Java程序设计语言

回顾面向对象

数据属性，实例变量，成员变量

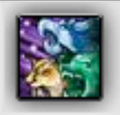
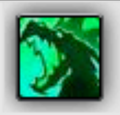
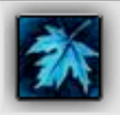
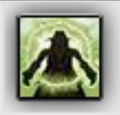
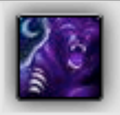
```
class Tauren:
    # Set the initial values
    def __init__(self, strength, agility, intelligence):
        self.strength = strength
        self.agility = agility
        self.intelligence = intelligence
```

实例化一个牛头人英雄

```
tau = Tauren(100, 60, 30)
print(tau.strength)
tau.strength += 30
```

点表达式来访问数据属性

回顾面向对象

15	 豹之迅捷	 野性位移	 野性冲锋
30	 伊瑟拉之赐	 新生	 塞纳里奥结界
45	 精灵虫群	 群体缠绕	 台风
60	 丛林之魂	 化身：乌索克之子	 自然之力
75	 夺魂咆哮	 乌索尔旋风	 蛮力猛击
90	 野性之心	 塞纳留斯的梦境	 自然的守护
100	 艾露恩的卫士	 粉碎	 鬃毛倒竖

函数属性，方法

```
class Tauren:
    # Set the initial values
    def   War-Stomp(self):
        """效果：影响范围有敌人的话，敌人眩晕"""
    def   Shockwave(self, direction):
        """效果：所在方向的敌人都会减少100血量"""
```

```
tau = Tauren(100, 60, 30)
tau.War-Stomp()
```

点表达式来调用方法

回顾面向对象

- 继承


```
class TaurenDruid (Tauren, Druid):
    # inherit the parents' attributes
```

回顾面向对象

- 多态

子类型多态

```
class TaurenDruid (Tauren, Druid):
    # change the parents' attributes
    def __init__(self, strength, agility, intelligence):
        self.strength = strength
        self.agility = agility
        self.intelligence = intelligence
```

Java程序设计语言入门

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

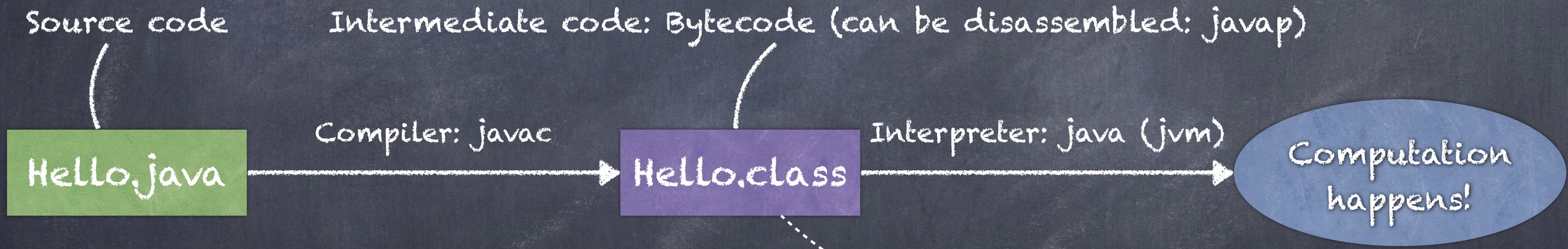
A good programmer must be multilingual

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，解析比源码快
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

回顾Lambda calculus

- 语法：合法的λ项
  - $M, N ::= x \mid \lambda x\,.\,M \mid MN$
- 语义：归约法则
  - $$\frac{}{(\lambda x\,.\,M)N \to M[N/x]}(\beta)$$
  - $$\frac{M \to M'}{MN \to M'N} \qquad \frac{N \to N'}{MN \to MN'} \qquad \frac{M \to M'}{\lambda x\,.\,M \to \lambda x\,.\,M'}$$

