Java程序设计语言

为什么要学Java

- 更加安全
  - Static type checking
  - Easy to understand
  - Easy to debug
- 更加广泛
  - Popular in both academic and industrial fields
  - Generally faster and more efficient than python
  - Mobile application
  - Server-side web programming
  - Many interesting libraries

A good programmer must be multilingual

安裝

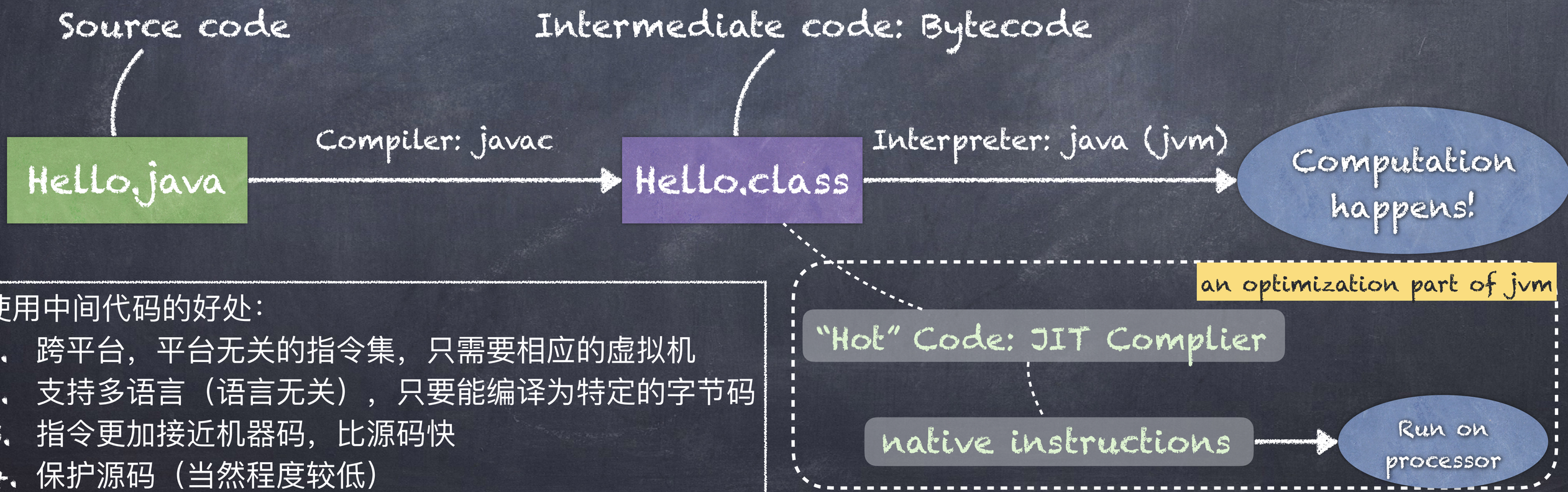
- Java Development Kit (JDK)
  - https://www.oracle.com/java/technologies/downloads/
- Integrated Development Environment (IDE)

https://www.eclipse.org/downloads/

https://www.jetbrains.com/idea/

运行

- 在Java中，编译和解释分两步执行



使用中间代码的好处：

1. 跨平台，平台无关的指令集，只需要相应的虚拟机
2. 支持多语言（语言无关），只要能编译为特定的字节码
3. 指令更加接近机器码，比源码快
4. 保护源码（当然程度较低）

An beginning example: Computing hailstones

Hailstones sequence: Sequences of integers generated in the Collatz problem

```java
// Java
int n = 3;
while (n != 1) {
    System.out.println(n);
    if (n % 2 == 0) {
        n = n / 2;
    } else {
        n = 3 * n + 1;
    }
}
System.out.println(n);
```

```python
# Python
n = 3
while n != 1:
    print(n)
    if n % 2 == 0:
        n = n / 2
    else:
        n = 3 * n + 1

print(n)
```

⚠️关键词意思相近（while、if、else）、java语句（statement）的结尾是分号";"，java的while、if的条件从句需要用小括号包裹、java使用大括号表达语句块（而不是python的缩进）

类型（Type）

类型

- Java和python在语法上最大的不同就是Java需要对变量进行类型声明（Type declaration）
  - 比如在上述例子中：

```
#Java
int n = 3;
```

```
#Python
n = 3
```

  - Java需要对 n 声明其类型是 int
- 一个类型本质上指的是所包含的所有值的集合，以及能够在这个集合上进行的操作的集合

为什么需要类型

优点

- 类型检查可以很早发现一些简单错误：比如： 2+ true + “a”
- 生成更加高效的目标代码：比如可以更加高效的组织函数帧（需要存储的空间）、寄存器的使用（需要预先准备多少寄存器来存储值），根据类型选择合适的机器指令。
- 类型给了程序更多信息语义，方便进一步的代码优化：比如有些类型安全的转化就可以去除异常处理函数部分。
- 模块化、有助于代码阅读和维护：比如有了类型之后，数据不再是单纯的0、1比特，而是有了抽象的意义

缺点

- 程序员会受限，如：
  - 有些正确的程序可能被编译不通过，比如：

conservative

Long to compute, but always return true

```
if f(x) then 1 else (5 + true)
```

静态（static）和动态（dynamic）类型

- 静态类型：变量的类型在编译阶段（compile time）就已经被获知（运行之前）
  - 一般情况下，静态类型语言（statically-typed language）需要程序员显式的标注变量类型（如Java, C, C++），但也有静态类型语言（如OCaml）利用类型推断（type inference）使得程序员不必标注。
- 动态类型：解释器在程序运行时（runtime）才根据变量的值来标注其类型（如python、Javascript）

类型检查（Type checking）

类型检查一般用来检测类型相关的错误（⚠️不是和动态、静态语言对应）

- 静态检查在程序运行之前发现相关错误：
  - 语法错误（python也会做相关的检查，如缩进错误等）、操作数类型错误、参数类型错误、返回类型错误等
- 动态检查在程序执行时发现类型错误（静态分析无法找到所有可能的类型错误（根据莱斯定理））：
  - 非法的具体的操作数值（比如除数为0）、非法的类型转换（比如Integer.valueOf(“hello”)）、越界错误、null对象的方法调用等。

Java语言中的类型

- Java的原始数据类型（Primitive data types）：

As a field

类型	数据位	范围	默认值
byte(字节型)	8	-128～127，即 - 2^7～2^7-1	0
short(短整型)	16	-32768～32767，即-2^{15}～2^{15}-1	0
int(整型)（默认）	32	-2, 147, 483, 648～2, 147, 483, 647，即-2^{31}～2^{31}-1	0
long(长整型)（L或L）	64	-9, 223, 372, 036, 854, 775, 808～9, 223, 372, 036, 854, 即-2^{63}～2^{63}-1	0

Java语言中的类型

- Java的原始数据类型（Primitive data types）：

类型	数据位	范围	默认值
float(单精度)（f或F）	32	1.4E-45～3.4E+38 (positive or negative)	0.0f
double(双精度)(默认)	64	4.9E-324～1.8E+308 (positive or negative)	0.0d

Java语言中的类型

- Java的原始数据类型（Primitive data types）：

类型	数据位	范围	默认值
boolean	1	true, false	false
char	16	'\u0000' ~ '\uffff'	'\u0000'

Unicode

Java语言中的类型

- Java的内建的一些常用对象类型（object types）：

类型	所属类	描述	默认值
String	java.lang.String	字符串，如 "hello!"	null
Number	java.lang.Number	数字类	null

BigInteger, Byte, Double, Float, Integer, Long, Short

常量（constant）和变量（variable）

- 常量：值永远不允许被改变的量（immutability）。用关键字final来修饰
  - final int MAX = 10;
  - final float PI =3.14f;
  - final double PI =5.1235;　　可以加一个d：5.1235d
- 可以连续声明
  - final int NUM1 = 14, NUM2 = 25, NUM3 = 36;

A convention: Uppercase

常量和变量

- 变量就是用来绑定"值"的。当然相对于常量，这些值是可以改变的
- 变量的声明：
  - double salary;
  - int vacationDays;
  - long earthPopulation;
  - boolean done;
- 可以一行连续声明：
  - int i, j; // both are integers

常量和变量

- 变量的可以声明的同时赋值（初始化），也可以之后赋值

```
int i, j=0;
i = 8;
float k;
k = 3.6f;
```

- 作为局部变量时（local variable），没有赋值之前无法使用

```
int vacationDays;
System.out.println(vacationDays); // ERROR--variable not initialized
```

操作（Operation）

功能	运算符
算数	+、-、*、/、%、++、--
关系	>、<、>=、<=、==、!=
逻辑	!、&&、||
位运算	>>、<<、>>>、&、|、^、~
赋值	=、+=、-=、/=、*= 、%= 等等
条件	? :

操作（Operation）

算数

操作符	描述	例子
+	加法	int a = 1, b = 1; int c = a + b; // c 等于2
-	减法	int a = 1, b = 1; int c = a - b; // c 等于0
*	乘法	int a = 1, b = 1; int c = a * b; // c 等于1
/	除法	int a = 2, b = 3; System.***out***.println(b/a); // ⚠打印1
%	取余	int a = 22, b = 3; System.***out***.println(b%a); //返回3

++	自增	int a = 22; System.***out***.println(a++); //打印22，但此时a已经为23
		int a = 22; System.***out***.println(++a); //打印23，此时a为23

--	自减	int a = 22; System.***out***.println(a--); //打印22，但此时a已经为21
		int a = 22; System.***out***.println(--a); //打印21，此时a为21

操作（Operation）

- 关系（返回true或者false）：
  - 3 > 4 //大于关系， 返回false
  - 3 < 4 //小于关系， 返回true
  - 2 >= 2 //大于等于关系， 返回true
  - 2 <= 2 //小于等于关系，返回true
  - 2 == 2 //相等关系，返回true
  - 2 != 2 //不等关系，返回false
- 不要在浮点数之间作"=="的比较，误差难免存在。

操作（Operation）

- 逻辑：
  - a && b： 逻辑与运算，当a和b都为true，才返回true （其他都为false）
  - a || b： 逻辑或运算，当a和b都为false，才返回false （其他都为true）
  - !a：逻辑非操作，当a为真，返回假，a为假返回真

操作（Operation）

位运算

位运算是对整数在比特位上进行的运算,比如：
12 = 00000000000000000000000000001100 (In Binary)
25 = 00000000000000000000000000011001 (In Binary)

操作符	描述	例子
&	对应位的“与”操作	int a = 12, b = 25, System.out.println(a & b); // 8， 即 0b1000
|	对应位的“或”操作	int a = 12, b = 25, System.out.println(a | b); // 29， 即 0b11101
^	对应位的“异或”操作	int a = 12, b = 25, System.out.println(a ^ b); // 21， 即 0b10101
~	对应位的“取反”操作	int a = 12; System.out.println(~a); // -13, 即 0b11111111111111111111111111110011
<<	按位左移指定的位数	int a = 12; System.out.println(a << 2); //返回48， 即0b110000
>>	按位右移指定的位数(有符号)	int a =-13; System.out.println(a >> 2); //返回-4, 即 0b11111111111111111111111111111100
>>>	按位右移指定的位数（无符号，即补0）	int a =-13; System.***out***.println(a >>> 2); //返回1073741820， 即 0b00111111111111111111111111111100

