Study Guidelines for CSE 11 at UCSD

Introduction

- 计算机科学，尤其是计算机科学的入门课程，是关于解决问题和学习新的编程语言。
  编程语言和计算机只是工具，他们不是计算机科学的最终目标，只是实现目标的工具。

- Program（程序）：是一系列的指令（instructions）或者算法（algorithms）。程序被人类理解和计算机理解的格式是不同的。

- Java：一个完全的 object-oriented programming（OOP，面向对象编程）语言。
  在 Java 中，你必须创建对象然后对对象进行操作
  你可以把对象想象成可以重复使用、模块化的程序组件。在现实中的例子，对象可以是微波炉，桌子，etc。。。

- 学习例如 Java 的编程语言可以学到：
  使用 Java 语言来设计和定义你自己的类（class）和方法（method，或者叫函数 function）
  如何应用已经定义于 Java 的 library 和 package 里面的类和方法来优化自己的程序

- 典型的 java 程序开发涉及：编辑（edit），编译（compile），运行（run），etc。。。
  编辑：可以用自己喜欢的 text editor（IDE）来写 code, 。推荐 VS Code（功能强大，简单易上手），Eclipse（更加专业，上手难度高）
  编译：将人类理解的 Java 程序翻译成计算机理解的语言（将在后续课程中学习）
  运行：运行 Java Application

- Java 例子：

(Edit)

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

(Compile)

```
javac Foo.java
```

(Run)
java Foo

Variables

- Knowledge Points: Primitive Data Type and Variable types
- Helpful Topics: Variable declaration
- Key Words: Variable

- Primitive Data Type（原始数据类型）：最基础的数据类型

名称	中文名	例子	说明	存储空间（byte）	范围
double	双浮点	4.20	存储小数，比 float 更大，全名也叫 double float	8	10^308
float	浮点	-1.25f	存储小数	4	10^38
long	长整型	420000000	存储较大的整数	8	10^18
int	整型	-10149	存储整数	4	10^9
short	短整型	3733	存储较小的整数	2	10^4
byte	字节	32	一个字节，8 位（bits）	1	100
char	字符	‘G'	一个字符，utf-16 字符标准	2	N/A

- 数据类型转换（conversion）

  automatic conversion：从存储空间小的到存储空间大的会自动转换

  e.g. double x = 7，从 int 转到 double

  type cast：明确的表示转换的类型。

  e.g. int x = (int) 2.5，从 double 转换成 int，x=2

- division

  integer division: 得到的结果是整数。如果结果是小数，则会自动丢弃小数点部分。

  float division: 得到的结果是浮点数。如果结果是整数，则会自动添上.0

  e.g.

  ```
  int x = 5;
  int y = x / 2;
  double z = x / 2;
  double p = x / 2.0;
  ```

  RESULT: x = 5, y = 2, z = 2.0, p = 2.5

按照结果，x/2 应该是 2.5，但因为 x 是 int，2 也是 int，所以这里是 int division，结果是 2。所以 y=2

同理，又因 z 是 double，所以 z 是 2.0

因为 x 是整型，2.0 是 double，两个 operands 中有一个是 non-int 型，所以会进行 float division。等号的右边结果是 2.5，因为 p 也是 double，所以 p 的结果是 2.5

- Variable（变量）：在内存（memory）中一个储存数据的区域。

  变量会有相应的名称。

  在 Java 中，变量会有类型（type）并且不会改变其类型

  在某个时刻，变量会有其储存的值（value）。

变量会首先被声明（declare），指出他的名字和类型。
然后变量会被初始化(initialize)，赋予值
但我们也可以同时声明和初始化一个变量。
e.g.

```
int someNumber;                declare
someNumber = 1;              initialize
```

等同于

```
int someNumber = 1;
```

Console IO

- output(System.out.print & System.out.println)

在例如 Java 和大部分语言中，代码运行的时候不会有任何显示，因此我们需要输出一些东西来给用户。

System.out.print(myVar); 将输出 myVar 里面的值到用户控制台。这个输出结果不会另起新的一行。

System.out.println(myVar); 将输出 myVar 里面的值到用户控制台。这个会将光标移至新的一行。

e.g.

```
System.out.print("Thi");
System.out.println("s is);
System.out.print("Java XD");
----- Output:
    This is
    Java XD
```

- input (Scanner)

  读值：nextInt，nextDouble，next

  检查是否还有可读的数据：hasNext，hasNextInt，hasNextDouble

  e.g.

  Scanner myScanner = new Scanner(System.in);　　//将新定义的 myScanner 绑定到一个 IO 设备。System.in 在大多数情况下是键盘

  int val;

  val = myScanner.nextDouble();

  System.out.println("I read: " + val);

Conditional Statements: if, if-else, switch

- boolean Data Type

  布朗（Boolean）只有两个值：True（真/是），False（假/否）。

- Relational Operators: > >= < <= == !=

  Relational Operators 是二元运算符，在其左右两侧需要有两个 operands。

  将根据数据类型和数学逻辑比较 operands 的值并返回一个布朗值。

>	大于	x > y	判断 x 是否大于 y
>=	大于等于	x >= y	判断 x 是否大于等于 y
<	小于	x < y	判断 x 是否小于 y
<=	小于等于	x <= y	判断 x 是否小于等于 y
==	相等	x == y	判断 x 是否等于 y
!=	不相等	x != y	判断 x 是否不等于 y

判断逻辑分为两种情况：

若为数值类型（Int，Double，Float），将按照数学逻辑判断数值的大小

若为字符类型（char，string），将根据 ASCII 表上对应字符的转义大小判断。按照字符顺序一个个比较。若同一位 ASCII 值相同，则比较下一位，以此类推；若同一位 ASCII 值不相同，则按照数学逻辑输出结果；若比较到一个字符串（string）完结，则完结的字符串值更小。

- Logical Operators: && || !

  用来做逻辑运算。Logical Operators 的 Operand(s)必须是布朗值。

  && 逻辑与

x	y	x && y
T	T	T
T	F	F
F	T	F
F	F	F

  || 逻辑或

x	y	x || y
T	T	T
T	F	T
F	T	T
F	F	F

  ！逻辑非

x	!x
T	F
F	T

- If，if-else，else-if

  最基础的逻辑模块，if（如果）用来实现 branching（根据 if 中的条件不同执行不同的结果）

  条件（conditional）的结果或返回值必须是布朗值。

  e.g.

  ```
  if (1 == 2)            if ("same" != "same)       if (1 > 2 && '1' > '2')
  double pi = 3.14;      if (3.14 >= pi)
  boolean myBool;        if (myBool)
  ```

  if 格式：

  ```
  if (conditional) {
      statements;
  }
  ```

```
// 如果 conditional 是 True，则执行 { } 内的语句；False 则跳过（不执行）
if (conditional) {
    statement1;
} else {
    statement2;
}
// 如果 conditional 是 True，则执行 statement1；否则，执行 statement2
```

在 else 中也能另起新的 if 或 if-else 判断：

```
if (conditional1) {
    statement1;
} else if (conditional2) {
    statement2;
} else {
    statement3;
}
```

// 如果 conditional1 是 True，执行 statement1；否则，判断 conditional2，如果是 True，则执行 statmenet2；如果是 False，执行 statement3；

- switch

  当我们需要进行多余两种情况的分支时，使用 if 和 if-else 的组合容易出错并且减少代码的可读性。这时我们可以使用 switch。

  格式：

```
switch(myInt) {
    case 1: statement1;      // 如果 myInt 的值是 1，则运行
            break;           //如果没有 break，则会继续判断 case2，3 。。。
                             //break 会在 loop 中讲解
    case 2：statement2；
            break；
    default：System.out.println("No match value");      // 如果没有 case match，则运行 default 中的指令
}
```

- pre- post- increment/decrement: ++x, x++

我们可以用++/-- 来代替掉 x = x + 1；x = x - 1；这样的语句来使代码更加精简。

其中自加（increment）和自减（decrement）分为两种：pre 和 post

以++为例来讲解（--同理）

pre：++x

在计算这个语句之前，更新 x 的值

e.g.

```
int x = 5;
System.out.println(++x);     //输出 6
System.out.println(x);       //输出 6
```

post：x++

在计算这个语句之后，更新 x 的值

e.g.

```
int x = 5;
System.out.println(x++);   //输出 5
System.out.println(x);       //输出 6
```

Loop

- for, while, do while

  循环（loop）指重复执行某些语句一定次数或重复执行至达成某些条件后停止。是编程中常用的逻辑形式。

loop 方法	格式	解释	例子

for	for(初始化；条件；变更）{
	statements;
	}
		初始化：初始化一些需要用到的变量
		条件：每次循环开始前会进行判读，如果满足条件则运行。
		变更：每次循环后需要变更某些变量。无变更或者未
			for (int i = 0; i < 5; i++) {
			System.out.println(i);
			}
			System.out.println("end");
			// i 一开始是 0
			// 当 i 小于 5 的时候运行
			// 每次循环后 i 自加
			// 所以将循环 5 次
			// 分别打印 0，1，2，3，4

