**上课时间**

* Java 每周六下午 5点 - 7 点 (太平洋时间) [laioffer.java@gmail.com](mailto:laioffer.java@gmail.com)
* C++ 每周六下午 7点 - 9 点 (太平洋时间) [laioffer.cpp@gmail.com](mailto:laioffer.cpp@gmail.com)

**Meeting ID:**

**===============================================================**

1. Please join my meeting.

[**https://global.gotomeeting.com/meeting/join/163634853**](https://global.gotomeeting.com/meeting/join/163634853)

2. Use your microphone and speakers (VoIP) - a headset is recommended. Or, call in using your telephone.

Dial +1 (571) 317-3112

Access Code: 163-634-853

Audio PIN: Shown after joining the meeting

Meeting ID: 163-634-853

**===============================================================**

[Prerequisite - Install Java and IDE](#h.hu6tjyyrfuvl)

[Class 1 - Hello Java & Array](#h.fsftqogtx81)

[Java “Hello World”](#h.fbw9sxi3rdfy)

[Object-Oriented Paradigm](#h.jsdw2vc5i8t8)

[Class, Object, Reference](#h.l2f9p6r3u4dv)

[“new”](#h.khl92p9jgykk)

[Static vs. non-static](#h.d0iywhuiagml)

[Access Modifiers](#h.pjvp5qm6hpuh)

[Data Types](#h.itghevhyx0vh)

[Array](#h.14qg4zomwyvo)

[Practice Of Homeworks](#h.l5varr4ijb8l)

[Class 2 - Array & List](#h.mxne8cyjzoxv)

[Array vs. Linked List Comparison](#h.6hv41fogihag)

[About Objects, References, null again](#h.ud16e9ac7ocb)

[Pass by value? Pass by reference?](#h.8udmcx1wrp35)

[ArrayList & LinkedList in Java](#h.66yj7e1zg5u0)

[List Interface](#h.yqn9r5ja730c)

[ArrayList](#h.qiyr6zwlsrh0)

[LinkedList](#h.pnej1nb0wl0t)

[Practice Of Homework](#h.9cfosa8dgy8w)

[Class 3 - PriorityQueue](#h.g8dklpyjqoj2)

[null & empty collections, array](#h.ve0h8tsp4mpo)

[Check null and/or empty](#h.1uyj4b31u58c)

[Check corner cases](#h.gf8bpgk10pdd)

[linked list questions, when to use dummy node?](#h.do7sxc5tl1k0)

[PriorityQueue](#h.nlzmun5pdg3)

[Class 4 - PriorityQueue, Queue & Stack Implementation](#h.t9gwwwknc6m7)

[Implement PriorityQueue(Heap)](#h.e4lw0dy7t3s2)

[Definition](#h.uyqg583xrqut)

[Representation](#h.56mk3ddrffhu)

[Basic Heap Internal Operations](#h.y8nkvkdrhgje)

[Queue & Stack](#h.lynakcpaetzg)

[Implementing Queue/Stack.](#h.5ppfyub4d5o0)

[Class 5 - HashMap](#h.usd7xc1wl8gx)

[practice of queue/stack](#h.cqe5u5701wqq)

[Set](#h.exso73wml4u6)

[HashMap](#h.a2li59m89yoi)

[HashSet](#h.r6etfmcczqjz)

[Class 6 - Java Collection & Tree](#h.8axemue5hjxh)

[Data Structures vs. Java Implementations](#h.nx29zymc3eml)

[Java Collections](#h.tk9g9jmwnow2)

[About Arrays and Collections](#h.28fg5z40biax)

[Binary (Search) Tree](#h.o2gv3ljn6l9v)

[Class 7 - Tree ctd.](#h.nwh3p2hns4qt)

[Binary Tree Traversals](#h.myryzdh46kvr)

[Autoboxing and Unboxing](#h.11thdl1gj9ys)

[Class 8 -Bits](#h.klsdjfyfrz6x)

[Code Style](#h.wcmgomc0uh5c)

[Bits](#h.6pfu0qxsd5fk)

[Bit Cheat Sheets](#h.48z76f1x8fss)

[Examples And Practices](#h.y4llfe3iyq1)

[Practice DFS](#h.e33epp70iyd4)

[Class 9 1-2 class review](#h.okc30a1ol9um)

[Java与C/C++的对比](#h.s6e71587aqtt)

# Prerequisite - Install Java and IDE

If you already have any Java SDK and any IDE(for example, NetBeans, eclipse or IntelliJ IDEA) installed on your machine, you can skip this section.

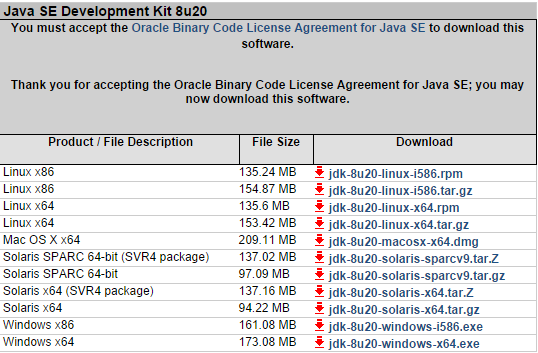
Note: The minimum required version of Java is 1.5(Java 5).

**1. Download JDK**

Note: If you are using Mac OS X, this step is not required since Java is pre-installed.

