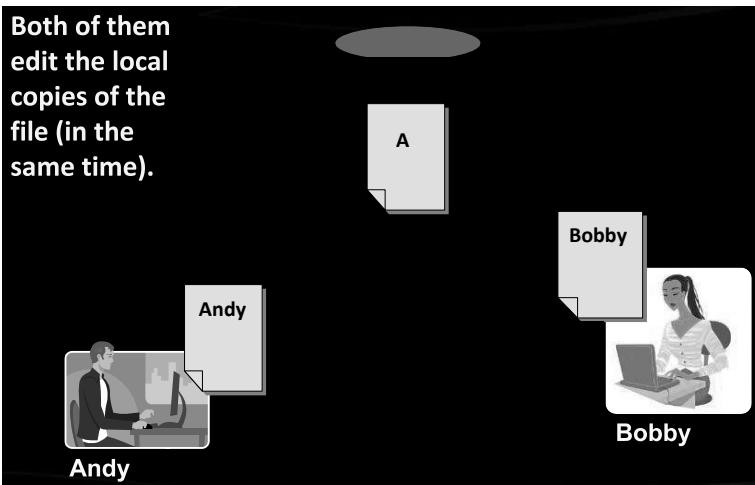
2.1. The Lock-Modify-Unlock Model (7)



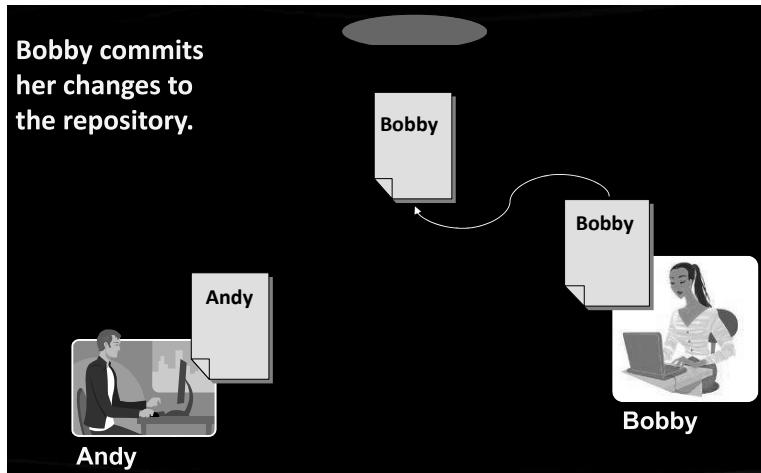
2.2. The Copy-Modify-Merge Model



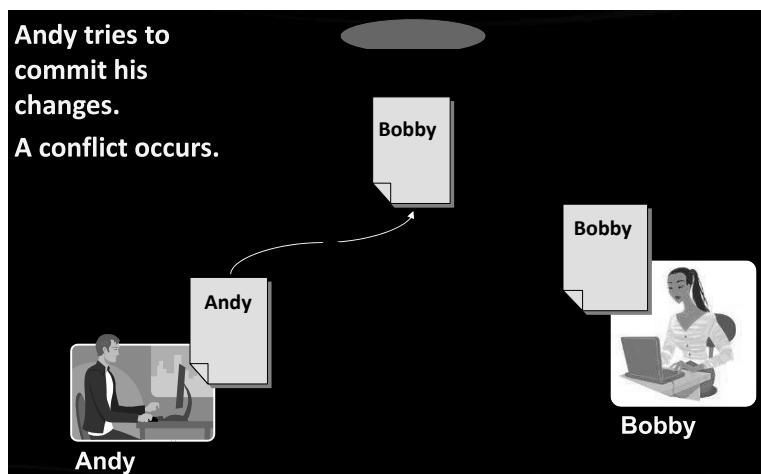
2.2. The Copy-Modify-Merge Model (2)



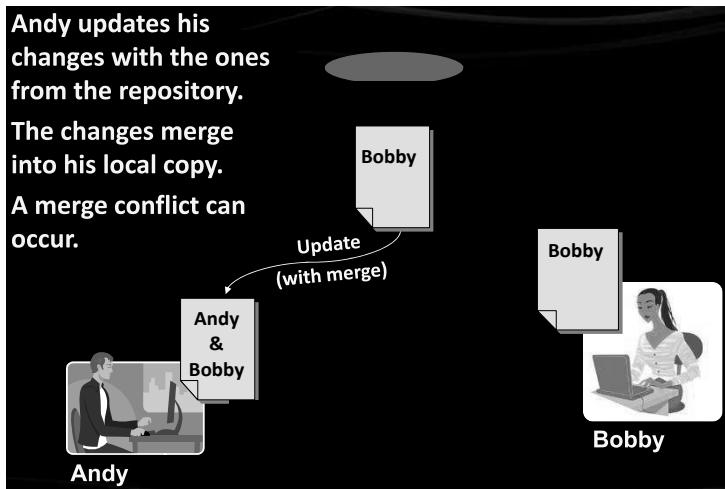
2.2. The Copy-Modify-Merge Model (3)



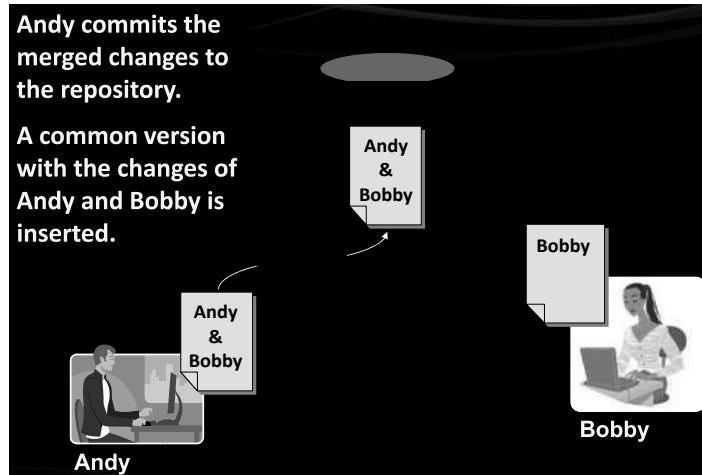
2.2. The Copy-Modify-Merge Model (4)



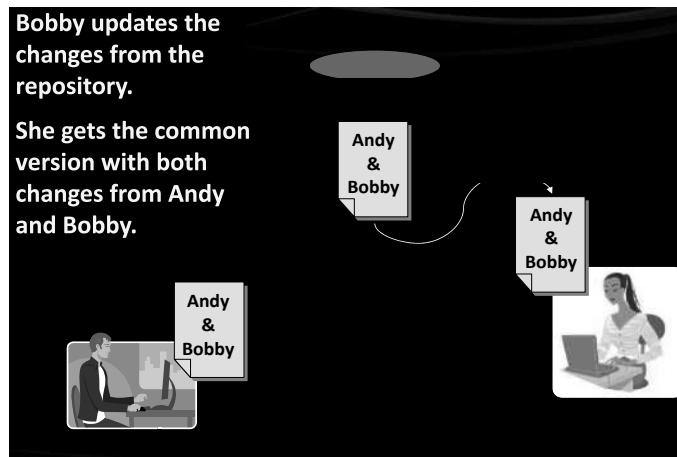
2.2. The Copy-Modify-Merge Model (5)



2.2. The Copy-Modify-Merge Model (6)



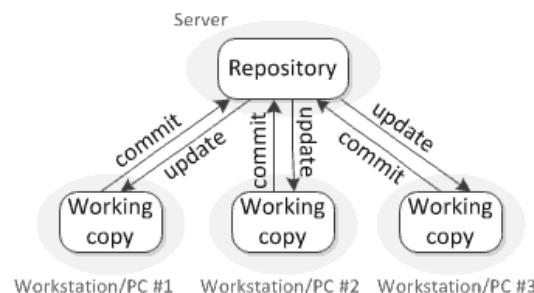
2.2. The Copy-Modify-Merge Model (7)



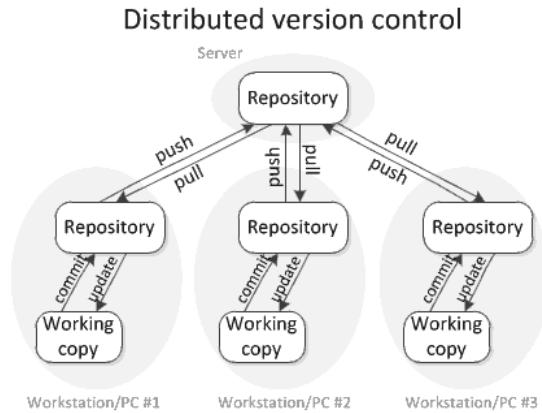
2.3. Distributed version control

- Compared to Central version control
 - Only one repository

Centralized version control



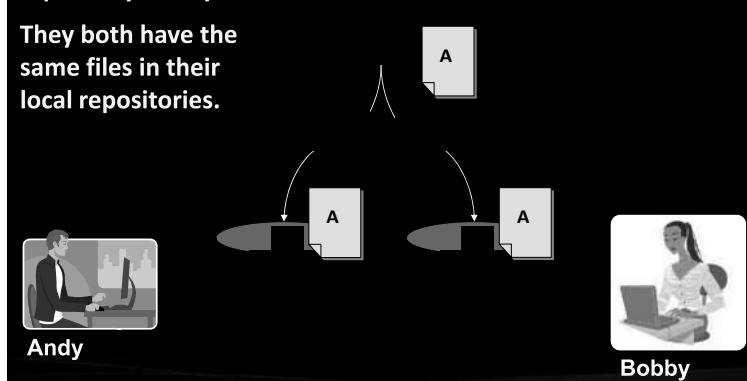
Distributed Version Control



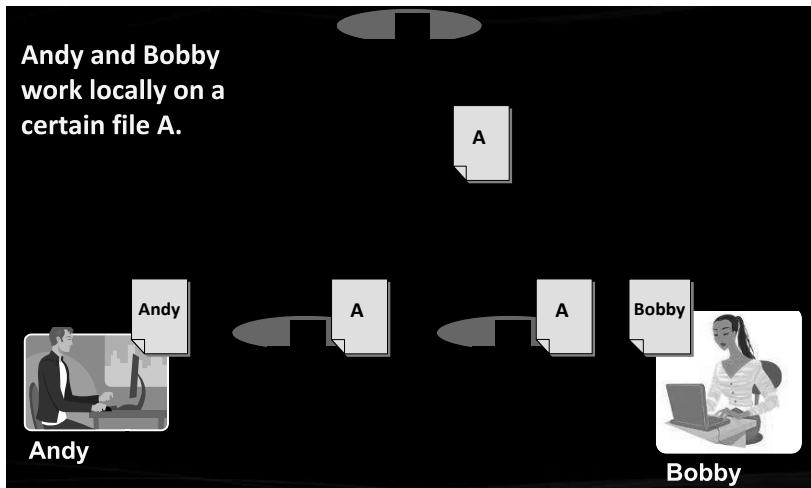
Distributed Version Control (1)

Andy and Bobby
clone the remote
repository locally.

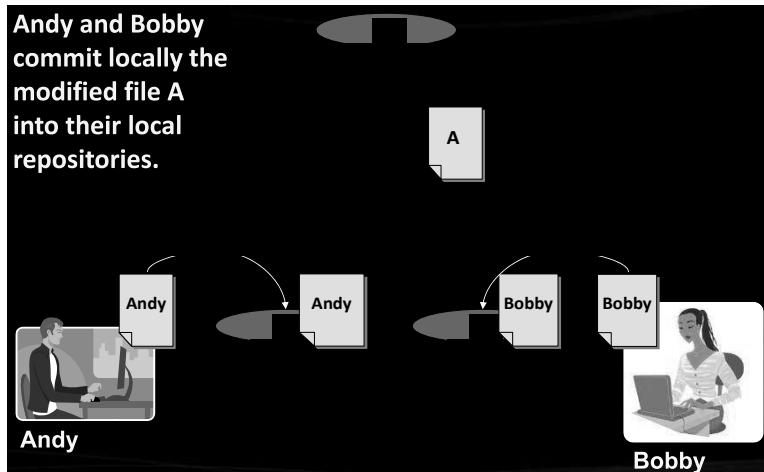
They both have the
same files in their
local repositories.



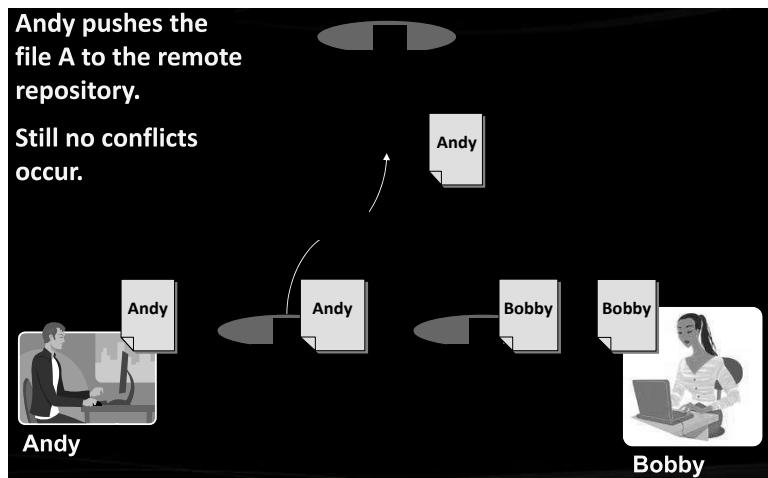
Distributed Version Control (2)



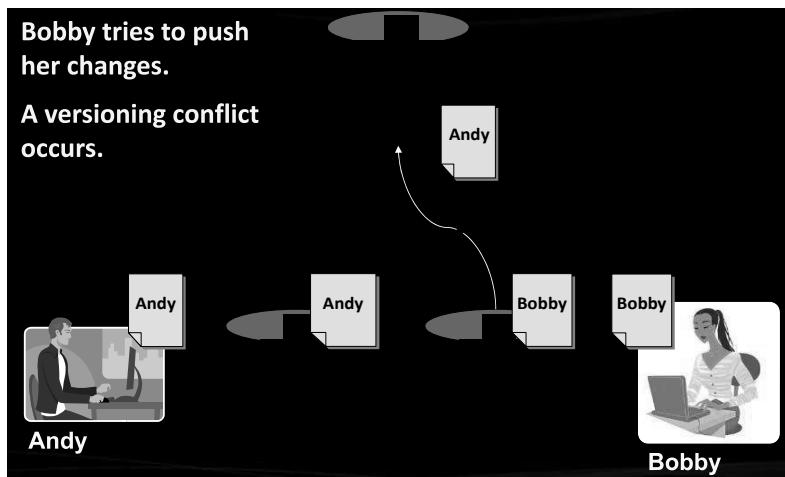
Distributed Version Control (3)



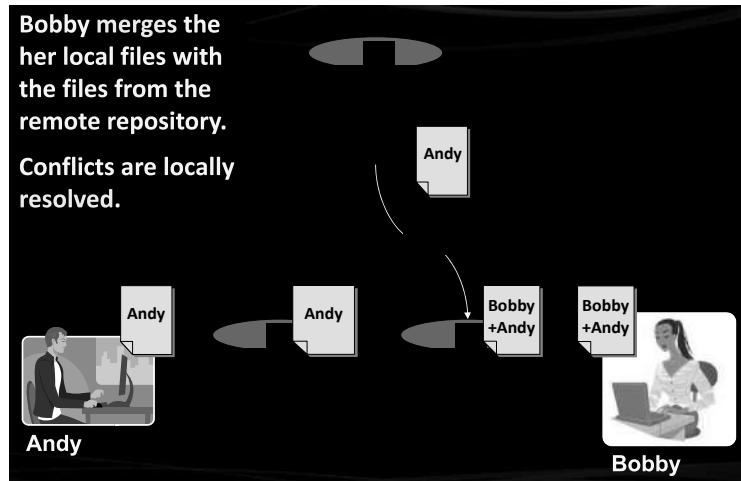
Distributed Version Control (4)



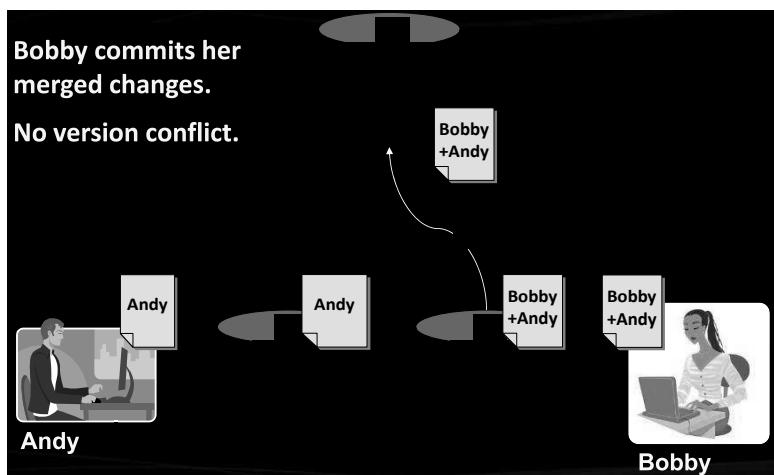
Distributed Version Control (5)



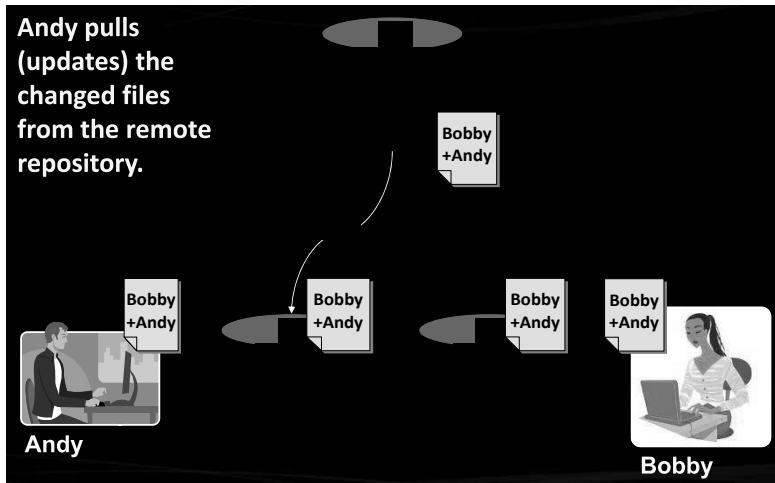
Distributed Version Control (6)



Distributed Version Control (7)



Distributed Version Control (8)



Outline

1. Introduction
2. Versioning Models
3. **Vocabulary**
4. Tools

3. Vocabulary

- Repository (source control repository)
 - A server that stores the files (documents)
 - Keeps a change log
- Revision, Version
 - Individual version (state) of a document that is a result of multiple changes
- Check-Out, Clone
 - Retrieves a working copy of the files from a remote repository into a local directory
 - It is possible to lock the files

Vocabulary

- Change
 - A modification to a local file (document) that is under version control
- Change Set / Change List
 - A set of changes to multiple files that are going to be committed at the same time
- Commit, Check-In
 - Submits the changes made from the local working copy to the repository
 - Automatically creates a new version
 - Conflicts may occur!

Vocabulary

- Conflict
 - The simultaneous change to a certain file by multiple users
 - Can be solved automatically and manually
- Update, Get Latest Version, Fetch / Pull
 - Download the latest version of the files from the repository to a local working directory + merge conflicting files
- Undo Check-Out, Revert / Undo Changes
 - Cancels the local changes
 - Restores their state from the repository

Vocabulary

- Merge
 - Combines the changes to a file changed locally and simultaneously in the repository
 - Can be automated in most cases
- Label / Tag
 - Labels mark with a name a group of files in a given version
 - For example a release
- Branch / Branching
 - Division of the repositories in a number of separate workflows

Outline

1. Introduction
2. Versioning Models
3. Vocabulary
- 4. Tools**

Tools

- Central version control
 - SVN (Subversion)
 - TFS
 - Source safe (commercial)
- Distributed version control
 - Git
 - Mercurial

What is Git?

- **Git**
 - Distributed source-control system
 - Work with local and remote repositories
 - Git bash – command line interface for Git
 - Free, open-source
 - Has Windows version (msysGit)
 - <http://msysgit.github.io>
 - <https://www.atlassian.com/git/tutorials/setting-up-a-repository>

Installing Git

- **msysGit Installation on Windows**
 - Download Git for Windows from: <http://msysgit.github.io>
 - “Next, Next, Next” does the trick
 - Options to select (they should be selected by default)
 - “Use Git Bash only”
 - “Checkout Windows-style, commit Unix-style endings”
- **Git installation on Linux:**
`sudo apt-get install git`

Basic Git Commands

- Cloning an existing Git repository
`git clone [remote url]`
- Fetch and merge the latest changes from the remote repository
`git pull`
- Preparing (adding / selecting) files for a commit
`git add [filename] ("git add ." adds everything)`
- Committing to the local repository
`git commit -m "[your message here]"`

Basic Git Commands

- Check the status of your local repository (see the local changes)
`git status`
- Creating a new local repository (in the current directory)
`git init`
- Creating a remote (assign a short name for remote Git URL)
`git remote add [remote name] [remote url]`
- Pushing to a remote (send changes to the remote repository)
`git push [remote name] [local name]`

Using Git: Example

```
mkdir work
cd work
git clone https://github.com/SoftUni/test.git dir
cd test
dir
git status
(edit some file)
git status
git add .
git commit -m "changes"
git push
```

Project Hosting Sites

- GitHub – <https://github.com>
 - The #1 project hosting site in the world
 - Free for open-source projects
 - Paid plans for private projects
- GitHub provides own Windows client
 - GitHub for Windows
 - <http://windows.github.com>
 - Dramatically simplifies Git
 - For beginners only

Project Hosting Sites

- Google Code – <http://code.google.com/projecthosting/>
 - Source control (SVN), file release, wiki, tracker
 - Very simple, basic functions only, not feature-rich
 - Free, all projects are public and open source
 - 1-minute signup, without heavy approval process
- SourceForge – <http://www.sourceforge.net>
 - Source control (SVN, Git, ...), web hosting, tracker, wiki, blog, mailing lists, file release, statistics, etc.
 - Free, all projects are public and open source

Project Hosting Sites

- CodePlex – <http://www.codeplex.com>
 - Microsoft's open source projects site
 - Team Foundation Server (TFS) infrastructure
 - Source control (TFS), issue tracker, downloads, discussions, wiki, etc.
 - Free, all projects are public and open source
- Bitbucket – <http://bitbucket.org>
 - Source control (Mercurial), issue tracker, wiki, management tools
 - Private projects, free and paid editions

UNIFIED MODELING LANGUAGE (UML)

02-1. UML & USE CASE DIAGRAM

Nguyen Thi Thu Trang
trangntt@soict.hust.edu.vn



@Nguyễn Thị Thu Trang, trangntt@soict.hust.edu.vn 2

Content

- 1. UML Overview
- 2. Requirement modeling with use-case
- 3. Use case diagrams

Discussion

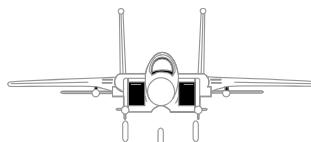
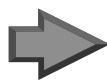
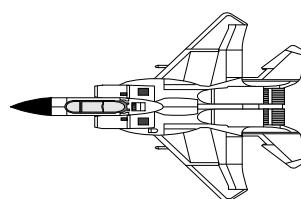
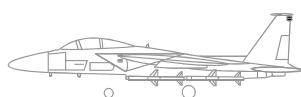
- You have a complicated object in the real world, e.g. an airplane



- How can you make it?
- How can you know its structure / design?
- ...

1.1. What Is a Model?

- A model is a simplification of reality.

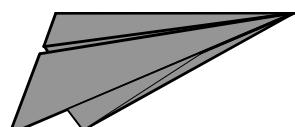


Why Model?

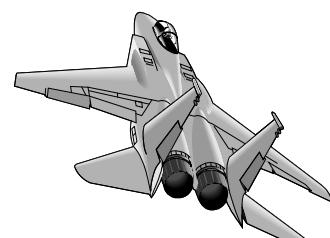
- Modeling achieves four aims:
 - Helps you to **visualize** a system as you want it to be.
 - Permits you to specify the **structure** or **behavior** of a system.
 - Gives you a **template** that guides you in constructing a system.
 - **Documents** the **decisions** you have made.
- You build models of complex systems because you cannot comprehend such a system in its entirety.
- You build models to better understand the system you are developing.

Discussion

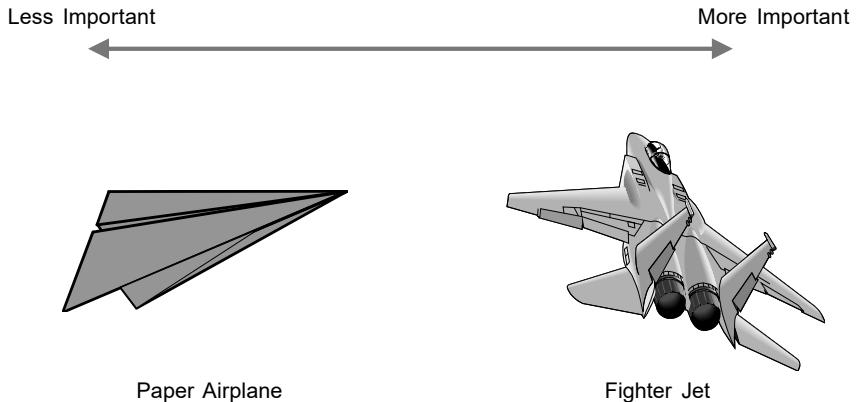
- How do you build a paper airplane?
- If it cannot fly, what will you do?



- What about a fighter jet?



The Importance of Modeling



Software Teams Often Do Not Model

- Many software teams build applications approaching the problem like they were building paper airplanes
 - Start coding from project requirements
 - Work longer hours and create more code
 - Lacks any planned architecture
 - Doomed to failure
- Modeling is a common thread to successful projects

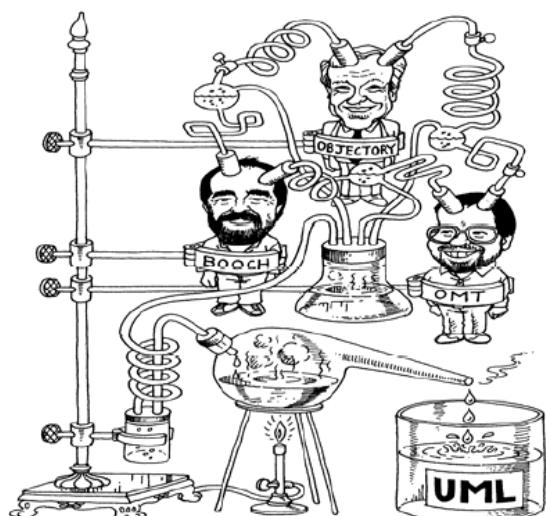
1.2. Why UML?

- 1980s: classical structural analysis and design
- 1990s: object-oriented analysis and design
- Mid-1990s: > 50 object-oriented methods with many design formats (*similar meta-models*)
 - Fusion, Shlaer-Mellor, ROOM, Class-Relation, Wirfs-Brock, Coad-Yourdon, MOSES, Syntropy, BOOM, OOSD, OSA, BON, Catalysis, COMMA, HOOD, Ooram, DOORS...

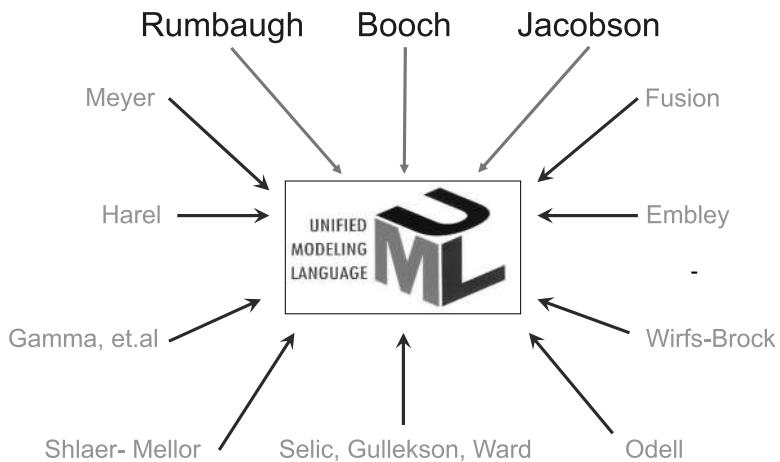
→ A unified modeling language is indispensable

UML is a standardization to a single unified language

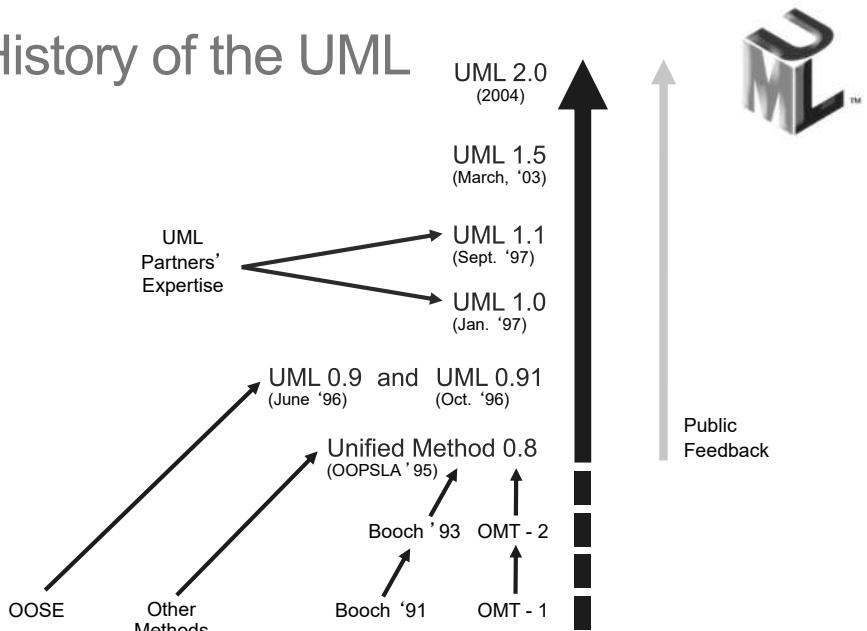
- An Object Management Group (OMG) standard.
- By 3 experts in Rational Software
 - Booch91 (Grady Booch): Conception, Architecture
 - OOSE (Ivar Jacobson): Use cases
 - OMT (Jim Rumbaugh): Analysis



Inputs to the UML



History of the UML



1.3. What Is the UML?

- The UML is a language for

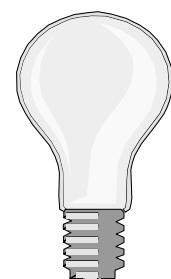
- Visualizing
- Specifying
- Constructing
- Documenting

the artifacts of a software-intensive system.



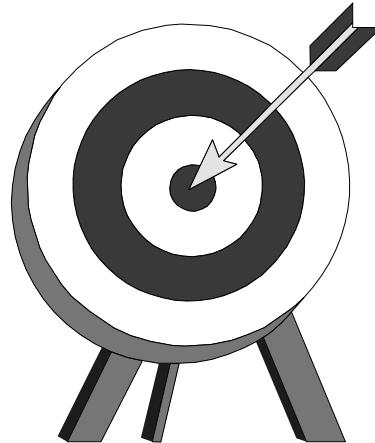
The UML Is a Language for Visualizing

- Communicating conceptual models to others is prone to error unless everyone involved speaks the same language.
- There are things about a software system you can't understand unless you build models.
- An explicit model facilitates communication.



The UML Is a Language for Specifying

- The UML builds models that are precise, unambiguous, and complete.

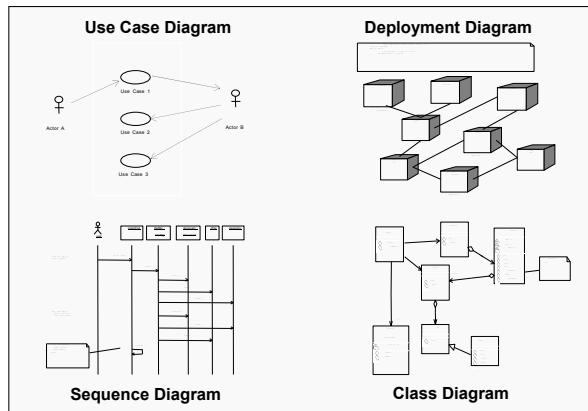


The UML Is a Language for Constructing

- UML models can be directly connected to a variety of programming languages.
 - Maps to Java, C++, Visual Basic, and so on
 - Tables in a RDBMS or persistent store in an OODBMS
 - Permits forward engineering
 - Permits reverse engineering

The UML Is a Language for Documenting

- The UML addresses documentation of system architecture, requirements, tests, project planning, and release management.



Content

1. UML Overview
2. Requirement modeling with use-case
3. Use case diagrams

Purpose of Requirement

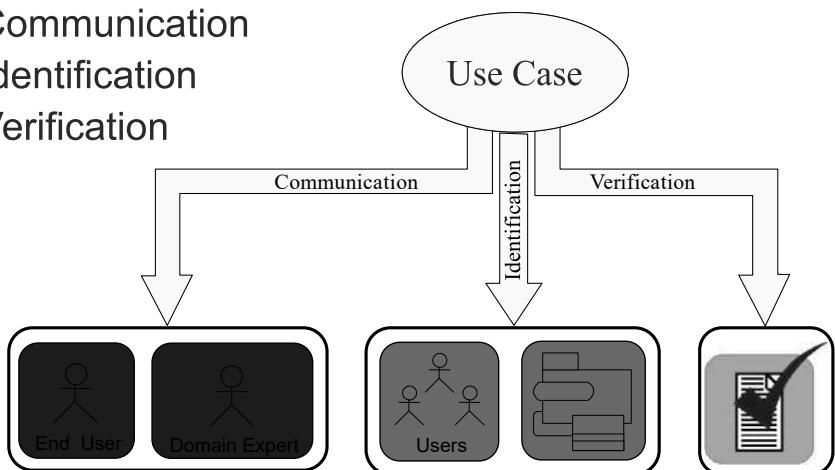
- Establish and maintain agreement with the customers and other stakeholders on what the software should do.
- Give software developers a better understanding of the requirements of the software.
- Delimit the software.
- Provide a basis for planning the technical contents of the iterations.
- Provide a basis for estimating cost and time to develop the software.
- Define a user interface of the software.

What Is Software Behavior?

- Software behavior is how a software acts and reacts.
 - It comprises the actions and activities of a software.
- Software behavior is captured in use cases.
 - Use cases describe the interactions between the software and (parts of) its environment.

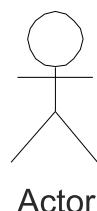
Benefits of a Use-Case Model

- Communication
- Identification
- Verification



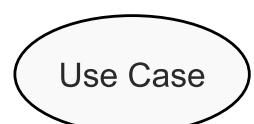
Major Concepts in Use-Case Modeling

- An actor represents anything that interacts with the software.



Actor

- A use case describes a sequence of events, performed by the software, that yields an observable result of value to a particular actor.



Use Case

Content

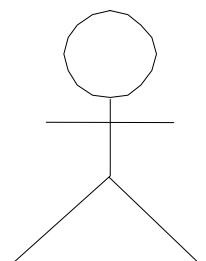
1. UML Overview

2. Requirement modeling with use-case

3. Use case diagrams

3.1. Actors

- Actors represent roles a user of the software can play
 - They can represent a human, a machine, or another software
 - They can be a peripheral device or even database
- They can actively interchange information with the software
 - They can be a giver of information
 - They can be a passive recipient of information
- Actors are not part of the software
 - Actors are EXTERNAL



Actor

Actors and Roles

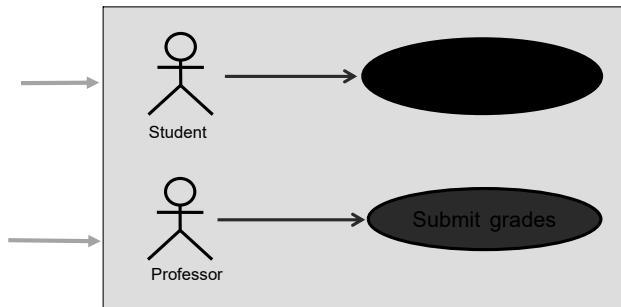


Charlie: Is employed as a math professor and is an economics undergraduate.

Jodie: Is a science undergraduate.

Charlie and Jodie both act as a Student.

Charlie also acts as a Professor.



Internet banking system

- The internet banking system, allowing interbank network, communicates with bank customers via a web application. To perform transactions, customers have to log in the software. Customers may change password or view personal information.
- Customers can select any of transaction types: transfer (internal and in interbank network), balance inquiries, transaction history inquiries, electric receipt payment (via EVN software), online saving.
- In the transfer transaction, after receiving enough information from the customer, the software asks the bank consortium to process the request. The bank consortium forwards the request to the appropriate bank. The bank then processes and responds to the bank consortium which in turn notifies the result to the software.
- The bank officers may create new account for a customer, reset password, view transaction history of a customer.

Some guideline to extract actors

- Pay attention to a noun in the problem description, and then extract a subject of action as a Actor.
- Ensure that there are no any excesses and deficiencies between the problem description and Actors extracted.
- Actor names
 - should clearly convey the actor's role
 - good actor names describe their responsibilities

Exercise: Find actors

Internet Banking Software

3.2. Use Cases

- Define a set of use-case instances, where each instance is a sequence of actions a software performs that yields an observable result of value to a particular actor.
 - A use case models a **dialogue** between one or more actors and the software
 - A use case describes the **actions the software takes** to deliver something of value to the actor

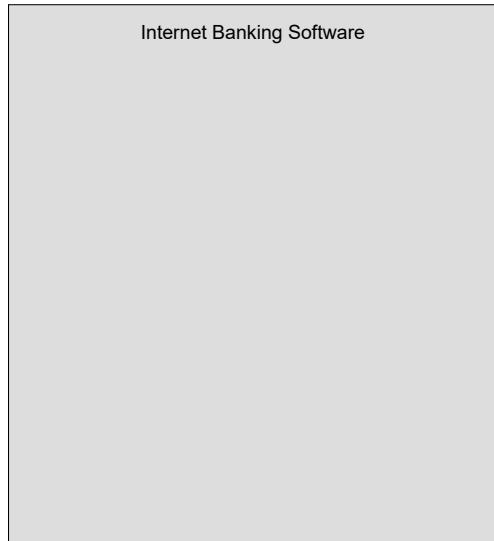


Use Case

Some guidelines to extract use cases

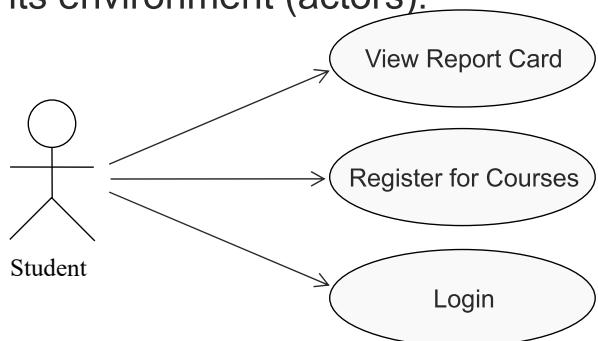
- Pay attention to a verb in the problem description, and then extract a series of Actions as a UC.
- Ensure that there are no any excesses and deficiencies between the problem description and Use cases extracted.
- Check the consistency between Use Cases and related Actors.
- Conduct a survey to learn whether customers, business representatives, analysts, and developers all understand the names and descriptions of the use cases

Exercise: Find use cases



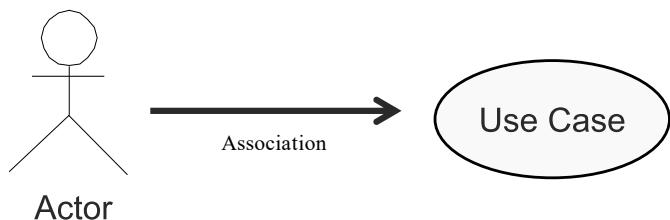
3.3. Use-Case Diagram

- A diagram modeling the dynamic aspects of softwares that describes a software's functional requirements in terms of use cases.
- A model of the software's intended functions (use cases) and its environment (actors).

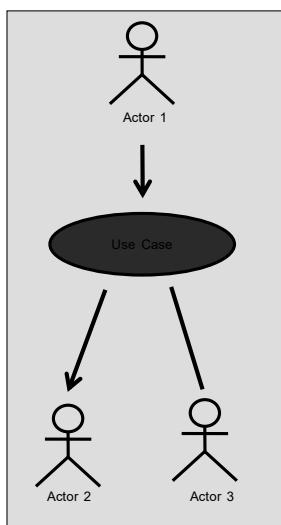


Association between actor and use case

- Establish the actors that interact with related use cases
 - Associations clarify the **communication** between the actor and use case.
 - Association indicate that the actor and the use case instance of the software communicate with one another, each one able to **send and receive messages**.
- The arrow head is optional but it's commonly used to denote the initiator.

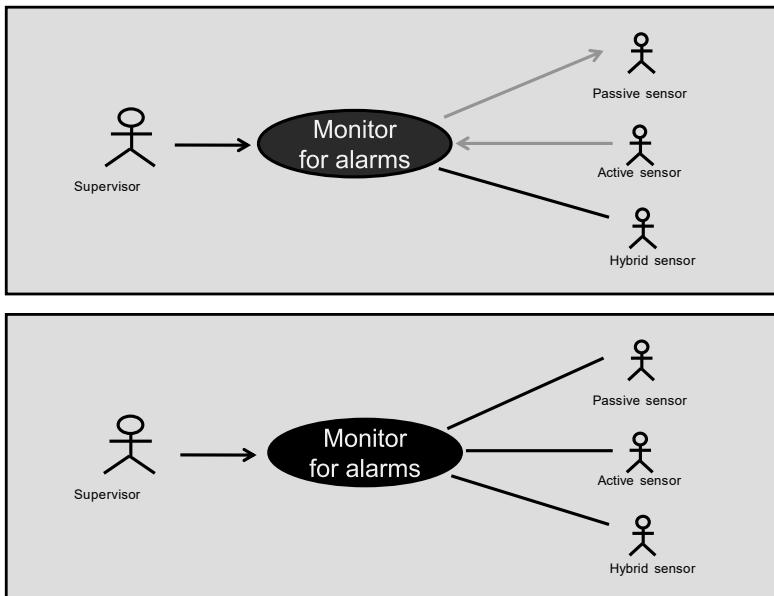


Communicates-Association

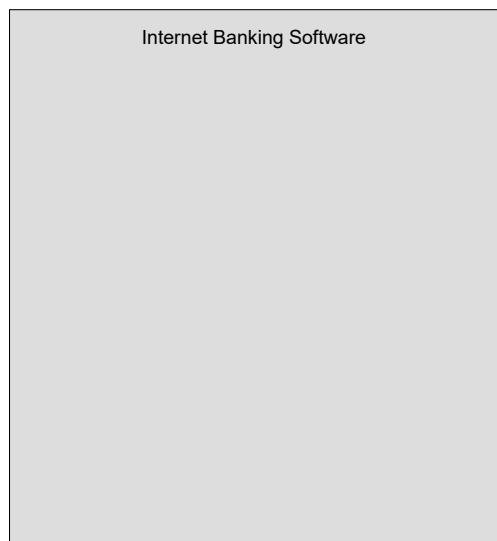


- A channel of communication between an actor and a use case.
- A line is used to represent a communicates-association.
 - An arrowhead indicates who initiates each interaction.
 - No arrowhead indicates either end **can** initiate each interaction.

Arrowhead Conventions



Exercise: Draw use case diagram



Question?



