Tour of Java

科协智能体部 计06 徐晨曦

PREFACE

Introduction to Java Platform

What is Java

- Java是一门*高层次，强类型，编译型，基于类的面向对象*的*通用*编程语言
- Java保证*一次编写，到处运行 (write once, run anywhere)*，即Java编译出的字节码文件可以保证在任何系统上的Java运行时环境运行
- Java保证向前兼容性
- Java是一门*类C语言*
- 从1995年开始，历史悠久的Java平台积累了兼具广度和可靠性的生态系统和开发者社区
- Java至今仍然是最流行、最知名、使用最广的编程语言之一

History of Java

- 1991, *James Gosling*, C++ and Oak
- 1995, *Sun Microsystems* and Java 1.0
- 1996, Dynamic Web & Java Applets
- 2006, Java EE, Java SE, Java ME
- 2007, Java visual machine is opened under GPL
- 2009, Sunset of the *Sun*
- 2014, Java 8 reached General Availability
- 2018, Java 10 started to adopt Time-Based Release Versioning
- Today, Java 17 is current LTS, Java 18 GA, Java 19 RDP 2

Why Java

- Easy!
- Cross-Platform
- Rich and Reliable Ecosystem
- Forward evolution with backwards compatibility
- Performance

Why NOT Java

- Performance ?
- 相对贫弱的语法特性削弱了对程序员的吸引力
- JavaScript, Python等语言的流行和其生态系统的逐步完善
- Go, Rust等新星语言的涌现
- Kotlin, Scala, Clojure, Groovy等JVM平台语言提供了利用Java生态系统的其他选项
  - Kotlin获得了Google的青睐和Android平台的优先支持
  - Scala极为灵活的语法和强大的表达能力长期以来收到科学计算领域和分布式数据处理领域的欢迎

Before we start,

Let's recognize some terms of Java

- JDK, Java Develop Kit, Java开发工具包
- JRE, Java Runtime Environment, Java运行时环境
- JVM, Java Virtual Machine, Java虚拟机
- OpenJDK, 开源Java平台实现的合作组织
- LTS， Long Term Support, 长期支持版本
- IDE, Integrated Development Environment, 集成开发环境

Chapter 0

Installation of JDK & IntelliJ IDEA

Eclipse Adoptium

- **TLDR**: Eclipse Adoptium是一个提供一大堆平台和一大堆不同版本的JDK的网站
- **太长不看版**
  - OpenJDK顾名思义，是开源的，因此很多公司和组织都提供了自己的构建版本
  - 2017年启动的AdoptOpenJDK项目是第一个提供一大堆平台和一大堆不同版本的JDK的网站，同时也提供了多种平台的构建和测试平台
  - Eclipse Adoptium是AdoptOpenJDK的继任者
  - 它们提供的OpenJDK分发版称为Eclipse Temurin

Java Version

- **TLDR**：作为最新的*长期支持版本LTS*，**Java 17**将成为本次课程的目标平台。Java平台保证前向兼容，因此更新版本的Java支持本课程讲得绝大多数内容。
- **太长不看版**
  - Java的语法特性和版本绑定，大版本的后续更新大多都是bug修复和性能增强。但是Java7-9的发布时间分别是2011-2014-2017，以多年为跨度的更新使Java语法特性的进化严重落后于时代。
  - 因此，从 *[JEP 322, Java10]* 开始，采用新的根据固定时间的新版本发布计划和命名方案。Java将会*每六个月*固定发布新版本，用于快速滚动开发、预览、发布新语法特性和修改。

Install JDK *(Eclipse Temurin)*

- 所有系统通用：进网站，点下载，双击安装包
- macOS & Homebrew

```
$ brew install --cask temurin      // Install latest JDK18
$ brew tap homebrew/cask-versions
$ brew install --cask temurin17
```

- Linux

```
$ apt install temurin-17-jdk     // Debian/Ubuntu
$ yum install temurin-17-jdk     // CentOS/RHEL/Fedora
$ zypper install temurin-17-jdk  // openSUSE
```

Install JDK *(Eclipse Temurin)*

- 如果Temurin网站或者Linux源下载速度较慢，可以使用TUNA镜像
- Homebrew如果下载速度慢，需要为其设置代理

Other OpenJDK Distribution ?

- Azul Zulu
- BellSoft Liberica JDK
  - 上面两个在业界均有一定应用率，可以信赖
  - 均提供JDK8在ARM Mac的构建版本
  - 均提供JavaFX的支持
- GraalVM
  - Oracle开发的下一代JDK和全新的多语言虚拟机
  - 将Java编译为本地代码以消除大部分根本上的性能问题
  - 提供高性能的Python, R, Ruby等语言的运行时

First Command

```
$ java --version
openjdk 17.0.3 2022-04-19
OpenJDK Runtime Environment Temurin-17.0.3+7 (build 17.0.3+7)
OpenJDK 64-Bit Server VM Temurin-17.0.3+7 (build 17.0.3+7, mixed mode)
```

CodeLab 0-1: helloworld

Wait... What is *CodeLab*?

- 编程小练习，仅此而已
- 我会带大家在课上都写一遍
- GitHub仓库里提供了未完成供大家联系，也同时提供了完成的版本供大家参考
- `tourofjava/chN/labM`

CodeLab 0-1: helloworld

`Helloworld.java`

```
package tourofjava.ch0.lab1;

public class Helloworld {
    public static void main(String[] args) {
        System.out.println("helloworld");
    }
}
```

```
tour-of-java/codelab $ javac tourofjava/ch0/lab1/Helloworld.java
tour-of-java/codelab $ java tourofjava.ch0.lab1.Helloworld
helloworld
```

CodeLab 0-1 helloworld

[JEP330, Java11] Launch Single-File Source-Code Programs

```
tour-of-java/codelab $ java tourofjava/ch0/lab1/Helloworld.java
helloworld
```

CodeLab 0-1 helloworld

[JEP222, Java9] `jshell`: The Java Shell (Read-Eval-Print Loop)

```
$ jshell
jshell
|  欢迎使用 JShell -- 版本 17.0.3
|  要大致了解该版本, 请键入: /help intro

jshell> System.out.println("helloworld")
helloworld
```

JetBrains IntelliJ IDEA

- https://www.jetbrains.com/idea/
- 目前最强大、使用最广泛的Java IDE
- IntelliJ IDEA Community Edition是免费开源版本，其包含了基础的Java语言支持、分析、构建与版本管理和Docker支持
- IntelliJ IDEA Ultimate是收费版本，支持性能剖析，企业级框架支持，Web技术支持，数据库工具，远程协作开发支持
  - 使用`@mails.thu.edu.cn`可以申请JetBrains学生包以获取全部开发工具

Chapter 1

Basic language structure

Java is a *C-style* language

Comments

```
// This is a one-line comment

/*
 * This is a multi-line comment
 */

/**
 * This is JavaDoc
 */
```

Java is a *C-style* language

Variable Declare & Assignment / Data types

```
jshell> int a;
a ==> 0
jshell> a = 1;
a ==> 1
jshell> long b = 1L
b ==> 1
jshell> char e = 'E'
e ==> 'E'
jshell> String f = "java"
f ==> "java"
jshell> boolean g = true
g ==> true
```

Java is a *C-style* language

Variable Declare & Assignment / Data types

```
jshell> float c = 1.0
|  错误:
|  不兼容的类型: 从double转换到float可能会有损失
|  float c = 1.0;
|            ^-^
jshell> float c = 1.0f
c ==> 1.0
jshell> c = (float) 2.0
c ==> 2.0
jshell> double d = 1.0
d ==> 1.0
```

Java is a *C-style* language

Operators: Arithmetic

```
jshell> 1 + 1
$2 ==> 2
jshell> 2 * 2
$3 ==> 4
jshell> 3 / 2
$4 ==> 1
jshell> 3.0 / 2
$5 ==> 1.5
jshell> 3 % 2
$6 ==> 1
jshell> -3 % 2
$6 ==> -1
```

Java is a *C-style* language

Operators: `++` `--`

```
jshell> int a = 1;
a ==> 1

jshell> ++a
$29 ==> 2
jshell> --a
$30 ==> 1

jshell> a++
$31 ==> 2
jshell> a--
$32 ==> 1
```

Java is a *C-style* language

Operators: Bitwise

```
jshell> 1 & 2
$11 ==> 0

jshell> 1 | 2
$12 ==> 3

jshell> 1 ^ 2
$13 ==> 3

jshell> ~1
$14 ==> -2
```

Java is a *C-style* language

Operators: Logic

```
jshell> true && false
$15 ==> false

jshell> true || false
$16 ==> true

jshell> !true
$17 ==> false
```

Java is a *C-style* language

Operators: String

```
jshell> "hello" + "world"
$10 ==> "helloworld"
```

Java is a *C-style* language

Function / Method Syntax

```
jshell> int mul(int a, int b) {
   ...>     return a * b;
   ...> }
|  已创建 方法 mul(int,int)

jshell> mul(2, 3)
$20 ==> 6
```

Java is a *C-style* language

Control Flow: `if-else`

```
jshell> boolean canIGetA(double score) {
   ...>     if (score >= 90) {
   ...>         System.out.println("Good Job! You got a 4.0!");
   ...>         return true;
   ...>     } else if (score >= 70) {
   ...>         System.out.println("Keep Going! You can do better!");
   ...>         return false;
   ...>     } else {
   ...>         System.out.println("Emm... What happends?");
   ...>         return false;
   ...>     }
   ...> }
|  已创建 方法 canIGetA(double)
```

Java is a *C-style* language

Control Flow: `if-else`

```
jshell> canIGetA(100)
Good Job! You got a 4.0!
$2 ==> true

jshell> canIGetA(80)
Keep Going! You can do better!
$3 ==> false

jshell> canIGetA(0)
Emm... What happends?
$4 ==> false
```

Java is a *C-style* language

Control Flow: `?:`

```
jshell> int gcd(int a, int b) {
   ...>     return b == 0 ? a : gcd(b, a % b);
   ...> }
|  已创建 方法 gcd(int,int)

jshell> gcd(48, 18)
$26 ==> 6
```

Java is a *C-style* language

Control Flow: `switch`

```
jshell> int calculate(int a, int b, char op) {
   ...>     int res;
   ...>     switch(op) {
   ...>         case '+': res = a + b; break;
   ...>         case '-': res = a - b; break;
   ...>         case '*': res = a * b; break;
   ...>         case '/': res = a / b; break;
   ...>         default: res = -1;
   ...>     }
   ...>     return res;
   ...> }
|  已创建 方法 calculate(int,int,char)
```

Java is a *C-style* language

Control Flow: `switch`

```
jshell> calculate(2, 3, '+')
$6 ==> 5
jshell> calculate(2, 3, '-')
$7 ==> -1
jshell> calculate(2, 3, '*')
$8 ==> 6
jshell> calculate(2, 3, '/')
$9 ==> 0
jshell> calculate(2, 3, '^')
$10 ==> -1
```

Java is a *C-style* language

Control Flow: `for`

```
jshell> boolean isPrime(int a) {
   ...>     if (a <= 1) return false;
   ...>     for (int i = 2; i * i <= a; i++) {
   ...>         if(a % i == 0) return false;
   ...>     }
   ...>     return true;
   ...> }
|  已创建 方法 isPrime(int)
```