		正确变更会导致无限循环（infinite loop）	// 当 i=4 时，打印完 4 后，i 将自加至 5 // 此时进入下一次循环，进行条件判断。i<5 不成立。跳出循环。打印“end“

while	while (条件) {
	statements;
	}
		每次循环前先判断条件，如果条件成立，执行循环体中的语句。否则结束循环	int i = 0;
			while (i < 5) {
			System.out.prinln(i++);
			}
			System.out.println("end");
			// 首先我们在循环外定义 i=0
			// 然后进入循环。
			// 每次循环，打印 i 并且将 i 自加。
			// 如果忘记将 i 自加会导致无限循环，因为 i 一直是 0，不会改变，判断条件又是 i<0，所以条件一直成立，循环用不停止
			// 当打印完 4 后，i++变成 5，此时进入下一次循环判断。i < 5 不成立，跳出循环，打印“end“

do while	do {
	statements;
	} while (条件）;
		跟 while 比较相似，不同的是 do while 是先执行 body 中的语句，再进行判断。而 wile 是先进行判断，后执行语句。
		所以 do while 中的语句无论如何都会至少运行一次
			int i = 0;
			do {
			System.out.println(i++);
			} while (i < 5);
			System.out.println("end");
			// 首先我们初始化 i=0
			// 然后进入 do while，依次打印 0，1，2，3，4
			// 打印完 4 后，此时 i 自加后成 5，然后进行条件判断，i<5 不成立，结束循环，打印“end”

- nested Loop

  以上提到的 3 中 loop 都能相互嵌套实用，达成 nested loop 的效果

  e.g.

```
for (int i = 1; i < 4; ++i) {
        int j = 1;
        while (j < 3) {
            System.out.println(i * j);
            j++;
        }
}
// 外侧循环（outer loop）将循环 3 次，分别为 i=1，2，3
// 内侧循环（inner loop）将循环 2 次，分别未 j=1，2
// 因此，将打印 1（1x1），2（1x2）
//                  2（2x1），4（2x2）
//                  3（3x1），6（3x2）
```

- for loop variants

  for loop 中的三个条件均可以省略不写，但也会产生影响

  1. 不写初始化

     可以在 for loop 之前声明需要的变量

     e.g.

```
for (int i = 0; i < 5; ++i) {}
```

     等同于

```
int i = 0;        for (; i < 5; ++i) {}
```

  2. 不写条件

     导致无限循环，因为没有判断条件（默认为 True），所以会不停循环

  3. 不写变化

     可以在循环体内部写需要的变化

     e.g.

```
for (int i = 0; i < 5; ++i) {}
```

     等同于

```
for (int i = 0; i < 5; ) {
        i++;
}
```

- break and continue

  break 和 continue 是控制循环的两个方法

  break：从当前循环中结束并跳出循环

  e.g.

```
for (int i = 0; i < 9999; ++i) {
        if (i == 2) break;   // 当 i 循环到 2 的时候，结束循环
        System.out.println(i);
}
System.out.println("Break at here");
 // 当 i=0 的时候，if 判断未否，继续循环，打印 0
 // 当 i=1 的时候，同理，继续循环
 // 当 i=2 的时候，if 判断成功，运行 break，直接结束当前循环并跳出
 //    因此不会再打印 2
 // 跳出循环，打印“Break at here”
```

  continue：从这次循环中结束，并开始下一次循环

  e.g.

```
for (int i = 0; i < 5; ++i) {
        if (i % 2 == 0) continue;            // 如果 i 是偶数
        System.out.println("odd: " + i);
}
// if 判断的是，当 i 是偶数的时候，continue，开始下一次循环，不打印
// 所以这段 code 会打印"odd: 1", "odd: 3"
```

String & StringBuilder

- String

  String（字符串）在 Java 中用来储存一些列的字符。

  与其他某些语言不同，String 在 Java 中并不是 primitive data type，他是一个 object。

  String 是不可更改的（immutable）。一旦 String object 的内容被声明后，变不可修改

  但是指向这个 object 的 String 变量是可以更改的（指向新的 String）

  e.g.

```
String a = "Hello";
```

a = a + " world"; // okay,但并不是更改了之前的“Hello” object，而是 a 指向了
// 重新定义的一个新的 object 并且内容未“Hello world”
// 之前的“Hello”不变

String 中的常量 literals 是唯一的。

e.g.

String a = "Hello"; // stack，将在后面讲 stack 和 heap

String b = "Hello"; // stack

String c = new String("Hello"); // dynamic allocation，这个 string 在 heap 中

// a == b，True，两个都是 literals，是恒定的唯一的。都位于 stack 中，同一个对象

// a == c， False，一个是 literal，另一个是动态定义的。一个在 stack，另一个在 heap

- 常用的 String operators
  - +（concatenation）：相连，将两个 String 连接到一起

    “a” + “b“ ==> “ab” （这个“ab”是重新定义的一个 String，因为 String 是 immutable 的
  - .length()：String 类中的方法，返回这个 String 的长度

    String a = "hello";

    System.out.println(a.length()); // 5
  - .concat(str)：类中的方法，在末尾出连接 str。返回一个新的 String 因为 String 是 immutable

    e.g.

    String a = "hello";

    System.out.println(a.concat(" world")); // "hello world"

    a.concat(" world");

    System.out.println(a); // "hello"，因为 a 仍然指向“hello”

    // 我们在上一步 concat 产生的新的

    // String 并没有重新赋值给 a

    // 注意！String is immutable
  - .substring(left_index, right_index): 类中的方法，返回一个子字符串

    注意！left index 是 inclusive，right index 是 exclusive。【left，right）

    注意！index 从 0 开始，不是 1

    String a = "hello";

    System.out.println(a.substring(0, 0)); // "h"

```
System.out.println(a.substring(2, 5));        // "ll0"
System.out.println(a.substring(2,6));         // Exception(error): IndexOutOfBound
```

- .charAt(index): 类中的方法，返回 index 中的字符

```
String a = "hello";
System.out.println(a.charAt(1));              // "e"
```

- .indexOf(char): 类中的方法，返回找到的第一个 char 的 index；返回-1 如果未找到

```
String a = "hello";
System.out.println(a.indexOf('o'));           // 4
System.out.println(a.indexOf('l'));           // 2
System.out.println(a.indexOf('g'));           // -1
```

- StringBuilder

另一种数据类型用来储存字符串。是 object 不是 primitive data type

也 String 的不同是 StringBuilder 可以改变对象的内容，是 mutable（可改变）的

e.g.

```
StringBuilder a = new StringBuilder("Hello");
a.append(" world");                   //  在末尾加上新内容，append 也是类方法
System.out.println(a.toString());     //  打印的时候要用.toString()类方法转换成 String
                                          // "Hello world"。StringBuilder is mutable
```

Operator Precedence

Precedence	Operator	Type	Associativity
15	() [] .	Parentheses Array subscript Member selection	Left to Right
14	++ --	Unary post-increment Unary post-decrement	Left to Right
13	++ -- + - ! ~ (*type*)	Unary pre-increment Unary pre-decrement Unary plus Unary minus Unary logical negation Unary bitwise complement Unary type cast	Right to left
12	* / %	Multiplication Division Modulus	Left to right
11	+ -	Addition Subtraction	Left to right
10	<< >> >>>	Bitwise left shift Bitwise right shift with sign extension Bitwise right shift with zero extension	Left to right
9	< <= > >= instanceof	Relational less than Relational less than or equal Relational greater than Relational greater than or equal Type comparison (objects only)	Left to right
8	== !=	Relational is equal to Relational is not equal to	Left to right
7	&	Bitwise AND	Left to right
6	^	Bitwise exclusive OR	Left to right
5	|	Bitwise inclusive OR	Left to right
4	&&	Logical AND	Left to right
3	||	Logical OR	Left to right
2	? :	Ternary conditional	Right to left
1	= += -= *= /= %=	Assignment Addition assignment Subtraction assignment Multiplication assignment Division assignment Modulus assignment	Right to left