@NGUYỄN Thị Thu Trang, trangntt@soict.hust.edu.vn

OBJECT-ORIENTED LANGUAGE AND THEORY

02-2. JAVA BASICS

Nguyen Thi Thu Trang
trangntt@soict.hust.edu.vn



Content

- 1. Identifiers
- 2. Data Types
- 3. Operators
- 4. Control Statements
- 5. Arrays

3

1. Identifiers

- Identifiers:
 - A set of characters representing variables, methods, classes and labels
- Naming rules:
 - Characters can be numbers, alphabets, '\$' or '_'
 - Name must not:
 - Start by a Number
 - Be the same as a keyword
 - Distinguish between UpperCases and LowerCases
 - Yourname, yourname, YourName and yourName are for different identifiers

An_Identifier
a_2nd_Identifier
Go2
\$10



An-Identifier
2nd_Identifier
goto
10\$



1. Identifiers (2)

- Naming convention:
 - Start with an Alphabet
 - Package: all in lowercase
 - theexample
 - Class: the first letter of word is in uppercase
 - TheExample
 - Method/field: start with a lowercase letter, the first letter of remaining word is in uppercase
 - theExample
 - Constants: All in uppercase
 - THE_EXAMPLE

1. Identifiers (3)

- Literals

null true false

- Keyword

abstract assert boolean break byte case catch
 char class continue default do double else
 extends final finally float for if implements
 import instanceof int interface long native new
 package private protected public return short
 static strictfp super switch synchronized this
 throw throws transient try void volatile while

- Reserved for future use

byvalue cast const future generic goto inner operator
 outer rest var volatile

Content

1. Identifiers

→ 2. Data Types

3. Operators

4. Control Statements

5. Arrays

7

2. Data Types

- Two categories:
 - Primitive
 - Integer
 - Float
 - Char
 - Logic (boolean)
 - Reference
 - Array
 - Object

2.1. Primitives

- Every variable must be declared with a data type:
 - Primitive data type contains a single value
 - Size and format must be appropriate to its data type
- Java has 4 primitive data types

Categories:

- a. **integer**
- b. **floating point**
- c. **character**
- d. **boolean**

Integer

- Signed integer
- Initially created with value 0

Categories:

- a. **integer**
- b. **floating point**
- c. **character**
- d. **boolean**

1. byte
2. short
3. int
4. long

Size: 1 byte
Range: $-2^7 \rightarrow 2^7 - 1$

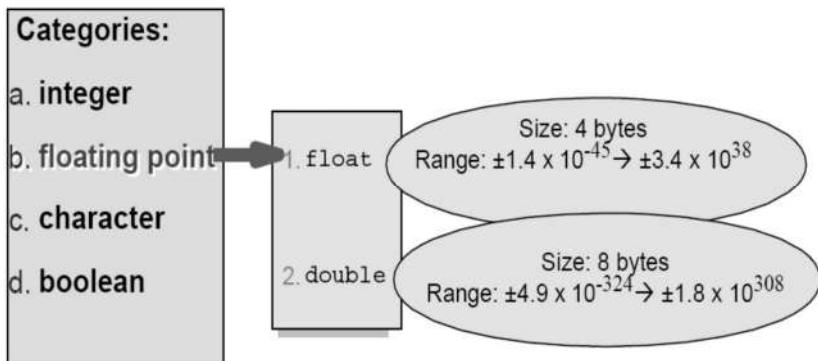
Size: 2 bytes
Range: $-2^{15} \rightarrow 2^{15} - 1$

Size: 4 bytes
Range: $-2^{31} \rightarrow 2^{31} - 1$

Size: 8 bytes
Range: $-2^{63} \rightarrow 2^{63} - 1$

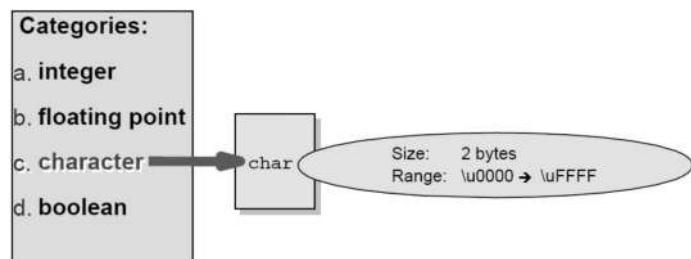
Real

- Initially created with value 0.0



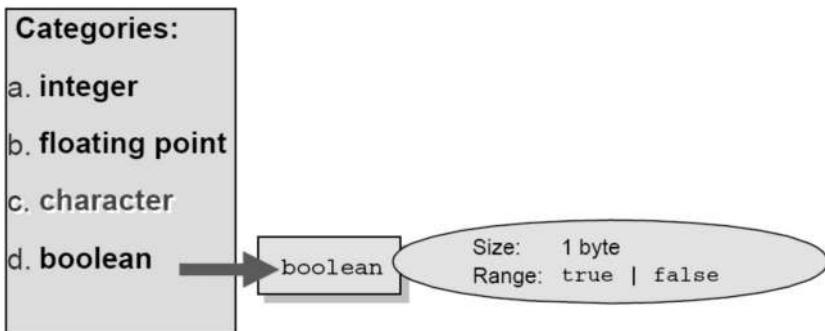
Characters

- Unsigned Unicode characters, placed between two single quotes
- 2 ways to assign value:
 - Using number in hexa: `char uni = '\u05Do';`
 - Using character: `char a = 'A';`
- Default value is zero (`\u0000`)



Logic value

- Boolean value is clearly specified in Java
 - An int value can not be used for a boolean value
 - Can store value “true” or “false”
- Boolean variable is initially created with false value



2.2. Literal

- Literal is a value of a set of primitive data types and character string.
- Has 5 categories:
 - integer
 - floating point
 - boolean
 - character
 - string

<u>Literals</u>	
integer.....	7
floating point...	7.0f
boolean.....	true
character.....	'A'
string.....	"A"

Literal of Integer

- Octals start with number 0
 - $032 = 011\ 010_2 = 16 + 8 + 2 = 26_{10}$
- Hexadecimals start with number 0 and character x
 - $0x1A = 0001\ 1010_2 = 16 + 8 + 2 = 26_{10}$
- Ends with character “L” representing data type long
 - 26L
- Uppercase characters, usually have the same values
 - 0x1a , 0x1A , 0X1a , 0X1A đều có giá trị 26 trong hệ decimal

Literal of Real

- Float** ends with character f (or F)
 - 7.1f
- Double** ends with character d (or D)
 - 7.1D
- e** (or **E**) is used in scientific representation:
 - 7.1e2
- A value without ending character is considered as a **double**
 - 7.1 giống như 7.1d

Literal of boolean, character and string

- boolean:
 - true
 - false
- Character:
 - Located between two single quotes
 - Example: 'a', 'A' or '\uffff'
- String:
 - Located between two double quotes
 - Example: "Hello world", "Xin chao ban",...

Escape sequence

- Characters for keyboard control
 - \b backspace
 - \f form feed
 - \n newline
 - \r return (về đầu dòng)
 - \t tab
- Display special characters in a string
 - \" quotation mark
 - \' apostrophe
 - \\ backslash

2.3. Casting

- Java is a strong-type language
 - Casting a wrong type to a variable can lead to a compiler error or exceptions in JVM
- JVM can implicitly cast a data type to a larger data type
- To cast a variable to a narrower data type, we need to do it explicitly

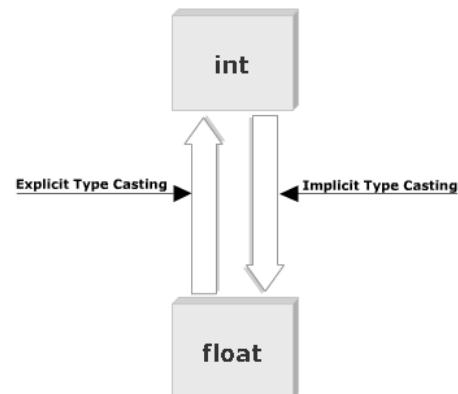
```
int a, b;
short c;
a = b + c;
```

```
int d;
short e;
e = (short)d;
```

```
double f;
long g;
f = g;
g = f; //error
```

c. Casting (2)

- Casting is done automatically if no information loss occurs
 - byte → short → int → long → float → double
- Explicit cast is required if there is a “risk” of reduced accuracy



Example - Casting

```
long p = (long) 12345.56; // p == 12345
int g = p;    // syntax error although int
              // can hold a value of 12345
char c = 't';
int j = c;    // implicit casting
short k = c; // error
short k = (short) c; // explicit casting
float f = 12.35; // error
```

2.4. Declaration and Creation of Variables

- Simple variables (that are not array) need to be initialized before being used in expressions
 - Can declare and initialize at the same time.
 - Using = to assign (including initialization)
 - Example:

```
int i, j;    // Variable declaration
i = 0;
int k = i+1;
float x=1.0f, y=2.0f;
System.out.println(i); // Print out 0
System.out.println(k); // Print out 1
System.out.println(j); // Compile error
```

Comments

- Java supports three types of comments:
 - // Comments in a single line
 // Without line break
 - /* Comments as a paragraph */
 - /** Javadoc * comments in form of Javadoc */

Command

- Command ends with;
- Multiple commands can be written on one line
- A command can be written in multiple lines
 - Example:

```
System.out.println(  
    "This is part of the same line");
```

```
a=0; b=1; c=2;
```

Content

1. Identifiers

2. Data Types

3. Operators

4. Control Statements

5. Arrays

3. Operators

- Combining single values or child expressions into a new expression, more complex and can return a value.
- Java provides the following operators:
 - Arithmetic operators
 - Bit operator, Relational operators
 - Logic operators
 - Assignment operators
 - Unary operators

✓ A = B + C
✓ Z = Y * Y
✓ Result = (A+B) > (C+D)



3. Operators (2)

- Arithmetic operators
 - +, -, *, /, %
- Bit operators
 - AND: &, OR: |, XOR: ^, NOT: ~
 - bit: <<, >>
- Relational operators
 - ==, !=, >, <, >=, <=
- Logic operators
 - &&, ||, !

3. Operators (3)

- Unary operators
 - Reverse sign : +, -
 - Increase/decrease by 1 unit: ++, --
 - Negation of a logic expression: !
- Assignment operators
 - =, +=, -=, %= similar to >>, <<, &, |, ^

Priorities of Operators

- Define the order of performing operators – are identified by parentheses or by default as follows:
 - Postfix operators [] . (params) `x++ x--`
 - Unary operators `++x --x +x -x ~ !`
 - Creation or cast `new (type)x`
 - Multiplicative `* / %`
 - Additive `+ -`
 - Shift `<< >> >>> (unsigned shift)`
 - Relational `< > <= >= instanceof`
 - Equality `== !=`
 - Bitwise AND `&`
 - Bitwise exclusive OR `^`
 - Bitwise inclusive OR `|`
 - Logical AND `&&`
 - Logical OR `||`
 - Conditional (ternary) `? :`
 - Assignment `= *= /= %= += -= >>= <<= >>>= &= ^= |=`

Content

1. Identifiers
2. Data Types
3. Operators
4. Control Statements
5. Arrays

4.1. if - else statement

- Syntax

```
if (condition){  
    statements;  
}  
  
else {  
    statements;  
}
```

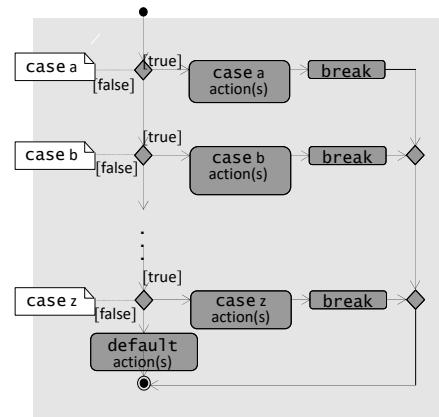
- condition expression can receive boolean value
- else expression is optional

Example - Checking odd - even numbers

```
class CheckNumber  
{  
    public static void main(String args[])  
    {  
        int num =10;  
        if (num %2 == 0)  
            System.out.println (num+ "la so chan");  
        else  
            System.out.println (num + "la so le");  
    }  
}
```

4.2. switch - case statement

- Checking a single variable with different values and perform the corresponding case
 - break: exits switch-case command
 - Default: manages values outside the values defined in case:



Example - switch - case

```

switch (day) {
    case 0:
    case 1:
        rule = "weekend";
        break;
    case 2:
        ...
    case 6:
        rule = "weekday";
        break;
    default:
        rule = "error";
}

```

```

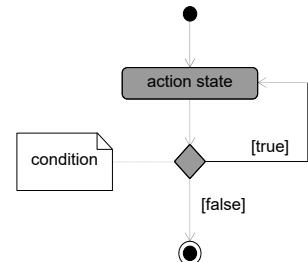
if (day == 0 || day == 1) {
    rule = "weekend";
} else if (day > 1 && day < 7) {
    rule = "weekday";
} else {
    rule = error;
}

```

4.3. while and do while statements

- Perform a command or a command block while the control expression is still true
 - while() performs 0 or multiple times
 - do...while() performs at least 1 time

```
int x = 2;
while (x < 2) {
    x++;
    System.out.println(x);
}
```



```
int x = 2;
do {
    x++;
    System.out.println(x);
} while (x < 2);
```

Example - while loop

```
class WhileDemo{
    public static void main(String args[]){
        int a = 5,fact = 1;
        while (a >= 1){
            fact *=a;
            a--;
        }
        System.out.println("The Factorial of 5
                           is "+fact);
    }
}
```

4.4. for loop

- Syntax:

```
for (start_expr; test_expr; increment_expr){  
    // code to execute repeatedly  
}
```

- 3 expressions can be absent
 - A variable can be declared in for command:
 - Usually declare a counter variable
 - Usually declare in “start” expression
 - Variable scope is in the loop

- Example:

```
for (int index = 0; index < 10; index++) {  
    System.out.println(index);  
}
```

Example - for loop

Commands for changing control structure

- **break**

- Can be used to exit switch command
- Terminate loops for, while or do...while
- There are two types:
 - Labeling: continue to perform commands after the labeled loop
 - No-Labeling: perform next commands outside the loop

- **continue**

- Can be used for for, while or do...while loops
- Ignore the remaining commands of the current loop and perform the next iteration.

Example - break and continue

```
public int myMethod(int x) {
```

```
    int sum = 0;
```

```
    outer: for (int i=0; i<x; i++) { ←
```

```
        inner: for (int j=i; j<x; j++) { ←
```

```
            sum++;
```

```
            if (j==1) continue; →
```

```
            if (j==2) continue outer; →
```

```
            if (i==3) break; →
```

```
            if (j==4) break outer;
```

```
        }
```

```
    }
```

```
    return sum;
```

```
}
```

Variable scope

- Scope of a variable is a program area in which that variable is referred to
 - Variables declared in a method can only be accessed inside that method
 - Variables declared in a loop or code block can only be accessed in that loop or that block

```
int a = 1;
for (int b = 0; b < 3; b++) {
    int c = 1;
    for (int d = 0; d < 3; d++) {
        if (c < 3) c++;
    }
    System.out.print(c);
    System.out.println(b);      abc
}
a = c; // ERROR! c is out of scope      a
```

Content

1. Identifiers
2. Data Types
3. Operators
4. Control Statements
5. Arrays

5. Array

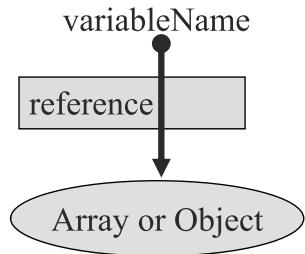
- Finite set of elements of the same type
- Must be declared before use
- Declaration:

- Syntax:

- `datatype[] arrayName= new datatype[ARRAY_SIZE];`
- `datatype arrayName[] = new datatype[ARRAY_SIZE];`

- Example:

- `char c[] = new char[12];`



Array declaration and initialization

- Declaration and initialization:

- Syntax:

- `datatype[] arrayName = {initial_values};`

- Example:

- `int[] numbers = {10, 9, 8, 7, 6};`

- Without initialization → receives the default value depending on the data type.

- Always starts with the element of index 0

Example - Array

The diagram shows a 1D array `c` with 12 elements. The array is represented as a vertical stack of 12 boxes, each containing a value. The values are: -45, 6, 0, 72, 1543, -89, 0, 62, -3, 1, 6453, and 78. To the left of the array, three annotations are present: 1) "Array name (all the elements of array have the same name, `c`)" with an arrow pointing to the first element `c[0]`. 2) "`c.length`: length of the array `c`" with an arrow pointing to the value 12, which is the total number of elements. 3) "Index (to access to the elements of array)" with an arrow pointing to the index `c[11]`, which is the last element shown.

	<code>c[0]</code>	-45
	<code>c[1]</code>	6
	<code>c[2]</code>	0
	<code>c[3]</code>	72
	<code>c[4]</code>	1543
	<code>c[5]</code>	-89
	<code>c[6]</code>	0
	<code>c[7]</code>	62
	<code>c[8]</code>	-3
	<code>c[9]</code>	1
	<code>c[10]</code>	6453
	<code>c[11]</code>	78

Array declaration and initialization - Example

```

int MAX = 5;
boolean bit[] = new boolean[MAX];
float[] value = new float[2*3];
int[] number = {10, 9, 8, 7, 6};
System.out.println(bit[0]); // prints "false"
System.out.println(value[3]); // prints "0.0"
System.out.println(number[1]); // prints "9"

```

Multi-dimensional array

- Table with rows and columns
 - Usually use two-dimensional array
 - Example of declaration `b[2][2]`
 - `int b[][] = { { 1, 2 }, { 3, 4 } };`
 - 1 and 2 are initialized for `b[0][0]` and `b[0][1]`
 - 3 and 4 are initialized for `b[1][0]` and `b[1][1]`
 - `int b[3][4];`

Multi-dimensional array (2)

	Column 0	Column 1	Column 2	Column 3
Row 0	<code>b[0][0]</code>	<code>b[0][1]</code>	<code>b[0][2]</code>	<code>b[0][3]</code>
Row 1	<code>b[1][0]</code>	<code>b[1][1]</code>	<code>b[1][2]</code>	<code>b[1][3]</code>
Row 2	<code>b[2][0]</code>	<code>b[2][1]</code>	<code>b[2][2]</code>	<code>b[2][3]</code>

The diagram illustrates a 3x4 multi-dimensional array. The array is represented by a grid of 12 cells. The first three rows are labeled Row 0, Row 1, and Row 2 from top to bottom. The first four columns are labeled Column 0, Column 1, Column 2, and Column 3 from left to right. The cells contain the following values: Row 0: b[0][0], b[0][1], b[0][2], b[0][3]. Row 1: b[1][0], b[1][1], b[1][2], b[1][3]. Row 2: b[2][0], b[2][1], b[2][2], b[2][3]. Below the array, a large bracket indicates the 'Array name'. To the left of the array, a vertical bracket indicates the 'Row index'. To the right of the array, a horizontal bracket indicates the 'Column index'.

OBJECT-ORIENTED LANGUAGE AND THEORY

3. ABSTRACTION & ENCAPSULATION

Nguyen Thi Thu Trang
trangtt@soict.hust.edu.vn



2

Outline

- 1. Abstraction
- 2. Encapsulation and Class Building
- 3. Object Creation and Communication

1.1. Abstraction

- Reduce and factor out details so that one can focus on a few concepts at a time
 - “abstraction – a concept or idea not associated with any specific instance”.
- Example: Mathematics definition
 - $1 + 2$

```
1) Store 1,Location A  
2) Store 2,Location B  
3) Add Location A, Location B  
4) Store Results
```

1.2. Abstraction in OOP

- Objects in reality are very complex

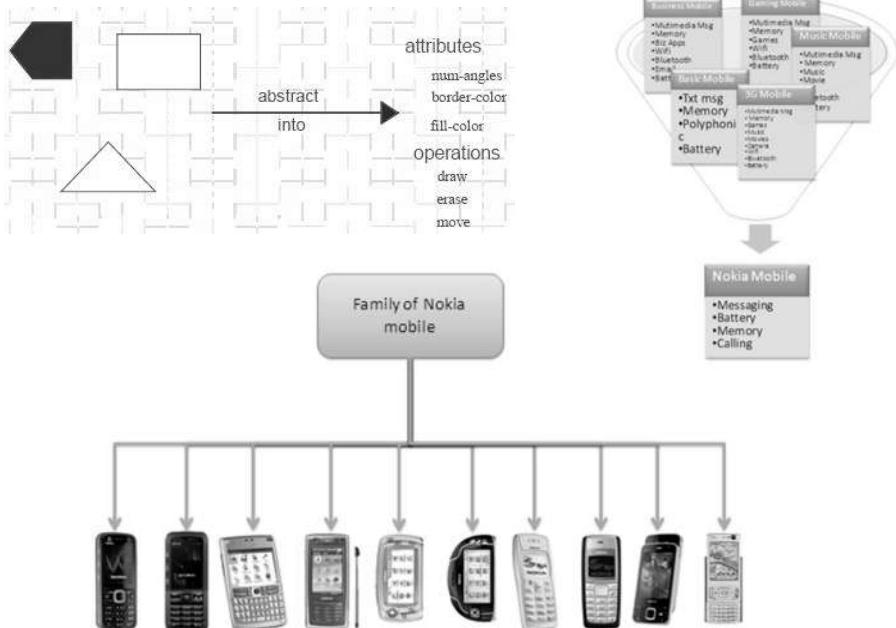


- Need to be simplified by ignoring all the unnecessary details
- Only “extract” related/involving, important information to the problem

Example: Abstracting Nokia phones



- What are the common properties of these entities? What are particular properties?
 - All are Nokia phones
 - Sliding, folding, ...
 - Phones for Businessman, Music, 3G
 - QWERTY keyboard, Basic Type, No-keyboard type
 - Color, Size, ...



1.2. Abstraction (3)

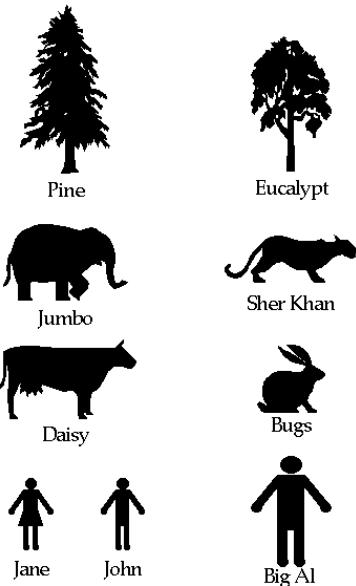
- Any model that includes the most important, essential, or distinguishing aspects of something while suppressing or ignoring less important, immaterial, or diversionary details. The result of removing distinctions so as to emphasize commonalities (*Dictionary of Object Technology*, Firesmith, Eykholt, 1995).
 - Allow managing a complex problem by focusing on important properties of an entity in order to distinguish with other entities

1.2. Abstraction (4)

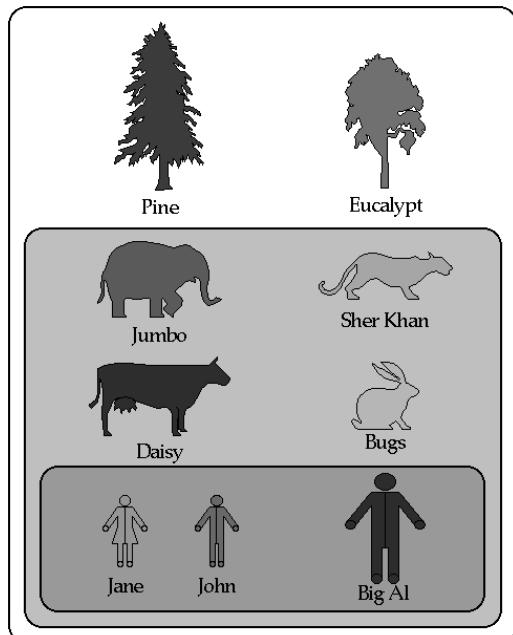
- **ABSTRACTION** is a view of an entity containing only related properties in a context
- **CLASS** is the result of the abstraction, which represents a group of entities with the same properties in a specific view



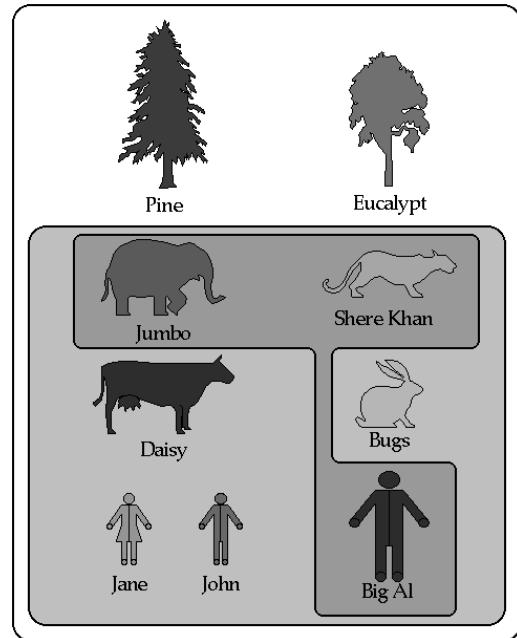
unclassified "things"



- organisms, mammals, humans



- organisms, mammals, dangerous mammals



1.3. Class vs. Objects

- Class is concept model, describing entities
- Class is a prototype/blueprint, defining common properties and methods of objects
- A class is an abstraction of a set of objects.
- ◆ Objects are real entities
- ◆ Object is a representation (instance) of a class, building from the blueprint
- ◆ Each object has a class specifying its data and behavior; *data of different objects are different*

Class representation in UML

Professor

- Class is represented by a rectangle with three parts:
 - Class name
 - Structure (Attributes)
 - Behavior (Operation)

Professor
<ul style="list-style-type: none"> - name - employeeID : UniqueID - hireDate - status - discipline - maxLoad
<ul style="list-style-type: none"> + submitFinalGrade() + acceptCourseOffering() + setMaxLoad() + takeSabbatical() + teachClass()

What is attribute?

- An attribute is a named characteristic of a class specifying a value range of its representations.
 - A class might have no property or any number of properties.

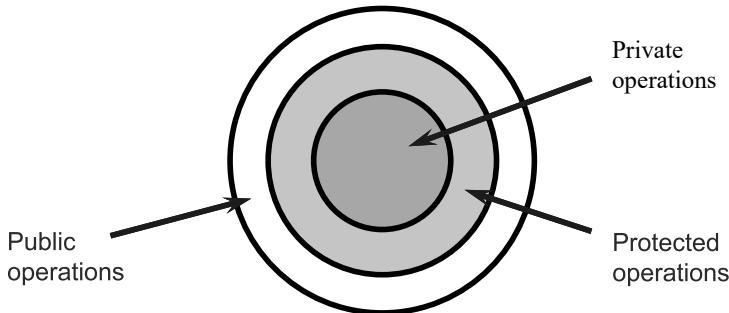
Attributes



Student
- name
- address
- studentID
- dateOfBirth

Operation Visibility

- Visibility is used to enforce encapsulation
- May be public, protected, or private

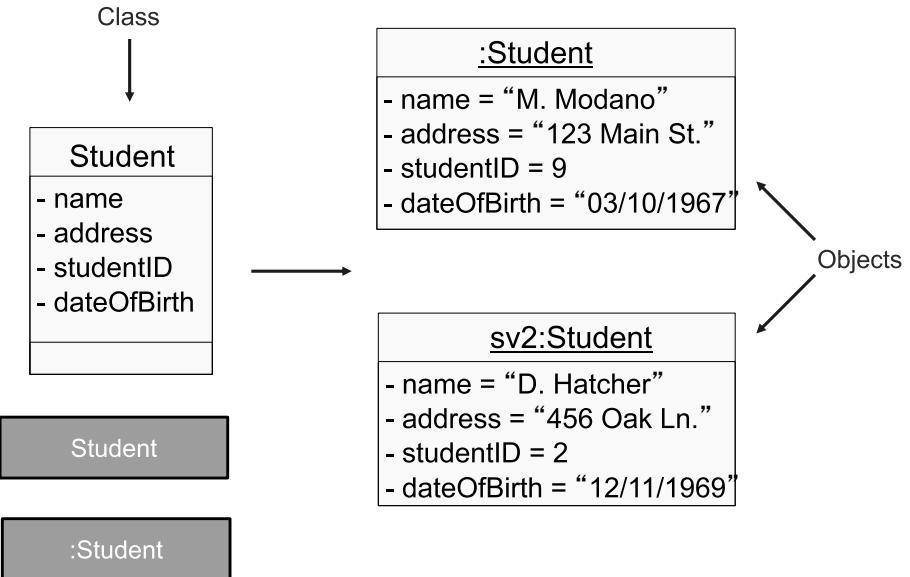


How Is Visibility Noted?

- The following symbols are used to specify export control:
 - + Public access
 - # Protected access
 - - Private access

ClassName
- privateAttribute + publicAttribute # protectedAttribute
- privateOperation () + publicOperation () # protecteOperation ()

Class and Object in UML



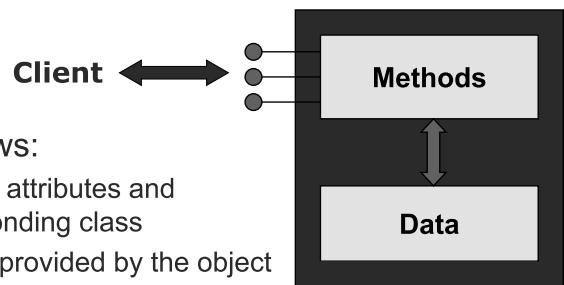
Outline

1. Abstraction

→ 2. Encapsulation and Class Building

3. Object Creation and Communication

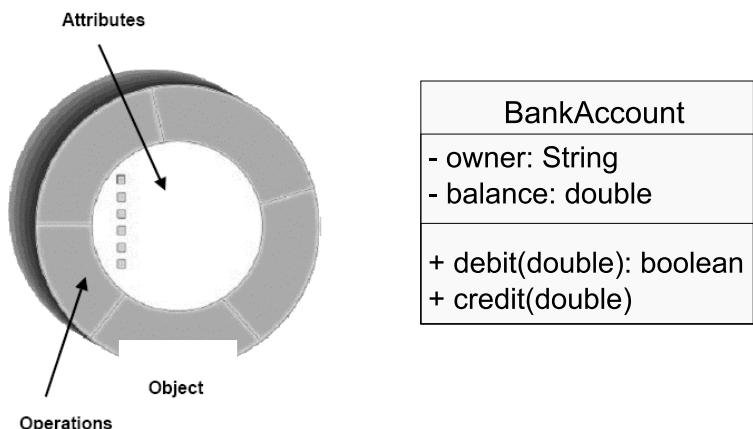
2.1. Encapsulation



- An object has two views:
 - Internal view: Details on attributes and methods of the corresponding class
 - External view: Services provided by the object and how the object communicates with all the rest of the system

2.1. Encapsulation (2)

- Data/attributes and behaviors/methods are encapsulated in a class → Encapsulation
 - Attributes and methods are members of the class



2.2. Class Building

BankAccount

- owner: String
- balance: double

+ debit(double): boolean
+ credit(double)

- **Class name**

- Specify what the abstraction is capturing
- Should be singular, short, and clear identify the concept

- **Data elements**

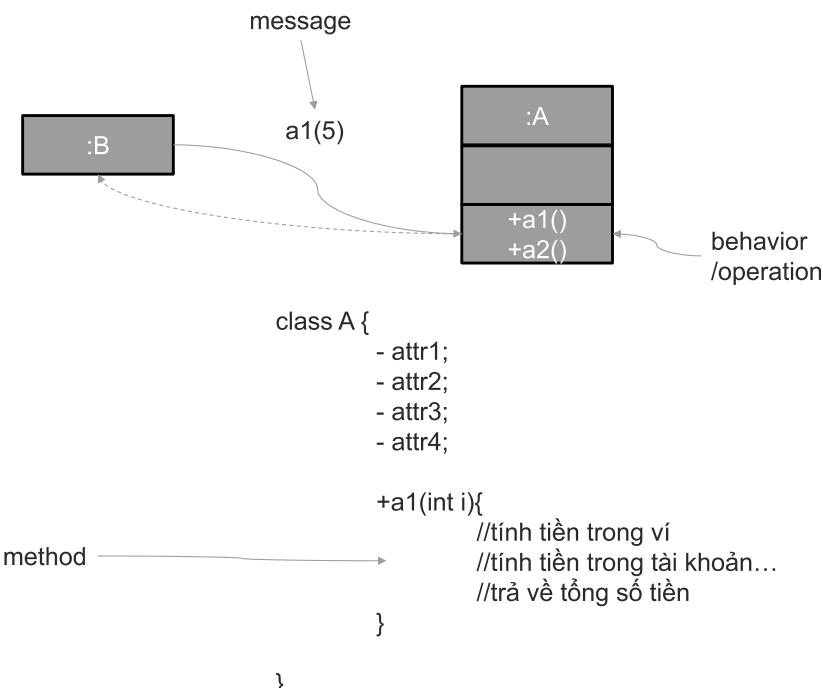
- The pieces of data that an instance of the class holds

- **Operations/Messages**

- List of messages that instances can receive

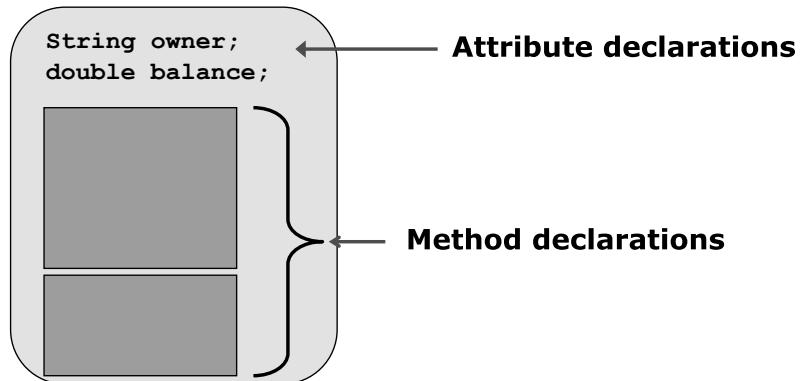
- **Methods**

- Implementations of the messages that each instance can receive



2.2. Class Building (2)

- Class encapsulating members
 - Attributes/Fields
 - Methods



Class Building in Java

- Classes are grouped into a package
 - Package is composed of a set of classes that have some logic relation between them,
 - Package is considered as a directory, a place to organize classes in order to locate them easily.
- Example:
 - Some packages already available in Java: `java.lang`, `javax.swing`, `java.io`...
 - Packages can be manually defined by users
 - Separated by “.”
 - Convention for naming package
 - Example: `package oolt.hedspi;`

a. Class declaration

- Declaration syntax:

```
package packagename;
access_modifier class ClassName{
    // Class body
}
```

BankAccount
- owner: String
- balance: double
+ debit(double): boolean
+ credit(double)

- **access_modifier:**

- **public:** Class can be accessed from anywhere, including outside its package.
- **private:** Class can only be accessed from inside the class
- **None (default):** Class can be accessed from inside its package

=> Class declaration for BankAccount class?

b. Member declaration of class

- Class members have access definition similarly to the class.

	public	None	private
Same class			
Same package			
Different package			

b. Member declaration of class

- Class members have access definition similarly to the class.

	public	None	private
Same class	Yes	Yes	Yes
Same package	Yes	Yes	No
Different package	Yes	No	No

Attribute

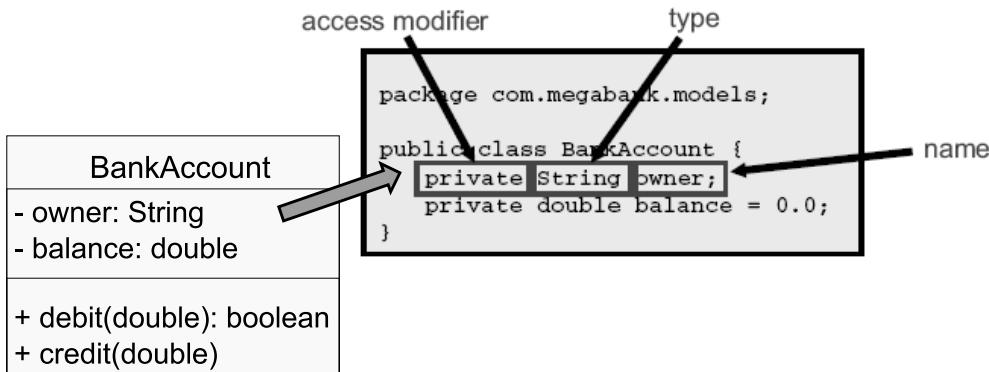
- Attributes have to be declared inside the class
- An object has its own copy of attributes
 - The values of an attribute of different objects are different.

Student
- name
- address
- studentID
- dateOfBirth



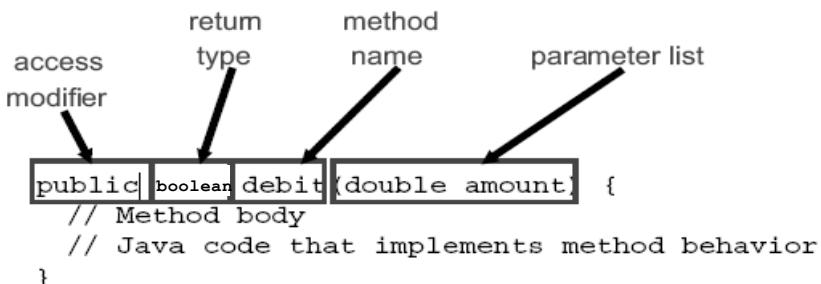
Attribute

- Attribute can be initialized while declaring
 - The default value will be used if not initialized.



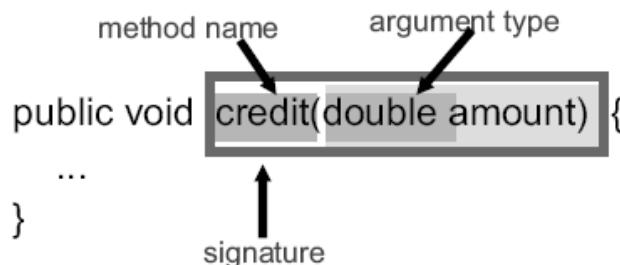
Method

- Define how an object responses to a request
- Method specifies the operations of a class
- Any method must belong to a class



* Method signature

- A method has its own signature including:
 - Method name
 - Number of parameters and their types



* Type of returned data

- When a method returns at least a value or an object, there must be a “return” command to return control to the caller object (object that is calling the method).
- If method does not return any value (void), there is no need for the “return” command
- There might be many “return”s in a method; the first one that is reached will be executed.

Class Building Example

- Example of a private field
 - Only this class can access the field

```
balance private double balance;
```

- Example of a public accessor method
 - Other classes can ask what the balance is

```
public double getBalance() {
    return balance;
}
```

- Other classes can change the balance only by calling deposit or withdraw methods

BankAccount	
- owner: String	
- balance: double	
+ debit(double): boolean	
+ credit(double)	

c. Constant member (Java)

- An attribute/method can not be changed its value during the execution.
- Declaration syntax:

```
access_modifier final data_type
CONSTANT_NAME = value;
```

- Example:

```
final double PI = 3.141592653589793;
public final int VAL_THREE = 39;
private final int[] A = { 1, 2, 3, 4, 5, 6 };
```

```

package com.megabank.models;
public class BankAccount {
    private String owner;
    private double balance;

    public boolean debit(double amount) {
        if (amount >= balance)
            return false;
        else {
            balance -= amount; return true;
        }
    }

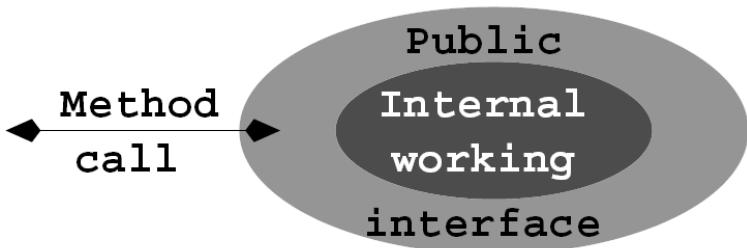
    public void credit(double amount) {
        //check amount . . .
        balance += amount;
    }
}

```

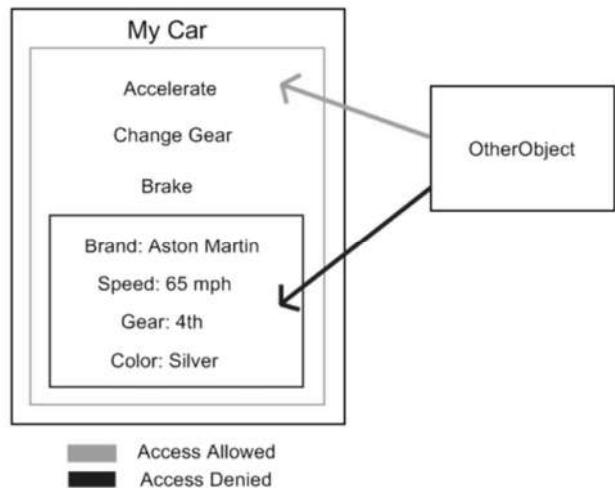
BankAccount
- owner: String
- balance: double
+ debit(double): boolean
+ credit(double)

2.3. Data hiding

- Data is hidden inside the class and can only be accessed and modified from the methods
 - Avoid illegal modification



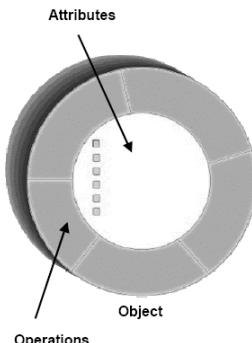
Example – Data hiding



Encapsulation with Java

Data hiding mechanism

- Data member
 - Can only be accessed from methods in the class
 - Access permission is **private** in order to protect data
- Other objects that want to access to the private data must perform via public functions



BankAccount
- owner: String
- balance: double
+ debit(double): boolean
+ credit(double)

Data hiding mechanism (2)

- Because data is private → Normally a class provides services to access and modify values of the data
 - Accessor (getter): return the current value of an attribute
 - Mutator (setter): modify value of an attribute
 - Usually getX and setX, where x is attribute name

```
package com.megabank.models;

public class BankAccount {
    private String owner;
    private double balance = 0.0;
}
```

```
public String getOwner() {
    return owner;
}
```

Get Method (Query)

- The Get methods (query method, accessor) are used to get values of data member of an object
- There are several query types:
 - Simple query ("what is the value of x?")
 - Conditional query ("is x greater than 10?")
 - Complex query ("what is the sum of x and y?")
- An important characteristic of getting method is that it should not modify the current state of the object
 - Do not modify the value of any data member

```

public class Time {
    private int hour;
    private int minute;
    private int second;

    public Time () {
        setTime(0, 0, 0);
    }

    public void setHour (int h) { hour = ( ( h >= 0 && h < 24 ) ? h : 0 ); }

    public void setMinute (int m) { minute = ( ( m >= 0 && m < 60 ) ? m : 0 ); }

    public void setSecond (int s) { second = ( ( s >= 0 && s < 60 ) ? s : 0 ); }

    public void setTime (int h, int m, int s) {
        setHour(h);
        setMinute(m);
        setSecond(s);
    }

    public int getHour () { return hour; }

    public int getMinute () { return minute; }

    public int getSecond () { return second; }
}

```

restricted access: private members are not externally accessible; but we need to know and modify their values

set methods: public methods that allow clients to modify private data; also known as mutators

get methods: public methods that allow clients to read private data; also known as accessors

Outline

1. Abstraction
2. Encapsulation and Class Building
3. Object Creation and Communication

3.1. Data initialization

- Data need to be initialized before being used
 - Initialization error is one of the most common ones
- For simple/basic data type, use operator =
- For object → Need to use constructor method

Student
- name
- address
- studentID
- dateOfBirth



Construction and destruction of object

- An existing and operating object is allocated some memory by OS in order to store its data values.
- When creating an object, OS will assign initialization values to its attributes
 - Must be done automatically before any developers' operations that are done on the object
 - Using construction function/method
- In contrast, while finishing, we have to release all the memory allocated to objects.
 - Java: JVM
 - C++: destructor

3.2. Constructor method

- Is a particular method that is automatically called when creating an object
- Main goal: Initializing attributes of objects

Student
- name
- address
- studentID
- dateOfBirth



3.2. Constructor method(2)

- Every class must have at least one constructor
 - To create a new representation of the class
 - Constructor name is the same as the class name
 - Constructor does not have return data type
- For example:

```
public BankAccount(String o, double b) {  
    owner = o;  
    balance = b;  
}
```

3.2. Constructor method (3)

- Constructor can use access attributes
 - **public**
 - **private**
 - None (default – can be used in package)
- A constructor can not use the keywords **abstract**, **static**, **final**, **native**, **synchronized**.
- Constructors can not be considered as *class members*.