Java is a *C-style* language

Control Flow: `for`

```
jshell> for (int a = 1; a <= 100; a++) {
   ...>     if (isPrime(a)) System.out.print(a + " ");
   ...> }
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
```

Java is a *C-style* language

Control Flow: `while`

```
jshell> import java.lang.Math;

jshell> double x;
x ==> 0.0

jshell> while( (x = Math.random()) < 0.8) {
   ...>     System.out.println(x + " < 0.8");
   ...> }
0.17636000596270907 < 0.8
0.5029857602823097 < 0.8
0.3349284868587292 < 0.8
0.3650514335114553 < 0.8
```

Java is a *C-style* language

Control Flow: `do-while`

```
jshell> do {
   ...>     x = Math.random();
   ...> } while(x < 0.8);

jshell> x
x ==> 0.8346328176085303
```

Other basic syntax & API

[JEP286, Java10] `var` type inference

```
jshell> var mapOfMap = new HashMap<String, Map<String, String>>()
mapOfMap ==> {}
jshell> mapOfMap.put("Java", Map.of("Type", "Strong-Typed", "Paradigm", "OOP"))
$111 ==> null
jshell> mapOfMap
mapOfMap ==> {Java={Paradigm=OOP, Type=Strong-Typed}}

jshell> mapOfMap.forEach((var key, var val) -> System.out.println(key + " : " + val))
Java : {Paradigm=OOP, Type=Strong-Typed}
```

Other basic syntax & API

[JEP378, Java15] Text Blocks

```
jshell> """
   ...> Line 1
   ...> Line 2
   ...> Line 3
   ...> Line 4
   ...> """
$4 ==> "Line 1\nLine 2\nLine 3\nLine 4\n"
```

Other basic syntax & API

Array

```
jshell> int[] intArr = {2, 0, 1, 9, 0, 1, 0, 8, 9, 5};
intArr ==> int[10] { 2, 0, 1, 9, 0, 1, 0, 8, 9, 5 }
jshell> intArr[1]
$52 ==> 0
jshell> intArr[20]
|  异常错误 java.lang.ArrayIndexOutOfBoundsException: Index 20 out of bounds for length 10
|        at (#53:1)
jshell> for (int i : intArr) System.out.print(i)
2019010895
jshell> intArr.length
$53 ==> 10

jshell> int[] intArr2 = new int[10];
intArr2 ==> int[10] { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 }
```

Other basic syntax & API

Array

```
jshell> var matrix = new int[5][5];
matrix ==> int[5][] { int[5] { 0, 0, 0, 0, 0 }, int[5] { 0,  ... int[5] { 0, 0, 0, 0, 0 } }
jshell> for (int i = 0; i<5; i++) {
   ...>     for(int j = 0; j<5; j++) {
   ...>         matrix[i][j] = i * j;
   ...>     }
   ...> }
jshell> matrix
matrix ==> int[5][] {
    int[5] { 0, 0, 0, 0, 0 },
    int[5] { 0, 1, 2, 3, 4 },
    int[5] { 0, 2, 4, 6, 8 },
    int[5] { 0, 3, 6, 9, 12 },
    int[5] { 0, 4, 8, 12, 16 } }
```

Other basic syntax & API

Array Utility

```
jshell> import java.util.Arrays;
jshell> Arrays.sort(intArr)
jshell> intArr
intArr ==> int[10] { 0, 0, 0, 1, 1, 2, 5, 8, 9, 9 }
jshell> Arrays.binarySearch(intArr, 5)
$64 ==> 6

jshell> System.out.println(intArr)
[I@2d6eabae
jshell> System.out.println(Arrays.toString(intArr))
[0, 0, 0, 1, 1, 2, 5, 8, 9, 9]
jshell> System.out.println(Arrays.deepToString(matrix))
[[0, 0, 0, 0, 0], [0, 1, 2, 3, 4], [0, 2, 4, 6, 8], [0, 3, 6, 9, 12], [0, 4, 8, 12, 16]]
```

Other basic syntax & API

String

```
jshell> String s = "I Love Java"
s ==> "I Love Java"
jshell> s.length()
$37 ==> 11
jshell> s.split(" ")
$38 ==> String[3] { "I", "Love", "Java" }
jshell> s.concat(" and Rust")
$39 ==> "I Love Java and Rust"
jshell> s.contains("C++")
$40 ==> false
jshell> s.charAt(2)
$41 ==> 'L'
```

Other basic syntax & API

String

```
jshell> s.replace("Java", "C++")
$42 ==> "I Love C++"
jshell> s.toUpperCase()
$46 ==> "I LOVE JAVA"
jshell> s.toLowerCase()
$47 ==> "i love java"
jshell> s.startsWith("I")
$48 ==> true
jshell> s.endsWith("Java")
$49 ==> true
jshell> s
s ==> "I Love Java"
```

Other basic syntax & API

String

```
jshell> s == "I Love Java"
$43 ==> true
jshell> s == new String("I Love Java")
$44 ==> false
jshell> s.equals(new String("I Love Java"))
$45 ==> true
jshell> s.equalsIgnoreCase("i lOve jAVA")
$50 ==> true
```

Other basic syntax & API

Console IO

```
jshell> System.out.println("I Love Java")
I Love Java

jshell> System.out.print("I Love Java")
I Love Java
jshell> System.out.printf("I Love %s\n", "Java")
I Love Java
$82 ==> java.io.PrintStream@4d95d2a2
```

Other basic syntax & API

Console IO

```
jshell> import java.util.Scanner;
jshell> var input = new Scanner(System.in);
jshell> input.hasNextInt()
100
$86 ==> true
jshell> input.nextInt()
$87 ==> 100
jshell> input.hasNextBoolean()
100
$89 ==> false
```

Other basic syntax & API

BigInteger **BigDecimal**

```
jshell> import java.math.BigInteger;

jshell> 1000000000000000 + 1
|  错误:
|  整数太大
|  1000000000000000 + 1
|  ^
jshell> new BigInteger("1000000000000000").add(BigInteger.ONE)
$92 ==> 1000000000000001

jshell> new BigInteger("19260817").isProbablePrime(10)
$93 ==> true
```

Other basic syntax & API

BigInteger **BigDecimal**

```
jshell> import java.math.BigInteger;

jshell> 0.1 + 0.2
$94 ==> 0.30000000000000004

jshell> new BigDecimal("0.1").add(new BigDecimal("0.2"))
$95 ==> 0.3

jshell> new BigDecimal("0.000000001").toEngineeringString()
$100 ==> "1E-9"
```

Other basic syntax & API

Math

```
jshell> Math.
E                IEEEremainder(   PI               abs(             absExact(
acos(            addExact(        asin(            atan(            atan2(
cbrt(            ceil(            class            copySign(        cos(
cosh(            decrementExact(  exp(             expm1(           floor(
floorDiv(        floorMod(        fma(             getExponent(     hypot(
incrementExact(  log(             log10(           log1p(           max(
min(             multiplyExact(   multiplyFull(    multiplyHigh(    negateExact(
nextAfter(       nextDown(        nextUp(          pow(             random()
rint(            round(           scalb(           signum(          sin(
sinh(            sqrt(            subtractExact(   tan(             tanh(
toDegrees(       toIntExact(      toRadians(       ulp(
```

Other basic syntax & API

Math

```
jshell> Math.sin(Math.PI / 4)
$103 ==> 0.7071067811865475
jshell> Math.floor(1.002)
$104 ==> 1.0
jshell> Math.log(Math.E)
$107 ==> 1.0
jshell> Math.pow(2, 1.5)
$108 ==> 2.82842712474619
jshell> Math.sqrt(2)
$109 ==> 1.4142135623730951
```

CodeLab 1-1 a +-*/^&| b

CodeLab 1-2 QuickSort

Chapter 2

Fundamental OOP

What is OOP

- OOP是面向对象编程 Object-Oriented Programming 的缩写
- OOP是围绕*对象 Object*进行组织的一种编程*范式*
  - 对象本身有利于程序员对一些概念进行自然的建模
  - 使用对象封装状态、数据和行为有利于组织大规模、复杂的软件系统，也有利于团队协作开发
- 经过多年的发展与经验的积累，已经有公认的一套*原则*和*设计模式*指导程序员正确有效的使用OOP解决现实问题

Terms used in OOP

- 为了避免混淆，仅在这里列出相应术语的中英文对应，后面提到时只使用斜体英文
- *OOP* 面向对象编程
- *object* 对象：数据和行为的集合
- *class* 类：定义对象的模版

Object

- *object*是数据和行为的集合
- 数据一般称为*member variable*, *field*, *state*, *data*等
- 行为一般称为*method*, *function*, *interface*, *protocol*等
- 面向对象编程原语是向*对象发送消息*，而不是面向过程编程中的*指令跳转*
- *发送消息*就应当遵循一定的协议*protocol*，比如发送消息的类型和消息的内容，这表现为方法的名字和参数

Class

- *class*是*object*的模版，是*object*的*类型*
- *class*定义了*object*所拥有的数据和行为，而*object*包含了具体的数据值和对应行为的真正执行者
- *class*本身可以拥有一些数据和行为
- 面向对象编程的基本流程为：
  1. 设计类 `class Car { /* ... */ }`
  2. 创建/实例化对象 `var myCar = new Car()`
  3. 向对象发送消息 `myCar.move()`

OOP in Java

- *基本上* 万物皆对象
  - 基本数据类型除外
- Java使用*引用*来控制对象
  - Java只有按值传递
- Java拥有自动内存管理机制
  - 你只需要关系如何 `new`，而无需担心对象是何时或如何被 `delete`
  - Java拥有多种优秀的垃圾收集器GC实现

CodeLab 2-1 Build a `Car` in Java

`package`

```
package tourofjava.ch2.lab1;
```

- Java使用*包 package*组织代码。
- 包名使用`.`建立层次结构，应当与文件目录结构相同。
- 包虽然有嵌套结构，但是父目录和子目录之间没有如包含等的任何关系，仅仅是名字不同
- 包名通常全小写且单词之间无分隔

CodeLab 2-1 Build a `Car` in Java

Declare a `class`

```
public class Car { ... }
```

- `public` 是*访问控制关键字*的一种
  - `public` 代表其他所有类都可以访问，`private` 代表只有这个类本身可以访问
  - 应用于*class field method*的声明中
  - Java要求名为 `<name>.java` 的源文件中必须包含同名的 `public class`

CodeLab 2-1 Build a `Car` in Java

Declare a `class`

```
public class Car { ... }
```

- 类名通常采用大驼峰

CodeLab 2-1 Build a `Car` in Java

Declare member fields

```
public class Car {
    private String brand;
    private String model;
    private String color;
    private int enginePower;
    private int currentSpeed;
    private int maxSpeed;
    private int price;

    // ...
}
```

CodeLab 2-1 Build a `Car` in Java

Declare member fields

```
private String brand;
```

- 访问控制 + 类型 + 成员域名 （通常采用小驼峰）
- *类成员变量的*默认值
  - 数字类型： `0` `0.0`
  - `boolean`： `false`
  - 对象： `null` 空指针
- 可以定义时赋初始值

