



# 9 Construction for Reuse

## 面向复用的软件构造技术

刘铭

May 27, 2021

# Objective of this lecture

- Advantages and disadvantages of software reuse
- Construction for/with reuse
- Characteristics of generic reusable components
- Methods of developing portable application systems

前几次课介绍了软件构造的核心理论（ADT）与技术（OOP），其核心是保证代码质量、提高代码安全性。

本次课面向一个重要的外部质量指标：可复用性——如何构造出可在不同应用中重复使用的软件模块/API？

首先探讨可复用的软件应该“长什么样”，然后学习“如何构造”

# Outline

- What is software reuse?
- How to measure “reusability”?
- Levels and morphology of reusable components
  - Source code level reuse 源代码级别的复用
  - Module-level reuse: class/interface 模块级别的复用：类/抽象类/接口
  - Library-level: API/package 库级别的复用：API/包
  - System-level reuse: framework 系统级别的复用：框架

# Outline

- Designing reusable classes 设计可复用的类
  - Inheritance and overriding 继承与重写
  - Overloading 重载
  - Parametric polymorphism and generic programming 参数多态与泛型编程
  - Behavioral subtyping and Liskov Substitution Principle (LSP) 行为子类型与Liskov替换原则
  - Composition and delegation 组合与委托
- Designing system-level reusable libraries and frameworks 设计可复用库与框架
  - API and Library
  - Framework
  - Java Collections Framework  
(an example)

从类、API、框架三个层面学习如何设计可复用软件实体的具体技术

# Reading

- CMU 17-214: Oct 10、Oct 15、Oct 17、Oct 22
- Java编程思想: 第14、15章
- Effective Java: 第5章

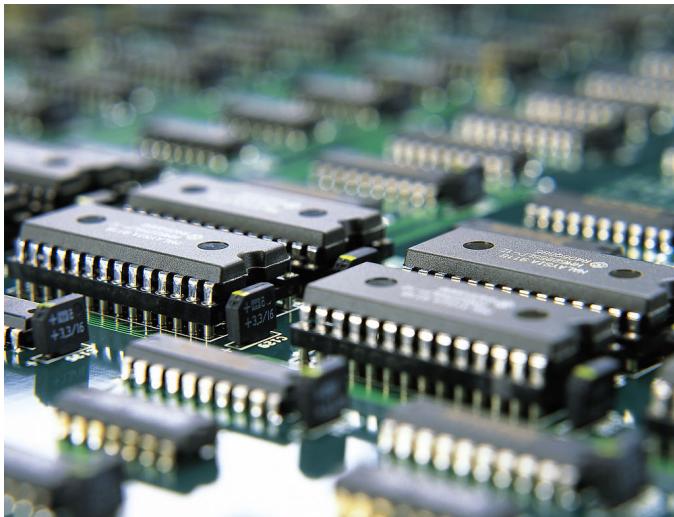
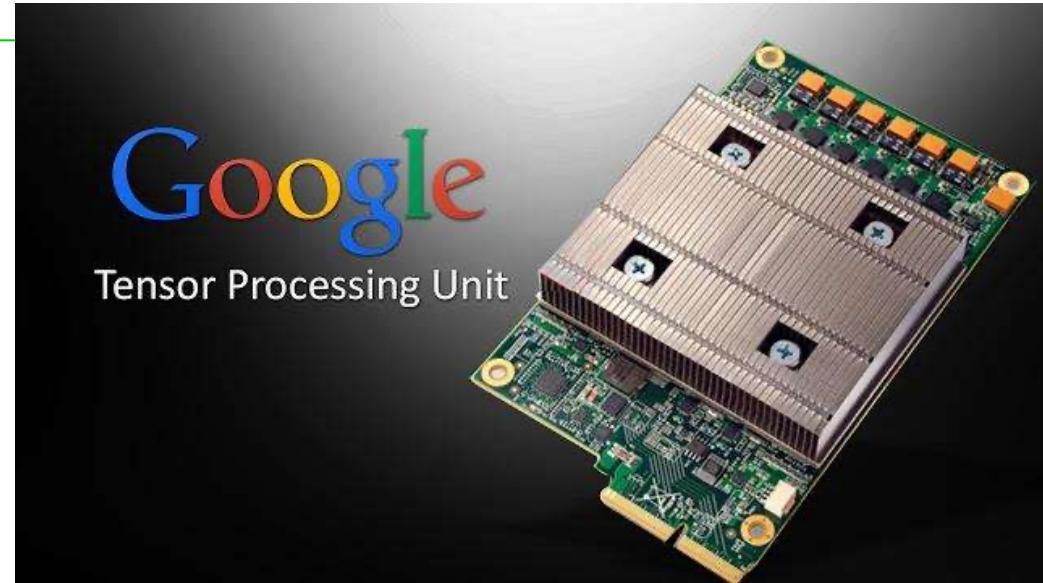




# 1 What is Software Reuse?



# Hardware is reused inherently

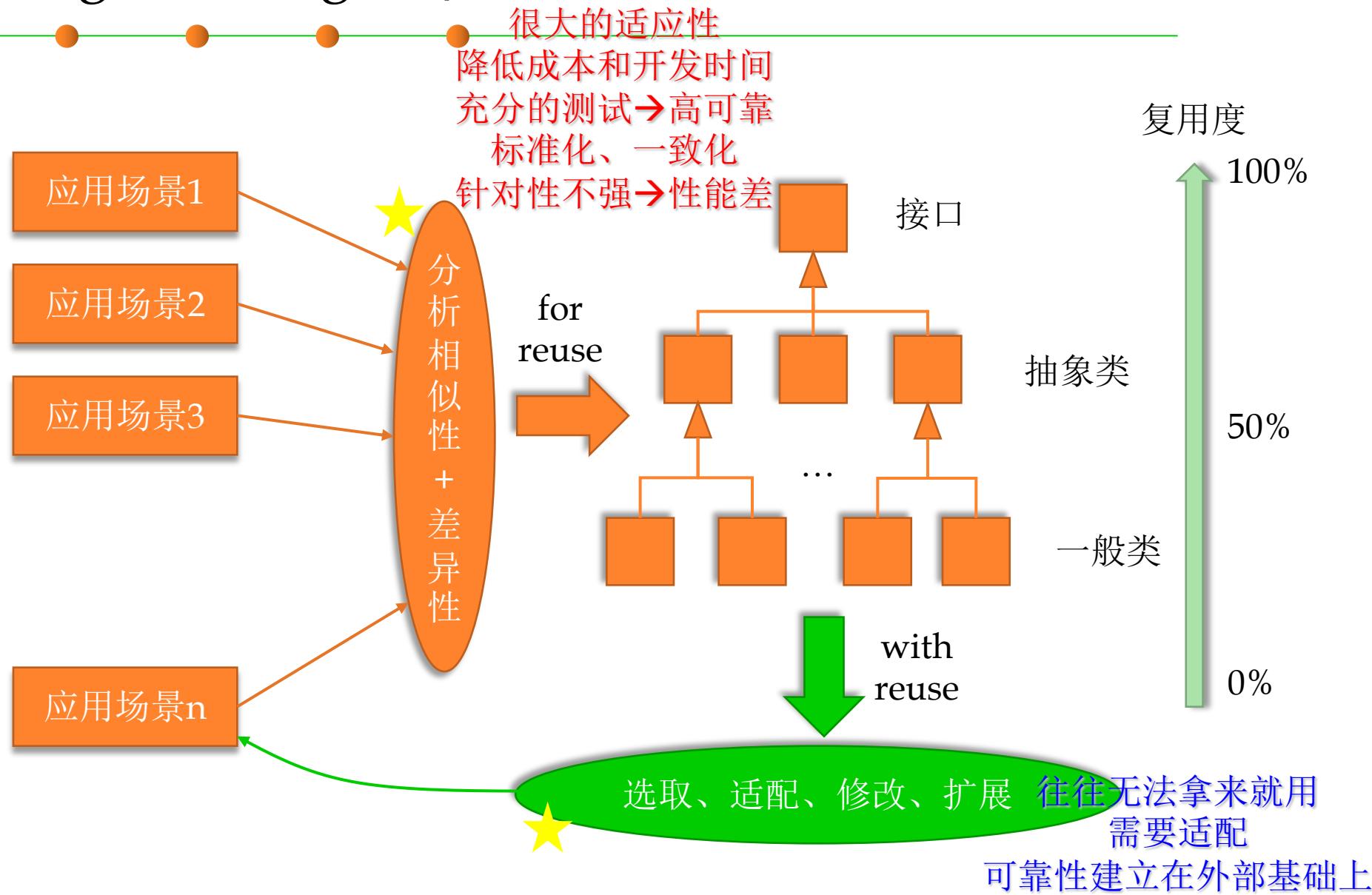


# Software reuse

---

- **Software reuse** is the process of implementing or updating software systems using existing software components.
- **Two perspectives of software reuse**
  - Creation: creating reusable resources in a systematic way (**programming for reuse** 面向复用编程: 开发出可复用的软件)
  - Use: reusing resources as building blocks for creating new systems (**programming with reuse** 基于复用编程: 利用已有的可复用软件搭建应用系统)
- **Why reuse?**
  - “The drive to create reusable rather than transitory artifacts has aesthetic and intellectual as well as **economic motivations** and is part of man’s desire for immortality.
  - It distinguishes man from other creatures and civilized from primitive societies” (Wegner, 1989).

# Programming for/with reuse



# Recall Lab2

- 在Lab2中：你开发了一个基于泛型的抽象接口**Graph<L>**，定义了支持图结构的ADT
- 针对该ADT，用两种不同的Rep，开发了两个不同的实现  
**ConcreteVertexGraph<L>**和**ConcreteEdgeGraph<L>**

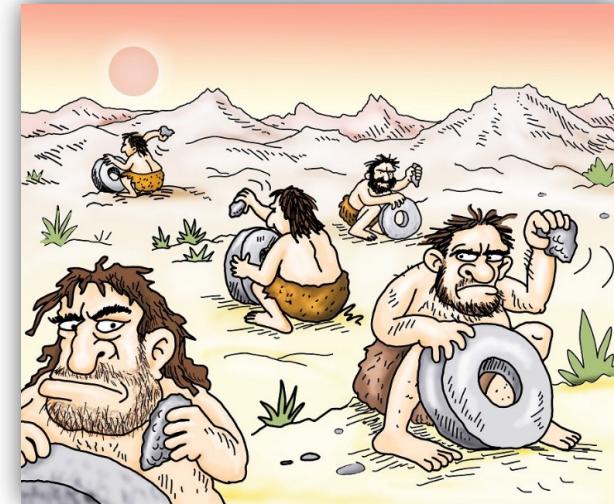
Programming for reuse  
面向复用编程：开发出可复用的软件

- 进而，你利用该ADT及其两个实现，完成了两个应用的开发：
  - Poetic Walks
  - Friendship Social Network

Programming with reuse  
基于复用编程：利用已有的可复用软件搭建应用系统

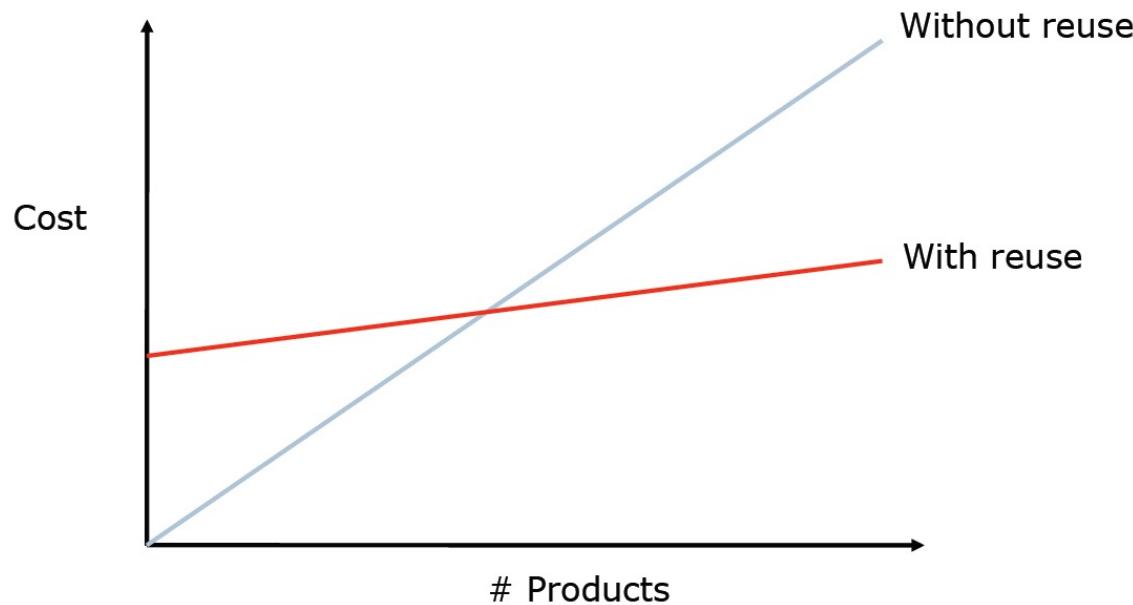
# Why reuse?

- **Reuse is cost-effective and with timeliness** 降低成本和开发时间
  - Increases software productivity by shortening software production cycle time (software developed faster and with fewer people)
  - Does not waste resources to needlessly "reinvent-the-wheel"
  - Reduces cost in maintenance (better quality, more reliable and efficient software can be produced)
- **Reuse produces reliable software** 经过充分测试，可靠、稳定
  - Reusing functionality that has been around for a while and is debugged is a foundation for building on stable subsystems
- **Reuse yields standardization** 标准化，在不同应用中保持一致
  - Reuse of GUI libraries produces common look-and-feel in applications.
  - Consistency with regular, coherent design.