3.2. Constructor method (4)

- Default constructor

- Is a constructor **without parameters**

```
public BankAccount() {  
    owner = "noname";  
    balance = 100000;  
}
```

- If we do not write any constructor in a class

- New JVM provides a default constructor

- The default constructor provided by JVM has the same access attributes as its class

- A class should have a default constructor

3.3. Object declaration and initialization

- An object is created and instantiated from a class.
- Objects have to be declared with **Types of objects** before being used:
 - Object type is object class
 - For example:
 - **String strName;**
 - **BankAccount acc;**

3.3. Object declaration and initialization (2)

- Objects must be initialized before being used
 - Use the operator = to assign
 - Use the keyword **new** for constructor to initialize objects:
 - Keyword **new** is used to create a new object
 - Automatically call the corresponding constructor
 - The default initialization of an object is **null**
- An object is manipulated through its *reference (~ pointer)*.
- For example:

```
BankAccount acc1;  
acc1 = new BankAccount();
```

3.3. Object declaration and initialization (3)

- We can combine the declaration and the initialization of objects
 - Syntax:

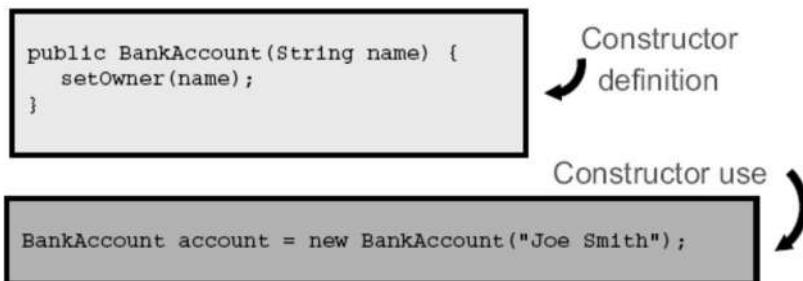
```
ClassName object_name = new  
                    Constructor(parameters);
```

- For example:

```
BankAccount account = new BankAccount();
```

3.3. Object declaration and initialization (4)

- Objects have
 - Identity: The object reference or variable name
 - State: The current value of all fields
 - Behavior: Methods
- Constructor does not have **return value**, but when being used with the keyword **new**, it returns a reference pointing to the new object.



3.3. Object declaration and initialization (5)

- Array of objects is declared similarly to the array of primitive data
- Array of objects is initialized with the value **null**.
- For example:

```
Employee emp1 = new Employee(123456);  
Employee emp2;  
emp2 = emp1;  
Department dept[] = new Department[100];  
Test[] t = {new Test(1), new Test(2)};
```

Example 1

```
public class BankAccount{  
    private String owner;  
    private double balance;  
}  
public class Test{  
    public static void main(String args[]){  
        BankAccount acc1 = new BankAccount();  
    }  
}
```

→ Default constructor provided by Java.

Example 2

```
public class BankAccount{  
    private String owner;  
    private double balance;  
    public BankAccount(){  
        owner = "noname";  
    }  
}  
public class Test{  
    public static void main(String args[]){  
        BankAccount acc1 = new BankAccount();  
    }  
}
```

→ Default constructor written by developers.

Example 3

```
public class BankAccount {  
    private String owner;  
    private double balance;  
    public BankAccount(String name){  
        setOwner(name);  
    }  
    public void setOwner(String o){  
        owner = o;  
    }  
}  
public class Test{  
    public static void main(String args[]){  
        BankAccount account1 = new BankAccount();  
        BankAccount account2 = new BankAccount("Hoang");  
    }  
}
```

The constructor BankAccount() is undefined

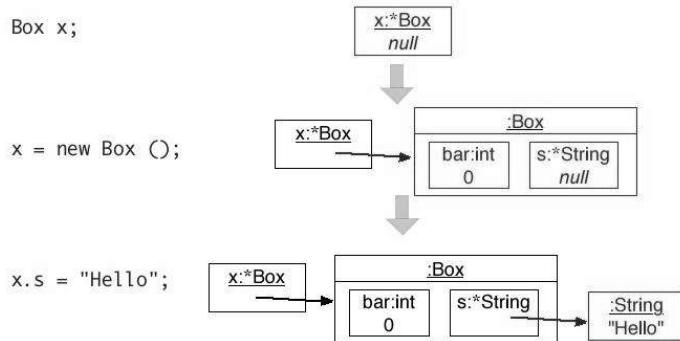
//Error

Objects in C++ and Java

- C++: objects in a class are created at the declaration:
 - Point p1;
- Java: Declaration of an object creates only a reference that will refer to the real object when **new** operation is used:
 - Box x;
 - x = new Box();
 - Objects are dynamically allocated in heap memory

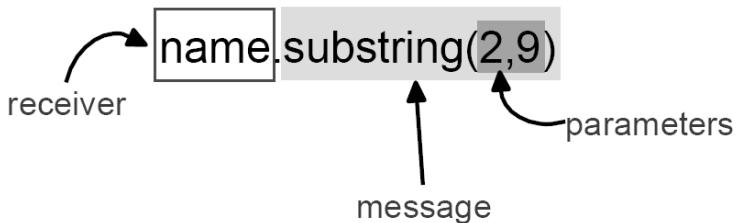
Object in Java

```
class Box  
{  
    int bar;  
    String s;  
}
```



3.4. Object usage

- Object provides more complex operations than primitive data types.
- Objects responds to messages
 - Operator `".."` is used to send a message to an object



3.4. Object usage (2)

- To call a member (data or attribute) of a class or of an object, we use the operator “.”
 - If we call method right in the class, the operator “.” is not necessary.

```
BankAccount account = new BankAccount();
account.setOwner("Smith");
account.credit(1000.0);
System.out.println(account.getBalance());
```

BankAccount methods

```
public void credit(double amount) {  
    setBalance(getBalance() + amount);  
}
```

```
public class BankAccount{  
    private String owner;  
    private double balance;  
    public BankAccount(String name){ setOwner(name); }  
    public void setOwner(String o){ owner = o; }  
    public String getOwner(){ return owner; }  
}  
public class Test{  
    public static void main(String args[]){  
        BankAccount acc1 = new BankAccount("");  
        BankAccount acc2 = new BankAccount("Hong");  
        acc1.setOwner("Hoa");  
        System.out.println(acc1.getOwner()  
                           + " " + acc2.getOwner());  
    }  
}
```

Example

```
// Create object and reference in one statement  
// Supply valued to initialize fields  
BankAccount ba = new BankAccount("A12345");  
BankAccount savingAccount = new BankAccount(2000000.0);  
  
// withdraw VNĐ5000.00 from an account  
ba.deposit(5000.0);  
// withdraw all the money in the account  
ba.withdraw(ba.getBalance());  
  
// deposit the amount by balance of saving account  
ba.deposit(savingAccount.getBalance());
```

Self-reference – this

- Allows to access to the current object of class.
- Is important when function/method is operating on two or many objects.
- Removes the mis-understanding between a local variable, parameters and data attributes of class.
- Is not used in static code block

```
public class BankAccount{  
    private String owner;  
    private double balance;  
    public BankAccount() { }  
    public void setOwner(String owner){  
        this.owner = owner;  
    }  
    public String getOwner(){ return owner; }  
}  
  
public class Test{  
    public static void main(String args[]){  
        BankAccount acc1 = new BankAccount();  
        BankAccount acc2 = new BankAccount();  
        acc1.setOwner("Hoa");  
        acc2.setOwner("Hong");  
        System.out.println(acc1.getOwner() + " " +  
                           acc2.getOwner());  
    }  
}
```

@NGUYỄN Thị Thu Trang, trangntt@soict.hust.edu.vn

OBJECT-ORIENTED LANGUAGE AND THEORY

4. SOME TECHNIQUES IN CLASS BUILDING

Nguyen Thi Thu Trang
trangntt@soict.hust.edu.vn



Goals

- Understand notions, roles and techniques for overloading methods and overloading constructors
- Object member, class member
- How to pass arguments of functions

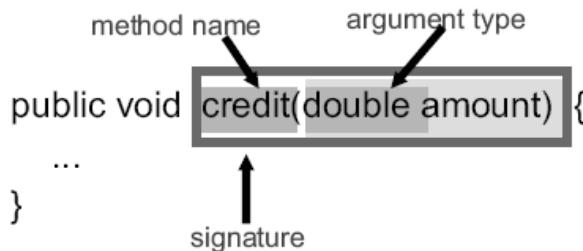
Outline



1. Method overloading
2. Classifier and constant members
3. Passing arguments to methods

Method recalls

- Each method has its own signature
- A method signature is composed of:
 - Method's name
 - Number of arguments and their types



1.1. Method overloading

- **Method Overloading:** Methods in a class might have the same name but different signatures:
 - **Numbers of arguments** are different
 - If the numbers of arguments are the same, **types of arguments** must be **different**
- **Advantages:**
 - The same name describes the **same task**
 - Is easier for developers because they don't have to remember **too many method names**. They remember only one with the appropriate arguments.

Method overloading – Example 1

- Method `println()` in `System.out.println()` has 10 declarations with different arguments: `boolean`, `char[]`, `char`, `double`, `float`, `int`, `long`, `Object`, `String`, and one without argument.
- Do not need to use different names (for example "`printString`" or "`printDouble`") for each data type to be displayed.

Method overloading – Example 2

```
class MyDate {  
    int year, month, day;  
    public boolean setMonth(int m) { ... }  
    public boolean setMonth(String s) { ... }  
}  
public class Test{  
    public static void main(String args[]){  
        MyDate d = new MyDate();  
        d.setMonth(9);  
        d.setMonth("September");  
    }  
}
```

Method overloading – More info.

- Methods are considered as **overloading** only if they belong to the **same class**
- Only apply this technique on methods describing the **same kind of task**; do not abuse
- When compiling, compilers rely on number or types of arguments to decide which **appropriate method** to call.
 - If there is no method or more than one method to call, an error will be reported.

Discussion

- Given a following method:
 0. `public double test(String a, int b)`
- Let select overloading methods of the given method 0 from the list below:
 1. `void test(String b, int a)`
 2. `public double test(String a)`
 3. `private int test(int b, String a)`
 4. `private int test(String a, int b)`
 5. `double test(double a, int b)`
 6. `double test(int b)`
 7. `public double test(String a, long b)`

Discussion

```
void prt(String s) { System.out.println(s); }
void f1(char x) { prt("f1(char)"); }
void f1(byte x) { prt("f1(byte)"); }
void f1(short x) { prt("f1(short)"); }
void f1(int x) { prt("f1(int)"); }
void f1(long x) { prt("f1(long)"); }
void f1(float x) { prt("f1(float)"); }
void f1(double x) { prt("f1(double)"); }
```

- What will happen if we do as follows:

- `f1(5);`
- `char x='a'; f1(x);`
- `byte y=0; f1(y);`
- `float z = 0; f1(z);...`

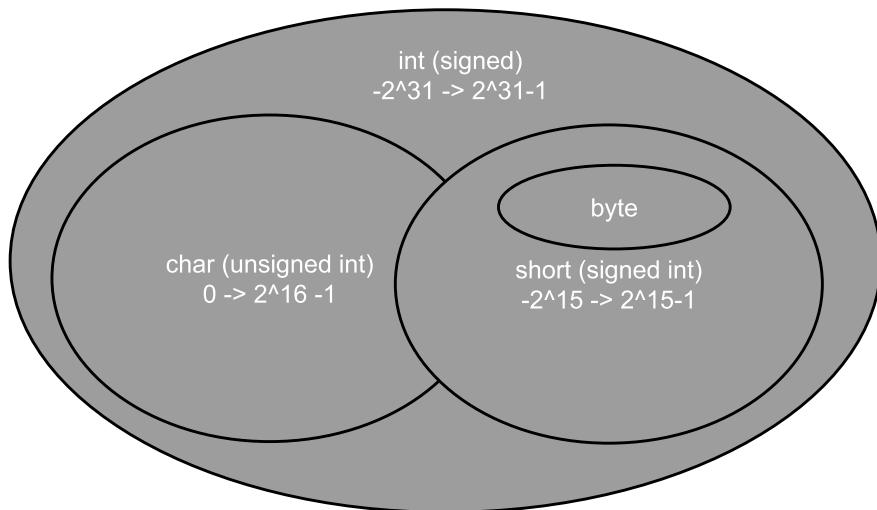
Discussion

```
void prt(String s) { System.out.println(s); }
void f2(short x) { prt("f3(short)"); } => 2 b
void f2(int x) { prt("f3(int)"); } => 4 b
void f2(long x) { prt("f5(long)"); } => 8 b
void f2(float x) { prt("f5(float)"); }
```

- What will happen if we do as follows:

- `f2(5);`
- `char x='a'; f2(x); => 2 b`
- `byte y=0; f2(y);`
- `float z = 0; f2(z);`

- What will happen if we call `f2(5.5)?`



1.2. Constructor overloading

- In different contexts => create objects in different ways
→ Any number of constructors with different parameters (following constructor overloading principles)
- Constructors are commonly overloaded to allow for different ways of initializing instances

```
BankAccount new_account =
    new BankAccount();

BankAccount known_account =
    new BankAccount(account_number);

BankAccount named_account =
    new BankAccount("My Checking Account");
```

Example

```
public class BankAccount{
    private String owner;
    private double balance;
    public BankAccount(){owner = "noname";}
    public BankAccount(String o, double b){
        owner = o; balance = b;
    }
}
public class Test{
    public static void main(String args[]){
        BankAccount acc1 = new BankAccount();
        BankAccount acc2 =
            new BankAccount("Thuy", 100);
    }
}
```

this keyword

- “this” refers to the **current object**, it is used **inside the class** of the object that it refers to.
- It uses attributes or methods of object through “.” operator, for example:

```
public class BankAccount{
    private String owner;
    public void setOwner(String owner){
        this.owner = owner;
    }
    public BankAccount() { this.setOwner("noname"); }
    ...
}
```

- Call another constructor of the class:

- `this(parameters); //first statement in another constructor`

this keyword

In a constructor, the keyword `this` is used to refer to other constructors in the same class

```

...
public BankAccount(String name) {
    super();
    owner = name;
}

public BankAccount() {
    this("TestName");
}

public BankAccount(String name, double initialBalance) {
    this(name);
    setBalance(initialBalance);
}
...

```

The diagram shows three constructor definitions for a class named `BankAccount`. A curved arrow originates from the `this("TestName");` call in the first constructor and points to the `this(name);` call in the second constructor. Another curved arrow originates from the `this(name);` call in the second constructor and points to the `this(name);` call in the third constructor.

- Example

```

public class Ship {
    private double x=0.0, y=0.0
    private double speed=1.0, direction=0.0;
    public String name;

    public Ship(String name) {
        this.name = name;
    }
    public Ship(String name, double x, double y) {
        this(name); this.x = x; this.y = y;
    }
    public Ship(String name, double x, double y,
               double speed, double direction) {
        this(name, x, y);
        this.speed = speed;
        this.direction = direction;
    }
    //to be continued...
}

```

```
//(cont.)  
private double degreeToRadian(double degrees) {  
    return(degrees * Math.PI / 180.0);  
}  
public void move() {  
    move(1);  
}  
public void move(int steps) {  
    double angle = degreesToRadians(direction);  
    x = x + (double)steps*speed*Math.cos(angle);  
    y = y + (double)steps*speed*Math.sin(angle);  
}  
public void printLocation() {  
    System.out.println(name + " is at ("  
                        + x + "," + y + ") .");  
}  
} //end of Ship class
```

Outline

1. Method overloading
2. Classifier and constant members
3. Passing arguments to methods

2.1. Constant members

- An attribute/method that can not change its values/content during the usage.
- Declaration syntax:

```
access_modifier final data_type
CONSTANT_VARIABLE = value;
```

- For example:

```
final double PI = 3.141592653589793;
public final int VAL_THREE = 39;
private final int[] A = { 1, 2, 3, 4, 5, 6 };
```

2.1. Constant members (2)

- Typically, constants associated with a class are declared as **static final** fields for easy access
 - A common convention is to use only uppercase letters in their names

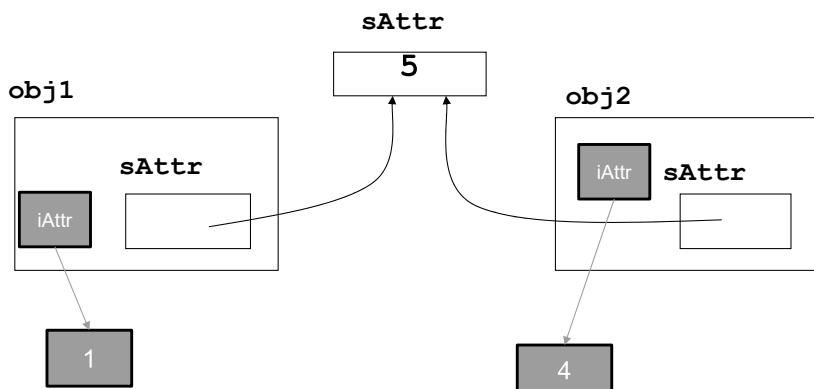
```
public class MyDate {
    public static final long SECONDS_PER_YEAR =
        31536000;
    ...
    ...
    long years = MyDate.getMillisSinceEpoch() /
        (1000*MyDate.SECONDS_PER_YEAR);
```

2.2. Classifier members

- Members may belong to either of the following:
 - The whole class (class variables and methods, indicated by the keyword **static** in Java)
 - Individual objects (instance variables and methods)
- Static attributes and methods belong to the class
 - Changing a value in one object of that class changes the value for all of the objects
- Static methods and fields can be accessed without instantiating the class
 - Static methods and fields are declared using the static keyword

Static parts: are shared between all objects

- sAttr: static (class/classifier scope)
- iAttr: instance (object-instance scope)



Instance member vs. Classifier member

- Attributes/methods can only be accessed via objects
 - Each object has its own copy of an object's attribute
 - **Values** of an attribute of different objects are different.
- Attributes/methods can be accessed through class
 - All objects have the same copy of class attributes
 - **Values** of a class attribute of different objects are the same.

Static members in Java

- Regular members are members of objects
- Class members are declared as **static**
- Syntax for declaring static member:
`access_modifier static data_type varName;`
- Example:

```
public class MyDate {
    public static long getMillisSinceEpoch() {
        ...
    }
    public String getMonth(){
        long ms = getMillisSinceEpoch();
        ...
    }
    long millis = MyDate.getMillisSinceEpoch();
```

```
MyDate date1 = new MyDate();
date1.getMonth(); date1.getMillisSinceEpoch();
```

Example: Class JOptionPane in javax.swing

- Attributes

Field Summary	
static int <u>CANCEL_OPTION</u>	Return value from class method if CANCEL is chosen
static int <u>CLOSED_OPTION</u>	Return value from class method if user closes window
static int <u>CANCEL_OPTION</u> or <u>NO_OPTION</u> .	
static int <u>DEFAULT_OPTION</u>	Type used for <code>showConfirmDialog</code> .
static int <u>ERROR_MESSAGE</u>	Used for error messages.
static int <u>WARNING_MESSAGE</u>	Used for warning messages.
static int <u>YES_NO_CANCEL_OPTION</u>	Type used for <code>showConfirmDialog</code> .
static int <u>YES_NO_OPTION</u>	Type used for <code>showConfirmDialog</code> .
static int <u>YES_OPTION</u>	Return value from class method if YES is chosen.

- Methods:

static void <u>showMessageDialog</u> (Component parentComponent, Object message)	Brings up an information-message dialog titled "Message".
static void <u>showMessageDialog</u> (Component parentComponent, Object message, String title, int messageType)	Brings up a dialog that displays a message using a default icon determined by the messageType parameter.
static void <u>showMessageDialog</u> (Component parentComponent, Object message, String title, int messageType, String messageText)	Brings up a dialog displaying a message specifying all parameters

Example – using static attributes and methods in class JOptionPane

```
JOptionPane.showMessageDialog(null, "Ban da thao tac loi", "Thong bao loi", JOptionPane.ERROR_MESSAGE);
```



```
JOptionPane.showConfirmDialog(null, "Ban co chac chan muon thoat?", "Hay lua chon", JOptionPane.YES_NO_OPTION);
```



Example – using static attributes and methods in class JOptionPane (2)

```
Object[] options = { "OK", "CANCEL" };  
JOptionPane.showOptionDialog(null,"Nhan OK de tiep tuc",  
    "Canh bao", JOptionPane.DEFAULT_OPTION,  
    JOptionPane.WARNING_MESSAGE,null,options,options[0]);
```



Static member (2)

- Modifying value of a **static** member in an object will modify the value of this member in all other objects of the class.
- **Static** methods can access only **static** attributes and can call **static** methods in the same class.

Example 1

```
class TestStatic{
    public static int iStatic;
    public int iNonStatic;
}

public class TestS {
    public static void main(String[] args) {
        TestStatic obj1 = new TestStatic();
        obj1.iStatic = 10; obj1.iNonStatic = 11;
        System.out.println(obj1.iStatic+","+obj1.iNonStatic);
        TestStatic obj2 = new TestStatic();
        System.out.println(obj2.iStatic+","+obj2.iNonStatic);
        obj2.iStatic = 12;
        System.out.println(obj1.iStatic+","+obj1.iNonStatic);
    }
}
```



```
10,11
10,0
12,11
```

Example 2

```
public class Demo {
    int i = 0;
    void increase(){ i++; }
    public static void main(String[] args) {
        increase();
        System.out.println("Gia tri cua i la" + i);
    }
}
```

non-static method increase() cannot be referenced from a static context
 non-static variable i cannot be referenced from a static context

Java static methods – Example

```
class MyUtils {  
    . . .  
    //===== mean  
    public static double mean(int[] p) {  
        int sum = 0;  
        for (int i=0; i<p.length; i++) {  
            sum += p[i];  
        }  
        return ((double)sum) / p.length;  
    }  
    . . .  
}  
...  
  
// Calling a static method from outside of a class  
double avgAtt = MyUtils.mean(attendance);
```

When static?

Outline

1. Method overloading
2. Classifier and constant members
3. Passing arguments to methods

3. Arguments passing to methods

- We can use any data types for arguments for methods or constructors
 - Primitive data types
 - References: array and object
- Example:

```
public Polygon polygonFrom(Point[] corners) {  
    // method body goes here  
}
```

3.1. Variable arguments

- An arbitrary number of arguments, called *varargs*
- Syntax in Java:
 - `methodName (data_type... parameterName)`
- Example
 - Declaration:

```
public PrintStream printf(String format,
                          Object... args)
```

- Usage:


```
System.out.printf ("%s: %d, %s\n",
                        name, idnum, address);
System.out.printf ("%s: %d, %s, %s, %s\n",
                        name, idnum, address, phone, email);
```

• Example

```
public Polygon polygonFrom(Point... corners) {
    int numberOfSides = corners.length;
    double squareOfSide1, lengthOfSide1;
    squareOfSide1 = (corners[1].x - corners[0].x)
                   * (corners[1].x - corners[0].x)
                   + (corners[1].y - corners[0].y)
                   * (corners[1].y - corners[0].y) ;
    lengthOfSide1 = Math.sqrt(squareOfSide1);
    //create & return a polygon connecting the Points
}
```

- **corners** is considered as an array
- You can pass an array or a sequence of arguments

3.2. Passing by values

- C++
 - Passing values, pointers
- Java
 - Passing values

39

Java: Pass-by-value for all types of data

- Java passes all arguments to a method in form of pass-by-value: Passing value/copy of the real argument
 - For arguments of value-based data types (primitive data types): passing value/copy of primitive data type argument
 - For argument of reference-based data types (array and object): passing value/copy of original reference.
- Modifying formal arguments does not effect the real arguments

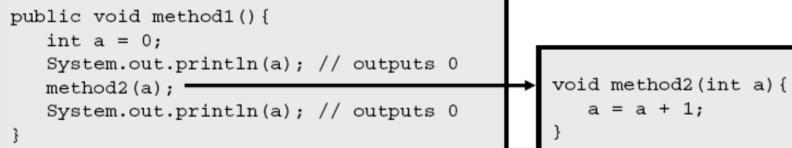
Discussion:

- What will happen if:
 - We modify the internal state of object parameters inside a method?
 - We modify the reference to an object?

41

a. With value-based date type

- Primitive values can not be changed when being passed as a parameter

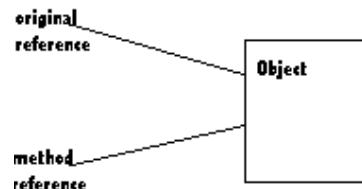


- Is this swap method correct?

```
public void swap(int var1, int var2) {
    int temp = var1;
    var1 = var2;
    var2 = temp;
}
```

b. With reference-based data type

- Pass the references by value, not the original reference or the object



- After being passed to a method, a object has at least two references

Passing parameters

```
public class ParameterModifier
{
    public void changeValues (int f1, Num f2, Num f3)
    {
        System.out.println ("Before changing the values:");
        System.out.println ("f1\tf2\tf3");
        System.out.println (f1 + "\t" + f2 + "\t" + f3 + "\n");

        f1 = 999;
        f2.setValue(888);
        f3 = new Num (777);

        System.out.println ("After changing the values:");
        System.out.println ("f1\tf2\tf3");
        System.out.println (f1 + "\t" + f2 + "\t" + f3 + "\n");
    }
}
```

Passing parameters

```
public class ParameterTester
{
    public static void main (String[] args)
    {
        ParameterModifier modifier = new ParameterModifier();

        int a1 = 111;
        Num a2 = new Num (222);
        Num a3 = new Num (333);

        System.out.println ("Before calling changeValues");
        System.out.println ("a1\ta2\ta3");
        System.out.println (a1 + "\t" + a2 + "\t" + a3);

        modifier.changeValues (a1, a2, a3);

        System.out.println ("After calling changeValues");
        System.out.println ("a1\ta2\ta3");
        System.out.println (a1 + "\t" + a2 + "\t" + a3);
    }
}
```

Before calling changeValues:
a1 a2 a3
111 222 333

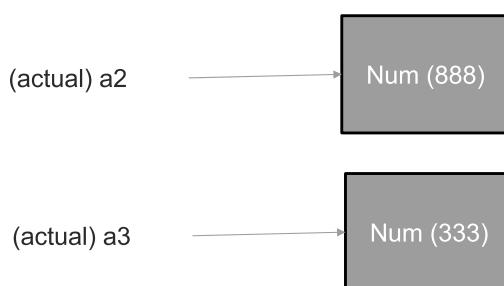
Before changing the values:
f1 f2 f3
111 222 333

After changing the values:
f1 f2 f3
999 888 777

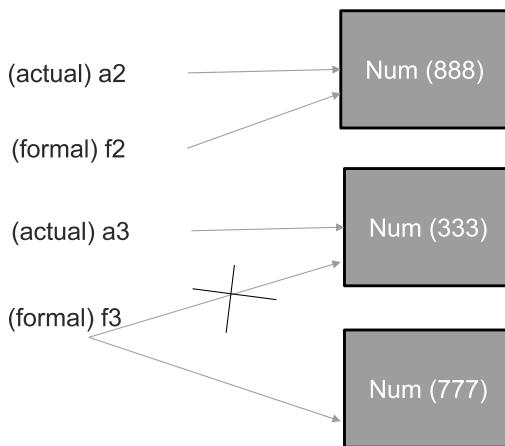
After calling changeValues:
a1 a2 a3
111 888 333

45

Inside the method changeValues()



Inside the method changeValues()



For example

```
public class Point {  
    private double x;  
    private double y;  
    public Point() { }  
    public Point(double x, double y) {  
        this.x = x; this.y = y;  
    }  
    public void setX(double x) { this.x = x; }  
    public void setY(double y) { this.y = y; }  
    public void printPoint() {  
        System.out.println("X: " + x + " Y: " + y);  
    }  
}
```

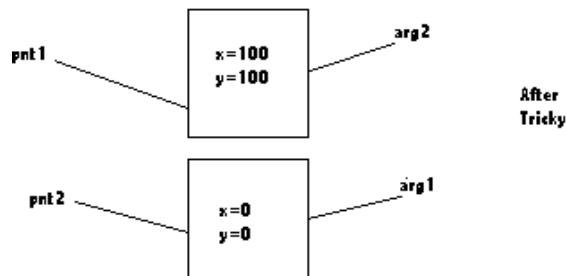
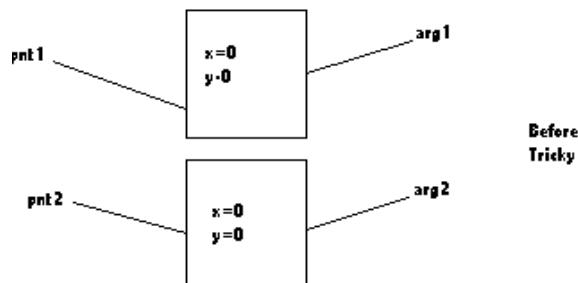
```

public class Test {
    public static void tricky(Point arg1, Point arg2) {
        arg1.setX(100); arg1.setY(100);
        Point temp = arg1;
        arg1 = arg2; arg2 = temp;
    }
    public static void main(String [] args) {
        Point pnt1 = new Point(0,0);
        Point pnt2 = new Point(0,0);
        pnt1.printPoint(); pnt2.printPoint();
        System.out.println(); tricky(pnt1, pnt2);
        pnt1.printPoint(); pnt2.printPoint();
    }
}

```

x: 0.0 y: 0.0
x: 0.0 y: 0.0
x: 100.0 y: 100.0
x: 0.0 y: 0.0
Press any key to continue . . .

- Only the method references are swap, not the original references



OBJECT-ORIENTED LANGUAGE AND THEORY

5. MEMORY MANAGEMENT AND CLASS ORGANIZATION

Nguyen Thi Thu Trang
trangntt@soict.hust.edu.vn



2

Outline

- 1. Memory management in Java
- 2. Class organization
- 3. Utility classes in Java

1. Memory management in Java

- Java does not use pointer, hence memory addresses can not be overwritten accidentally or intentionally.
- The allocation or re-allocation of memory, management of memory that is controlled by JVM, are completely transparent with developers.
- Developers do not need to care about the allocated memory in heap in order to free it later.

```
byte i;  
i = 4;
```

```
byte *j;  
  
j+2
```

3FE4

5

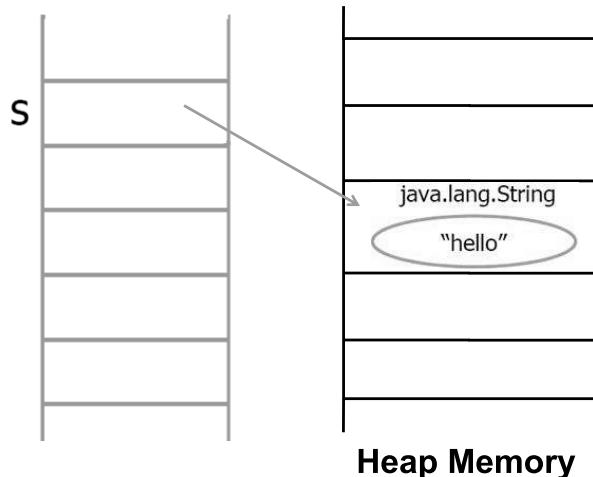
4



1.1. Heap memory

```
String s = new String("hello");
```

- Heap memory is used to write information created by **new** operator.

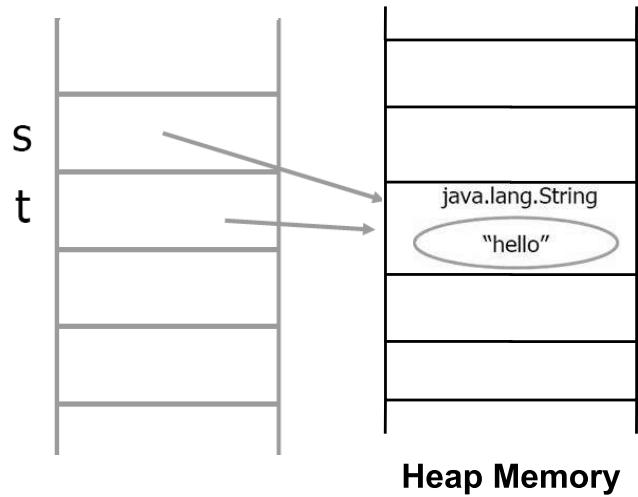


1.1. Heap memory (2)

```
String s = new String("hello");
```

```
String t = s;
```

- Heap memory is used to write information created by **new** operator.



1.2. Stack memory

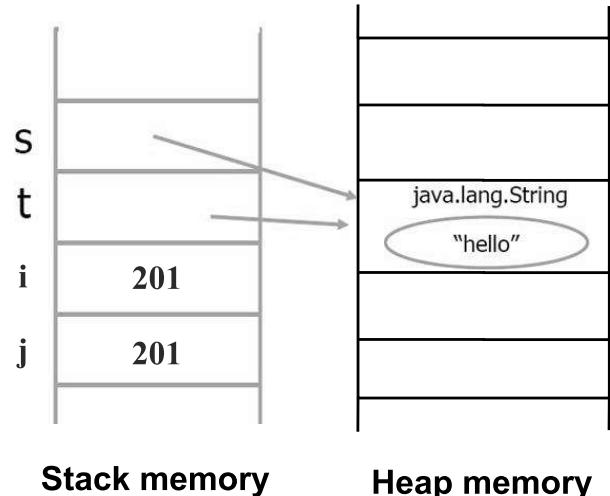
```
String s = new String("hello");
```

```
String t = s;
```

```
int i = 201;
```

```
int j = i;
```

- Local value in Stack memory is used as a reference pointer to Heap
- Value of primitive data is written directly in Stack



1.3. Garbage collector (gc)

- The garbage collector sweeps through the JVM's list of objects periodically and reclaims the resources held by unreferenced objects
- All objects that have no object references are eligible for garbage collection
 - References out of scope, objects to which you have assigned null, and so forth
- The JVM decides when the gc is run
 - Typically, the gc is run when memory is low
 - May not be run at all
 - Unpredictable timing

Working with the garbage collector

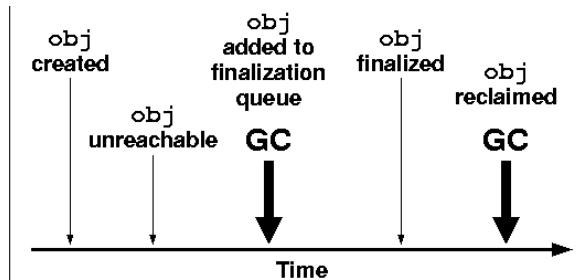
- You cannot prevent the garbage collector from running, but you can request it to run soon
 - `System.gc();`
 - This is only a request, not a guarantee
- The `finalize()` method of an object will be run immediately before garbage collection occurs
 - This method should only be used for special cases (e.g. cleaning up memory allocation from native calls) because of the unpredictability of the garbage collector
 - Things like open sockets, files, and so forth should be cleaned up during normal program flow before the object is dereferenced

Java destructors?

- Java does not have the concept of a destructor for objects that are no longer in use
- Deallocation of memory is done automatically by the JVM through the `finalize()` method
 - A background process called the garbage collector reclaims the memory of unreferenced objects
 - The association between an object and an object reference is severed by assigning another value to the object reference, for example:
 - `objectReference = null;`
 - An object with no references is a candidate for deallocation during garbage collection

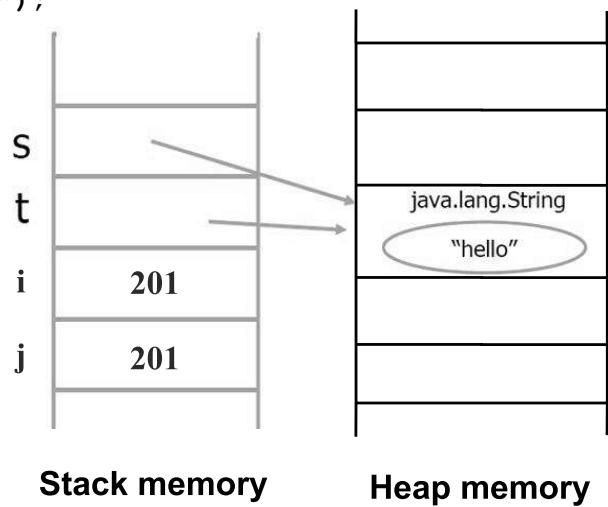
finalize() method

- Any class has method finalize() – that is executed right after the garbage collection process takes place (considered as destructor in Java despite not)
- Override this method in some special cases in order to “self-clean” used resources when objects are freed by gc
 - E.g. pack socket, file,... that should be handled in the main thread before the objects are disconnected from reference.



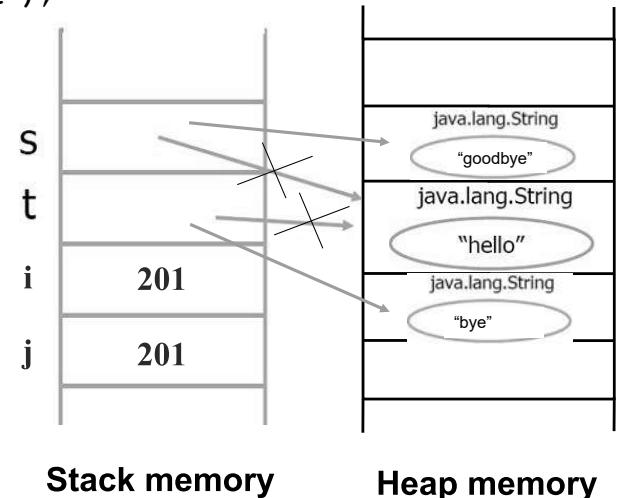
```
String s = new String("hello");
String t = s;
int i = 201;
int j = i;
s = new String("goodbye");
t = new String("bye");
```

12



```
String s = new String("hello");
String t = s;
int i = 201;
int j = i;
s = new String("goodbye");
t = new String("bye");
```

13



14

Memory Management in Java (Method and variables)



1.4. Object comparison

- **Primitive data types:** == checks whether their values are the equal

```
int a = 1;
int b = 1;
if (a==b) ... // true
```

- **Objects:** == checks whether two objects are unique ~ whether they refer to the same object

```
Employee a = new Employee(1);
Employee b = a;
if (a==b) ... // true
```

```
Employee a = new Employee(1);
Employee b = new Employee(1);
if (a==b) ... // false
```

equals() method

- For primitive data types → does not exist.
- For objects: every object has this method
 - Compares values of objects

```
public class Equivalence {
    public static void main(String[] args) {
        Integer n1 = new Integer(47);
        Integer n2 = new Integer(47);
        System.out.println(n1 == n2);
        System.out.println(n1.equals(n2));
    }
}
```

```
false
true
```

equals() method of your class

```
class Value {  
    int i;  
    public Value(int i) { this.i = i; }  
}  
public class EqualsMethod2 {  
    public static void main(String[] args) {  
        Value v1 = new Value(10);  
        Value v2 = new Value(10);  
        System.out.println(v1.equals(v2));  
    }  
}
```

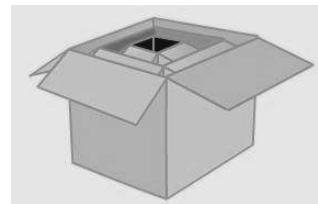


Outline

1. Memory management in Java
2. Class organization
3. Utility classes in Java

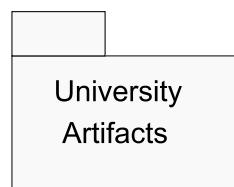
Class organization with Package

- Package is as a folder that helps:
 - Organize and locate easily the classes and use classes in a appropriate manner
 - Avoid conflict in naming classes
 - Different packages can contains classes with same name
 - Protect classes, data and methods in a larger area compared to relation between classes
- A package can also contain another package
 - “com” package contains “google” package
 - com.google



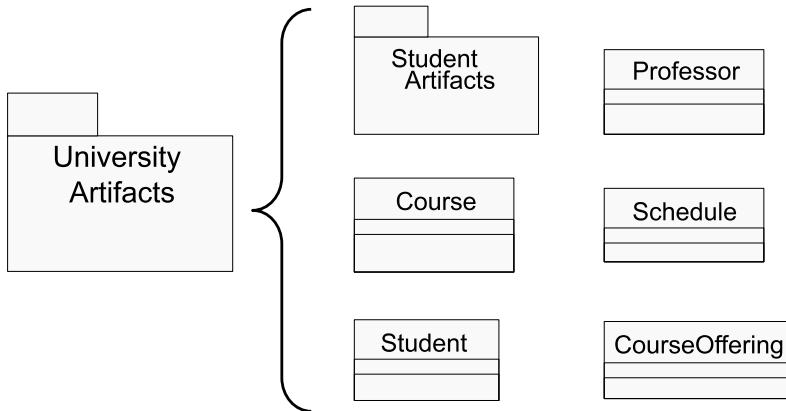
Package in UML

- A general purpose mechanism for organizing elements into groups.
- A model element that can contain other model elements.
- A package can be used:
 - To organize the model under development
 - As a unit of configuration management



A Package Can Contain Classes

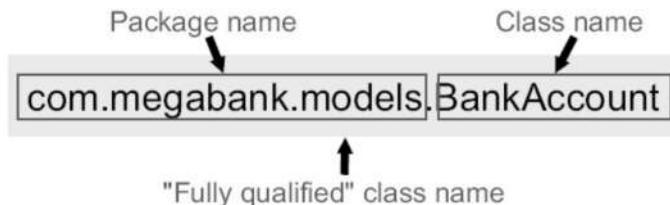
- The package, University Artifacts, contains one package and five classes.