CodeLab 2-1 Build a `Car` in Java

构造器 constructor

```
public class Car {
    public Car(String brand, String model, String color,
               int enginePower, int maxSpeed, int price) {
        this.brand = brand;
        this.model = model;
        this.color = color;
        this.enginePower = enginePower;
        this.maxSpeed = maxSpeed;
        this.price = price;
    }
}
```

CodeLab 2-1 Build a `Car` in Java

构造器 constructor

```
public Car(...)
```

- 访问控制 + 类名 + (参数列表)
- 用于初始化对象数据

CodeLab 2-1 Build a `Car` in Java

构造器 constructor

```
this.brand = brand;
```

- 类的成员方法可以访问类的域和方法而不需要 `this` 指定
- 但是如果出现了变量或者形参名字冲突，则需要 `this.` 手动指定

CodeLab 2-1 Build a `Car` in Java

Member methods

```
public class Car {
    public double move(double timeInSec) {
        double distance = currentSpeed / 3.6 * timeInSec;
        System.out.println("Move " + distance + " m in " + timeInSec + " s");
        return distance;
    }
    public void honk() {
        System.out.println("DiDi...");
    }
    public void introduce() {
        System.out.println("I'm a/an " + color + " " + brand + " " + model);
    }
}
```

CodeLab 2-1 Build a `Car` in Java

Member methods

- 成员方法的命名习惯一般为*小驼峰*
- 成员方法就是定义在类内部的方法/函数
- 并且也可以使用类内的成员变量和成员方法
- 类的`public`方法组成了类所定义的对象的公开接口/功能

CodeLab 2-1 Build a `Car` in Java

Getters & Setters

```
public class Car {
    private String brand;

    public String getBrand() {
        return brand;
    }
    public void setBrand(String brand) {
        this.brand = brand;
    }
}
```

CodeLab 2-1 Build a `Car` in Java

Getters & Setters

- 在大多数情况下，类的成员变量应当被封装为`private`，对他们的访问和修改应该由相应的方法处理
- 对于`boolean`型域，一般使用`isXXX` `setXXX`
- 对于其他类型域，一般使用`getXXX` `setXXX`
- 对于简单的读写操作，这些代码确实是没什么用的模版代码，但是
  - IDE提供代码自动生成功能
  - Lombok等库提供了编译时生成方法机制
  - Record

CodeLab 2-1 Build a `Car` in Java

Using class & object

```
public class Car {
    public static void main(String[] args) {
        var myCar = new Car("BMW", "X7", "Black", 250, 245, 1_000_000);
        myCar.introduce();
        myCar.honk();
        myCar.setSpeed(100);
        myCar.move(600);
    }
}
```

CodeLab 2-1 Build a `Car` in Java

Using class & object

```
var myCar = new Car("BMW", "X7", "Black", 250, 245, 1_000_000);
```

- 使用`new`创建新的对象，这个对象的生命周期、内存位置将由Java自动管理
- `new`会调用与提供参数类型对应的构造器
  - 如果类不提供自定义构造器，那么默认提供无参数构造器

CodeLab 2-1 Build a `Car` in Java

Using class & object

```
myCar.introduce();
```

- 使用`.`调用对象的方法/接口

Reuse classes

所有人都喜欢代码重用。
在面向对象的语义中，类定义了几乎所有的行为和数据，那么代码重用指的*重用类*，即*用已有的类来生成新的类*的过程。

下面就来介绍面向对象中最基本的*重用类*方式：

- 组合 Composition
- 继承 Inheritance

Reuse classes

Composition

- 组合指的是在新类中包含已有类的对象 *（的引用）*。
  - 你可能会说：就这？那确实，就这。
  - 组合是最简单、但也是最基本的重用类的方式
- 在我们的 `Car` 类中，我们实际上组合了若干个 `String` 对象来描述车的一些基本信息

```
public class Car {
    private String brand, model, color;
}
```

Reuse classes

Composition: 组合描述了 *"has-a"* 关系

- 我们可以这种语义来建模更复杂、更贴合实际的汽车，如：
  - 一辆汽车 *有一个* 引擎
  - 一辆汽车 *有四个* 轮子
- 那么*汽车*的最大速度，就可以通过重用*引擎*中的数据或方法来进行计算

Reuse classes

Composition: 组合对象的初始化

- 前面提到，对象*引用*的默认值是`null`，对其进行任何访问都会引发空指针异常`java.lang.NullPointerException`

```
class Demo {
   private String s1 = "String 1", s2, s3, s4; // 1. definition
   public Demo() { s2 = "String 2"; }          // 2. constructor
   { s3 = "String 3"; }                        // 3. instance init
   public void useS4() {
      if (s4 == null) s4 = "String 4";         // 4. delayed init
   }
}
```

Reuse classes

Inheritance

- *继承*是面向对象编程的重要特征之一
- 在创建类的时候，总是使用了继承。在Java中，如果不手动指定，将会隐式继承自 `java.lang.Object` 类，表达了任何对象的通用接口
- *继承*将会使*子类*拥有所有*父类*的公开接口及其实现，并且可以访问*父类*中受保护的域和方法
- 使用 `extends` 关键字声明继承

```
class Parent { ... }
class Child extends Parent { ... }
```

Reuse classes

Inheritance

- Java不允许多重继承。原因在于：
  - 规避*diamond problem*，即继承具有相同接口的不同的类导致子类的接口实现的歧义
  - 多重继承带来的设计负担与带来的架构优势不对等

Reuse classes

Inheritance: 组合描述了 *"is-a"* 关系

让我们考虑任何OOP课程都会讲的几何图形的例子：

```
public class Shape {
   public double area() { return 0; }
   public void draw() {}
   public boolean inside(double x, double y) { return false; }
}
public class Rectangle extends Shape { ... }
public class Circle extends Shape { ... }
public class Triangle extends Shape { ... }
```

Reuse classes

Inheritance: 组合描述了 *"is-a"* 关系

在这个例子中：

- 矩形 `Rectangle` 圆形 `Circle` 三角形 `Triangle` 都 *是一个* 几何图形 `Shape`
- `Rectangle` `Circle` `Triangle` 都包含了 `Shape` 提供的三个公开接口，即获取表面积 `area()` 绘制 `draw()` 判断点是否在图形内部 `inside(double, double)`

Reuse classes

Inheritance: Override/重载方法

子类会继承父类的接口，但是大多数情况下，子类需要提供自己的实现。这时就需要*override/重载*方法。比如`Shape`的`area()`和`inside(double, double)`都是无意义的。

```
public class Circle extends Shape {
   double x, y, r;
   public Circle(double x, double y, double r) { ... }
   public double area() { return Math.PI * r * r; }
   public boolean inside(double x1, double y1) {
      return (x1 - x) * (x1 - x) + (y1 - y) * (y1 - y) < r * r;
   }
}
```

Reuse classes

Inheritance: Override/重载方法

子类的重载方法实现中，可能需要重用父类方法的实现。Java中使用`super`关键字表示使用父类的接口与数据。`super`名字取自*超类superclass*。

```
public class Person { public String introduce() { ... } }

public class Student extends Person {
    public String introduce() {
        return super.introduce() + "24岁, 事学生";
    }
}
```

Reuse classes

Inheritance: `@Override`

重载父类方法需要函数名和参数类型与顺序都相同，如果误写那么这个方法就是子类全新的方法，这会导致可能错误的调用父类的方法。
如果明确知道自己想重载方法，可以使用 `@Override` *注解*来让编译器“提醒”你。

```
public class Circle extends Shape {
    @Override
    public double area() { return Math.PI * r * r; }
}
```

Reuse classes

Inheritance: `@Override`

如果函数名不小心输入错误，那么编译器会很不高兴：

```
public class Circle extends Shape {
    @Override
    public double arae() { return Math.PI * r * r; }
}
```

```
java: 方法不会覆盖或实现超类型的方法
```

Reuse classes

Inheritance: `@Override`

为了避免这种typo，除了使用`@Override`，我还建议使用IDE来为你自动生成函数签名。如在IntelliJ IDEA中，使用`control/ctrl + O`来自动重载方法。

毕竟Java语法还是比较繁琐的，我使用IDE就是为了延长我和我的键盘的寿命。

Reuse classes

Inheritance: 初始化父类

父类中自然也会存在一些数据，所以可能有自定义的初始化/构造器实现。因为父类的成员域可能对子类不公开，即`private`，为了正确的初始化这些域，我们需要正确的使用父类的构造器。

Reuse classes

Inheritance: 初始化父类

如果父类*有无参数构造器*，那么子类的所有构造器，如果没有显式指定，将会*首先调用*父类的无参构造器。

```
public class Shape {
    // ...
}

public class Circle extends Shape {
    double x, y, r;
    public Circle(double x, double y, double r) { ... }
}
```

Reuse classes

Inheritance: 初始化父类

如果父类*没有无参数构造器*，那么子类必须使用 `super` 关键字调用父类的构造器，而且这个语句*必须*在子类构造器的第一行。

Reuse classes

Inheritance: 初始化父类

```
public class Person {
    private String name;
    public Person(String name) { this.name = name; }
}
public class Student extends Person {
    private String university;
    public Student(String name, String university) {
        super(name);
        this.university = university;
    }
}
```

Reuse classes

Inheritance: 初始化父类

```
public class Student extends Person {
    private String university;
    public Student(String name, String university) {
        this.university = university;
        super(name);
    }
}
```

java: 对super的调用必须是构造器中的第一个语句

Reuse classes

Inheritance: 初始化父类

这个限制是为了确保子类可以正确使用父类的域和方法。进而，`super()` 的参数不可以传入、使用父类和子类的域或方法的返回值。

PS：不过确实有绕过这种限制的方法。但是试图访问未初始化的数据仍然是非常危险的，我在这里不介绍相关的方法，好奇的同学可以去查一查。

Reuse classes

Inheritance: 初始化父类

下面的代码是非法的：

```
class Parent {
    public String name;
    public Parent(String name) { this.name = name; }
}
class Child extends Parent {
    public Child() {
        super(super.name);
        // super(name);
    }
}
```

Reuse classes

Inheritance: Upcasting & Polymorphism

继承带来的重要性质是*向上转型 Upcasting*和*多态 Polymorphism*。

*Upcasting*指的是子类对象可以转型为父类。

```
Shape shape = new Circle(0, 0, 1);
```

*Polymorphism*指的是对父类函数的调用可能使用的子类的实现，会根据代码运行时，对象的实际类型进行动态分发。

```
shape.area(); // Invoking Circle#area()
```

Reuse classes

Inheritance: Polymorphism in Depth

- 数据抽象、继承与多态时面向对象程序设计语言的三大基本特征
- 多态的作用在于，消除实际类型之间的耦合关系，使得代码设计中尽可能的依赖于公开接口而非具体实现

Reuse classes

Inheritance: Polymorphism in Depth

多态可以提升扩展性。考虑下面更加实际的绘制图形的代码，图形的绘制函数接收一个画板对象，用于执行实际的绘图指令，如 `drawLine(int, int, int, int)` 等：

```
class Shape { public void draw(Canvas c) {} }
class Circle extends Shape { @Override public void draw(Canvas c) { ... } }
class Rectangle extends Shape { @Override public void draw(Canvas c) { ... } }
class Triangle extends Shape { @Override public void draw(Canvas c) { ... } }
```