操作（Operation）

- 赋值：
  - int a = 3*4；// 将右操作数赋值给左边
  - a += c 等价于 a = a + c （-=、*=、/=、%=、&=、<<=、>>=、^=、|= 类似）
  - 注意如果左右类型不同，则会发生转换
  - 比如 float a = 3;
  - int a = (int) 3.4;
- 可以连续赋值：
  - a = b = c = 3;

操作（Operation）

- 条件：
  - 三元（ternary）运算符： <conditional exp> ? <true-exp> : <false-exp>

```
int a , b;
a = 10;
b = (a == 1) ? 20 : 30;// 如果 a 等于 1 成立，则设置 b 为 20，否则为 30
```

操作符的优先级

优先级	运算符	结合性
1	()、[]、{}	从左向右
2	!、+（正号）、-（负号）、~、++、--	从右向左
3	*、/、%	从左向右
4	+、-	从左向右
5	<<、>>、>>>	从左向右
6	<、<=、>、>=	从左向右
7	==、!=	从左向右
8	&	从左向右
9	^	从左向右
10	|	从左向右
11	&&	从左向右
12	||	从左向右
13	?:	从右向左
14	=、+=、-=、*=、/=、&=、|=、^=、~=、«=、»=、>>>=	从右向左

比如：x = 7 + 3 * 2

> 先做乘法、然后加法、最后赋值

> 大体上，括号>一元>算术>关系>逻辑>三元>赋值
> 同级的运算大部分是从左到右
> 赋值、一元和三元是从右到左

编写程序时尽量的使用括号（）运算符号来实现想要的运算次序，以免产生难以阅读或含糊不清的计算顺序

类型转换（type conversion）

- 隐式类型转换：自动地被JVM进行类型转换（implicit conversion, also known as type coercion）
- 显式类型转换：手动地由程序员进行转换。（explicit conversion, also known as type casting）

隐式类型转换

- 当需要从低级类型向高级类型转换时，Java会自动完成类型转换。
- 低级类型是指取值范围相对较小的数据类型，高级类型则指取值范围相对较大的数据类型。

隐式类型转换

```
byte b = 75;
char c = 'c';
int i = 794215;
long l = 9876543210L;
long result = b * c - i + l;
```

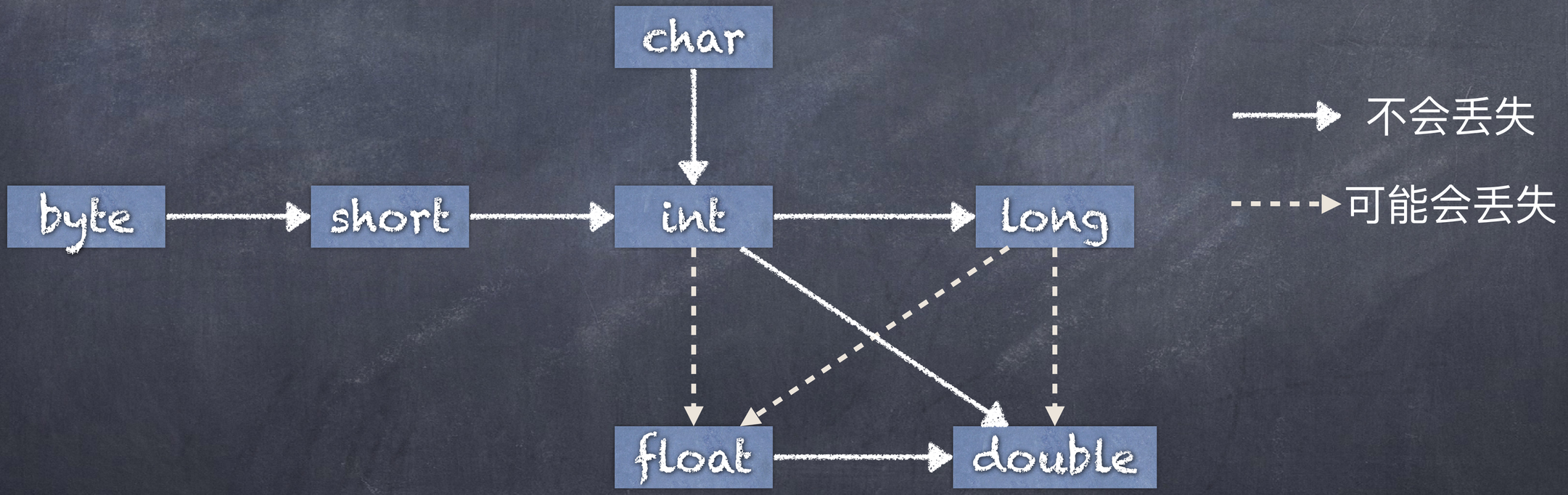
转化为运算中的最高的类型

```
byte b = 75;
short s = 9412;
char c = 'c';
int result = b + s * c;
```

当只含有byte、short、char时，运算会统一转为int

隐式类型转换

- 对于数据类型为byte、short、int、long、float和double的变量，可以将数据类型较小的数据或变量，直接赋值给数据类型较大的变量（注意精度可能会丢失）。




```
int n = 123456789;
float f = n; // f is 1.23456792E8
```

显式类型转换

- 如果要将较长的数据转换成较短的数据时（不安全），就要进行显式的类型转换　　<type> variable

比如：　int i = (int) 7.5;

在执行强制类型转换时，可能会导致数据溢出或精度降低。例如上面语句中变量i的值最终为7，导致数据精度降低。

字符串与数值之间的类型转换

- 字符串转可以换成数值型数据
  - 比如 String MyNumber = "1234.56"; float MyFloat = Float.parseFloat(MyNumber);
- 还有一些其他的相应转换：

  Byte.parseByte(String)

  Short.parseShort(String)

  Integer.parseInt(String)

  Float.parseFloat(String)

  Double.parseDouble(String)

字符串与数值之间的类型转换

- 在Java语言中，字符串可用加号"+"来实现连接操作

```
int MyInt=1234;
String MyString=""+MyInt;
```

控制

分支

```
if (<conditional>) {
   statements
}
```

```
if (<conditional>) {
   statements
} else {
   statements
}
```

```
if (<conditional>) {
   statements
} else if <conditional2> {
   statements
} else if <conditional3>{
   statements
}
```

```
if (<conditional>) {
   statements
} else if <conditional2> {
   statements
} else if <conditional3>{
   statements
} else {
   statements
}
```

分支

- 可以使用switch语句来写多分支

```
enum Day { SUNDAY, MONDAY, TUESDAY,
WEDNESDAY, THURSDAY, FRIDAY, SATURDAY; }
int numLetters = 0;
Day day = Day.WEDNESDAY;
switch (day) {
    case MONDAY:
    case FRIDAY:
    case SUNDAY:
        numLetters = 6;
        break;
    case TUESDAY:
        numLetters = 7;
        break;
    case THURSDAY:
    case SATURDAY:
        numLetters = 8;
        break;
    case WEDNESDAY:
        numLetters = 9;
        break;
    default:
        throw new IllegalStateException("Invalid day: " + day);
}
System.out.println(numLetters);
```

Java14新特性：switch表达式

```
Day day = Day.WEDNESDAY;
System.out.println(
    switch (day) {
        case MONDAY, FRIDAY, SUNDAY -> 6;
        case TUESDAY                -> 7;
        case THURSDAY, SATURDAY     -> 8;
        case WEDNESDAY              -> 9;
        default -> throw new IllegalStateException("Invalid day: " + day);
    }
);
```

分支

- 可以使用yield关键字来给出"switch复合语句的值"，即yield将switch语句变为了一个switch表达式

```
Day day = Day.WEDNESDAY;
int numLetters = switch (day) {
    case MONDAY:
    case FRIDAY:
    case SUNDAY:
        System.out.println(6);
        yield 6;
    case TUESDAY:
        System.out.println(7);
        yield 7;
    case THURSDAY:
    case SATURDAY:
        System.out.println(8);
        yield 8;
    case WEDNESDAY:
        System.out.println(9);
        yield 9;
    default:
        throw new IllegalStateException("Invalid day: " + day);
};
System.out.println(numLetters);
```

```
int numLetters = switch (day) {
    case MONDAY, FRIDAY, SUNDAY -> {
        System.out.println(6);
        yield 6;
    }
    case TUESDAY -> {
        System.out.println(7);
        yield 7;
    }
    case THURSDAY, SATURDAY -> {
        System.out.println(8);
        yield 8;
    }
    case WEDNESDAY -> {
        System.out.println(9);
        yield 9;
    }
    default -> {
        throw new IllegalStateException("Invalid day: " + day);
    }
};
```

yield可以让 —> 包含复合语句

循环

- java中的三种循环结构：
  - while循环
  - do...while循环
  - for 循环

循环

```
while (<conditional>) {
    statements
}
```

```
int i, sum;
sum = 0;
i = 0;
while (i <= 100) {
        sum += i;
        i++;
}
```

```
do {
    statements
} while (<conditional>);
```

```
int i, sum;
sum = 0;
i = 0;
do {
        sum += i;
        i++;
} while (i <= 100);
```

```
for(<exp1>; <conditional>; <exp2>){
    statements
}
```

```
int sum = 0;
for (int i = 0; i <= 100; i++){
    sum += i;
}
```

跳出循环

- continue： 结束所在的本次循环
- break：终止所在的循环

跳出循环

1!+2!+3!+4!+5!

```
int sum = 0;
int[] sequence = {1, 2, 3, 4, 5};
for (int j : sequence) {
    int temp = 1;
    for (int i = 1; i <=j; i++)
        temp *= i;
        sum +=temp;
     }
System.out.println(sum);
```

1!+2!+3!+5!

```
int sum = 0;
int[] sequence = {1, 2, 3, 4, 5};
for (int j : sequence) {
    int temp = 1;
    if(j == 4)
        continue;
     for (int i = 1; i <=j; i++){
        temp *= i;
        sum +=temp;
      }
System.out.println(sum);
```

1!+2!+3!

```
int sum = 0;
int[] sequence = {1, 2, 3, 4, 5};
for (int j : sequence){
    int temp = 1;
    if(j == 4)
        break;
     for (int i = 1; i <=j; i++){
        temp *= i;
        sum +=temp;
      }
System.out.println(sum);
```

容器

数组

- 数组:相同数据类型的元素按一定顺序排列的集合。
- Java中，数组元素可以为简单数据类型，也可为复杂的对象

```
type[ ] variable;                    //declaration
variable =new type[size]; //dynamic memory allocation

type[ ] variable= new type[size]; // declaration and allocation
```

```
int[ ] x;
x =new int[size];

int[ ] x= new int[10];
```