References and Primitives

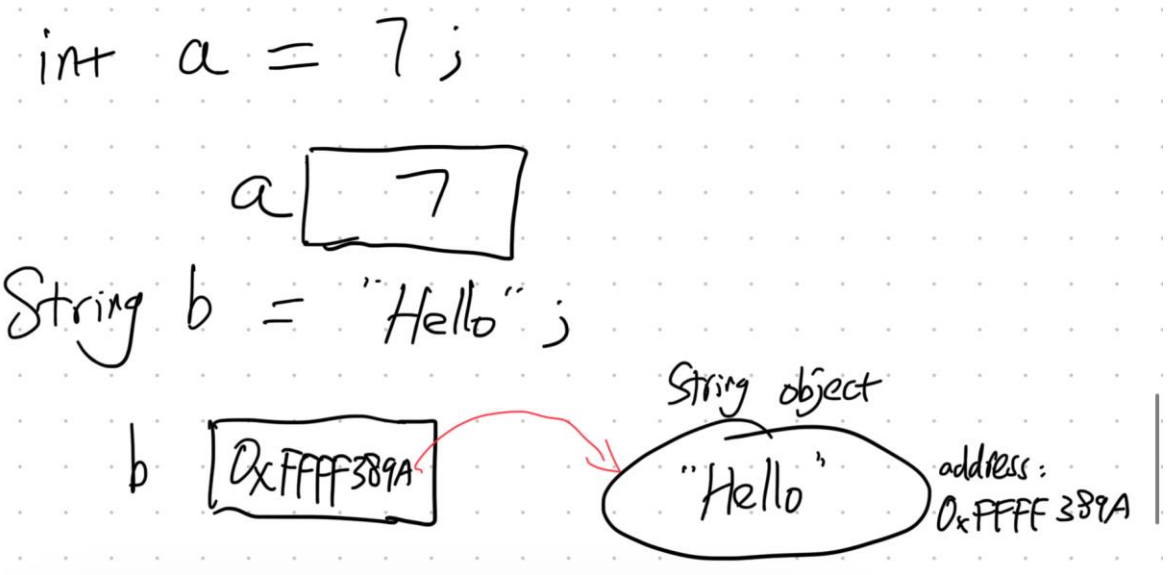
- primitive vs reference（引用）

  相同点：都是存储 value 的变量

  不同点：

  primitive 存储的是这个 value 本身

  reference 存储的是这个 value 在 memory（内存）中的地址（address）



- null reference

  是一个 reference，但其中的地址是 null（无）。

  这个 reference 不指向任何东西

Memory

在 Java 中，有两块区域用与存储，分别为 stack（堆）和 heap（栈）

- stack

  当我们在 main 函数或其他方法中声明变量的时候，这些变量称为 local variables，他们存在于 stack 中

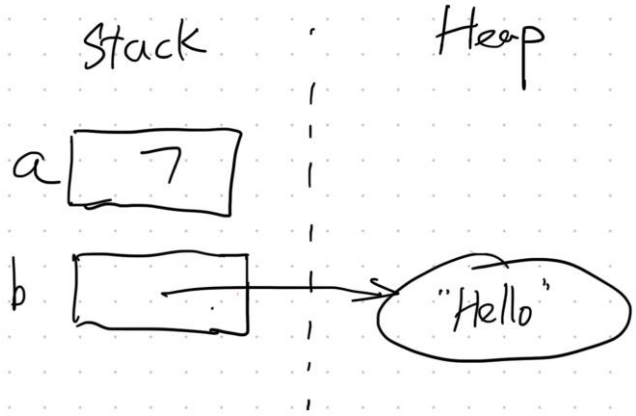
- heap

  当我们用 new 这个 keyword 动态声明 object 的时候，生成的 object 存在于 heap 中，但引用他们的 reference 仍然在 stack 中。

e.g.

int a = 7;

String b = "Hello";



- memory model

  用于便于我们理解程序内存实用的一个模型。例如上述例子中的图示

  life span（生命周期）：变量或者对象从产生到摧毁的这段周期

  scope（范围）：每个变量或对象都有自己的范围，只有在这个范围内能使用他们

  一般来说：变量的周期是从他定义的位置，到所处的语句块{ }结束的地方。

  在其定义之前不能使用他，因为从未被声明过

  在语句块结束的地方，也就是 } 后，也不再能使用。因为 Java 会自动清理掉这已经超出范围的变量来释放内存空间。

  对象也相似。Java 也会有自己的 garbage cleaning 指令来帮助我们自动清理掉我们 new 的一些对象。但在其他语言（例如 C，C++），这个对象的自动清理是不存在的，需要我们手动用 delete 关键词来清理，否则会造成内存泄漏（memory leak）

- 关于构建 memory model 的一些 principles
  - 位置
    - 变量存在与 stack 中
    - 对象存在于 heap 中
  - 类型
    - primitive：最简单的数据类型（int，double，etc。。。）
    - reference：手握对象的缰绳（注意！未正确合理使用 reference 将导致某些对象无法再被访问，因为你丢掉了找到他的绳子）
  - 值

- primitive：变量变身的值
- reference：对象的地址

Method

非常基础的变成思路。将程序细分成一个个功能独立的模块，每个模块执行特定的功能。

更容易思考，使代码更加易读懂和除 bug，并使代码重复使用率增加（重复使用模块）

- methods

  方法（或叫函数）。

  e.g.

```
public static int addition(int a, int b) {        // 1
    return a + b;                                  // 2
}
```

  // 1：method signature，是函数的签名，可以理解为函数的名字

  // public：公开的，表示所有人都可以使用这个函数（后续细讲）

  // static：表示这个 method 在所属类中只会有一个 copy。不论用这个类声明多少个不同的对象，当调用他们的 addition 时，都将来到这个类定义的内存位置来运行函数；如果定义时不写 static，将默认与 static 相反，即声明的不同对象中都会有一个 addition 函数的 copy，在他们各自的内存中，当调用时，会运行各自内存中 addition 的副本中的内容。

  // int：函数的返回类型。这个 method 结束后将返回一个整型。可理解为函数的运行结果

  // addition：函数的名字

  // （int a，int b）：formal parameters，调用这个方法的时候需要提供两个值。而 a 和 b 将相应的被赋与提供的这两个值

  // 2: method body. 函数体。这个函数被调用后将运行的内容。

  // return：关键字，一个是提供这个函数的返回值，另一个是在这里结束函数调用。return 后面的代码（若有）不会运行。

```
public static void main(String args[ ] ) {        // 3
    System.out.println(addition(5, 6));           // 4
}                                                 // 5
```

  // 3: main 函数。所有的程序都是从 main 开始运行的。

  // void（无），表示 main 函数不返回任何值。在其他语言中，也有习惯让 main 返回一个 int，根据返回的值来判断 main 是否正常结束

// (String args[ ]): formal parameters，这个适用于命令行（command line）调用 main 的借口。在后续在 Formatted IO 中会讲到。

// 4: main 的 body。这里只有一条 output 语句。output 的内容是 addition 的调用。因此，当运行到次时，由于必须知道 addition 的返回值才能进行 output，系统将运行的步骤跳至 addition 中，并将 5，6 作为 arguments 传入 method 里。

// 到 addition 中，会返回 5+6（11），这个返回值被 System.out.println 接收到，并打印

// 5: 此处 main 结束。因为 main 的 return type 是 void，则可以不写 return 语句。

// 当然，也可以写不带任何值的“return；”来控制 main 的结束。

- stack frame

  stack 的增常方向是像上的（实际上是向更低的 address 增长，但方便理解，我们写的时候说向上增长，就像堆东西一样），具有先进后出的特点（First In Last Out）

  当调用方法是，被调用的方法会在 stack 上留下 record。我们把一个个这样的 record 叫做 frame

  程序当前运行的部分永远是 stack 最上面的 frame

  程序的开始都是从 main 函数开的。一开始 stack 是空的，当执行 main 的时候，在 stack 上留下 main frame

  ```
                  ----------------------------------      stack top
  main      |     some local variables     |
                  ----------------------------------      stack bottom
  ```

  假如我们在 main 中调用 addition method，则会在 main 上方插入，并执行 addition

  ```
                  ----------------------------------      stack top
  addition  |     local variables: a, b      |
                  ----------------------------------
  main      |     some local variables     |
                  ----------------------------------      stack bottom
  ```

  当 addition 结束并 return 是，Java 将释放 addition 在 stack 中的 frame，并返回到 main

  ```
                  ----------------------------------      stack top
  main      |     some local variables     |
                  ----------------------------------      stack bottom
  ```

  main 可能会使用 method 的结果进行一些操作。最终 main 也将结束 return。释放 main

  ```
                  ----------------------------------      stack bottom & top （empty stack）
  ```

  更多关于 memory 的内容将在 memory 的 CSE 课程中学习，此处只是简单了解

- Overloading

  当函数的名字相同但函数的签名（method signature）不同时，我们称之为 overload（重载）

  函数的签名(signature) = 函数的名字(name) + 函数的参数列表(formal parameters)

  因此，如果两个函数重名，要重载他们就得改变参数列表。

  改变参数列表可以有：参数的类型，参数的数量，参数的顺序

  注意，改变参数的名称并不会改变参数列表。Java 是根据数据类型类来区别的。

  e.g.

```
public static int addition( int a, int b);              // 1
public static float addition( float a, float b);        // 2, overload
public static void main(String args [ ] ) {
    System.out.println(addition(5, 6));                 // call 1, 11
    System.out.println(addition(3.7, 1.1));             // call 2, 4.8
    System.out.println(addition(5, 1.1));               // Error. No matching signature
}
```

- methods testing

  测试函数是一个良好的习惯，也是一个优秀的程序员必备的技能。

  因为我们把程序模块化成一个个函数，因此为保证程序完整正确的运行，必须保证每个函数完整正确的运行。

  单独测试（unit test）每个函数可以更我们更容易除 bug。

  - how to test

    现阶段我们可以在 main 中调用函数并打印调用的结果来判断。例如上面的例子。

    后续会有更高级的方法和工具，将在后续课程中学习

  - general case

    在测试函数中我们需要测试通常情况（general case）。因为是通常情况，我们无法穷举他们，因此我们只需要适量的几个 general case 测试即好。

    例如：在 addition（int，int）这个 method 中，general case 就可以是正常的整型，如 5，3；100，-1；-5，-6 等

  - edge case

    在测试中，我们也不要忘记边缘情况（edge case），往往这类情况最容易被忽略导致 bug。

    edge case 的情况不多，所以在合适的情况下我们需要穷举测试。

例如：在 addition（int，int）中，edge case 可以是

如果有 0 能正常运行吗

如果其中的一个不是 int 类型会报错嘛

等等。。。

- Function call tracing

  当我们写更长更加复杂的程序的时候，如果出现了 bug 但不知道在哪里，每个 method 的 test 都正常时，可能是 method 的调用过程中出现了问题。

  我们可以借助 stack frame 中的方法，把程序每个阶段的 stack 样子作出判断，看看时候正确。这个过程叫做 function call tracing（函数调用追溯）