Reuse classes

Inheritance: Polymorphism in Depth

如果没有多态，我们需要根据具体的类型来调用具体的绘图实现：

```
class GUI {
   private void render(Circle    shape) { shape.draw(getCanvas()); }
   private void render(Rectangle shape) { shape.draw(getCanvas()); }
   private void render(Triangle  shape) { shape.draw(getCanvas()); }
}
```

先不论这份代码做了多少重复工作，我们考虑加入一个新的图形子类 `Ellipse`，那么 `GUI` 类还需要再添加一行 `render` 函数。

Reuse classes

Inheritance: Polymorphism in Depth

如果没有多态，我们需要根据具体的类型来调用具体的绘图实现：

```
class GUI {
    private void render(Circle    shape) { shape.draw(getCanvas()); }
    private void render(Rectangle shape) { shape.draw(getCanvas()); }
    private void render(Triangle  shape) { shape.draw(getCanvas()); }
}
```

更有可能的是，`GUI`是别人提供的代码实现，我们没有修改他们的权力。

Reuse classes

Inheritance: Polymorphism in Depth

如果使用多态，那么直接接收一个 `Shape` 基类的对象，便可以实现运行时动态分发方法，并且也可以支持未来可能会添加的任意具体的图形子类。

```
class GUI {
   private void render(Shape shape) { shape.draw(getCanvas()); }
}
```

Reuse classes

Inheritance: Polymorphism in Depth

尽量不要在构造器中使用多态函数

否则基类构造器在执行时，可能会错误的调用了子类的多态函数，但是此时子类的数据还未被正确初始化，这可能会导致错误的结果甚至异常崩溃。

因此，尽可能的用简单的语句进行成员变量初始化。

Reuse classes

Inheritance: Polymorphism in Depth

Java中不是所有事物都支持多态：

1. `static`和`final`方法是*前期调用绑定*，即他们所调用的方法在编译时已经确定
2. 成员域不存在多态。如果存在重名，只会单纯的出现了标识符覆盖。对重名域的访问只和方法实现所在的类绑定。
3. `private`方法默认为`final`，因此即使重名也不会出现重载也不会支持多态。

Reuse classes

Inheritance: Downcasting

可以向上转型就可以向下转型。前提是需要判断一下是否真的是某个具体子类然后再尝试转型，否则会触发 `java.lang.ClassCastException` 异常。

判断某个对象是否是一个类的实例，可以使用 `instanceof` 关键字：

```
if (shape instanceof Circle) {
    Circle circle = (Circle) shape;
    // do something with circle
}
```

Reuse classes

Inheritance: Downcasting

一个小语法糖：*[JEP394, Java16]* Pattern Matching for `instanceof`

```
if (shape instanceof Circle circle) {
    // do something with circle
}
```

Reuse classes

Composition vs Inheritance

- 两种方法都可以重用现有类的代码
- 组合适用于我们需要*使用*现有类的功能与接口的情况，继承适用于我们想新建一个现有类的特例，我们可以重新实现和扩展现有类的接口。
- 继承可能带来的问题是*破坏封装*，即子类被允许暴露父类的受保护域和方法，这通常是不推荐的。
- **组合优先于继承**

Reuse classes

Delegate

考虑一辆可以载一些人的公共汽车，我们可以让公共汽车继承自 `HashSet` 来实现保存车上乘客信息的功能：

```
class Person { ... }
class Bus extends HashSet<Person> { ... }

var bus = new Bus();
bus.add(new Person("c7w"));
bus.add(new Person("lambda"));
bus.add(new Person("xsun2001"));
```

Reuse classes

Delegate

显然，这个设计并不好。一方面，公共汽车应当*是一辆*载具，*有一些*乘客，我们应当*组合*一个乘客的集合。另一方面，直接继承会给公共汽车一堆本来不需要的数据和实现。

但是，我们确实需要提供类似*继承*的功能，因为我们确实需要加入*add remove size clear*这种和集合很类似的接口。

很多其他的语言提供了*代理 delegate*或者*混入 mixin*等语法特性和概念。但是Java~~显然~~没有相关的语法支持。

Reuse classes

Delegate

```
class Bus {
    private Set<Person> passengers = new HashSet();
    public boolean add(Person p) { return passengers.add(p); }
    public boolean remove(Person p) { return passengers.remove(p); }
    public int size() { return passengers.size(); }
    public void clear() { return passengers.clear(); }
}
```

在Java中，可以通过组合+手动转发的方法实现delegate。

Reuse classes

Abstract class

某些类本身就是作为其他类的父类而存在。比如上面提到的`Shape`类，它提供的接口实际上都是没有意义的哑实现，必须要借助具体的子类才能起作用。在Java中可以使用*抽象类 Abstract Class*来表达。

```
public abstract class Shape {
    public abstract double area();
    public abstract void draw();
    public abstract boolean inside(double x, double y);
}
```

Reuse classes

Abstract class

抽象类可以包含*抽象方法 Abstract Methods*，也就是在签名中带有`abstract`关键字的方法。这些方法没有实现，类似于C++中的*纯虚函数*。

抽象类不能被直接实例化，因为它们不提供具体接口的实现，还因为这在概念上是没有意义的：

```
var shape = new Shape(); // Error
```

Base of all: `java.lang.Object`

`java.lang.Object`是所有类的超类，它代表的语义也非常直观：所有对象都*是一个*对象，确切的来说，是Java语言内的对象`java.lang.Object`

根据继承的规则，任何类都包含了`java.lang.Object`中提供的公开接口，并应该在需要的时候提供正确的重载：

- `String toString()`
- `boolean equals(Object obj)`
- `int hashCode()`

Base of all: `java.lang.Object`

对象的字符串描述：`String toString()`

`System.out.println`和字符串拼接都接受对象作为参数或操作数，它们利用的就是`toString()`方法将任何对象都转化为字符串。设计好的字符串描述对于日志、调试等方面都有重要作用。

默认的`toString()`实现返回的是`<类型>@<内存地址>`，如`Circle@68be2bc2`，这显然没什么用。与其介绍怎么写比较好，我还是更推荐直接让IDEA自动生成一个就好了。

Base of all: `java.lang.Object`

对象的字符串描述：`String toString()`

IDEA内置了大量的`toString()`实现模版，我推荐使用第一个，也就是最简单的一个：`String concat (+)`。*[JEP280, Java9]*使用了`invokedynamic`指令提升了字符串拼接的性能表现，因此可以放心使用。

```
@Override
public String toString() {
    return "Circle{" + "r=" + r + '}';
}
```

Base of all: `java.lang.Object`

是否相同：`boolean equals(Object obj)`

`equals`方法提供了比较对象*内容*是否相同、或者说在*逻辑上*而不是在*物理地址*上相同的实现。正确实现`equals`可以保证Java集合框架中判断包含、`Set`去重复等基础语义的正确性。

Base of all: `java.lang.Object`

是否相同：`boolean equals(Object obj)`

`equals`方法应当满足：对于非`null`对象引用`x,y,z`，有：

1. *自反性*：`x.equals(x) == true`
2. *对称性*：`x.equals(y) == y.equals(x)`
3. *传递性*：`x.equals(y) && y.equals(z)` $\Rightarrow$ `x.equals(z) == true`
4. *一致性*：连续的调用`equals`方法的结果应当一致
5. `x.equals(null) == false`

Base of all: `java.lang.Object`

是否相同：`boolean equals(Object obj)`

但是，在Java中实现正确的`equals`实际上是一个较为困难的事情。我推荐的实现方法是：

```
class Class {
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Class clazz)) return false;
        return /* 依次比较各个域 */;
    }
}
```

Base of all: `java.lang.Object`

是否相同：`boolean equals(Object obj)`

- `if(this==o) return true;` 如果两个对象引用相同，那内容一定相同
- `if (!(o instanceof Class)) return false;` 如果不是这个类的对象，那么一定不相同
  - 有时这个实现为：

    ```
    if (o == null || getClass() != o.getClass()) return false;
    ```

  - 两种方法的区别在于是否接受子类与父类相同。一般情况是接受的，例如不同实现的集合`Set`，如果包含元素相同，也应当认为是相同的。当然，具体的实现选用需要具体情况讨论。

Base of all: `java.lang.Object`

是否相同：`boolean equals(Object obj)`

- 比较各个域的方法
  - 若为整数类型，直接`==`判断
  - 若为浮点类型，使用`Double.compare`或`Float.compare`判断，这两个静态类函数可以帮忙处理`NaN`相关的情况
  - 若为对象引用，使用`Objects.equals`判断，这个静态函数可以帮忙处理空指针

Base of all: `java.lang.Object`

是否相同：`boolean equals(Object obj)`

当然，如果没听懂，那就交给IDEA自动生成。如果你听懂了，也最好交给IDEA自动生成。~~如果不是讲课我也快忘了怎么写了，完全被IDEA惯坏了（悲~~

选择：Template = `java.util.Objects.equals and hashCode (java 7+)`
并且勾选 "Accept subclasses are parameter to equals() method"

Base of all: `java.lang.Object`

散列函数：`int hashCode()`

散列函数的有效实现可以提供`HashMap` `HashSet`等哈希表相关数据结构的正确性和高性能。

应当保证：

1. 相同的对象散列值必须相同：`x.equals(y) == true` $\Rightarrow$ `x.hashCode() == y.hashCode()`
2. 不同对象的散列值尽可能的不同
3. `equals()`和`hashCode()`**必须同时重载**。IDEA会帮你同时生成。

Base of all: `java.lang.Object`

散列函数：`int hashCode()`

为了减轻程序员设计散列函数的负担，Java提供了`Objects#hash`静态方法计算散列值，你只需要：

```
class Class {
    @Override
    public int hashCode() {
        return Objects.hash( /* 填入各个成员域 */ );
    }
}
```

Enhance Functionality

在上一大节中，我们学习了重用现有类的技术。在这一大节中，我们将会认识扩展类的功能的技术：*接口 interface*

`interface`实际上是`abstract class`的进一步抽象形式。`abstract class`允许含有不包含实现的抽象方法，而`interface`只定义了抽象方法，并且也不被允许有成员域（毕竟没有具体实现也不需要关联数据）。

`interface`描述了对象之间可以使用的方法，定义了对象之间互相通讯的协议。因为其只含有方法声明，因此也没有了继承的*只允许单一继承*的限制。实现接口会让类的功能扩展，而不会引入其他的实现依赖。

Enhance Functionality

Using `interface`

观察发现，我们的图形基类`Shape`只包含了抽象方法，因此可以直接转化为接口：

```
interface Shape {
    double area();
    void draw();
    boolean inside(double x, double y);
}
```

Enhance Functionality

Using `interface`

```
interface Shape {
    double area();
    void draw();
    boolean inside(double x, double y);
}
```

- 使用`interface`关键字声明接口
- 接口中的方法应当都为`public abstract`，因此无需再次声明

Enhance Functionality

Using `interface`

其他具体的图形类也都应该进行相应的修改：

```
class Circle implements Shape {
    private double r;
    public Circle(double r) { this.r = r; }

    @Override public double area() { return Math.PI * r * r; }
    @Override public void draw() { /* ... */ }
    @Override public boolean inside(double x, double y) {
        return x * x + y * y < r * r;
    }
}
```