<http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>

Select appropriate option (32 bit/ 64 bit, windows/linux/Mac OS X)



**2. Install JDK (All default options).**

**3. Download and Install Eclipse.**

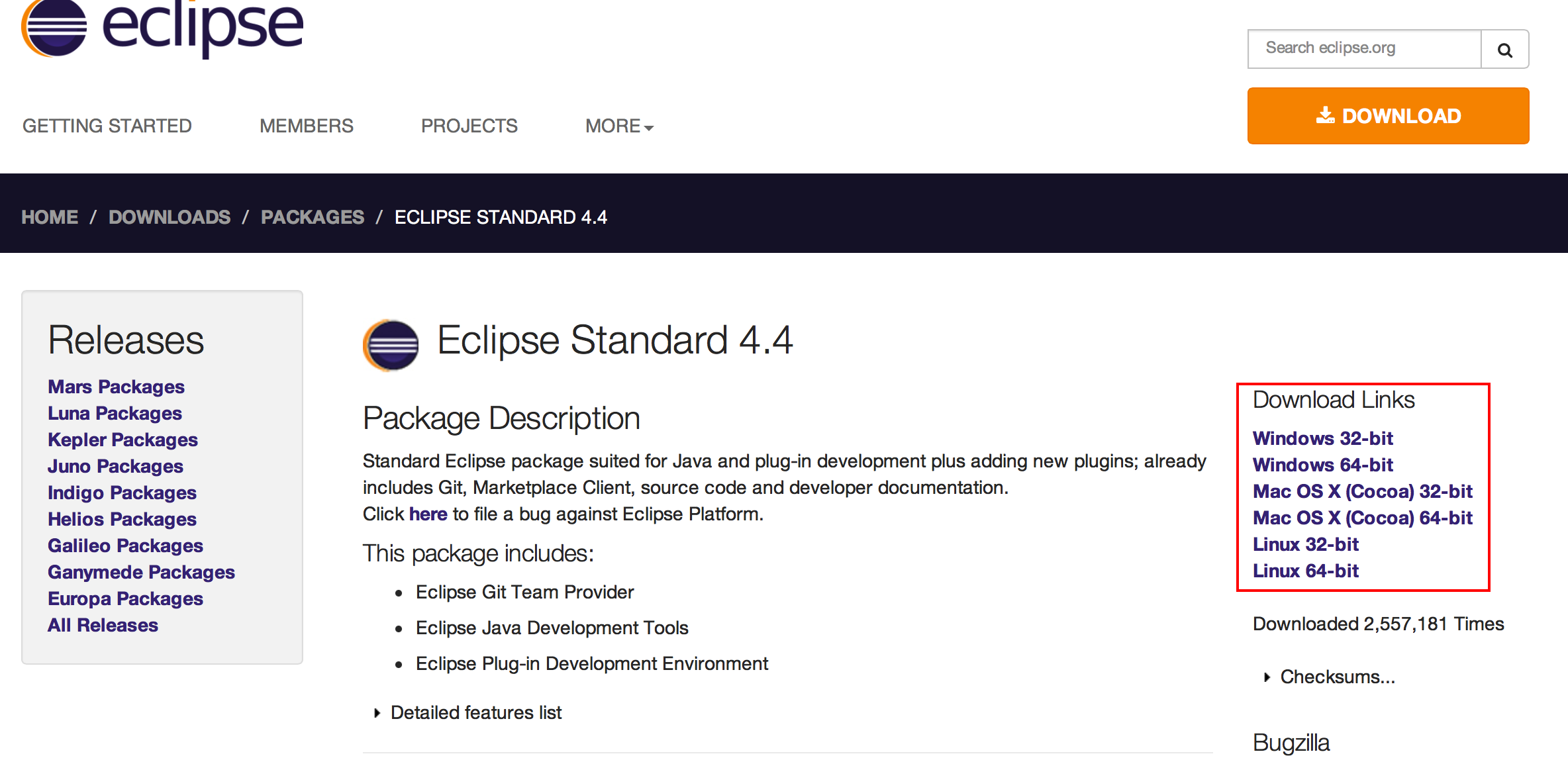
The eclipse Lunar package is at:

<https://www.eclipse.org/downloads/packages/eclipse-ide-java-developers/lunasr1>

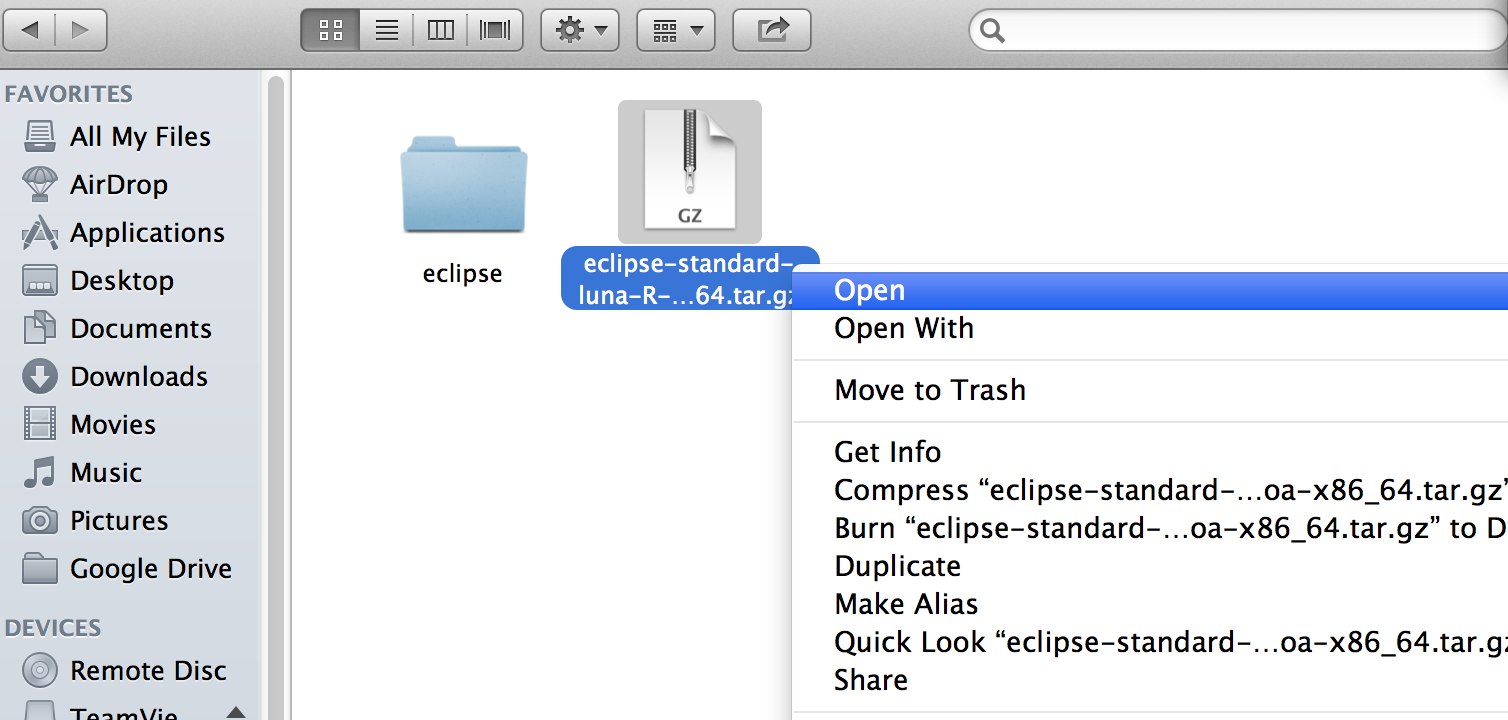
(lunar package is least version providing Java 8 support, it is required if you are using Java 8.)