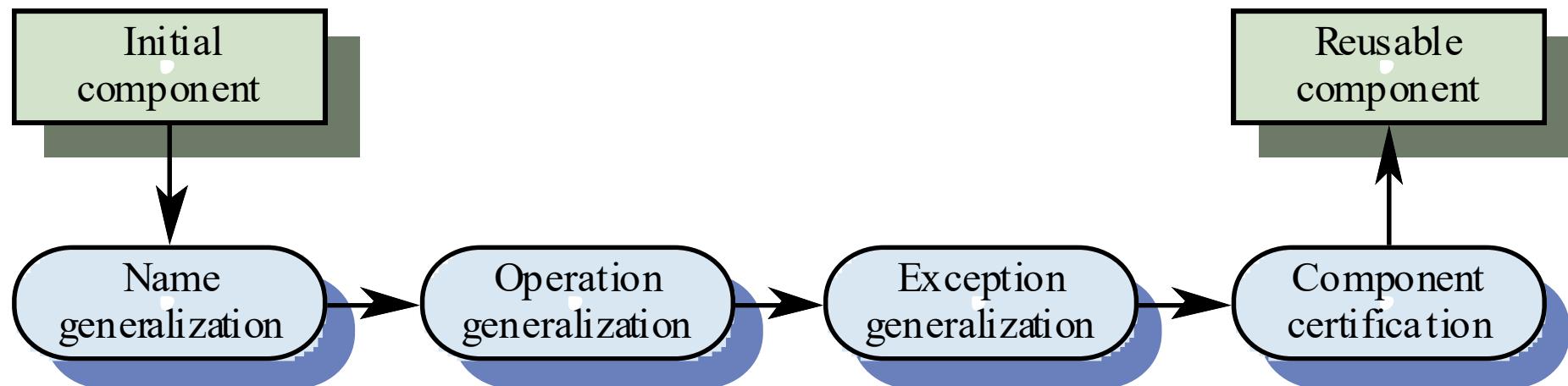
# Reuse costs

- **Reusable components** should be designed and built in a *clearly* defined, *open* way, with *concise* interface specifications, *understandable* documentation, and an eye towards *future use*. 做到这些，需要代价
- **Reuse is costly:** it involves spans organizational, technical, and process changes, as well as the cost of tools to support those changes, and the cost of training people on the new tools and changes. 不仅  
**program for reuse**代价高，**program with reuse**代价也高



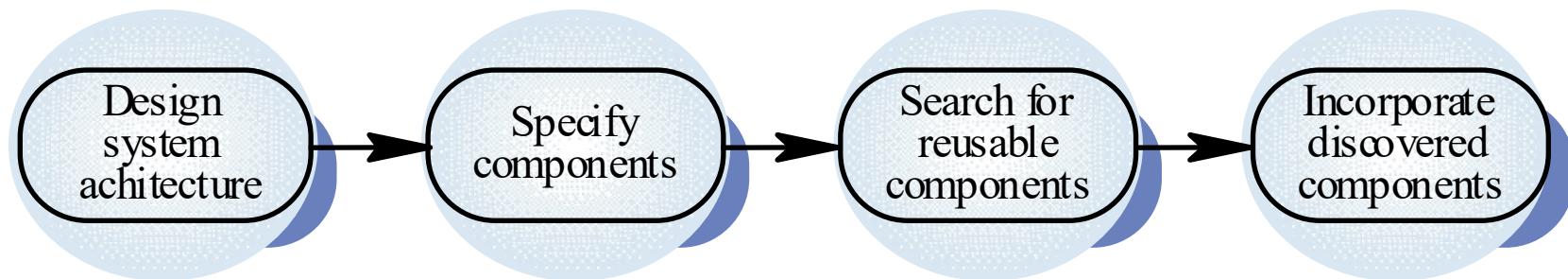
# Development for reuse: 开发可复用的软件

- The development cost of reusable components is higher than the cost of specific equivalents. This extra reusability enhancement cost should be an organization rather than a project cost. **开发成本高于一般软件的成本：要有足够高的适应性**
- Generic components may be less space-efficient and may have longer execution times than their specific equivalents. **性能差些：针对更普适场景，缺少足够的针对性**



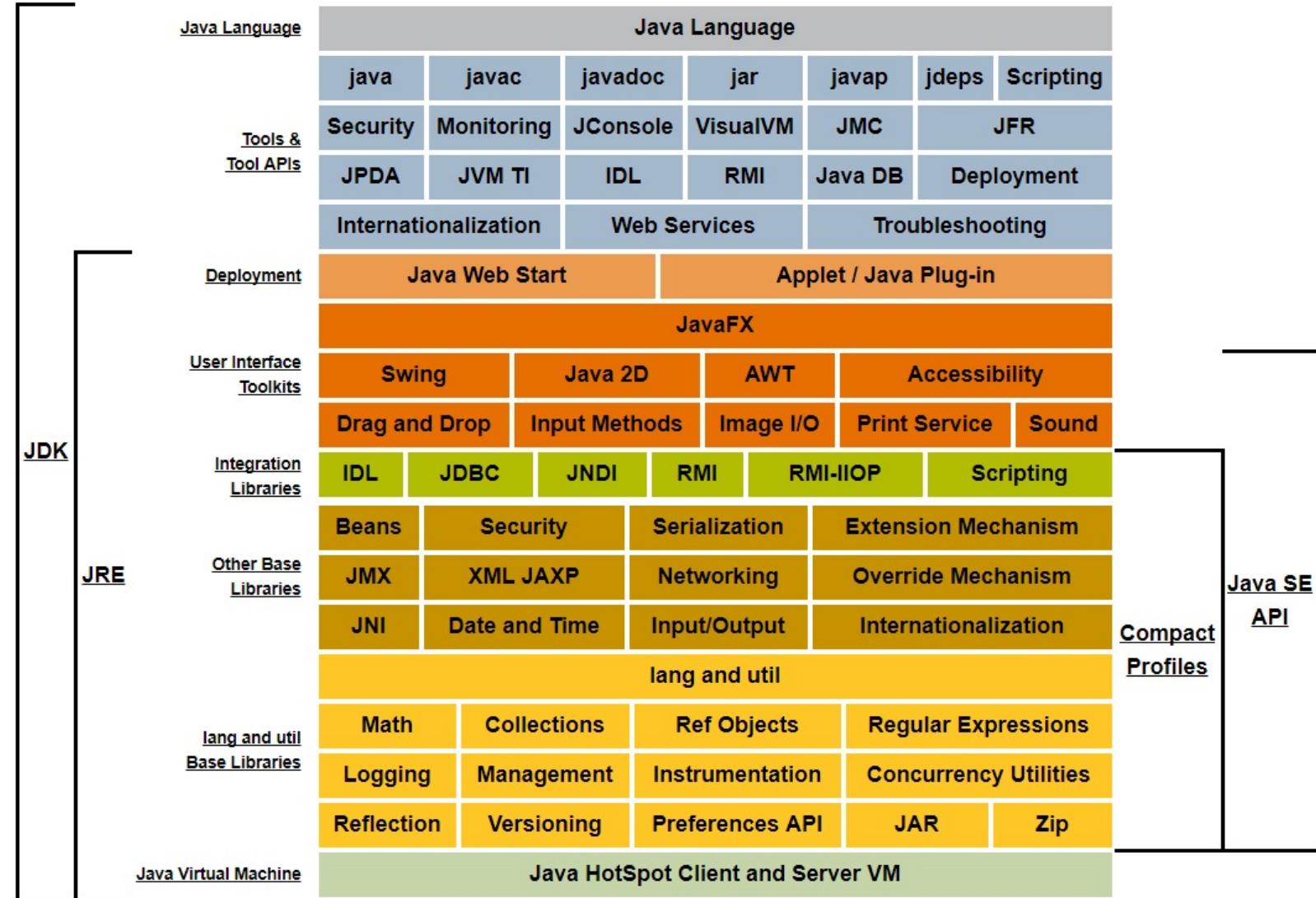
# Development with reuse: 使用已有软件进行开发

- Component management tools, such as repositories, for architectures, designs, documentation, and code must be developed and maintained. 可复用软件库，对其进行有效的管理



- A key issue: **adaptation** 往往无法拿来就用，需要适配
  - **Extra** functionality may have to be added to a component. When this has been **added**, the new component may be made available for reuse.
  - **Unneeded** functionality may be **removed** from a component to improve its performance or reduce its space requirements
  - The implementation of some component operations may have to be **modified**.

# Reusable Libraries and APIs in JDK

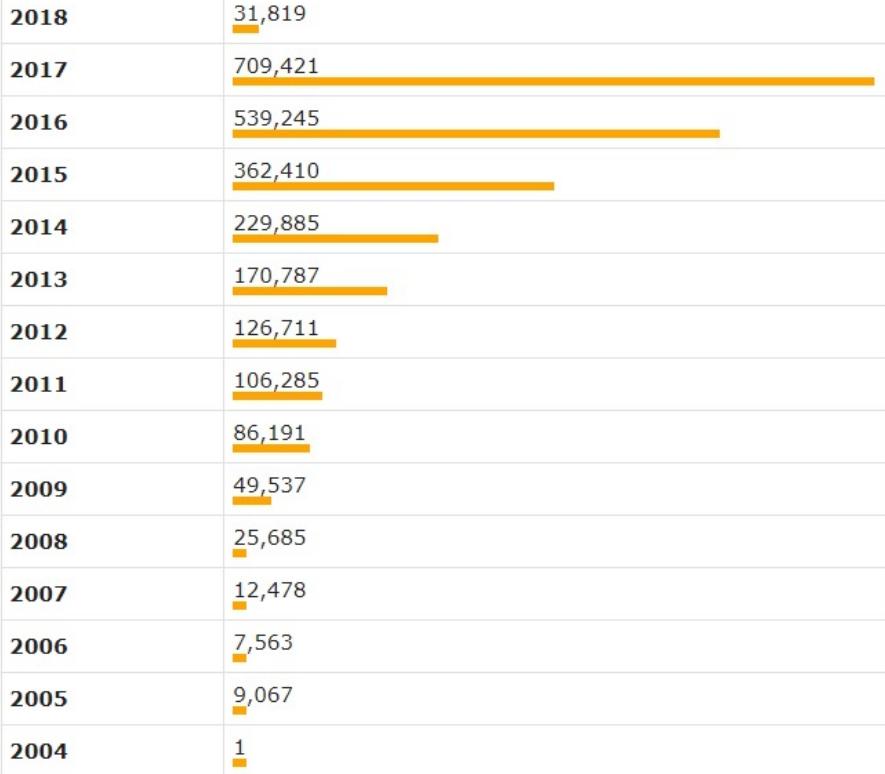


# Rich third-party libraries and APIs in Java

 **Central Repository**  
<http://central.maven.org/maven2/>

<b>URL</b>	<a href="http://central.maven.org/maven2/">http://central.maven.org/maven2/</a>
<b>Jars</b>	2,467,181 indexed jars

**Published Jars by Year**



Year	Jars
2018	31,819
2017	709,421
2016	539,245
2015	362,410
2014	229,885
2013	170,787
2012	126,711
2011	106,285
2010	86,191
2009	49,537
2008	25,685
2007	12,478
2006	7,563
2005	9,067
2004	1

Home » log4j » log4j » 1.2.17

 **Apache Log4j » 1.2.17**  
 Apache Log4j 1.2

<b>License</b>	Apache 2.0
<b>Categories</b>	Logging Frameworks
<b>Organization</b>	Apache Software Foundation
<b>HomePage</b>	<a href="http://logging.apache.org/log4j/1.2/">http://logging.apache.org/log4j/1.2/</a>
<b>Date</b>	(May 26, 2012)
<b>Files</b>	<a href="#">pom (21 KB)</a> <a href="#">bundle (478 KB)</a> <a href="#">View All</a>
<b>Repositories</b>	<a href="#">Central</a> <a href="#">Apache Releases</a> <a href="#">Redhat GA</a> <a href="#">Sonatype Releases</a> <a href="#">Spring Plugins</a>
<b>Used By</b>	11,493 artifacts

Maven Gradle SBT Ivy Grape Leiningen Buildr

```
<!-- https://mvnrepository.com/artifact/log4j/log4j -->
<dependency>
  <groupId>log4j</groupId>
  <artifactId>log4j</artifactId>
  <version>1.2.17</version>
</dependency>
```

Include comment with link to declaration

# Rich RubyGems

The screenshot shows the RubyGems website interface. At the top left is a search bar with placeholder text "Search Gems...". Next to it is an "Advanced Search" button. Below the search bar, a large orange banner features the text "寻找、安装以及发布 RubyGems" (Search, Install, and Publish RubyGems). On the right side of the banner, there's a "Stats" section with three metrics: "TOTAL GEMS 139,663", "TOTAL USERS 118,801", and "18,947,744,300" total downloads. Below the banner, a red bar displays the total download count "18,947,733,215" with the note "总下载次数" (Total Downloads) underneath. To the right of this bar is a button labeled "安装 RubyGems" (Install RubyGems). The main content area is titled "ALL TIME MOST DOWNLOADED" and shows a list of the top 10 most downloaded gems, each accompanied by a colored bar chart indicating its total downloads.

Gem	Downloads
bundler	217,398,175
multi_json	189,349,841
rake	186,861,120
rack	185,815,759
json	178,094,053
mime-types	173,826,045
activesupport	165,309,018
rspec-core	164,624,569
diff-lcs	164,590,605
rspec-expectations	163,709,140



## 2 How to measure “reusability”?



# Measure resuability

---

- **How frequently can a software asset be reused in different application scenarios?** 复用的机会有多频繁？复用的场合有多少？
  - The more chance an asset is used, the higher reusability it has.
  - Write once, reuse multiple times.
  
- **How much are paid for reusing this asset?** 复用的代价有多大？
  - Cost to buy the asset and other mandatory libraries 搜索、获取
  - Cost for adapting and extending it 适配、扩展
  - Cost for instantiating it 实例化
  - Cost for changing other parts of the system that interact with it 与软件其他部分的互连的难度

# Reusability

- Reusability implies some explicit management of build,  
packaging, distribution, installation, configuration, deployment,  
maintenance and upgrade issues.
- A software asset with high reusability should:
  - Brief (small size) and Simple (low complexity) 小、简单
  - Portable and Standard Compliance 与标准兼容
  - Adaptable and Flexible 灵活可变
  - Extensibility 可扩展
  - Generic and Parameterization 泛型、参数化
  - Modularity 模块化
  - Localization of volatile (changeable) design assumptions 变化的局部性
  - Stability under changing requirements 稳定
  - Rich documentation 丰富的文档和帮助



# 3 Levels and morphology of reusable components



# Levels of Reuse



- A reusable component may be code 最主要的复用是在代码层面
  - Most prevalent: what most programmers relate with reuse
- But benefits result from a broader and higher-level view of what can be reused. 但软件构造过程中的任何实体都可能被复用
  - Requirements 需求
  - Design and specifications 设计/规约spec
  - Data 数据
  - Test cases 测试用例
  - Documentation 文档

# What we concern in this lecture

- 
- Source code level: **methods, statements, etc**
  - Module level: **class and interface**
  - Library level: **API**
    - Java Library, .jar
  - Architecture level: **framework 框架**

# Types of Code Reuse

---

- **White box reuse 白盒复用：源代码可见，可修改和扩展**
  - Reuse of code when code itself is available. Usually requires some kind of modification or adaptation 复制已有代码到正在开发的系统，进行修改
  - **Pro:** You can customize the module to fit the specific situation, this allows reuse in more situations 可定制化程度高
  - **Con:** You now own the customized result, so it adds to your code complexity. You require intrinsic knowledge on component internals. 对其修改增加了软件的复杂度，且需要对其内部充分的了解
- **Black box reuse 黑盒复用：源代码不可见，不能修改**
  - Reuse in the form of combining existing code by providing some “glue”, but without having to change the code itself - usually because you do not have access to the code 只能通过API接口来使用，无法修改代码
  - **Pro:** Simplicity and Cleanliness 简单，清晰
  - **Con:** Many times it is just not possible 适应性差些

# Formats for reusable component distribution

- Forms:

- Source code
- Package such as .jar, .gem, .dll,

以及最重要的：自己  
日积月累出来的可复  
用代码库

- Sources of reusable software components:

- Internal (corporate) code libraries      组织的内部代码库 (Guava)
- Third party libraries      第三方提供的库 (Apache)
- Built-in language libraries      语言自身提供的库 (JDK)
- Code samples from tutorials, examples, books, etc. 代码示例
- Local code guru or knowledgeable colleague      来自同事
- Existing system code      已有系统内的代码
- Open source products (be sure to follow any licensing agreements)      开源软件的代码



# (1) Source code reuse



# Reusing Code - Lowest Level

- 
- Copy/paste parts/all into your program
  
  - Maintenance problem
    - Need to correct code in multiple places
    - Too much code to work with (lots of versions)
  - High risk of error during process
  - May require knowledge about how the used software works
  - Requires access to source code
  
  - 相关研究1：如何从互联网上快速找到需要的代码片段？
  - 反向研究：如何从源代码中检测出克隆代码(clone code)？

# Example code search: [grepcode.com](http://grepcode.com)

The screenshot shows the grepcode.com search interface. At the top, there is a navigation bar with links to Stack Trace Search, Eclipse, IntelliJ, Contact, FAQ, and social media icons for LinkedIn, GitHub, and Facebook. The main search bar contains the query "hashset". Below the search bar are three tabs: "Stack Trace" (selected), "Search", and "Types". The "Types" tab is active, displaying a list of Java classes found in various repositories. The first result is `java.util.HashSet`, which is described as implementing the Set interface. It lists several projects and their versions where this class is used, including gwt-user (2.7.0, 2.7.0-rc1, 2.7.0-beta1, 2.6.1, 2.6.1-rc2, 2.6.1-rc1, 2.6.0, 2.6.0-rc4, 2.5.1, 2.5.0, 2.4.0, 2.3.0, 2.2.0, 2.1.1, 2.1.0, 2.0.4, ...), emul (0.15, 0.14, 0.13, 0.12, 0.11, 0.10, 0.9, 0.8.1, 0.8, 0.7.2, 0.7, 0.6, 0.5, 0.4), gwt-servlet (2.7.0, 2.7.0-rc1, 2.6.1, 2.6.1-rc2, 2.6.1-rc1, 2.6.0, 2.6.0-rc4, 2.6.0-rc3, 2.5.1, 2.5.0, 2.4.0, 2.3.0, 2.2.0, 2.1.1, 2.1.0, 2.0.4, ...), and org.apache.servicemix.bundles.gwt-user (2.6.0\_1, 2.4.0\_1). Other results listed include `com.thaiopensource.validate.mns.Hashset` and `com.thaiopensource.validate.nrl.Hashset`, both associated with jing (20091111, 20120724.0.0, 20091111.0.0) and wicketstuff-jing (1.11, 1.10, 1.9, 1.8, 1.7, 1.6). The final result shown is `com.thaiopensource.validate.nvdl.Hashset`, described as a utility class that stores a set of objects, also associated with jing (20091111).