Enhance Functionality

Using `interface`

```
class Circle implements Shape
```

使用 `implements` 关键字声明实现的接口列表。

Enhance Functionality

`static` fields

可以为`interface`内声明变量，但是这不是成员变量，而是默认的`public static final`变量，也就是接口的静态常量。

```
interface Shape {
    int MAX_WIDTH = 100, MAX_HEIGHT = 100;
}

int width = Shape.MAX_WIDTH;
```

Enhance Functionality

`static` methods

`interface`内允许有`static`方法，同样默认为`public`可见性。

```
interface Shape {
    static Shape createUnitCircle() { return new Circle(0, 0, 1); }
}

var unit = Shape.createUnitCircle();
```

Enhance Functionality

`default` methods

Java允许接口内存在带有实现的`default`方法。

```
interface Shape {
   boolean inside(double x, double y);
   default boolean outside(double x, double y) {
      return !inside(x, y);
   }
}
```

Enhance Functionality

`default` methods

- `default`提供默认的方法实现，这也意味着实现接口的类可以进行重载
- 接口的某些方法可以由已有的方法导出，比如判断集合`Set`是否为空`isEmpty()`时，对于大多数集合来说，可以通过判断`size() == 0`实现。这时就可以使用`default`来避免每个子类都写一句`size() == 0`
- `default`用于保证二进制兼容性。修改已有接口可能导致以前使用这个接口的类库出现方法未实现的错误。通过添加`default`函数可以避免这个问题。

Enhance Functionality

`default` methods

但是这又引入了*diamond problem*，即类实现了两个具有相同`default`方法的接口时出现的实现歧义。

在Java中，如果出现这种情况，你*必须*重载这个冲突的方法来提供自己的实现。如果想使用提供的`default`实现，你应当通过`<interface_name>.super.<func_name>()`手动指定使用哪一个接口。

Enhance Functionality

`default` methods

```
interface Inter1 {
    default void foo() { System.out.println("Inter1"); }
}

interface Inter2 {
    default void foo() { System.out.println("Inter2"); }
}

class Test implements Inter1, Inter2 {
    @Override
    public void foo() { Inter1.super.foo(); }
}
```

Enhance Functionality

`default` methods

- **`default` method VS `abstract` class**
- 可能有同学回想起了抽象类。确实，这两种机制都允许具体实现和纯接口混合，那么我们应当选择哪一种方案呢?
- **优先选择使用`interface`**
- 如果确实需要类提供的功能，如需要成员域、构造器、完全访问控制等只有类才具有的特性，才应该考虑抽象类。
- 接口没有单一继承限制，因此可以提升系统扩展性。

Inner class

内部类就是定义在类内部的类。内部类的定义也需要访问控制符。

```
public class Outer {
    public class Inner {
        // ...
    }
    // ...
}
```

Inner class

除了命名上的嵌套关系，内部类最重要的一点就是，具有对*外围对象*的访问权，甚至可以访问*外围对象*的 `private` 成员。

```
public class Outer {
    private String name = "Outer class";

    public class Inner {
        public String outerName () {
            return name;
        }
    }
}
```

Inner class

如果出现命名覆盖或者想明确使用外围对象的成员，需要使用 `.this` 语法：

```
public class Outer {
    private String name = "Outer class";

    public class Inner {
        private String name = "Inner class";
        public String outerName () {
            return Outer.this.name;
        }
    }
}
```

Inner class

可以在外部类里直接实例化内部类。如果不在外部类内，那么必须显式提供外围对象的引用，并使用 `.new` 语法实例化：

```
public class Outer {
    public class Inner { /* ... */ }
    public Inner innerInstance() { return new Inner(); }
}

// Outside of class Outer
Outer outer = new Outer();
Outer.Inner inner = outer.new Inner();
```

Inner class

内部类提供了隐藏实细节的一种方式。通过调整内部类的访问控制，外部系统甚至有可能找不到具体接口实现或子类的具体对象。这完全阻止了依赖于类型的编码，完全隐藏了实现细节。

Inner class

内部类提供了另一种*多重继承*的实现方案。通过返回内部类的实例，可以向外部提供继承自不同父类的对象，并且这些对象都可以访问外围对象内的任何数据和方法。

比如我们想让一个类同时提供`Class1` `Class2`的功能，但是我们又不能同时继承自这两个类，那么我们可以为我们的类提供两个内部类`SubClass1` `SubClass2`，分别继承自两个父类，并在需要某个类的功能时，实例化并返回对应内部类的对象。

这相当于共享充电宝的三个不同插口的充电线，我们需要哪个规格的插口，就“调用对应的方法”获取充电线的实例。

Inner class

```
class AppleLightning { /* ... */ }
class UsbTypeC { /* ... */ }
class UsbMicroTypeB { /* ... */ }

public class PowerBank {
    private int powerLeft;

    class AppleLightningImpl extends AppleLightning { /* ... */ }
    class UsbTypeCImpl extends UsbTypeC { /* ... */ }
    class UsbMicroTypeBImpl extends UsbMicroTypeB { /* ... */ }

    public AppleLightning lightning() { return new AppleLightningImpl(); }
    public UsbTypeC       typeC()     { return new UsbTypeCImpl(); }
    public UsbMicroTypeB  microB()    { return new UsbMicroTypeBImpl(); }
}
```

Inner class

Local inner class

局部内部类：你可以在方法内部甚至任意的作用域内定义类。

```
public class PowerBank {
    private int powerLeft;

    public AppleLightning lightning() {
        class AppleLightningImpl extends AppleLightning { /* ... */ }

        return new AppleLightningImpl();
    }
}
```

Inner class

Anonymous inner class

匿名内部类：观察下面的例子：

```
public class PowerBank {
    private int powerLeft;

    public AppleLightning lightning() {
        return new AppleLightning() {
            // Implements of AppleLightningImpl
        };
    }
}
```

Inner class

Anonymous inner class

`new BaseClass() { /* ... */ };` 创建了一个匿名内部类的对象，这实际上等价与（类名为编译器生成）：

```
class BaseClass$1 extends BaseClass { /* ... */ }
new BaseClass$1();
```

Inner class

Anonymous inner class

匿名内部类不允许有构造器，因为显然我们并不知道匿名类的名字。如果我们需要对类进行初始化操作，需要使用*实例初始化*:

```
public AppleLightning lightning() {
    return new AppleLightning() {
        { System.out.println("Init AppleLightning"); }
    };
}
```

Inner class

Anonymous inner class

在匿名内部类中使用的外部变量必须为 `final` 或者 等效 *`final`* 的。

```
interface Calculator { int apply(int a); }

Calculator mul(int x) {
    // x = 10; x is effectively final
    return new Calculator() {
        @Override int apply(int a) { return x * a; }
    }
}

mul(5).apply(3); // == 15
```

Inner class

Anonymous inner class

上一页的例子展示了类似函数式编程中*柯里化 Currying*的语法，只不过Java没有运算符重载，因此还是得写一个`.apply`。

这也展示了使用匿名内部类创建*闭包*的特性。*闭包*是将一个函数实现和周围状态/词法环境捆绑到一起的组合，它拥有能访问创建自己时的作用域的能力，在栗子中，则是可以访问外围方法的参数`int x`，即使函数已经退出运行。

Inner class

Anonymous inner class

Java对匿名内部类所用外部变量的`final`要求源自于内部管理闭包的机制：Java会在创建匿名内部类时，自动将所需的外部变量拷贝进去。如果后面对外部变量进行修改，那么匿名内部类中的值和外部变量的值会不一致，出现令人疑惑的语义问题。因此Java设计者让这些外部变量为`final`，不允许修改。

Inner class

Lambda Expression

有熟悉JavaScript或任何现代一点的语言的同学可能会说：你这Java的闭包保熟吗？怎么语法这么繁琐？不仅要`new`还得重写一遍方法签名？

这时自然就要介绍~~对Java来说~~重量级的语法特性了：

```
Calculator mul(int x) {
    return (int y) -> x * y;
}
```

是不是一下子就觉得现代起来了！

Inner class

Lambda Expression

Lambda表达式的参数列表也支持 `var` 自动推断：

```
Calculator mul(int x) {
    return (var y) -> x * y;
}
```

Inner class

Lambda Expression

Lambda表达式实际上可以看做一种特殊的匿名内部类：

- 只允许实现*函数式接口*，也就是只含有一个方法的接口。这种接口只代表了这个方法，也称为函数。
- 比如我们的`Calculator`就只包含一个方法`int apply(int)`，因此可成为函数式接口。

关于Lambda表达式的其他用法和标准库支持，我们会在*高级语法特性*章节中详细讲解。

Recap

在本章节*Fundamental OOP*中，我们学习了

- 面向对象的基本概念
- 创建、使用类的语法
- 重用类的方法：组合和继承
- 扩展类的方法：接口
- 使用内部类进一步封装实现
- 使用匿名内部类和Lambda表达式创建闭包

至此，Java的所有基本语法特性均已介绍，大家已经正式入门Java啦～

Chapter 3

Design Patterns

What are Design Patterns

- *设计模式 Design Patterns*是一套最佳实践，描述了软件开发人员在面临一些一般性问题时使用和总结出的解决方案。
- 1994年， Erich Gamma、Richard Helm、Ralph Johnson 和 John Vlissides合著了*Design Patterns - Elements of Reusable Object-Oriented Software*（《设计模式——可复用面向对象软件的基础》），首次提出了设计模式的概念，并对大多数常见的设计模式进行了分类组织和讨论。这四位作者也被称为*四人帮 GoF, Gang of Four*。
- 合理的利用设计模式让代码组织真正的工程化，减少耦合、提升可维护性、提升复用度。
- 本章我会带大家认识一些常见的设计模式，并介绍Java标准库中相应的设计范例。

OOP Principle

- 对接口编程而不是对实现编程
- 组合优先于继承
- 单一职责原则：类的职责应当尽可能的少而清晰
- 合理使用设计模式，而不要滥用

Creational Patterns

Catalog

创造型模式描述了如何创建对象：

1. *Builder* 建造者模式
2. *Factory Method* 工厂方法模式
3. *Abstract Factory* 抽象工厂模式
4. *Singleton* 单例模式

Creational Patterns

Builder

考虑我们现在看到的这个课件。我使用Marp将Markdown转化为PPT形式的课件。它也有将Markdown转化为Html等其他格式的能力。

Markdown的结构都是统一的，由多级标签、列表、代码块等基础元素组成。但是背后成品的表现形式是多样化的。如果我们直接写Markdown到PDF/Html的方法，那么可能需要重复写读取、解析Markdown的过程。这时我们就需要*构建者*模式来优化设计。

Creational Patterns

Builder

构建者模式通常由一个*指导者 Director*类、一个*构造者 Builder*接口/抽象类和若干个具体构建者实现类组成。
在我们的例子中，*Builder*提供构建其他格式文档所需要的接口：

```
interface DocumentBuilder {
    void appendText(String para);
    void appendHeader(String header, int level);
    void appendInlineCode(String code);
    // ... More elements
    String build();  // Get build result
}
```