22

Fully qualified class name

- A fullname of a class includes package name and class name:



- package oolt.hedspi;
- class AS1{
 - int as11;
 - void as1_method(){
 - IS1 as1 = new IS1();
 - is1.is1_method();
 - }
- }

- package oolt.hedspi;
- class IS1{
 - void is1_method(){}
 - }

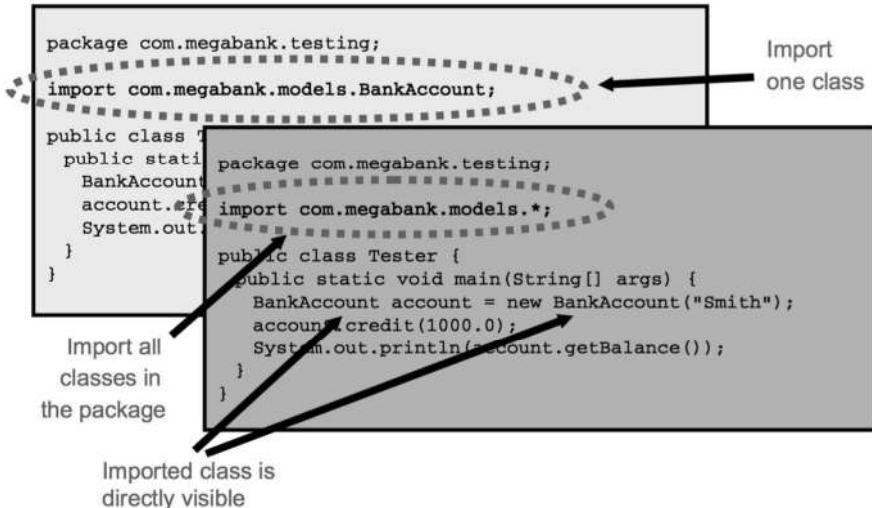
2.1. References between classes

- In the same package: use class name
- In different packages: must provide the full-name of class defined in other packages.
- Example:

```
package oolt.hedspi;
public class HelloNameDialog{
    public static void main(String[] args){
        String result;
        result = javax.swing.JOptionPane.
            showInputDialog("Please enter your name:");
        javax.swing.JOptionPane.
            showMessageDialog(null, "Hi " + result + "!");
    }
}
```

Using import command

To import packages or classes to make other classes directly visible to your class



More example

```

package oolt.hedspi;
public class HelloNameDialog{
    public static void main(String[] args){
        System.out.print("Hello world!");
    }
}

```

2.2. Packages in Java

- `java.applet`
- `java.awt`
- `java.beans`
- `java.io`
- `java.lang`
- `java.math`
- `java.net`
- `java.nio`
- `java.rmi`
- `java.security`
- `java.sql`
- `java.text`
- `java.util`
- `javax.accessibility`
- `javax.crypto`
- `javax.imageio`
- `javax.naming`
- `javax.net`
- `javax.print`
- `javax.rmi`
- `javax.security`
- `javax.sound`
- `javax.sql`
- `javax.swing`
- `javax.transaction`
- `javax.xml`
- `org.apache.commons`
- `org.ietf.jgss`
- `org.omg.CORBA`
- `org.omg.IOP`
- `org.omg.Messaging`
- `org.omg.PortableInterceptor`
- `org.omg.PortableServer`
- `org.omg.SendingContext`
- `org.omg.stub.java.rmi`
- `org.w3c.dom`
- `org.xml`

Basic packages in Java

• `java.lang`

- Provides classes that are fundamental to the design of the Java programming language
- Includes wrapper classes, String and StringBuffer, Object, and so on
- Imported implicitly into all classes

• `java.util`

- Contains the collections framework, event model, date and time facilities, internationalization, and miscellaneous utility classes

• `java.io`

- Provides for system input and output through data streams, serialization and the file system

Basic packages in Java

- **java.math**

- Provides classes for performing arbitrary-precision integer arithmetic and arbitrary-precision decimal arithmetic

- **java.sql**

- Provides the API for accessing and processing data stored in a data source (usually a relational database)

- **java.text**

- Provides classes and interfaces for handling text, dates, numbers, and messages in a manner independent of natural languages

- **javax.swing**

- Provides classes and interfaces to create graphics

Sample package: java.lang

- **Basic Entities**

- Class, Object, Package, System

- **Wrappers**

- Number, Boolean, Byte, Character, Double, Float, Integer, Long, Short, Void

- **Character and String Manipulation**

- Character.Subset, String, StringBuffer, Character.UnicodeBlock

- **Math Functions**

- Math, StrictMath

- **Runtime Model**

- Process, Runtime, Thread, ThreadGroup, ThreadLocal, InheritableThreadLocal, RuntimePermission

- **JVM**

- ClassLoader, Compiler, SecurityManager

- **Exception Handling**

- StackTraceElement, Throwable

- Also contains Interfaces, Exceptions and Errors

Outline

1. Memory management in Java
2. Class organization
3. Utility classes in Java

3.1. Wrapper class

- Primitives have no associated methods; there is no behavior associated with primitive data types
- Each primitive data type has a corresponding class, called a wrapper
 - Each wrapper object simply stores a single primitive variable and offers methods with which to process it
 - Wrapper classes are included as part of the base Java API

Wrapper classes

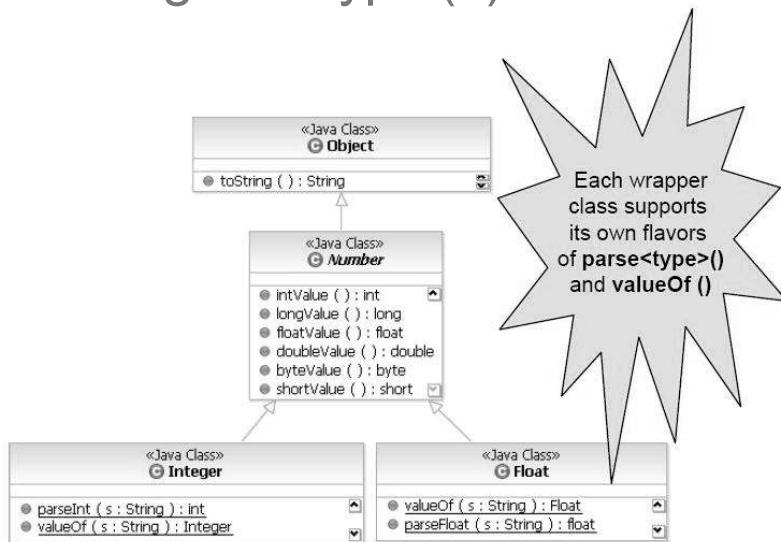
Primitive Type	Wrapper Class
boolean	Boolean
byte	Byte
char	Character
double	Double
float	Float
int	Integer
long	Long
short	Short

Converting data type

- Use `toString()` to convert number values to string.
 - Use `<type>Value()` to convert an object of a wrapper class to the corresponding primitive value
- ```
Float objF = new Float("4.67");
float f = objF.floatValue(); // f=4.67F
int i = objF.intValue(); //i=4
```
- Use `parse<type>()` and `valueOf()` to convert string to number values.

```
int i = Integer.parseInt("123"); //i=123
double d = Double.parseDouble("1.5"); // d=1.5
Double objF2 = Double.valueOf("-36.12");
long l = objF2.longValue(); // l=-36L
```

# Converting data type (2)



# Constants

- **Boolean**
  - Boolean FALSE
  - Boolean TRUE
- **Byte**
  - byte MIN\_VALUE
  - byte MAX\_VALUE
- **Character**
  - int MAX\_RADIX
  - char MAX\_VALUE
  - int MIN\_RADIX
  - char MIN\_VALUE
  - Unicode classification constants
- **Double**
  - double MAX\_VALUE
  - double MIN\_VALUE
  - double NaN
  - double NEGATIVE\_INFINITY
  - double POSITIVE\_INFINITY
- **Float**
  - float MAX\_VALUE
  - float MIN\_VALUE
  - float NaN
  - float NEGATIVE\_INFINITY
  - float POSITIVE\_INFINITY
- **Integer**
  - int MIN\_VALUE
  - int MAX\_VALUE
- **Long**
  - long MIN\_VALUE
  - long MAX\_VALUE
- **Short**
  - short MIN\_VALUE
  - short MAX\_VALUE

## Example

```

double d = (new Integer(Integer.MAX_VALUE)).

 doubleValue();

System.out.println(d); // 2.147483647E9

String input = "test 1-2-3";

int output = 0;

for (int index = 0; index < input.length(); index++)

{

 char c = input.charAt(index);

 if (Character.isDigit(c))

 output = output * 10 + Character.digit(c, 10);

}

System.out.println(output); // 123

```

## 3.2. String

- The String type is a class, not a primitive data type
- A String literal is made up of any number of characters between double quotes:

```
String a = "A String";
```

```
String b = "";
```

- A String object can be initialized in other ways:

```
String c = new String();
```

```
String d = new String("Another String");
```

```
String e = String.valueOf(1.23); // "1.23"
```

```
String f = null;
```

## a. String concatenation

- The + operator concatenates Strings:

```
String a = "This" + " is a " + "String";
//a = "This is a String"
```

*There are more efficient ways to concatenate Strings  
(this will be discussed later)*

- Primitive data types used in in a call to println() are automatically converted to String

```
System.out.println("answer = " + 1 + 2 + 3);
System.out.println("answer = " + (1+2+3));
```

→ Do two above commands print out the same output?

## b. Methods of String

Strings are objects; objects respond to messages

- ✓ Use the dot (.) operator to send a message
- ✓ String is a class, with methods

```
String name = "Joe Smith";
name.toLowerCase(); // "joe smith"
name.toUpperCase(); // "JOE SMITH"
"Joe Smith ".trim(); // "Joe Smith"
"Joe Smith".indexOf('e'); // 2
"Joe Smith".length(); // 9
"Joe Smith".charAt(5); // 'm'
"Joe Smith".substring(5); // "mith"
"Joe Smith".substring(2,5); // "e S"
```

## c. String comparison

- `oneString.equals(anotherString)`

- Tests for equivalence
- Return `true` or `false`

```
String name = "Joe";
if ("Joe".equals(name))
 name += " Smith";
```

- `oneString.equalsIgnoreCase(anotherString)`

- Case insensitive test for equivalence

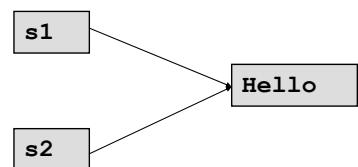
```
boolean same = "Joe".equalsIgnoreCase("joe");
```

- `oneString == anotherString` is problematic

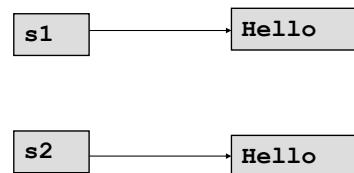
- Compare two objects

## c. Comparing two Strings (2)

```
String s1 = new String("Hello");
String s2 = s1;
(s1==s2) returns true
```

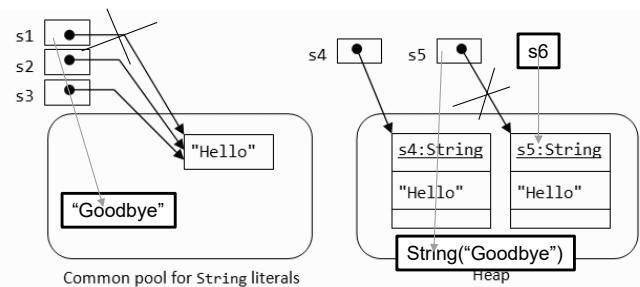


```
String s1 = new String("Hello");
String s2 = new String("Hello");
(s1==s2) returns false
s1.equals(s2) return true
```



# String Literal vs. String Object

- `String s1 = "Hello";` // String literal
- `String s2 = "Hello";` // String literal
- `String s3 = s1; // same reference`
- `String s4 = new String("Hello");` // String object
- `String s5 = new String("Hello");` // String object
- `String s6 = s5;`
- `s5 = new String("Goodbye");`
- `s1 = "Goodbye";`



44

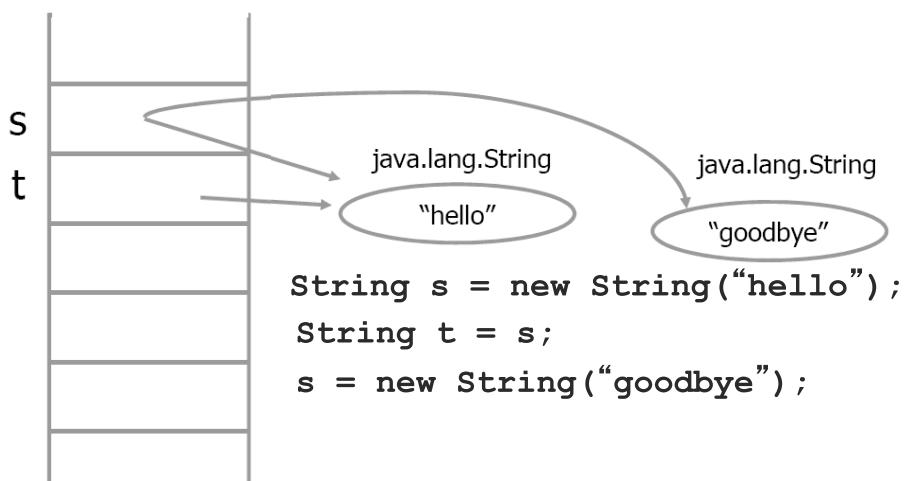
- `String str = "";`
- `for (int i=0; i<1.000.000; i++){`
  - //read a line from a file
  - `str += line;`
- `}`
  
- `StringBuffer str = "";`
- `for (int i=0; i<1.000.000; i++){`
  - //read a line from a file
  - `str.append(line);`
- `}`

### 3.3. StringBuffer/StringBuilder

- String is an immutable type:
  - Object does not change the value after being created → Strings are designed for not changing their values.
  - Concatenating strings will create a new object to store the result → String concatenation is memory consuming.
- StringBuffer/StringBuilder is a mutable type:
  - Object can change the value after being created

=> String concatenation can get very expensive,  
only use in building a simple String

### 3.3. StringBuffer (2)



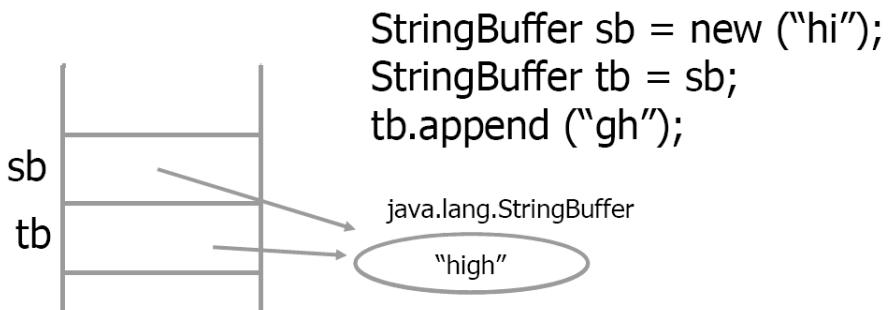
### 3.3. StringBuffer (3)

- **StringBuffer:**

- Provides String object that can change the value → Use **StringBuffer** when:
  - Predict that characters in the String can be changed
  - When processing a string, e.g. reading text data from a text file or building a String through a loop
- Provides a more efficient mechanism for building and concatenating strings:
  - String concatenation is often done by compiler in class **StringBuffer**

### 3.3. StringBuffer (4)

- Changing attribute: If an object is changed, all the relations with the object will receive the new value.



### 3.3. StringBuffer (5)

- If we create a String by a loop, we should use **StringBuffer**

```
StringBuffer buffer = new StringBuffer(15);
buffer.append("This is ");
buffer.append("String");
buffer.insert(7, " a");
buffer.append('.');
System.out.println(buffer.length()); // 17
System.out.println(buffer.capacity()); // 32
String output = buffer.toString();
System.out.println(output); // "This is a String."
```

### 3.4. Math class

- java.lang.Math** provides static data:
  - Math constants:
    - Math.E
    - Math.PI
  - Math functions:
    - max, min...
    - abs, floor, ceil...
    - sqrt, pow, log, exp...
    - cos, sin, tan, acos, asin, atan...
    - random

| «Java Class»       |
|--------------------|
| Math               |
| ■ Math ()          |
| ● sin ()           |
| ● cos ()           |
| ● tan ()           |
| ● asin ()          |
| ● acos ()          |
| ● atan ()          |
| ● toRadians ()     |
| ● toDegrees ()     |
| ● exp ()           |
| ● log ()           |
| ● sqrt ()          |
| ● IEEEremainder () |
| ● ceil ()          |
| ● floor ()         |
| ● rint ()          |
| ● atan2 ()         |
| ● pow ()           |
| ● round ()         |
| ● round ()         |
| ● random ()        |
| ● abs ()           |
| ● abs ()           |
| ● abs ()           |
| ● abs ()           |
| ● max ()           |
| ● min ()           |
| ▲ <clinit> ()      |

- Math ()
- sin ()
- cos ()
- tan ()
- asin ()
- acos ()
- atan ()
- toRadians ()
- toDegrees ()
- exp ()
- log ()
- sqrt ()
- IEEEremainder ()
- ceil ()
- floor ()
- rint ()
- atan2 ()
- pow ()
- round ()
- round ()
- initRNG ()
- random ()
- abs ()
- abs ()
- abs ()
- max ()
- max ()
- max ()
- max ()
- min ()
- min ()
- min ()
- min ()
- ▲ <clinit> ()

### 3.4. Math class (2)

- Most of functions receive arguments with type **double** and also return values with type **double**
- Example:

$$e^{\sqrt{2\pi}}$$

`Math.pow(Math.E,  
 Math.sqrt(2.0*Math.PI))`

Or:

`Math.exp(Math.sqrt(2.0*Math.PI))`

✉@NGUYỄN Thị Thu Trang, trangntt@soict.hust.edu.vn

## OBJECT-ORIENTED LANGUAGE AND THEORY

## 6. AGGREGATION AND INHERITANCE

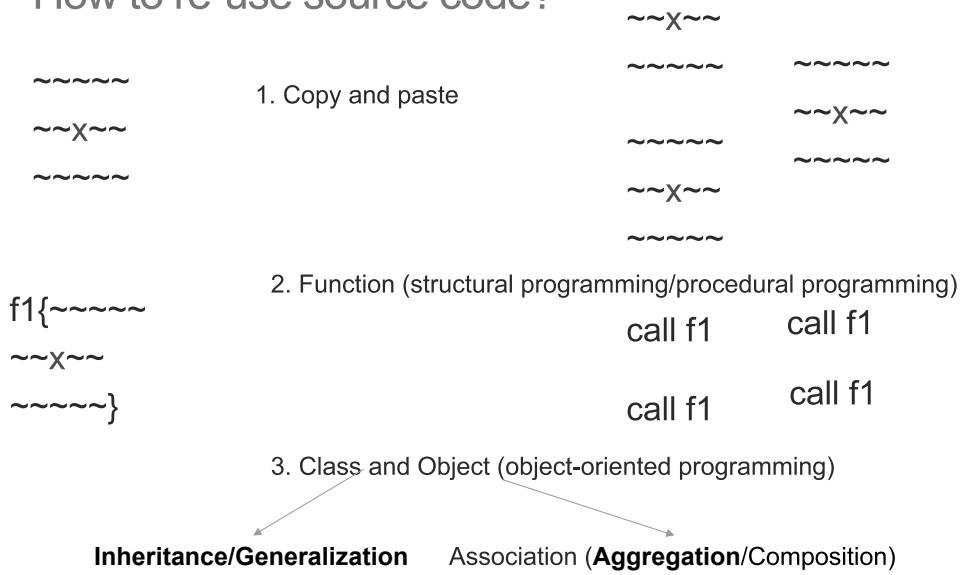
Nguyen Thi Thu Trang  
trangntt@soict.hust.edu.vn



# Tái sử dụng mã nguồn?

- Copy paste
- Viết hàm
- Thư viện, package...

## How to re-use source code?



# Lesson Goals

- Explaining concepts of source code re-usability
- Showing the nature, description of concepts relating to aggregation and inheritance
- Comparison of aggregation and inheritance
- Representing aggregation and inheritance in UML
- Explaining principles of inheritance and initialization order, object destruction in inheritance
- Applying techniques, principles of aggregation and inheritance in Java programming language

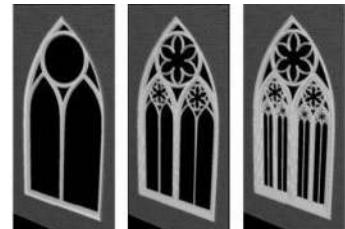
## Outline



1. Source code re-usability
2. Aggregation
3. Inheritance

# 1. Re-usability

- Source code re-usability: re-use already existing source code
  - Structure programming: Re-use function/sub-program
  - OOP: When modeling real world, there exist many object types that have similar or related attributes and behaviors
    - *How to re-use already-written classes?*



# 1. Re-usability (2)

- How to use existing classes:
  - *Copying existing classes* → Redundant and difficult to manage if any changes
  - Creating new classes that re-use of **objects** of existing classes → **Aggregation**
  - Creating new classes based on the extension of existing **classes** → **Inheritance**

# 1. Re-usability (2)

- Advantages

- Reducing man-power, cost.
- Improving software quality
- Improving modeling capacity of the real world
- Improving maintainability

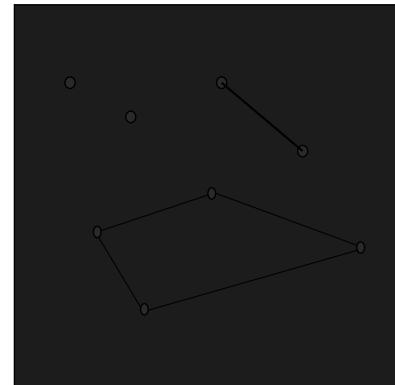


## Outline

1. Source code re-usability
2. Aggregation
3. Inheritance

## 2. Aggregation

- Example:
  - Point
    - A Quadrangle consists of 4 points  
→ Aggregation
  - Aggregation
    - Has-a or **is-a-part-of** relations

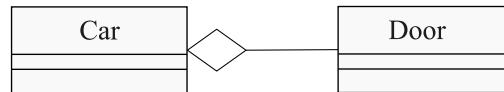


## Main terms

- Aggregate
  - Members of a new class are objects of existing classes.
  - Aggregation re-uses via *objects*
- New class
  - Called Aggregate/Whole class
- Existing class
  - Member class (part)

## 2.1. What is aggregation?

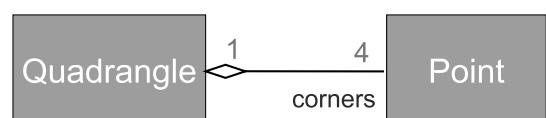
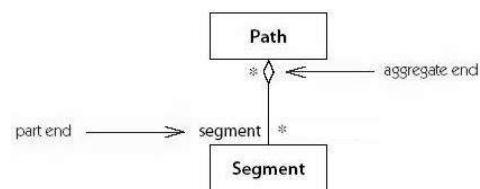
- The whole class contains objects of member classes
  - Is-a-part of the whole class
  - Re-use data and behavior of member classes via member objects



10 13

## 2.2. Representing aggregation in UML

- Using “diamond” at the head of whole class
- Using multiplicity at two heads:
  - A positive integer: 1, 2, ...
  - A range (0..1, 2..4)
  - \*: Any number
  - None: By default is 1





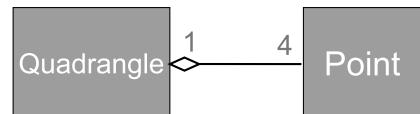
## 2.3. Example in Java

```
class Point {
 private int x, y;
 public Point(){}
 public Point(int x, int y) {
 this.x = x; this.y = y;
 }
 public void setX(int x){ this.x = x; }
 public int getX() { return x; }
 public void print(){
 System.out.print("(" + x + ", "
 + y + ")");
 }
}
```

```

class Quadrangle{
 private Point[] corners = new Point[4];
 public Quadrangle(Point p1,Point p2,Point p3,Point p4){
 corners[0] = p1; corners[1] = p2;
 corners[2] = p3; corners[3] = p4;
 }
 public Quadrangle(){
 corners[0]=new Point(); corners[1]=new Point(0,1);
 corners[2]=new Point(1,1); corners[3]=new Point(1,0);
 }
 public void print(){
 corners[0].print(); corners[1].print();
 corners[2].print(); corners[3].print();
 System.out.println();
 }
}

```

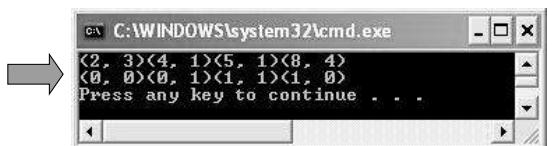


```

public class Test {
 public static void main(String arg[])
 {
 Point p1 = new Point(2,3);
 Point p2 = new Point(4,1);
 Point p3 = new Point(5,1);
 Point p4 = new Point(8,4);

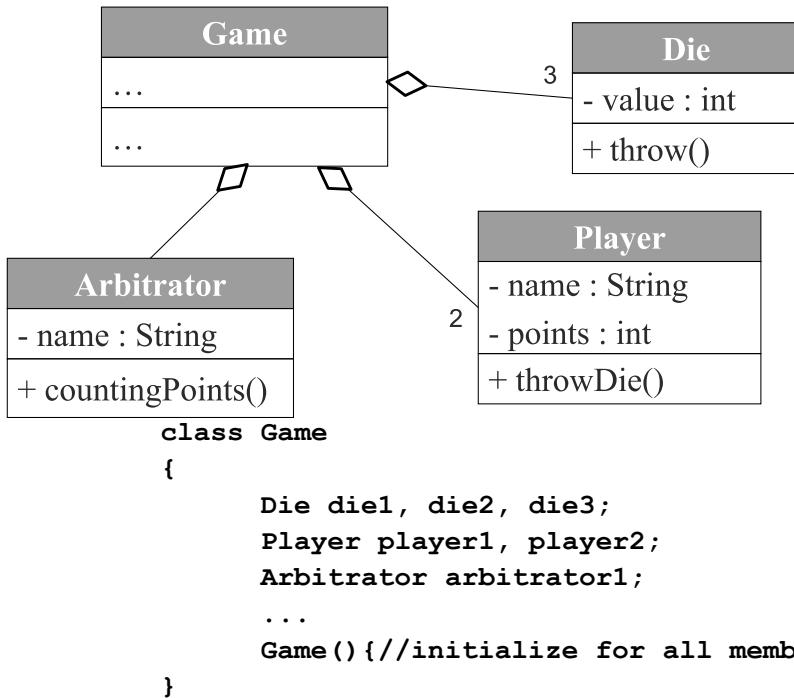
 Quadrangle q1 = new Quadrangle(p1,p2,p3,p4);
 Quadrangle q2 = new Quadrangle();
 q1.print();
 q2.print();
 }
}

```



# Another example of Aggregation

- A game consisting of two players, 3 dies and an arbitrator.
  - Need 4 classes:
    - Player
    - Die
    - Arbitrator
    - Game
- Game class is the aggregation of the 3 remaining classes



## 2.4. Initialization order in aggregation

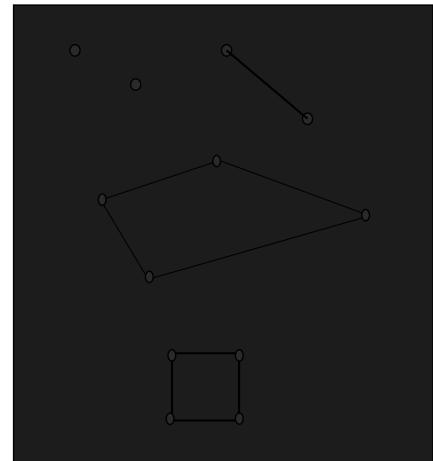
- When an object is created, the attributes of that object must be initialized and assigned corresponding values.
- Member attributes must be initialized first  
→ Constructor methods of member classes must be called first

## Outline

1. Source code re-usability
2. Aggregation
3. Inheritance

## 3.1. What is Inheritance?

- Example:
  - Point
    - A quadrangle has 4 points  
→ Aggregation (*is a part of*)
    - Quadrangle
    - Square
      - Inheritance (*is a kind of*)
      - Generalization



## Main terms

- Inherit, Derive
  - Creating new class by extending existing classes.
  - New class inherits what are in existing classes and can have its own new features.
- Existing class:
  - Parent, superclass, base class
- New class:
  - Child, subclass, derived class

# What is Inheritance?

- Principles to describe a class based on the extension of an existing class (single inheritance) or a set of existing classes (in case of multi-inheritance)
- Inheritance specifies a relationship between classes when a class shares its structure and/or behavior of a class or of other classes
- Inheritance is also called is-a-kind-of (or is-a) relationship
  - Child is a kind of parent

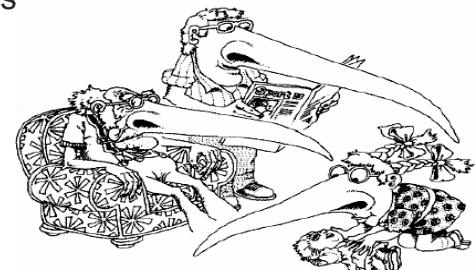
# What is Inheritance?

- On "modularization" view: If B inherits A, all services of A will be available in B
- On "type" view: If B inherits A, at anywhere a representation of A is required, the representation of B might be a good replacement.

=> Polymorphism

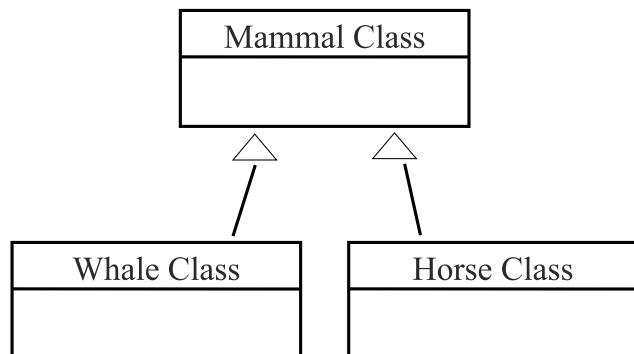
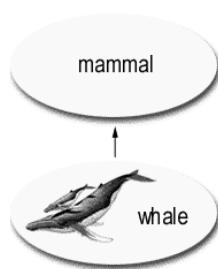
# Child classes?

- Re-use by inheriting data and behavior of parent classes
- Can be customized in two ways (or both):
  - Extension: Add more new attributes/behaviors
  - Redefinition (Method Overriding): Modify the behavior inheriting from parent class



# More example

- Whale class inherits from mammal class.
- A whale *is-a* mammal
- Whale class is *subclass*, mammal class is *superclass*



# Similarity

- Both Whale and Horse have *is-a* relation with mammal class
- Both Whale and Horse have some common behaviors of Mammal
- Inheritance is a key to re-use source code – If a parent class is created, the child class can be created and can add some more information

## 3.2. Aggregation and Inheritance

- Comparing aggregation and inheritance?
  - Similarity
    - Both are techniques in OOP in order to re-use source code
  - Difference?

# Difference between Aggregation and Inheritance

## Inheritance

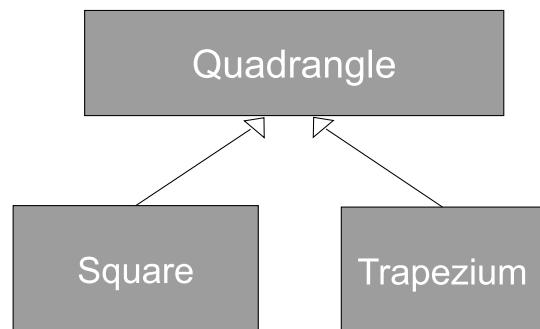
- Inheritance re-uses via class
  - Creating new class by extending existing classes
- “is a kind of” relation
- Example: Car is a kind of transportation mean

## Aggregation

- Aggregation re-uses via objects.
  - Create a reference to objects of existing classes in the new class
- “is a part of” relation
- Example: Car has 4 wheels

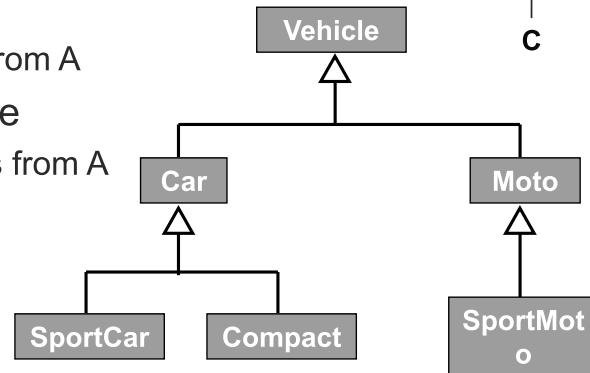
## 3.3. Representing Inheritance in UML

- Using “empty triangle” at parent class



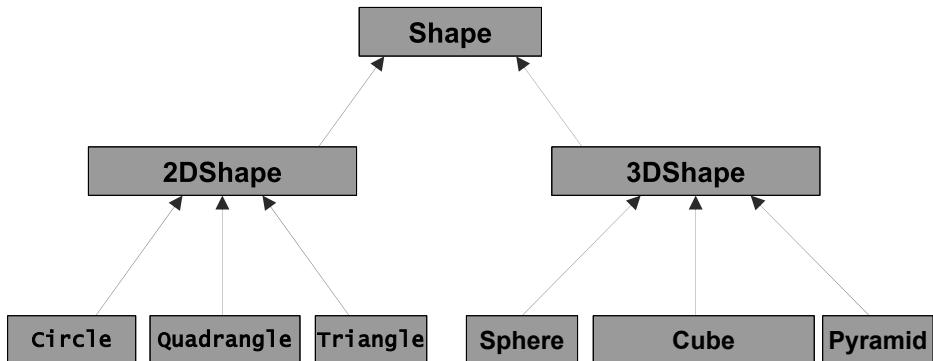
## 3.4. Inheritance hierarchy

- Is hierarchy tree structure, representing inheritance relation between classes.
- Direct inheritance
  - B directly inherits from A
- Indirect inheritance
  - C indirectly inherits from A



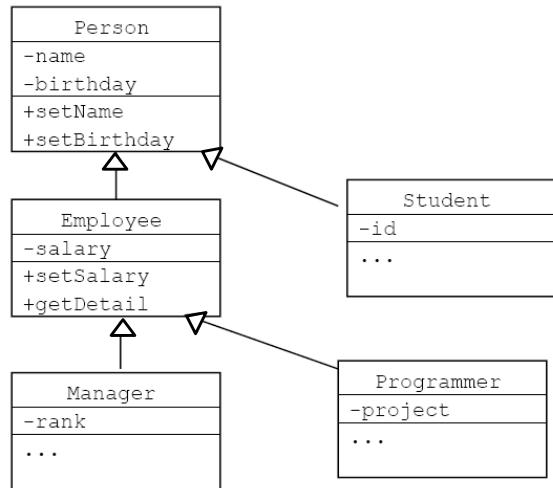
## 3.4. Inheritance hierarchy (2)

- Child classes having the same parent class are called **siblings**
- A child class inherits **all its ancestors**



## 3.4. Hierarchy tree (2)

All objects inherit  
from the basic  
class **Object**

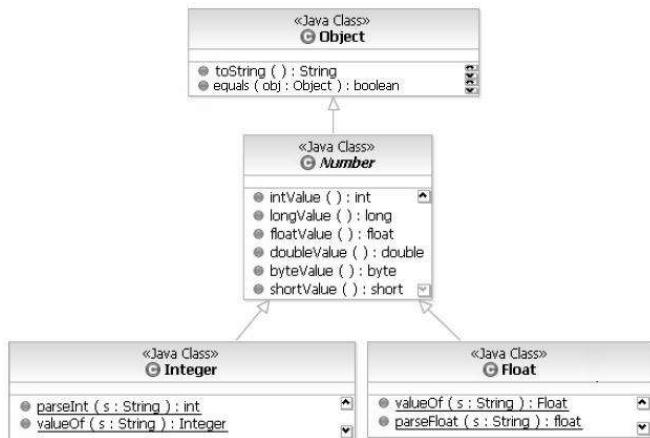


## Class Object

- Class **Object** is defined in the standard package `java.lang`
- If a class is not defined as a child of another class, it is by default a direct child of class **Object**.  
→ Class **Object** is the root class on the top level in the hierarchy tree

## Class Object (2)

- Contains some useful methods that are inherited by all other classes, for example: `toString()`, `equals()`...



## 3.5. Inheritance rules

- Access attribute: **protected** (access modifier)
- Protected members in a parent class is accessed by:
  - Members of parent classes
  - Members of children classes
  - Members of classes in the same package as the parent class
- What does a child class inherit?
  - Inherit all the attributes/methods that are declared as public and protected in the parent class.
  - Does not inherit private attributes/methods.

## 3.5. Inheritance rules (2)

| Visibility of members in parent class | <b>public</b> | <b>None<br/>(default)</b> | <b>protected</b> | <b>private</b> |
|---------------------------------------|---------------|---------------------------|------------------|----------------|
| Classes in the same package           |               |                           |                  |                |
| Child classes – same package          |               |                           |                  |                |
| Child classes – different package     |               |                           |                  |                |
| Different package, non-inher          |               |                           |                  |                |

## 3.5. Inheritance rules (2)

|                                   | <b>public</b> | <b>None</b> | <b>protected</b> | <b>private</b> |
|-----------------------------------|---------------|-------------|------------------|----------------|
| Same package                      | Yes           | Yes         | Yes              | No             |
| Child classes – same package      | Yes           | Yes         | Yes              | No             |
| Child classes – different package | Yes           | No          | Yes              | No             |
| Different package, non-inher      | Yes           | No          | No               | No             |

### 3.5. Inheritance rules (3)

- Methods that can not be inherited:
  - Construction and destruction methods
  - Methods that initialize and delete objects
  - These methods are only defined to work in a specific class
- Assignment operation =
  - Performs the same task as construction method

### 3.6. Inheritance syntax in Java

- Inheritance syntax in Java:
  - <SubClass> extends <SuperClass>
- Example:

```
class Square extends Quadrangle {
 ...
}
class Bird extends Animal {
 ...
}
```

```
public class Quadrangle {
 protected Point corners = new Point[4];
 public Quadrangle(){ ... }
 public void print(){...}
 ...
}
```

### Example 1

Using protected attributes of the parent class in the child class

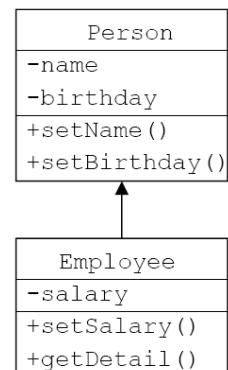
```
public class Square extends Quadrangle {
 public Square(){
 corners[0]=new Point(0,0); corners[1]=new Point(0,1);
 corners[2]=new Point(1,0); corners[3]=new Point(1,1);
 }
}
public class Test{
 public static void main(String args[]){
 Square sq = new Square();
 sq.print(); ←
 }
}
```

Calling public method of parent class

### Example 2

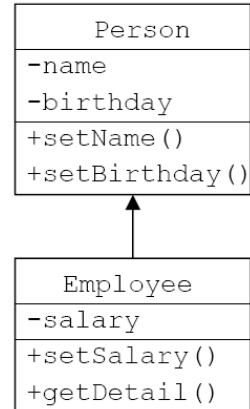
**protected**

```
class Person {
 private String name;
 private Date birthday;
 public String getName() {return name;}
 ...
}
class Employee extends Person {
 private double salary;
 public boolean setSalary(double sal){
 salary = sal;
 return true;
 }
 public String getDetail(){
 String s = name+", "+birthday+", "+salary;//Error
 }
}
```



## Example 2 (cont.)

```
public class Test{
 public static void main(String args[]){
 Employee e = new Employee();
 e.setName("John");
 e.setSalary(3.0);
 }
}
```



## Example 3 – Same package

```
public class Person {
 Date birthday;
 String name;
 ...
}

public class Employee extends Person {
 ...
 public String getDetail() {
 String s;
 String s = name + "," + birthday;
 s += "," + salary;
 return s;
 }
}
```

## Example 3 – Different package

```
package abc;
public class Person {
 protected Date birthday;
 protected String name;
 ...
}

import abc.Person;
public class Employee extends Person {
 ...
 public String getDetail() {
 String s;
 s = name + "," + birthday + "," + salary;
 return s;
 }
}
```

## Construction and destruction of objects in inheritance

- Object construction:
  - A parent class is initialized before its child classes.
  - Construction methods of a child class always call construction methods of its parent class at the very first command
    - Implicit call: whe the parent class has a **default constructor**
    - Explicit call (explicit)
- Object destruction:
  - Contrary to object initialization

### 3.4.1. Implicit call of constructor of parent class

```

public class Quadrangle {
 public Quadrangle(){
 System.out.println
 ("Parent Quadrangle()");
 }
 //...
}

public class Square
 extends Quadrangle {
 public Square(){
 //Implicit call "Quadrangle()"
 System.out.println
 ("Child Square()");
 }
}

```

```

public class Test {
 public static void
 main(String arg[])
 {
 HinhVuong hv =
 new HinhVuong();
 }
}

```

↓

Parent Quadrangle()  
Child Square()

### Example

```

public class Quadrangle {
 protected Point[] corners=new Point[4];
 public Quadrangle(Point p1,Point p2,
 Point p3,Point p4){
 corners[0] = p1; corners[1] = p2;
 corners[2] = p3; corners[3] = p4;
 }
}

public class Square extends
Quadrangle {
 public Square(){
 System.out.println
 ("Child Square()");
 }
}

```

```

public class Test {
 public static void
 main(String arg[])
 {
 Square sq = new
 Square();
 }
}

```

Error

Cannot find symbol ...

### 3.4.2. Implicit constructor call of parent class

- The first command in constructor of a child class can call the constructor of its parent class
  - `super(Danh_sach_tham_so);`
- This is obliged if the parent class does not have any default constructor
  - Parent class already has a constructor with arguments
  - The constructor of child class must not have arguments.

```

public class Quadrangle {
 protected Point corners = new Point[4];
 public Quadrangle(){ ... }
 public Quadrangle(Point d1,Point d2,Point d3, Point d4)
 { ... }
 public void print(){...}
}
public class Square extends Quadrangle {
 public Square(){ super(); }
 public Square(Point p1,Point p2,Point p3,Point p4){
 super(d1, d2, d3, d4);
 }
}
public class Test{
 public static void main(String args[]){
 Square sq = new Square();
 sq.print();
 }
}

```

Example 1.1

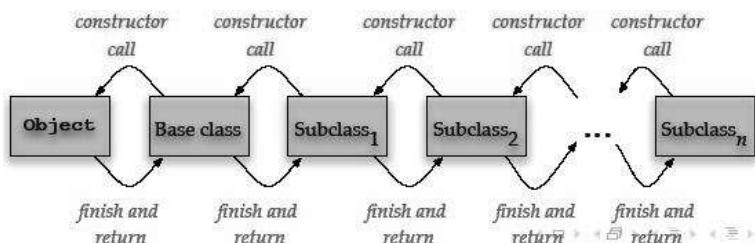
## Explicit constructor call of parent class

Constructor of child class has no arguments

```
public class Quadrangle {
 protected Point[] corners=new Point[4];
 public Quadrangle(Point p1,Point p2,
 Point p3,Point p4){
 System.out.println("Parent Quadrangle()");
 corners[0] = p1; corners[1] = p2;
 corners[2] = p3; corners[3] = p4;
 }
}
public class Square extends Quadrangle {
 public Square(){
 super(new Point(0,0),new Point(0,1),new Point(1,1),
 new Point(1,0));
 System.out.println("Child Square()");
 }
}
```

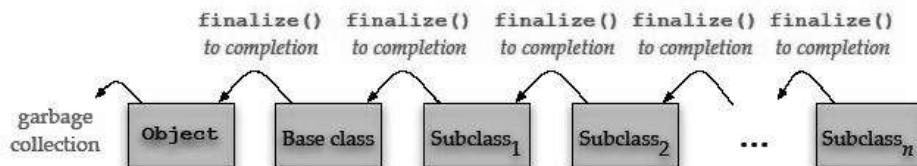
## Implicit call of constructor

- When initializing an object, a series of constructors will be called explicitly (via super() method call or implicitly)
- Constructor call of the most basic class in the hierarchy tree will be done last, but will finish first. The constructor of the derived class will finish at the last.