Stack Trace Search | Eclipse | IntelliJ | Contact | FAQ | LinkedIn | GitHub | Facebook

grepcode™

hashset

Stack Trace Search | Eclipse | IntelliJ | Contact | FAQ | LinkedIn | GitHub | Facebook

grepcode™

Types Projects Methods

All Repositories

- JDK
- Maven-Central
- Cloudera
- Hyracks
- Java.net
- JBoss
- NetBeans
- Pentaho
- PrimeFaces
- SpringSource
- Eclipse-4.4.2
- Eclipse-4.4.1
- Eclipse-4.4.0
- Eclipse-4.3.1
- Eclipse-4.3
- Eclipse-4.2.2
- Eclipse-4.2
- Eclipse-3.7.2
- Eclipse-3.6.2
- Eclipse-Virgo
- EclipseLink
- GrepCode
- GrepCode-Eclipse

All Kinds

- Class
- Interface
- Enum
- Annotation

java.util.HashSet - This class implements the Set interface, backed by a hash table (actually a HashMap instance)

- gwt-user 2.7.0 2.7.0-rc1 2.7.0-beta1 2.6.1 2.6.1-rc2 2.6.1-rc1 2.6.0 2.6.0-rc4 2.5.1 2.5.0 2.4.0 2.3.0 2.2.0 2.1.1 2.1.0 2.0.4 ...
- emul 0.15 0.14 0.13 0.12 0.11 0.10 0.9 0.8.1 0.8 0.7.2 0.7 0.6 0.5 0.4
- gwt-servlet 2.7.0 2.7.0-rc1 2.6.1 2.6.1-rc2 2.6.1-rc1 2.6.0 2.6.0-rc4 2.6.0-rc3 2.5.1 2.5.0 2.4.0 2.3.0 2.2.0 2.1.1 2.1.0 2.0.4 ...
- org.apache.servicemix.bundles.gwt-user 2.6.0\_1 2.4.0\_1
- ...

com.thaiopensource.validate.mns.Hashset

- jing 20091111
- jing 20120724.0.0 20091111.0.0
- wicketstuff-jing 1.11 1.10 1.9 1.8 1.7 1.6
- jing 20150629VNU 20130806VNU
- ...

com.thaiopensource.validate.nrl.Hashset

- jing 20091111
- jing 20120724.0.0 20091111.0.0
- wicketstuff-jing 1.11 1.10 1.9 1.8 1.7 1.6
- jing 20030619
- ...

com.thaiopensource.validate.nvdl.Hashset - Utility class, stores a set of objects

- jing 20091111

# GitHub code search: [github.com/search](https://github.com/search)

The screenshot shows the GitHub code search interface. At the top, there's a navigation bar with links for Pull requests, Issues, Marketplace, and Explore. Below the navigation bar is an advanced search bar containing the query "hashmap language:Java". To the right of the search bar is a "Search" button. The main content area displays search results for "Showing 13,052,763 available code results". On the left, there's a sidebar with sections for Repositories (974), Code (13M), Commits (291K), Issues (28K), Wikis (8K), and Users (1). Another section lists Languages: Java (13,052,763), C++ (669,400), HTML (620,079), Smali (489,293), JavaScript (245,495), XML (176,562), Scala (151,977), Java Server Pages (143,091), PHP (104,555), and C (77,620). The main results area shows two examples of Java code. The first example is from the repository "leijiangping/history" in the file "UserDao.java". It shows code for initializing a HashMap and putting key-value pairs. The second example is from the same repository in the file "RedisCaptchaStore.java", showing code for returning keys from a Redis client.

hashmap language:Java

Showing 13,052,763 available code results

Sort: Recently indexed

Repositories 974

Code 13M

Commits 291K

Issues 28K

Wikis 8K

Users 1

Languages

Language	Count
Java	13,052,763
C++	669,400
HTML	620,079
Smali	489,293
JavaScript	245,495
XML	176,562
Scala	151,977
Java Server Pages	143,091
PHP	104,555
C	77,620

leijiangping/history – UserDao.java

```
1 package com.systoon.integration.user.dao;
2
3 import java.util.Collection;
4 import java.util.HashMap;
...
30     for (int i = 0; i < ids.length; i++) {
31         map = new HashMap<String, Object>();
32         map.put("disabled", disable ? 1 : 0);
33         map.put("id", ids[i]);
```

leijiangping/history – RedisCaptchaStore.java

```
98             return client.hkeys(CaptchaRedisKey);
99
100        });
101        return ret;
102    }
103
104    @Override
105    public void empty() {
106        // this.store = new HashMap();
107        deleteAll();
```

Advanced search Cheat sheet

# Searchcode: searchcode.com

regionmatches

search

lopers

Updates

searchcode server

Google Cloud — Use GCP products for free up to non-expiring usage limits. AD

About 598 results: "regionmatches"

Page 1 of 30

◀ Previous    Next ▶



Project tracking, teamwork & client reporting like you've never seen before.  
ads via Carbon

**RegionMatches.java in ManagedRuntimeInitiative** [git://github.com/GregBowyer/ManagedRuntimeInitiative.git](https://github.com/GregBowyer/ManagedRuntimeInitiative.git) | 41 lines | Java Show 3 matches

```
26. * @bug 4016509
27. * @summary test regionMatches corner case
30.
31. public class RegionMatches {
32.
33.     if (!s1.regionMatches(0,s2,0,Integer.MIN_VALUE))
34.         throw new RuntimeException("Integer overflow in RegionMatches");
35. }
```

**OldStringTest.java in android\_libcore** [https://github.com/PAmoto/android\\_libcore.git](https://github.com/PAmoto/android_libcore.git) | 525 lines | Java

```
141.
142.     public void test_regionMatchesILjava_lang_StringII() {
143.         assertFalse("Returned true for negative offset.", hw1.regionMatches(-1,
144.                     hw2, 2, 5));
145.         assertFalse("Returned true for negative offset.", hw1.regionMatches(2,
146.                     assertFalse("Returned true for toffset+len is greater than the length.",
147.                                hw1.regionMatches(5, hw2, 2, 6));
148.         assertFalse("Returned true for ooffset+len is greater than the length.",
149.                    hw1.regionMatches(2, hw2, 5, 6));
150.
151.
152.
153.     public void test_regionMatchesZILjava_lang_StringII() {
154.
155.
156.         assertFalse("Returned true for negative offset.", hw1.regionMatches(true,
157.                     -1, hw2, 2, 5));
```

Filter Results

Remove    Apply

Sources

Github 598

Languages

Filter Languages

Java 598

Filter Results



## (2) Module-level reuse: class/interface



Inheritance

Use

Composition/aggregation

Delegation/association

# Reusing classes

---

- A class is an atomic unit of code reuse
  - Source code not necessary, class file or **jar/zip**
  - Just need to include in the **classpath**
  - Can use **javap** tool to get a class's public method headers
- Documentation very important (Java API)
- Encapsulation helps reuse
- Less code to manage
- Versioning, backwards-compatibility still problem
- Need to package related classes together -- **Static Linking**

# Approaches of reusing a class: inheritance 继承

- Java provides a way of code reuse named **Inheritance**
  - Classes extend the properties/behavior of existing classes
  - In addition, they might **override** existing behavior
- No need to put dummy methods that just forward or delegate work
- Captures the real world better
- Usually need to design inheritance hierarchy before implementation
- Cannot cancel out properties or methods, so must be careful not to overdo it

# Approaches of reusing a class: delegation 委托

---

- **Delegation** is simply when one object relies on another object for some subset of its functionality (one entity passing something to another entity)
  - e.g. the Sorter is delegating functionality to some Comparator
- **Judicious delegation enables code reuse**
  - Sorter can be reused with arbitrary sort orders
  - Comparators can be reused with arbitrary client code that needs to compare integers
- **Explicit delegation:** passing the sending object to the receiving object
- **Implicit delegation:** by the member lookup rules of the language
- **Delegation** can be described as a low level mechanism for sharing code and data between entities.

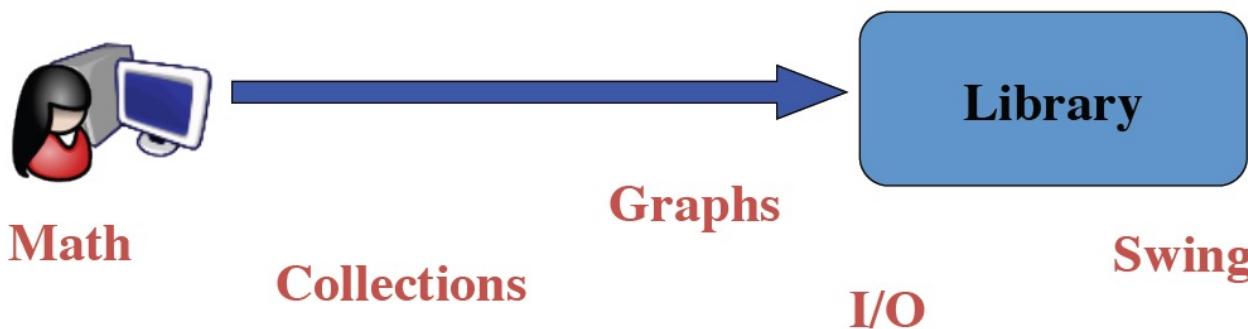


### (3) Library-level reuse: API/Package



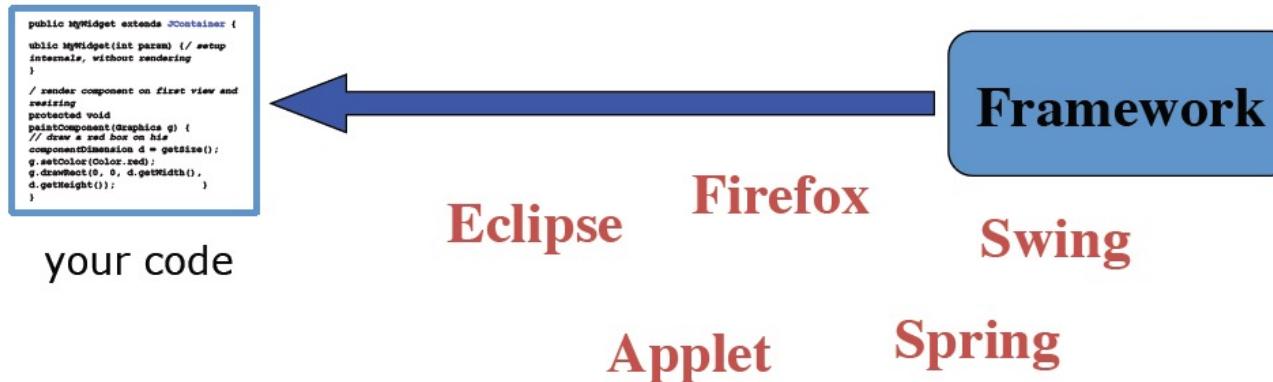
# Libraries

- Library: A set of classes and methods (APIs) that provide reusable functionality

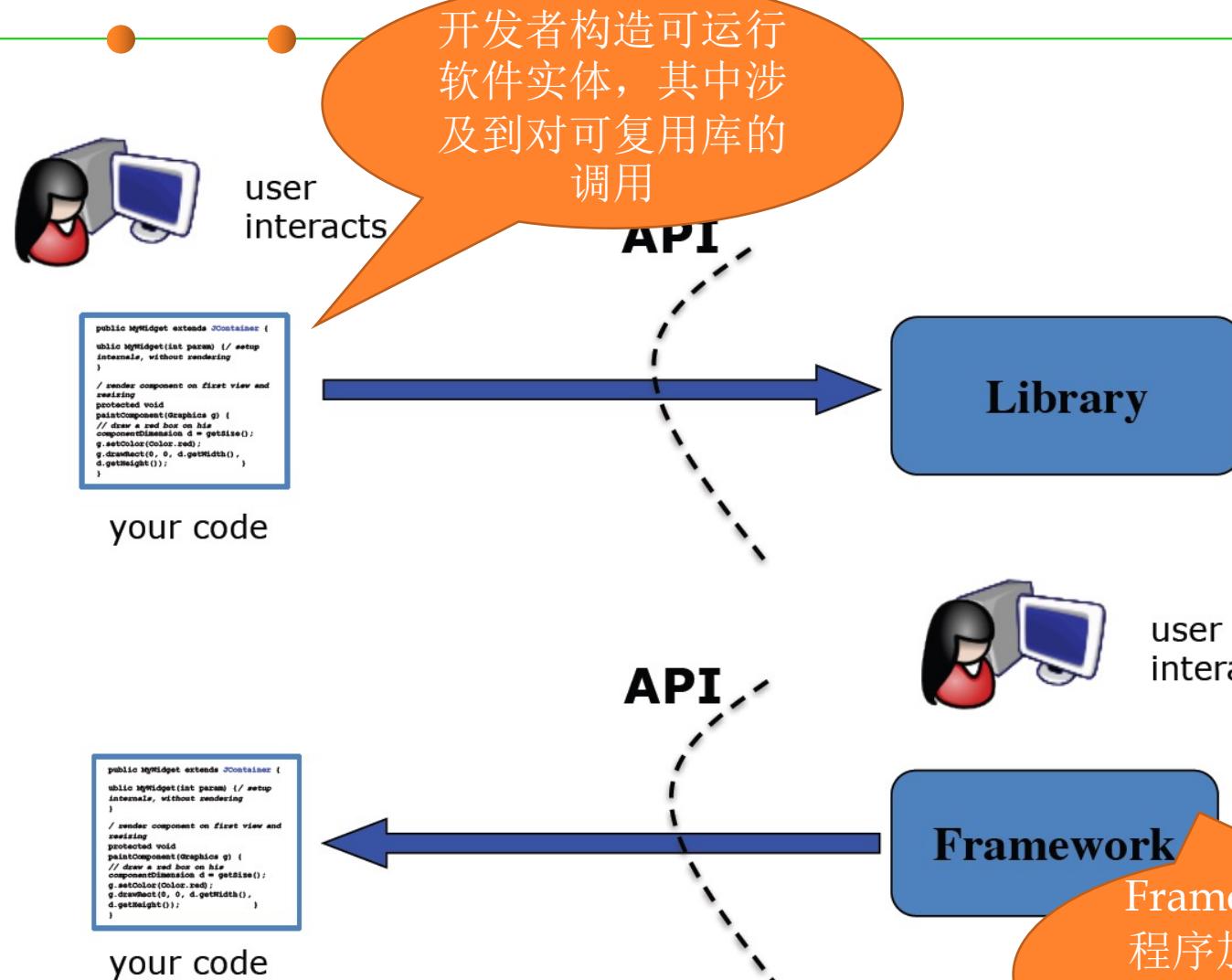


# Framework

- Framework: Reusable skeleton code that can be customized into an application
- Framework calls back into client code
  - The Hollywood principle: “Don’t call us. We’ll call you.”