Creational Patterns

Builder

*Director*管理Markdown的解析，并使用*Builder*构建转化后的文档：

```
class MarkdownDirector {
    private final DocumentBuilder builder;
    public MarkdownDirector(DocumentBuilder builder) { this.builder = builder; }
    public String convert(String markdownText) {
        markdownText.lines().forEach(this::parseLine);
        return builder.build();
    }
    private void parseLine(String line) {
        // ...
    }
}
```

Creational Patterns

Builder

*Director*管理Markdown的解析，并使用*Builder*构建转化后的文档：

```
// Demo implement
private void parseLine(String line) {
    if (line.startsWith("###### ")) builder.appendHeader(line.substring(7), 6);
    if (line.startsWith("##### ")) builder.appendHeader(line.substring(6), 5);
    // ...
    if (line.startsWith("# ")) builder.appendHeader(line.substring(2), 1);
    if (line.startsWith("- ")) builder.appendUnorderedList(line.substring(2));
    // ...
}
```

Creational Patterns

Builder

具体的*Builder*实现包含了对应具体目标文件格式的转化实现，比如：

```
class HtmlBuilder implements DocumentBuilder {
    private StringBuilder strBuf;
    @Override public void appendText(String para) {
        strBuf.append("<p>").append(para).append("</p>\n");
    }
    @Override public void appendHeader(String header, int level) {
        strBuf.append("<h" + level + ">")
              .append(header)
              .append("</h" + level + ">\n")
    }
}
```

Creational Patterns

Builder

具体的*Builder*实现包含了对应具体目标文件格式的转化实现，比如：

```
class HtmlBuilder implements DocumentBuilder {
    private StringBuilder strBuf;
    @Override public String build() {
        return strBuf.toString();
    }
}
```

Creational Patterns

Builder

作为这个格式转化系统的使用者，我们可以根据需要的文件格式提供相应的构建器：

```
public static void main(String[] args) throws IOException {
    DocumentBuilder builder = switch (args[2]) {
        case "pdf"  -> new PDFBuilder();
        case "html" -> new HtmlBuilder();
        default     -> { System.exit(-1); yield null; }
    }
    String markdown = Files.readString(Path.of(args[0]));
    String result = new MarkdownDirector(builder).convert(markdown);
    Files.writeString(Path.of(args[1]), result);
}
```

Creational Patterns

Builder

使用建造者模式的好处在于：

1. 分离*指导者*和*构建者*的具体实现。*指导者*无需关心具体结果对象的具体内部表示形式、实现方法和组成顺序。
2. 提升模块化，有利于合作开发，也有利于快速定位错误。

在以下情况使用建造者模式：

1. 复杂对象的创建*流程*和*具体内部实现*、*数据表示*是分离的

Creational Patterns

Builder

Java内最知名的构建者实例就是`java.lang.StringBuilder`，它提供了构建大型字符串所需的所有方法：

- `append(*)` `insert(int, *)` `replace(int, int, String)`
- `length()` `charAt(int)` `substring(int, int)`
- `toString()`

Creational Patterns

Abstract Factory

考虑IntelliJ IDEA和现在绝大多数GUI编辑器/IDE，它们都支持换主题皮肤的功能。我们可以为每一套皮肤都提供一整套GUI实现，但是显然这个工作量就过大了。

我们发现，GUI一般都由一些基本的元素组成，如按钮、菜单、文本框、文本区等。这些基本元素被称为*组件 Widget*。它们可能会有不同的渲染实现，但是GUI的布局总是相同的。因此我们可以分离这两者。

不同的皮肤实际上是提供了不同实现的组件的类，这个类被称为*抽象工厂 Abstract Factory*。

Creational Patterns

Abstract Factory

```
class Button    { /* ... */ }
class TextField { /* ... */ }
class TextArea  { /* ... */ }
class Toolbar   { /* ... */ }

// Abstract Factory
interface Theme {
    Button    createButton();
    TextField createTextField();
    TextArea  createTextArea();
    Toolbar   createToolbar();
}
```

Creational Patterns

Abstract Factory

```
class DarkButton    extends Button    { /* ... */ }
class DarkTextField extends TextField { /* ... */ }
class DarkTextArea  extends TextArea  { /* ... */ }
class DarkToolbar   extends Toolbar   { /* ... */ }

class DarkTheme implements Theme {
    @Override public Button    createButton()    { return new DarkButton(); }
    @Override public TextField createTextField() { return new DarkTextField(); }
    @Override public TextArea  createTextArea()  { return new DarkTextArea(); }
    @Override public Toolbar   createToolbar()   { return new DarkToolbar(); }
}
```

Creational Patterns

Abstract Factory

```
class LightButton    extends Button    { /* ... */ }
class LightTextField extends TextField { /* ... */ }
class LightTextArea  extends TextArea  { /* ... */ }
class LightToolbar   extends Toolbar   { /* ... */ }

class LightTheme implements Theme {
    @Override public Button    createButton()    { return new LightButton(); }
    @Override public TextField createTextField() { return new LightTextField(); }
    @Override public TextArea  createTextArea()  { return new LightTextArea(); }
    @Override public Toolbar   createToolbar()   { return new LightToolbar(); }
}
```

Creational Patterns

Abstract Factory

```
class GUI {
    private final Theme theme;
    public GUI(Theme theme) {
        this.theme = theme;
        build();
    }

    private void build() {
        var btn1 = theme.createButton();
        btn1.setGeometry(10, 10, 30, 10);
        btn1.setText("This is a button");
    }
}
```

Creational Patterns

Abstract Factory

应用抽象工厂模式可以：

1. 隔离具体实现类。
2. 更换具体实现类的家族，并保证这些实现类的行为一致性。

应当使用抽象工厂模式，当：

1. 你的系统和它所以依赖的组件的创建、组合和内部表示是独立的
2. 你的系统需要更换组件家族，而组件家族被设计为需要一起使用

Creational Patterns

Abstract Factory

Java Swing是Java提供的跨平台GUI库，它就提供了和我们例子很类似的更换主题，或者叫*Java Swing Look&Feel (L&F)*的功能，这是抽象工厂模式的最经典例子。

Creational Patterns

Factory Method

考虑一个需要数据库连接的应用。对不同数据库的连接是由不同类来实现和处理的，但是具体的数据库类型一般不是编译期决定的，而是由运行时的配置文件等控制。

通过*工厂方法 Factory Method*，可以将创建具体数据库连接的责任转移给工厂方法内部。其得名，就是因为它负责“制造”对象，就如同一个工厂一样。

Creational Patterns

Factory Method

```
interface DatabaseConnection {
    void open();
    Result executeSQL(String sql);
    void close();
}

class MySqlConnection    implements DatabaseConnection { /* ... */ }
class SqliteConnection   implements DatabaseConnection { /* ... */ }
class PostgresConnection implements DatabaseConnection { /* ... */ }
```

Creational Patterns

Factory Method

```
class DatabaseConnectionFactory {
    private String type;
    public DatabaseConnectionFactory(String type) { this.type == type; }

    public DatabaseConnection createConnection() {
        return switch(type) {
            case "mysql"    -> new MySqlConnection();
            case "sqlite"   -> new SqliteConnection();
            case "postgres" -> new PostgresConnection();
            default         -> null;
        }
    }
}
```

Creational Patterns

Factory Method

```
public static main(String[] args) {
    var factory = new DatabaseConnectionFactory(args[0]);
    var conn = factory.createConnection();
    conn.open();
    conn.executeSQL("SELECT * FROM user");
    conn.close();
}
```

Creational Patterns

Factory Method

工厂方法通常也可以是静态的 `static`：

```
class DatabaseConnectionFactory {
    public static DatabaseConnection createConnection(String type) {
        return switch(type) {
            case "mysql"    -> new MySqlConnection();
            case "sqlite"   -> new SqliteConnection();
            case "postgres" -> new PostgresConnection();
            default         -> null;
        }
    }
}
var conn = DatabaseConnectionFactory.createConnection(args[0]);
```

Creational Patterns

Factory Method

工厂方法是Java中应用最广泛的设计模式之一。

1. 消除对具体实现类的编译期绑定，进而强化了面向接口编程
2. 工厂方法实际上和构造器的功能非常相似，这个设计模式有时还被称作*虚拟构造器 virtual constructor*。工厂方法命名和语法都更加自由，可以作为更灵活、语义更明显的构造器使用。

Creational Patterns

Factory Method

Java中工厂方法的应用随处可见：

- 各个接口中的`of`系列静态方法几乎都是静态工厂方法：
  - `Set.of(...)` `Map.of(...)` `List.of(...)`
  - `Year.of()` `Mouth.of()` `DayOfWeek.of()`
  - `Path.of()`

Creational Patterns

Factory Method

Java中工厂方法的应用随处可见：

- `java.util.concurrent.Executors` 提供了若干工厂方法来实例化满足不同线程调度需求的执行器实现
- `java.sql.DriverManager` 是Java标准库对本节例子的具体体现
  - `java.sql.DriverManager#getConnection(String url)` 通过解析url中的协议名，自动在类路径中寻找符合要求的数据库驱动类，然后通过数据库驱动建立连接

Creational Patterns

Singleton

*单例 Singleton*模式说明某些类只应当存在一个实例。比如OS只应该有一个窗口管理球，也只应该有一个（逻辑上的）文件系统。对Java程序来说，Java运行时环境也应该是唯一的。事实也是如此，`java.lang.Runtime`类的确是一个单例，需要使用`java.lang.Runtime#getRuntime()`获取这个对象。

Creational Patterns

Singleton

```
class Singleton {
    private static INSTANCE;                    // 1.
    private Singleton() { /* ... */ }           // 2.

    public static Singleton getInstance() {     // 3.
        if(INSTANCE == null) {                  // 4.
            INSTANCE = new Singleton();
        }
        return INSTANCE;
    }
}
```

Creational Patterns

Singleton

1. 用私有静态成员域保存单例对象
2. 将构造器设置为私有的*private*，这可以防止他人新建其他的对象来破坏单例
3. 如果单例需要在并发环境下使用，可以加入`synchronized`关键字来保证这个函数同时只能有一个线程进入执行
4. 一般采用*懒初始化*的方法，在真正需要单例的时候再创建。

Structural Patterns

Catalog

结构化模式告诉我们如何组织、重用不同的类：

1. *Adapter* 适配器模式
2. *Decorator* 装饰器模式
3. *Flyweight* 享元模式
4. *Proxy* 代理模式

Structural Patterns

Adapter

*适配器 Adapter*模式，顾名思义，是通过*Adapter*类将两个不兼容的接口连接到一起，如下面的例子：

```
interface AppleLightning { /* ... */ }
interface UsbTypeC       { /* ... */ }

class TypeC2Lightning implements AppleLightning {
    private final UsbTypeC usb;
    public TypeC2Lightning(UsbTypeC usb) { this.usb = usb; }
    // Lightning methods
}
```

Structural Patterns

Adapter

Java标准库中最常用的适配器，就是连接`Reader` `Writer`与`InputStream` `OutputStream`的`InputStreamReader` `OutputStreamWriter`