The download links are at the right corner of the page, please choose the download link according to your system environment.

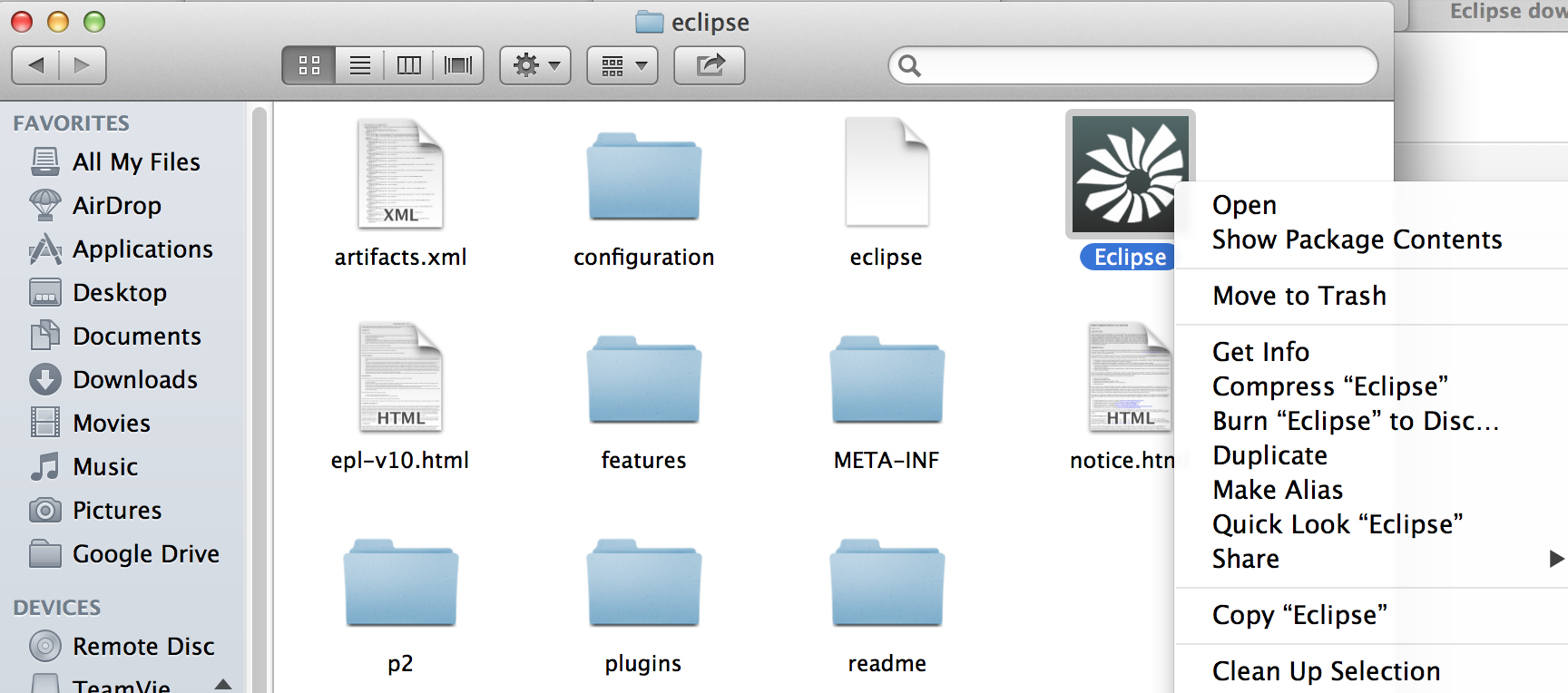
(For example, I am using 64 bit Mac OS X, so the one I need is [**Mac OS X (Cocoa) 64-bit**](http://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/luna/SR1/eclipse-java-luna-SR1-macosx-cocoa-x86_64.tar.gz))