数组

- Java语言内存分配：
  - 栈内存：定义的基本类型的变量和对象的引用变量，超出作用域将自动释放。
  - 堆内存：存放由new运算符创建的对象和数组。由Java虚拟机的自动垃圾回收器来管理。

environment diagram

数组

- 用new分配内存的同时，数组的每个元素都会自动赋值默认值，即
  - 整型为0，实数为0.0，布尔型为false，引用型为null

```
int[] x = new int[2] ==> {0, 0}
double[] x = new double[2] ==> {0.0, 0.0}
boolean[] x = new boolean[2] ==> {false, false}
```

数组

- 数组的访问： <variable> [index]

```
int[] x = new int[10];
x[0] = 1;
x[1] = 2;
System.out.println(x[5]);
```

赋值

- 对于每个数组都有一个属性length指明它的长度

```
System.out.println(x.length); // 10
```

Java会在运行时对数组的越界进行检查

数组

- 初始化：

```
int[]  a = {1,2,3,4,5};
```

```
int[]  a = new in[] {1,2,3,4,5};
```

```
int[] a = new int[5];
for(int i = 0; i < a.length; i ++ ){
        a[i] = i+1;
}
```

迭代

```
for (type element : iterable) {
    statements
}
```

```
int[] arr={1,2,3,4,5};
for (int element : arr)
    System.out.println(element);
```

```
int[] arr={1,2,3,4,5};
for (int i = 0; i < arr.length; i++){
    int element = arr[i];
    System.out.println(element);
}
```

多维数组

声明

type[][] variable;

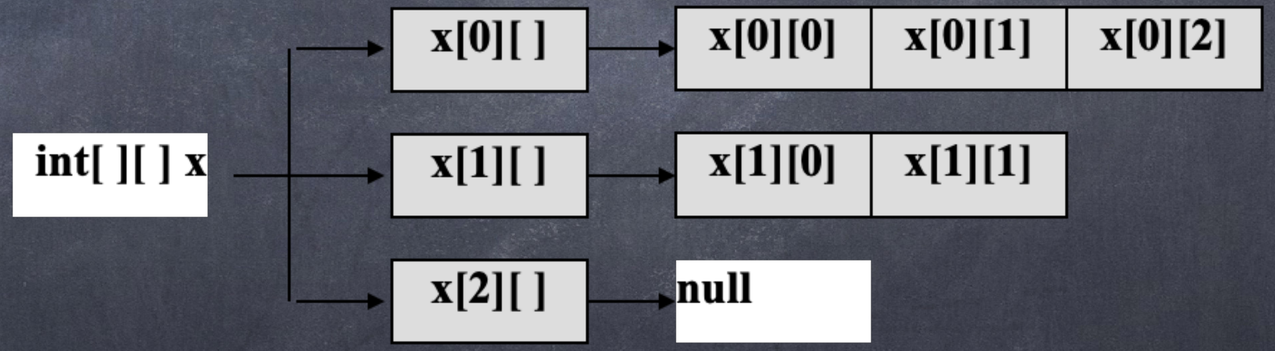
variable = new type [size1][size2];

type[][] variable= new type [size1][size2];

optional

int[][] a = new int[3][4]; //矩阵

int[][] a = new int[3][]; //不规则



初始化

int[][] a = {{11,22,33,44}, {66,77,88,99}};

字符串

- 字符串：一对双引号括起来的字符序列。

```
String variable;
variable = new String("XXX");
```

```
String variable = new String("XXX");
```

```
String variable = "XXX";
```

字符串常用方法

```
String a = "abc";
a.length();
String str1 = new String("abc");
String str2 = new String("abc");
System.out.println(str1.equals(str2));
System.out.println(str1 == str2);
System.out.println( a.substring(1)); // "bc"
System.out.println( a.substring(1,2)); // "b"
char c = a.charAt(0); // 'a'
int  first_index_of = a.indexOf("bc"); // 1
System.out.println( s.replace('a', 'd')); // "dbc"
String d = " abc ";
System.out.println(d.trim()); // “abc”
```

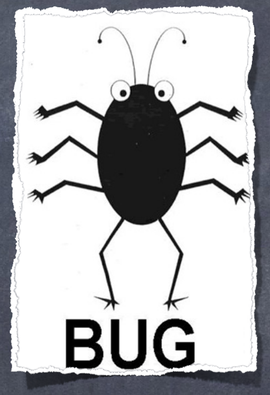
内容相等

对象引用相同（同一）

一个优化导致的"bug"

Java keeps a hash-indexed pool of string objects, particularly to save memory space for literal strings, like "abc" here.

```
String str1 = "abc";
String str2 = "abc";
System.out.println(str1.equals(str2));
System.out.println(str1  ==  str2);
```



尽量用equals去判断两个字符串

列表

- A list is an ordered sequence of objects

```
import java.util.List;
import java.util.ArrayList;

public class ListDemo {
  public static void main(String[] args) {
    List L = new ArrayList();
    L.add("a");
    L.add("b");
    L.add("c");
    System.out.println(L);
  }
}
```

Abstract template a.k.a. interface

Concrete implementation

```
L = []
L.append("a")
L.append("b")
L.append("c")
print(L)
```

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/List.html

列表

- 常用的列表：ArrayList、LinkedList

```
import java.util.List;
import java.util.LinkedList;

public class ListDemo {
  public static void main(String[] args) {
      List L = new LinkedList();
       L.add("a");
       L.add("b");
       L.add("c");
       System.out.println(L);
  }
}
```

映射（Map）

- A map is collection of key-value pairs

```
m = {}
m["cat"] = "meow"
m["dog"] = "woof"
sound = m["cat"]
```

```
import java.util.Map;
import java.util.TreeMap;

public class MapDemo {
  public static void main(String[] args) {
    Map<String, String> L = new TreeMap<>();
    L.put("dog", "woof");
    L.put("cat", "meow");
    String sound = L.get("cat");
  }
}
```

另一个常用的是HashMap

容器类的常见方法

- List常用方法

`lst.size();`	count the number of elements
`lst.add(e);`	append an element to the end
`lst.addAll(E);`	append a set(list) to the end
`let.remove(i);`	remove the *i*th element
`lst.isEmpty();`	test if the list is empty
`lst.contains(e);`	test if an element is in the list
`lst.get(i);`	get the *i*th element
`lst.sort(c)`	sort the list according to the comparator c
`lst.subList(f, t)`	get the sub list of lst from f(inclusive) to t(exclusive)
`lst.toArray(T[] a)`	return an array of the same elements of list

⚠️java中List所包含的元素类型是“对象”，不是原始类型。当用list.add(1)等诸如此类操作时，会自动转换为对应类的对象

容器类的常见方法

- Map常用方法

`map.size()`	returns the number of key-value mappings in the map
`map.put(key, val)`	add the mapping *key* → *val*
`map.get(key)`	get the value for a key
`map.containsKey(key)`	test whether the map has a key
`map.remove(key)`	delete a mapping
`map.repalce(key, val)`	replace the old value of key to val
`map.keySet()`	returns the set of all the keys
`map.entrySet()`	returns a set view of the mappings contained in this
`map.isEmpty()`	returns if the map is empty

⚠️同样，java中Map的key和value都是对象类型，不能是原始类型，一般为Integer、String、int[]等等

容器类的迭代

- 可以用Iterator或者for语句（for语句是用Iterator语句的语法糖）

```
List<String> cities         = new ArrayList<String>();
Set<Integer> numbers        = new HashSet<Integer>();
Map<String,Integer> turtles = new HashMap<String, Integer>();

for (String city : cities) {
    System.out.println(city);
}
for (int num : numbers) {
    System.out.println(num);
}
for (String key : turtles.keySet()) {
    System.out.println(key + ": " + turtles.get(key));
}
```

Java automatically converts between int and Integer

当然，你也可以用：

```
for (int i = 0; i < cities.size(); i++) {
    System.out.println(cities.get(i));
}
```

容器类的迭代

- 可以用Iterator或者for语句（for语句是用Iterator语句的语法糖）

java.util.Iterator

```
Iterator<String> i_ci = cities.iterator();
while( i_ci.hasNext()) {
    System.out.println(i_ci.next());
}
Iterator<Integer> i_nu = numbers.iterator();
while( i_nu.hasNext()) {
    System.out.println(i_nu.next());
}

for (Entry<String, Integer> entry : turtles.entrySet()) {
    System.out.println(entry.getKey() + ": " + entry.getValue());
}
```

Map也可以用entry来迭代

容器类的迭代

⚠Be careful not to mutate a collection while you're iterating over it.

```python
###python
numbers = [100,200,300]
for num in numbers:
    numbers.remove(num) # danger!!! mutates the list we're iterating over
print(numbers) # list should be empty here -- is it?
```

```java
/** java
*/
List<Integer> numbers = new ArrayList<Integer>(Arrays.asList(100, 200, 300));
for (Integer num : numbers){
    numbers.remove(num); // danger!!! mutates the list we're iterating over
}
System.out.println(numbers); /* list should be empty here -- is it? */
```

```
Exception in thread "main" java.util.ConcurrentModificationException
        at java.base/java.util.ArrayList$Itr.checkForComodification(ArrayList.java:1042)
        at java.base/java.util.ArrayList$Itr.next(ArrayList.java:996)
```

容器类的迭代

⚠Be careful not to mutate a collection while you're iterating over it.

```
###python
numbers = [100,200,300]
newList = []
print(numbers)
```

```
/**java
*/
List<Integer> numbers = new ArrayList<Integer>(Arrays.asList(100, 200, 300));
List<Integer> newList = new ArrayList<Integer>();
System.out.println(newList);
```

```
###python
numbers = [100,200,300]
newList = [x for x in numbers if x > 100]
print(numbers)
```

```
/**java
*/
List<Integer> numbers = new ArrayList<Integer>(Arrays.asList(100, 200, 300));
List<Integer> newList = new ArrayList<Integer>();
for (Integer num : numbers){
    if (num > 100)
        newList.add(num);
}
System.out.println(newList);
```

构建一个新的列表，尽量不要再迭代中对列表进行改变

容器类的迭代

- 一个不推荐的做法（Generally not safe!）

```
List<Integer> numbers = new ArrayList<Integer>(Arrays.asList(100, 200, 300));
Iterator<Integer> iter = numbers.iterator();
while(iter.hasNext()){
    Integer num = iter.next();
    if (num <= 100)
        iter.remove();
}
System.out.println(numbers);
```

What if there are other Iterators currently active over the same list? They won't all be informed!

what if the mutation of the list is something more complicated than just removing an element or appending an element - say, sorting the list into a different order?

面向对象

类

- 类由数据成员与函数成员封装而成。
  - Java语言把数据成员称为域变量（field varaible）、属性、成员变量等；
  - 而把函数成员称为成员方法，简称为方法。

类声明

The items between [and] are optional.