Recursion

- Recursion（递归），指函数通过重复调用本身来实现功能。

递归的理解有一定难度。写出复杂的递归算法能减少代码空间，增加运行效率。

熟练的掌握递归将使你称为一个更有竞争力的程序员

以下将通过一个例子来讲解 recursion

- Fibonacci Sequence: 是一个经典的数学 sequence。其中的某 x 项是第 x-1 和第 x-2 项的和

  公示描述如下：

  $f(x) = f(x - 1) + f(x - 2),\quad x >= 2$

  $f(0) = 1, f(1) = 1$

其中 f(x)的公示定义了 recursive case，是我们递归过程中每一次计算的过程。

f(0), f(1)定义了 base case，是我们递归结束或者说返回的标志，他告诉了我们什么时候停止递归并一步步返回。

假如我们要计算出第 4 项的值，通过数学我们能罗列出

1，1，2，3

我们如果把他写成 method，便可以有如下代码

```
public static int f( int i) {
    if (i == 0 || i == 1) return 1;            // base case
    return f (i - 1) + f (i - 2);                    // recursive case
}
public static void main(String args[ ]) {
    System.out.println(f (3));                        // recursion call
                                                                // 3 since index start from 0
```

}

我们来一步步分析。

1. 程序从 main 开始，执行 output 语句，调用 output 函数。output 函数中需要的 argument 来自 f（3），于是中止 output，跳入 f（3）计算结果。stack frame 如下

   bot || main | System.out.println() | f (3) || top

2. 开始运行 f（3），因为 3 不等于 0 或 1，不是 base case，所以我们要返回 f(3-1) + f(3 - 2)，也就是 f(2) + f(1)。因为语句的计算是从左到有，所以 return 发现需要 f(2)的值的时候，便会终止语句的计算，先跳入 f(2)进行计算（即，还没看到后面还要计算 f(1)，看到 f(2)就调入调用了

   bot || main | System.out.println() | f (3) | f (2) || top

3. 开始运行 f（2），同理，不是 base case，需要返回 f(1) + f(0)。同理，需要 f(1)所以进入 f(1)的调用

   bot || main | System.out.println() | f (3) | f (2) | f (1) || top

4. 当进入 f（1）的时候，发现是 base case，直接返回 1

   bot || main | System.out.println() | f (3) | f (2) || top

5. 此时运行返回到 f（2）中，f（2）return 语句的左侧（f(1)）我们已经取得了返回值 1，因此将继续运算。发现运算还需要右侧的 f（0）。再次进入 f（0）的函数调用

   bot || main | System.out.println() | f (3) | f (2) | f (0) || top

6. f（0）同 f（1）理，是 base case。返回 1

   bot || main | System.out.println() | f (3) | f (2) || top

7. 此时我们的 f（2）的到了加法左右两侧的值（都是 1），f（2）返回 1+1 = 2

   bot || main | System.out.println() | f (3) || top

8. f（3）得到了加法需要的左侧 f（2）的值，还需要右侧 f（1）的值。同样，将 f（1）放到 stack frame 中

   bot || main | System.out.println() | f (3) | f (1) || top

9. f（1）是 base case， 返回 1

   bot || main | System.out.println() | f (3) || top

10. 此时 f（3）得到了右侧需要的值。f（3）返回语句中左右两侧的值都得到了，计算结果后返回 2 + 1 = 3。返回后结束 f（3）的调用，返回 Output 中

    bot || main | System.out.println() || top

11. output 得到了需要打印的结果是 3。完成打印后，返回 main

    bot || main || top

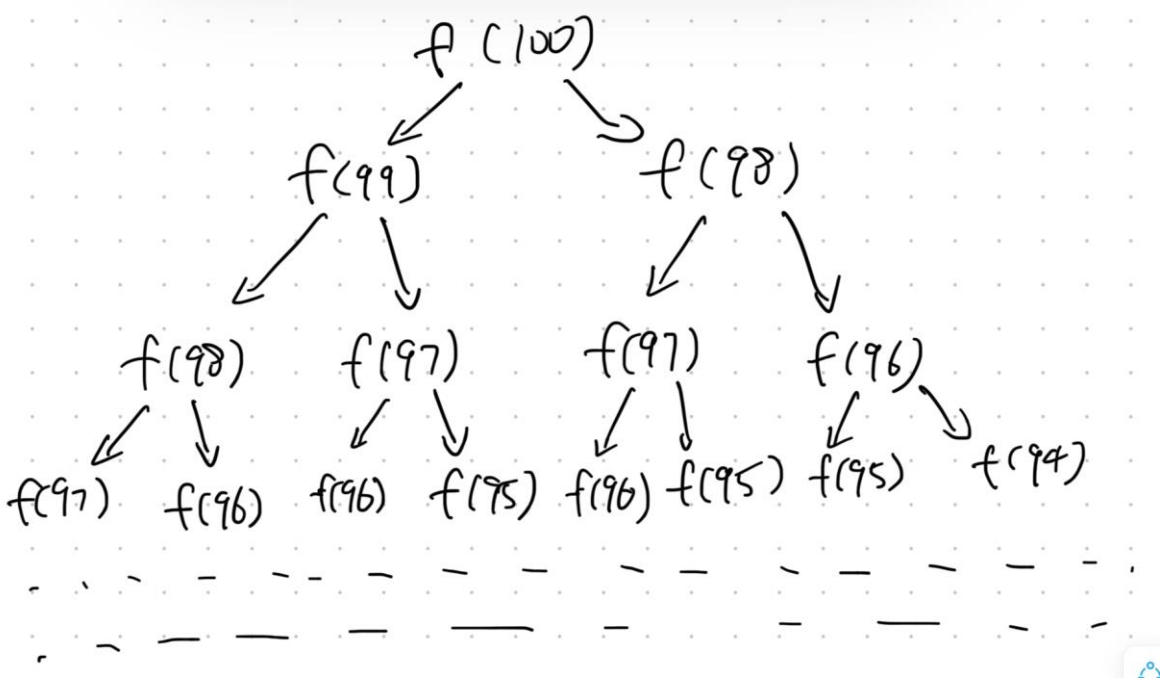
12. 此时 main 已经没有任何剩余语句了。结束。stack 被清空

    bot || || top

- Efficiency

  通过上面一个简单的 fib 例子，相信大家发现了这个 recursion 算法中的内存效率问题。

  如果需要计算的值很大（例如 100），那么将会变得非常不效率。



  因为语句的求值计算是线性的，从左到右，我们会发现会一直沿着最左侧的线路走到底，在一步步往右边的走计算右边的值。这样会产生两个问题：

  1. 有很多重复的计算。例如图中 f（96）就计算了 4 次。想象下这样链式下去，f（3）会计算多少次？（几万几十万次 !）而我们的算法却没有任何记忆这些已经算过的函数值的方法。重复了很多无意义的运算
  2. 在计算靠近 base case 的值时，上层函数的 stack frame 会一直存在于 stack 中。占用内存。一个程序分配到的 stack 空间是有限的，这样链式计算下去中有一颗会超出 stack 的空间，这会导致程序直接被 kill（结束），我们永远计算不出 f（100）的值

我们会发现，在某些情况下，递归并不一定比循环要有效率。比如在 f（100）的情况下，如果我们通过循环一步步从小到大计算出每一项的值，则可以计算出 f（100）并且比递归的方法要快上好多好多。

P. S. 这并不是说递归在计算量大的情况下一定不如循环。在后续的 CSE 课程中会学习不同的算法和数据结构来优化此类的递归。那时候递归的效率会大大增加。

- iteration vs recursion

  iteration（循环）和 recursion（递归）在一定程度上可以相互改写

  e.g.