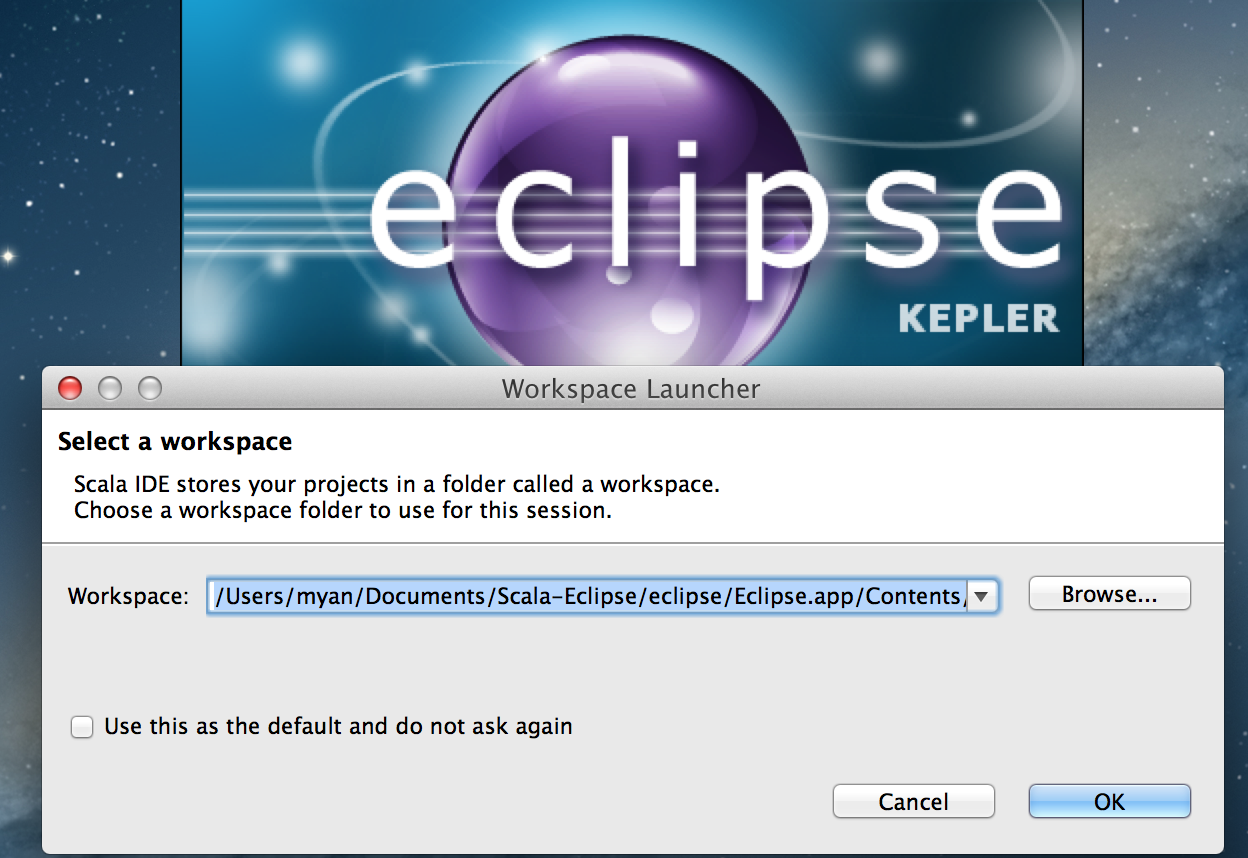
The downloaded package is a compressed file, either in zip or gz format, please extract the file into any directory that is convenient for you.



There will be an executable file available in the “eclipse” directory and you can start using eclipse by running it directly.

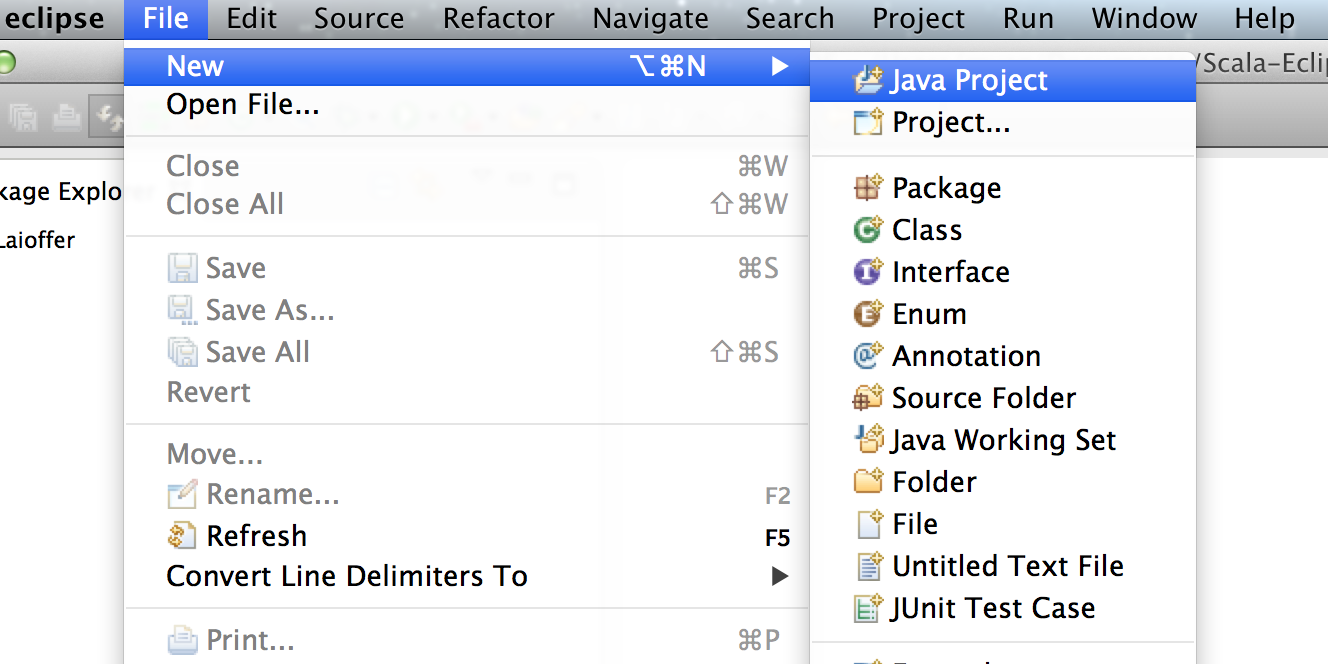


When starting eclipse, it will ask for your workspace directory, you can choose any directory as your workspace.



**4. Check if the IDE is installed correctly by running a sample Java project.**

Create a new Java project.