# General distinction: Library vs. framework



开发者构造可运行软件实体，其中涉及到对可复用库的调用

Library

Framework

Framework作为主程序加以执行，执行过程中调用开发者所写的程序

# Characteristics of a good API

- 
- Easy to learn
  - Easy to use, even without documentation
  - Hard to misuse
  - Easy to read and maintain code that uses it
  - Sufficiently powerful to satisfy requirements
  - Easy to evolve
  - Appropriate to audience

# Guava: Google core libraries for Java

The screenshot shows the GitHub repository page for the 'guava' project under the 'google' organization. The repository has 4,673 commits, 6 branches, 76 releases, and 137 contributors. The license is Apache-2.0. The latest commit was made 6 days ago. The repository contains files like .gitattributes, .gitignore, .travis.yml, CONTRIBUTING.md, CONTRIBUTORS, COPYING, README.md, cycle\_whitelist.txt, javadoc-stylesheet.css, and pom.xml. Most commits are from netdb and ronshapiro.

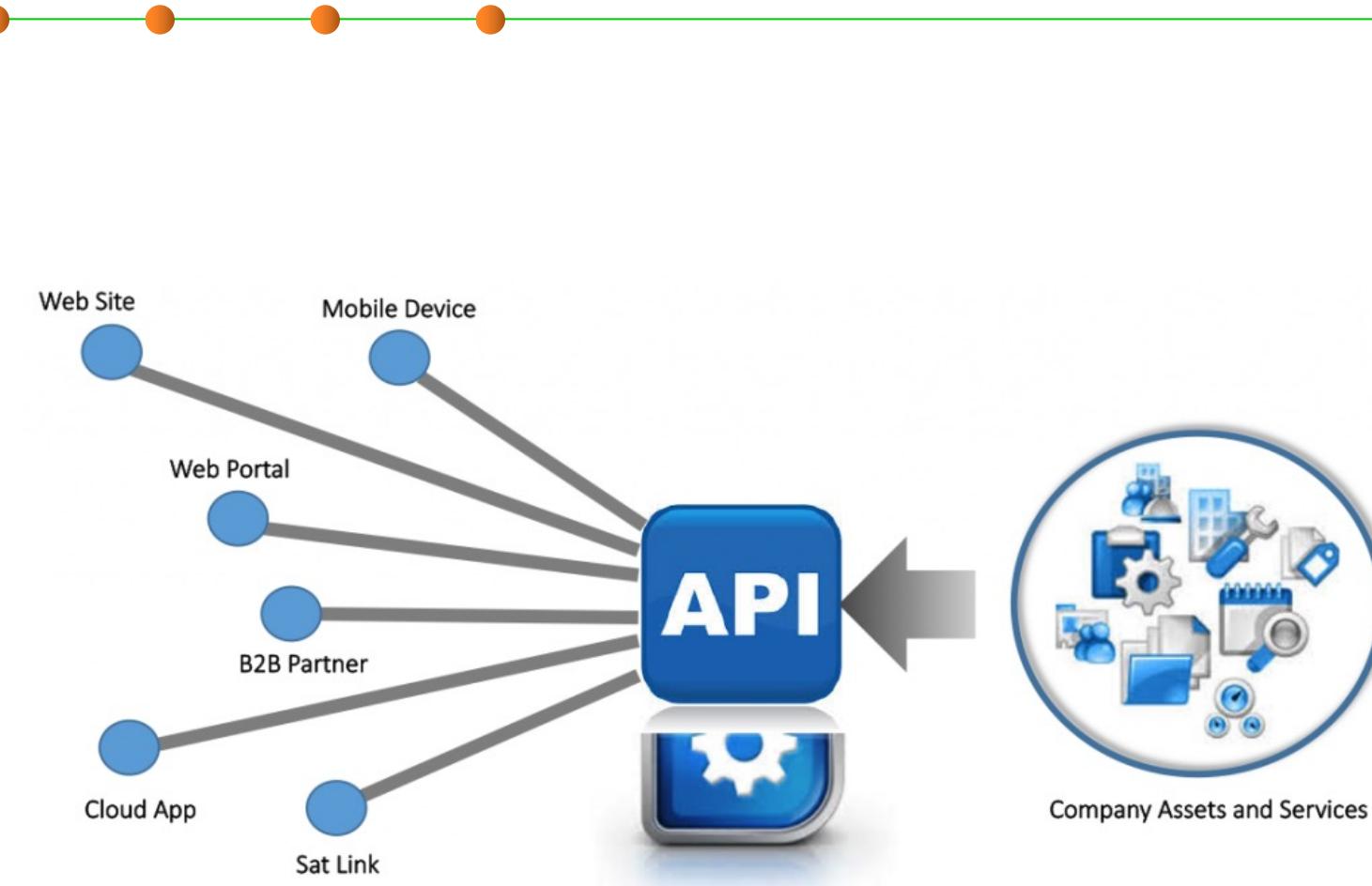
File	Commit Message	Time Ago
.gitattributes	Add a .gitattributes file to control line ending normalization, which...	3 years ago
.gitignore	Add .DS_Store to .gitignore.	2 years ago
.travis.yml	Hide Downloading/Downloaded lines from output.	a month ago
CONTRIBUTING.md	usage of American English spelling for "license"	2 years ago
CONTRIBUTORS	fix indentation	7 years ago
COPYING	fix indentation	7 years ago
README.md	Prepare for release 24.1.	18 days ago
cycle_whitelist.txt	Fix two bugs in MinMaxPriorityQueue (introduced in [] First is a bug ...)	a year ago
javadoc-stylesheet.css	Fix a couple of issues with JDK7 javadoc style by using a slightly cu...	5 years ago
pom.xml	Work around breakage in which release builds run maven-javadoc-plugin...	2 months ago

# Apache Commons

- Apache Commons is an Apache project focused on all aspects of reusable Java components.
  - <https://commons.apache.org>
  - [https://github.com/apache/commons-\\*](https://github.com/apache/commons-*)



# API on Web/Internet: Web Services/Restful APIs



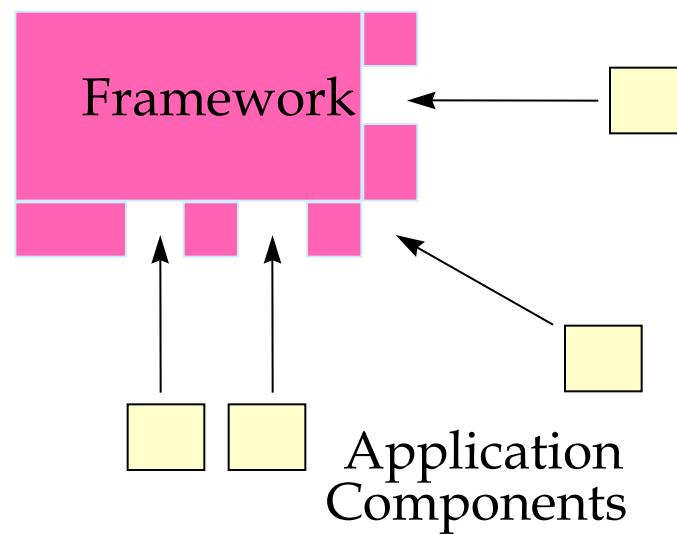


## (4) System-level reuse: Framework



# Application Frameworks

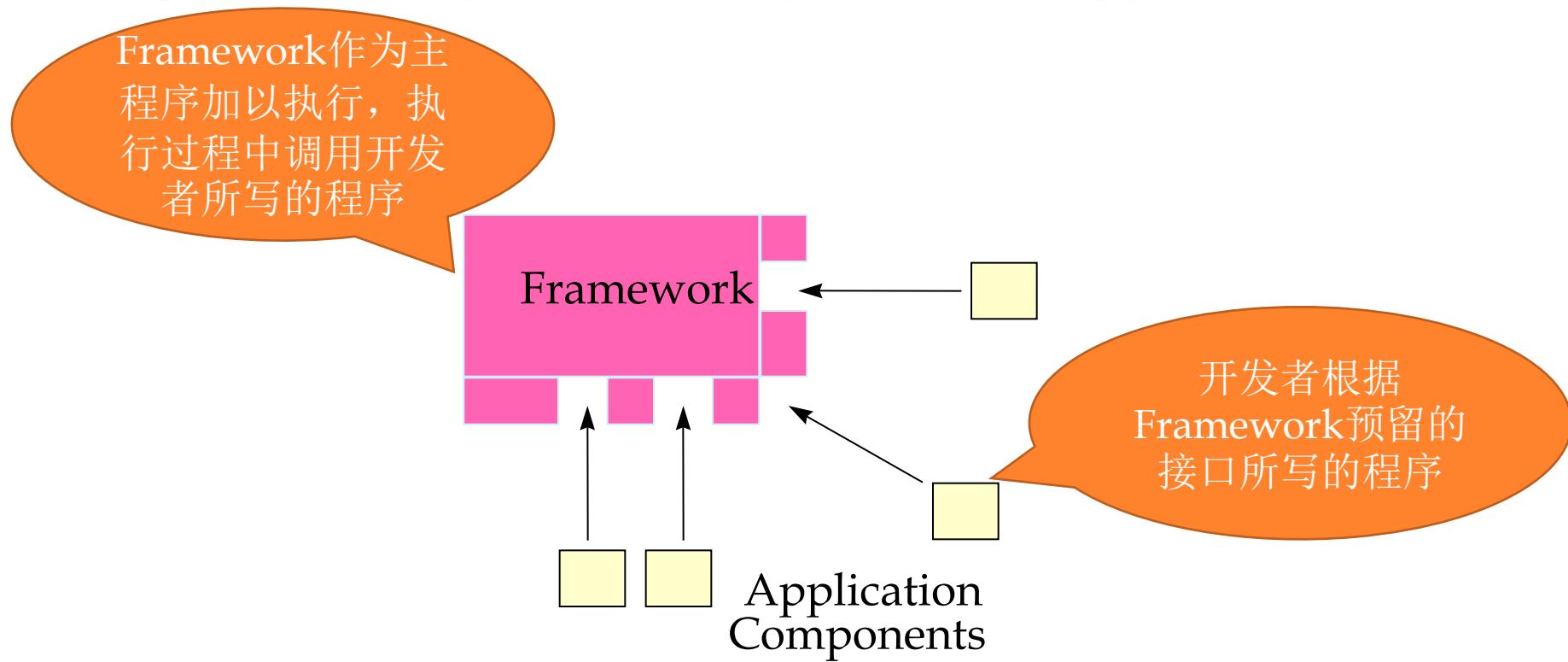
- Frameworks are sub-system design containing a collection of **abstract and concrete classes** along with interfaces between each class 框架：一组具体类、抽象类、及其之间的连接关系
  - 只有“骨架”，没有“血肉”
- A framework is an abstraction in which software providing **generic functionality** can be selectively changed by additional **user-written code**, thus providing application-specific software. 开发者根据 **framework** 的规约，填充自己的代码进去，形成完整系统



# Application Frameworks

骨骼清奇，天赋异禀

- Reusability leverages of the application domain knowledge and prior effort of experienced developers 领域知识的复用
  - Data processing, GUI, etc
  - 将framework看作是更大规模的API复用，除了提供可复用的API，还将这些模块之间的关系都确定下来，形成了整体应用的领域复用



# Framework Design

---

- **Frameworks differ from applications**
  - The level of abstraction is different as frameworks provide a solution for a family of related problems, rather than a single one.
  - To accommodate the family of problems, the framework is incomplete, incorporating hot spots and hooks to allow customization
  
- **Frameworks can be classified by the techniques used to extend them.**
  - Whitebox frameworks                      黑盒框架
  - Blackbox frameworks                      白盒框架

# White-box and Black-Box Frameworks

- 
- **Whitebox frameworks** 白盒框架，通过代码层面的继承进行框架扩展
    - Extensibility achieved through **inheritance** and **dynamic binding**.
    - Existing functionality is extended by **subclassing** framework base classes and **overriding** predefined hook methods
    - Often design patterns such as the template method pattern are used to **override** the hook methods.
  - **Blackbox frameworks** 黑盒框架，通过实现特定接口/**delegation**进行框架扩展
    - Extensibility achieved by defining **interfaces** for components that can be plugged into the framework.
    - Existing functionality is reused by defining components that **conform to a particular interface**
    - These components are integrated with the framework via **delegation**.



# 5 Designing reusable classes



# Designing reusable classes in OOP

---

- Encapsulation and information hiding
- Inheritance and overriding
- Polymorphism, subtyping and overloading
- Generic programming

Already introduced in  
section 3.4 OOP

- Behavioral subtyping and Liskov Substitution Principle (LSP)
- Delegation and Composition



# (1) Behavioral subtyping and Liskov Substitution Principle (LSP)



# Behavioral subtyping

- Subtype polymorphism: Different kinds of objects can be treated uniformly by client code. 子类型多态：客户端可用统一的方式处理不同类型的对象
  - If the type Cat is a subtype of Animal, then an expression of type Cat can be used wherever an expression of type Animal is used.

```
Animal a = new Animal();
Animal c1 = new Cat();
Cat c2 = new Cat();
```

在可以使用a的场景，都可以用c1和c2代替而不会有任何问题

a = c1;  
a = c2;

- Let  $q(x)$  be a property provable about objects  $x$  of type  $T$ , then  $q(y)$  should be provable for objects  $y$  of type  $S$  where  $S$  is a subtype of  $T$ .