  // print 0, 1, 2

```
// Iteration Version
for (int i = 0; i < 3; ++i) {
    System.out.println(i);
}
// Recursion Version
public static void foo(int i ) {
    if (i == 3) return;
    System.out.println(i);
    foo(++i);
}
public static void main(String args[ ]) {
    foo(0);
}
```

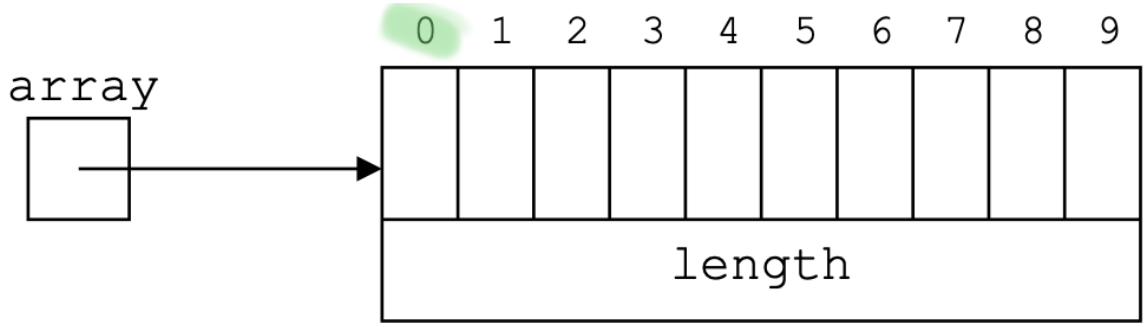
1D array

- Array

  array（数组），是一个内存连续的储存相同数据类型的数据结构

  在 Java 中是一个对象（在 heap 中）。

  数组的长度在定义的时候便确定了，并且无法被更改。



因为 array 是一个 object，而我们定义 reference 去指向他们。所以我们可以改变 reference 指向的 array。

```
e.g.
int[ ] my_array = new int[5];          // old array, length of 5
 my_array = new int[6];          // new array, length of 6. Lose old array addr
```

- array.length

  可以用 array.length 来检视数组的长度（上面图中 length 是 10）

- 定义 array 的几种方法：

  1. int[ ] array; // array 是一个 reference，目前指向 null

// int[ ]表示一个整型的数组

array = new int [10]; // 在 heap 中定义一个大小为 10 的数组，让 array 指向它

2. int[ ] array = new int[10]; // 也可以写一起
3. int array[ ] = new int[10]; // 这是 c/c++语法的书写习惯，同样可以

- Array Initialization

当 array 被定义的时候，里面的值会被自动初始化（如果没有明确指出初始化值）

如果是数字类的数组，初始化成 0

如果是布朗数组，初始化成 False

如果是储存 reference 的数组，里面的 reference 初始化成 null

一些初始化的方法：

1. int[ ] array = {1, 2, 3};
2. int[ ] my_arrayarray = new int[ ] {1, 2, 3, 4, 5};

- Element Access

要访问 array 中的元素（element），我们用 subsrcipt[ ]

e.g.

```
my_array[0] = 3;                    // set 1st element to value 3.
System.out.println(my_array[4]);// print the last (5th) element. should be 0
```

- Array Traversal

如果要遍历数组，我们可以借助 length

e.g.

```
for (int i = 0; i < my_array.length; ++i) {
    statements;
}
```

- IndexOutOfBoundException

当我们访问不存在的 index 时，会产生 IndexOutOfBoundException 报错。

e.g.

```
int[ ] a = {1, 2, 3, 4, 5};
System.out.println(a[4]);           // Okay, print last element
System.out.println(a[5]);           // Exception! Out of bound.
```

- Vectors & ArrayList

Vector 和 ArrayList 是两类类似于 array 的 class。与 array 不同的是，他们是动态的，即数组的长度可以变化（就像 StringBuilder 和 String 的关系一样）

使用他们需要 import 他们的 library。他们 class 中还有其他很多方便 array 操作的 methods。详细可以查 Java Library Reference

- Array of Char

  An array of Characters is NOT a String

  虽然说 String 是一些列 char 组成的。但在 Java 中一个储存 char 的数组并不等于 String。（不像 C 中，char 的数组可以表示 String）

  如果你定义了一个 char array，也可以通过 String method 将它转化成 String

  e.g.

  ```
  char[ ] c_array = {'h', 'e', 'l', 'l', 'o'};
  String hello = new String (c_array);          // constructor, become String
  char[ ] new_c_array = hello.toCharArray();    // become char array.
  ```

Formatted IO

- Scanner

  之前我们见到过 Scanner，在键盘输入的时候用 Scanner。

  但 Scanner 不仅能从键盘输入，也可以用于文件输入。

  e.g.

  ```
  File file = new File("source.txt"); // suppose source.txt is the file we wanna read
  Scanner in = new Scanner(file);   // link the Scanner to the source.txt
  while (in.hasNext()) {                    // check if remain things
   System.out.println(in.nextLine());     // do something we want
  }
  ```

  注意，当我们 link Scanner 的时候，可能会报错 IOException，其原因可能是

  1. 文件名不存在。找不到目标文件
  2. 没有权限访问目标文件。

- PrinterWriter

  PrinterWrite 让我们能够将 output 引流到文件类的输出，而不只单单输出到控制台。

  e.g.

  ```
  File file = new File("output.txt");              // declare a new file named "output.txt"
  PrinterWrite out = new PrinterWrite(file);       // file object goes here
  out.println("Hello World");                      // write to "output.txt"
  out.close();                  // don't forget to close the file. some program can't open the
          // file if it was linked but never close.
  ```

  如果 output.txt 文件不存在，则会在当前目录下新建一个 output.txt 文件，并写入。

  如果 output.txt 文件已经存在，则会覆盖其之前的内容。

- Command Line

  我们可以通过 command line 像程序传入一些参数，使得 main 不再是单独运行的，而是可以与外界交互。

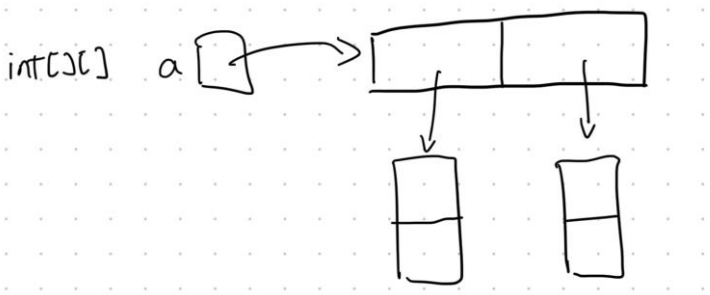
  e.g.

```
public static void main (String args[ ]) {              // args in fact is just array!
        for (int i = 0; i < args.length; ++i) {
              System.out.println(args[i]));              // note that args elements are String!
        }
}
----------------------------Terminal-----------------------------
javac myCode.java                                  // compile
java myCode hello wolrd I love CSSA // run
hello world I love CSSA                            // program running result output
```

2D Array

- 2D array

  2D array 其实就是 1D array，但是其每个元素都是一个 1D array



  声明方法：

```
int[ ] [ ] a = new int [2] [3];          // 2 rows and 3 columns.
```

  遍历方法：

```
for (int i = 0; i < a.length; ++i) {          // a.length = 2
 for (int j = 0; j < a[0].length; ++j) {     //a[0].length = 3; meaning its element's length
        System.out.println(a[i][j]);
  }
}
```

- Array class in Java

  我们可以调用 Java 中的 Array library 来使用一些已经被定义好的数组操作方法。

```
import java.util.Arrays;          // import array library
```

以下是一些常用的方法(均为 static method）：

1. Deep Copy

   Deep Copy（深度复制）会声明一个新的 array object 来储存其中的元素。

   与之相反的是 shadow copy（表面复制）。

   ```
   int[ ] a = new int[ ] {1, 2, 3};
   int[ ] b = a;                    // shadow copy. a and b points to same array object
   int[ ] c = Arrays.copyOf(a, a.length);          // deep copy. Copy a new array
                                                   // object with same elements using
                                                   // Arrays method
   ```

2. copyOfRange

   我们可以明确 copy 的范围

   ```
   int[ ] d = Arrays.copyOfRange(a, 0, 2);      // d = {1, 2, 3}, [0, 2) range
   int[ ] e = Arrays.copyOfRange(a, 1, 4);      // range can be larger.
                                                        // d = {2, 3, 0, 0}
   ```

3. fill

   我们可以把数组填充为某个特定的值

   ```
   int[ ] empty = new int[5];
   Arrays.fill(empty, 100);        // empty = {100, 100, 100, 100, 100}
   ```

4. toString

   为了方便打印，Arrays 提供了 toString 函数

   ```
   System.out.println(Arrays.toString(empty));
   ```

5. Deep Comparison

   与 Deep Copy 和 Shadow Copy 类似的，我们也有 Deep/Shadow comparison

   Deep comparison 会比较其中每个元素的值是否相等

   Shadow comparison 只会比较是否是同一个数组对象

   ```
   int[ ] array1 = new int[ ] {1, 2, 3};
   int[ ] array2 = new int[ ] {1, 2, 3};
   System.out.println(array1 == array2);         // Shadow Copy. Not Same object.
                                                 // print 0
   System.out.println(Arrays.equals(array1, array2));          // Deep Copy.
                                                               // equal elements
                                                               // print 1
   ```

6. Sort

   sort（排序）会直接作用在数组对象上，而不是产生一个新的数组对象

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

```
Arrays.sort(a);        // a = {1, 2, 3}. Default ascending order
String[ ] b = new String[ ] {"hello", "world", "123"};
Arrays.sort(b);        // b = {"123", "hello", "world"}; Based on ASCII value
```

ArrayList

ArrayList 是类似于 array 的类。与 array 不同的是他是动态长度的

capacity: ArrayList 占用的空间

size：ArrayList 中元素的数量。size <= capacity

当 size 将要超过 capacity 时，默认会在现 capacity 的基础上 double。也可以自己定义

ArrayList 是设计用来储存 reference 的，而不是一般数据类型如 int，double

```
ArrayList<Type> arraylist_name = new ArrayList<Type>( );
```

如果需要用来储存一般数据类型如 int，需要用 wrapper class

```
ArrayList<Integer> arraylist_name = new ArrayList<Integer>( );
                    // Integer is a class in Java. Store int values. Have some methods
```