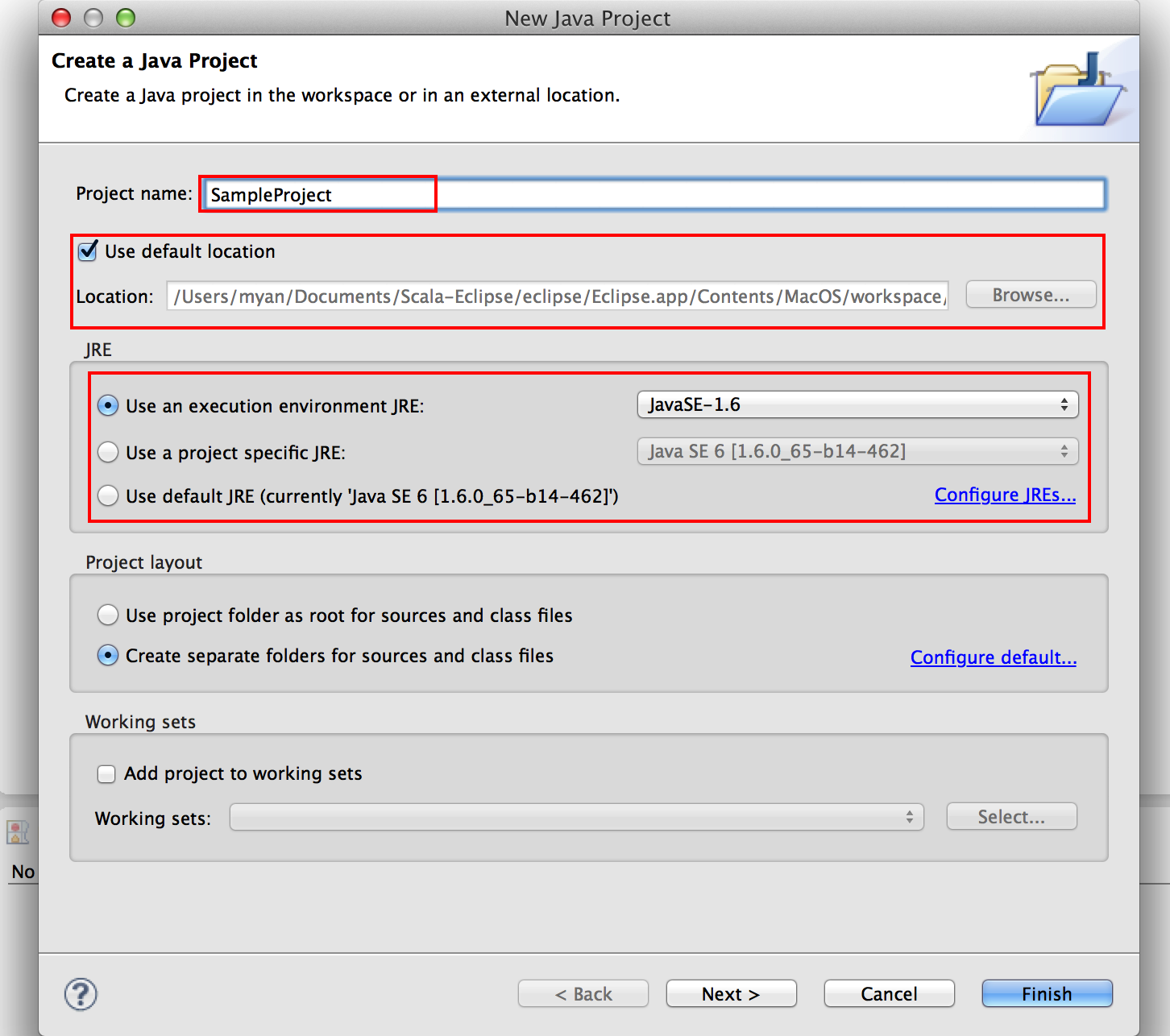


1). Give a name to the new Java project

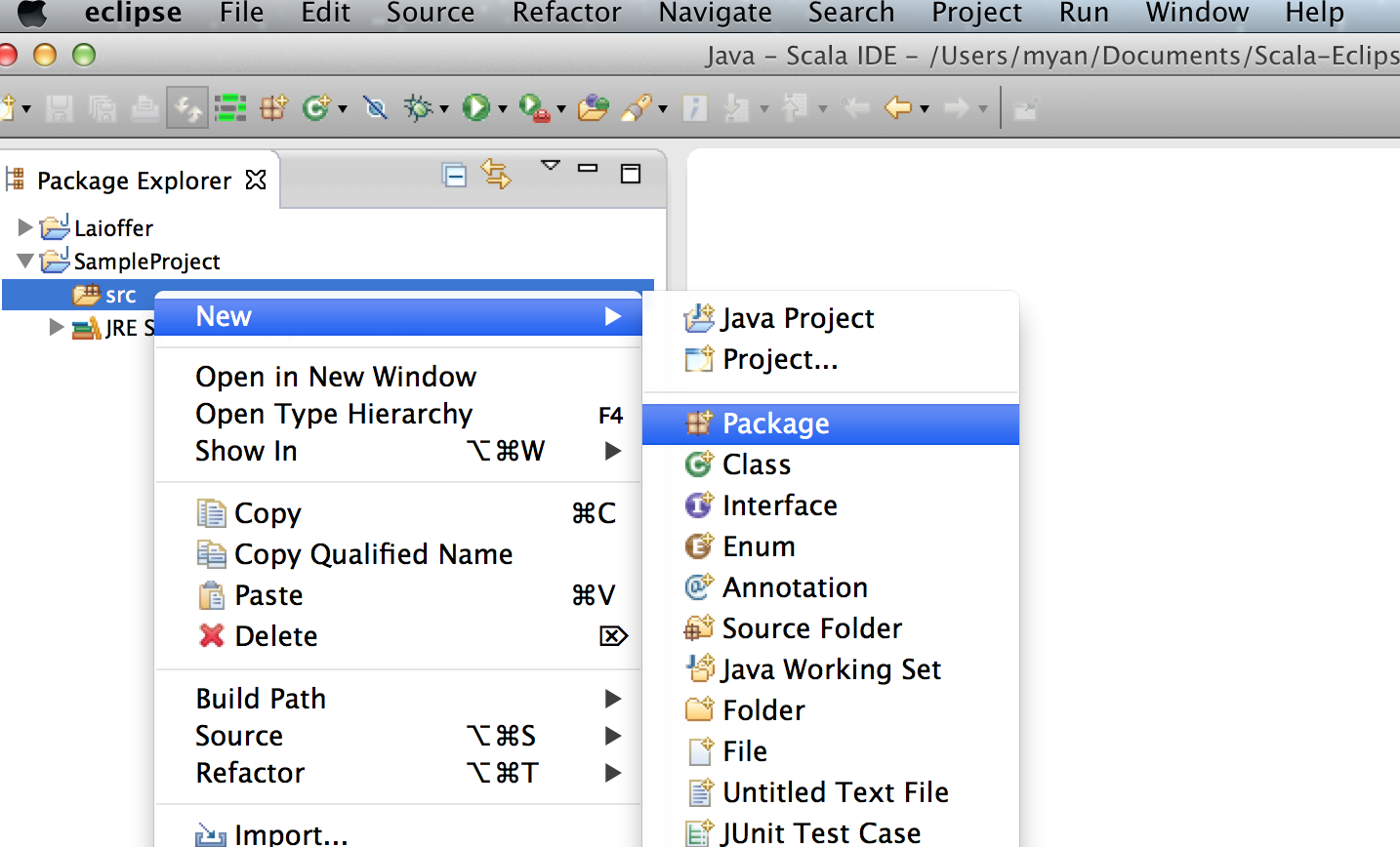
2). You can specify a custom directory for your project location