— — Barbara Liskov

# Barbara Liskov

---



**Barbara Liskov (1939- )**

MIT

<http://www.pmg.csail.mit.edu/~liskov>

美国第一位计算机科学方向的女博士  
2008年图灵奖获得者

提出了第一个支持数据抽象的面向对象编程语言CLU，对现代主流语言如C++/Java/Python/Ruby/C#都有深远的影响。她所提炼出来的数据抽象思想，成为软件工程的重要精髓之一。

她提出的“Liskov替换原则”，是面向对象最重要的几大原则之一。

# Behavioral subtyping

- Compiler-enforced rules in Java (**static type checking**)
  - Subtypes can add, but not remove methods 子类型可以增加方法，但不可删
  - Concrete class must implement all undefined methods 子类型需要实现抽象类型中的所有未实现方法
  - Overriding method must return same type or subtype 子类型中重写的方法必须有相同或子类型的返回值或者符合co-variance的参数
  - Overriding method must accept the same parameter types 子类型中重写的方法必须使用同样类型的参数或者符合contra-variance的参数
  - Overriding method may not throw additional exceptions 子类型中重写的方法不能抛出额外的异常
- Also applies to specified behavior (methods):
  - Same or stronger invariants 更强的不变量
  - Same or weaker preconditions 更弱的前置条件
  - Same or stronger postconditions 更强的后置条件

Liskov  
Substitution  
Principle  
(LSP)

# Example 1 for Behavioral subtyping (LSP)

- Subclass fulfills the same invariants (and additional ones)
- Overridden method has the same pre- and post-conditions

```
abstract class Vehicle {  
  
    int speed, limit;  
    //@ invariant speed < limit;  
  
    //@ requires speed != 0;  
    //@ ensures speed < \old(speed)  
    void brake();  
}
```

```
class Car extends Vehicle {  
    int fuel;  
    boolean engineOn;  
    //@ invariant speed < limit;  
    //@ invariant fuel >= 0;  
  
    //@ requires fuel > 0 && !engineOn;  
    //@ ensures engineOn;  
    void start() { ... }  
    void accelerate() { ... }  
  
    //@ requires speed != 0;  
    //@ ensures speed < \old(speed)  
    void brake() { ... }  
}
```

# Example 2 for Behavioral subtyping (LSP)

- Subclass fulfills the same invariants (and additional ones)
- Overridden method start has weaker precondition
- Overridden method brake has stronger postcondition

```
class Car extends Vehicle {
    int fuel;
    boolean engineOn;
    // @ invariant speed < limit;
    // @ invariant fuel >= 0;
    // @ requires fuel > 0 && !engineOn;
    // @ ensures engineOn;
    void start() { ... }
    void accelerate() { ... }

    // @ requires speed != 0;
    // @ ensures speed < \old(speed)
    void brake() { ... }
}
```

```
class Hybrid extends Car {
    int charge;
    // @ invariant charge >= 0;
    // @ requires (charge > 0
    //             || fuel > 0) && !engineOn;
    // @ ensures engineOn;
    void start() { ... }
    void accelerate() { ... }

    // @ requires speed != 0;
    // @ ensures speed < \old(speed)
    // @ ensures charge > \old(charge)
    void brake() { ... }
}
```

# Behavioral subtyping (LSP)

- How about these two classes? Is LSP satisfied?

```
class Rectangle {
    int h, w;
    Rectangle(int h, int w) {
        this.h=h; this.w=w;
    }
    //methods
}
```

```
class Square extends Rectangle {
    Square(int w) {
        super(w, w);
    }
}
```

Is this **Square** a behavioral subtype of **Rectangle**?

```
class Rectangle {
    //@ invariant h>0 && w>0;
    int h, w;
    Rectangle(int h, int w) {
        this.h=h; this.w=w;
    }
    //methods
}
```

```
class Square extends Rectangle {
    //@ invariant h>0 && w>0;
    //@ invariant h==w;
    Square(int w) {
        super(w, w);
    }
}
```

# Behavioral subtyping (LSP)

- How about these two classes? Is LSP satisfied?

```
class Rectangle {  
    // @ invariant h>0 && w>0;  
    int h, w;  
    Rectangle(int h, int w) {  
        this.h=h; this.w=w;  
    }  
  
    // @ requires factor > 0;  
    void scale(int factor) {  
        w=w*factor;  
        h=h*factor;  
    }  
}
```

```
class Square extends Rectangle {  
    // @ invariant h>0 && w>0;  
    // @ invariant h==w;  
    Square(int w) {  
        super(w, w);  
    }  
}
```

Is this **Square** a behavioral subtype of **Rectangle**?

# Behavioral subtyping (LSP)



Is this **Square** a behavioral subtype of **Rectangle**?

```
class Rectangle {
    //@ invariant h>0 && w>0;
    int h, w;
    Rectangle(int h, int w) {
        this.h=h; this.w=w;
    }
    //@ requires factor > 0;
    void scale(int factor) {
        w=w*factor;
        h=h*factor;
    }

    //@ requires neww > 0;
    void setWidth(int neww) {
        w=neww;
    }
}
```

```
class Square extends Rectangle {
    //@ invariant h>0 && w>0;
    //@ invariant h==w;
    Square(int w) {
        super(w, w);
    }
}
```

```
class GraphicProgram {
    void scaleW(Rectangle r, int factor) {
        r.setWidth(r.getWidth() * factor);
    }
}
```



**Invalidates stronger invariant ( $w==h$ ) in subclass**

# Behavioral subtyping (LSP)

```

class Rectangle {
    //@ invariant h>0 && w>0;
    int h, w;
    Rectangle(int h, int w) {
        this.h=h; this.w=w;
    }
    //@ requires factor > 0;
    void scale(int factor) {
        w=w*factor;
        h=h*factor;
    }
    //@ requires neww > 0;
    //@ ensures w=neww
        && h not changed
    void setWidth(int neww) {
        w=neww;
    }
}

```

```

class Square extends Rectangle {
    //@ invariant h>0 && w>0;
    //@ invariant h==w;
    Square(int w) {
        super(w, w);
    }
    //@ requires neww > 0;
    //@ ensures w=neww && h=neww
    @Override
    void setWidth(int neww) {
        w=neww;
        h=neww;
    }
}

```

# Liskov Substitution Principle (LSP)

- LSP is a particular definition of a **subtyping relation**, called **(strong) behavioral subtyping** 强行为子类型化
- In programming languages, LSP is relied on the following restrictions:
  - **Preconditions** cannot be strengthened in a subtype. 前置条件不能强化
  - **Postconditions** cannot be weakened in a subtype. 后置条件不能弱化
  - **Invariants** of the supertype must be preserved in a subtype. 不变量要保持
  - **Contravariance** of method arguments in a subtype 子类型方法参数：逆变
  - **Covariance** of return types in a subtype. 子类型方法的返回值：协变
  - No new **exceptions** should be thrown by methods of the subtype, except where those exceptions are themselves subtypes of exceptions thrown by the methods of the supertype. 异常类型：协变 (This is to be discussed in Section 7-2)

# Covariance (协变)

父类型 → 子类型：越来越具体 specific  
 返回值类型：不变或变得更具体  
 异常的类型：也是如此。

- See this example:

```
class T {
    Object a() { ... }
}
```

```
class S extends T {
    @Override
    String a() { ... }
}
```



- More specific classes may have more specific return types
- This is called **covariance** of return types in the subtype.

```
class T {
    void b( ) throws Throwable { ... }
}

class S extends T {
    @Override
    void b( ) throws IOException { ... }
}

class U extends S {
    @Override
    void b( ) { ... }
}
```

- Every exception declared for the subtype's method should be a subtype of some exception declared for the supertype's method.

# Contravariance (反协变、逆变)

- What do you think of this code?

```
class T {
    void c( String s ) { ... }
}
```

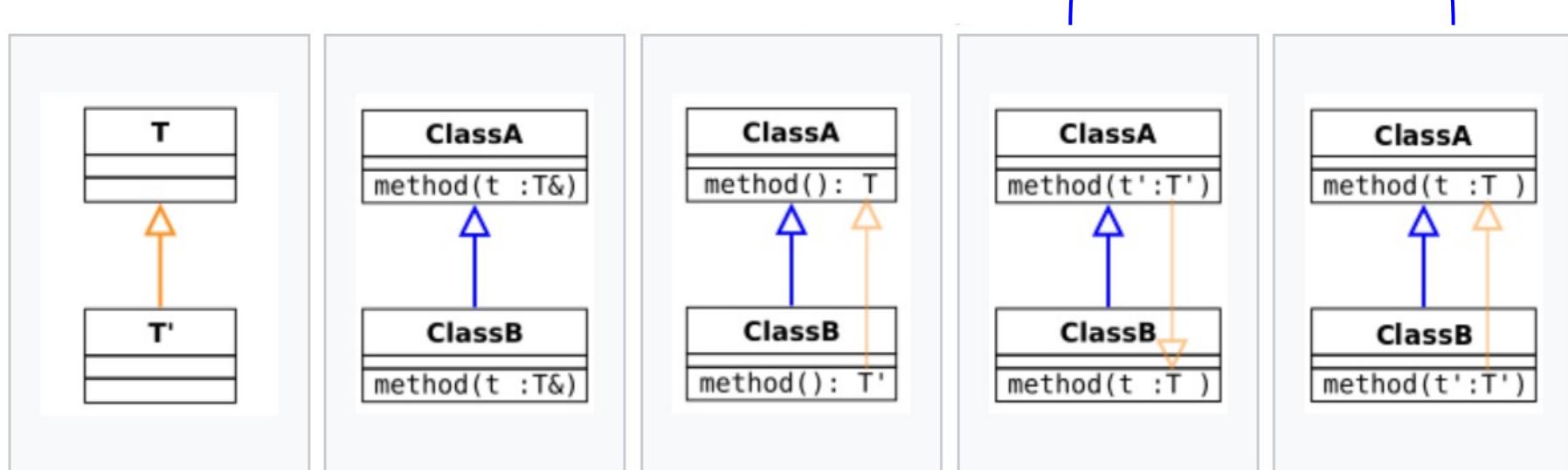
```
class S extends T {
    @Override
    void c( Object s ) { ... }
}
```

父类型 → 子类型：越来越具体 specific  
 参数类型：要相反的变化，要不变或越来越抽象

- Logically, it is called **contravariance** of method arguments in the subtype.
- This is actually not allowed in Java, as it would complicate the overloading rules. 目前Java中遇到这种情况，当作overload看待 ☺

The method c(Object) of type S must override or implement a supertype method

# Summary on subtyping and LSP



Subtyping of the argument/return type of the method.

*Invariance.* The signature of the overriding method is unchanged.

*Covariant return type.* The subtyping relation is in the same direction as the relation between ClassA and ClassB.

*Contravariant argument type.* The subtyping relation is in the opposite direction to the relation between ClassA and ClassB.

*Covariant argument type.* Not type safe.

# Co-variance and Contra-variance

- **Arrays are covariant** 协变的: given the subtyping rules of Java, an array of type  $T[]$  may contain elements of type  $T$  or any subtype of  $T$ .

```
Number[] numbers = new Number[2];
numbers[0] = new Integer(10);
numbers[1] = new Double(3.14);
```

```
Integer[] myInts = {1,2,3,4};
Number[] myNumber = myInts;
```

```
myNumber[0] = 3.14; //run-time error!
```

- At run-time Java knows that this array was actually instantiated as an array of integers which simply happens to be accessed through a reference of type  $Number[]$ .
  - 区分: Type of an object vs. Type of a reference

# Consider LSP for generics 泛型中的LSP

```
Object obj = new Object();
Integer integer = new Integer(10);
obj = integer;
```

```
void someMethod(Number n)
{ ... }

someMethod(new Integer(10));
someMethod(new Double(10.1));
```

- So, how about in generics?
- Generics are type invariant
  - ArrayList<String> is a subtype of List<String>
  - List<String> is not a subtype of List<Object>
- The type information for type parameters is discarded by the compiler after the compilation of code is done; therefore this type information is not available at run time.
- This process is called type erasure 类型擦除
- Generics are not covariant.

# What is type erasure?

- **Type erasure:** Replace all type parameters in generic types with their bounds or `Object` if the type parameters are unbounded. The produced bytecode, therefore, contains only ordinary classes, interfaces, and methods.

```
public class Node<T> {  
  
    private T data;  
    private Node<T> next;  
  
    public Node(T data, Node<T> next) {  
        this.data = data;  
        this.next = next;  
    }  
  
    public T getData() { return data; }  
    // ...  
}
```

```
public class Node {  
  
    private Object data;  
    private Node next;  
  
    public Node(Object data,  
               Node next) {  
        this.data = data;  
        this.next = next;  
    }  
  
    public Object getData() {  
        return data; }  
    // ...  
}
```

# An example

```
List<Integer> myInts = new ArrayList<Integer>();  
myInts.add(1);  
myInts.add(2);  
List<Number> myNums = myInts; //compiler error  
myNums.add(3.14);
```

```
static long sum(List<Number> numbers) {  
    long summation = 0;  
    for(Number number : numbers) {  
        summation += number.longValue();  
    }  
    return summation;  
}
```

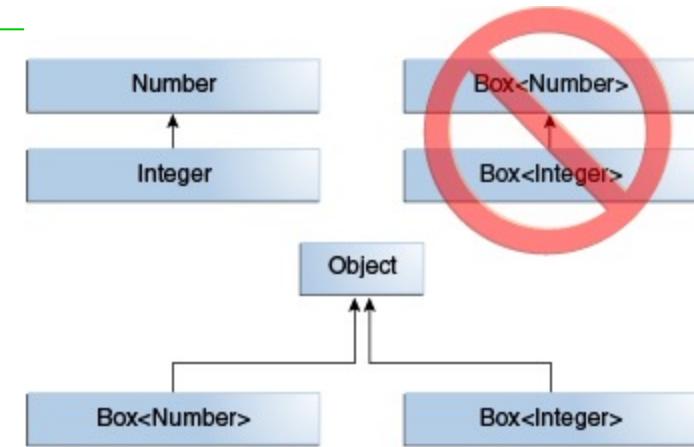
We cannot consider a list of integers to be subtype of a list of numbers.

That would be considered unsafe for the type system and compiler rejects it immediately.

```
List<Integer> myInts = asList(1,2,3,4,5);  
List<Long> myLongs = asList(1L, 2L, 3L, 4L, 5L);  
List<Double> myDoubles = asList(1.0, 2.0, 3.0, 4.0, 5.0);  
sum(myInts);  
sum(myLongs); //compiler error  
sum(myDoubles);
```

# Consider LSP for generics 泛型中的LSP

- **Box<Integer> is not a subtype of Box<Number> even though Integer is a subtype of Number.**
- Given two concrete types A and B (for example, Number and Integer), MyClass<A> has no relationship to MyClass<B>, regardless of whether or not A and B are related. The common parent of MyClass<A> and MyClass<B> is Object.
- For information on how to create a subtype-like relationship between two generic classes when the type parameters are related, see **Wildcards**.
  - <https://docs.oracle.com/javase/tutorial/java/generics/wildcards.html>



# Wildcards in Generics

- The unbounded wildcard type is specified using the wildcard character (?), for example, `List<?>`.
  - This is called a list of unknown type.
- There are two scenarios where an unbounded wildcard is a useful approach:
  - If you are writing a method that can be implemented using functionality provided in the `Object` class.
  - When the code is using methods in the generic class that don't depend on the type parameter. For example, `List.size` or `List.clear`.
  - In fact, `Class<?>` is so often used because most of the methods in `Class<T>` do not depend on `T`.

# Wildcards in Generics

```
public static void printList(List<Object> list) {  
    for (Object elem : list)  
        System.out.println(elem + " ");  
    System.out.println();  
}
```

The goal of `printList` is to print a list of any type, but it fails to achieve that goal – it prints only a list of `Object` instances; it cannot print `List<Integer>`, `List<String>`, `List<Double>`, and so on, because they are not subtypes of `List<Object>`.

To write a generic `printList` method, use `List<?>`

```
public static void printList(List<?> list) {  
    for (Object elem: list)  
        System.out.print(elem + " ");  
    System.out.println();  
}  
  
List<Integer> li = Arrays.asList(1, 2, 3);  
List<String> ls = Arrays.asList("one", "two", "three");  
printList(li);  
printList(ls);
```

# Wildcards in Generics

- Lower Bounded Wildcards: `<? super A>`
  - `List<Integer>`                           `List<? super Integer>`
  - The former matches a list of type `Integer` only, whereas the latter matches a list of any type that is a supertype of `Integer` such as `Integer`, `Number`, and `Object`.