一些常用的 ArrayList 方法(not static)：

1. boolean add(E e);
   在数组的最后附上一个新的 reference e
   E 代表 type，这是一种 generic type，用来表示所有的 type
   这里会被特定为 ArrayList 声明时候的 type
2. void add(int index, E element);
   把新的元素添加到 index 的位置。后续的元素依次后退一位
3. E get(int index);
   获得在 index 位置的元素
4. int size( );
   返回 ArrayList 的 size（即元素数量）
5. E remove(int index);
   将在 index 位置的元素移除。后续的元素依次前移一位
   被移除的元素将作为函数的返回值。

- ArrayList 是 list interface 的一个应用
  我们可以用 list 的 reference 指向 ArrayList 对象，因为后者是 list 的一个 implementation
  同样的 list reference 可以指向另外的对象，只要指向的对象所属的类是 list interface 的一个 implementation。
  （具体查看 composition and inheritance）

Class and Object

- OOP (Object-Oriented Programming)

  OOP（面向对象编程），是现代编程的一种重要思想

  我们把程序分类并封装成对象，每个对象包括

  对象的属性　　和　　对象的方法

  对象的属性和方法可以分为实现层面（implementation）和交互层面（interface）

  1. implementation

     隐藏在对象内部，其中包括对象实现其功能的逻辑和函数

     由关键词 private 表示。外界不能调用 private 的属性或方法

  2. interface

     面向外界，外界可以通过这类交互借口来使用这个对象

     由关键词 public 表示。外界可以调用 public 的属性和方法

  在 OOP 中，class（类）是基础单位。class 定义了 type

  class 定义了属性，方法。包括其中的 implementation 和 interface

  object（对象）是 class 的实例。我们通过 new 来创建一个实例并通过 constructor 来将其初始化。这些发生在 runtime（运行过程）而不是 compile time（编译过程）。因为我们用 new 来创建，而 new 是动态声明的

  class name 是用户自己定义的数据类型名字。

  现实中对象的例子：例如在家中，会有

  1. 桌子
     a. 类：桌子。对象：某品牌的 7 月份生产的某个桌子
     b. 属性：木质，4 脚。。。
     c. 方法：可以放东西，可以办公。。。
     d. private：桌子是怎么拼接起来的，木头是怎么处理的
     e. public：把桌子四角着地就能站立起来，并且可以承重放东西
  2. 微波炉对象：
     a. 类：微波炉。对象：某品牌某生产线上某日的一个产品
     b. 属性：用电，金属的。。。
     c. 方法：可以加热，可以解冻。。。
     d. private：电路，加热的微波等物理原理方法
     e. pulic：通过微波炉的案件可以选择模式或者设置加热时间

Java 中一个 class 的例子：

```
public class Circle {                    // Usually, Capital first character
    private int x;
    private int y;                  // (x, y) in 2D plane
    private int radius;             // these are all members (attributes)

    public Circle() {                   // constructor. constructor has no return type
        this.x = 1;
        this.y = 2;
        this.radius = 3;                // default circle initialization
     }
    public void changeSize ( int r) {       // a public method. Change circle size
        this.radius = r;                        // "this" is a keyword specify this class
                                            // member
    }
}                                   // class definition ends here

public static void main(String args[ ]) {
    Circle myCircle = new Circle();         // declare by default constructor
    myCircle.changeSize(5);                     // call public method to change size
}
```

- Instance Variable

  instance variable 指的是我们在 class 中定义的 members，也就是那些属性。

  他们存在于 heap 中，是 class object 的一部分。

  他们的生命周期跟所属的 object 一致。

  他们的范围（scope）是在 class 的定义范围内

  去访问他们的时候，分为两种情况：

  1. 外界访问

     ref.instanceVariableName;

     注意这样的访问只能是访问 public 的 variable

     e.g.　　myArray.length;

  2. 内部访问

     this.instanceVariableName;

     内部访问指在 class 定义中的一些使用。例如我们需要在一些 method 访问这些

variable

this 是一个关键词，特指 class 的 variable（有些 method 中会有重名的 variable）

e.g.

```
public void changeSize( int radius) {
  this.radius = radius;                    // duplicate name! use this to specify
}
```

这些 instance variable 如果没有被标注初始化，则会被自动初始化为类 0 值

- Class Variable

  通过 static 关键词来定义　e.g. public static int objects_number;

  也是存在于 heap 中，但与 instance variable 不同的是，前者存在于 heap 中关于此 class 的定义中，而不是存在与 object 中

  static 将使这个 variable 变得唯一且共享。所有的 class objects 都共享这个变量。

  所以如果他是 public 的，那么所有的对象都可以直接访问并且更改他。

  例如可以通过上述例子中的 objects_number 来跟踪这个 class 创建的 object 的数量

  他的生命周期从 class 的定义加载至 class 的定义卸载的时间段。一般就是这个程序运行的周期（而 instance variable 的周期是对象的生命周期）。

  他们的范围（scope）是在 class 的定义范围内。

- Local variable and Formal Parameter

  Formal Parameter 在前面有定义过，指的是 method signature 的参数列表内的参数

  Local Variable（本地变量）：指的是在 method 函数体内定义的变量。他们的范围只在所定义的函数体内{ }.

  e.g.

```
public void changeSize ( int radius) {        // radius is formal parameter
      int r;                                  // r is local variable
      r = radius;                             // local variable must be initialized before use
                                              // they don't have default value
      this.radius = r;                        // use local variable to modify instance variable
}
```

- Block Scope

  我们定义的变量通常范围都是在所属的 block 内 { }。block 之外的不能调用他们

  e.g.

```
int i = 0;                              // scope in outer block
for (int i = 0; ... ; ... ) {...}       // Okay. Within the loop, will access the inner i
                                        // instead of the i declared before
```

---

e.g.

```
System.out.println(i);                    // Error. i has only scope in the for loop.
for (int i = 0; ... ; ... ) {...}
```

---

e.g.

```
for (int i = 0; ... ; ...) {...}
for (int i = 0; ... ; ...) {...}          // Okay. last i ends its for loop scope. This is a
                                          // new i which has scope in its for loop.
```

Composition and Inheritance

- Composition of classes (has-a relationship)

  我们可以让一个 class 内包含另一个 class，这两个 class 形成 composition，也是 has-a 关系，意思是什么有什么的关系

  e.g.

  ```
  public class Student {
   public int age;
   public String name;            // String is a class. Has-a relationship
  }
  ```

- Inheritance (is-a relationship)

  Inheritance（继承），是 class 的另一种关系。是一种 is-a 关系，意思是什么是什么

  在 Java 中，inheritance 用 extends 和 implements 来表示继承关系

  extends 用于已经实现的 class（concrete class）

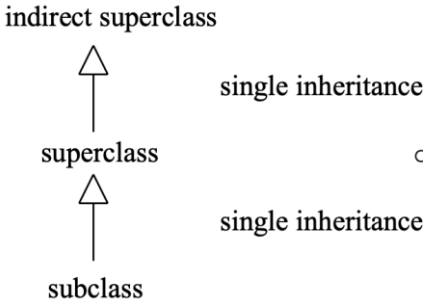
  implements 用于交互（interface）

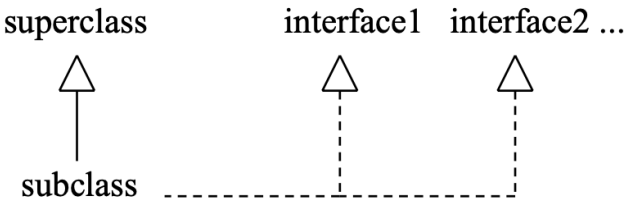
  当 B 继承 A 的时候，B 继承了 A 所有的 public and protected attributes 和 methods。

  B 可以重新定义继承的 methods 来实现其目的。

  注意：

  1. subclass 只能 extends 一个 superclass
  2. subclass 可以 implements 多个 interface
  3. subclass 可以既 extends 又 implements





- Subclass and Superclass

  如果 B 继承了 A，我们称 A 为 superclass，B 为 subclass。

  B 可以使用所有 A 的 public 和 protected

- Private, Protected, Public

  public：可以被所有的 Java 代码访问

  protected：可以由 subclass 访问

  private：只能在这个 class 的定义中被访问

- this, super keywords

  this：用来表明访问这个 class 的对象本身

  super：用来表明访问这个 class 的 super class 的内容

- constructor

  constructor（构建函数），会在对象生成的时候自动调用，用来初始化对象

  他是一种特殊的函数，与 class 同名并且没有返回类型

  我们也可以显性（explicitly）地调用它

  this()：调用本类的构建函数

  super()：调用 superclass 的构建函数。当我们继承的时候，会得到 superclass 的一些属性，这些属性就可以由 super() 来构建。

  constructor 也可以被 overload

- constructor rules
  - 如果没有定义 constructor，Java 会自动生成一个默认 constructor。其函数体只有一个语句：super()
  - constructor 的第一条语句必须是 super()。如果不是，Java 会在编译的过程中自动插入 super()。所以我们也可以不写它。

- Abstract Class, Concrete Classes and interface

  1. Concrete Class

     concrete class 是可以用来创建对象的。

其中不能有 abstract method（用 abstract 关键词标注的函数）。

当我们 extends 的时候，extends 后的 class 也是 concrete class，因为它完整的定义了所有的函数（即，没有 abstract method）

2. Abstract class

我们用 abstract 关键词来标注类（public abstract class something....)