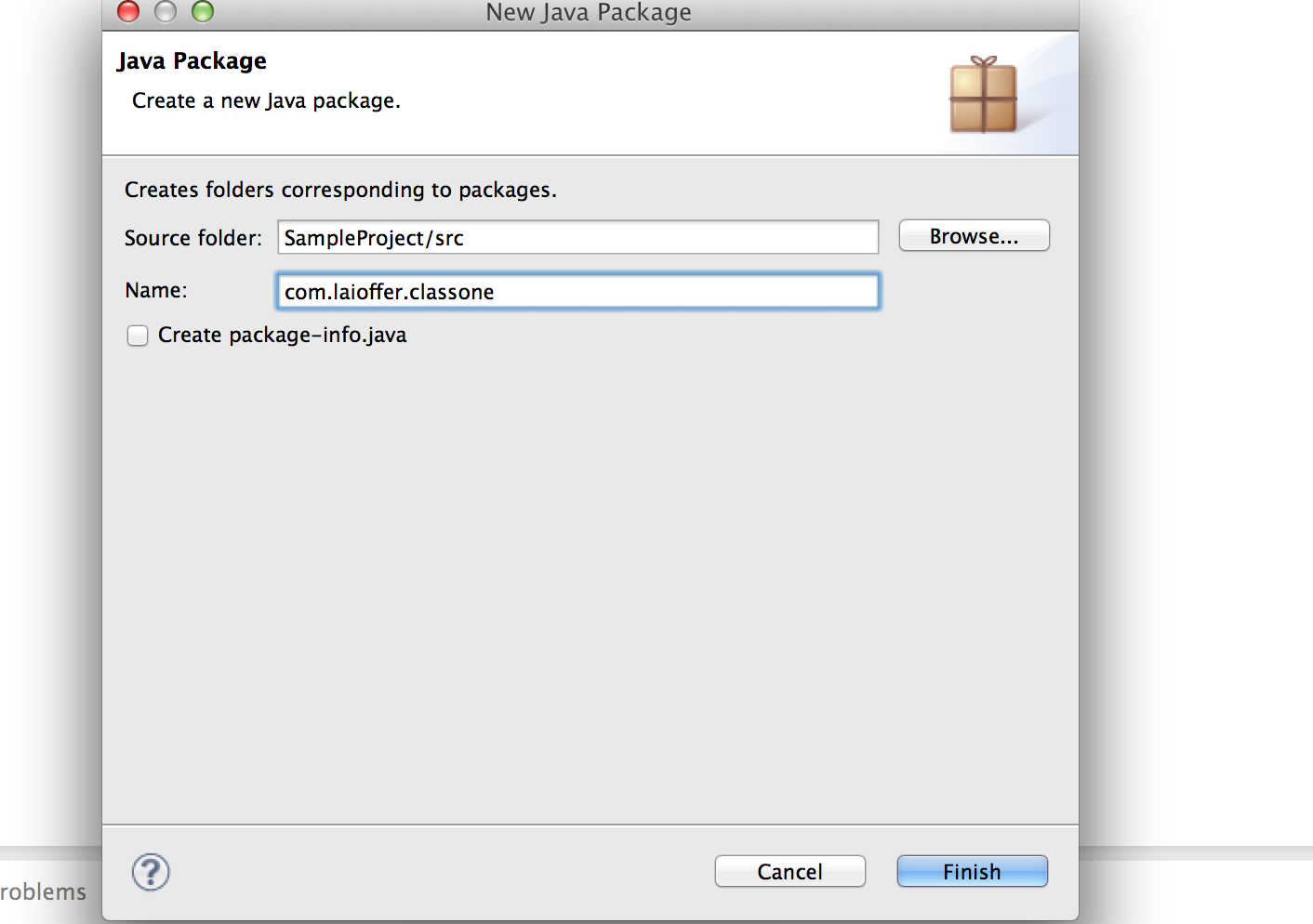
3). Make sure there is a valid JRE selected