[modifiers] class *ClassName* [**extends** *SuperClassName*] [**implements** *InterfaceNames*] {

. . .

}

- 例子：

```
class NameOfClass {
    . . .
}
```

```
class ImaginaryNumber extends Number implements Arithmetic {
    . . .
}
```

```
public class NameOfClass extends SuperClassName {
    . . .
}
```

```
abstract class ImaginaryNumber implements Arithmetic, Collection{
    . . .
}
```

类声明

[modifiers] class *ClassName* [**extends** *SuperClassName*] [**implements** *InterfaceNames*] {

. . .

}

- modifiers 可以声明这个类是abstract, final还是public
- ClassName 是当前定义的类的名字
- SuperClassName 是当前定义类ClassName的超类（父类）（只能有一个，即Java是单继承）
- InterfaceNames 是一个接口的名字的集合（以逗号隔开），当前定义的类会实现这些接口

类声明

[modifiers] class *ClassName* [extends *SuperClassName*] [implements *InterfaceNames*] {

. . .

}

一般是为了安全方面的考虑，一些hackers可以通过定义一些给定类的子类（重写一些关键方法），然后替换原有的类，来获取一些关键隐私信息，或造成破坏

如果modifiers是final，那么其不可以有任何子类

如果modifiers是public，那么其可以被包外的类访问

如果modifiers为空（即没有修饰符），那么就是一个缺省类（the default, also known as package-private），只可以被当前包里的类所访问

如果modifiers是abstract，那么其是一个虚类（或抽象类），不可以实例化对象，是用来继承的。

包（package）结构

- 包为java的类提供了一种类似文件系统的组织形式，可以有效避免命名冲突。将相关的类组织到相关的包中是很好的模块化设计。
- 每一个Java的类都隶属于一个包
- 每个Java的源文件的第一句就是包声明

  package pkg1[.pkg2[.pkg3...]];

- 如果源文件没有包声明语句，那么其隶属于一个无名的默认包中

包结构

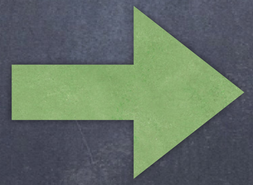
- 例子

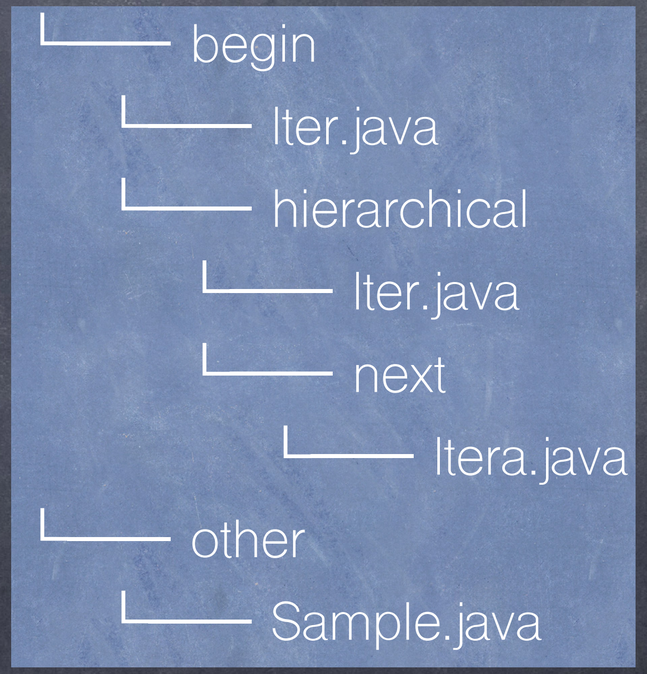
```
package begin;
public class Itera {
}
```

```
package begin.hierarchical;
public class Itera {
}
```

```
package begin.hierarchical.next;
public class Itera {
}
```

```
package other;
public class Sample {
}
```





具体解释执行class文件时，需要添加上包，比如： java begin.hierarchical.next.Itera

包结构

- 使用时，利用import语句可以将包的具体类引入

```
package begin;
public class Itera {

}
```

```
package other;
public class Sample {

}
```

```
import begin.Itera;
import other.*;
public class ImportSample {
    Itera i;
    Sample s;
}
```

等价于

```
public class ImportSample {
    begin.Itera i;
    other.Sample s;
}
```

包结构

- 例子：使用built-in的包

```
import java.util.Scanner;

class MyClass {
  public static void main(String[] args) {
    Scanner myObj = new Scanner(System.in);
    System.out.println("Enter username");

    String userName = myObj.nextLine();
    System.out.println("Username is: " + userName);
  }
}
```

```
import java.util.Scanner;
import java.io.File;
import java.io.PrintWriter;

class MyClass {
  public static void main(String[] args) {
    try{
        String fileInput = "file.txt";
        String fileOutput = "out.txt";
        Scanner myObj = new Scanner(new File(fileInput));
        PrintWriter fileOut = new PrintWriter(new File(fileOutput));
        while(myObj.hasNextLine()){
            String line = myObj.nextLine();
            fileOut.println(line);
        }
        myObj.close();
        fileOut.close();
    }
    catch(Exception e){
        throw new RuntimeException(e);
    }
  }
}
```

Python的包结构

```
sound/
    __init__.py
    formats/
            __init__.py
            wavread.py
            aiffread.py
            ...
    effects/
            __init__.py
            echo.py
            surround.py
            ...
    filters/
            __init__.py
            equalizer.py
            vocoder.py
            ...
```

A directory must contain a file named `__init__.py` in order for Python to consider it as a package.

This file can be left empty can also be placed by some the initialization code (These code will be implicitly executed when the package is imported)

- 用法：

```
from sound.formats import aiffread

aiffread.XXX()
```

```
import sound.formats

sound.aiffread.XXX()
```

成员变量

- 类的成员变量描述了该类的对象的内部信息，一个成员变量可以是简单变量，也可以是对象、数组等其他结构型数据。成员变量的格式如下：

  [accessSpecifier] [final] type varaibleName [=initial_value];

  其中accessSpecifier 一般为public、protected、private、和缺省（package-private）

  当final被声明时，该成员变量是一个常量（不可改变）

- 在定义类的成员变量时，可以同时赋初值，但对成员变量的操作只能放在方法中。

访问控制（ public, protected, default, private ）

Modifier	Class 当前类	Package 当前类所在的包	Subclass 当前类的子类	All the others 所有其他
public	✓	✓	✓	✓
protected	✓	✓	✓	✗
(Default)	✓	✓	✗	✗
private	✓	✗	✗	✗

访问控制例子

```
class A {
  public int x;
  public void print() { ... }
}
class B {
  void test() {
  A a = new A();
  a.x = 100;
  a.print();
  }
}
```

```
package  abc;
class A {
  public int x;
  public void print() { ... }
}

package xyz;
import abc.A;
class B {
  void test() {
  A a = new A();
  a.x = 100;
  a.print();
  }
}
```

Public允许全权访问，无任何限制，访问成员变量（或者成员方法），需要先实例化对象。

访问控制例子

```
class A {
  private int x;
  private void print() { ... }
}
class B {
  void test() {
  A a = new A();
  a.x = 100;
  a.print();
  }
}
```

编译不通过!

```
x has private access in A
                a.x = 100;
                  ^
print() has private access in A
                a.print();
                  ^
2 errors
```

```
class A {
  private int x;
  private void print() { ... }
  boolean comare(A other){
     return this.x > other.x
  }
}
```

The same class can access!

访问控制例子

```
//same package
class A {
  protected int x;
  protected void print() { ... }
}
//same package
class B {
  void test() {
  A a = new A();
  a.x = 100;
  a.print();
  }
}
```

编译不通过！

正确

```
package  abc;
class A {
  protected int x;
  protected void print() { ... }
}
package xyz;
import abc.A;
class B extends A {
  void test() {
  A a = new A();
  B b = new B();
  a.x = 100;
  a.print();
  b.x = 100;
  b.print();
  }
}
```

Protected 允许类本身、其子类（⚠️有一定限制）以及同一个包中所有类访问

访问控制例子

```
package abc;
class A {
  int x;
  void print() { ... }
}
package abc;
class B {
  void test() {
  A a = new A();
  a.x = 100;
  a.print();
  }
}
```

缺省的变量允许类本身以及同一个包（其他包则不行，即使是其子类）中所有类访问

访问控制

- 一般而言，对于"敏感"的数据应该设为私有变量，从而对外界进行隔离
- 通过公共方法（get和set）来进行访问和修改这些变量

这就是JAVA的封装性

封装性例子

```
public class Person {
  private String name; // private = restricted access

  // Getter
  public String getName() {
    return name;
  }

  // Setter
  public void setName(String newName) {
    this.name = newName;
  }
}
```

方法名一般以get开头，接相应的变量名，其中变量名的第一个字母大写，这个方法其实就是选择子，或则访问子，也叫观察子（observer）

变异子方法名一般以set开头，接相应的变量名，这个方法其实就是变异子

方法

- 类的成员方法是类的对象的一些行为的描述。成员方法的格式如下：

[accessSpecifier] [final] returnType methodName ([paramlist]) [throws exceptionsList]

同样，其中accessSpecifier 一般为public、protected、private、和缺省（package-private），刻画了和成员变量一样的访问规则。

当final被声明时，该成员方法不可被重写（overriding）。

throws exceptionsList 用来声明方法可能会在运行时出现哪些异常（如IO异常）

javadoc:
用来提供了类或者方法的一些说明
如参数、返回值，
这是java注释的一种。
其他类型的注释：
1. // 单行注释
2. /* 多行注释 */

例子

```
public class Hailstone {
    /**
     * Compute a hailstone sequence.
     * @param n  starting number for sequence; assumes n > 0.
     * @return hailstone sequence starting with n and ending with 1.
     */
    public List<Integer> hailstoneSequence(int n) {
        List<Integer> list = new ArrayList<Integer>();
        while (n != 1) {
            list.add(n);
            if (n % 2 == 0) {
                n = n / 2;
            } else {
                n = 3 * n + 1;
            }
        }
        list.add(n);
        return list;
    }
}
```