在 abstract class 中必须定义至少一个 abstract method (public abstract void ...( ))

当 subclass extends asbtract class 的时候，subclass 必须定义 super class 所有的 abstract method。不然会产生编译错误。

不能创建 abstract class 的对象，因为 abstract class 的作用一般就是提供一个模板，所有的 subclass 必须都实现模板中的一些功能。

e.g.

```
public abstract class Mammal {                  // asbtract class
        private String name;
        public abstract String speak();         // subclass should speak differently
 }

 public class Human extends Mammal {
        @Override           // indicate it's overriding abstract method
        public String speak() {
                return "Hello";
        }
 }

 public class Cat extends Mammal {
        @Override
        public String speak() {
                return "Meow";
        }
 }
 // 注意
 // overwrite 指同一个 class 中同名函数的不同定义
 // override 指在 inheritance 关系中，subclass 对 superclass 的同名函数的重新定义
```

3. Interface

Interface 与 abstract 类似，但是 interface 中全部都是 abstract method。而 abstract class 只要求至少有一个 abstract method

interface 定义了所有 implements 它的 subclass 所需要的 public method 和他们所需要完成的功能

interface 不能有任何的 method 定义，不能由 constructor，也不能有 instance variable。但是可以定义 public static final 常量。这些常量会成为 subclass 的定义中的一部分。

一个 class 可以 implements 许多 interface，但只能 extends 一个 class。

4. summary

Concrete Classes	Abstract Classes	Interfaces
class keyword in definition	abstract class. ~~class~~ keyword in definition	**interface** keyword in definition
extends keyword - pure inheritance of implementation	**extends** keyword - mixture of inheritance of interface and inheritance of implementation	**implements** keyword - pure inheritance of interface
is-a relationship	*is-a* relationship	*is-a* relationship
Defines a new type	Defines a new type	Defines a new type
Can extend from only 1 base class -single inheritance of implementation	Can extend from only 1 base class -single inheritance of implementation	Can implement many interfaces -multiple inheritance of interface
Can create instances of objects of concrete classes; fully defined	Cannot create instances of objects of **abstract** classes; incomplete	No notion of creating instances of interfaces
No **abstract** method declarations; only fully implemented (concrete) methods	Mixture of fully implemented methods and **abstract** methods	Only **abstract** method declarations and **final static** constants

- static compile time check & dynamic run time check

  polymorphism：我们可以定义一个 super class 的变量，将它赋值成其 sub class 的对象。详情请见 Polymorphism 章节。

  那么，当有多个不同的 class extends 或 implements 时，如何判断他们的实际类型呢？

  在编译(static compile time)的过程中，编译器会查看 reference 的类型，并且会判断调用的函数名、参数列表等是否复合 super class 中的定义。如果不是合法的应用，会产生 compiler error

  在运行(dynamic run time)的过程中，reference 指向的对象会根据程序的不同而指向不同的 subclass 对象。此时程序将在运行这个动态过程中，去检查一些函数调用（因为可能 subclass1 可以但是 subclass2 不可以）。如果不是合法的应用，会产生 runtime error

Polymorphism

- implicit subclass-object-to-superclass-object conversion

在继承关系中，假如说 B 继承了 A，那么

1. B 也是 A 的类型

   在继承关系中，subclass 也是 super class 的类型，因此我们可以用 superclass 的 reference 来指向 subclass 的对象。这个是 polymorphism 的核心基础。

2. B 可以调用由 A 中继承到的 method
3. A 不能调用 B 自己的（非继承）的函数和内容。

e.g.

```
superClass reference    = superClass object          // Okay
subClass reference          = subClass object        // Okay
superClass reference    = subClass object        // Okay. Polymorphism
subClass reference      = superClass object          // No! Compiler error
```

- Polymorphism

  Polymorphism（多态性），表示我们可以通过 superClass 的 reference 来操作控制不同 subclass 的对象。

  1. 程序可以被由许多 subclass 的对象来组成并运行。这些 subclass 可以通过继承的关系把他们连接起来
  2. 一些还没设计的类后续会比较轻松地进入我们搭建好的继承网络。因为他们也满足继承关系，所以我们可以直接使用已经写好了的 superClass reference 操作代码来控制这些新的 class。不需要再更改旧代码的例如类型和函数名等部分。省去了很多修改的过程。提高程序的可延展性。
  3. 所有的对象可以通过同一个 superClass reference 来多态操作。省去很多用于判断 object 类型的 switch 或者 if-else 语句。做到动态绑定（dynamic binding）。

- dynamic binding

  我们将通过一个例子来解释。

  假如我们有个 Shape interface，他只有一个函数 draw( )。我们有 Circle，Square，Triangle 这三个 subclass，他们都 implements Shape 并且定义了自己的 draw（）函数（例如 Circle 的 draw 会打印一个圆，Square 会打印正方形...）。

```
Shape shape;
shape = new Circle( );
shape.draw( );        // dynamically bind to Circle object. Actuall Call Circle.draw( )
shape = new Square( );
shape.draw( );        // bind to Square. Call Square.draw( )
shape = new Triangle( );
shape.draw;                  // bind to Triangle. Call Triangle.draw( ).
```

这些判断调用不同对象的 draw 是在 runtime 中进行的（所以叫 dynamic）。

在 compile time 中，compiler 只会判断我们的 reference 和 object 是否合法，不回去分配在不同的运行阶段 shape 到底是什么 object。这个判断是在 runtime 中进行的。shape 会随着 program 的运行指向不同的 object 并且调用他们自己的 draw

- final methods and classes

  final 是 Java 中的一个关键词，用来表示最终版/不能被再拓展的意思，具体用法如下：

  final method 不能再被 subclass override

  当我们声明 method 为 static 或 private 的时候，默认其为 final

  编译器会优化 final method 成 inline code，每次调用的时候不会再到相应的内存中寻找函数的定义，而是直接用函数题的内容替换掉原代码的位置，从而提高运行效率。这个过程发生在编译（compile）中。

  final class 不能是 super class

  即 final class 不能被 extends

  当一个 class 是 final 时，其所有的函数默认为 final

- polymorphism rules: compile time rules and runtime rules

  1. compile time rule

     编译器只知道 reference 的类型，而不知道其指向的具体 class 类型

     因为只知道 reference 的类型，所以只能看到其类型的 class 定义（superclass 的）

     解析 class 定义中的 method signature 并标记其 signature

  2. runtime rule

     根据 compile time 中 reference 的 type 并决定其实际指向的 object 的 class

     根据 compile time 中标记的 signature 运行实际 object class 中的 method（subclass

- Casting

  在 Polymorphism 中，casting 只是改变了编译器看待 reference type 的视角，而不是 cast runtime 中 reference 实际指向的 object 的类型

  1. 如果我们要 cast subclass 到 super class

     Super superRef = new Sub(); // implicitly upcast (widening)

  2. 如果我们要 cast super class 到 subclass

     Sub subRef = (Sub)superRef; // explicitly downcast (narrowing)

  3. 如果我们需要通过 super class reference 调用 subclass 的 method

     ((Sub) superRef).subClassMethod( ); // don't forget ( ). Precedence matter!

     // 如果没有最外侧的 ( )，则会现运行 superRef.subClassMethod( )

     // 再将其 cast 成 (Sub)。报错！

4. 对于 explicitly cast，如果我们需要 downcast，即 super -> sub，编译器要求必须要 explicit cast

- Equal method

  equals( )是 Object 这个类中定义的函数。Object 函数是 Java 中很原始的一个 class，所有的 class 都是从 Object inherit 出来的（即使我们没有写 extends Object，但实际上都是从 Object inherit 出来的）。我们来看一下 equals 的定义：

  ```
  public boolean equals( Object o ) {
      return this == o;
  }
  ```

  我们发现这是 shadow comparison，只比较了 reference 是否指向同一个 object。

  所以我们定义自己的 class 时，要注意 override equals 函数，使其实现 deep comparison，（其实当我们用一些 Java library 中的 class 例如 String 时，在这些 class 的定义中也 override 了相应的 equals 函数，所以我们使用起来没有问题）。

  在我们 override equals 要注意：

  1. 检查 null，然后比较两个是否是同一个 class 通过 getClass( )函数
  2. 比较内部的值，例如 instance variable 的 value 是否都相等，以此来实现 deep comparison
  3. 根据前两点的结果来返回 true/false

  当我们定义自己的 class 时，有 3 个函数通常我们都会 override：equals( ), toString( ), hashCode( )。其他两个有兴趣可以去查相关资料

Exception

- Exception（报错），是编程中很常用的一种函数处理技巧。

当我们（caller）调用一个函数（callee），callee 会有两种“返回”方式：

1. return。callee 完成其任务并且没有任何问题，通过 return 函数正常返回 caller
2. exception。callee 在运行中出现问题，中断运行返回 caller 的位置并抛出一个 exception 告诉 caller 出现了问题。

当 caller 接到 callee 抛出的 exception 时，通常会进行判断 excpetion 的类型来决定相应的处理方式。可以当时就解决，也可以忽略，也可以直接中止程序的运行。

exception 的出现使程序在运行的过程中允许错误的出现。我们可以根据错误的类型来决定程序的走向，而不会一味的出现问题就中止程序。例如当出现除以 0 的数学逻辑错误时，我们可以提示用户重新输入，而不是整个程序结束需要重新运行（当你用计算器时，除以 0 会报错，而不是整个计算器直接关机然后你得重新打开）。