- `Reader` `Writer`：用于读取**字符流**
- `InputStream` `OutputStream`：用于读取**字节流**
- `InputStreamReader` `OutputStreamWriter`：适配器类，用于处理原始**字节流**，通过字符集编码等信息转化为**字符流**

Structural Patterns

Adapter

```
var fileInput = new FileInputStream("input.txt");
var fileReader = new InputStreamReader(fileInput, "UTF-8");
var charBuf = new char[100];
var charCount = fileReader.read(charBuf, 0, 100);
var str = new String(charBuf);
```

Structural Patterns

Adapter

适配器模式的使用场景比较明显，写法也比较直接，因此介绍的篇幅较短。但是要注意，请不要滥用适配器，否则会对类型系统带来混乱，增加无用代码的数量。

Structural Patterns

Decorator

*装饰器 Decorator*模式相信大家都已经很熟悉了，大家的python课程应该讲过python对装饰器函数的语法支持。装饰器和适配器都是*包装*对象的模式，但是装饰器的重要区别在于，它是将接口包装到*同一个*接口，只不过装饰上增强的功能性。而适配器是包装为另一个不兼容的接口。

考虑VSCode的代码区，是在最朴素的文本区上，加入了缩略图、滚动条等组件，而且缩略图这种组件实际上在配置中是可以移除的。需要添加额外功能并支持动态添加/移除时，就应当使用装饰器模式。

Structural Patterns

Decorator

```
class Component { /* ... */ }
class TextArea extends Component { /* ... */ }

class ThumbnailDecorator extends TextArea {
    private final TextArea internal;
    public ThumbnailDecorator(TextArea textArea) { internal = textArea;}
}

class ScrollBarDecorator extends Component {
    private final Component internal;
    public ScrollBarDecorator(Component comp) { internal = comp;}
}
```

Structural Patterns

Decorator

```
class GUIFactory {
    public static Component createCodeArea(boolean enableThumbnail) {
        var textArea = new TextArea();
        return enableThumbnail ? new ScrollBarDecorator(new ThumbnailDecorator(textArea))
                               : new ScrollBarDecorator(textArea);
    }
}
```

Structural Patterns

Decorator

`java.io` 包的设计处处体现了装饰器模式。我们以输入流为例：

- `java.io.InputStream` 是通用的基类，基本功能就是 `read` 读取一个字节或者字节数组
- 终端流，即真正产生数据的流：
  - `java.io.ByteArrayInputStream` 从字节数组中读
  - `java.io.FileInputStream` 从文件中读
  - `java.io.PipedInputStream` 从连接到的另一个输出流中读

Structural Patterns

Decorator

`java.io`包的设计处处体现了装饰器模式。我们以输入流为例：

- 装饰器流，即为现有流添加额外功能的流：
  - `java.io.BufferedInputStream` 为现有流加个缓冲区以提升性能
  - `java.util.zip.CheckedInputStream` 边读边算CRC32等校验码
  - `java.util.zip.Inflater/DeflaterInputStream` 解压缩/压缩原始数据
  - `java.security.DigestInputStream` 边读边算MD5/SHA-1等消息摘要
  - `javax.crypto.CipherInputStream` 将原始数据通过AES/RSA等密钥算法加密/解密

Structural Patterns

Decorator

```
var fileInput = new FileInputStream("data.gzip");
var checkedInput = new CheckedInputStream(fileInput, new CRC32());
var gzipInput = new GZIPInputStream(checkedInput);

var data = gzipInput.readAllBytes();
var checksum = checkedInput.getChecksum().getValue();
assert(checksum == 1234567890L);
```

Structural Patterns

Decorator

装饰器模式带来的好处有：

1. 保证*单一职责原则*。可以看到，上面我介绍的一大堆输入流子类都可以用简短的一句话描述它们的全部功能。
2. 提供比继承更多的灵活性。我们可以任意组合装饰器流来加任何我们想使用的功能。

但是也有一些小缺点需要注意：

1. 可能会留下一大堆散件，比如获取校验码你仍然需要保留 `checkedInput`

Structural Patterns

Flyweight

考虑Minecraft，一个（最初）使用Java编写的、全球最知名的沙盒游戏之一。Minecraft的地形由一大堆方块组成，而这些方块都带有自己的数据，比如位置、方向、材质、碰撞箱、变种与其他信息（如铁砧损坏程度等）。我们很容易给出下面的简单设计：

Structural Patterns

Flyweight

```
abstract class Block {
    protected int x, y, z, facing;
    protected int hardness, toolLevel, resistance;
    protected Image texture;
    protected AABB aabb;
    public Block( /* ... */ ) { /* ... */ }

    public abstract void onTick();
    public abstract void onClick(Player player, int button);
    public abstract void onDestroy(Entity destroyer);
    // ...
}
```

Structural Patterns

Flyweight

```
class LogBlock extends Block {
    private int woodType;
    // Implementation of abstract methods ...
}
class AnvilBlock extends Block {
    private int damageType;
    // Implementation of abstract methods ...
}
class Cobblestone extends Block { /* ... */ }
class Bedrock extends Block { /* ... */ }
```

Structural Patterns

Flyweight

```
class World {
    private List<Block> blocks = new ArrayList<Block>();

    Block getBlockAt(int x, int y, int y) {
        // ...
    }
}
```

Structural Patterns

Flyweight

但是这种方法显然存在较大的性能问题：

有很多数据对于相同种类的方块都是重复的，没有必要每个位置都单独存一遍。对于Minecraft这种，世界大小上限约为$6 \times 10^6 \times 6 \times 10^6 \times 400$个`Block`的巨型对象，这些重复数据对于内存和硬盘空间来说都是不可接受的。

Structural Patterns

Flyweight

我们可以通过分离重复的数据和不重复的数据，或称为*固有状态 intrinsic*和*外围状态 extrinsic*，通过分离存储两种状态的对象，来减少运行时内存占用。比如，方块的材质、硬度、爆炸抗性等信息是固有状态，方块的位置、朝向等信息是外围状态。

Structural Patterns

Flyweight

为了区分外围状态，一种方法是为方块的更新或事件处理函数加入外围状态参数：

```
abstract class Block {
    public abstract void onTick(ExtrinsicBlockState states);
    // ...
}
```

Structural Patterns

Flyweight

另一种方式是直接为外围状态添加公开的 `set` 方法：

```
abstract class Block {
    void setX(int x) { this.x = x; }
    void setY(int y) { this.y = y; }
    void setZ(int z) { this.z = z; }
    // ...
}
```

Structural Patterns

Flyweight

然后通过工厂类和享元对象池等技术储存、构建不同固有状态的方块的享元：

```
class BlockFactory {
    private static Block createBlockInstance(int blockId) {
        return switch(blockId) {
            case 0 -> new Cobblestone();
            case 1 -> new AnvilBlock();
            // ...
        }
    }
}
```

Structural Patterns

Flyweight

然后通过工厂类和享元对象池等技术储存、构建不同固有状态的方块的享元：

```
class BlockFactory {
    private Map<Integer, Block> blockPool = new HashMap<>();
    public Block getBlockInstance(int blockId) {
        return blockPool.computeIfAbsent(
            blockId,
            BlockFactory::createBlockInstance
        );
    }
}
```

Structural Patterns

Flyweight

这样，`World`就不需要存储整个`Block`了，而只需要`blockId`指定享元对象，并在具体操作之前设置外围状态即可。

Structural Patterns

Flyweight

```
class World {
    private int[] blocks;
    private BlockFactory factory = new BlockFactory();
    public void tick() {
        // for xyz...
            var block = factory.getBlockInstance(blocks[idx]);
            block.setX(x); block.setY(y); block.setZ(z);
            block.onTick();
        // end for
    }
}
```

Structural Patterns

Flyweight

Java中的享元实例是基本整数数据类型的包装类的`valueOf`方法：

```
public final class Integer extends Number implements Comparable<Integer>, Constable, ConstantDesc {
    private static class IntegerCache {
        static final Integer[] cache;
    }

    @IntrinsicCandidate
    public static Integer valueOf(int i) {
        if (i >= IntegerCache.low && i <= IntegerCache.high)
            return IntegerCache.cache[i + (-IntegerCache.low)];
        return new Integer(i);
    }
}
```

Structural Patterns

Flyweight

如果基本类型的值在缓存范围内，就返回缓存的对象引用。对于`Integer`，根据Java语言规范，默认缓存$[-128, 127]$的整数。

Structural Patterns

Proxy

考虑远程过程调用（Remote Procedure Call, RPC），我们需要在本地调用远程代码。但是显然我们不可能 `new` 一个远程对象，我们不知道它具体的实现，更不知道怎么管理别的机器的内存。这时我们就需要一个*代理 Proxy*类作为本地的占位符，代替本地代码发送远程过程调用请求。这样我们可以和具体实现解耦合，甚至可以无缝在本地实现和远程代码实现之间转换。

Structural Patterns

Proxy

```
interface UserInfoRepo {
    List<String> getAllUserName();
}

class RPCUserInfoRepo implements UserInfoRepo {
    private final String httpEndpoint;
    public RPCUserInfoRepo(String endpoint) { httpEndpoint = endpoint; }
    @Override public List<String> getAllUserName() {
        // 1. Send HTTP Request
        // 2. Parse JSON Result
        // 3. Return List<String>
    }
}
```

Structural Patterns

Proxy

```
interface UserInfoRepo {
    List<String> getAllUserName();
}

class DBUserInfoRepo implements UserInfoRepo {
    private final Connection db;
    public DBUserInfoRepo(Connection db) { this.db = db; }
    @Override public List<String> getAllUserName() {
        // 1. sql: SELECT name FROM user
        // 2. Return List<String>
    }
}
```

Structural Patterns

Proxy

但是真正让代理模式在Java发光发热的是*真正的动态性*。Java允许在运行时改变字节码，创造新的类，换句话说，*运行时编译代码*。比如上例中的`RPCUserInfoRepo`和`DBUserInfoRepo`都不需要我们手动编写，我们只要声明：我们需要使用一个用户信息仓库，支持框架会自动生成、创建实现类并实例化对象供我们使用。

我们这次使用**Spring Data JPA**作为例子。*Spring*是Java最流行的后端框架的集合，*Spring Data*是其关系型数据库支持，*JPA*是*Java/Jakarta Persistence API*的缩写，是持久化层的API描述。

Structural Patterns

Proxy

我们声明一个 `User` 类：

```
@Entity
@Data
public class User {
  @Id
  @GeneratedValue(strategy=GenerationType.AUTO)
  private Long id;
  @NonNull private String nickName;
  @NonNull private String email;
}
```

Structural Patterns

Proxy

我们声明我们需要的数据仓库的接口，比如我们需要寻找对应昵称的所有用户：

```
@Repository
public interface UserRepository extends CrudRepository<User, Long> {
    List<User> findAllByNickName(String nickName);
}
```

CrudRepository 也内置了大量常用函数，包括 `findAll` `save` `delete` `count` 等。

Structural Patterns

Proxy

```
@SpringBootApplication
public class AccessingDataJpaApplication {
    public static void main(String[] args) {
        SpringApplication.run(AccessingDataJpaApplication.class);
    }
    @Bean
    public CommandLineRunner demo(UserRepository repo) {
        return (args) -> {
            repo.save(new User("xsun2001", "xcx19@mails.tsinghua.edu.cn"));
            var userList = repo.findAll();
        };
    }
}
```