# Implicit call of finalize()

- When an object is destroyed (by GC), a series of finalize() methods will be called automatically.
- The order is inverse compared to the calls of constructors
  - Method finalize() of derived class is called first, then the ones of its parent class



@NGUYỄN Thị Thu Trang, trangntt@soict.hust.edu.vn

## OBJECT-ORIENTED LANGUAGE AND THEORY

### 7. ABSTRACT CLASS AND INTERFACE

Nguyen Thi Thu Trang  
trangntt@soict.hust.edu.vn



## Outline

1. Redefine/Overiding
2. Abstract class
3. Single inheritance and multi-inheritance
4. Interface

## Outline

- 1. Redefine/Overiding
- 2. Abstract class
- 3. Single inheritance and multi-inheritance
- 4. Interface

# 1. Re-definition or Overriding

- A child class can define a method with the **same name** of a method in its parent class:
  - If the new method has the same name but different signature (number or data types of method's arguments)  
→ Method Overloading
  - If the new method has the same name and signature  
→ Re-definition or Overriding  
(Method Redefine/Override)

```
• class A {
 a(){ }
}

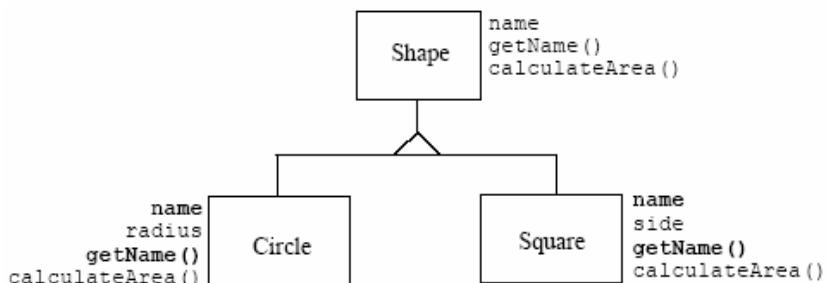
• class B extends A {
 a(String) {}
}

... B b = new B();
b.a();
b.a("test");
```

- ParentClass: aMethod() => overridden method
  - ChildClass1: aMethod(), aMethod(String) => Overloading
  - ChildClass2: aMethod() => Overriding/Redefinition method
  
- ChildClass1 cc1 = new ChildClass1();
- cc1.aMethod(); cc1.aMethod("a string");
- ChildClass2 cc2 = new ChildClass2();
- cc2.aMethod();

## 1. Re-definition or Overriding (2)

- Overriding method will replace or add more details to the overriden method in the parent class
- Objects of child class will use the re-defined method



- `this()` and `this => current object`
- `super() => Constructor of the parent class`
- `super: object of the parent class`

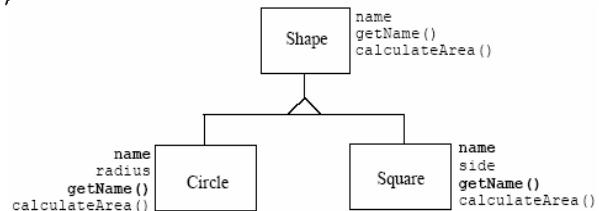
```
class Shape {
 protected String name;
 Shape(String n) { name = n; }
 public String getName() { return name; }
 public float calculateArea() { return 0.0f; }
}
class Circle extends Shape {
 private int radius;
 Circle(String n, int r){
 super(n);
 radius = r;
 }

 public float calculateArea() {
 float area = (float) (3.14 * radius * radius);
 return area;
 }
}
```

```

class Square extends Shape {
 private int side;
 Square(String n, int s) {
 super(n);
 side = s;
 }
 public float calculateArea() {
 float area = (float) side * side;
 return area;
 }
}

```



## Class Triangle

```

class Triangle extends Shape {
 private int base, height;
 Triangle(String n, int b, int h) {
 super(n);
 base = b; height = h;
 }
 public float calculateArea() {
 float area = 0.5f * base * height;
 return area;
 }
}

```

## this and super

- **this** and **super** can use non-static methods/attributes and constructors
  - **this**: searching for methods/attributes in the current class
  - **super**: searching for methods/attributes in the direct parent class
- Keyword **super** allows re-using the source-code of a parent class in its child classes

```
package abc;
public class Person {
 private String name;
 private int age;
 public String getDetail() {
 String s = name + "," + age;
 return s;
 }
 private void pM(){}
}

import abc.Person;
public class Employee extends Person {
 double salary;
 public String getDetail() {
 String s = super.getDetail() + "," + salary
 return s;
 }
}
```

# Overriding Rules

- Overriding methods must have:
  - An argument list that is the same as the overriden method in the parent class => signature
  - The same return data types as the overriden method in the parent class
- Can not override:
  - Constant (final) methods in the parent class
  - Static methods in the parent class
  - Private methods in the parent class

# Overriding Rules (2)

- Accessibility can not be more restricted in a child class (compared to in its parent class)
  - For example, if overriding a protected method, the new overriding method can only be protected or public, and can not be private.

## Example

```
class Parent {
 public void doSomething() {}
 protected int doSomething2() {
 return 0;
 }
}

class Child extends Parent {
 protected void doSomething() {}
 protected void doSomething2() {}
}
```

cannot override: attempting to use incompatible return type

cannot override: attempting to assign weaker access privileges; was public

## Example: private

```
class Parent {
 public void doSomething() {}
 private int doSomething2() {
 return 0;
 }
}

class Child extends Parent {
 public void doSomething() {}
 private void doSomething2() {}
}
```

# Outline

1. Redefine/Overiding
2. Abstract class
3. Single inheritance and multi-inheritance
4. Interface

## Abstract Class

- An abstract class is a class that we can not create its objects. Abstract classes are often used to define "Generic concepts", playing the role of a basic class for others "detailed" classes.
- Using keyword abstract

```
public abstract class Product
{
 // contents
}
...Product aProduct = new Product(); //error
```

concrete class vs. abstract class

## 2. Abstract Class

- Can not create objects of an abstract class
- Is not complete, is often used as a parent class. Its children will complement the un-completed parts.

### Abstract Class

- Abstract class can contain un-defined abstract methods
- Derived classes must re-define (overriding) these abstract methods
- Using abstract class plays an important role in software design. It defines common objects in inheritance tree, but these objects are too abstract to create their instances.

## 2. Abstract Class (2)

- To be abstract, a class needs:
  - To be declared with **abstract** keyword
  - May contain abstract methods – that have only signatures without implementation
    - public abstract float calculateArea();
  - Child classes must implement the details of abstract methods of their parent class → Abstract classes can not be declared as final or static.
- If a class has one or more abstract methods, it must be an abstract class

```
abstract class Shape {
 protected String name;
 Shape(String n) { name = n; }
 public String getName() { return name; }
 public abstract float calculateArea();
}

class Circle extends Shape {
 private int radius;
 Circle(String n, int r) {
 super(n);
 radius = r;
 }
 public float calculateArea() {
 float area = (float) (3.14 * radius * radius);
 return area;
 }
}
```

Child class must override all the abstract methods of its parent class

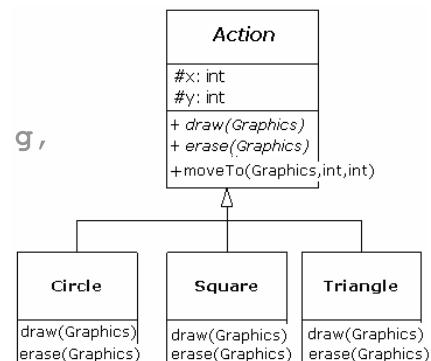
## Example of abstract class

```

import java.awt.Graphics;
abstract class Action {
 protected int x, y;
 public void moveTo(Graphics g,
 int x1, int y1) {
 erase(g);
 x = x1; y = y1;
 draw(g);
 }
 public abstract void erase(Graphics g);
 public abstract void draw(Graphics g);
}

..Circle c = new Circle();
c.moveTo(...);

```



## Example of abstract class (2)

```

class Circle extends Action {
 int radius;
 public Circle(int x, int y, int r) {
 super(x, y); radius = r;
 }
 public void draw(Graphics g) {
 System.out.println("Draw circle at (" +
 + x + "," + y + ")");
 g.drawOval(x-radius, y-radius,
 2*radius, 2*radius);
 }
 public void erase(Graphics g) {
 System.out.println("Erase circle at (" +
 + x + "," + y + ")");
 // paint the circle with background color...
 }
}

```

# Abstract Class

```
abstract class Point {
 private int x, y;
 public Point(int x, int y) {
 this.x = x;
 this.y = y;
 }
 public void move(int dx, int dy) {
 x += dx; y += dy;
 plot();
 }
 public abstract void plot();
}
```

# Abstract Class

```
abstract class ColoredPoint extends Point {
 int color;
 public ColoredPoint(int x, int y, int color) {
 super(x, y); this.color = color; }
}

class SimpleColoredPoint extends ColoredPoint {
 public SimpleColoredPoint(int x, int y, int color){
 super(x,y,color);
 }
 public void plot() {
 ...
 // code to plot a SimplePoint
 }
}
```

# Abstract Class

- Class ColoredPoint does not implement source code for the method plot(), hence it must be declared as abstract
- Can only create objects of the class SimpleColoredPoint.
- However, we can have:  
Point p = new SimpleColoredPoint(a, b, red); p.plot();

29

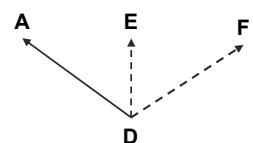
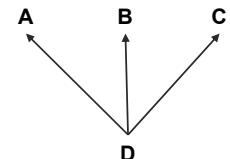
- abstract class A {  
    abstract void a();
- }
- class B extend A {  
• }

# Outline

1. Redefine/Overriding
2. Abstract class
3. Single inheritance and multi-inheritance
4. Interface

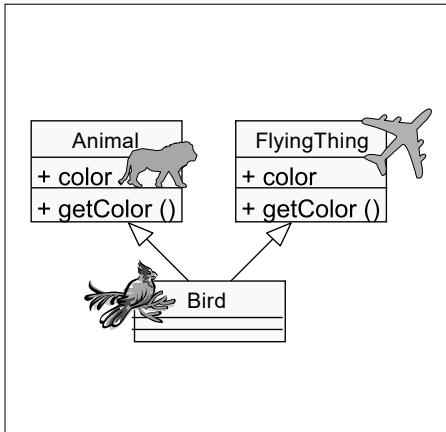
## Multiple and Single Inheritances

- Multiple Inheritance
  - A class can inherit several other classes
  - C++ supports multiple inheritance
- Single Inheritance
  - A class can inherit only one other class
  - Java supports only single inheritance
  - → Need to add the notion of Interface

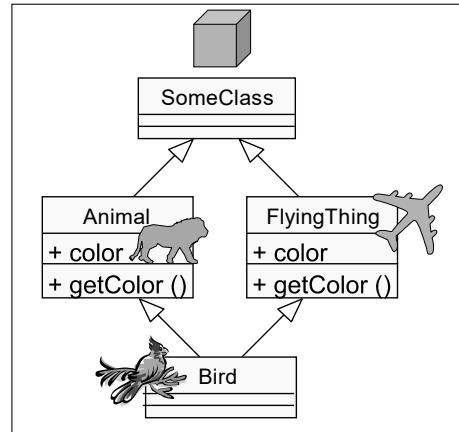


# Problems in Multiple Inheritance

Name clashes on  
attributes or operations



Repeated inheritance



Resolution of these problems is implementation-dependent.

33

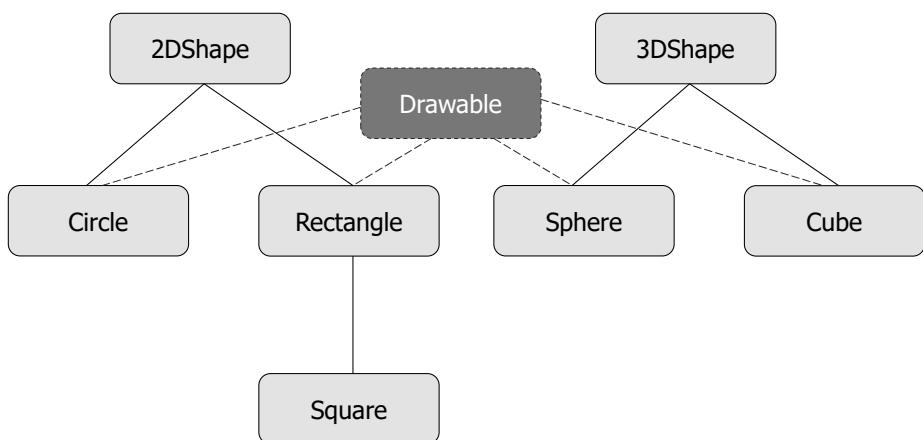
## Outline

1. Redefine/Overriding
2. Abstract class
3. Single inheritance and multi-inheritance
4. Interface

# Mix-in inheritance

- In this inheritance, a "class" will provide some functions in order to mix with other classes.
- A mixed class often re-uses some functions defined in the provider class but also inherits from another class.
- Is a mean that allows objects without relation in the hierarchy tree can communicate to each other.
- In Java the mix-in inheritance is done via Interface

## Interface



# Interface

- Interface: Corresponds to different implementations.
- Defines the border:
  - What How
  - Declaration and Implementation.

# Interface

- Interface does not implement any methods but defines the design structure in any class that uses it.
- An interface: 1 contract – in which software development teams agree on how their products communicate to each other, without knowing the details of product implementation of other teams.

# Example

- Class Bicycle – Class StoreKeeper:
  - StoreKeepers does not care about the characteristics what they keep, they care only the price and the id of products.
- Class AutonomousCar– GPS:
  - Car manufacturers produce cars with features: Start, Speed-up, Stop, Turn left, Turn right,..
  - GPS: Location information, Traffic status – Making decisions for controlling car
  - How does GPS control both car and space craft?

## Interface OperateCar

```
public interface OperateCar {

 // Constant declaration– if any

 // Method signature
 int turn(Direction direction, // An enum with values RIGHT, LEFT
 double radius, double startSpeed, double endSpeed);
 int changeLanes(Direction direction, double startSpeed, double
 endSpeed);
 int signalTurn(Direction direction, boolean signalOn);
 int getRadarFront(double distanceToCar, double speedOfCar);
 int getRadarRear(double distanceToCar, double speedOfCar);

 // Signatures of other methods
}
```

# Class OperateBMW760i

## // Car Manufacturer

```
public class OperateBMW760i implements OperateCar {
```

```
// cài đặt hợp đồng định nghĩa trong giao diện
```

```
int signalTurn(Direction direction, boolean signalOn) {
```

```
 //code to turn BMW's LEFT turn indicator lights on
```

```
 //code to turn BMW's LEFT turn indicator lights off
```

```
 //code to turn BMW's RIGHT turn indicator lights on
```

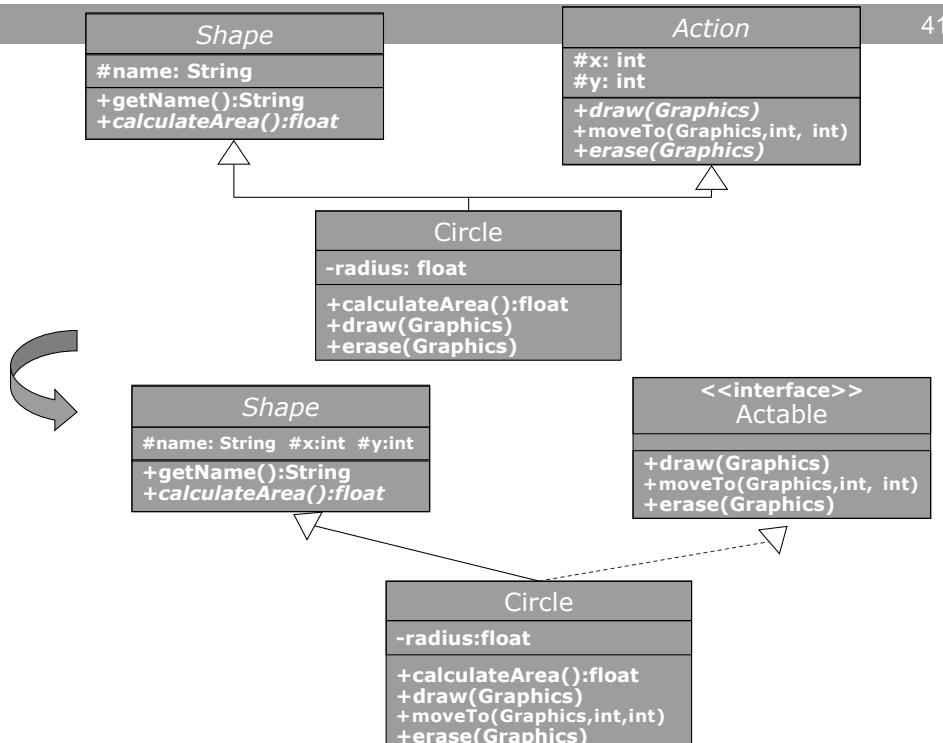
```
 //code to turn BMW's RIGHT turn indicator lights off
```

```
}
```

```
// Các phương thức khác, trong suốt với các clients của
interface
```

```
}
```

41



## 4. Interface

- Allows a class to inherit (implement) multiple interfaces at the same time.
- Can not directly instantiate

### Interface – Technique view (JAVA)

- An interface can be considered as a “class” that
  - Its methods and attributes are implicitly public
  - Its attributes are static and final (implicitly)
  - Its methods are abstract
- ```
interface TVInterface {  
    public void turnOn();  
    public void turnOff();  
    public void changeChannel(int i);  
}  
class PanasonicTV implements TVInterface{  
    public void turnOn() { .... }  
}
```

4. Interface (2)

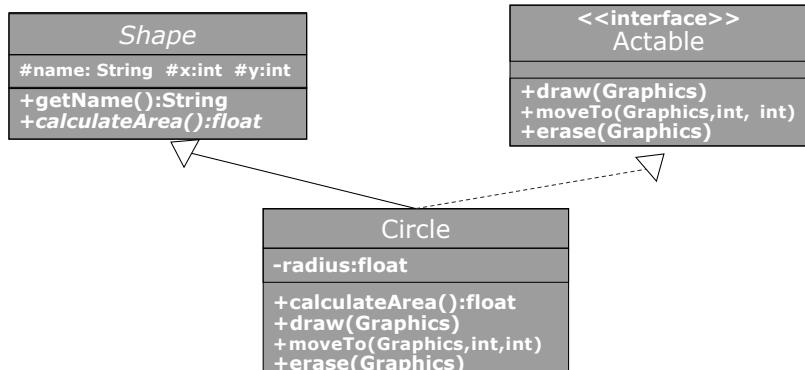
- To become an interface, we need
 - To use interface keyword to define
 - To write only:
 - method signature
 - static & final attributes
- Implementation class of interface
 - Abstract class
 - Concrete class: Must implement all the methods of the interface

4. Interface (3)

- Java syntax:
 - SubClass **extends** SuperClass **implements** ListOfInterfaces
 - SubInterface **extends** SuperInterface
- Example:

```
public interface Symmetrical {...}
public interface Movable {...}
public class Square extends Shape
    implements Symmetrical, Movable {
    ...
}
```

Example



```

import java.awt.Graphics;
abstract class Shape {
    protected String name;
    protected int x, y;
    Shape(String n, int x, int y) {
        name = n; this.x = x; this.y = y;
    }
    public String getName() {
        return name;
    }
    public abstract float calculateArea();
}
interface Actable {
    public void draw(Graphics g);
    public void moveTo(Graphics g, int x1, int y1);
    public void erase(Graphics g);
}
  
```

```

class Circle extends Shape implements Actable {
    private int radius;
    public Circle(String n, int x, int y, int r){
        super(n, x, y); radius = r;
    }
    public float calculateArea() {
        float area = (float) (3.14 * radius * radius);
        return area;
    }
    public void draw(Graphics g) {
        System.out.println("Draw circle at (" +
                           + x + "," + y + ")");
        g.drawOval(x-radius,y-radius,2*radius,2*radius);
    }
    public void moveTo(Graphics g, int x1, int y1){
        erase(g); x = x1; y = y1; draw(g);
    }
    public void erase(Graphics g) {
        System.out.println("Erase circle at (" +
                           + x + "," + y + ")");
        // paint the region with background color...
    }
}

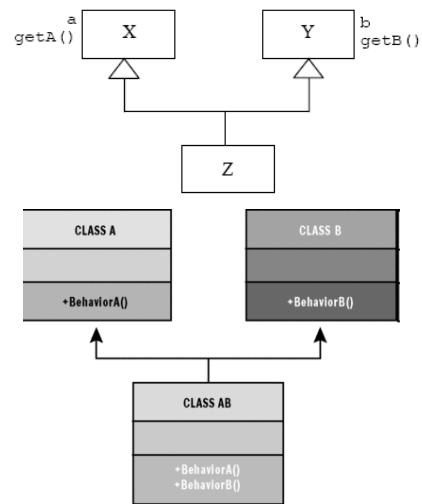
```

Abstract class vs. Interface

- May or may not contain abstract methods, can contain instance methods
- Can contain protected and static methods
- Can contain final and non-final attributes
- A class can inherit only one abstract class
- Can contain only method signature
- Can contain only public functions without implementation
- Can contains only constant attributes
- A class can inherit multiple interfaces

Disadvantages of Interface in solving Multiple Inheritance problems

- Does not provide a nature way for situations without inheritance conflicts
- Inheritance is to re-uses source code but Interface can not do this



Example

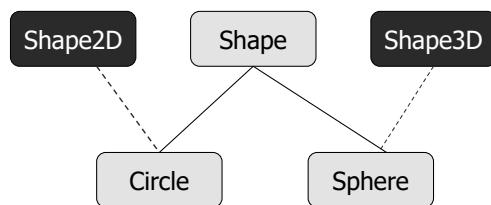
```

interface Shape2D {
    double getArea();
}

interface Shape3D {
    double getVolume();
}

class Point3D {
    double x, y, z;

    Point3D(double x, double y, double z) {
        this.x = x;
        this.y = y;
        this.z = z;
    }
}
  
```



```
abstract class Shape {
    abstract void display();
}

class Circle extends Shape
implements Shape2D {
    Point3D center; p; // p is an point on circle

    Circle(Point3D center, Point3D p) {
        this.center = center;
        this.p = p;
    }

    public void display() {
        System.out.println("Circle");
    }

    public double getArea() {
        double dx = center.x - p.x;
        double dy = center.y - p.y;
        double d = dx * dx + dy * dy;
        double radius = Math.sqrt(d);
        return Math.PI * radius * radius;
    }
}
```

```
class Sphere extends Shape
implements Shape3D {
    Point3D center;
    double radius;

    Sphere(Point3D center, double radius) {
        this.center = center;
        this.radius = radius;
    }

    public void display() {
        System.out.println("Sphere");
    }

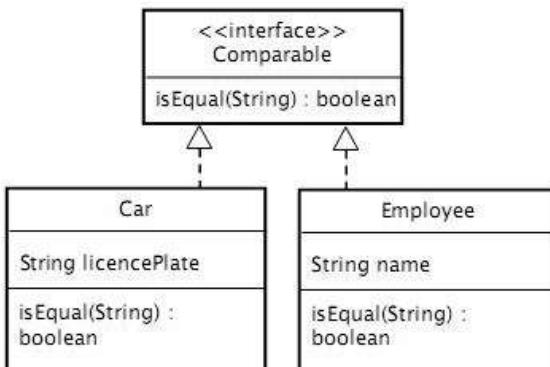
    public double getVolume() {
        return 4 * Math.PI * radius * radius * radius / 3;
    }
}

class Shapes {
    public static void main(String args[]) {
        Circle c = new Circle(new Point3D(0, 0, 0), new
            Point3D(1, 0, 0));
        c.display();
        System.out.println(c.getArea());
        Sphere s = new Sphere(new Point3D(0, 0, 0), 1);
        s.display();
        System.out.println(s.getVolume());
    }
}
```

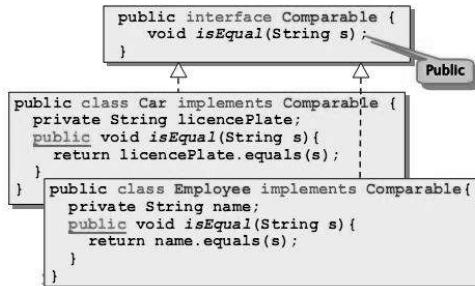
Result :

```
Circle
3.141592653589793
Sphere
4.1887902047863905
```

interface Comparable /java.lang



Application



Application

```
public class Foo {
    private Comparable objects[];
    public Foo(){
        objects = new Comparable[3];
        objects[0] = new Employee();
        objects[1] = new Car();
        objects[2] = new Employee();
    }
    public Comparable find(String s){
        for(int i=0; i< objects.length; i++)
            if(objects[i].isEqual(s))
                return objects[i];
    }
}
```

@NGUYỄN Thị Thu Trang, trangntt@soict.hust.edu.vn

OBJECT-ORIENTED LANGUAGE AND THEORY

8. POLYMORPHISM

Nguyen Thi Thu Trang
trangntt@soict.hust.edu.vn

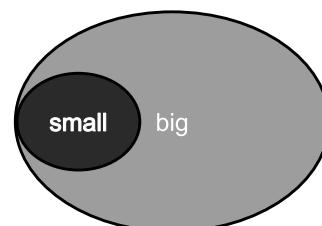


Outline

- 1. Upcasting and Downcasting
- 2. Static and dynamic bindings
- 3. Polymorphism
- 4. Generic programming

Primitive data

- Upcasting:
 - small to big range
 - implicitly cast
 - e.g. byte => short => int => double
 - byte b = 2;
 - short s = b;
- Downcasting
 - big to small
 - explicitly cast
 - e.g. int => short
 - (short)



Object/Class

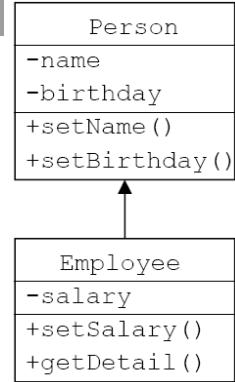
- Parent and child: Child is a kind of Parent
- If parent is smaller: Person and Employee
 - Parent is always a child
 - Child is not always a parent
- **If child is smaller => TRUE**
 - Employee is always a person
 - Person is not always an employee

1.1. Upcasting

- Moving up the inheritance hierarchy
- Up casting is the capacity to view an object of a derived class as an object of its base class.
- Automatic type conversion (implicitly)

Example

```
public class Test1 {  
    public static void main(String arg[]) {  
        Person p;  
        Employee e = new Employee();  
        p = e; //upcasting  
        p.setName("Hoa");  
        p.setSalary(350000); // compile error  
  
        Employee e1 = (Employee) p; //downcasting  
        e1.setSalary(350000); //ok  
    }  
}
```



7

Example (2)

```
class Manager extends Employee {  
    Employee assistant;  
    // ...  
    public void setAssistant(Employee e) {  
        assistant = e;  
    }  
    // ...  
}  
  
public class Test2 {  
    public static void main(String arg[]) {  
        Manager junior, senior;  
        // ...  
        senior.setAssistant(junior);  
    }  
}
```

Example (3)

```
public class Test3 {  
    String static teamInfo(Person p1, Person p2) {  
        return "Leader: " + p1.getName() +  
               ", member: " + p2.getName();  
    }  
  
    public static void main(String arg[]) {  
        Employee e1, e2;  
        Manager m1, m2;  
        // ...  
        System.out.println(teamInfo(e1, e2));  
        System.out.println(teamInfo(m1, m2));  
        System.out.println(teamInfo(m1, e2));  
    }  
}
```

1.2. Downcasting

- Move back down the inheritance hierarchy
- Down casting is the capacity to view an object of a base class as an object of its derived class.
- Does not convert types automatically
→ Must cast types explicitly.

Example

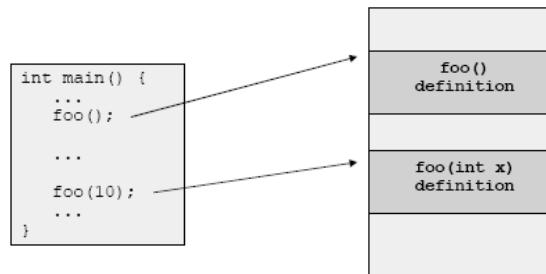
```
public class Test2 {  
    public static void main(String arg[]) {  
        Employee e = new Employee();  
        Person p = e; // up casting  
        Employee ee = (Employee) p; // down casting  
        Manager m = (Manager) ee; // run-time error  
  
        Person p2 = new Manager();  
        Employee e2 = (Employee) p2;  
    }  
}
```

Outline

1. Upcasting and Downcasting
2. Static and dynamic bindings
3. Polymorphism
4. Generic programming

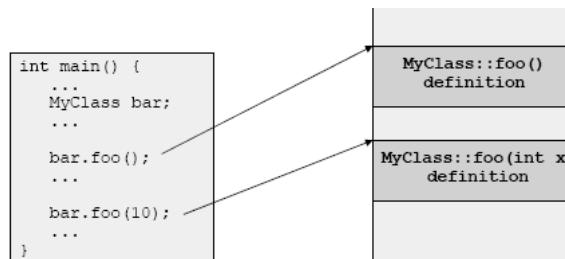
Function call binding

- Function call binding is a procedure to **specify the piece of code that need to be executed** when calling a function
- E.g. C language: a function has a unique name



OOP languages (method call binding)

- For independent classes (are not in any inheritance tree), the procedure is almost the same as function call binding
 - Compare function name, argument list to find the corresponding definition

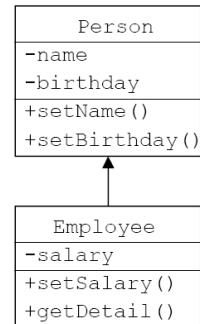


2.1. Static Binding

- Binding at the compiling time
 - Early Binding/Compile-time Binding
 - Function call is done when compiling, hence there is only one instance of the function
 - Any error will cause a compiling error
 - Advantage of speed
- C/C++ function call binding, and C++ method binding are basically examples of static function call binding

Example

```
public class Test {
    public static void main(String arg[]) {
        Person p = new Person();
        p.setName("Hoa");
        p.setSalary(350000); //compile-time error
    }
}
```

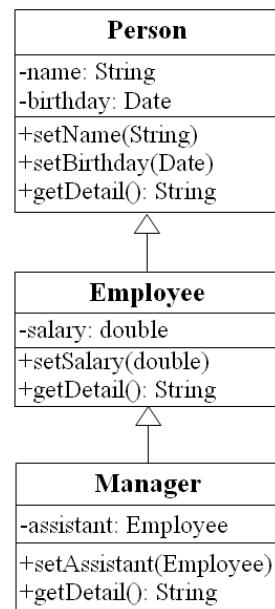


2.2. Dynamic binding

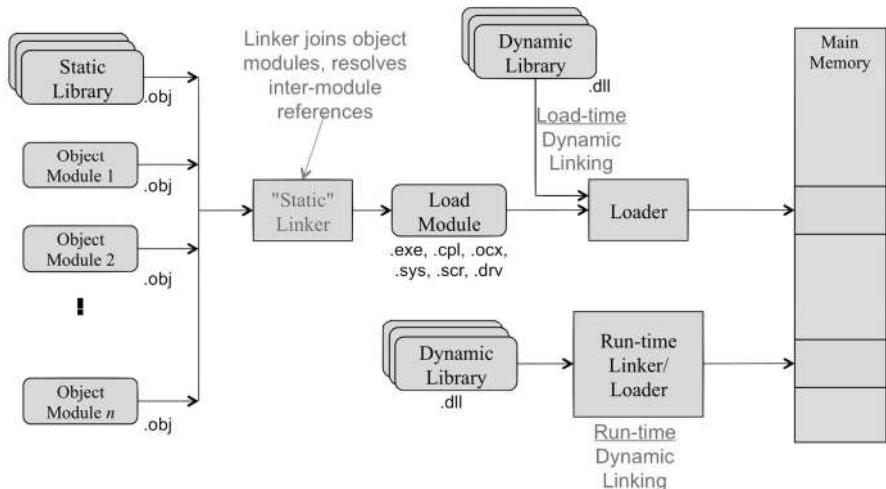
- The method call is done at run-time
 - Late binding/Run-time binding
 - Instance of method is suitable for called object.
 - Java uses dynamic binding by default

Example

```
public class Test {
    public static void main(String arg[]){
        Person p = new Person();
        // ...
        Employee e = new Employee();
        // ...
        Manager m = new Manager();
        // ...
        Person pArr[] = {p, e, m}; //upcasting
        for (int i=0; i<pArr.length; i++){
            System.out.println(
                pArr[i].getDetail());
        }
    }
}
```



Linker and Loader



Outline

1. Upcasting and Downcasting
2. Static and dynamic bindings
3. Polymorphism
4. Generic programming

3. Polymorphism

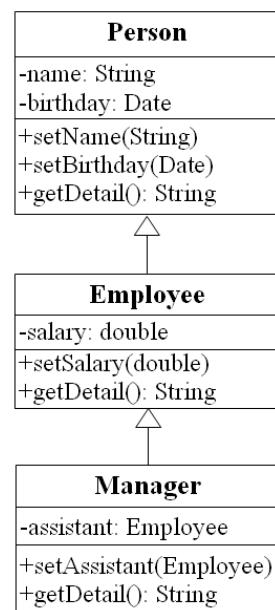
- Polymorphism: multiple ways of performance, of existence
- Polymorphism in OOP
 - Method polymorphism:
 - Methods with the same name, only difference in argument lists => method overloading
 - Object polymorphism
 - **Multiple types:** A single object to represent multiple different types (upcasting and downcasting)
 - **Multiple implementations/behaviors:** A single interface to objects of different types (upcasting+overriding – dynamic binding)

3. Polymorphism (2)

- A single symbol to represent multiple different types
→ Upcasting and Downcasting

```
public class Test3 {
    public static void main(String args[]) {
        Person p1 = new Employee();
        Person p2 = new Manager();

        Employee e = (Employee) p1;
        Manager m = (Manager) p2;
    }
}
```



3. Polymorphism (5)

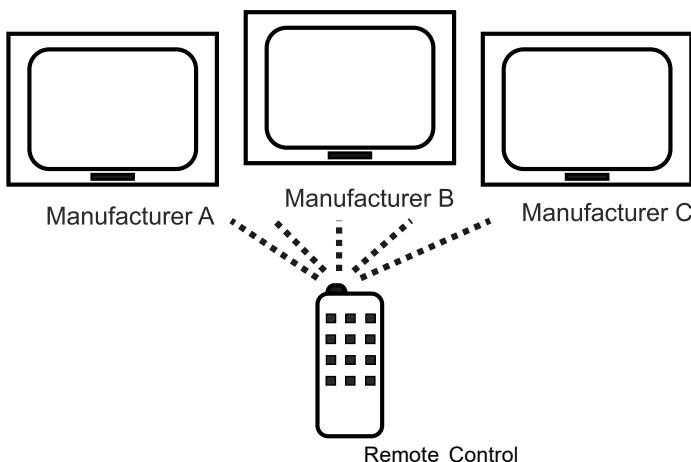
- A single interface to entities of different types
→ Dynamic binding (Java)

- Example:

```
Person p1 = new Person();  
Person p2 = new Employee();  
Person p3 = new Manager();  
// ...  
System.out.println(p1.getDetail());  
System.out.println(p2.getDetail());  
System.out.println(p3.getDetail());
```

Why Polymorphism?

- The ability to hide many different implementations behind a single interface



```

• interface TVInterface {
    public void turnOn();
    public void volumeUp(int steps);
    ...
}

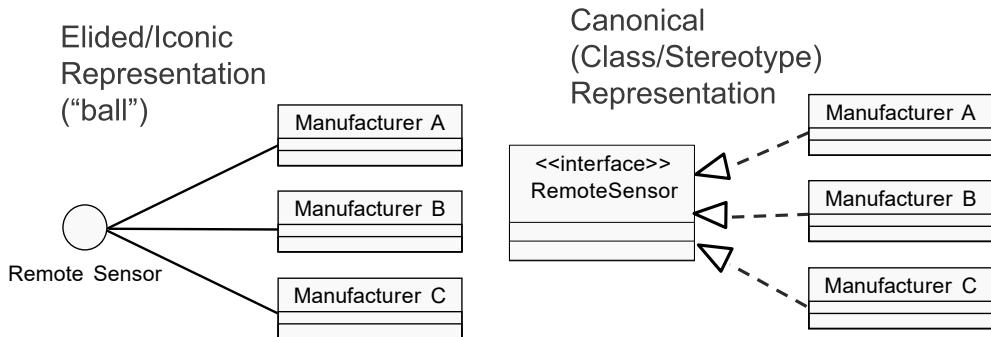
• class TVA implements TVInterface {
    public void turnOn() { ... }
    ...
}

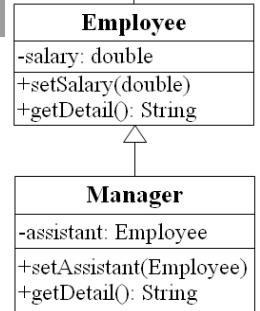
• class TVB implements TVInterface {...}
• class TVC implements TVInterface {...}
• class RemoteControl {
    TVInterface tva = new TVA(); tva.turnOn(); tva.volumeUp(2);
    TVInterface tvb = new TVB(); tvb.turnOn(); tvb.volumeUp(2);
    TVInterface tvc = new TVC(); tvc.turnOn(); tvc.volumeUp(2);
}

```

What Is an Interface?

- A declaration of a coherent set of public features and obligations
 - A contract between providers and consumers of services





Other examples

```

class EmployeeList {
    Employee list[];
    ...
    public void add(Employee e) { ... }
    public void print() {
        for (int i=0; i<list.length; i++) {
            System.out.println(list[i].getDetail());
        }
    }
    ...
    EmployeeList list = new EmployeeList();
    Employee e1; Manager m1;
    ...
    list.add(e1); list.add(m1);
    list.print();
}

```

27

Operator instanceof

```

public class Employee extends Person {}
public class Student extends Person {}

public class Test{
    public doSomething(Person e) {
        if (e instanceof Employee) {...
        } else if (e instanceof Student) {... }
        } else {...}
    }
}

```

Outline

1. Upcasting and Downcasting
2. Static and dynamic bindings
3. Polymorphism
4. Generic programming

4. Generic programming

- Generalizing program so that it can work with different data types, including some future data types
 - Algorithm is already defined
- Example:
 - C: using pointer void
 - C++: using template
 - Java: take advantage of upcasting
 - Java 1.5: Template

Example: C using void pointer

- Memcpy function:

```
void* memcpy(void* region1,
             const void* region2, size_t n) {
    const char* first = (const char*)region2;
    const char* last = ((const char*)region2) + n;
    char* result = (char*)region1;
    while (first != last)
        *result++ = *first++;
    return result;
}
```

Example: C++ using template

When using, we can replace ItemType by int, string,... or any object of any class

```
template<class ItemType>
void sort(ItemType A[], int count ) {
    // Sort count items in the array, A, into increasing order
    // The algorithm that is used here is selection sort
    for (int i = count-1; i > 0; i--) {
        int index_of_max = 0;
        for (int j = 1; j <= i ; j++)
            if (A[j] > A[index_of_max]) index_of_max = j;
        if (index_of_max != i) {
            ItemType temp = A[i];
            A[i] = A[index_of_max];
            A[index_of_max ] = temp;
        }
    }
}
```

Example: Java using upcasting and Object

```
class MyStack {
    ...
    public void push(Object obj) {...}
    public Object pop() {...}
}
public class TestStack{
    MyStack s = new MyStack();
    Point p = new Point();
    Circle c = new Circle();
    s.push(p); s.push(c); //upcasting
    Circle c1 = (Circle) s.pop(); //downcasting
    Point p1 = (Point) s.pop(); //downcasting
}
```

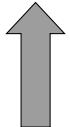
Recall – equals



```
class MyValue {
    private int number;
    public MyValue(int number){this.number = number;}
    public boolean equals(Object obj){

    }
    public int getNumber(){return number;}
}

public class EqualsMethod2 {
    public static void main(String[] args) {
        MyValue v1 = new MyValue(100);
        MyValue v2 = new MyValue(100);
        System.out.println(v1.equals(v2));
        System.out.println(v1==v2);
    }
}
```



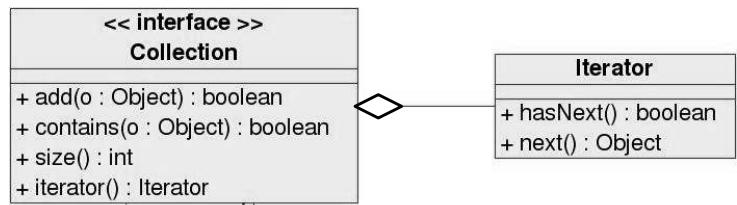
Exercise

- Re-write method **equals** for the class **MyValue** (this method is inherited from the class Object)

35

```
class MyValue {  
    int i;  
    public boolean equals(Object obj) {  
        return (this.i == ((MyValue) obj).i);  
    }  
}  
  
public class EqualsMethod2 {  
    public static void main(String[] args) {  
        MyValue v1 = new MyValue();  
        MyValue v2 = new MyValue();  
        v1.i = v2.i = 100;  
        System.out.println(v1.equals(v2));  
        System.out.println(v1==v2);  
    }  
}
```

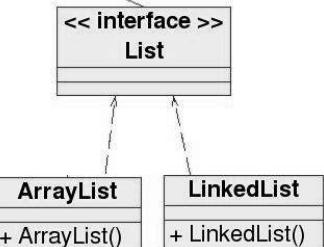
Example: Java 1.5: Template



- Without Template

```

List myList = new LinkedList();
myList.add(new Integer(0));
Integer x = (Integer)
    myList.iterator().next();
  
```



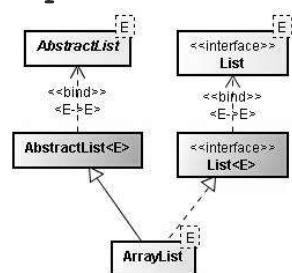
Example: Java 1.5: Template (2)

- Using Template:

```

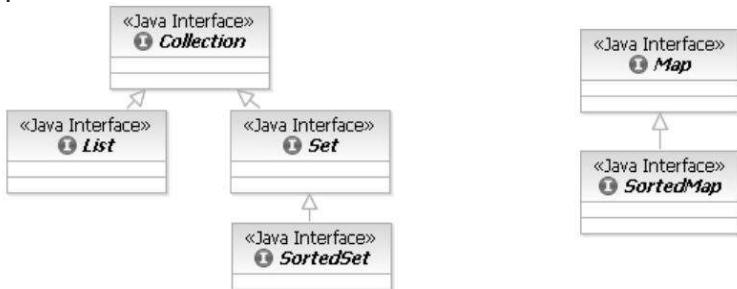
List<Integer> myList = new LinkedList<Integer>();
myList.add(new Integer(0));
Integer x = myList.iterator().next();
  
```

//myList.add(new Long(0)); → Compile error



4.1. Java generic data structure

- Collection: a collection of objects
 - List: a collection of objects that are sequential, consecutive and repeatable
 - Set: a collection of objects that are not repeatable
- Map: Collection of key-value pairs (key is unique)
 - Linking objects in this set to other sets as a dictionary/a telephone book.