method signature

method name

parameter declaration

return type

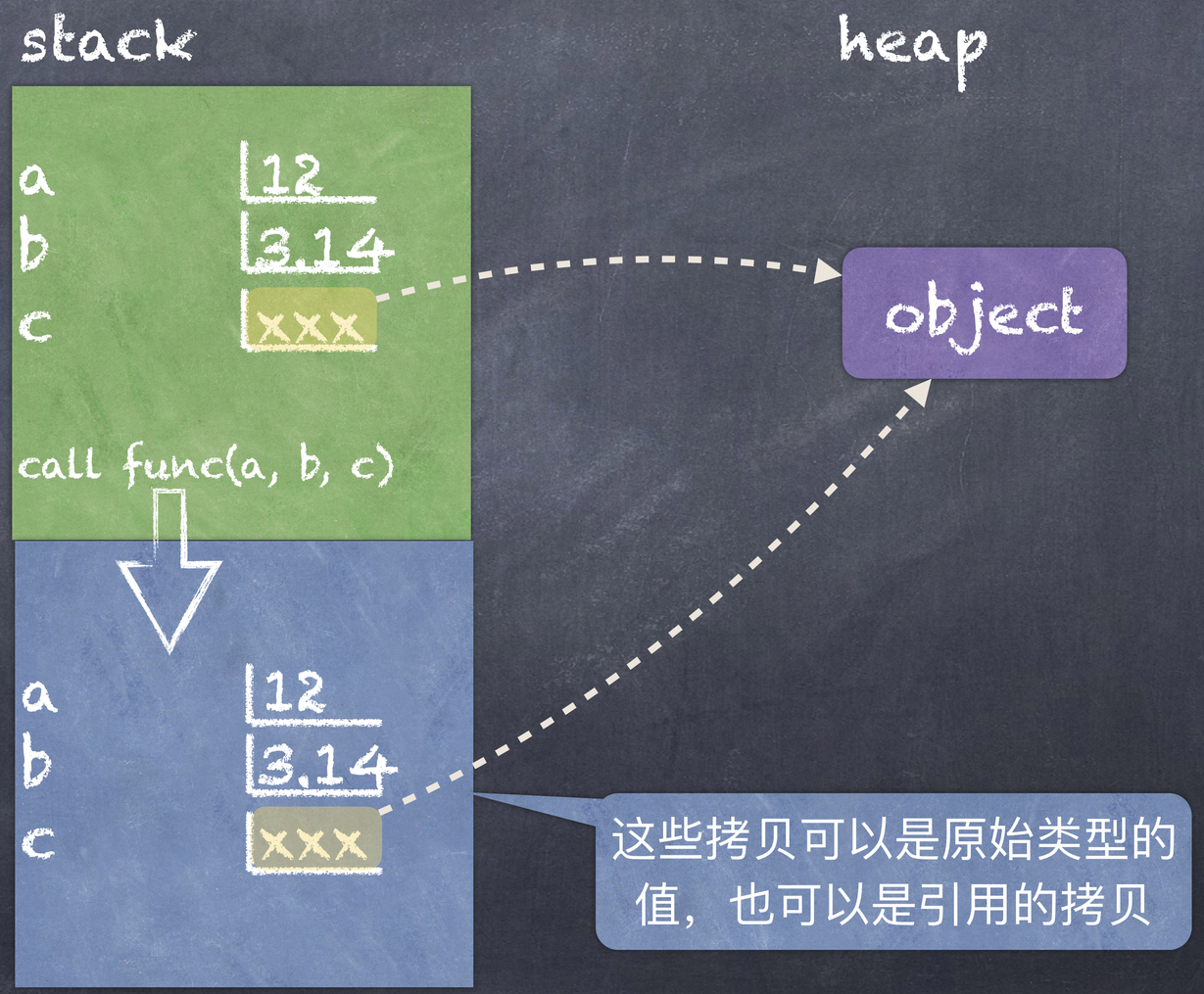
method body

return statement

为什么函数签名不包含返回值?
避免调用时的不确定性

值传递（Pass by value）

- 在java中，方法的实参是通过值传递的。当方法调用时，实参的值的拷贝会赋给方法中的参数变量.
- 方法中可以对该拷贝值做出改变，但该改变不会影响到原来的变量。



```
public class PassBy {
  public void tryPrimitives(int i, double f, char c, boolean test)
  {
    i += 10;     //This is legal, but the new values
    c = 'z';     //won't be seen outside tryPrimitives.
    f -= 3.0;
    if(test)
      test = false;
    else
      test = true;
  }

  public static void main(String[] args) {
        PassBy p = new PassBy();
        int ii = 1;
        double ff = 1.0;
        char cc = 'a';
        boolean bb = false;
        p.tryPrimitives(ii, ff, cc, bb);
        System.out.println("ii = " + ii + ", ff = " + ff +
                           ", cc = " + cc + ", bb = " + bb);

  }
}
```

原始数据的值的拷贝传进被调用方法内（callee），callee对其改变不会影响调用方法（caller）中变量的值

```
class Record
{
  int num;
  String name;
}

public class PassBy {
  public void tryObject(Record r)
  {
    r = new Record();
    r.num = 100;
    r.name = "Fred";
  }
  public static void main(String[] args) {
        PassBy p = new PassBy();
        Record id = new Record();
        id.num = 2;
        id.name = "Barney";
        p.tryObject(id);
        System.out.println(id.name + "  " + id.num);
  }
}
```

对象的引用的拷贝（可以认为就是该对象的内存中地址的拷贝）传进被调用方法内（callee），callee对其重新赋值不会影响调用方法（caller）中变量的值

```
class Record
{
  int num;
  String name;
}

public class PassBy {
  public void tryObject(Record r)
  {
    r.num = 100;
    r.name = "Fred";
  }
  public static void main(String[] args) {
        PassBy p = new PassBy();
        Record id = new Record();
        id.num = 2;
        id.name = "Barney";
        p.tryObject(id);
        System.out.println(id.name + "  " + id.num);
  }
}
```

但是通过传递的引用可以更改其指向的对象的内容，该改变会反映到原来的引用上

It is often not good programming style to change the values of instance variables outside the object.

Normally, the object should have a method (mutator) to set the values of its instance variables !

参数传递

- python中的方法中参数是什么传递？

官方文档：赋值传递（Pass by assignment）

实际上，和Java的值传递语义一致，
只不过python中没有原始类型而已

构造方法

- 对象的实例化通过构造方法（即构造子Constructor）来实现（当使用new语句时自动调用，不能显式地调用）
- 构造方法的名字与类名相同
- 构造方法没有返回值
- 构造方法可以有多个，构成方法可以重载
- 当没有声明构造函数时，默认含有一个无参数的构造函数，但当显示地声明了构造函数之后，该默认的构造函数就不存在。

构造方法例子

```java
class Main {

  int a;
  boolean b;
  public static void main(String[] args) {

    // A default constructor is called
    Main obj = new Main();

    System.out.println("Default Value:");
    System.out.println("a = " + obj.a);
    System.out.println("b = " + obj.b);
  }
}
```

等价于

```java
class Main {

  int a;
  boolean b;

  Main() {
    a = 0;
    b = false;
  }

  public static void main(String[] args) {
    // call the constructor
    Main obj = new Main();

    System.out.println("Default Value:");
    System.out.println("a = " + obj.a);
    System.out.println("b = " + obj.b);
  }
}
```

与类名相同，没有返回值

通过new来隐式地调用构造子

构造方法例子

```java
class Main {

  String language;

  // constructor with no parameter
  Main() {
    this.language = "Java";
  }

  // constructor with a single parameter
  Main(String language) {
    this.language = language;
  }

  public void getName() {
    System.out.println("Programming Langauage: " + this.language);
  }

  public static void main(String[] args) {

    // call constructor with no parameter
    Main obj1 = new Main();

    // call constructor with a single parameter
    Main obj2 = new Main("Python");

    obj1.getName();
    obj2.getName();
  }
}
```

多个构造方法即重载了构造方法）

会根据参数形式来选择具体的构造方法

```java
class Main {

  String language;

  Main(String language) {
    this.language = language;
  }

  public void getName() {
    System.out.println("Programming Langauage: " + this.language);
  }

  public static void main(String[] args) {

    Main obj1 = new Main();
    obj1.getName();
  }
}
```

⚠error! 声明了构造方法后，默认的无参数构造子就不存在了

构造函数调用其他构造函数

```
int sum;
// first constructor
Main() {
  // calling the second constructor
  this(5, 2);
}
// second constructor
Main(int arg1, int arg2) {
  // add two value
  this.sum = arg1 + arg2;
}
void display() {
  System.out.println("Sum is: " + sum);
}
// main class
public static void main(String[] args) {
  // call the first constructor
  Main obj = new Main();
  // call display method
  obj.display();
}
}
```

利用this(...)语句，⚠注意这里和调用普通函数或者数据属性的dot expression不同（即this.XXX）

隐式地调用了第二个构造函数

标识符（Identifier）名字规则和约定

- 标识符：标识类、变量、常量和方法的名字，由字母（A~Z、a~z）、特殊符号（$、_）和数字（0~9）构成，区分大小写，且名字的第一个符号不能为数字，标识符不能是Java关键字
  - 类：一般首字母大写
  - 变量：一般小写开始（中间不同的单词的开头大写，如getCurrentValue）
  - 常量：一般全部大写
  - 方法：和变量一致

static关键字

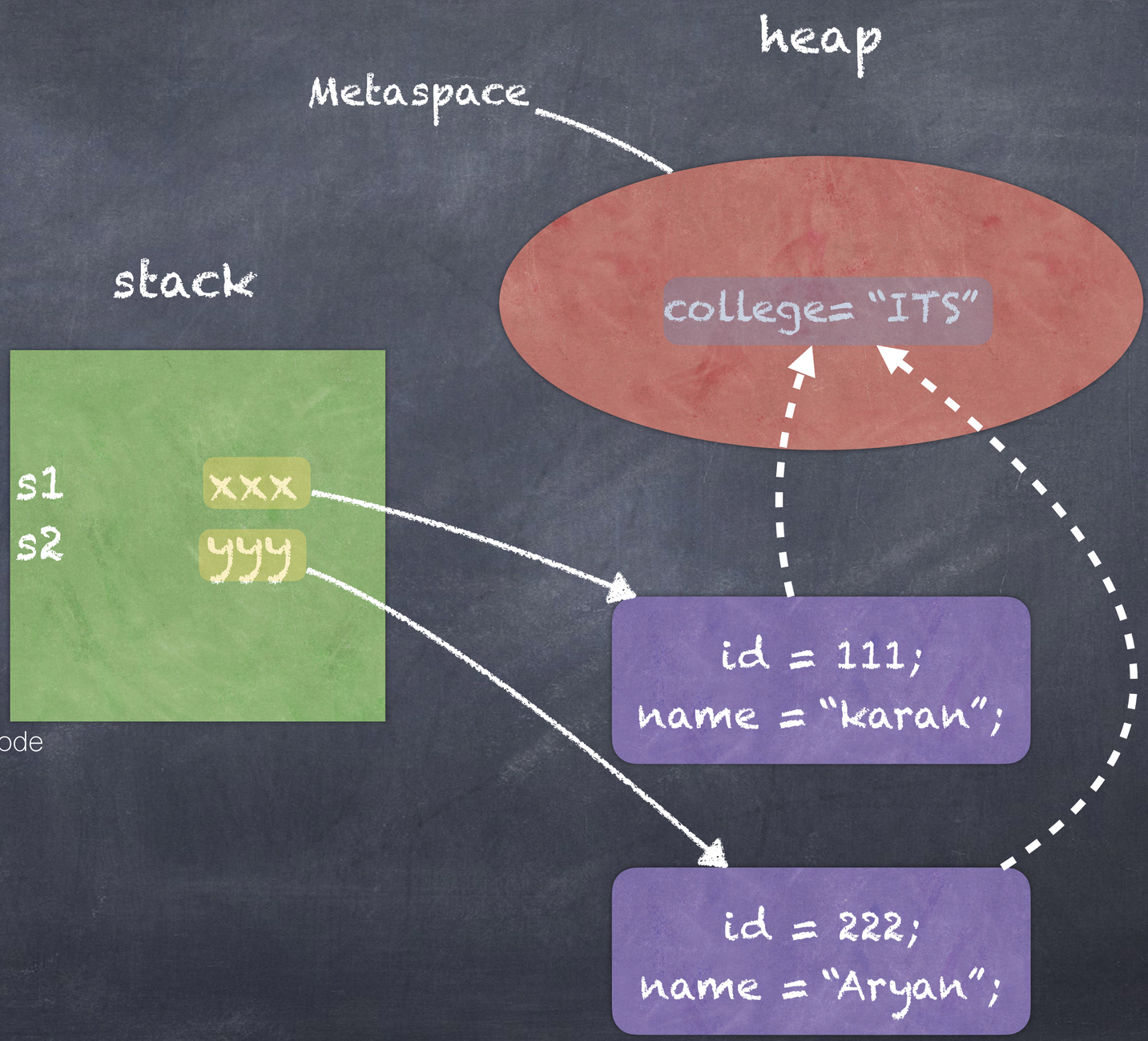
- static是java中用来表达隶属于"类"本身，而不隶属于类的对象的一个关键字
- static可以修饰：
  - 变量（该变量即为类变量或静态变量）
  - 方法（该方法即为类方法或静态方法）
  - 语句块（用来初始化类变量）