- try, catch, finally

try catch block 是使用 exception 的一种标准格式

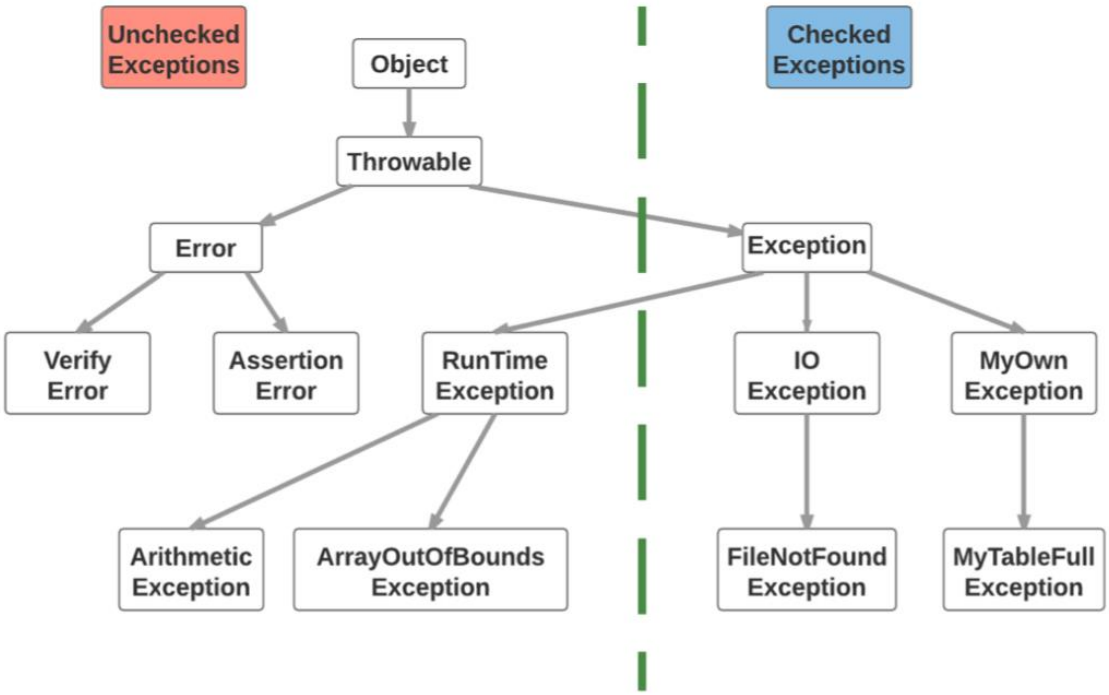
```
try {
        foo( );             // put method call that you think might throw expcetion in try
}
catch (ExceptionType1 e1) {         // if happen Exception Type 1 error
        // do something to handle it
        // you can also ignore
        // e1 is a reference of the exception. You can call method like
       // e1.printStackTrace( ) to see what happened in detailed.
 }
 catch (ExceptionType2 e2) {                // you can have as many catch as you wish
        // do something
 }
 finally {                  // like a default state, if all catch above didn't catch exception
        // do something
 }
 // 注意
 // 1. catch 的数量是任意的，但是每种 catch 只能 catch 一种 exception class
 //    你可以直接用 Exception（包含所有的 exception，类似于 Object 对于 class 的关
 //    系）来 catch 所有类型的 exception，因为 exception 也是一个庞大的继承网
 // 2. 如果没有任何 catch 接住了相应的 exception，则会运行 finally（如果有）。
 //    如果也没有 finally，则 exception 会被抛到更上一层的 caller，由上一层的 caller 来处
 //    理。如果这个函数已经是 main 了，即没有更上一层的 caller 了，则程序会中止，在
 //    terminal 中会显示 Exception 的报错信息
 // 3. catch 是按照顺序的，先检查第一个 catch，再第二个。。。所以如果你的第一个
 // catch 的 exception type 包括了后续的 catch 的 type，则后续的是没有意义的
```

- checked vs. unchecked exceptions

  Exception 和 Error 两个 class 是继承 Throwable class 的

  Error 是 unchecked exceptions，他会导致程序直接 crash 并且不能被 handle。

  Exception 是 checked exceptions，他们可以被 handle 来保证程序继续运行。但其中 runtimeException 不应该被 handle，应该让他直接 crash。因为 runtimeException 一般是由于一些内存原因导致的。runtimeException 也是 unchecked exception



- throw exceptions

  如果我们要声明一个 method 可能会抛出 excpetion，我们在函数的 signature 中声明

  e.g.

  ```
  import java.lang.*;
  ...
  public int division (int a, int b) throw ArithmeticExcpetion {
        if (b == 0) throw new ArithmeticException( );
        return a / b;
  }
  ```

Containers and Generics

这里会介绍一些 Java library 经常使用的数据类型

- HashSet

  提供比 ArrayList 更快的 Add，Delete，Search 函数（thanks to hash）

  HashSet 中的元素排列顺序跟 add 顺序无关。

  HashSet 中的元素不允许重复（thanks to set）。我们可以用这个特点来实现去重。

  HashSet 中的元素不能通过 index 来访问。

  常用 method：

  1. boolean Add (E e)

True: success. False: e already exist (duplicate)

2. void clear( )

   清除，清空

3. boolean isEmpty( )

   返回是否是空的

4. Iterator<E> iterator( )

   返回一个 hashset 的 iterator。

   iterator 是一种用于遍历数据类型的 class。其功能类似于我们用 for loop 来访问数组。

   但是因为 hashset 不能通过 index 来访问元素，我们需要 Iterator class 来访问。

5. boolean remove(Object o)

   True: success；False：o 这个元素不在这个数据类型中

6. int size( )

   返回 hashset 中元素的数量

定义和使用 HashSet：

```
Set<Integer> set = new HashSet<String>( );          // Set is interface. HashSet
                                                    // implements it
                                                // no primitive type allowed (not int but Integer)
set.Add( 1 );           // insert 1
set.Add( 1 );           // duplicate. ignore.
```

- HashMap

HashMap 储存的是一个 mapping 关系，即关键词-值的数据组（key-value pair）。

key：索引，用来检索数据的。唯一不能重复

data：相应 key 对应的值。可以有重复

常用 method：

1. void clear( )

   清除，清空

2. boolean containsKey(Object key)

   返回是否存在特定的 key

3. boolean containsValue(Object value)

   返回是否存在特定的值

4. Set<Map.Entry<K, V>> entrySet( )

   返回一个 Set 版本的 hashMap

5. boolean equals(Object o)

   比较是否相同（deep comparison）。

6. V get(Object key)

   返回特定 key 对应的 value。如果没有这个 key 则返回 null

7. boolean isEmpty( )

   返回 hashmap 是否为空

8. Set<K> keySet( )

   返回一个包含所有 key 的 set

9. V put (K key, V value)

   如果存在特定 key，则改变其对应的 data 为 value

   如果不存在特定 key，则插入一对新的 key-value pair

   返回 value

10. V remove (Object key)

    删除特定 key 和其相应的 data，并返回这个 data 的值

    如果不存在这个 key，返回 null

11. int size( )

    返回 key-value pairs 的数量

12. Collection<V> values( )

    返回包含所有 values 的 collection（会有重复，所以不用 set）

定义和使用 HashMap：

```
Map<String, Integer> map = new HashMap<String, Integer> ( );
                    // String is key, Integer is value
                    // no primitive type allowed (not int, but Integer)
map.put ("Hello", 0);              // insert ("Hello", 0)
map.put ("Hello", 1);              // now it's ("Hello", 1)
```

- Java generics

当我们定义例如加法的函数是，我们需要写许多不同的 method signature，例如 int 的加法，float 的加法，int 和 float 的加法。。。会有很多重复的代码块。

所以我们可以用 generics 来实现一种通用的数据类型

e.g.

```
public <E> E addition( E a, E b) {
 // < > is a type indentifier (diamond shape). means generic type.
 // E is it's name. You can name it whatever you want
 return a + b;
}
```

为保证有些函数能够正常使用，我们给这个通用类型加一些限制使其称为 subclass 或者一

些 interface 的 implementataion

e.g.

```
public <E extends Comparable> addition (E a, E b)
        // E must be subclass of comparable
```

我们也可以有 generic class（HashSet 等就是这样定义的）：

e.g.

```
public class myClass <T> {
 private T[ ] values;       // generic array
 ...
}

public static void main (String args[ ]) {
 myClass<Integer> myclass = new myClass<Integer> ( );
 ...
}
```

Tutorial Links

- CSE11 的教科书涵盖了更多的例子和练习，课程一般都会要求阅读&做完教科书内的习题。习题比较基础，如果想要提前预习的话可以查看。
  - https://stepik.org/course/100177/syllabus
- 如果想要了解更多基础的 Java 实用案例，可以参考以下的黑马程序员免费提供的 Java 教程。里面囊括了 Java 的下载、安装，
  - https://www.bilibili.com/video/BV17F411T7Ao/?spm_id_from=333.337.search-card.all.click&vd_source=129c4403c26cf9ead6045535883e6bac