- Upper Bounded Wildcards: `<? extends A>`

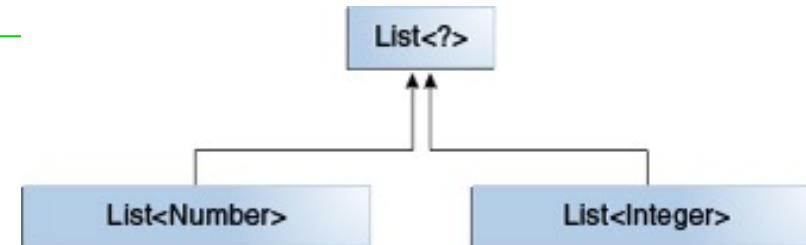
- `List<? extends Number>`

```
public static double sumOfList(List<? extends Number> list) {  
    double s = 0.0;  
    for (Number n : list)  
        s += n.doubleValue();  
    return s;  
}
```

```
List<Integer> li = Arrays.asList(1, 2, 3);  
List<Double> ld = Arrays.asList(1.2, 2.3, 3.5);
```

# Consider LSP for generics with wildcards

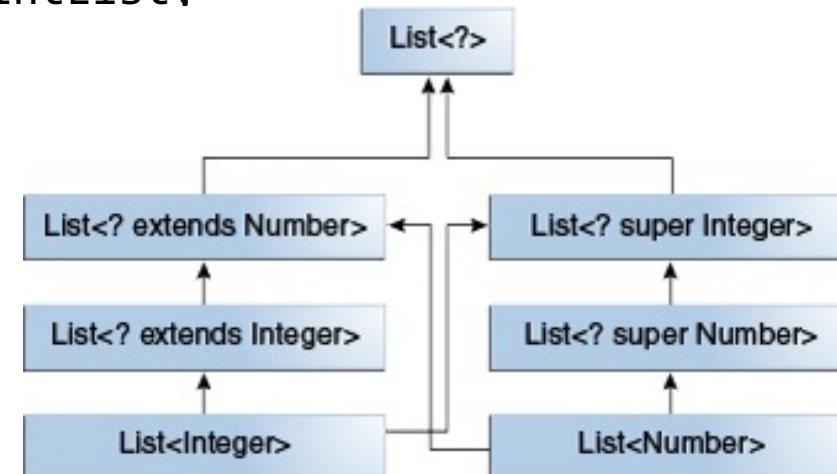
- `List<Number>` is a **subtype** of `List<?>`



- `List<Number>` is a **subtype** of `List<? extends Object>`
- `List<Object>` is a **subtype** of `List<? super String>`

```
List<? extends Integer> intList = new ArrayList<>();
```

```
List<? extends Number> numList = intList;
```



# Consider LSP for generics with wildcards

In the class `java.util.Collections`:

```
public static <T> void copy(  
    List<? super T> dest,  
    List<? extends T> src)
```

```
List<Number> source = new LinkedList<>();  
source.add(Float.valueOf(3));  
source.add(Integer.valueOf(2));  
source.add(Double.valueOf(1.1));
```

```
List<Object> dest = new LinkedList<>();
```

```
Collections.copy(dest, source);
```



## (2) Delegation and Composition



# A Sorting example

- Version A:

```
static void sort(int[] list, boolean ascending) {  
    ...  
    boolean mustSwap;  
    if (ascending) {  
        mustSwap = list[i] < list[j];  
    } else {  
        mustSwap = list[i] > list[j];  
    }  
    ...  
}
```

- Version B:

```
interface Comparator {  
    boolean compare(int i, int j);  
}  
final Comparator ASCENDING = (i, j) -> i < j;  
final Comparator DESCENDING = (i, j) -> i > j;  
  
static void sort(int[] list, Comparator cmp) {  
    ...  
    boolean mustSwap =  
        cmp.compare(list[i], list[j]);  
    ...  
}
```

# Interface Comparator<T>

- **int compare(T o1, T o2): Compares its two arguments for order.**
  - A comparison function, which imposes a total ordering on some collection of objects.
  - Comparators can be passed to a sort method (such as `Collections.sort` or `Arrays.sort`) to allow precise control over the sort order. Comparators can also be used to control the order of certain data structures (such as sorted sets or sorted maps), or to provide an ordering for collections of objects that don't have a natural ordering.
- 如果你的ADT需要比较大小，或者要放入`Collections`或`Arrays`进行排序，可实现`Comparator`接口并override `compare()`函数。

# Interface Comparator<T>

```
public class Edge {  
    Vertex s, t;  
    double weight;  
    ...  
}
```

```
public class EdgeComparator  
    implements Comparator<Edge>{  
  
    @Override  
    public int compare(Edge o1, Edge o2) {  
        if(o1.getWeight() > o2.getWeight())  
            return 1;  
        else if(.. == ..) return 0;  
        else return -1;  
    }  
}
```

```
public void sort(List<Edge> edges) {  
    Comparator comparator = new EdgeComparator();  
    Collections.sort(edges, comparator);  
}
```

# Interface Comparable<T>

- This interface imposes a total ordering on the objects of each class that implements it.
- This ordering is referred to as the class's natural ordering, and the class's `compareTo` method is referred to as its natural comparison method.
- 另一种方法：让你的ADT实现`Comparable`接口，然后override `compareTo()` 方法
- 与使用`Comparator`的区别：不需要构建新的`Comparator`类，比较代码放在ADT内部。
- This is not delegation any longer.

# Interface Comparable<T>



```
public class Edge implements Comparable<Edge> {
    Vertex s, t;
    double weight;
    ...
    public int compareTo(Edge o) {
        if(this.getWeight() > o.getWeight())
            return 1;
        else if(.. == ..) return 0;
        else return -1;
    }
}
```

# Delegation

---

- **Delegation** is simply when one object relies on another object for some subset of its functionality (one entity passing something to another entity) 委派/委托：一个对象请求另一个对象的功能
  - e.g. the Sorter is delegating functionality to some Comparator
- **Judicious delegation enables code reuse** 委派是复用的一种常见形式
  - Sorter can be reused with arbitrary sort orders
  - Comparators can be reused with arbitrary client code that needs to compare integers
- **Delegation** can be described as a low level mechanism for sharing code and data between entities.
  - **Explicit delegation**: passing the sending object to the receiving object
  - **Implicit delegation**: by the member lookup rules of the language

# A simple Delegation example

```
class A {  
    void foo() {  
        this.bar();  
    }  
  
    void bar() {  
        print("a.bar");  
    }  
}  
  
class B {  
    private A a; // delegation link  
  
    public B(A a) {  
        this.a = a;  
    }  
  
    void foo() {  
        a.foo(); // call foo() on the a-instance  
    }  
  
    void bar() {  
        print("b.bar");  
    }  
}  
  
A a = new A();  
B b = new B(a); // establish delegation between two objects
```

# Using delegation to extend functionality

- Consider `java.util.List`

我需要一个能log的List

```
public interface List<E> {
    public boolean add(E e);
    public E remove(int index);
    public void clear();
    ...
}
```

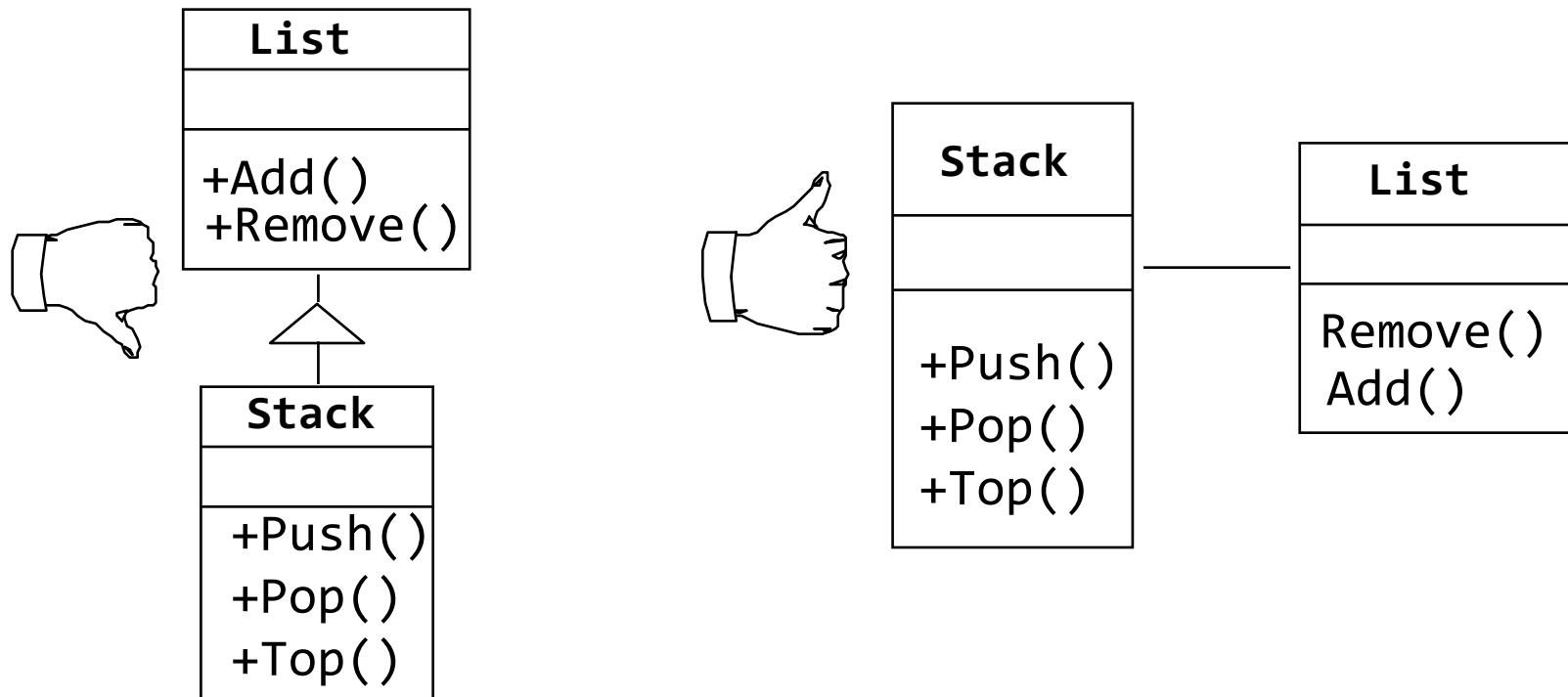
- Suppose we want a list that logs its operations to the console...

- The `LoggingList` is composed of a `List`, and delegates (the non-logging) functionality to that `List`.

```
public class LoggingList<E> implements List<E> {          这里实现动态绑定
    private final List<E> list;
    public LoggingList<E>(List<E> list) { this.list = list; }
    public boolean add(E e) {
        System.out.println("Adding " + e);
        return list.add(e);  这里进行功能委派
    }
    public E remove(int index) {
        System.out.println("Removing at " + index);
        return list.remove(index);
    }
}
```

# Delegation vs. Inheritance

- **Inheritance:** Extending a Base class by a new operation or overwriting an operation.
- **Delegation:** Catching an operation and sending it to another object.
- Many design patterns use a combination of inheritance and delegation.



# Delegation vs. Inheritance

- Lab2的P2中，要复用P1中开发出的Graph<L>完成社交网络相关功能

```
public class FriendshipGraph {  
    private Graph<Person> graph = Graph.empty();  
    public void addVertex(Person p) {  
        graph.add(p);  
    }  
    ...  
    public void getDistance(Person a, Person b) {  
        //Graph<L>不具备该功能，需要自己实现  
    }  
}
```

# Delegation vs. Inheritance

- Lab2的P2中，要复用P1中开发出的Graph<L>完成社交网络相关功能

```
public class FriendshipGraph<Person>
    extends ConcreteEdgeGraph<Person> {
    @Override
    public void addVertex(Person p) {
        ...
        super.add(p);
    }
    ...
    public void getDistance(Person a, Person b) {
        //Graph<L>不具备该功能，需要自己实现
    }
}
```

# Replace Inheritance with Delegation

---

- **Problem:** You have a subclass that uses only a portion of the methods of its superclass (or it's not possible to inherit superclass data). **如果子类只需要复用父类中的一小部分方法**
- **Solution:** Create a field and put a superclass object in it, delegate methods to the superclass object, and get rid of inheritance. **可以不需要使用继承，而是通过委派机制来实现**
- **In essence, this refactoring** splits both classes and makes the superclass the helper of the subclass, not its parent.
  - Instead of inheriting all superclass methods, the subclass will have only the necessary methods for delegating to the methods of the superclass object. **一个类不需要继承另一个类的全部方法，通过委托机制调用部分方法**
  - A class does not contain any unneeded methods inherited from the superclass. **从而避免大量无用的方法**

# Replace Inheritance with Delegation

```
class RealPrinter {  
    void print() {  
        System.out.println("Printing Data");  
    }  
}  
  
class Printer extends RealPrinter {  
    void print(){  
        super.print();  
    }  
}  
  
Printer printer = new Printer();  
printer.print();
```

Inheritance

Delegation

```
class RealPrinter {  
    void print() {  
        System.out.println("The Delegate");  
    }  
}  
  
class Printer {  
    RealPrinter p = new RealPrinter();  
  
    void print() {  
        p.print();  
    }  
}  
  
Printer printer = new Printer();  
printer.print();
```

# Composite over inheritance principle

- Or called **Composite Reuse Principle (CRP)**
  - Classes should achieve polymorphic behavior and code reuse by their composition (by containing instances of other classes that implement desired functionality) rather than inheritance from a base or parent class.
  - It is better to compose what an object can do (**has\_a** or **use\_a**) than extend what it is (**is\_a**).
- **Delegation** can be seen as a reuse mechanism at the **object level**, while **inheritance** is a reuse mechanism at the **class level**. “委托”发生在**object**层面，而“继承”发生在**class**层面

# CRP example

- An `Employee` class has a method for computing the employee's annual bonus:

```
class Employee {  
    Money computeBonus() {... // default computation}  
    ...  
}
```

- Different subclasses of `Employee`: `Manager`, `Programmer`, `Secretary`, etc. may want to override this method to reflect the fact that some types of employees get more generous bonuses than others:

```
class Manager extends Employee {  
    @Override  
    Money computeBonus() {... // special computation}  
    ...  
}
```

# CRP example

- There are several problems with this solution.
  - All **Manager** objects get the same bonus. What if we wanted to vary the bonus computation among managers? — — To introduce a special subclass of **Manager**?

```
class SeniorManager extends Manager {  
    @Override  
    Money computeBonus() {... // more special computation}  
    ...  
}
```

- What if we wanted to change the bonus computation for a particular employee? For example, what if we wanted to promote Smith from **Manager** to **SeniorManager**?
- What if we decided to give all managers the same bonus that programmers get? Should we copy and paste the computation algorithm from **Programmer** to **Manager**?