静态变量

- 静态变量用来表达所有对象的共有属性
- 静态变量在JVM对类加载之后就得到了相应的内存，其只有一份，不会随着对象的创建增多。

```
class Student{
    int rollno;//instance variable
    String name;
    static String college ="ITS";//static variable
    //constructor
    Student(int r, String n){
        rollno = r;
        name = n;
    }
    //method to display the values
    void display (){System.out.println(rollno+" "+name+" "+college);}
}

public class TestStaticVariable1{
    public static void main(String args[]){
    Student s1 = new Student(111,"Karan");
    Student s2 = new Student(222,"Aryan");
    //we can change the college of all objects by the single line of code
    //Student.college="BBDIT";
    s1.display();
    s2.display();
  }
}
```



静态方法

- 静态方法隶属于类，可以直接用类名来调用，而无需实例化一个对象来调用
- 静态方法可以访问和更改静态变量，但不能直接访问非静态的成员（变量和方法）
- 在静态方法中，this（表示当前对象的引用）和super（表示当前对象的相应的父类对象引用）无法使用

靜態方法

```
class Calculate{
  static int cube(int x){
  return x*x*x;
  }

  public static void main(String args[]){
  int result=Calculate.cube(5);
  System.out.println(result);
  }
}
```

```
class A{
 int a=40;//non static

 public static void main(String args[]){
  System.out.println(a);
 }
}
```

Compile Time Error

静态语句块

- 静态语句块用来初始化静态变量
- 静态语句块不能访问非静态成员
- 静态语句块在 main 方法之前调用

```
class A1{
    static int a;
    static int b;
    static{
        a = 1;
        b = 2;
    }
    public static void main(String args[]){
        System.out.println("hello");
    }
}
```

```
class A2{
  static{System.out.println("static block is invoked");}
  public static void main(String args[]){
   System.out.println("Hello main");
  }
```

Output: static block is invoked
Hello main

继承

继承

- Java通过关键字extends来表达继承关系

```
class SubClass extends SuperClassName {
  ...
}
```

- 子类可调用父类的方法和变量（所以继承破坏了封装性），子类可增加父类中没有的方法和变量
- Java只支持单继承，即只有一个"直接"父类。父类的父类也是该子类的父类，但不是直接父类。Java中所有类都是java.lang.Object的子类。

例子

```
class Vehicle {
  String brand;
  public void setB(String s) { brand = s; }
  public void showB() { System.out.println(brand); }
}
class Bus extends Vehicle {
  int gas;
  public void setG(int g) { gas = g; }
  public void showG() { System.out.println(gas); }
  public static void main(String[] args){
      Bus b = new Bus();
      b.setB("audi");
      b.setG(100);
      b.showB();
      b.showG();
  }
}
```

继承了相应的属性，所以可以直接使用

重新定义变量例子

```
class A {
    private int m = 0;
    int i=256, j =64;
    static int k = 32;
    final float e = 2.718f;
}
public class B extends A {
    public char j='x';
    final double k =5;
    static int e =321;
    void show() { System.out.println(i + " " + j + " " +  k + " " + e );  }
    void showA() { System.out.println(super.j + " " + super.k + " " + super.e); }
    public static void main(String[] args) {
        B b = new B();
        b.show();
        b.showA();
    }
}
```

Private不能继承

重新定义父类中的变量，父类中相应的变量被隐藏（hidden）

可以通过super关键词来访问这些隐藏的变量，当然static的可以通过类名来访问

重新定义方法例子

```
class Vehicle {
    String brand;
    public void setB(String s) { brand = s; }
    public void showB() { System.out.println(brand); }
}

class Bus extends Vehicle {
    public void setB(String g) {System.out.println("sub set"); brand = g; }
    public void showB() {System.out.println("sub show");  System.out.println(brand); }
    public static void main(String[] args){
        Bus b = new Bus();
        b.setB("audi");
        b.showB();
        }
}
```

签名相同的方法，重新定义就是方法重写（Overriding）

隐藏和重写（Override）的区别

- 重写对应运行时，Java会在运行时判断哪个方法会被调用（It is for non-static methods.）
- 隐藏对应编译时，Java在编译阶段就已经确定好了调用的对象（即静态和实例变量、静态方法）

隐藏和重写的区别

```
public class Animal {
    public static void something() {
        System.out.println("animal.something");
    }
    public void eat() {
        System.out.println("animal.eat");
    }
    public Animal() {
    }
}
```

```
public class Dog extends Animal {
    public static void something() {
        // This method merely hides Animal.something(),
        // but does not override it
        System.out.println("dog.something");
    }
    public void eat() {
        // This method overrides eat(),
        // and will affect calls to eat()
        System.out.println("dog.eat");
    }
    public Dog() {
    }
    public static void main(String[] args) {
        Animal  animal = new Dog();
        animal.something();
        animal.eat();
    }
}
```

父子类中签名相同的函数，Static 和non-static必须一致

Upcasting

子类型多态

Output:

animal.something

dog.eat

父类的构造方法

- 在Java中，任何类的构造方法，第一行语句必须是调用父类的构造方法。
- 如果没有明确地调用父类的构造方法，编译器会帮我们自动加一句`super();`
- 构造方法无法继承，它隶属于特定的类。因此，如果即使类没有写构造方法，那么其也会有一个默认的构造方法，而不是继承于父类的构造方法

例子

```
class Art {
    Art() {
    System.out.println("Art Constructor");
    }
}
class Drawing extends Art {
    Drawing() {
    System.out.println("Drawing Constructor");
    }
}
public class Cartoon extends Drawing {
    Cartoon() {
        super();
        System.out.println("Cartoon Constructor");
    }
    public static void main(String args[]) {
        Cartoon c = new Cartoon();
    }
}
```

隐式地调用super()

Output:

Art Constructor
Drawing Constructor
Cartoon Constructor

子类的构造法的第一句必须调用父类的构造方法

例子

```
class Game {
    Game(int i) {
        System.out.println("Game Constructor");
    }
}
class BoardGame extends Game {
    BoardGame(int i) {
        System.out.println("BoardGame Constructor");
    }
}
public class Chess extends BoardGame {
    Chess() {
        super(3);
            System.out.println("Cartoon Constructor");
    }
    public static void main(String args[]) {
        Chess c = new Chess();
    }
}
```

Compile error! Why?

类成员访问修饰符与继承的关系

- 私有的（private）类成员不能被子类继承
- 构造方法不被继承
- 公共的（public）和保护性的（protected）类成员能被子类继承，且子类和父类可以属于不同的包
- 无修饰的父类成员，仅在同一个包中才能被子类继承

类成员访问修饰符与继承的关系

- 一些注意点⚠️：
  - 重写可继承的函数时，其访问权限不能比父类中被重写的方法访问权限更低
    - 比如，如果父类的一个方法访问修饰符是protected，那么子类重写时可变为public、也可不变，但不能为private 或缺省。
  - 重写可继承的函数时，返回值类型如果是原始类型，必须一致，如果是对象类型，那么可以不一致（但必须是原函数返回值的子类）
  - 对于隐藏而言，则没有限制（你可以认为隐藏本质上其实和原来的是两种不同的成员，因为在编译阶段就已经分开了）

函数重载（Overloading）

- 方法的重载是指方法带有不同的参数，但使用相同的名字。
  - 一般方法的参数不同则表示实现不同的功能，但功能相似。
- 所谓参数不同是指：参数个数不同、参数类型不同、参数的顺序不同。
- 函数重载即特设多态（Ad hoc polymorphism）

例子

```
double sum(double a, double b){
    …
}

double sum(double a, double b, double c){
    …
}

float sum(float a, float b){
    …
}
```

Same method name

different types or numbers of parameters

⚠️重载是定义多个函数，这些函数名一样，参数不同，而重写(也叫覆盖)是重新定义父类中签名相同的函数。 重载是特设多态，重写是子类型多态。重载在函数调用时所调用的具体函数（函数绑定）在编译时确定（early binding），重写的函数绑定在运行时（late binding）。

instanceof

- 子类的每个对象也是父类的对象
- 可以直接用子类对象赋给一个父类变量

```
SuperClass variable = new SubClass();
```

- 这种转化叫作：向上转型（upcasting）

instanceof

- 但父类对象不能直接赋给子类

```
public class Animal {
   ...
}

public class Dog extends Animal {
      ...
      public static void main(String[] args) {
         Dog   animal = new Animal();
      }
   }
```

incompatible types

向下转型

downcasting

```
public class Animal {
   ...
}

public class Dog extends Animal {
      ...
      public static void main(String[] args) {
         Dog   animal = (Dog) new Animal();
      }
   }
```

explicit conversion

class Animal cannot be cast to class Dog

instanceof

```
public class Dog extends Animal {
    ...
    public static void main(String[] args) {
      Animal animal = new Dog();
      Dog  dog = (Dog) animal;
    }
}
```

Correct

- 想要安全地进行向下转型时，一般需要用运算符instanceof来进行判断

```
if (animal instanceof Dog)
    Dog  dog = (Dog) animal;
```

dog instanceof Animal 也为true， 即子类的对象即是其父类的对象

抽象类

抽象类（abstract class）

- 如果一个类中没有包含足够的信息来描绘一个具体的对象（方法没实现），这样的类就是抽象类。
- 并不能直接由抽象类创建对象，只能通过抽象类派生出新的子类，再由其子类来创建对象。
  - 也就是说，抽象类就是不能用new运算符来创建实例对象的类