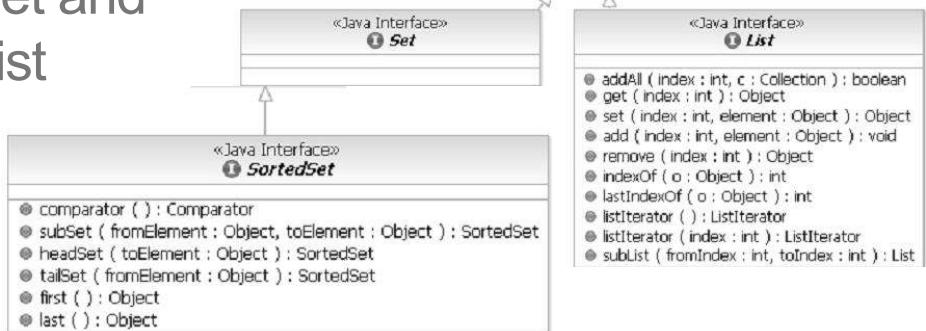


a. Interface of Collection

- Specifies basic interface for manipulating a set of objects
 - Add to collection
 - Remove from collection
 - Check if existing
- Contains methods to manipulate individual objects or a set of objects
- Provide methods to traverse objects in a repeatable collection and convert a collection to an array

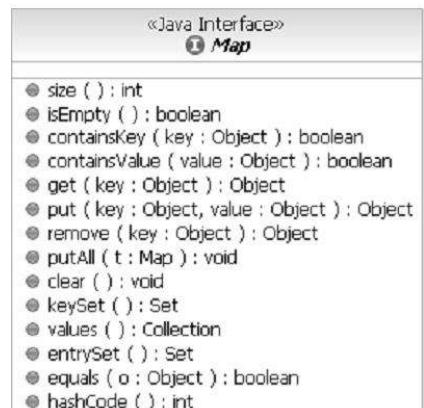
<<Java Interface>>	
&Collection	
●	size () : int
●	isEmpty () : boolean
●	contains (o : Object) : boolean
●	iterator () : Iterator
●	toArray () : Object [*]
●	toArray (a : Object [*]) : Object [*]
●	add (o : Object) : boolean
●	remove (o : Object) : boolean
●	containsAll (c : Collection) : boolean
●	addAll (c : Collection) : boolean
●	removeAll (c : Collection) : boolean
●	retainAll (c : Collection) : boolean
●	clear () : void
●	equals (o : Object) : boolean
●	hashCode () : int

Collection, Set and List



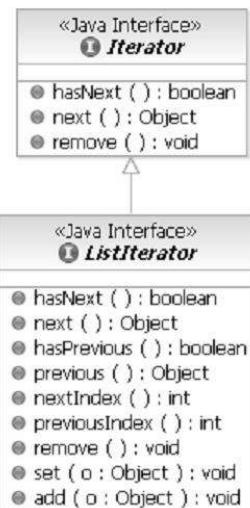
b. Interface of Map

- A basic interface for manipulating a set of pairs key-value
- Add a pair key-value
- Remove a pair key-value
- Get a value of a given key
- Check if existing
(key or value)
- 3 views for the
content of collections:
- Key collection
- Value collection
- Mapping collection of key-value



c. Iterator

- Provide a mechanism to visit (repeat) all the members of a collection
 - Similar to SQL cursor
- ListIterator has methods to show the sequential attribute of the basic list
- Iterator of a sorted collection will visit in the sorting order



Source code for Iterator

```

Collection c;
// Some code to build the collection

Iterator i = c.iterator();
while (i.hasNext()) {
    Object o = i.next();
    // Process this object
}
  
```

Interface and Implementation

- Set<String> mySet = new TreeSet<String>();
- Map<String, Integer> myMap = new HashMap<String, Integer>();

		IMPLEMENTATIONS				
		Hash Table	Resizable Array	Balanced Tree	Linked List	Legacy
I N T E R F A C E s	Set	HashSet		TreeSet		
	List		ArrayList		LinkedList	Vector, Stack
	Map	HashMap		TreeMap		HashTable, Properties

```

public class MapExample {                                         45
    public static void main(String args[]) {
        Map map<String, Integer> = new HashMap<String, Integer>();
        Integer ONE = new Integer(1);
        for (int i=0, n=args.length; i<n; i++) {
            String key = args[i];
            Integer frequency = map.get(key);
            if (frequency == null) { frequency = ONE; }
            else {
                int value = frequency.intValue();
                frequency = new Integer(value + 1);
            }
            map.put(key, frequency);
        }
        System.out.println(map);
        Map sortedMap = new TreeMap(map);
        System.out.println(sortedMap);
    }
}

```

4.2. Defining and using Template

```
class MyStack<T> {  
    ...  
    public void push(T x) { ... }  
    public T pop() {  
        ...  
    }  
}
```

Using template

```
public class Test {  
    public static void main(String args[]) {  
  
        MyStack<Integer> s1 = new MyStack<Integer>();  
        s1.push(new Integer(0));  
        Integer x = s1.pop();  
  
        //s1.push(new Long(0)); → Error  
  
        MyStack<Long> s2 = new MyStack<Long>();  
        s2.push(new Long(0));  
        Long y = s2.pop();  
  
    }  
}
```

Defining Iterator

```
public interface List<E>{
    void add(E x);
    Iterator<E> iterator();
}

public interface Iterator<E>{
    E next();
    boolean hasNext();
}

class LinkedList<E> implements List<E> {
// implementation
}
```

4.3. Wildcard

```
public class Test {
    public static void main(String args[]) {
        List<String> lst0 = new LinkedList<String>();
        //List<Object> lst1 = lst0; → Error
        //printList(lst0); → Error
    }

    void printList(List<Object> lst) {
        Iterator it = lst.iterator();
        while (it.hasNext())
            System.out.println(it.next());
    }
}
```

Example: Using Wildcards

```
public class Test {  
    void printList(List<?> lst) {  
        Iterator it = lst.iterator();  
        while (it.hasNext())  
            System.out.println(it.next());  
    }  
  
    public static void main(String args[]) {  
        List<String> lst0 =  
            new LinkedList<String>();  
        List<Employee> lst1 =  
            new LinkedList<Employee>();  
  
        printList(lst0);    // String  
        printList(lst1);    // Employee  
    }  
}
```

Wildcards of Java 1.5

- "? extends Type": Specifies a set of children types of Type. This is the most useful wildcard.
- "? super Type": Specifies a set of parent types of Type
- "?": Specifies all the types or any types.

Example of wildcard (1)

```
public void printCollection(Collection c) {
    Iterator i = c.iterator();
    for(int k = 0;k<c.size();k++) {
        System.out.println(i.next());
    }
}
```

→ Using wildcard:

```
void printCollection(Collection<?> c) {
    for(Object o:c) {
        System.out.println(o);
    }
}
```

Example of wildcard (2)

```
public void draw(List<Shape> shape) {
    for(Shape s: shape) {
        s.draw(this);
    }
}
```

→ What is the difference compared with:

```
public void draw(List<? extends Shape> shape) {
    // rest of the code is the same
}
```

Template Java 1.5 vs. C++

- Template in Java does not create new classes
- Check the consistency of types when compiling
 - All the objects are basically of the type Object

```
class MyStack<T> {  
    // T a[];  
    Object a[];  
    public void push(T x) {...}  
    public T pop() {  
        Object x;  
        ...  
        return (T) x;  
    }  
}
```

Backward Compatibility

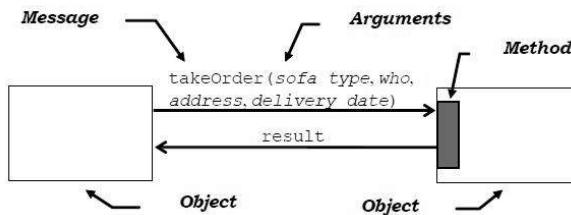
```
public class Test {  
    static public void main(String args[]) {  
  
        MyStack<Integer> s1 = new MyStack<Integer>();  
        s1.push(new Integer(0));  
        Integer x = s1.pop();  
  
        MyStack s2 = new MyStack();  
        s2.push(new Integer(0));  
        s2.push(new Long(1));  
        Long n = (Long) s2.pop();  
    }  
}
```

Function call vs. Message passing

- Call function
 - Indicate the exact piece of code to be executed.
 - Has only an execution of a function with some specific name.
 - There are no functions with the same name
- Message passing
 - **Request a service from an object and the object will decide what to do**
 - **Different objects will have different re-actions/behaviors for a message.**

Message vs. Method

- Message
 - Is sent from an object to another object and does not contain any piece of code to be executed
- Method
 - Method/function in structure programming languages
 - Is an execution of service that is requested in the message
 - Is a piece of code to be executed in order to respond to a message sent to an object



@NGUYỄN Thị Thu Trang, trangntt@soict.hust.edu.vn

OBJECT-ORIENTED LANGUAGE AND THEORY

9. GUI PROGRAMMING

Nguyen Thi Thu Trang
trangntt@soict.hust.edu.vn



Outline

1. GUI Programming in Java

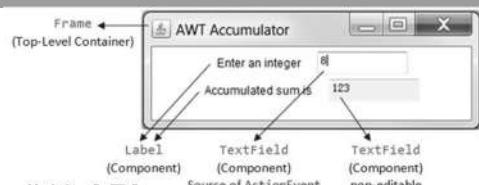
2. AWT

3. Swing

4. JavaFX

AWT and Swing

- **AWT (Abstract Windowing Toolkit) API**
 - From JDK 1.0
 - Most components have **become obsolete** and should be **replaced by Swing** components



- **Swing API**
 - From JDK 1.1, as a part of **Java Foundation Classes (JFC)**
 - A much more comprehensive set of graphics libraries that enhances the AWT
 - JFC consists of **Swing**, **Java2D**, **Accessibility**, **Internationalization**, and **Pluggable Look-and-Feel Support APIs**.



JavaFX

- **JavaFX** is a software platform for creating and delivering desktop applications, as well as **rich internet applications** (RIAs) that can run across a wide variety of devices.
- **JavaFX** is intended to replace **Swing** as the standard **GUI library for Java SE**, but both will be included for the foreseeable future.

*IFX is just a name, which is normally related with sound or visual effects in the javafx i was in the belief that the fx was function. ...
FIPS stands for the Federal Information Processing Standardization*

Which should we choose?



- **AWT:** for simple GUI, but not for comprehensive ones
 - Native OS GUI
 - Platform-independent and device-independent interface
 - Heavyweight components
- **Swing:** Pure Java code with a more robust, versatile, and flexible library
 - Use AWT for windows and event handling
 - Pure-Java GUI, 100% portable and same across platform
 - Most components are light-weight, different look-and-feel
- **JavaFX:** for developing rich Internet applications
 - Can run across a wide variety of devices
 - More consistent in style and has additional options, e.g. 3D, chart, audio, video...



Outline

1. GUI Programming in Java



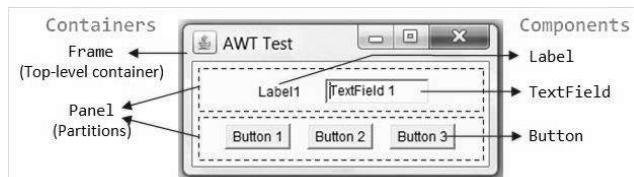
2. AWT

3. Swing

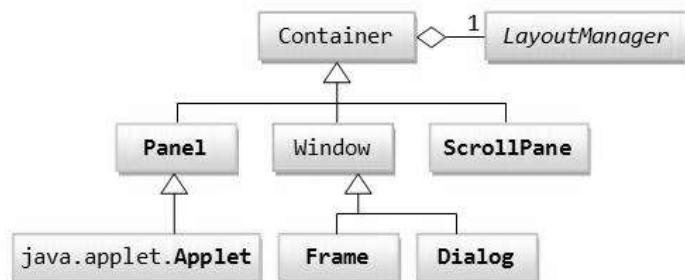
4. JavaFX

AWT Containers and Components

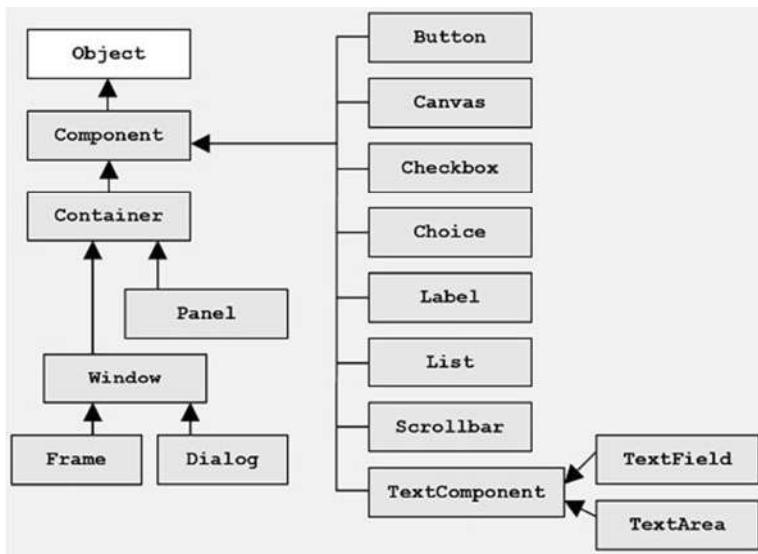
- There are **two types of GUI elements**:
- **Component**: Components are elementary GUI entities (e.g. Button, Label, and TextField.)
- **Container**: Containers (e.g. Frame, Panel and Applet) are used to *hold components in a specific layout* (such as flow or grid). A container can also hold sub-containers.
- **GUI components are also called **controls** (Microsoft ActiveX Control), **widgets** (Eclipse's Standard Widget Toolkit, Google Web Toolkit)**, which allow users to interact with the application through these components (*such as button-click and text-entry*).



Hierarchy of the AWT Container Classes



AWT Component Classes



```

import java.awt.Button;
import java.awt.Frame;
import java.awt.Label;
import java.awt.TextField;
import java.awt.event.ActionListener;

public class AWTCounter
    extends Frame implements ActionListener {
    ...
}

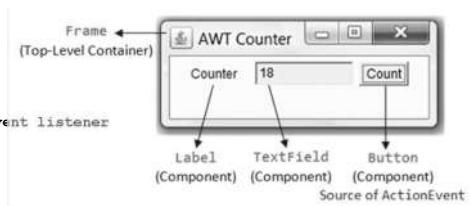
public AWTCounter () {
    ...
}

// Constructor to setup GUI components and event handling
}

// The entry main() method */
public static void main(String[] args) {
    ...
}

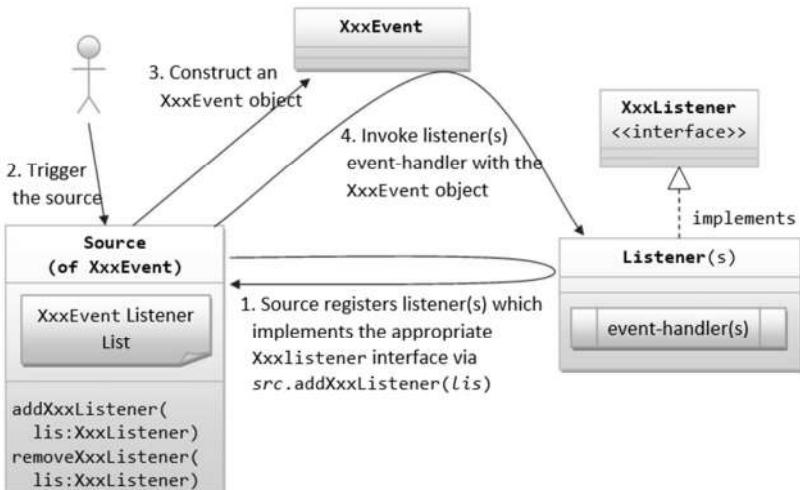
/* ActionEvent handler - Called back upon button-click. */
public void actionPerformed(ActionEvent evt) {
    ...
}
}

```

AWT Event Handling

Event Driven / Event Delegation



E.g. MouseListener (XxxListener) interface

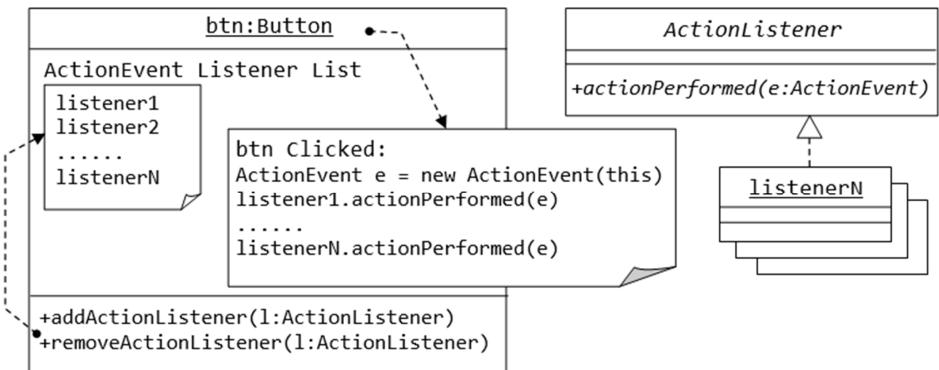
```
//A MouseListener interface, which declares the signature of the handlers
//for the various operational modes.
public interface MouseListener {
    // Called back upon mouse-button pressed
    public void mousePressed(MouseEvent evt);
    // Called back upon mouse-button released
    public void mouseReleased(MouseEvent evt);
    // Called back upon mouse-button clicked (pressed and released)
    public void mouseClicked(MouseEvent evt);
    // Called back when mouse pointer entered the component
    public void mouseEntered(MouseEvent evt);
    // Called back when mouse pointer exited the component
    public void mouseExited(MouseEvent evt);
}
```

Add or remove XxxListener in the source:

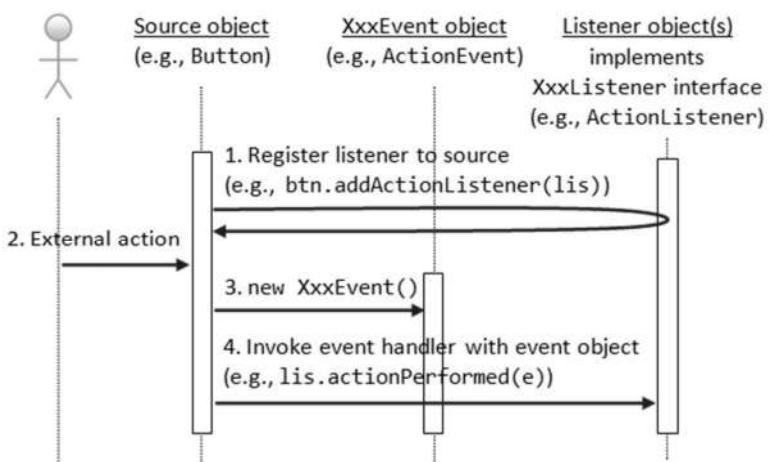
```
public void addXxxListener(XxxListener lis);
public void removeXxxListener(XxxListener lis);
```

```
//An example provides implementation to the event handler methods
class MouseDemo implements MouseListener {
    private Button btn;
    public MouseDemo(){
        ...
        btn.addMouseListener(this);
    }
    @Override
    public void mousePressed(MouseEvent e) {
        System.out.println("Mouse-button pressed!");
    }
    @Override
    public void mouseReleased(MouseEvent e) {
        System.out.println("Mouse-button released!");
    }
    @Override
    public void mouseClicked(MouseEvent e) {
        System.out.println("Mouse-button clicked (pressed and released)!");
    }
    @Override
    public void mouseEntered(MouseEvent e) {
        System.out.println("Mouse-pointer entered the source component!");
    }
    @Override
    public void mouseExited(MouseEvent e) {
        System.out.println("Mouse exited-pointer the source component!");
    }
}
```

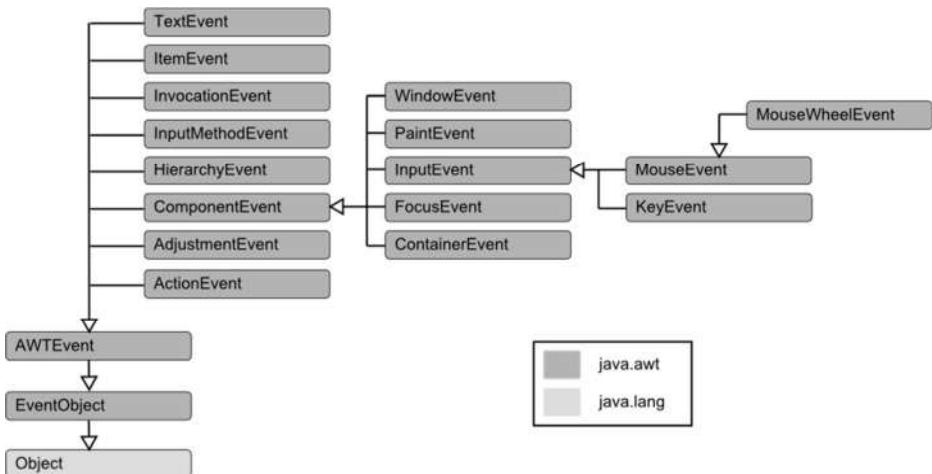
Revisit AWTCounter example



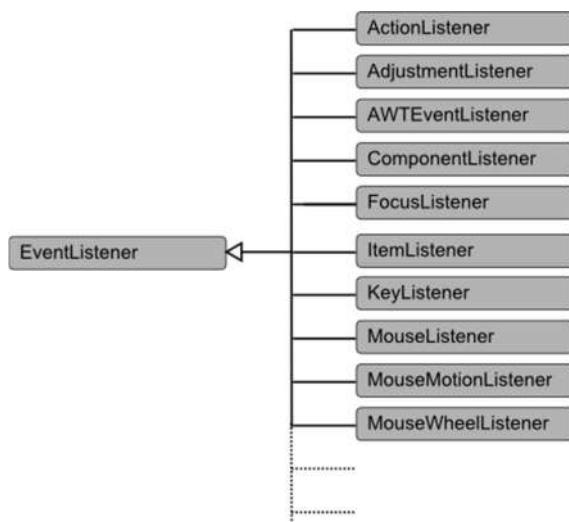
Revisit AWTCounter example



Event Hierarchy



EventListener Hierarchy



Outline

1. GUI Programming in Java

2. AWT

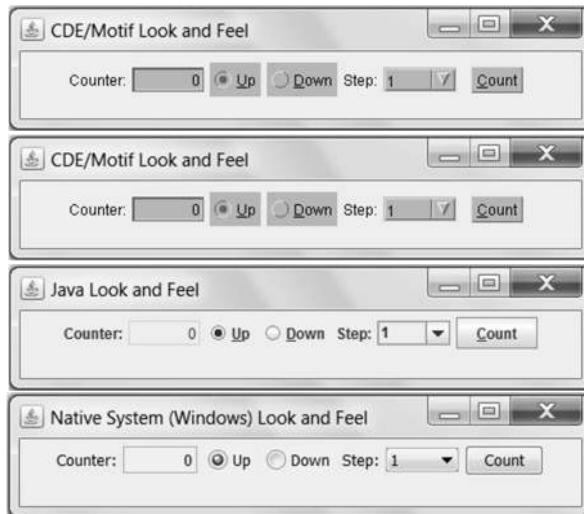
3. Swing

4. JavaFX

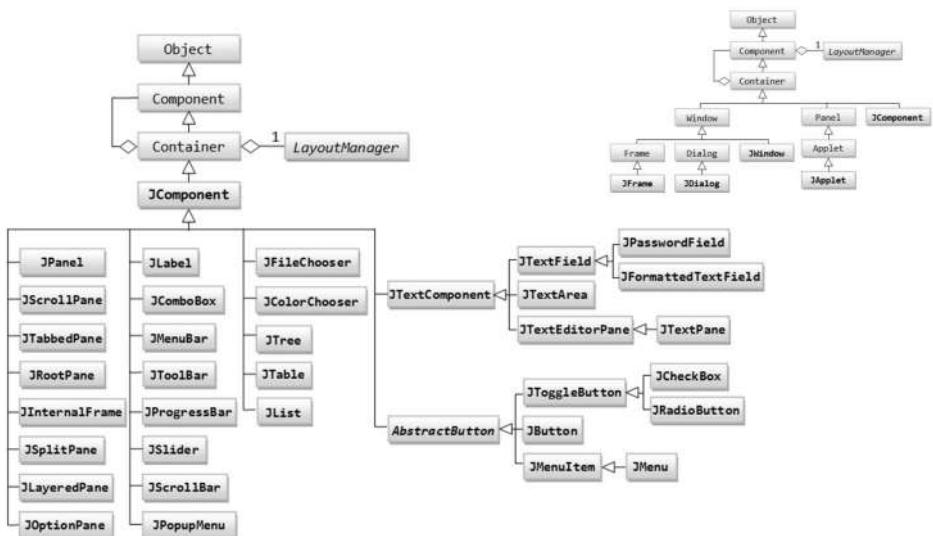
Java Swing

- Light Weight: Pure JAVA code
 - Freelance of native operational System's API
- Use the Swing components with prefix "J", e.g. JFrame, JButton, JTextField, JLabel, etc.
 - Advanced controls like Tree, color picker, table controls, TabbedPane, slider.
- Uses the AWT event-handling classes
- Highly Customizable
 - Often made-to-order in a very simple method as visual appearance is freelance of content.
- Pluggable look-and-feel
 - Modified at run-time, supported by accessible values.

Swing – Different Look & Feel

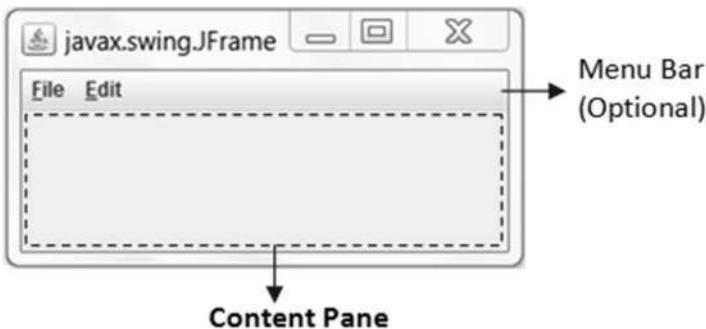


Swing Container and Component



Containers and ContentPane

`javax.swing.JFrame`

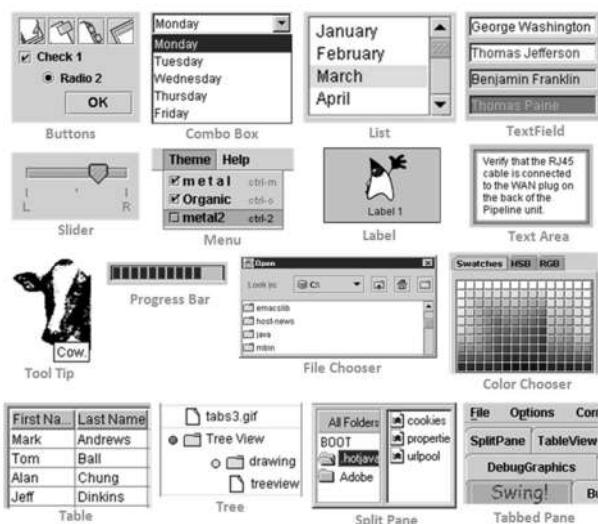


```
Container cp = aJFrame.getContentPane();
aJFrame.setContentPane(aPanel);
```

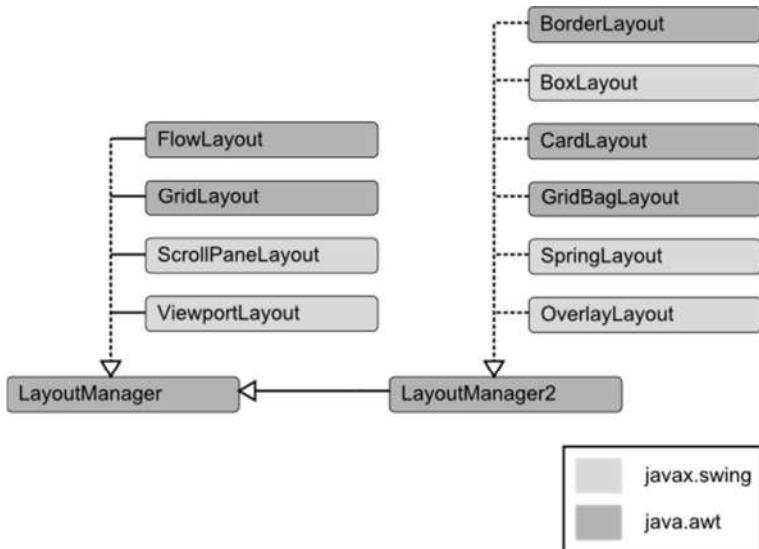
Swing components

Swing is huge
(consists of 18
packages of
737 classes as
in JDK 1.8) and
has great depth

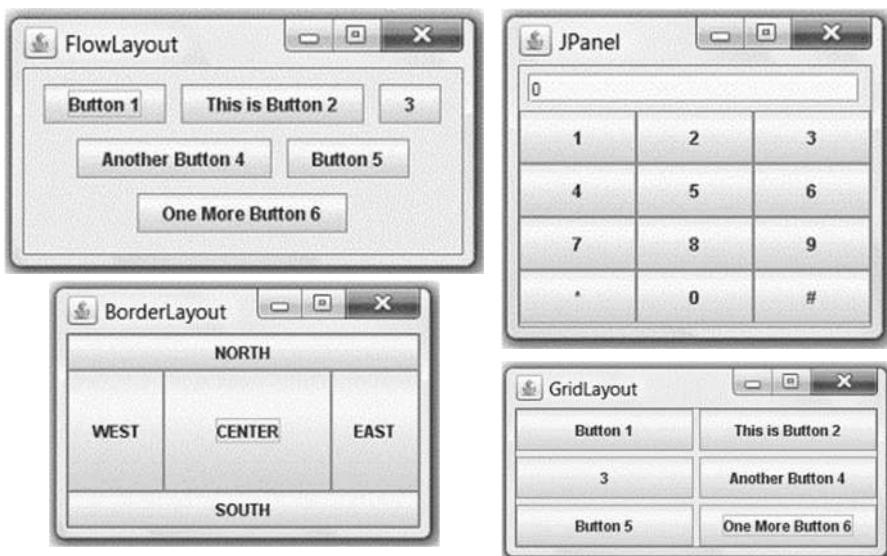
Compare to
AWT:
12 packages of
370 classes



Swing Layout



Layout management



```
public class SwingDemo extends JFrame {  
    // Private instance variables  
    // Constructor to setup the GUI components and event handlers  
    public SwingDemo() {  
        // top-level content-pane from JFrame  
        Container cp = getContentPane();  
        cp.setLayout(new ....Layout());  
  
        // Allocate the GUI components  
        cp.add(....);  
  
        // Source object adds listener  
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        // Exit the program when the close-window button clicked  
        setTitle("....."); // "super" JFrame sets title  
        setSize(300, 150); // "super" JFrame sets initial size  
        setVisible(true); // "super" JFrame shows  
    }  
}
```

(cont.)

```
// The entry main() method  
public static void main(String[] args) {  
    // Run GUI codes in Event-Dispatching thread  
    // for thread-safety  
    SwingUtilities.invokeLater(new Runnable() {  
        @Override  
        public void run() {  
            new SwingDemo();  
        }  
    });  
}
```

Outline

1. GUI Programming in Java
2. AWT
3. Swing
4. JavaFX

Why JavaFX?



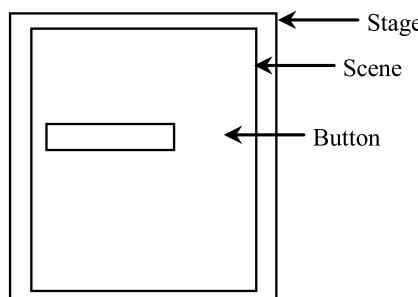
- Additional consistent in its style than Swing
 - Contains WebView supported the popular WebKit browser => Introduce Website within JavaFX
- Design GUI like Web apps with XML and CSS (FXML) than doing in Java code
 - Save building time
- Integrate 3D graphics, charts, and audio, video, and embedded Website inside GUI
 - Easy to develop Game/Media applications
- Light-weight and hardware accelerated

Why JavaFX? (2)

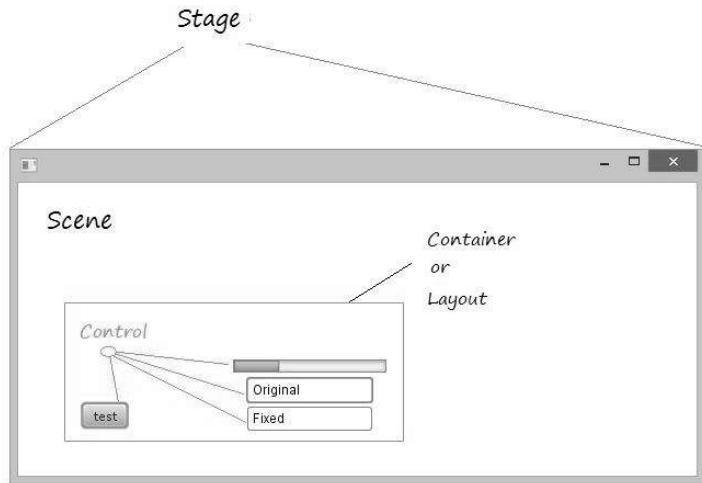
- Support stylish animations
 - Resembling fades, Rotations, Motion ways
 - Custom animations with KeyFrame and Timeline
- Support for modern touch devices
 - Resembling scrolling, swiping, rotating and zooming...
- Many eye-catching controls
 - Collapsible Titled Pane
- Events are higher thought-out and additional consistent

Basic Structure of JavaFX

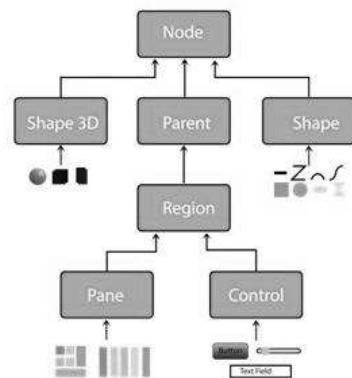
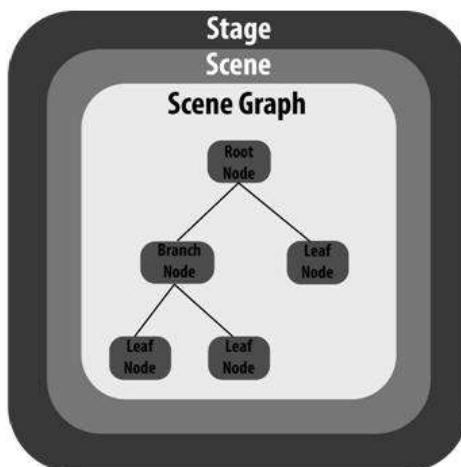
- Application
- Override the start(Stage) method
- Stage, Scene, and Nodes



Basic Structure of JavaFX

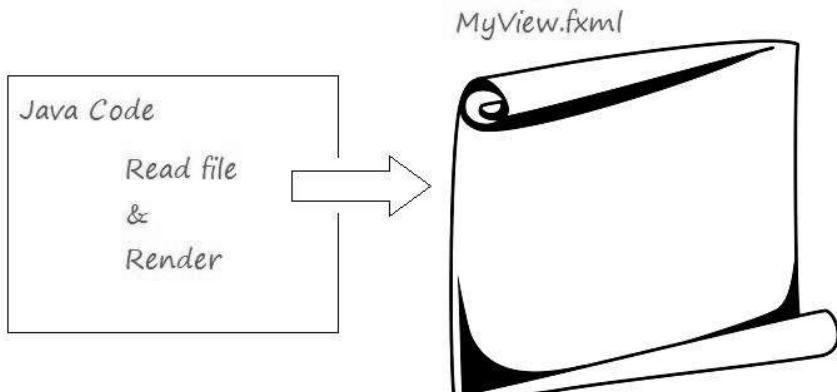


Basic Structure of JavaFX



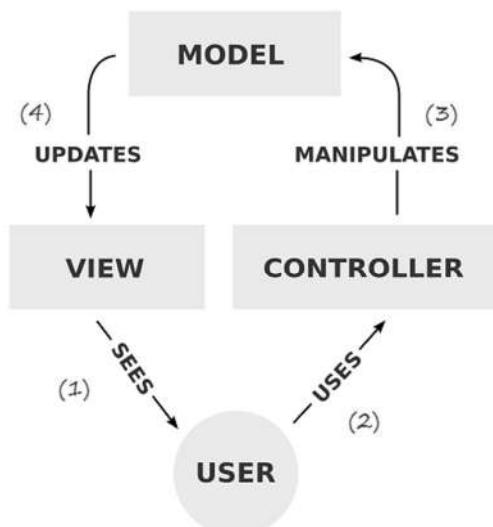
```
3+ import javafx.application.Application;□
4
5
6
7 public class Main extends Application {
8
9     @Override
10    public void start(Stage primaryStage) {
11        try {
12            BorderPane root = new BorderPane();
13            Scene scene = new Scene(root,400,400);
14            scene.getStylesheets().add(getClass().getResource("res
15            primaryStage.setScene(scene);
16            primaryStage.show();
17        } catch(Exception e) {
18            e.printStackTrace();
19        }
20    }
21
22
23    public static void main(String[] args) {
24        launch(args);
25    }
26}
27
```

JavaFX Scene Build



MVC pattern

1. After seeing it on VIEW
2. Users use CONTROLLER
3. Manipulate data (Update, modify, delete, ..), the data on MODEL has been changed.
4. Displaying the data of MODEL on VIEW.



View: FXML

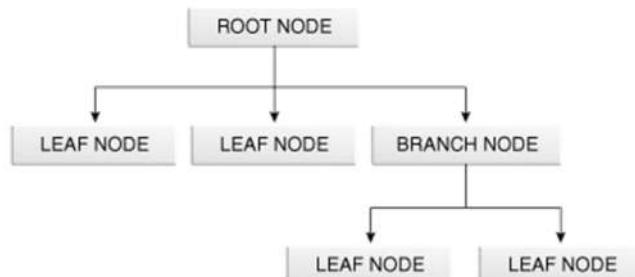
```

MyScene.fxml
1 <?xml version="1.0" encoding="UTF-8"?>
2
3 <?import javafx.scene.text.*?>
4 <?import javafx.scene.control.*?>
5 <?import java.lang.*?>
6 <?import javafx.scene.layout.*?>
7 <?import javafx.scene.layout.AnchorPane?>
8
9<AnchorPane prefHeight="238.0" prefWidth="269.0"
10      xmlns="http://javafx.com/javafx/8"
11      xmlns:fx="http://javafx.com/fxml/1"
12      fx:controller="org.o7planning.javafx.MyController">
13
14<children>
15
16    <Button fx:id="myButton" layoutX="51.0" layoutY="44.0"
17              mnemonicParsing="false"
18              onAction="showDateTime" text="Show Date Time" />
19
20    <TextField fx:id="myTextField" layoutX="28.0"
21              layoutY="107.0" prefHeight="25.0" prefWidth="201.0" />
22
23  </children>
24
25 </AnchorPane>
  
```

```
public class MyController implements Initializable {  
    @FXML  
    private Button myButton;  
    @FXML  
    private TextField myTextField;  
  
    // When user click on myButton, this method will be called  
    public void showDateTime(ActionEvent event) {  
        System.out.println("Button Clicked!");  
        Date now= new Date();  
        DateFormat df = new SimpleDateFormat(  
            "dd-MM-yyyy HH:mm:ss.SSS");  
        // Model Data  
        String dateTimeString = df.format(now);  
        // Show in VIEW  
        myTextField.setText(dateTimeString);  
    }  
}
```

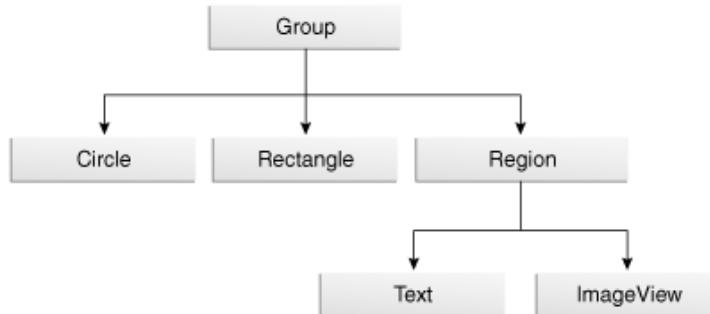
JavaFX Architecture

- A JavaFX user interface is based on a scene graph, which is a tree, much like an html document. To review, the CS conception of a tree looks like this:



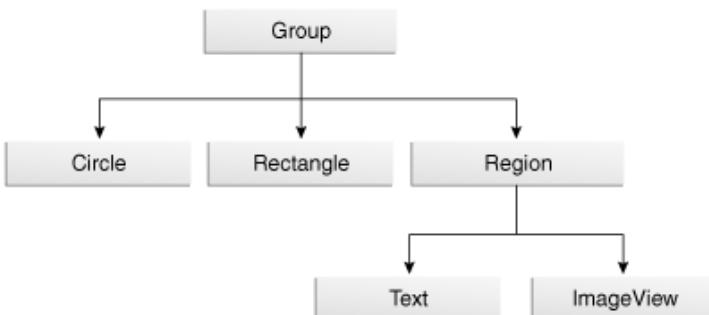
JavaFX Architecture: Example

- In JavaFX, the root of the scene graph tree is the pane.

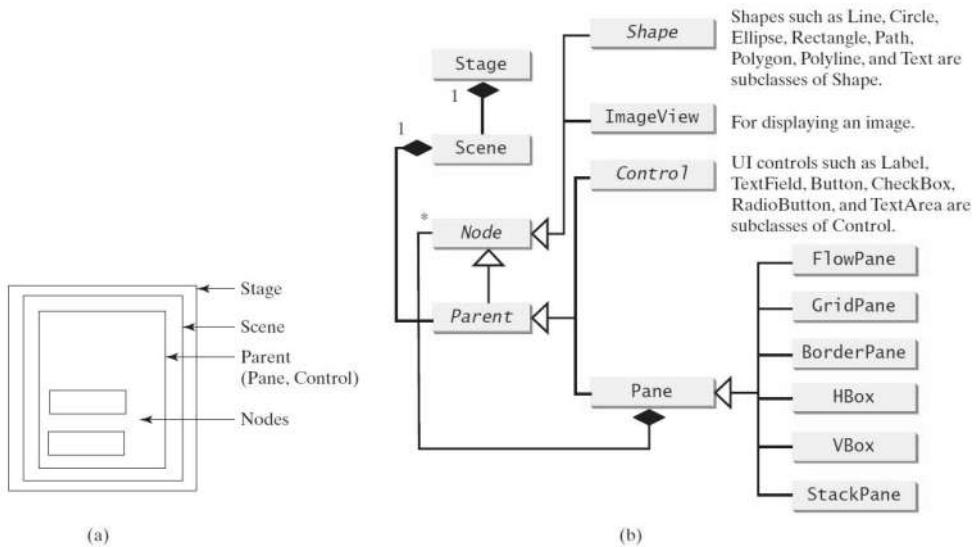


JavaFX Architecture

- In JavaFX, the root of the scene graph tree is the pane

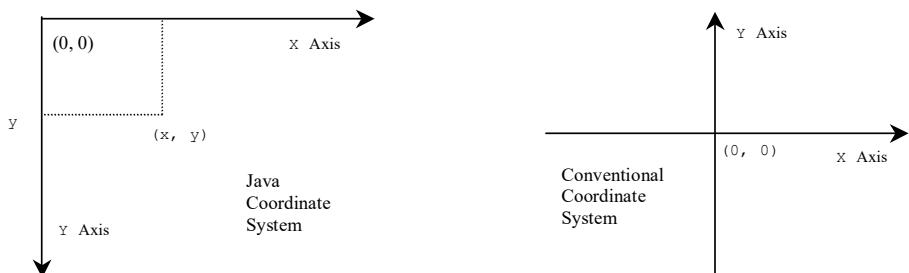


Panes, UI Controls, and Shapes



Display a Shape

This example displays a circle in the center of the pane.