# CRP Example

- The problem is: the bonus calculator for each employer may be different and be changed frequently.
  - The relation between Employee and bonus calculator should be in the object level instead of class level. 核心问题: 每个Employee对象的奖金计算方法都不同，在object层面而非class层面。
- A CRP solution:

```
class Manager {  
    ManagerBonusCalculator mbc = new ManagerBonusCalculator();  
    Money computeBonus() {  
        return mbc.computeBonus();  
    }  
}
```

And so does for class  
Programmer and others

```
class ManagerBonusCalculator {  
    Money computeBonus {... // special computation}  
}
```

# CRP example: more general design

```
class Employee {  
    BonusCalculator bc;  
    ...  
}
```

Delegation: 委托

```
interface BonusCalculator {  
    Money computeBonus();  
}
```

Inheritance: 继承

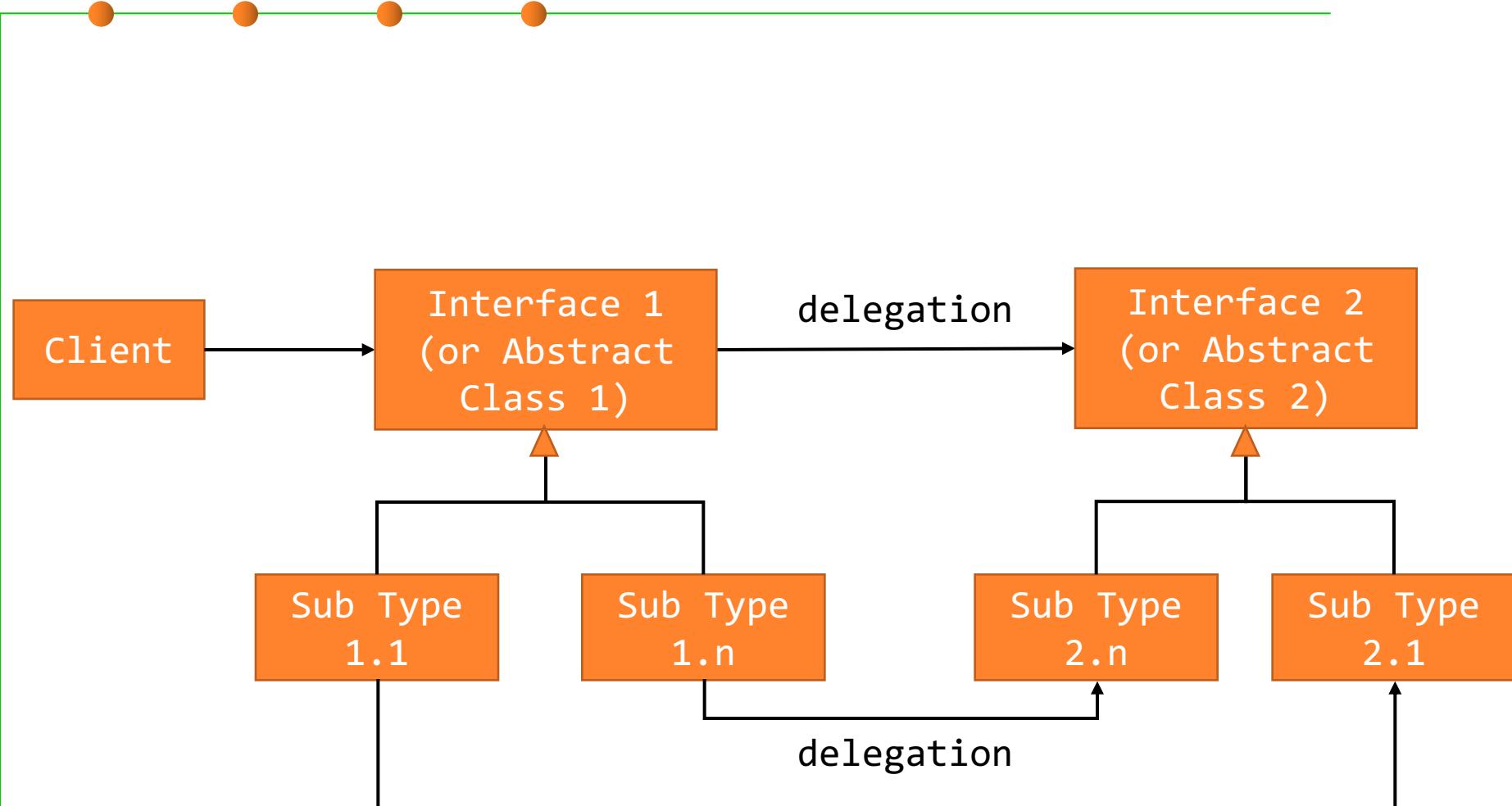
```
class Manager extends Employee {  
    Money computeBonus() {  
        bc = new ManagerBonusCalculator();  
        return bc.computeBonus();  
    }  
}
```

Implement: 接口实现

Delegation: object层面的委托

```
class ManagerBonusCalculator implements BonusCalculator {  
    Money computeBonus {... // special computation}  
}
```

# CRP example: more general design



# Composite over inheritance principle 更普适的



- 一个假象的场景：

- 你要开发一套动物ADT，各种不同种类的动物，每类动物有自己的独特“行为”，某些行为可能在不同类型的动物之间复用。
- 考虑到生物学和AI的进展，动物的“行为”可能发生会发生变化；

- 例如：

- 行为：飞、叫、...
- 动物：既会飞又会叫的鸭子、天鹅；不会飞但会叫的猫、狗；...
- 有10余种“飞”的方式、有10余种“叫”的方式；而且持续增加

```
class duck {
    void fly() {...//飞法1}
    void quack() {...//叫法1}
}
```

```
class swan {
    void fly() {...//飞法2}
    void quack() {...//叫法2}
}
```

直接面向具体类型动物的编程：类  
缺陷：存在大量的重复；不易变化

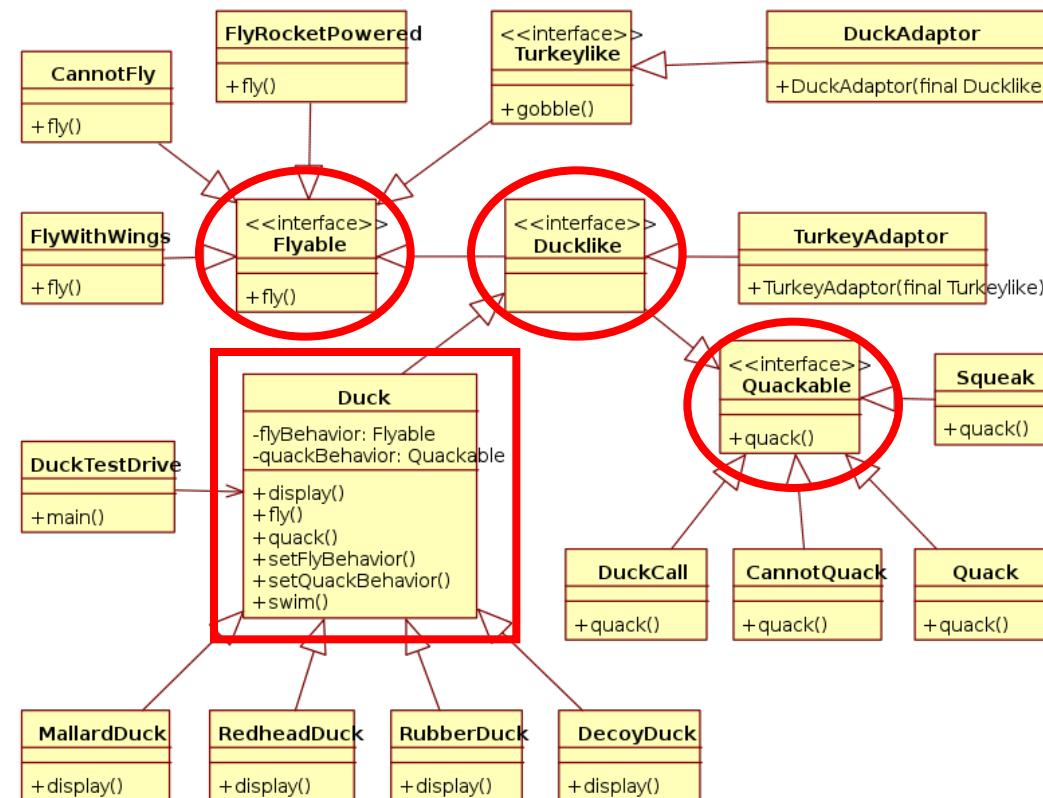
# Composite over inheritance principle 更普适的

```
class flyableAnimal {  
    //实现通用的飞法  
    void fly() {...//飞法1}  
    abstract void quack();  
}  
  
class duckLikeAnimal extends flyableAnimal {  
    @Override //如果某类动物飞法不同，则override重写新“飞法”  
    void fly() {...//飞法2}  
    @Override //针对该动物的叫法，写具体实现；不会叫则保持实现体为空即可  
    void quack() {...//叫法1}  
}
```

通过inheritance实现对某些通用行为的复用  
缺点：需要针对“飞法”设计复杂的继承关系树；不能同时支持针对“叫法”的继承；动物行为发生变化时，继承树要随之变化。

# Composite over inheritance principle 更普适的

- An implementation of composition over inheritance typically begins with the creation of various interfaces representing the behaviors that the system must exhibit.
- Classes implementing the identified interfaces are built and added to business domain classes as needed.
- Thus, system behaviors are realized without inheritance.
- 使用接口定义系统必须对外展示的不同侧面的行为
- 接口之间通过extends实现行为的扩展（接口组合）
- 类implements组合接口
- 从而规避了复杂的继承关系



# Composite over inheritance principle

```
interface Flyable {  
    public void fly();  
}
```

两个接口，定义抽象行为

```
interface Quackable {  
    public void quack();  
}
```

```
class FlyWithWings implements Flyable {  
    @Override  
    public void fly() {  
        System.out.println("fly with wings");  
    }  
}
```

```
class Quack implements Quackable {  
    @Override  
    public void quack() {  
        System.out.println("quack Like duck");  
    }  
}
```

接口的具体实现，可以有多种方式，在具体类中实现具体行为

# Composite over inheritance principle

```
interface Ducklike extends Flyable, Quackable {}
```

接口的组合，定义了行为的组合

```
public class Duck implements Ducklike {
    Flyable flyBehavior;
    Quackable quackBehavior;

    void setFlyBehavior(Flyable f) {
        this.flyBehavior = f;
    }
    void setQuackBehavior(Quackable q) {
        this.quackBehavior = q;
    }
}
```

Delegation

```
@Override
public void fly() {
    this.flyBehavior.fly();
}
@Override
public void quack() {
    this.quackBehavior.quack();
}
```

通过  
Delegation实  
现具体行为

从组合接口中派  
生具体类

设置  
Delegation  
对象实例

对接口编程

**Client code:**

```
Flyable f = new FlyWithWings();
Quackable q = new Quack();
```

```
Duck d = new Duck();
d.setFlyBehavior(f);
d.setQuackBehavior(q);
d.fly();
d.quack();
```

# Types of delegation

- **Use** (A **use** B)
- **Association** (A **has** B)
- **Composition/aggregation** (A **owns** B)
  - 可以认为Composition/Aggregation是Association的两种具体形态
- This classification is in terms of the “**coupling degree**” between the delegate and delegator.



**Use**  
**Composition**  
**Aggregation**  
**Association**

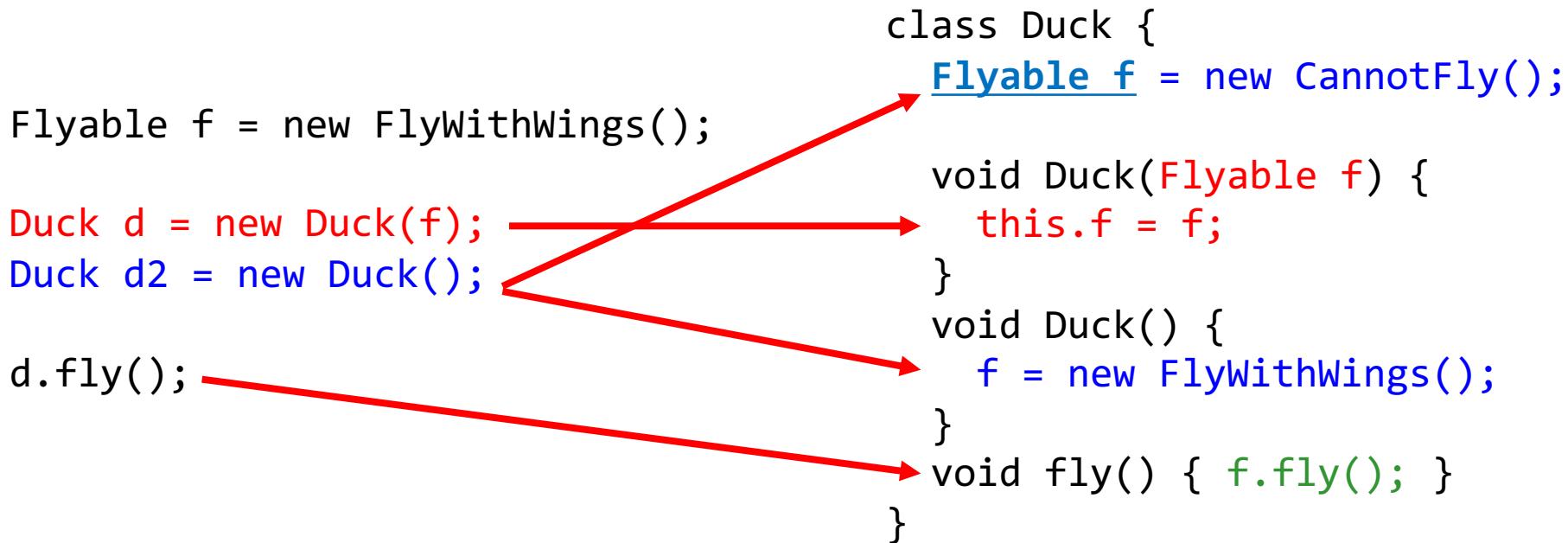
# (1) Dependency: 临时性的delegation

- The simplest form of using classes is **calling its methods**;
- This form of relationship between two classes is called “**uses-a**” relationship, in which one class makes use of another without actually incorporating it as a property. -it may, for example, be a parameter or used locally in a method.
- **Dependency**: a **temporary** relationship that an object requires other objects (suppliers) for their implementation.

```
Flyable f = new FlyWithWings();    class Duck {  
Quackable q = new Quack();          //no field to keep Flyable object  
  
Duck d = new Duck();  
d.fly(f);  
d.quack(q);  
                    void fly(Flyable f) {  
                        f.fly();  
                    }  
                }
```

## (2) Association: 永久性的 delegation

- **Association:** a **persistent** relationship between classes of objects that allows one object instance to cause another to perform an action on its behalf.
  - **has\_a:** one class has another as a property/instance variable
  - This relationship is structural, because it specifies that objects of one kind are connected to objects of another and does not represent behavior.



### (3) Composition: 更强的association , 但难以变化

- Composition is a way to combine simple objects or data types into more complex ones.
  - `is_part_of`: one class has another as a property/instance variable
  - Implemented such that an object contains another object.

```
Duck d = new Duck(); → class Duck {  
    Flyable f = new FlyWithWings();  
  
d.fly(); → void fly() {  
    f.fly();  
}  
}
```

## (4) Aggregation: 更弱的association , 可动态变化

- **Aggregation:** the object exists outside the other, is created outside, so it is passed as an argument to the constructor.
  - has\_a

```
Flyable f = new FlyWithWings();
```

```
Duck d = new Duck(f);
```

```
d.fly();
```

```
d.setFlyBehavior(new CannotFly());
```

```
d.fly();
```

```
class Duck {  
    Flyable f;  
  
    void Duck(Flyable f) {  
        this.f = f;  
    }  
  
    void setFlyBehavior(f) {  
        this.f = f;  
    }  
  
    void fly() { f.fly();}  
}
```

# Composition vs. Aggregation

---

```
public class WebServer {  
    private HttpListener listener;  
    private RequestProcessor processor;  
    public WebServer(HttpListener listener, RequestProcessor processor) {  
        this.listener = listener;  
        this.processor = processor;  
    }  
}  
  
public class WebServer {  
    private HttpListener listener;  
    private RequestProcessor processor;  
    public WebServer() {  
        this.listener = new HttpListener(80);  
        this.processor = new RequestProcessor(“/www/root”);  
    }  
}
```

Which is Composition?  
Which is Aggregation?