抽象类

- 抽象类的定义：

```
abstract class classname
{
    ... // other declarations
    abstract  type methodname (parameters);
}
```

抽象方法，在抽象方法里，不能定义方法体。只需声明不需实现

继承抽象类

- 抽象类的子类可以是抽象类
- 如果子类不是抽象类，那么其必须实现父类中的所有抽象方法，即抽象方法必须被子类的方法所覆盖。
  - 抽象类某种意义上限制了子类的行为，因此，抽象类有点类似"模板"的作用，其目的是根据它的格式来创建和修改新的类。

例子

```
abstract public class Shape {
    protected String name;
    public Shape(String xm){
        name = xm;
        System.out.println("名称 ： " + name);
    }
    abstract public double getArea();  //面积
    abstract public double getLength(); //周长
}
```

```
public class Circle extends Shape{
    private final static double PI = 3.14;
    private double radius;

    public Circle(String xm, double r) {
        super(xm);
        this.radius = r;
    }

    public double getArea() {
        return PI*radius*radius;
    }

    public double getLength() {
        return 2*PI*radius;
    }
}
```

必须实现，否则Circle必须要声明为Abstract 类

一些抽象类注意点

- 由于抽象类是需要被继承的，所以abstract类不能用final来修饰。也就说，一个类不能既是最终类，又是抽象类，即关键字abstract与final不能合用。
- 抽象类中不一定包含抽象方法，但包含抽象方法的类一定要声明为抽象类。

接口

接口（Interface）

- 抽象类可以作为子类的模版。但由于Java的单继承特性，使得一个子类只有一个直接父类，无法具备多个模版
- 此外，继承也会有很多问题，如果滥用继承，会导致继承层次过深。而复杂的继承关系会导致维护性降低。
- Java中的接口提供了多个模版的可能
  - 可以看成抽象方法的集合。一个类可以实现多个接口

接口

```
[public] interface interfaceName [extends List-of-super-interfaceNames]
  {
     [public][static][final] type variableName = constantValue;
     ......
     [public][abstract] type methodName (parameterList);
     ......
    [public] static type methodName(parameterList)
    {
        //method body
    }
    ......
   [public] default type methodName(parameterList)
    {
        //method body
   }
   ......
}
```

接口的一些注意点

- 接口不能用于实例化对象。
- 接口没有构造方法。
- 接口只有三种类型的方法（抽象方法（缺省）、静态方法、default 方法）。
- 接口的变量都是 static final 修饰的（静态常量），缺省的也是静态常量。
- 接口的成员可访问性都是public（缺省也是public）。

例子

```
public interface IShape {
    public static final double PI = 3.14;
    double getArea();
    public abstract double getLength();
    public static void showPI(){
        System.out.println(PI);
    }
    public default void getInfo(){
        System.out.println("这是一个图形");
    }
}
```

接口的实现与引用

- 接口的实现：利用接口的特性来建造类的过程，类似继承，但是关键词为implements
- 一个类可以实现多个接口，每个接口用逗号隔开

```
class  className  implements  List-of-interfaceNames
{
    ......
}
```

接口的实现与引用

- 一个类必须实现接口的所有抽象方法（除非该类是抽象类）
- 类实现的所有抽象方法都必须声明public
- 接口可以作为一种引用类型使用，可以声明接口类型的变量或数组，并用它来访问实现该接口的类的对象。

why

例子

```
public interface IShape {
    public static final double PI = 3.14;
    double getArea();
    public abstract double getLength();
    public static void showPI(){
        System.out.println(PI);
    }
    public default void getInfo(){
        System.out.println("这是一个图形");
    }
}
```

```
public class CireCleForI implements IShape{
    double radius;
    public CireCleForI(double r){
        this.radius = r;
    }
    public double getArea() {
        return PI*radius*radius;
    }
    public double getLength() {
        return 2*PI*radius;
    }
}
```

接口作为变量类型

```
public class TestIShape {

    public static void main(String[] args){
        IShape cir = new CireCleForI(2.2);
        System.out.println("面积 :  "+ cir.getArea());
        System.out.println("周长 :  "+ cir.getLength());
        cir.getInfo();
        IShape.showPI();
    }

}
```

对象调用方法（默认方法、抽象方法）

静态方法必须接口名访问

接口继承

- 接口可通过extends关键字声明该新接口是某个已存在的父接口的子接口，它将继承父接口的所有变量与方法（静态方法除外，静态方法只能通过接口名来访问）。
- 接口支持多继承（一个接口可以继承多个接口，接口不可以继承类，多个父接口用逗号隔开）。
- 如果接口中定义了与父接口同名的常量或相同的方法，则父接口中的常量和静态方法被隐藏，默认方法和抽象方法被重写。

例子

```
public interface Face1 {
    static final double PI = 3.14;
    abstract double area();
}
public interface Face2 {
    default void setColor(String c){
        System.out.println("颜色是 ：  "+ c);
    }
    abstract void volume();
}
public interface Face3 extends Face1, Face2 {
    static final double PI = 3.1415;
    public default void setColor(String c){
        System.out.println("颜色是 ：  "+ c + "3");
    }
}
```

多继承

隐藏父接口变量

重写默认方法

接口实现混入（Mixin）

- 利用default方法实现混入

```
public interface Flyable {
  default void fly() {
    System.out.println("I can fly!");
  }
}

public interface Swimmable {
  default void swim() {
    System.out.println("I can swim!");
  }
}
```

Cannot instantiate

It is able to fly and swim

```
public class Duck implements Flyable, Swimmable {

}
```

接口多继承中的名字冲突问题

- 接口的多重继承中可能存在常量名或方法名重复的问题，即名字冲突问题
- 对于常量，若名称不冲突，子接口可以继承多个父接口中的常量，但如果多个父接口中有同名的常量，则必须通过 接口名.常量名 区分。
- 对于多个父接口中存在同名的方法包含默认方法时，也会发生命名冲突，这时不能通过 接口名.默认方法名 来解决
  - 必须通过在当前类中定义一个同名的方法才行

例子

```
public interface Face1 {
    static final double PI = 3.14;
    abstract double area();
    default void setColor(String c) {}
}

public interface Face2 {
    static final double PI = 3.1415;
    default void setColor(String c){
        System.out.println("颜色是 ：  "+ c);
    }
    abstract void volume();
}
```

Face3 inherits unrelated defaults for setColor(String) from types Face1 and Face2

```
/** error */
public interface Face3 extends Face1, Face2 {
}
```

重新定义该方法，可以是默认方法，也可以是抽象方法

```
/** correct */
public interface Face3 extends Face1, Face2 {
    default void setColor(String c){
        System.out.println(Face2.PI);
        System.out.println("颜色是 ：  "+ c);
    }
}
```

变量名只要加接口名即可

```
/** correct */
public interface Face3 extends Face1, Face2 {
    default void setColor(String c){
        Face2.super.setColor(c);
    }
}
```

通过 接口名.super.方法名 访问父接口的方法

如果发生了继承类和实现接口的命名冲突

- 那么"类"优先，继承的父方法中的同名方法

```
public class SimpleSetColor {
    protected double x;
    public void setColor(String c){
        System.out.println("颜色是Simple：  "+ c);
    }
    protected String set(){return "0";}
}

public interface Face2 {
    static final double PI = 3.12;
    default void setColor(String c){
        System.out.println("颜色是：  "+ c);
        System.out.println(this.PI);
    }
    abstract void volume();
}
```

```
public class Cylinder extends SimpleSetColor implements Face2 {

    private double radius;
    private int height;
    protected String color;
    private boolean x;

    public Cylinder( double r,int  h){
        this.radius = r;
        this.height = h;
    }
    public double area() {
        return Face1.PI*radius*radius;
    }
    public void volume() {
        System.out.println("体积为：  " + area()*height);
    }
    public static void main(String[] args){
        Cylinder c = new Cylinder(3.0, 2);
        c.setColor("红色");
        c.volume();
    }
    public String set(){return "1.0";}
}
```

这里会调用SimpleSetColor的方法

泛型

泛型

- 作为一种静态类型语言，Java对参数需要固定类型，虽然增加了安全性，但却失去了灵活性
  - 当然可以通过重载来增加函数对与不同参数的支持，但是这样会增加很多冗余代码，因为很多时候我们对于不同类型的参数处理逻辑是一致的，这就是"参数多态"
- 通过引入"泛型"（generics）的概念，Java可以将类型也作为一种参数，从而实现参数多态

泛型的语法

注意这里T、E可以是其他符号，代表类型参数。

- 泛型类的定义：[modifiers] class className <T, E, ...> {...}
- 泛型接口的定义：[modifiers] interface interfaceName <T, E, ...> {}
- 泛型方法的定义：[modifiers] <T, E, ... > returnType *functionName* (parameterList) {...}

例子

形式类型参数

```
public class GenericsSample<T, E, F> {
    private T obj;
    protected E something;
    protected F otherthing;
    public T getObj(){
        return obj;
    }
    public void set(E something, F otherthing){
        this.something = something;
        this.otherthing = otherthing;
    }
    public void setObj(T obj){
        this.obj = obj;
    }
    public static void main(String[] args){
        GenericsSample<String, Integer, Double> t1 = new GenericsSample<>();
        GenericsSample<Integer, Long, List> t2 = new GenericsSample<Integer, Long, List>();
        t1.setObj("Tom");
        t2.set(12L, new ArrayList<Integer>(Arrays.asList(100, 200, 300)));
    }
}
```

这些形式类型和普通类型用法一样

实际类型参数，必须为对象类型

构造对象时类型可显式声明，也可缺省

例子

```
public interface GenericsInterface<E> {
    E getAge();
    void printAge(E e);
}


public class Sample_implments implements GenericsInterface<Integer> {
    public Integer getAge() {
        return 0;
    }
    public void printAge(Integer o) {
        System.out.println("年龄："+o);
    }
}
```

例子

在返回类型前

```
public class GenericsMethod {
    public static <E> void display(E[] list){
        for(E e : list){
            System.out.print(e + " ");
        }
        System.out.println();
    }
    public <E, V> void display(E[] list, V[] list2){
        for(E e : list){
            System.out.print(e + " ");
        }
        System.out.println();

        for(V e : list2){
            System.out.print(e + " ");
        }
        System.out.println();
    }
    public static void main(String[] args){
        Integer[] num = {1, 2 ,3, 4, 5};
        String[] strs = {"赤", "橙","黄", "绿", "青", "蓝","紫"};
        GenericsMethod.display(num);
        GenericsMethod.display(strs);
        GenericsMethod app = new GenericsMethod();
        app.display(num, strs);
    }
}
```