Structural Patterns

Proxy

我们做了如下事情：

1. 定义了数据格式`User`类
2. 定义了我们想要的查询、更新方法`UserRepository`
3. 直接使用`UserRepository`进行数据操作
4. 让Spring启动程序

但是Spring替我们做的事情包括但不限于：

Structural Patterns

Proxy

1. 自动发现`User`，根据`User`内部的注解和成员变量，自动生成数据库表生成语句，如有必要，自动合并、迁移、重整已有数据表中的数据
2. 自动发现`UserRepository`，根据方法名称、参数、返回值等信息，自动解析生成SQL语句，自动处理数据库连接和SQL执行，自动组织数据库的执行结果，生成本节所说的*Proxy*类
3. 自动发现`@Bean public CommandLineRunner demo(UserRepository repo)`，根据参数得知我们想要一个`UserRepository`实现，因此实例化代理类并注入。根据返回值类型判断需要运行返回结果。

Structural Patterns

Proxy

*动态代理*配合*依赖注入*是究极解耦合方案。因此Java自己也不知道实现代码到底放在哪里，直到真正需要的时候才会自动生成。

可以从*Spring Data JPA*的例子看到，我们只关心*接口*和*数据*的设计与使用，消除了我们手动操作、处理数据库的繁琐代码。

再加上依赖注入，我们甚至都没有`new`过任何相关的类，全都交给*Spring*处理。

Behavioral Patterns

Catalog

行为模式告诉我们如何让系统中的对象协作：

- *Template Method* 模版方法模式
- *Iterator* 迭代器模式
- *Visitor* 访问者模式
- *Strategy* 策略模式

Behavioral Patterns

Catalog

行为模式实际上还有很多，但是其中有些模式我认为只是把比较直观的做法赋予了个名字，没有本质上的提升。因此我挑选了几个最为常用、最有代表性的行为模式进行讲解。

行为模式描述的更多的是实现细节，因此后续描述时会直接介绍Java标准库中的例子而不介绍过于抽象的表达。~~绝对不是我想偷懒！~~

Behavioral Patterns

Template Method

*模版方法 Template Method*模式可以为方法比较多的接口提供一组共享的默认实现，来简化子类派生的过程。Java的集合框架也大量应用了这个模式，典型的命名特征是`Inter`接口和`AbstractInter`抽象类来提供模板实现。

比如`List`接口有27个需要实现的方法。但实际上，一个最简单的、不可变的列表只需要实现一个`get(int)`方法即可。这种列表实际上用途也非常广泛，比如在整个程序中传递支持的文件类型列表信息等。`AbstractList`就为这种用途提供了除了`get(int)`之外的所有接口的模版实现。

Behavioral Patterns

Template Method

`AbstractList<E>` 为不同功能的列表提供了不同的重载方式：

- 不可变的列表只需要实现 `get(int)` `size()`
- 可变的列表需要实现 `get(int)` `set(int, E)`
- 可以扩展大小的列表还应该实现 `add(int, E)` `remove(int)`

其余的包括 `addAll` `removeAll` `iterator` `subList` `contains` 等其他方法全都可以由上述基本方法派生出来，并在模版方法里提供具体实现。

Behavioral Patterns

Iterator

*迭代器 Iterator*相信大家或多或少都听说过。*Iterator*提供了顺序访问一组*聚合*对象的通用方法，而无需关心内部存储实现，比如用数组还是用哈希表等。

Behavioral Patterns

Iterator

Java集合框架提供了`java.util.Iterator`用于描述集合元素迭代器：

```
package java.util;
public interface Iterator<E> {
    boolean hasNext();
    E next();
    default void remove() { throw new UnsupportedOperationException("remove"); }
    default void forEachRemaining(Consumer<? super E> action) {
        Objects.requireNonNull(action);
        while (hasNext())
            action.accept(next());
    }
}
```

Behavioral Patterns

Iterator

它的接口都很简洁明了：

- `hasNext()` 是否还有下一个元素，如果有，
- `next()` 获取下一个元素
- `remove()` 删除当前元素。默认不支持删除。
- `forEachRemaining(Consumer<? super E> action)` 对后续所有元素应用 `action`

Behavioral Patterns

Iterator

`Iterator`的基本使用可以直接参考`forEachRemaining`的实现：

```
var iter = getIterator();
whiel(iter.hasNext()) {
    var obj = iter.next();
    // do something with obj
}
```

Behavioral Patterns

Iterator

`java.lang.Iterable<T>` 接口描述了可以被迭代的聚合对象：

```
public interface Iterable<T> {
    Iterator<T> iterator();

    default void forEach(Consumer<? super T> action) {
        Objects.requireNonNull(action);
        for (T t : this) { action.accept(t); }
    }
    default Spliterator<T> spliterator() {
        return Spliterators.spliteratorUnknownSize(iterator(), 0);
    }
}
```

Behavioral Patterns

Iterator

- `java.lang.Iterable<T>` 只有一个接口函数： `iterator`，返回一个迭代器 `Iterator<T>`。
- `forEach` 方法的实现展现了 `Iterable` 的另一种用法
  - 除了使用 `iterator()` 方法获取迭代器并按照常规方法访问，Java支持*for-each*循环来遍历可迭代对象：
  - `for (T t : iterable) { /* do something with t */ }`
- `spliterator()` 提供了*可并发的*迭代器实现

Behavioral Patterns

Iterator

Java集合框架和标准库的很多类都是可迭代的：

- 集合框架本身就是元素的集合，因此自然都是可迭代的
- `java.nio.file.Path` 是自身的可迭代对象，便于分离目录结构
- `java.nio.file.DirectoryStream<T>` 是 `T` 的可迭代对象，用于获取目录下的所有文件和子目录信息

Behavioral Patterns

Visitor

*访问者 Visitor*模式也是遍历一组对象所用的设计模式，但是与迭代器不同的是，访问者一般用于访问层次化、结构化对象，并且子对象的类型或者处理方式可能不同。最经典的例子就是文件目录访问者：

`java.nio.file.FileVisitor`

Behavioral Patterns

Visitor

```
package java.nio.file;

public interface FileVisitor<T> {
    FileVisitResult preVisitDirectory(T dir, BasicFileAttributes attrs)
        throws IOException;    // 进入目录之前
    FileVisitResult visitFile(T file, BasicFileAttributes attrs)
        throws IOException;    // 处理文件
    FileVisitResult visitFileFailed(T file, IOException exc)
        throws IOException;    // 处理文件错误
    FileVisitResult postVisitDirectory(T dir, IOException exc)
        throws IOException;    // 目录遍历完成
}
```

Behavioral Patterns

Visitor

`FileVisitResult`是一个枚举类型，用于控制后续访问目录的过程，比如继续遍历文件，或者在找到所需要的文件后退出等等：

```
package java.nio.file;

public enum FileVisitResult {
    CONTINUE,         // 继续遍历
    TERMINATE,        // 终止遍历
    SKIP_SUBTREE,     // 继续遍历，但是跳过子目录下的所有文件/目录
    SKIP_SIBLINGS;    // 继续遍历，但是跳过同目录下的其他文件/目录
}
```

Behavioral Patterns

Visitor

有时我们可能只需要某个方法的实现，这时就是*模版方法*体现作用的时候了。Java提供了 `java.nio.file.SimpleFileVisitor<T>` 作为 `FileVisitor<T>` 的空实现。实现的方式大概是没有错误的时候就 `CONTINUE`，有异常就抛出异常终止整个过程。

Behavioral Patterns

Visitor

使用 `java.nio.file.Files#walkFileTree` 遍历文件树：

```
var start = Path.of(".");

Files.walkFileTree(start, new SimpleFileVisitor<Path>() {
    // ... Implementations
}
```

Behavioral Patterns

Visitor

比如我们来统计目录下java源代码文件的数量，就可以：

```
class FileCounter<T extends Path> extends SimpleFileVisitor<T> {
    private int count = 0;
    public int getCount() { return count; }

    private final String ext;
    public FileCounter(String ext) { this.ext = ext; }

    // ...
}
```

Behavioral Patterns

Visitor

```
class FileCounter<T extends Path> extends SimpleFileVisitor<T> {
    // ...

    @Override
    public FileVisitResult visitFile(T file, BasicFileAttributes attrs) throws IOException {
        if (file.endsWith(ext)) ++count;
        return FileVisitResult.CONTINUE;
    }
    @Override
    public FileVisitResult visitFileFailed(T file, IOException exc) throws IOException
        { return FileVisitResult.CONTINUE; }
    @Override
    public FileVisitResult postVisitDirectory(T dir, IOException exc) throws IOException
        { return FileVisitResult.CONTINUE; }
}
```

Behavioral Patterns

Visitor

```
public static void main(String[] args) throws IOException{
    var visitor = new FileCounter<Path>(args[1]);

    Files.walkFileTree(Path.of(args[0]), visitor);

    System.out.println(
        "Counts of " + args[1] + " file in " + args[0] + " = " + visitor.getCount()
    );
}
```

Behavioral Patterns

Strategy

一个功能可能有多种不同的算法实现。抽象出*算法*实例的模式称为*策略 Strategy*模式。*Strategy*可以在运行时调配不同的算法实现，这对使用功能的客户端来说是无缝的。这个模式最经典的例子是对数组排序使用的不同*排序器 Comparator*。

```
@FunctionalInterface
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

Behavioral Patterns

Strategy

比如我们要根据用户的年龄进行排序，可以定义这样的排序器：

```
@Data
class User {
    private String name;
    private int age;
}

public class UserComparator implements Comparator<User> {
    int compare(User o1, User o2) {
        return Integer.compare(o1.getAge(), o2.getAge());
    }
}
```

Behavioral Patterns

Strategy

使用 `Arrays#sort` 排序数组，使用 `Collections#sort` 排序列表。它们均可以接受一个排序器作为第二个参数：

```
User[] userArray = fetchUsers();

Arrays.sort(userArray, new UserComparator());
```

Behavioral Patterns

Strategy

有同学可能发现了，`Comparator`是一个*函数式接口*`@FunctionalInterface`，因此可以使用Lambda表达式：

```
Arrays.sort(
    userArray,
    (var o1, var o2) -> Integer.compare(o1.getAge(), o2.getAge())
);
```

Behavioral Patterns

Strategy

`Comparator` 还提供了一些工厂方法，用于产生各种各样的比较器实现：

- `reversed()`：创建当前比较器的反转版
- `thenComparing()` 系列：当前比较器如果相等，那么再用提供的比较器或键值比较
- `comparing()` 系列：提取对象中的某个键进行比较

Behavioral Patterns

Strategy

比如我们要提供一个先根据年龄排序，再根据名字的长度排序，长度相同的忽略大小写进行排序的比较器，我们可以使用：

```
var comparator = Comparator<User>
        .comparingInt(User::getAge)
        .thenComparing(User::getName,
            Comparator<>.comparingInt(String::length)
                        .thenComparing(String.CASE_INSENSITIVE_ORDER);
        )
```