Binding Properties

JavaFX introduces a new concept called *binding property* that enables a *target object* to be bound to a *source object*. If the value in the source object changes, the target property is also changed automatically. The target object is simply called a *binding object* or a *binding property*.

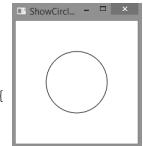
```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.stage.Stage;

public class ShowCircleCentered extends Application {
    public void start(Stage primaryStage) {
        // Create a pane to hold the circle
        Pane pane = new Pane();
        // Create a circle and set its properties
        Circle circle = new Circle();
        circle.centerXProperty().bind(pane.widthProperty().divide(2));
        circle.centerYProperty().bind(pane.heightProperty().divide(2));
        circle.setRadius(50);
        circle.setStroke(Color.BLACK);
        circle.setFill(Color.WHITE);
        pane.getChildren().add(circle); // Add circle to the pane

        // Create a scene and place it in the stage
        Scene scene = new Scene(pane, 200, 200);
        primaryStage.setTitle("ShowCircleCentered"); //Set the stage title
        primaryStage.setScene(scene); // Place the scene in the stage
        primaryStage.show(); // Display the stage
    }

    /**
     * The main method is only needed for the IDE with limited
     * JavaFX support. Not needed for running from the command line.
     */
    public static void main(String[] args) {
        launch(args);
    }
}
```

ShowCircleCentered



45

Example: Binding Properties

Unidirectional Binding

```
label1.textProperty().bind(text1.textProperty());
label2.textProperty().bind(text2.textProperty());
```



Bidirectional Binding

```
text1.textProperty().
    bindBidirectional(text2.textProperty());
```

The Color Class

<code>javafx.scene.paint.Color</code>
<code>-red: double</code>
<code>-green: double</code>
<code>-blue: double</code>
<code>-opacity: double</code>
<code>+Color(r: double, g: double, b: double, opacity: double)</code>
<code>+brighter(): Color</code>
<code>+darker(): Color</code>
<code>+color(r: double, g: double, b: double): Color</code>
<code>+color(r: double, g: double, b: double, opacity: double): Color</code>
<code>+rgb(r: int, g: int, b: int): Color</code>
<code>+rgb(r: int, g: int, b: int, opacity: double): Color</code>

The getter methods for property values are provided in the class, but omitted in the UML diagram for brevity.

- The red value of this `Color` (between 0.0 and 1.0).
- The green value of this `Color` (between 0.0 and 1.0).
- The blue value of this `Color` (between 0.0 and 1.0).
- The opacity of this `Color` (between 0.0 and 1.0).
- Creates a `Color` with the specified red, green, blue, and opacity values.
- Creates a `Color` that is a brighter version of this `Color`.
- Creates a `Color` that is a darker version of this `Color`.
- Creates an opaque `Color` with the specified red, green, and blue values.
- Creates a `Color` with the specified red, green, blue, and opacity values.
- Creates a `Color` with the specified red, green, and blue values in the range from 0 to 255.
- Creates a `Color` with the specified red, green, and blue values in the range from 0 to 255 and a given opacity.

The Font Class

<code>javafx.scene.text.Font</code>
<code>-size: double</code>
<code>-name: String</code>
<code>-family: String</code>
<code>+Font(size: double)</code>
<code>+Font(name: String, size: double)</code>
<code>+font(name: String, size: double)</code>
<code>+font(name: String, w: FontWeight, size: double)</code>
<code>+font(name: String, w: FontWeight, p: FontPosture, size: double)</code>
<code>+getFamilies(): List<String></code>
<code>+getFontNames(): List<String></code>

The getter methods for property values are provided in the class, but omitted in the UML diagram for brevity.

- The size of this `Font`.
- The name of this `Font`.
- The family of this `Font`.
- Creates a `Font` with the specified size.
- Creates a `Font` with the specified full font name and size.
- Creates a `Font` with the specified name and size.
- Creates a `Font` with the specified name, weight, and size.
- Creates a `Font` with the specified name, weight, posture, and size.
- Returns a list of font family names.
- Returns a list of full font names including family and weight.



FontDemo

Run

The Image Class

`javafx.scene.image.Image`

- error: ReadOnlyBooleanProperty
- height: ReadOnlyBooleanProperty
- width: ReadOnlyBooleanProperty
- progress: ReadOnlyBooleanProperty
- +Image(filenameOrURL: String)

The getter methods for property values are provided in the class, but omitted in the UML diagram for brevity.

Indicates whether the image is loaded correctly?

The height of the image.

The width of the image.

The approximate percentage of image's loading that is completed.

Creates an `Image` with contents loaded from a file or a URL.

The ImageView Class

`javafx.scene.image.ImageView`

- fitHeight: DoubleProperty
- fitWidth: DoubleProperty
- x: DoubleProperty
- y: DoubleProperty
- image: ObjectProperty<Image>
- +ImageView()
- +ImageView(image: Image)
- +ImageView(filenameOrURL: String)

The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

The height of the bounding box within which the image is resized to fit.

The width of the bounding box within which the image is resized to fit.

The x-coordinate of the ImageView origin.

The y-coordinate of the ImageView origin.

The image to be displayed in the image view.

Creates an `ImageView`.

Creates an `ImageView` with the specified image.

Creates an `ImageView` with image loaded from the specified file or URL.



ShowImage

Run

Layout Panes

JavaFX provides many types of panes for organizing nodes in a container.

Class	Description
Pane	Base class for layout panes. It contains the <code>getChildren()</code> method for returning a list of nodes in the pane.
StackPane	Places the nodes on top of each other in the center of the pane.
FlowPane	Places the nodes row-by-row horizontally or column-by-column vertically.
GridPane	Places the nodes in the cells in a two-dimensional grid.
BorderPane	Places the nodes in the top, right, bottom, left, and center regions.
HBox	Places the nodes in a single row.
VBox	Places the nodes in a single column.

FlowPane

`javafx.scene.layout.FlowPane`

```
-alignment: ObjectProperty<Pos>
-orientation: ObjectProperty<Orientation>
-hgap: DoubleProperty
-vgap: DoubleProperty

+FlowPane()
+FlowPane(hgap: double, vgap: double)
+FlowPane(orientation: ObjectProperty<Orientation>)
+FlowPane(orientation: ObjectProperty<Orientation>, hgap: double, vgap: double)
```

The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

The overall alignment of the content in this pane (default: `Pos.LEFT`). The orientation in this pane (default: `Orientation.HORIZONTAL`).

The horizontal gap between the nodes (default: 0).
The vertical gap between the nodes (default: 0).

Creates a default `FlowPane`.
Creates a `FlowPane` with a specified horizontal and vertical gap.
Creates a `FlowPane` with a specified orientation.
Creates a `FlowPane` with a specified orientation, horizontal gap and vertical gap.



ShowFlowPane

Run

GridPane

`javafx.scene.layout.GridPane`

```
-alignment: ObjectProperty<Pos>
-gridLinesVisible:
    BooleanProperty
-hgap: DoubleProperty
-vgap: DoubleProperty

+GridPane()
+add(child: Node, columnIndex: int, rowIndex: int): void
+addColumn(columnIndex: int, children: Node...): void
+addRow(rowIndex: int, children: Node...): void
+getRowIndex(child: Node): int
+setRowIndex(child: Node, columnIndex: int)
+getRowIndex(child: Node): int
+setRowIndex(child: Node, rowIndex: int): void
+setHalignment(child: Node, value: HPos): void
+setValignment(child: Node, value: VPos): void
```

The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

The overall alignment of the content in this pane (default: `Pos.LEFT`).
Is the grid line visible? (default: `false`)

The horizontal gap between the nodes (default: 0).
The vertical gap between the nodes (default: 0).

Creates a `GridPane`.

Adds a node to the specified column and row.

Adds multiple nodes to the specified column.

Adds multiple nodes to the specified row.

Returns the column index for the specified node.

Sets a node to a new column. This method repositions the node.

Returns the row index for the specified node.

Sets a node to a new row. This method repositions the node.

Sets the horizontal alignment for the child in the cell.

Sets the vertical alignment for the child in the cell.

[ShowGridPane](#)

[Run](#)

BorderPane

`javafx.scene.layout.BorderPane`

```
-top: ObjectProperty<Node>
-right: ObjectProperty<Node>
-bottom: ObjectProperty<Node>
-left: ObjectProperty<Node>
-center: ObjectProperty<Node>

+BorderPane()
+setAlignment(child: Node, pos: Pos)
```

The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

The node placed in the top region (default: `null`).
The node placed in the right region (default: `null`).
The node placed in the bottom region (default: `null`).
The node placed in the left region (default: `null`).
The node placed in the center region (default: `null`).

Creates a `BorderPane`.

Sets the alignment of the node in the `BorderPane`.



[ShowBorderPane](#)

[Run](#)

HBox

`javafx.scene.layout.HBox`

```
-alignment: ObjectProperty<Pos>
-fillHeight: BooleanProperty
-spacing: DoubleProperty

+HBox()
+HBox(spacing: double)
+setMargin(node: Node, value: Insets): void
```

The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

The overall alignment of the children in the box (default: Pos.TOP_LEFT).
Is resizable children fill the full height of the box (default: true).
The horizontal gap between two nodes (default: 0).

Creates a default HBox.

Creates an HBox with the specified horizontal gap between nodes.
Sets the margin for the node in the pane.

VBox

`javafx.scene.layout.VBox`

```
-alignment: ObjectProperty<Pos>
-fillWidth: BooleanProperty
-spacing: DoubleProperty

+VBox()
+VBox(spacing: double)
+setMargin(node: Node, value: Insets): void
```

The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

The overall alignment of the children in the box (default: Pos.TOP_LEFT).
Is resizable children fill the full width of the box (default: true).
The vertical gap between two nodes (default: 0).

Creates a default VBox.

Creates a VBox with the specified horizontal gap between nodes.
Sets the margin for the node in the pane.

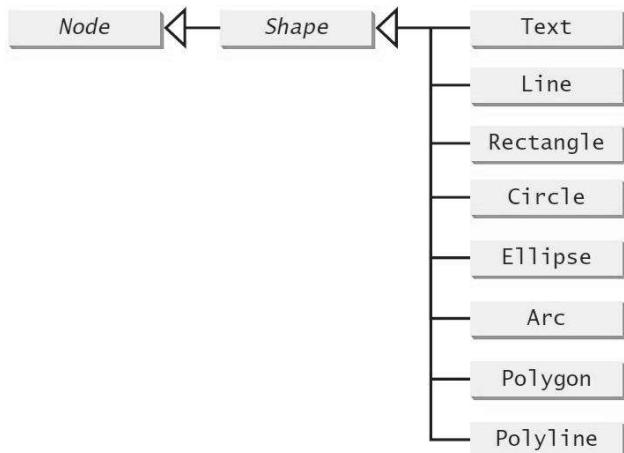


ShowHBoxVBox

Run

Shapes

JavaFX provides many shape classes for drawing texts, lines, circles, rectangles, ellipses, arcs, polygons, and polylines.



Text

`javafx.scene.text.Text`

```

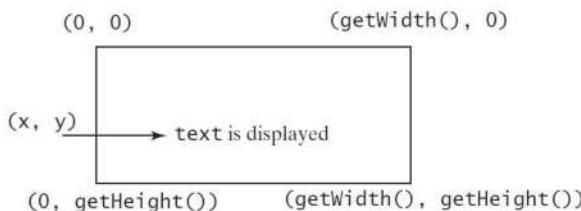
-text: StringProperty
-x: DoubleProperty
-y: DoubleProperty
-underline: BooleanProperty
-strikethrough: BooleanProperty
-font: ObjectProperty<Font>

+Text()
+Text(text: String)
+Text(x: double, y: double,
      text: String)
  
```

The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

<p>-text: StringProperty Defines the text to be displayed.</p> <p>-x: DoubleProperty Defines the x-coordinate of text (default 0).</p> <p>-y: DoubleProperty Defines the y-coordinate of text (default 0).</p> <p>-underline: BooleanProperty Defines if each line has an underline below it (default <code>false</code>).</p> <p>-strikethrough: BooleanProperty Defines if each line has a line through it (default <code>false</code>).</p> <p>-font: ObjectProperty Defines the font for the text.</p>	<p>+Text() Creates an empty Text.</p> <p>+Text(text: String) Creates a Text with the specified text.</p> <p>+Text(x: double, y: double, text: String) Creates a Text with the specified x-, y-coordinates and text.</p>
--	---

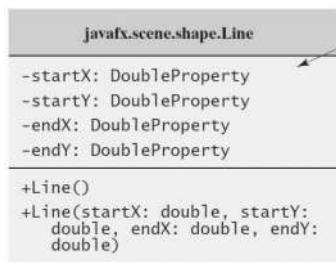
Text Example



(b) Three Text objects are displayed



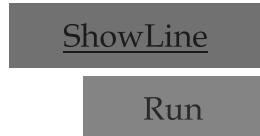
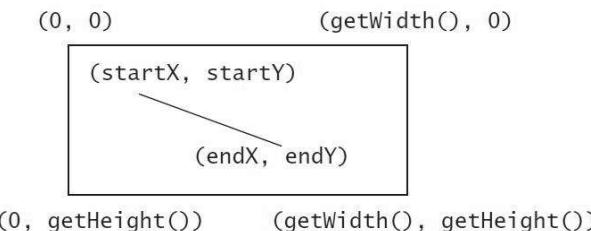
Line



The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

The x-coordinate of the start point.
The y-coordinate of the start point.
The x-coordinate of the end point.
The y-coordinate of the end point.

Creates an empty Line.
Creates a Line with the specified starting and ending points.



Rectangle

`javafx.scene.shape.Rectangle`

-x: DoubleProperty
 -y: DoubleProperty
 -width: DoubleProperty
 -height: DoubleProperty
 -arcWidth: DoubleProperty
 -arcHeight: DoubleProperty

+Rectangle()
 +Rectanlge(x: double, y: double, width: double, height: double)

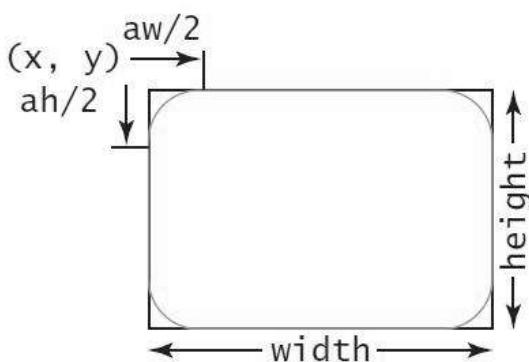
The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

The x-coordinate of the upper-left corner of the rectangle (default 0).
 The y-coordinate of the upper-left corner of the rectangle (default 0).
 The width of the rectangle (default: 0).
 The height of the rectangle (default: 0).
 The `arcWidth` of the rectangle (default: 0). `arcWidth` is the horizontal diameter of the arcs at the corner (see Figure 14.31a).
 The `arcHeight` of the rectangle (default: 0). `arcHeight` is the vertical diameter of the arcs at the corner (see Figure 14.31a).

Creates an empty `Rectangle`.

Creates a `Rectangle` with the specified upper-left corner point, width, and height.

Rectangle Example



(a) `Rectangle(x, y, w, h)`



ShowRectangle

Run

Circle

`javafx.scene.shape.Circle`

-centerX: DoubleProperty
 -centerY: DoubleProperty
 -radius: DoubleProperty

+Circle()
 +Circle(x: double, y: double)
 +Circle(x: double, y: double,
 radius: double)

The getter and setter methods for property values
 and a getter for property itself are provided in the class,
 but omitted in the UML diagram for brevity.

The x-coordinate of the center of the circle (default 0).
 The y-coordinate of the center of the circle (default 0).
 The radius of the circle (default: 0).

Creates an empty `Circle`.

Creates a `Circle` with the specified center.

Creates a `Circle` with the specified center and radius.

Ellipse

`javafx.scene.shape.Ellipse`

-centerX: DoubleProperty
 -centerY: DoubleProperty
 -radiusX: DoubleProperty
 -radiusY: DoubleProperty

+Ellipse()
 +Ellipse(x: double, y: double)
 +Ellipse(x: double, y: double,
 radiusX: double, radiusY:
 double)

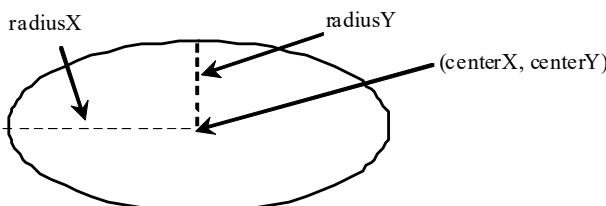
The getter and setter methods for property values
 and a getter for property itself are provided in the class,
 but omitted in the UML diagram for brevity.

The x-coordinate of the center of the ellipse (default 0).
 The y-coordinate of the center of the ellipse (default 0).
 The horizontal radius of the ellipse (default: 0).
 The vertical radius of the ellipse (default: 0).

Creates an empty `Ellipse`.

Creates an `Ellipse` with the specified center.

Creates an `Ellipse` with the specified center and radii.



ShowEllipse

Run

Arc

`javafx.scene.shape.Arc`

```
-centerX: DoubleProperty
-centerY: DoubleProperty
-radiusX: DoubleProperty
-radiusY: DoubleProperty
-startAngle: DoubleProperty
-length: DoubleProperty
-type: ObjectProperty<ArcType>

+Arc()
+Arc(x: double, y: double,
      radiusX: double, radiusY:
      double, startAngle: double,
      length: double)
```

The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

The x-coordinate of the center of the ellipse (default 0).

The y-coordinate of the center of the ellipse (default 0).

The horizontal radius of the ellipse (default: 0).

The vertical radius of the ellipse (default: 0).

The start angle of the arc in degrees.

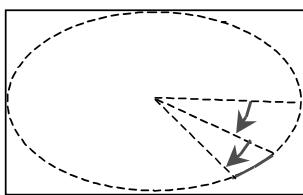
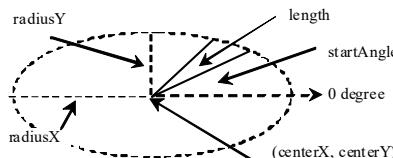
The angular extent of the arc in degrees.

The closure type of the arc (`ArcType.OPEN`, `ArcType.CHORD`, `ArcType.ROUND`).

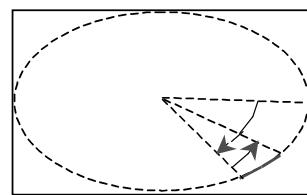
Creates an empty `Arc`.

Creates an `Arc` with the specified arguments.

Arc Examples



(a) Negative starting angle -30° and negative spanning angle -20°



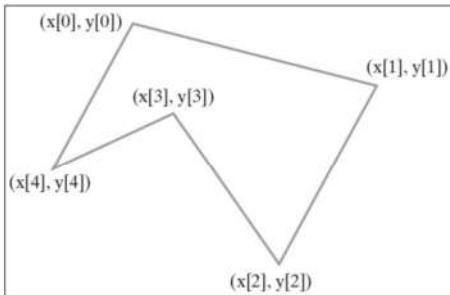
(b) Negative starting angle -50° and positive spanning angle 20°



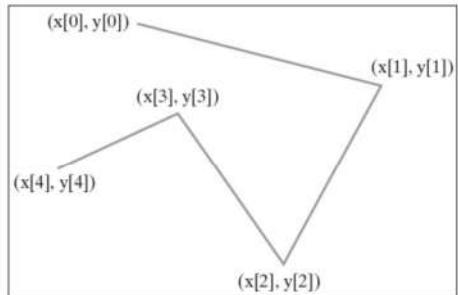
ShowArc

Run

Polygon and Polyline



(a) Polygon



(b) Polyline

ShowArc

Run

Polygon

```
javafx.scene.shape.Polygon
+Polygon()
+Polygon(double... points)
+getPoints():
    ObservableList<Double>
```

The `getter` and `setter` methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

Creates an empty polygon.

Creates a polygon with the given points.

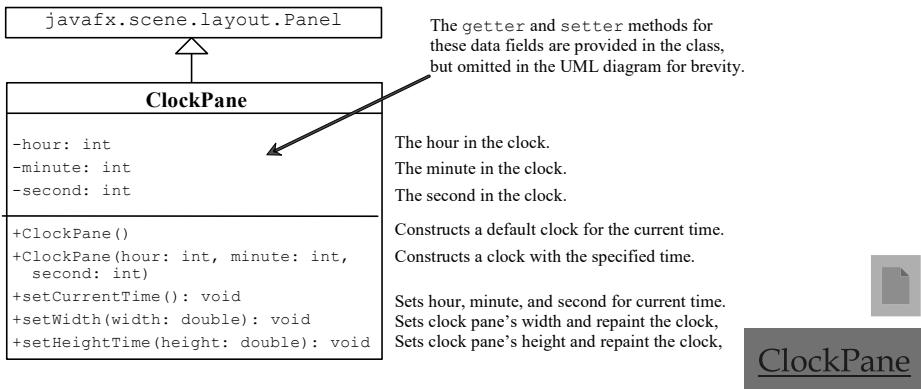
Returns a list of double values as x- and y-coordinates of the points.

ShowPolygon

Run

Case Study: The ClockPane Class

This case study develops a class that displays a clock on a pane.



@NGUYỄN Thị Thu Trang, trangntt@soict.hust.edu.vn

OBJECT-ORIENTED LANGUAGE AND THEORY

10. EXCEPTION AND EXCEPTION HANDLER

Nguyen Thi Thu Trang
trangntt@soict.hust.edu.vn



Outline



1. Exceptions

2. Catching and handling exceptions
3. Exception delegation
4. User-defined exceptions

1.1. What is exception?

- Exception = Exceptional event
- Definition: An exception is an event that occurs in the **execution** of a program and it **breaks** the expected flow of the program.

Example: $4 / 0 =$ 

1.1. What is exception? (2)

- Exception is an particular error
 - Unexpected results
- When an exception occurs, if it is not handled, the program will exit immediately and the control is returned to the OS



1.2. Classical Error Handler

- Writing handling codes where errors occur
 - Making programs more complex
 - Not always have enough information to handle
 - Some errors are not necessary to handle
- Sending status to upper levels
 - Via arguments, return values or global variables (flag)
 - Easy to mis-understand
 - Still hard to understand

Example

```
int devide(int num, int denom, int *error)
{
    if (denom != 0) {
        *error = 0;
        return num/denom;
    } else {
        *error = 1;
        return 0;
    }
}
```

Disadvantages

- Difficult to control all cases
 - Arithmetic errors, memory errors,...
- Developers often forget to handle errors
 - Human
 - Lack of experience, deliberately ignore

Outline

1. Exceptions

2. Catching and handling exceptions

3. Exception delegation

4. User-defined exceptions

2.1. Goals of exception handling

- Making programs more reliable, avoiding unexpected termination
- Separating blocks of code that might cause exceptions and blocks of code that handle exceptions

```
.....  
IF B IS ZERO GO TO ERROR  
C = A/B  
PRINT C  
GO TO EXIT
```

```
ERROR:  
DISPLAY "DIVISION BY ZERO"
```

} Error handling block

```
EXIT:  
END
```

Separating code

- Classic programming: `readFile()` function: not separate the main logic processing and error handling.

```
errorCodeType readFile() {
    initialize errorCode = 0;
    open the file;
    if (theFileIsOpen) {
        determine the length of the file;
        if (gotTheFileLength) {
            allocate that much memory;
            if (gotEnoughMemory) {
                read the file into memory;
                if (readFailed) {
                    errorCode = -1;
                }
            } else {
                errorCode = -2;
            }
        }
    } else {
        errorCode = -3;
    }
    close the file;
    if (theFileDidntClose && errorCode == 0) {
        errorCode = -4;
    } else {
        errorCode = errorCode and -4;
    }
} else {
    errorCode = -5;
}
return errorCode;
}
```

Classic Programming

```
} else {
    errorCode = -3;
}
close the file;
if (theFileDidntClose && errorCode == 0) {
    errorCode = -4;
} else {
    errorCode = errorCode and -4;
}
} else {
    errorCode = -5;
}
return errorCode;
}
```

Exception Handling

- Exception mechanism allows focusing on writing code for the main thread and then handling exception in another place

```
readFile() {
    try {
        open the file;
        determine its size;
        allocate that much memory;
        read the file into memory;
        close the file;
    } catch (fileOpenFailed) {
        doSomething;
    } catch (sizeDeterminationFailed) {
        doSomething;
    } catch (memoryAllocationFailed) {
        doSomething;
    } catch (readFailed) {
        doSomething;
    } catch (fileCloseFailed) {
        doSomething;
    }
}
```

2.2. Models for handling exceptions

- Object oriented approach

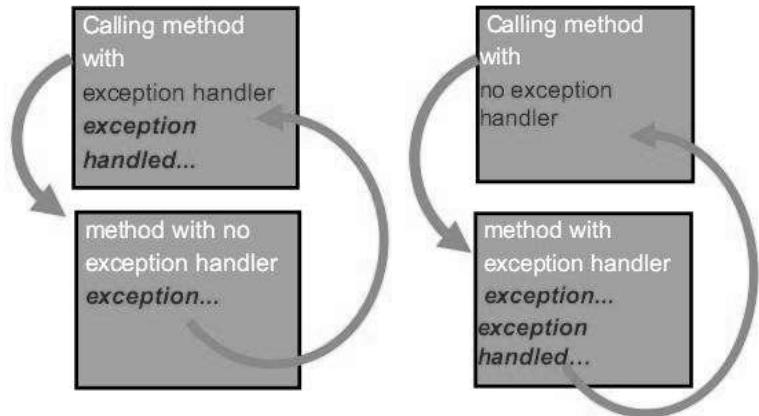
- Packing unexpected conditions in **an object**
- When an exception occurs, the object corresponding to the exception is created and stores all the detailed information about the exception
- Providing an efficient mechanism in handling errors
- Separating irregular control threads with regular threads

```
float sales = getSales();
int staffsize = getStaff().size;
float avg_sales = sales/staffsize;
System.out.println(avg_sales);
```

handler

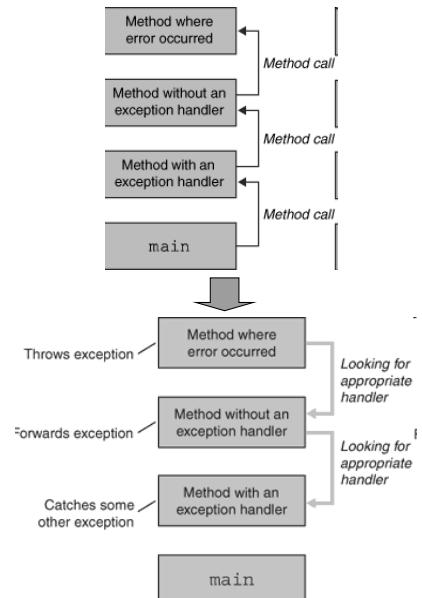
2.2. Models for handling exceptions (2)

- Exceptions need to be handled at the method that causes the exceptions or delegated to its caller method



2.3. Exception handling in Java

- Java has a strong mechanism for handling exceptions
- Exception handling in Java is done via object-oriented model:
 - All the exceptions are representations of a class derived from the class Throwable or its child classes
 - These objects must send the information of exceptions (type and status of the program) from the exceptions place to where they are controlled/handled



2.3. Exception handling in Java (2)

- Key words

- **try**
- **catch**
- **finally**
- **throw**
- **throws**

2.3.1. try/catch block

- try ... catch block: Separating the regular block of program and the block for handling exceptions
 - **try {...}:** Block of code that might cause exceptions
 - **catch() {...}:** Catching and handling exceptions

```
try {  
    // Code block that might cause exception  
}  
catch (ExceptionType e) {  
    // Handling exception  
}
```

- **ExceptionType** is a descendant of the **Throwable**

Example of not handling exceptions

```
class NoException {
    public static void main(String args[]) {
        String text = args[0];
        System.out.println(text);
    }
}
```



```
D:\FIT-HUT\Lectures\OOP\OOP-Java\Demo>java NoException
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0
        at NoException.main(NoException.java:3)
D:\FIT-HUT\Lectures\OOP\OOP-Java\Demo>
```

Example of handling exceptions

```
class ArgExceptionDemo {
    public static void main(String args[]) {
        try {
            String text = args[0];
            System.out.println(text);
        }
        catch(Exception e) {
            System.out.println("Hay nhap tham so khi chay!");
        }
    }
}
```



```
D:\FIT-HUT\Lectures\OOP\OOP-Java\Demo>java ArgExceptionDemo
Hay nhap tham so khi chay!
D:\FIT-HUT\Lectures\OOP\OOP-Java\Demo>
```

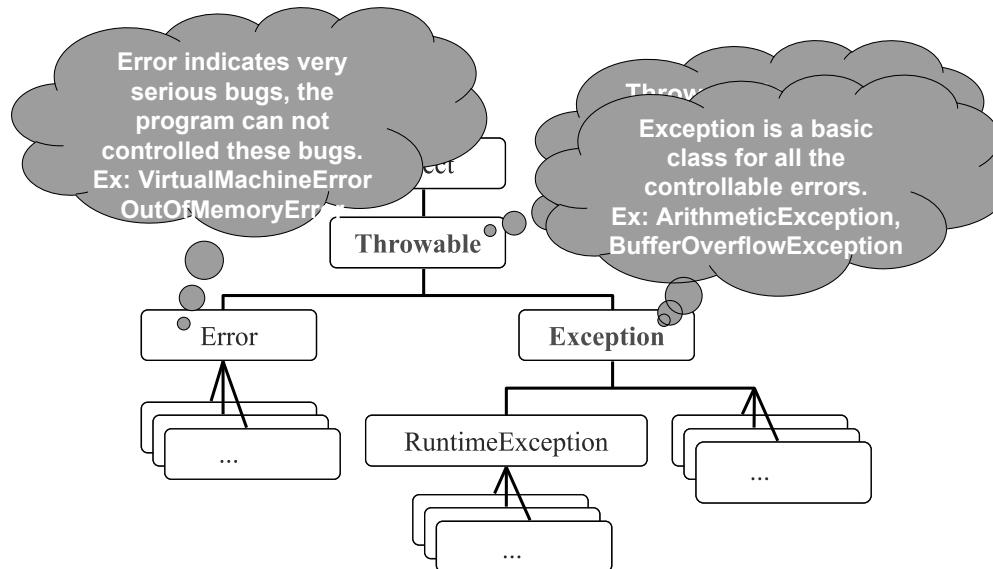
Example of division by 0

```

public class ChiaCho0Demo {
    public static void main(String args[]){
        try {
            int num = calculate(9,0);
            System.out.println(num);
        }
        catch(Exception e) {
            System.err.println("Co loi xay ra: " + e.toString());
        }
    }
    static int calculate(int no, int no1){
        int num = no / no1;
        return num;
    }
} → Co loi xay ra: java.lang.ArithmeticException: / by zero
Press any key to continue . . .

```

2.3.2. Exception hierarchical tree in Java



a. Class Throwable

- A variable of type String to store detailed information about exceptions that already occurred
- Some basic functions
 - **new Throwable(String s)**: Creates an exception and the exception information is s
 - **String getMessage()**: Get exception information
 - **String getString()**: Brief description of exceptions
 - **void printStackTrace()**: Print out all the involving information of exceptions (name, type, location...)
 - ...

```
public class StckExceptionDemo {
    public static void main(String args[]){
        try {
            int num = calculate(9,0);
            System.out.println(num);
        }
        catch(Exception e) {
            System.err.println("Co loi xay ra :"
                               + e.getMessage());
            e.printStackTrace();
        }
    }
    static int calculate(int no, int no1) {
        int num = no / no1;
        return num;
    }
}
```

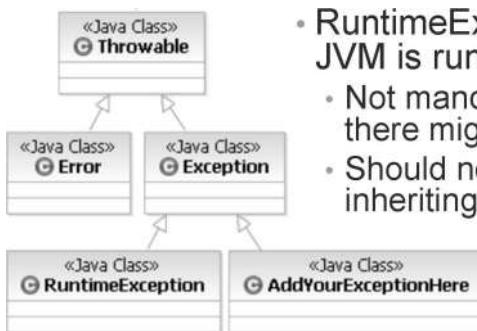
 Co loi xay ra :/ by zero
 java.lang.ArithmetricException: / by zero
 at StckExceptionDemo.calculate(StckExceptionDemo.java:14)
 at StckExceptionDemo.main(StckExceptionDemo.java:4)
 Press any key to continue . . .

b. Class Error

- Contains critical and unchecked exceptions (unchecked exception) because it might occur at many parts of the program.
- Is called un-recoverable exception
- Do not need to check in your Java source code
- Child classes:
 - VirtualMachineError: InternalError, OutOfMemoryError, StackOverflowError, UnknownError
 - ThreadDeath
 - LinkageError:
 - IncompatibleClassChangeError
 - AbstractMethodError, InstantiationException, NoSuchFieldError, NoSuchMethodError...
 - ...
 - ...

c. Class Exception

- Has exception types that should/must be caught and handled or delegated.
- Developers might create their own exceptions by inheriting from Exception
- RuntimeException might appear while JVM is running
 - Not mandatory to catch exceptions even there might be errors
 - Should not write your own exception inheriting from this class



Some derived classes of Exception

- ClassNotFoundException, SQLException
- java.io.IOException:
 - FileNotFoundException, EOFException...
- RuntimeException:
 - NullPointerException, BufferOverflowException
 - ClassCastException, ArithmeticException
 - IndexOutOfBoundsException:
 - ArrayIndexOutOfBoundsException,
 - StringIndexOutOfBoundsException...
 - IllegalArgumentException:
 - NumberFormatException, InvalidParameterException...
 - ...

Example of IOException

```
import java.io.InputStreamReader;
import java.io.IOException;
public class HelloWorld{
    public static void main(String[] args) {
        InputStreamReader isr = new
            InputStreamReader(System.in);
        try {
            System.out.print("Nhập vào 1 ký tự: ");
            char c = (char) isr.read();
            System.out.println("Ký tự vừa nhập: " + c);
        }catch(IOException ioe) {
            ioe.printStackTrace();
        }
    }
}
```

Nhập vào 1 ký tự: b
Ký tự vừa nhập: b
Press any key to continue . . .



2.3.3. Nested try – catch blocks

- A small part of a code block causes an error, but the whole block cause another error → Need to have nested exception handlers.
- When there are nested try blocks, the inner try block will be done first.

```
try {
    // May cause IOException
    try {
        // May cause NumberFormatException
    }
    catch (NumberFormatException e1) {
        // Handle NumberFormatException
    }
} catch (IOException e2) {
    // Handle IOException
}
```

2.3.4. Multiple catch block

- A block of code might cause more than one exception
→ Need to use multiple catch block.

```
try {
    // May cause multiple exception
} catch (ExceptionType1 e1) {
    // Handle exception 1
} catch (ExceptionType2 e2) {
    // Handle exception 2
} ...
```

- **ExceptionType1** must be a derived class or an level-equivalent class of the class **ExceptionType2** (in the inheritance hierarchy tree)

- ExceptionType1 must be a derived class or an level-equivalent class of the class ExceptionType2 (in the inheritance hierarchy tree)

```

class MultipleCatch1 {
    public static void main(String args[])
    {
        try {
            String num = args[0];
            int numValue = Integer.parseInt(num);
            System.out.println("Dien tich hv la: "
                + numValue * numValue);
        } catch(Exception e1) {
            System.out.println("Hay nhap canh cua
hv!");
        } catch(NumberFormatException e2) {
            System.out.println("Not a number!");
        }
    } Error D:\exception java.lang.NumberFormatException
}

```

has already been caught

- ExceptionType1 must be a derived class or an level-equivalent class of the class ExceptionType2 (in the inheritance hierarchy tree)

```

class MultipleCatch1 {
    public static void main(String args[])
    {
        try {
            String num = args[0];
            int numValue = Integer.parseInt(num);
            System.out.println("Dien tich hv la: "
                + numValue * numValue);
        } catch(ArrayIndexOutOfBoundsException e1) {
            System.out.println("Hay nhap canh cua hv!");
        } catch(NumberFormatException e2) {
            System.out.println("Hay nhap 1 so!");
        }
    }
}

```

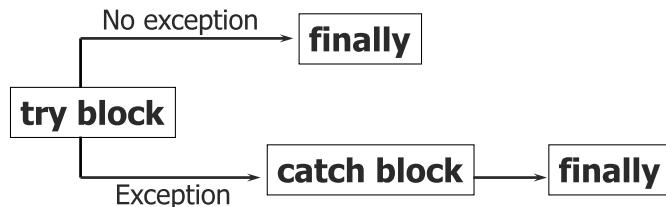
```
class MultiCatch2 {  
    public static void main( String args[] ) {  
        try {  
            // format a number  
            // read a file  
            // something else...  
        }  
        catch( IOException e) {  
            System.out.println("I/O error "+e.getMessage());  
        }  
        catch( NumberFormatException e) {  
            System.out.println("Bad data "+e.getMessage());  
        }  
        catch( Throwable e) { // catch all  
            System.out.println("error: " + e.getMessage());  
        }  
    }  
}
```

33

```
...  
public void openFile(){  
    try {  
        // constructor may throw FileNotFoundException  
        FileReader reader = new FileReader("someFile");  
        int i=0;  
        while(i != -1) {  
            //reader.read() may throw IOException  
            i = reader.read();  
            System.out.println((char) i );  
        }  
        reader.close();  
        System.out.println(" --- File End --- ");  
    } catch (FileNotFoundException e) {  
        //do something clever with the exception  
    } catch (IOException e) {  
        //do something clever with the exception  
    }  
}
```

2.3.5. finally block

- Ensure that every necessary tasks are done when an exception occurs
 - Closing file, closing socket, connection
 - Releasing resource (if necessary)...
- Must be done even there is an exception occurring or not.



The syntax try ... catch ... finally

```

try {
    // May cause exceptions
}
catch(ExceptionType e) {
    // Handle exceptions
}
finally {
    /* Necessary tasks for all cases:
    exception is raised or not */
}
  
```

- ❑ If there is a block try, there must be a block catch or a block finally or both

```

class StrExceptionDemo {
    static String str;
    public static void main(String s[]) {
        try {
            System.out.println("Before exception");
            staticLengthmethod();
            System.out.println("After exception");
        }
        catch(NullPointerException ne) {
            System.out.println("There is an error");
        }
        finally {
            System.out.println("In finally");
        }
    }

    static void staticLengthmethod() {
        System.out.println(str.length());
    }
}

```

```

public void openFile(){
    try {
        // constructor may throw FileNotFoundException
        FileReader reader = new FileReader("someFile");
        int i=0;
        while(i != -1) {
            //reader.read() may throw IOException
            i = reader.read();
            System.out.println((char) i );
        }
    } catch (FileNotFoundException e) {
        //do something clever with the exception
    } catch (IOException e) {
        //do something clever with the exception
    } finally {
        reader.close();
        System.out.println("--- File End ---");
    }
}

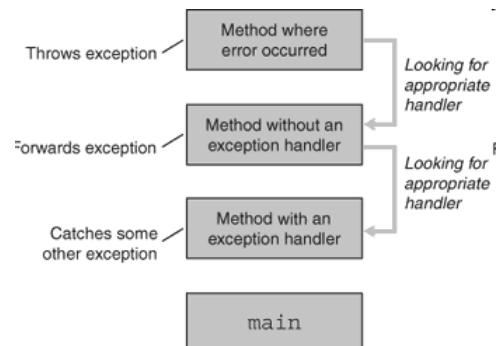
```

Outline

1. Exceptions
2. Catching and handling exceptions
3. Exception delegation
4. User-defined exceptions

Two ways to deal with exceptions

- Handle immediately
 - Using the block try ... catch (finally if necessary).
- Delegating to its caller:
 - If we don't want to handle immediately
 - Using throw and throws



3.1. Exception delegation

- A method can delegate exceptions to its caller:
 - Using **throws** at the method definition to tell its caller of ExceptionType that it might cause an exception ExceptionType
 - Using **throw** anExceptionObject in the body of function in order to throw an exception when necessary
- For example

```
public void myMethod(int param) throws Exception{
    if (param < 10) {
        throw new Exception("Too low!");
    }
    //Blah, Blah, Blah...
}
```

3.1. Exception delegation (2)

- If a method has some code that throws an exception, its declaration must declare a “throw” of that exception or the parent class of that exception

```
public void myMethod(int param) {
    if (param < 10) {
        throw new Exception("Too low!");
    }
    //Blah, Blah, Blah...
}
```

→ unreported exception java.lang.Exception; must be caught or declared to be thrown

3.1. Exception delegation (3)

- A method without exception declaration will throw `RuntimeException` because this exception is delegated to JVM

- Example

```
class Test {  
    public void myMethod(int param) {  
        if (param < 10) {  
            throw new RuntimeException("Too low!");  
        }  
        //Blah, Blah, Blah...  
    }  
}
```

3.1. Exception delegation (3)

- At the caller of the method that has exception delegation (except `RuntimeException`):
 - Or the caller method must delegate to its caller
 - Or the caller method must catch the delegated exception (or its parent class) and handle immediately by `try... catch (finally if necessary)`

```

public class DelegateExceptionDemo {
    public static void main(String args[]) {
        int num = calculate(9,3);
        System.out.println("Lan 1: " + num);
        num = calculate(9,0);
        System.out.println("Lan 2: " + num);
    }
    static int calculate(int no, int no1)
        throws ArithmeticException {
        if (no1 == 0)
            throw new
                ArithmeticException("Cannot devide by 0!");
        int num = no / no1;
        return num;
    }
}

```

```

public class DelegateExceptionDemo {
    public static void main(String args[]) {
        int num = calculate(9,3);
        System.out.println("Lan 1: " + num);
        num = calculate(9,0);
        System.out.println("Lan 2: " + num);
    }
    static int calculate(int no, int no1)
        throws Exception {
        if (no1 == 0)
            throw new
                ArithmeticException("Cannot divide by 0!");
        int num = no / no1;
        return num;
    }
}

```

G:\Java Example\DelegateExceptionDemo.java:3: unreported exception java.lang.Exception; must be caught or declared to be thrown

int num = calculate(9,3);



G:\Java Example\DelegateExceptionDemo.java:5: unreported exception java.lang.Exception; must be caught or declared to be thrown

num = calculate(9,0);

```

public class DelegateExceptionDemo {
    public static void main(String args[]) {
        try {
            int num = calculate(9,3);
            System.out.println("Lan 1: " + num);
            num = calculate(9,0);
            System.out.println("Lan 2: " + num);
        } catch(Exception e) {
            System.out.println(e.getMessage());
        }
    }
    static int calculate(int no, int no1)
        throws Exception {
        if (no1 == 0)
            throw new
                ArithmeticException("Cannot devide by 0!");
        int num = no / no1;
        return num;
    }
}

```

3.1. Exception delegation (4)

- A method can delegate more than 1 exception

```

public void myMethod(int age, String name)
    throws ArithmeticException, NullPointerException{
    if (age < 18) {
        throw new ArithmeticException
            ("Age must be at least 18");
    }
    if (name == null) {
        throw new NullPointerException
            ("Name must be provided");
    }
    //Blah, Blah, Blah...
}

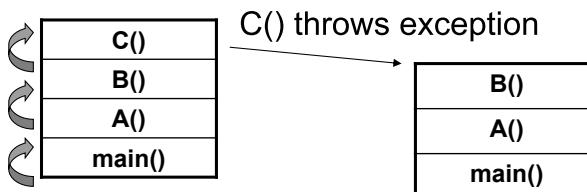
```

3.2. Exception propagation

- Scenario:

- Assuming that in main() method A() is called, B() is called in A(), C() is called in B(). Then a stack of method is created.
- Assuming that in C() there is an exception occurring.