泛型的类型可以做一些限制

- 语法： <T extends anyClass>

就可以可以限制T的实际参数必须是anyClass的子类，或者实现了anyClass接口的类， 其中anyClass无论是类或者接口，都必须用extends关键字。这里anyClass也被称为该形式类型的限定类型（ bounding type ）

```
class GeneralType <T extends Number> {
    T obj;
    public void GeneralType (T obj){
        this.obj = obj;
    }
    public T getObj(){
        return obj;
    }
}
```

```
public class Test {
    public static void main(String[] args){
        GeneralType<Integer> num = new GeneralType<Integer>(5);
        System.out.println("给出的参数是： "+ num.getObj());
    }
}
```

```
public class Test {
    public static void main(String[] args){
        GeneralType<String> num = new GeneralType<String>("5");
        System.out.println("给出的参数是： "+ num.getObj());
    }
}
```

类型擦除（Erasure）

- 其实Java的泛型都是伪泛型，其为了能够后向兼容Java旧时代（Java 4.0及以前，还没有泛型时）代码，并不是运行时进行泛型的支持，而是通过编译器在编译阶段对类型进行擦除

```
class Foo<T> {
        T x;
        T someMethod(T y) { ... }
}

public Class Test{
  public static void main(String[] args){
      Foo<Integer> q = new Foo<Integer>();
      Integer r = q.someMethod(s);
  }
}
```

Actually

```
class Foo {
        Object x;
        Object someMethod(Object y) { ... }
}

public Class Test{
  public static void main(String[] args){
      Foo q = new Foo();
      Integer r = (Integer) q.someMethod( (Integer) s);
  }
}
```

类型擦除（Erasure）

- 如果类型是受限的，则会替换为其限定类型

```
class Foo<T extends Number> {
        T x;
        T someMethod(T y) { ... }
}

public Class Test{
  public static void main(String[] args){
      Foo<Integer> q = new Foo<Integer>();
      Integer r = q.someMethod(s);
  }
}
```

Actually

```
class Foo {
        Number x;
        Number someMethod(Number y) { ... }
}

public Class Test{
  public static void main(String[] args){
      Foo q = new Foo();
      Integer r = (Integer) q.someMethod( (Integer) s);
  }
}
```

一些注意点

- 泛型的实际参数类型不能是原始类型（因为都会被擦除为Object或其限定类）
- instanceof 判断不了泛型，比如： T<String> a = ... a instanceof T<Integer> ; //Error
- 不能用泛型创建对象，即 T a = new T(); （因为本质上是new Object()）
- 不能用泛型创建数组对象，即 T[] a = new T[size]；
- 不能声明静态的泛类型的变量，如：

```
public class Singleton<T>
   {
      public static T singleInstance; // ERROR
   }
```

思考擦除后发生什么?

泛型的类型通配符（Wildcard）

- 之前一个泛型对象名只能引用同一种泛型对象，如 GeneralType<String> a 只能指向GeneralType<String>的对象。
- 如果要使用同一个泛型对象名去引用不同的泛型对象，就需要使用通配符 "?" 创建泛型类对象。
  - 但要求不同泛型对象的类型实参必须是某个类或者其子类，或实现某个接口

    泛型类名 <? extends T> x = null;

例子

只能是Number的子类，如果没有extends，那么默认extends Objects

```
class GeneralType <T>{
    T obj;
    public void setObj (T obj){
        this.obj = obj;
    }
    public T getObj(){
        return obj;
    }
}
```

```
GeneralType <? extends Number> x = null;
x = new GeneralType <Long> ();
x = new GeneralType <Integer> ();
Number a = x.getObj(); //Correct
x.setObj(Integer.valueOf(1)); //Error
```

对于JVM而言，其无法确定x的对象的实际参数类型到底指向Number哪个子类，
如果实际是Integer类型，此时写入Long修改会出错 （Integer和Long不是父子关系！）
但是读取是不影响的（都以Number类型的方式读取）

泛型的类型通配符

- 除了可以利用extends限定实际类型参数是某个类型的子类外（设置上限），还可以用super 限定其是某个类的父类（设置下限）。
- 语法： 泛型类型 <? super anyclass> x = null;

例子

```
class GeneralType <T>{
    T obj;
    public void setObj (T obj){
        this.obj = obj;
    }
    public T getObj(){
        return obj;
    }
}
```

只能是Integer的父类

```
GeneralType <? super Integer> x = null;
x = new GeneralType <Object> ();
x = new GeneralType <Number> ();
Number a  = x.getObj(); //Error
x.setObj(Integer.valueOf(1)); //Correct
```

编译器并不知道到底是Integer的哪个父类
（注：这里可以强制转化为Object类使得编译通过）

可以写入Integer，因为其赋给任何其父类都是合法的

其他特性

对象的一些特殊函数

- 对于对象类而言
  - String toString()，默认返回 class名称 + @ + hashCode 的十六进制字符串，重写可以定制一个对象的具体打印结果（System.out.println()）
  - boolean equals()，判断两个对象是否相等，默认两个对象是否是同一个对象，但一般将其重写为连个对象的内容是否相等（自己定义"内容相等"）。比如字符串类就重写了这个函数，只要字符串包含的内容相等，equals()就返回 True
  - int hashCode()，返回对象的散列值（hash code），当重写了equals()之后，必须也重写该函数，使得equals()为真的两个对象，hashCode()的值也必须一致（hashCode是hashMap的核心）。

Lambda表达式

Lambda表达式不可对外面变量作修改!

```
parameter -> expression
(parameter1, parameter2, ...) -> { code block }
```

例子：

```
import java.util.ArrayList;
public class Main {
  public static void main(String[] args) {
    ArrayList<Integer> numbers = new ArrayList<Integer>();
    numbers.add(5);
    numbers.add(9);
    numbers.add(8);
    numbers.add(1);
    numbers.forEach( (n) -> { System.out.println(n); } );
  }
}
```

```
List<Integer> primes = Arrays.asList(new Integer[]{2, 3,5,7});
int factor = 2;
primes.forEach(element -> { factor++; });
```

error

流（Stream）

- Java流是一种对集合进行链状流式的操作，但是操作不会改变原来的数据（函数式编程思想）

例子：

```
List<String> strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");
List<String> filtered = strings.stream().filter(string -> !string.isEmpty()).collect(Collectors.toList());
```

流（Stream）

map

```
List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);
List<Integer> squaresList = numbers.stream().map( i -> i*i).collect(Collectors.toList());
squaresList = numbers.stream().map( i -> i*i).distinct().collect(Collectors.toList());
```

filter

```
List<String>strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");
long count = strings.stream().filter(string -> string.isEmpty()).count();
```

sorted

```
List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);
List<Integer> sortedList = numbers.stream().sorted().collect(Collectors.toList());
List<Integer> sortedList = numbers.stream().sorted(Comparator.reverseOrder()).collect(Collectors.toList());
numbers.stream().sorted((a, b) -> {return a.compareTo(b);}).forEach( n ->  System.out.println(n) );
```

reduce

```
int reduced = IntStream.range(1, 4).reduce((a, b) -> a + b).getAsInt();
```

流（Stream）

Iterate

```
Stream<Integer> stream = Stream.iterate(0, (x) -> x + 2).limit(6);
stream.forEach(System.out::println);
```

Match

```
List<Integer> list = Arrays.asList(1, 2, 3, 4, 5);
boolean allMatch = list.stream().allMatch(e -> e > 3); //false
boolean anyMatch = list.stream().anyMatch(e -> e > 3);  //true
boolean noneMatch = list.stream().noneMatch(e -> e > 10); //true
```

Min, Max

```
List<Integer> list = Arrays.asList(1, 2, 3, 4, 5);
Integer max = list.stream().max((a, b) -> a.compareTo(b)).get(); //5
Integer min = list.stream().min(Integer::compareTo).get(); //1
```

反射（Reflection）

- 反射是一种在运行时可以检视自身程序和操纵程序内部属性的一种语言特性。
  - 比如对于Java而言，反射可以使其运行时动态的加载类并获取类的详细信息，从而可以操作类和对象的属性和方法

例子

```
import java.lang.reflect.*;

    public class DumpMethods {
        public static void main(String args[])
        {
            try {
                Class c = Class.forName("java.util.Stack");
                Method m[] = c.getDeclaredMethods();
                for (int i = 0; i < m.length; i++)
                System.out.println(m[i].toString());
            }
            catch (Throwable e) {
                System.err.println(e);
            }
        }
    }
```

Output:

```
public boolean java.util.Stack.empty()
public synchronized java.lang.Object java.util.Stack.peek()
public synchronized int java.util.Stack.search(java.lang.Object)
public java.lang.Object java.util.Stack.push(java.lang.Object)
public synchronized java.lang.Object java.util.Stack.pop()
```

例子

```
import java.lang.reflect.*;
    public class TestInvoke {
        public int add(int a, int b) {
            return a + b;
        }
        public static void main(String args[])
        {
            try {
                Class cls = Class.forName("TestInvoke");
                Class partypes[] = new Class[2];
                partypes[0] = Integer.TYPE;
                partypes[1] = Integer.TYPE;
                Method meth = cls.getMethod("add", partypes);
                TestInvoke methobj = new TestInvoke();
                Object arglist[] = new Object[2];
                arglist[0] = new Integer(37);
                arglist[1] = new Integer(47);
                Object retobj  = meth.invoke(methobj, arglist);
                Integer retval = (Integer)retobj;
                System.out.println(retval.intValue());
            }
            catch (Throwable e) {
                System.err.println(e);
            }
        }
    }
```

```
import java.lang.reflect.*;

    public class TestConstruct {
        public TestConstruct(){
        }
        public TestConstruct(int a, int b){
            System.out.println(
                "a = " + a + " b = " + b);
        }
        public static void main(String args[])
        {
            try {
                Class cls = Class.forName("TestConstruct");
                Class partypes[] = new Class[2];
                 partypes[0] = Integer.TYPE;
                 partypes[1] = Integer.TYPE;
                 Constructor ct = cls.getConstructor(partypes);
                 Object arglist[] = new Object[2];
                 arglist[0] = new Integer(37);
                 arglist[1] = new Integer(47);
                 Object retobj = ct.newInstance(arglist);
            }
            catch (Throwable e) {
                System.err.println(e);
            }
        }
    }
```

一些常见的有用的Java库

一些常见的有用的Java库

	描述	常用的类
java.lang	语言包（默认引入）	Object、String、Math、System、Exception、Class、Thread、Throwable
java.io	输入输出流的文件包	OutputStream、InputStream、PrintWriter、File、FileInputStream、FileOutputStream、BufferedReader、BufferedWriter
java.util	实用工具包	Date、Calendar、List、Map、Set、Stack、Random、Currency、Locale
java.net	网络包	URL、Socket、ServerSocket、HttpCookie
java.sql	数据库处理包	Connection、Statement、PreparedStatement、ResultSet
java.text	文本处理包	Format、DateFormat、NumberFormat

此外还有一些常用的第三方Java库：Junit、Log4j、JavaFx、Weka、Hadoop、SpringBoot、Android...

Any questions ?