# Types of delegation

- Use (A **use one or multiple B**)
- Association (A **has one or multiple B**)
- Composition/aggregation (A **owns one or multiple B**)
- 都支持1对多的delegation——

```
class Professor {  
    List<Student> ls;           //永久保存delegation关系  
  
    void enroll(Student s) {  
        this.ls.add(s);          //建立delegation关系  
    }  
    void evaluate() {  
        double score = 0;  
        for(Student s : ls)  
            score += s.evaluate(this); //逐个delegate  
    }  
}
```

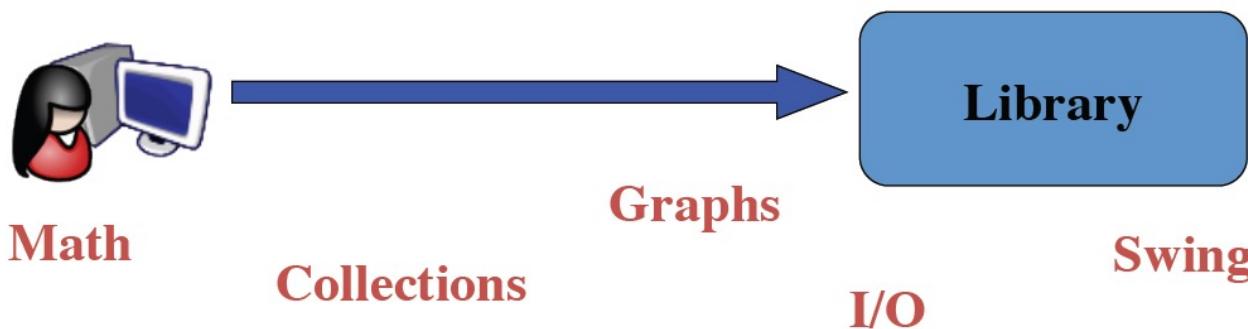


# 6 Designing system-level reusable API libraries and Frameworks



# Libraries

- **Library:** A set of classes and methods (APIs) that provide reusable functionality



# Why is API design important?

- If you program, you are an API designer, and APIs can be among your greatest assets **API是程序员最重要的资产和“荣耀”，吸引外部用户，提高声誉**
  - Good code is modular - each module has an API
  - Users invest heavily: acquiring, writing, learning
  - Thinking in terms of APIs improves code quality
  - Successful public APIs capture users
- Can also be among your greatest liabilities
  - Bad API can cause unending stream of support calls
  - Can inhibit ability to move forward
- Public APIs are forever – one chance to get it right
  - Once module has users, can't change API at will

建议：始终以开发API的标准面对任何开发任务

面向“复用”编程  
而不是面向“应用”编程

难度：要有足够良好的设计，一旦发布就无法再自由改变

# Whitebox and Blackbox frameworks



## ■ Whitebox frameworks

- Extension via **subclassing** and **overriding** methods
- Common design pattern(s): **Template Method**
- Subclass has main method but gives control to framework

## ■ Blackbox frameworks

- Extension via implementing a **plugin interface**
- Common design pattern(s): **Strategy, Observer**
- Plugin-loading mechanism loads plugins and gives control to the framework

# A calculator example (without a framework)



```
public class Calc extends JFrame {  
    private JTextField textField;  
    public Calc() {  
        JPanel contentPane = new JPanel(new BorderLayout());  
        contentPane.setBorder(new BevelBorder(BevelBorder.LOWERED));  
        JButton button = new JButton();  
        button.setText("calculate");  
        contentPane.add(button, BorderLayout.EAST);  
        textField = new JTextField("");  
        textField.setText("10 / 2 + 6");  
        textField.setPreferredSize(new Dimension(200, 20));  
        contentPane.add(textField, BorderLayout.WEST);  
        button.addActionListener(/* calculation code */);  
        this.setContentPane(contentPane);  
        this.pack();  
        this.setLocation(100, 100);  
        this.setTitle("My Great Calculator");  
        ...  
    }  
}
```

# A simple whitebox framework

```
public abstract class Application extends JFrame {  
    protected String getApplicationTitle() { return ""; }  
    protected String getButtonText() { return ""; }  
    protected String getInitialText() { return ""; }  
    protected void buttonClicked() {}  
    private JTextField textField;  
    public Application() {  
        JPanel contentPane = new JPanel(new BorderLayout());  
        contentPane.setBorder(new BevelBorder(BevelBorder.LOWERED));  
        JButton button = new JButton();  
        button.setText(getButtonText());  
        contentPane.add(button, BorderLayout.EAST);  
        textField = new JTextField("");  
        textField.setText(getInitialText());  
        textField.setPreferredSize(new Dimension(200, 20));  
        contentPane.add(textField, BorderLayout.WEST);  
        button.addActionListener((e) -> { buttonClicked(); });  
        this.setContentPane(contentPane);  
        this.pack();  
        this.setLocation(100, 100);  
        this.setTitlegetApplicationTitle();  
        ...  
    }  
}
```

子类中由开发者通过 override 完成定制的功能

# Using the whitebox framework

```
public class Calculator extends Application {  
    protected String getApplicationTitle() { return "My Great Calculator"; }  
    protected String getButtonText() { return "calculate"; }  
    protected String getInitialText() { return "(10 - 3) * 6"; }  
    protected void buttonClicked() {  
        JOptionPane.showMessageDialog(this, "The result of " + getInput() +  
            " is " + calculate(getInput()));  
    }  
    private String calculate(String text) { ... }  
}
```

Overriding

Extension via subclassing and overriding methods  
Subclass has main method but gives control to framework

```
public class Ping extends Application {  
    protected String getApplicationTitle() { return "Ping"; }  
    protected String getButtonText() { return "ping"; }  
    protected String getInitialText() { return "127.0.0.1"; }  
    protected void buttonClicked() { ... }  
}
```

Overriding

# An example blackbox framework

```
public class Application extends JFrame {  
    private JTextField textField;  
    private Plugin plugin;  
    public Application() { }  
    protected void init(Plugin p) {  
        p.setApplication(this);  
        this.plugin = p;  
        JPanel contentPane = new JPanel(new BorderLayout());  
        contentPane.setBorder(new BevelBorder(BevelBorder.LOWERED));  
        JButton button = new JButton();  
        button.setText(plugin != null ? plugin.getButtonText() : "ok");  
        contentPane.add(button, BorderLayout.EAST);  
        textField = new JTextField("");  
        if (plugin != null)  
            textField.setText(plugin.getInitialText());  
        textField.setPreferredSize(new Dimension(200, 20));  
        contentPane.add(textField, BorderLayout.WEST);  
        if (plugin != null)  
            button.addActionListener((e) -> { plugin.buttonClicked(); } );  
        this.setContentPane(contentPane);  
        ...  
    }  
    public String getInput() { return textField.getText(); }  
}
```

```
public interface Plugin {  
    String getApplicationTitle();  
    String getButtonText();  
    String getInitialText();  
    void buttonClicked();  
    void setApplication(Application app);  
}
```

# Using the blackbox framework

```
public class CalcPlugin implements Plugin {  
    private Application app;  
    public void setApplication(Application app) { this.app = app; }  
    public String getButtonText() { return "calculate"; }  
    public String getInitialText() { return "10 / 2 + 6"; }  
    public void buttonClicked() {  
        JOptionPane.showMessageDialog(null, "The result of "  
            + application.getInput() + " is "  
            + calculate(application.getInput()));  
    }  
    public String getApplicationTitle() { return "My Great Calculator"; }  
}
```

Extension via implementing a plugin interface  
Plugin-loading mechanism loads plugins and gives  
control to the framework

# Another example of white-box framework

```
public abstract class PrintOnScreen {  
  
    public void print() {  
        JFrame frame = new JFrame();  
        JOptionPane.showMessageDialog(frame, textToShow());  
        frame.dispose();  
    }  
    protected abstract String textToShow();  
}
```

```
public class MyApplication extends PrintOnScreen {
```

```
    @Override  
    protected String textToShow() {  
        return "printing this text on "  
            + "screen using PrintOnScreen "  
            + "white Box Framework";  
    }  
}
```

Main body of  
framework  
REUSE

Extension  
point

Overriding

```
MyApplication m = new MyApplication();  
m.print();
```

# Another example of black-box framework

```
public final class PrintOnScreen {  
  
    TextToShow textToShow;  
    public PrintOnScreen(TextToShow tx) {  
        this.textToShow = tx;  
    }  
  
    public void print() {  
        JFrame frame = new JFrame();  
        JOptionPane.showMessageDialog(frame,  
            textToShow.text());  
        frame.dispose();  
    }  
}  
public interface TextToShow {  
    String text();  
}
```

```
public class MyTextToShow  
    implements TextToShow {  
  
    @Override  
    public String text() {  
        return "Printing";  
    }  
}
```

PrintOnScreen m =  
new PrintOnScreen(new MyTextToShow());  
m.print();

Extension  
point by  
composition

Plugin

# Another example of black-box framework

```
public final class PrintOnScreen {
    TextToShow textToShow;
    public PrintOnScreen(TextToShow tx) {
        this.textToShow = tx;
    }
    public void print() {
        JFrame frame = new JFrame();
        JOptionPane.showMessageDialog(frame,
            textToShow.text());
        frame.dispose();
    }
}
public interface TextToShow {
    String text();
}
```

```
public class MyTextToShow
    implements TextToShow {
    @Override
    public String text() {
        return "Printing";
    }
}
```

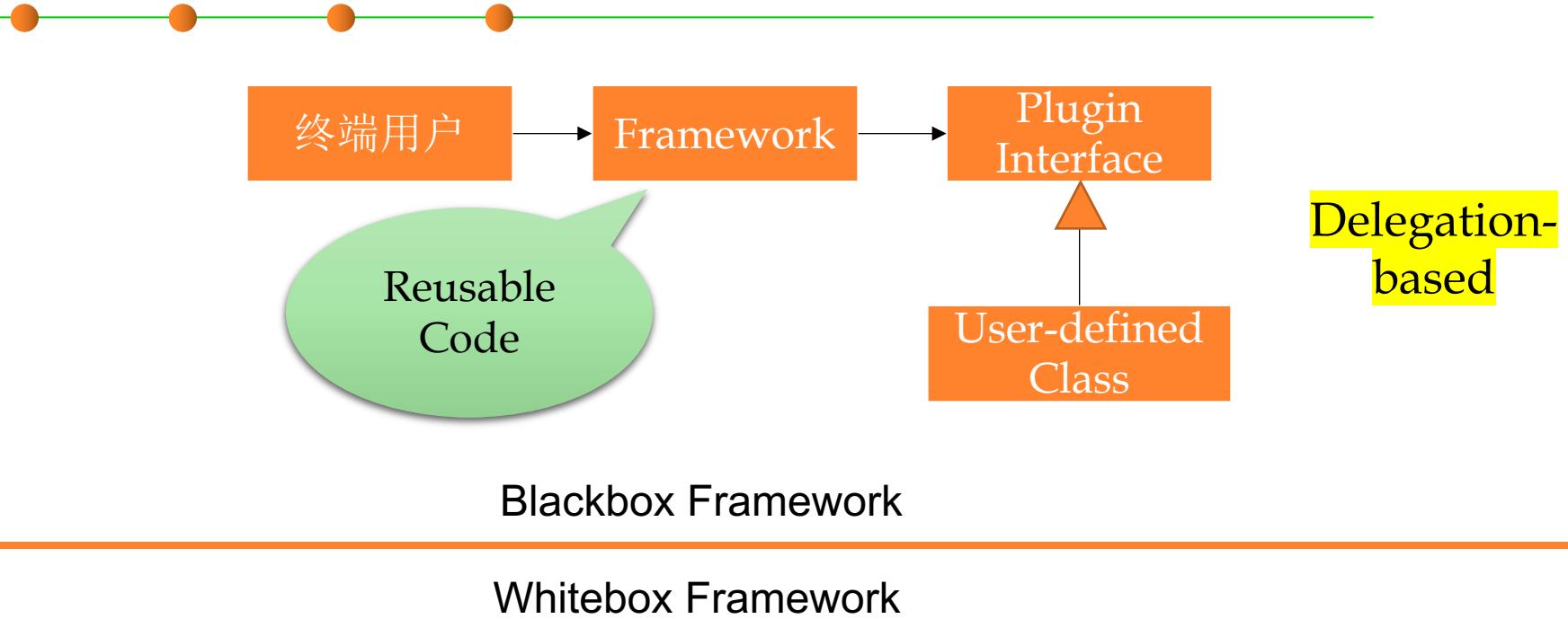
如果有多个plugin怎么办?  
如果通过配置文件选择加载某个plugin, 怎么做?

`PrintOnScreen m =  
 new PrintOnScreen(new MyTextToShow());  
m.print();`

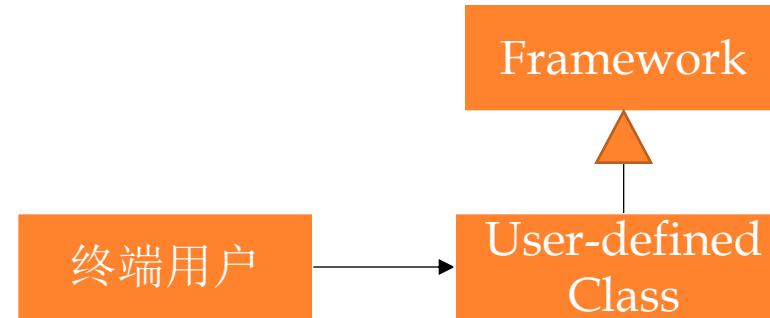
# Whitebox vs. Blackbox Frameworks

- ● ●
- **Whitebox frameworks use **subclassing/subtyping** ---继承**
  - Allows extension of every non-private method
  - Need to understand implementation of superclass
  - Only one extension at a time
  - Compiled together
  - Often so-called developer frameworks
- **Blackbox frameworks use **composition** -- 委派/组合**
  - Allows extension of functionality exposed in interface
  - Only need to understand the interface
  - Multiple plugins
  - Often provides more modularity
  - Separate deployment possible (.jar, .dll, ...)
  - Often so-called end-user frameworks, platforms

# Whitebox vs. Blackbox Frameworks



Inheritance-  
based



Reusable  
Code



# Summary

# Summary

- 
- **What is software reuse?**
  - **How to measure “reusability”?**
  - **Levels and morphology of reusable components**
    - Source code level reuse
    - Module-level reuse: class/interface
    - Library-level: API/package
    - System-level reuse: framework

# Summary

- 
- **Designing reusable classes**
    - Inheritance and overriding
    - Overloading
    - Parametric polymorphism and generic programming
    - Behavioral subtyping and Liskov Substitution Principle (LSP)
    - Composition and delegation
  - **Designing system-level reusable libraries and frameworks**
    - API and Library
    - Framework



The end

May 27, 2021