4). Click “Finish” to let eclipse create the project for you

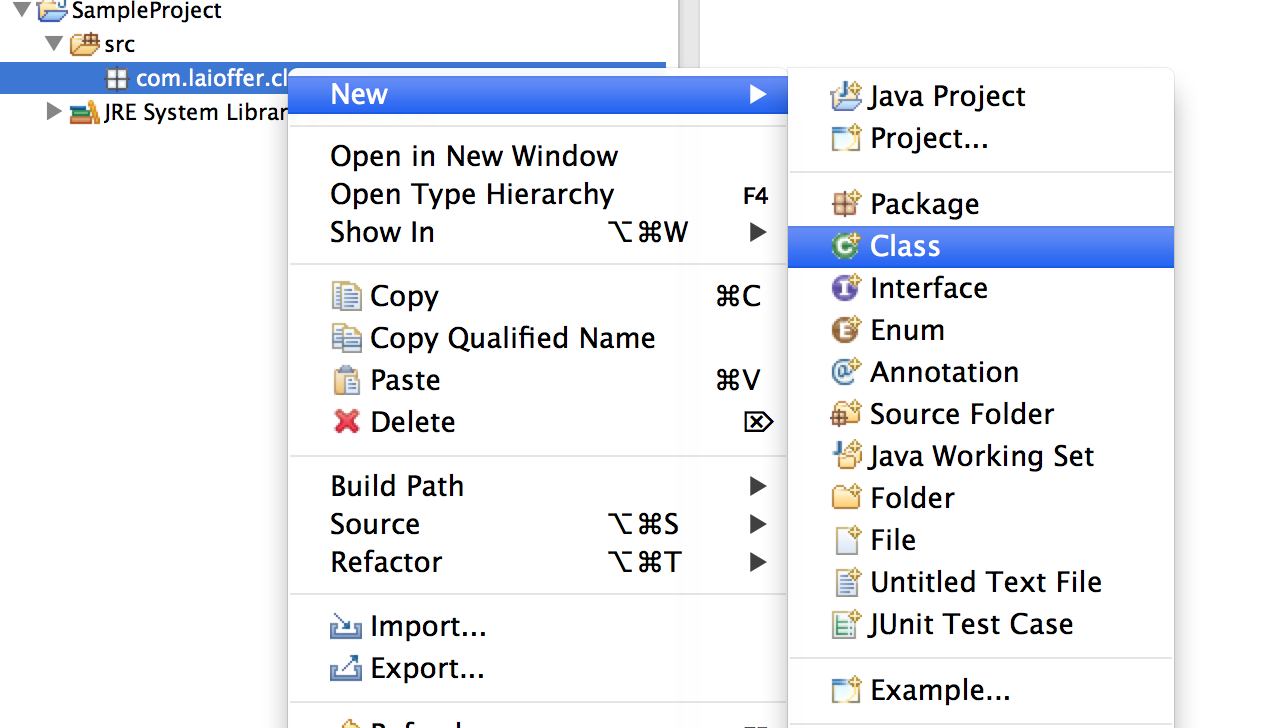


Right click the “src” folder under your newly created Java project, and choose new “Package”, then give a name for the package, for example, “com.laioffer.classone”.