3.2. Exception Propagation (2)



If C() has an error and throws an exception but in C() that exception is not handled, hence there is only one place that handles the exception, that place is where C() is called, it is the method B().

If in B() there is no exception handling, then the exception must be handled in A() ... This is called Exception Propagation

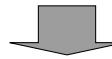
If in main(), the exception thrown from C() can not be handled, the program will be interrupted.

3.3. Inheritance and exception delegation

- When overriding a method of a parent class, methods in its child classes can not throw any new exception

→ Overridden method in a child class can only throw a set of exceptions that are/similar to/ a subset of exceptions thrown from the parent class.

3.3. Inheritance and exception delegation(2)

```
class Disk {  
    void readFile() throws EOFException {}  
}  
  
class FloppyDisk extends Disk {  
    void readFile() throws IOException {} // ERROR!  
}  
  
  
  
class Disk {  
    void readFile() throws IOException {}  
}  
  
class FloppyDisk extends Disk {  
    void readFile() throws EOFException {} //OK  
}
```

3.4. Advantages of exception delegation

- Easy to use
 - Making programs easier to read and more reliable
 - Easy to send control to the places that can handle exceptions
 - Can throw many types of exceptions
- Separating exception handling from the main code
- Do not miss any exception (throw automatically)
- Grouping and categorizing exceptions
- Making program easier to read and more reliable

52

Outline

1. Exceptions
2. Catching and handling exceptions
3. Exception delegation
4. User-defined exceptions

4. User-defined exception

- Exceptions provided can not control all the errors → Need to have exceptions that are defined by users.
 - Inheriting from the class **Exception** or one of its child classes
 - Having all the methods of the class **Throwable**

```
public class MyException extends Exception {
    public MyException(String msg) {
        super(msg);
    }
    public MyException(String msg, Throwable cause) {
        super(msg, cause);
    }
}
```

Using self-defined exceptions

Declaring that an exception might be thrown

```
public class FileExample
{
    public void copyFile(String fName1, String fName2)
throws MyException
    {
        if (fName1.equals(fName2))
            throw new MyException("File trùng tên");
        // Copy file
        System.out.println("Copy completed");
    }
}
```

Throwing an exception

Using self-defined exceptions

- Catching and handling exceptions

```
public class Test {
    public static void main(String[] args) {
        FileExample obj = new FileExample();
        try {
            String a = args[0];
            String b = args[1];
            obj.copyFile(a,b);
        } catch (MyException e1) {
            System.out.println(e1.getMessage());
        }
        catch(Exception e2) {
            System.out.println(e2.toString());
        }
    }
}
```



```
C:\>java Test a1.txt a1.txt
File trung ten
C:\>java Test
Java.lang.ArrayIndexOutOfBoundsException: 0
```

Quiz Modify the following source code so that `copyFile()` method will throw 2 exceptions:

- MyException if the 2 file names are equal, and
- IOException if there is any error during the copy file process

```
public class FileExample {
    public void copyFile(String fName1, String fName2)
        throws MyException{
        if (fName1.equals(fName2))
            throw new MyException("Duplicate file name");

        // Copy file

        System.out.println("Copy completed");
    }
}
```

Conclusion

- Anytime there is an error while running the program, an exception appears.
- All the exceptions must be handled to avoid unexpected termination of the program.
- Handling exceptions allows to handle all the exception in a place.
- Java uses the block try/catch to manage exceptions.

Conclusion (2)

- Code blocks in the block try throw exception, and the exception handling must be done in the block catch.
- Many blocks of catch can be used to handle separately different exceptions.
- The keyword throws is used to list all the exceptions that a method can throw.
- The keyword throw is used to throw an exception.
- The block finally performs necessary tasks even there is an exception occurring or not.

Conclusion (3)

- Types of exception handling:
 - Fix errors and call again the method that caused these errors
 - Fix errors and continue running the method
 - Handling differently instead of ignoring the result
 - Exit the program

Outline

1. Exception
2. Catching and handling exception
3. Exception delegation
4. Create self-defined exception
5. Assertion

5.1. Assertion là gì?

- Assertion cho phép lập trình viên kiểm tra các giả thiết về chương trình.
 - Trong chương trình giả lập hệ thống giao thông, bạn muốn khẳng định rằng tốc độ dương nhưng nhỏ hơn một giá trị giới hạn nào đó.
- Một assertion chứa một biểu thức boolean mà bạn tin rằng sẽ đúng khi thực hiện – nếu không đúng hệ thống sẽ ném ra một lỗi
 - Bằng việc kiểm tra biểu thức boolean là đúng, assertion xác nhận giả thuyết của bạn, và giúp bạn tự tin hơn rằng chương trình không có lỗi.

5.2. Sử dụng Assertion

- assert expression;
 - expression trả về kiểu boolean, nếu giá trị của nó là false thì hệ thống sẽ tung ra AssertionError.
 - → Không thể thu được bất cứ thông tin gì về lỗi đã xảy ra.
- assert expression1:expression2;
 - expression1 trả về giá trị boolean, biểu thức expression2 có bất kỳ kiểu giá trị nào ngoại trừ lời gọi phương thức trả về kiểu void.
 - Nếu expression1 trả về false
 - Hệ thống tung ra AssertionError.
 - Giá trị trong expression2 sẽ được truyền vào hàm tạo của lớp AssertionError và giá trị đó sẽ được hiển thị để thông báo lỗi.

5.2. Sử dụng Assertion

- Việc kiểm tra assertion mặc định bị disable
 - Cần được enable lên sử dụng câu lệnh enableassertions
 - Nếu không được enable thì câu lệnh assertion sẽ không được thực hiện.

5.2. Lợi ích của Assertion

- Nhanh và hiệu quả để tìm ra lỗi và sửa lỗi
- Ghi lại các công việc bên trong của chương trình của bạn, giúp nâng cao tính bảo trì
- Lập trình theo thiết kế
 - Các tiền điều kiện (Pre-conditions)
 - Đảm bảo các tiền điều kiện như yêu cầu của khách hàng
 - Các hậu điều kiện (Post-conditions)
 - Đảm bảo hậu điều kiện là kết quả của phương thức gọi
 - Các biến bên trong
 - Lập trình viên sử dụng để xác nhận giả thuyết của mình

5.2. Lợi ích của Assertion (2)

- Ví dụ:

```

if (i % 3 == 0) { ... }
else if (i % 3 == 1) { ... }
else { // We know (i % 3 == 2)
    ...
}

if (i % 3 == 0) { ... }
else if (i % 3 == 1) { ... }
else { assert i % 3 == 2 : i; ... }

```

Luồng điều khiển

- Nếu một chương trình không bao giờ đi đến một điểm nào đó, thì một assertion hằng false được sử dụng

```

void foo() {
    for (...) {
        if (... )
            return;
    }
    assert false; // Execution should never get here
}

```

OBJECT LANGUAGE AND THEORY

11. CLASS DIAGRAMS

Nguyen Thi Thu Trang
trangntt@soict.hust.edu.vn



Objectives

- Describe the static view of the system and show how to capture it in a model.
- Demonstrate how to read and interpret a class diagram.
- Model an association and aggregation and show how to model it in a class diagram.
- Model generalization on a class diagram.

Content

→ 1. Class diagrams

2. Association

3. Aggregation and Composition

4. Generalization

1.1. Classes in the UML

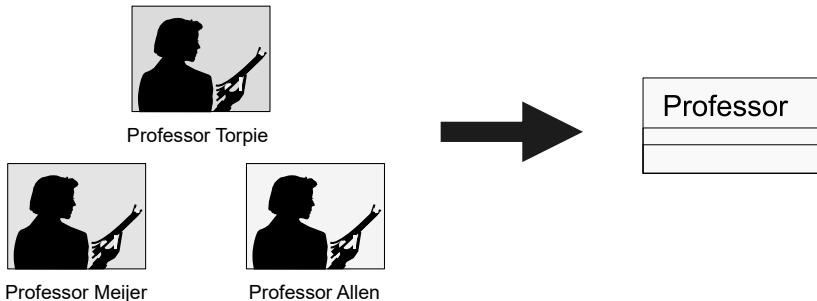
- A class is represented using a rectangle with three compartments:

- The class name
- The structure (attributes)
- The behavior (operations)

Professor
<ul style="list-style-type: none"> - name - employeeID : UniqueId - hireDate - status - discipline - maxLoad
<ul style="list-style-type: none"> + submitFinalGrade() + acceptCourseOffering() + setMaxLoad() + takeSabbatical() + teachClass()

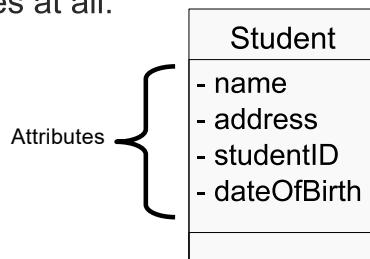
Classes and Objects

- A class is an abstract definition of an object
 - It defines the structure and behavior of each object in the class.
 - It serves as a template for creating objects.
- Classes are not collections of objects

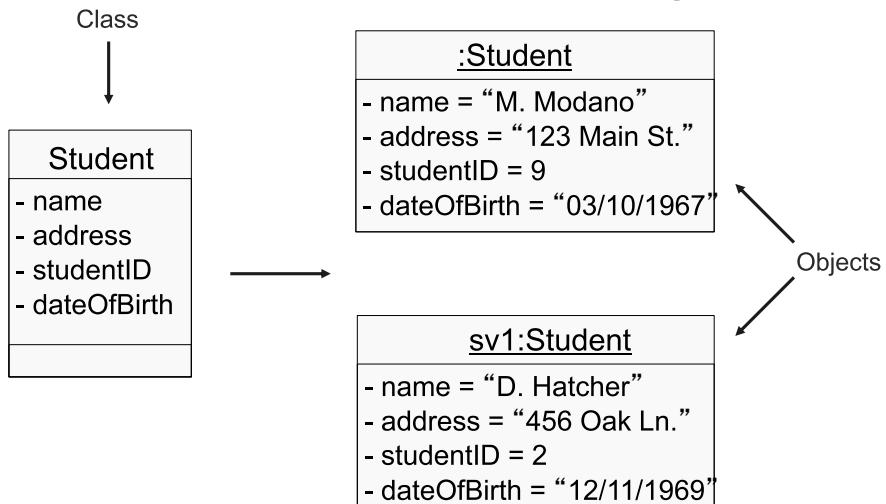


What Is an Attribute?

- An attribute is a named property of a class that describes the range of values that instances of the property may hold.
 - A class may have any number of attributes or no attributes at all.

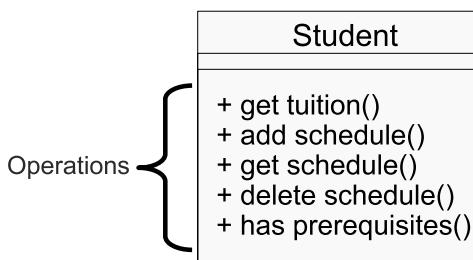


Attributes in Classes and Objects



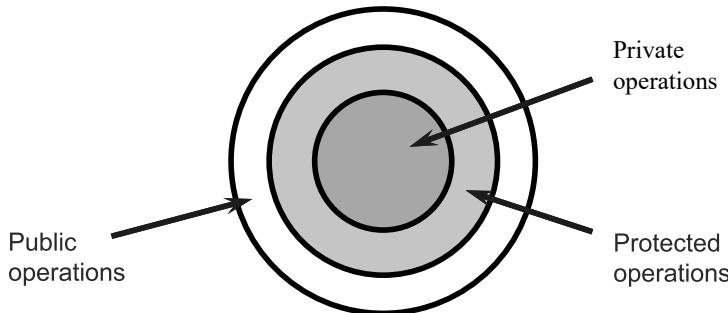
What Is an Operation?

- A service that can be requested from an object to effect behavior. An operation has a signature, which may restrict the actual parameters that are possible.
- A class may have any number of operations or none at all.



Member Visibility

- Visibility is used to enforce encapsulation
- May be public, protected, or private



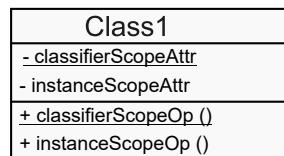
How Is Visibility Noted?

- The following symbols are used to specify export control:
 - + Public access
 - # Protected access
 - - Private access

ClassName
- privateAttribute + publicAttribute # protectedAttribute
- privateOperation () + publicOperation () # protecteOperation ()

Scope

- Determines number of instances of the attribute/operation
 - Instance: one instance for each class instance
 - Classifier: one instance for all class instances
- Classifier scope is denoted by underlining the attribute/operation name



1.2. What Is a Class Diagram?

- Static view of a system

CloseRegistrationForm
+ open()
+ close registration()

Schedule
<u>- semester</u>
+ commit()
+ select alternate()
+ remove offering()
+ level()
+ cancel()
+ get cost()
+ delete()
+ submit()
+ save()
+ any conflicts?()
+ create with offerings()
+ update with new selections()

CloseRegistrationController
+ is registration open?()
+ close registration()

Student
+ get tuition()
+ add schedule()
+ get schedule()
+ delete schedule()
+ has pre-requisites()

Professor
<u>- name</u>
<u>- employeeID : UniqueId</u>
<u>- hireDate</u>
<u>- status</u>
<u>- discipline</u>
<u>- maxLoad</u>
+ submitFinalGrade()
+ acceptCourseOffering()
+ setMaxLoad()
+ takeSabbatical()
+ teachClass()

Static Structure vs. Dynamic Behavior

- **Static aspects:** Software component and how they are related to one another
- **Dynamic aspects:** How the components interact with one another and/or change state internally over time.



static

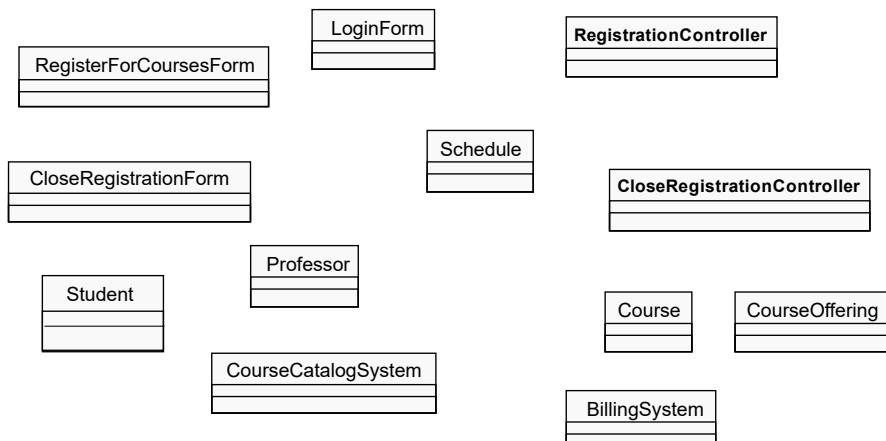


dynamic

@Nguyễn Thị Thu Trang, trangntt@soict.hust.edu.vn 14

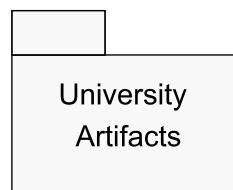
Example: Class Diagram

- Is there a better way to organize class diagrams?

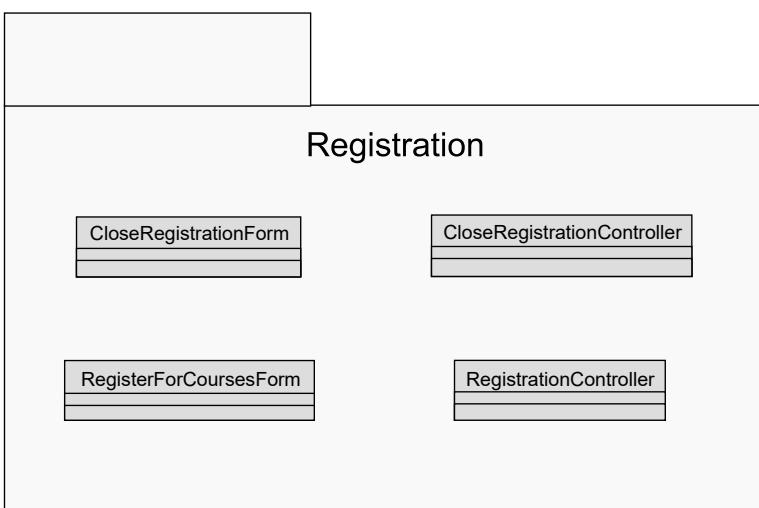


Review: What Is a Package?

- A general purpose mechanism for organizing elements into groups.
- A model element that can contain other model elements.
- A package can be used:
 - To organize the model under development
 - As a unit of configuration management

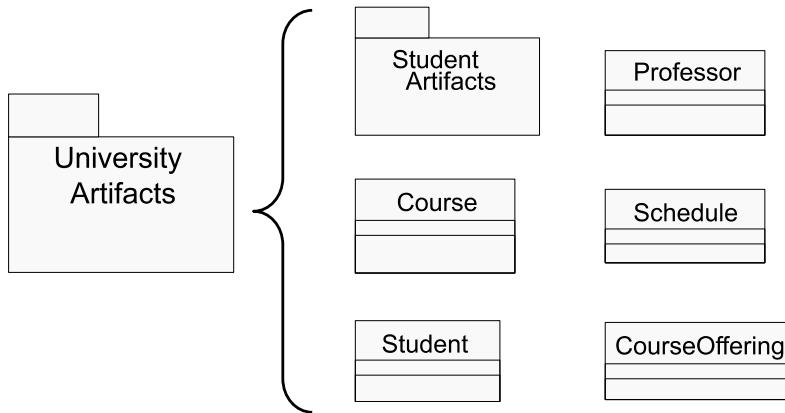


Example: Registration Package



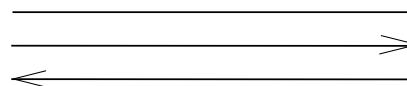
A Package Can Contain Classes

- The package, University Artifacts, contains one package and five classes.



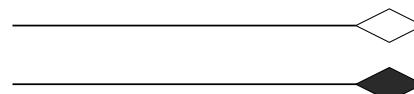
Class Relationships

- Association

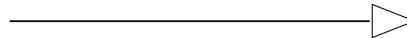


- Aggregation

- Composition



- Generalization



- Realization



Content

1. Class diagrams

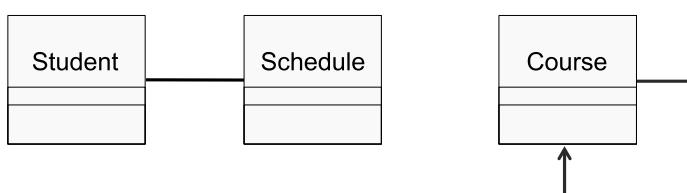
2. Association

3. Aggregation and Composition

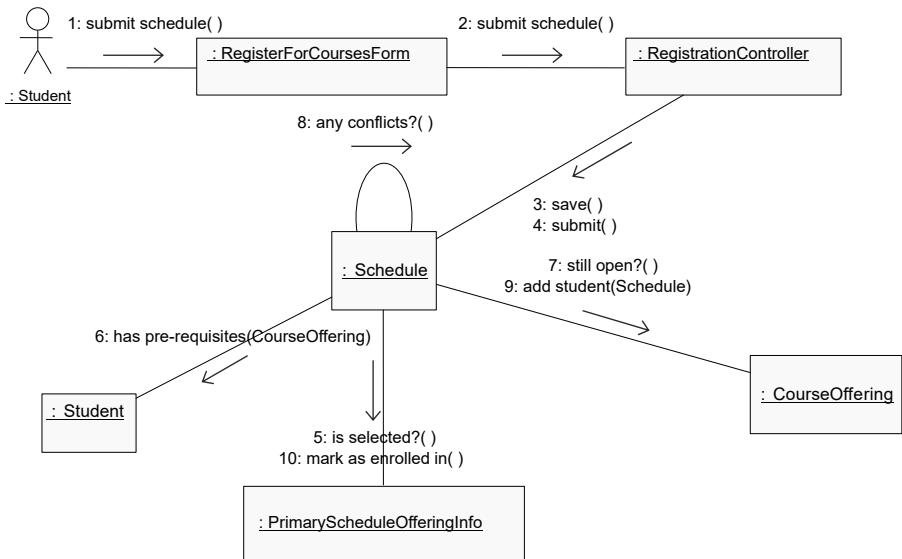
4. Generalization

What Is an Association?

- The semantic relationship between two or more classifiers that specifies connections among their instances.
- A structural relationship specifying that objects of one thing are connected to objects of another thing.



Example: What Associations?



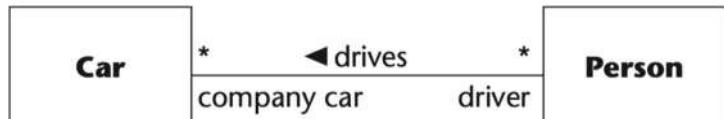
Role

- Role

- Useful technique for specifying the context of a class and its objects
- Optional

- Role name

- String placed near the end of the association next to the class to which it applies
- Indicates the role played by the class in terms of the association.
- Part of the association and not part of the classes



What Is Multiplicity?

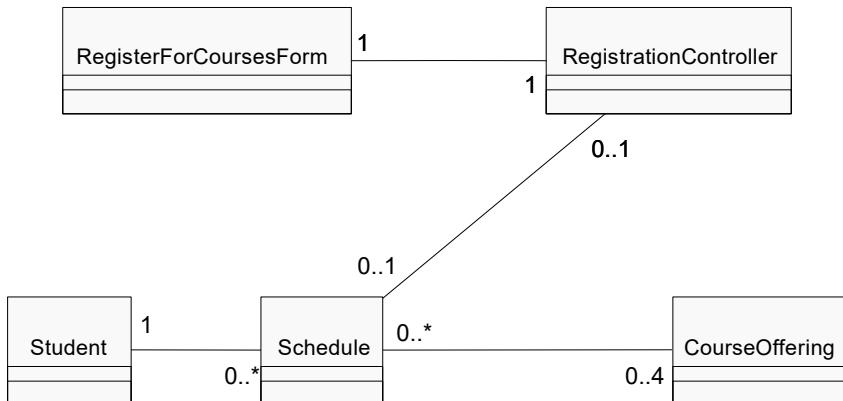
- Multiplicity is the number of instances one class relates to ONE instance of another class.
- For each association, there are two multiplicity decisions to make, one for each end of the association.
 - For each instance of Professor, many Course Offerings may be taught.
 - For each instance of Course Offering, there may be either one or zero Professor as the instructor.



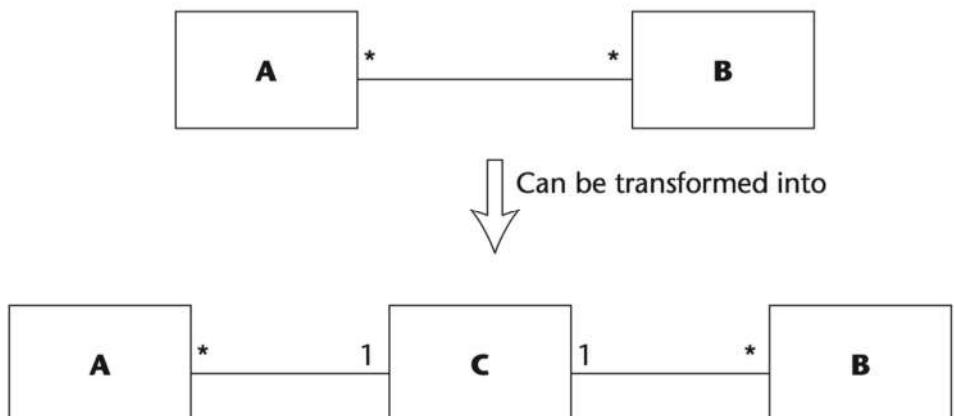
Multiplicity Indicators

Unspecified	
Exactly One	1
Zero or More	0..*
Zero or More	*
One or More	1..*
Zero or One (optional value)	0..1
Specified Range	2..4
Multiple, Disjoint Ranges	2, 4..6

Example: Multiplicity



Many-to-many association



Java implementation



```
//InsuranceCompany.java file
public class InsuranceCompany
{
    // Many multiplicity can be implemented using Collection
    private List<InsuranceContract> contracts;

    /* Methods */
}

// InsuranceContract.java file
public class InsuranceContract
{
    private InsuranceCompany refers_to;

    /* Methods */
}
```

Content

1. Class diagrams
2. Association
3. Aggregation and Composition
4. Generalization

What Is an Aggregation?

- A special form of association that models a whole-part relationship between the aggregate (the whole) and its parts.
 - An aggregation is an “is a part-of” relationship.
- Multiplicity is represented like other associations.



What is Composition?

- A special form of aggregation with strong ownership and coincident lifetimes of the part with the aggregate
 - Also called composition aggregate
- The whole “owns” the part and is responsible for the creation and destruction of the part.
 - The part is removed when the whole is removed.
 - The part may be removed (by the whole) before the whole is removed.



Examples: Association Types

- Association

- use-a

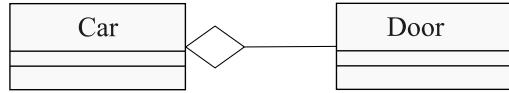
- Objects of one class are associated with objects of another class



- Aggregation

- has-a/is-a-part

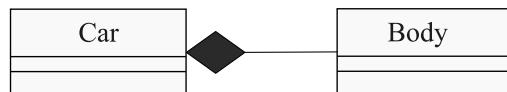
- Strong association, an instance of one class is made up of instances of another class



- Composition

- Strong aggregation, the composed object can't be shared by other objects and dies with its composer

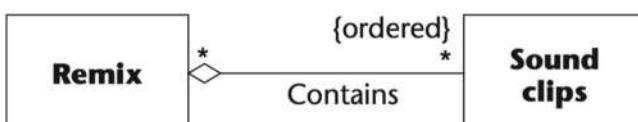
- Share life-time



Aggregation Example



- A *shared aggregation* is one in which the parts may be parts in any wholes

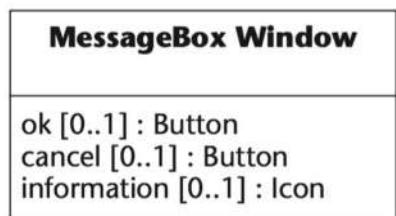
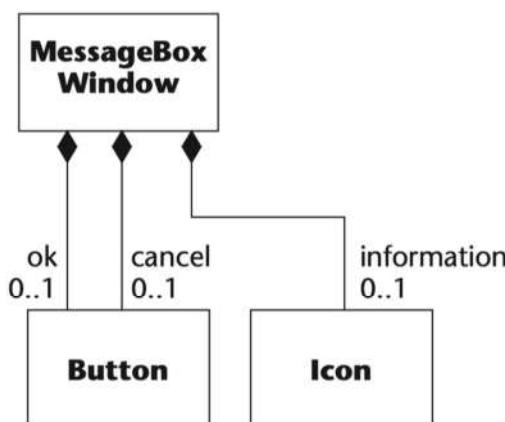


Aggregation – Java implementation

```
class Car {  
    private List<Door> doors;  
    Car(String name, List<Door> doors) {  
        this.doors = doors;  
    }  
  
    public List<Door> getDoors() {  
        return doors;  
    }  
}
```

Composition Example

- A compound aggregate is shown as attributes in a class



Composition – Java implementation

```
final class Car {  
    // For a car to move, it need to have a engine.  
    private final Engine engine; // Composition  
    //private Engine engine; // Aggregation  
  
    Car(Engine engine) {  
        this.engine = engine;  
    }  
  
    // car start moving by starting engine  
    public void move() {  
        //if(engine != null)  
        {  
            engine.work();  
            System.out.println("Car is moving ");  
        }  
    }  
}  
  
class Engine {  
    // starting an engine  
    public void work() {  
        System.out.println("Engine of car has been started ");  
    }  
}
```

Content

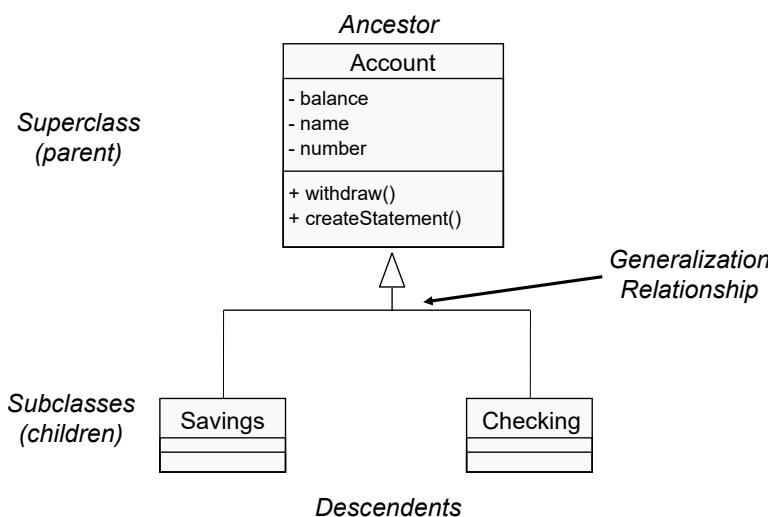
1. Class diagrams
2. Association
3. Aggregation and Composition
4. Generalization

Review: What Is Generalization?

- A relationship among classes where one class shares the structure and/or behavior of one or more classes.
- Defines a hierarchy of abstractions where a subclass inherits from one or more superclasses.
 - Single inheritance
 - Multiple inheritance
- Is an “is a kind of” relationship.

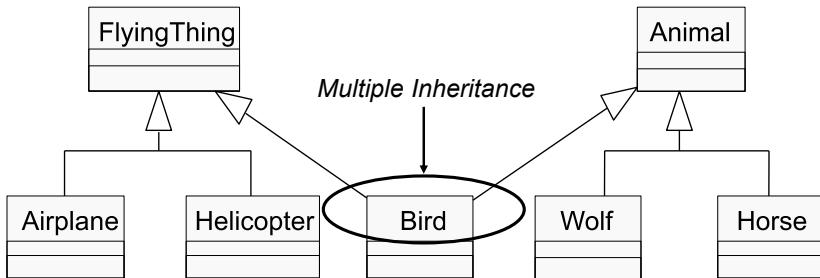
Example: Single Inheritance

- One class inherits from another.



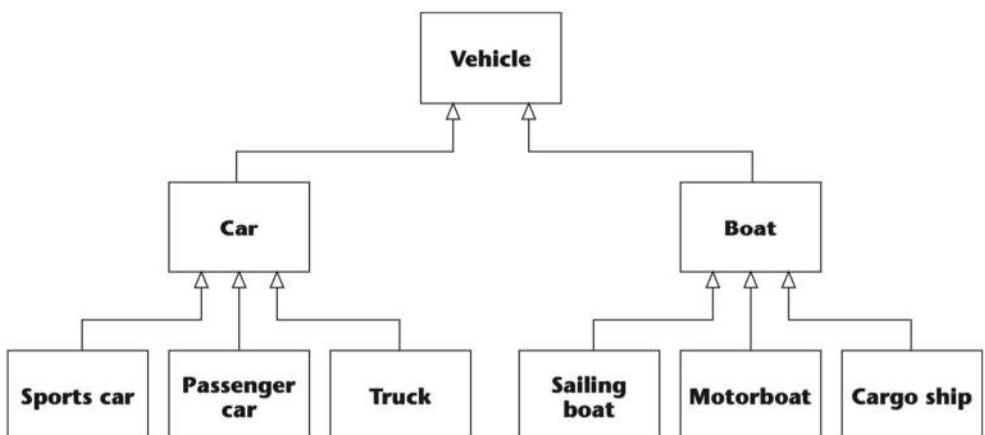
Example: Multiple Inheritance

- A class can inherit from several other classes.



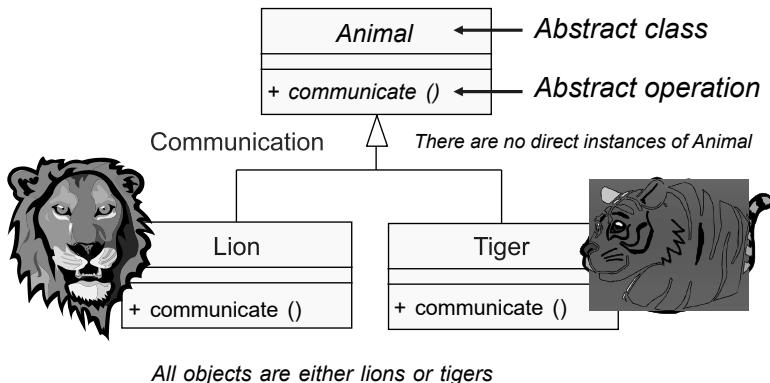
Use multiple inheritance only when needed and always with caution!

Inheritance Tree Example

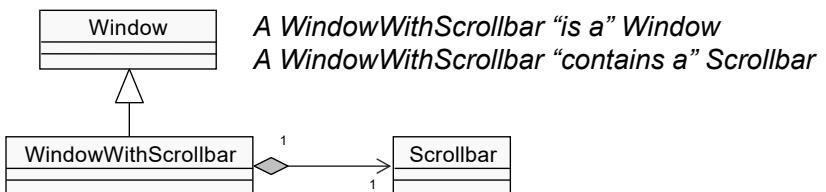
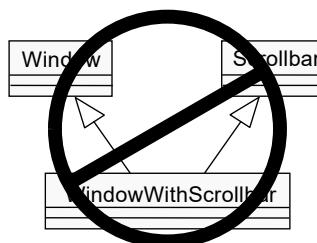


Abstract and Concrete Classes

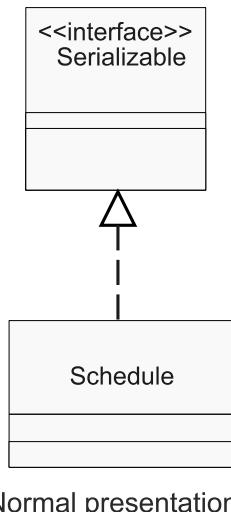
- Abstract classes cannot have any objects
- Concrete classes can have objects



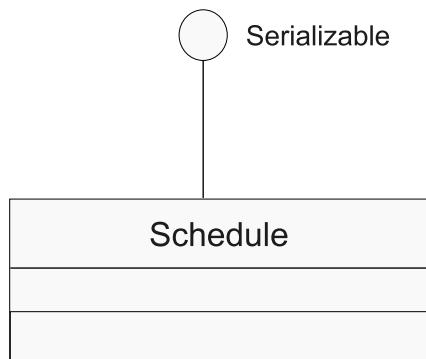
Generalization vs. Aggregation



Interfaces and Realizes Relationships



Normal presentation



Icon presentation



Exercise

Document a class diagram using the following information

- A class diagram containing the following classes:
Personal Planner Profile, Personal Planner Controller,
Customer Profile, and Buyer Record.
- Associations drawn using the following information:
 - Each Personal Planner Profile object can be associated with up to one Personal Planner Controller object.
 - Each Personal Planner Controller object must be related to one Personal Planner Profile.
 - A Personal Planner Controller object can be associated with up to one Buyer Record and Customer Profile object.
 - An instance of the Buyer Record class can be related to zero or one Personal Planner Controller.
 - Zero or one Personal Planner Controller objects are associated with each Customer Profile instance.

OBJECT-ORIENTED LANGUAGE AND THEORY

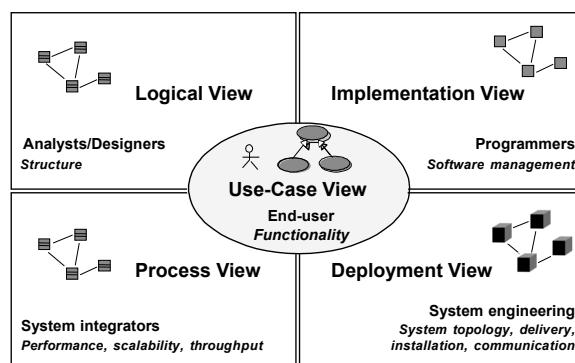
13. UML DIAGRAMS

Nguyen Thi Thu Trang
trangntt@soict.hust.edu.vn



4+1 UML Views

- No single model is sufficient. Every non-trivial system is best approached through a small set of nearly independent models.
- Create models that can be built and studied separately, but are still interrelated.

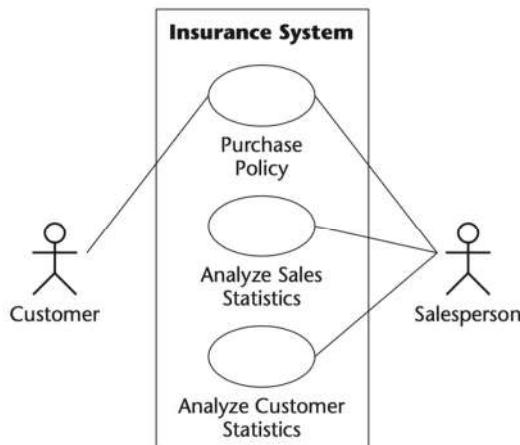


Common diagrams in UML

- Use-case diagram
- Class diagram
- Object Diagram
- State machine
- Activity diagram
- Interaction diagrams
- Deployment diagram

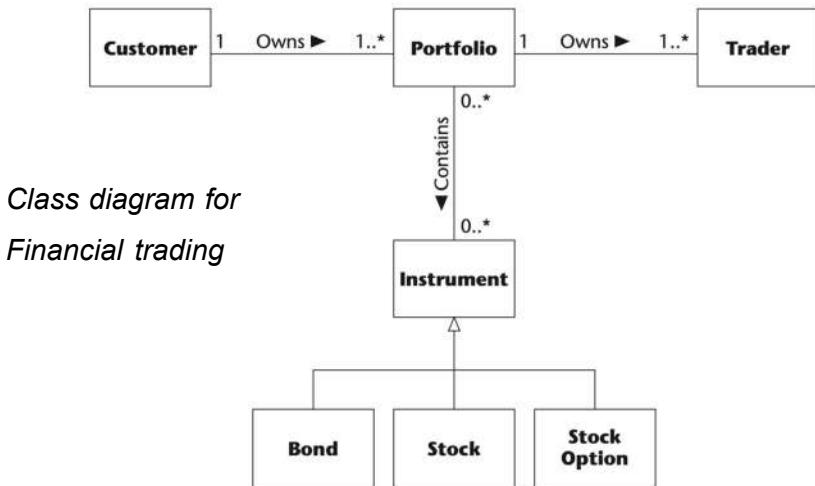
Use case diagram

- A number of external actors and their connection to the use cases that the system provides



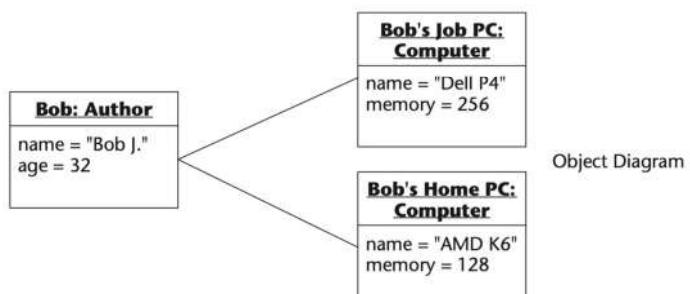
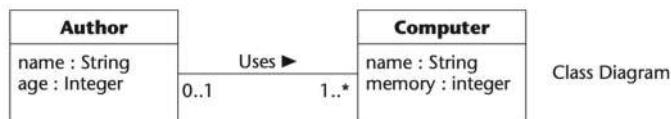
Class diagram

- Static structure of classes in the system



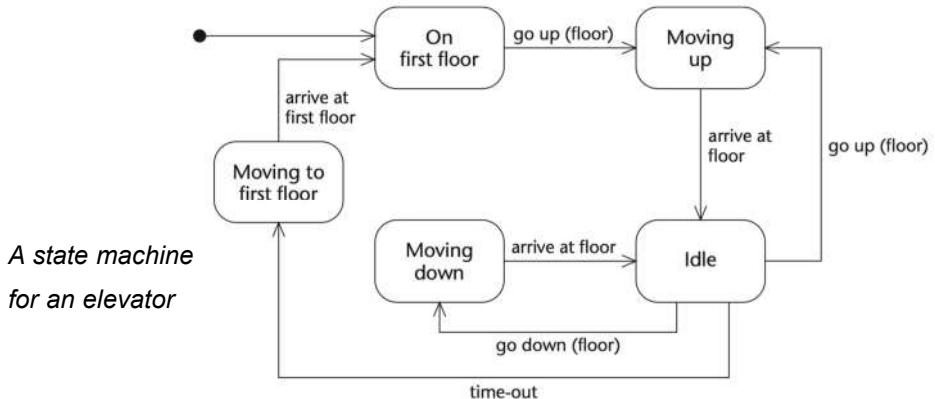
Object diagram

- Shows a number of object instances of classes, instead of the actual classes



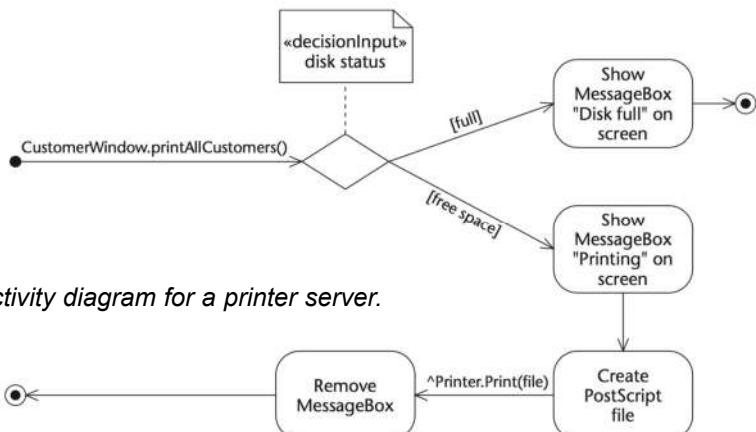
State machine

- Shows all the possible states that objects of the class can have during a life-cycle instance, and which events cause the state to change



Activity diagram

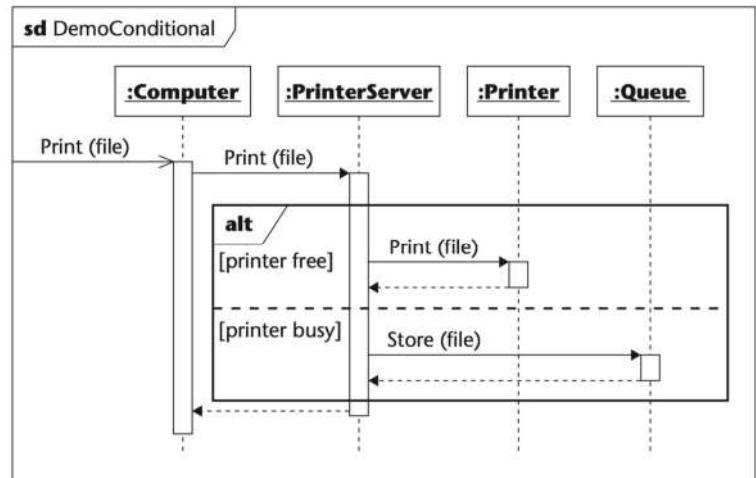
- Shows a sequential flow of actions to describe
 - the activities performed in a general process workflow
 - or other activity flows, such as a use case or a detailed control flow



Interaction Diagrams

- Show the interaction between objects during the execution of the software

A sequence diagram for a print server



Deployment Diagram

- Shows the physical architecture of the hardware and software in the system