回顾Lambda calculus

- 回顾一个λ项$(\lambda x\,.\,x\ x)(\lambda x\,.\,x\ x)$

$$\rightarrow (\lambda x\,.\,x\ x)(\lambda x\,.\,x\ x)$$

$$\rightarrow \ldots$$

为Lambda calculus增加类型*

- 类型
  - $\tau, \sigma ::= T \mid \sigma \to \tau$

base type (e.g. int, bool)

function type

- λ项
  - $M, N ::= x \mid \lambda x : \tau . M \mid MN$

为Lambda calculus增加类型*

- 归约法则
  - $$\frac{}{(\lambda x : \tau \,.\, M)N \to M[N/x]}(\beta)$$
  - $$\frac{M \to M'}{MN \to M'N} \qquad \frac{N \to N'}{MN \to MN'} \qquad \frac{M \to M'}{\lambda x : \tau \,.\, M \to \lambda x : \tau \,.\, M'}$$

为Lambda calculus增加类型*

- 类型判断（judgement）
  - 判断就是对某个形式化属性的声明语句
  - 对于类型而言，即声明该term的类型
  - ⊢ 表明该声明有一个推导过程（即证明）

为Lambda calculus增加类型*

- 类型判断（judgement）
  - $\Gamma \vdash M : \tau$

    M is of type τ in context Γ
  - 上下文Γ是对于M中每个自由变元的类型设定，比如如果出现了变元x, Γ必须包含一个$x : \sigma$这样的设定
  - Γ可以是·表示为一个空的上下文

为Lambda calculus增加类型*

- 类型推导规则（Typing rules）

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}\ \text{(var)}$$

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash (\lambda x : \sigma . M) : \sigma \to \tau}\ \text{(abstraction)}$$

$$\frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau}\ \text{(application)}$$

为Lambda calculus增加类型*

- 例子

$$\frac{}{x : \tau \vdash x : \tau}\ (\text{var})$$

$$\frac{}{\cdot \vdash (\lambda x : \tau\,.\,x) : \tau \to \tau}\ (\text{abstraction})$$

为Lambda calculus增加类型*

- 例子

$$\frac{}{x : \tau, y : \sigma \vdash x : \tau}\ (\text{var})$$

$$\frac{}{x : \tau \vdash (\lambda y : \sigma . x) : \sigma \to \tau}\ (\text{abstraction})$$

$$\frac{}{\cdot \vdash (\lambda x : \tau . \lambda y : \sigma . x) : \tau \to (\sigma \to \tau)}\ (\text{abstraction})$$

→一般默认右结合，可以省去括号

为Lambda calculus增加类型*

- 例子

$$\frac{}{x : \tau \to \tau, y : \tau \vdash x : \tau \to \tau}\ (\text{var}) \qquad \frac{}{x : \tau \to \tau, y : \tau \vdash y : \tau}\ (\text{var})$$

$$\frac{x : \tau \to \tau, y : \tau \vdash x : \tau \to \tau \qquad x : \tau \to \tau, y : \tau \vdash y : \tau}{x : \tau \to \tau, y : \tau \vdash x\ y : \tau}\ (\text{application})$$

$$\frac{x : \tau \to \tau, y : \tau \vdash x\ y : \tau}{x : \tau \to \tau \vdash (\lambda y : \tau . x\ y) : \tau \to \tau}\ (\text{abstraction})$$

$$\frac{x : \tau \to \tau \vdash (\lambda y : \tau . x\ y) : \tau \to \tau}{\cdot \vdash (x : \tau \to \tau . \lambda y : \tau . x\ y) : (\tau \to \tau) \to (\tau \to \tau)}\ (\text{abstraction})$$

再次回顾之前的例子*

- $(\lambda x\,.\,x\ x)(\lambda x\,.\,x\ x)$无法给定一个类型：

这里x只能是σ, 不可能是$\sigma \to \tau$

$$\dfrac{\dfrac{}{x : \sigma \vdash x : ?}\ \cdots \qquad x : \sigma \vdash x : \sigma}{x : \sigma \vdash x\ x : ?}\ \text{(application)}$$

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;
```

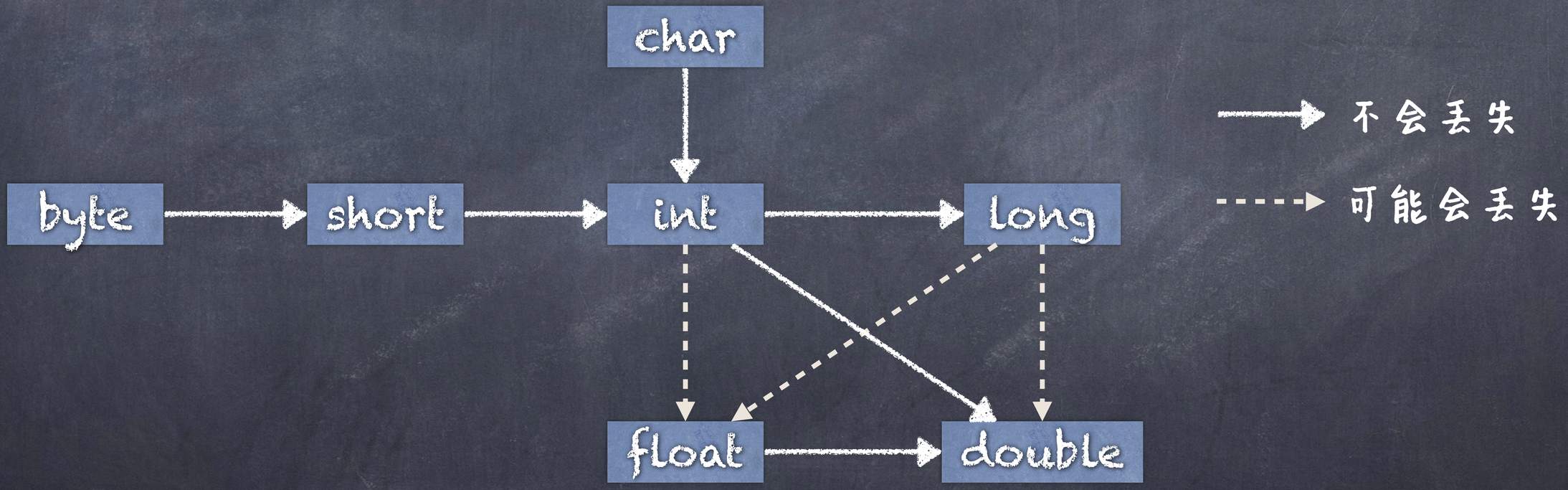
转化为运算中的最高的类型

```
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之后的特性：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);
}
```

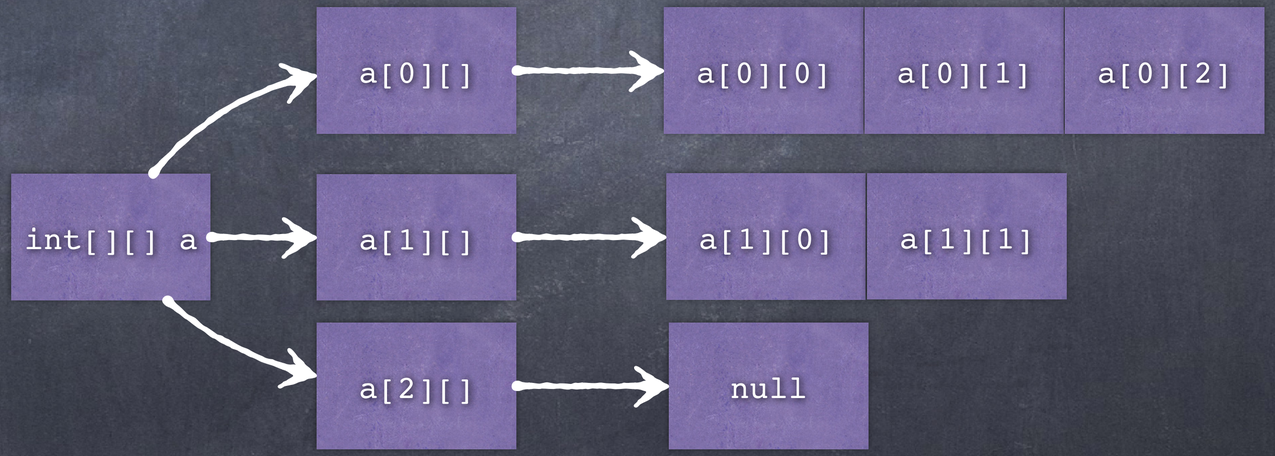
多维数组

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”
```

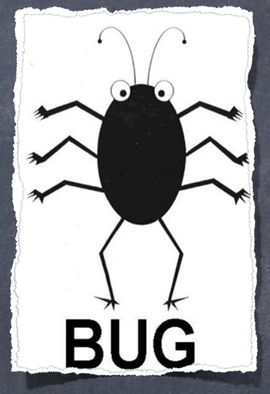
内容相等

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?

Any questions ?