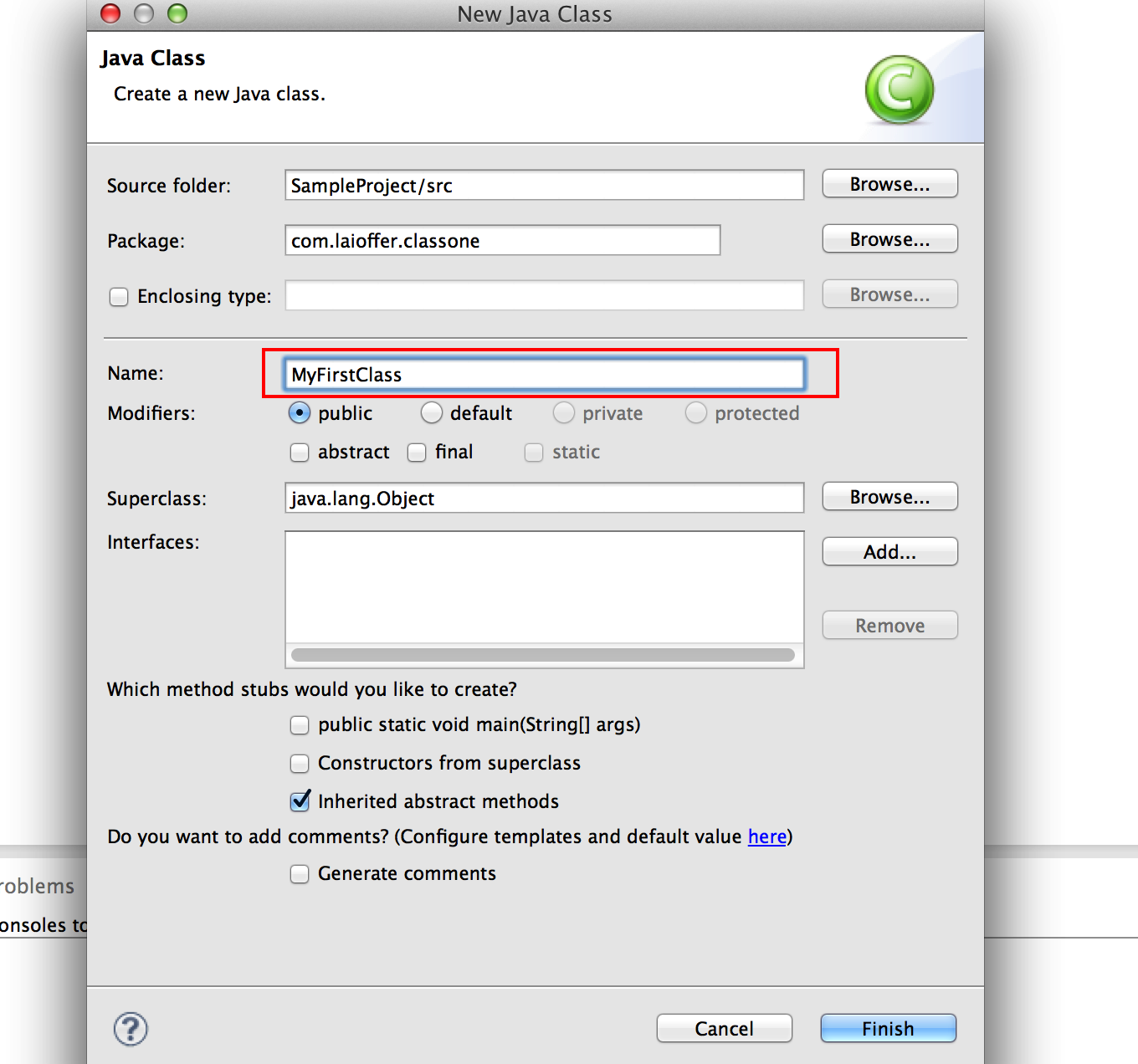




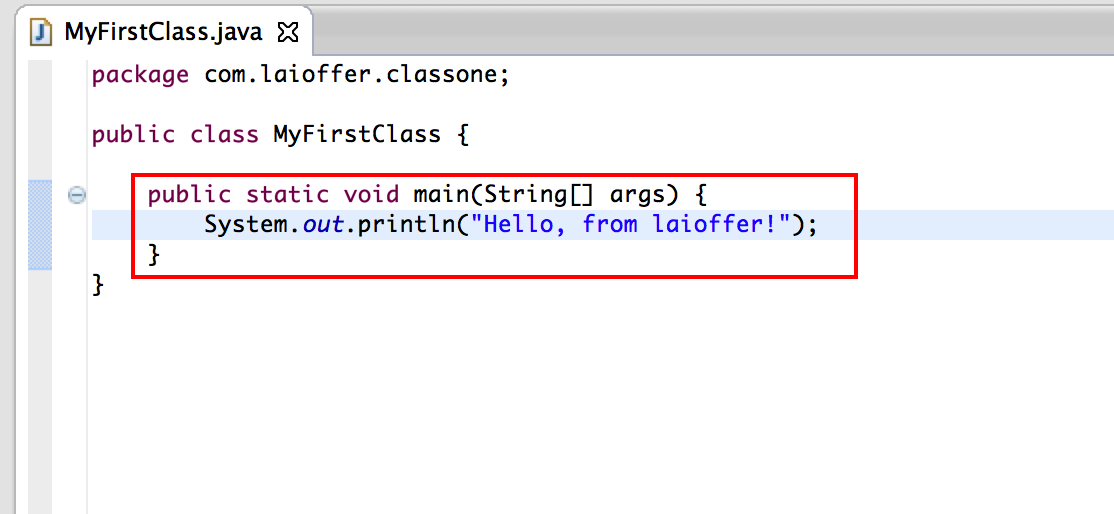
Create the first class for this Java project.



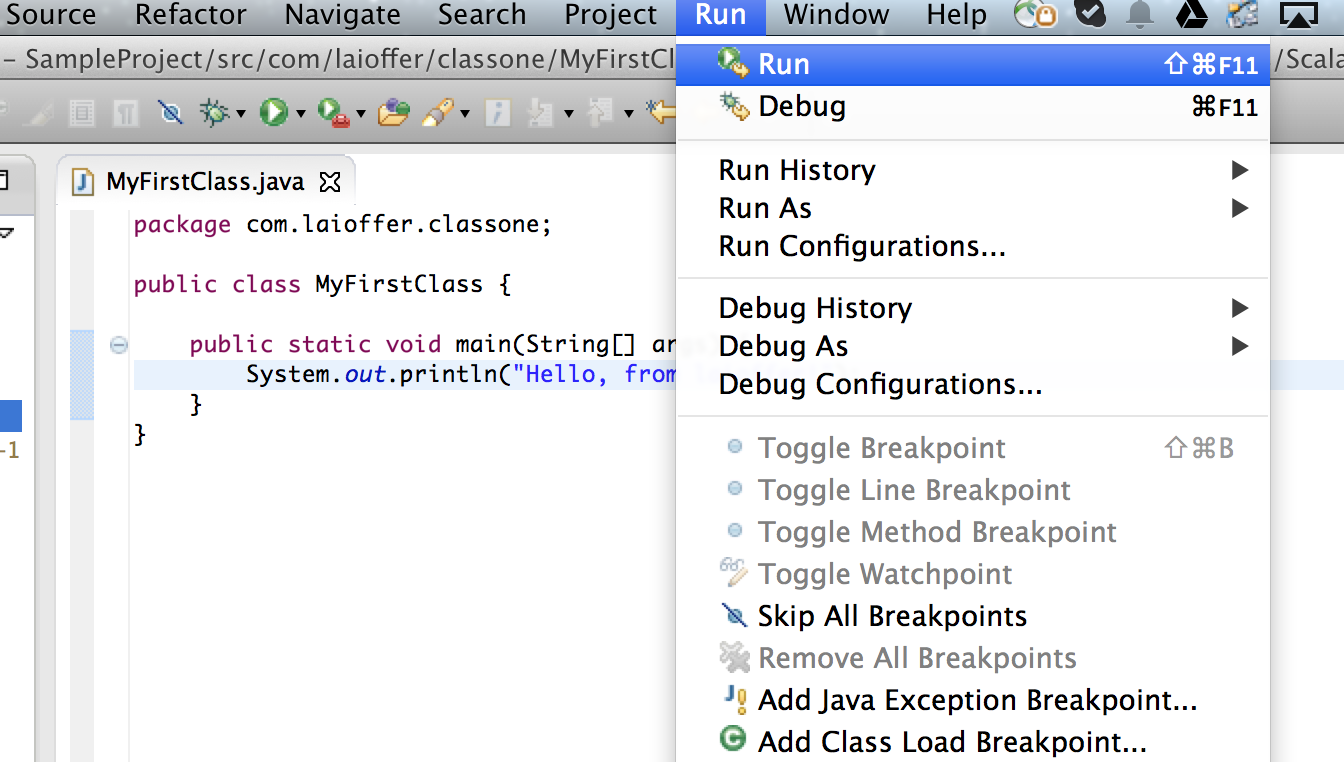
Give a name to your new class.



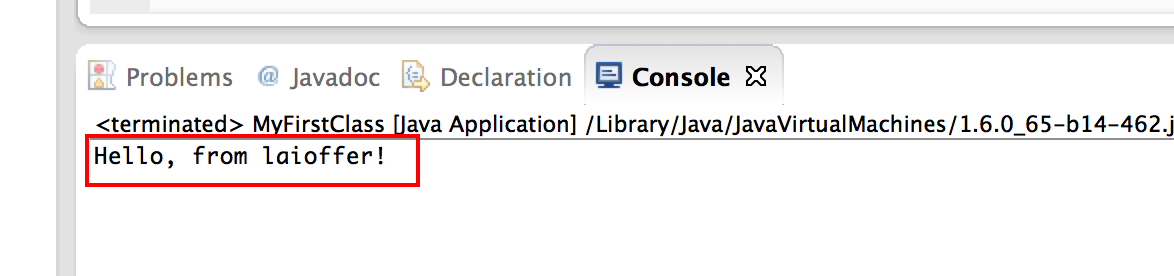
Make the class “Hello World”.



And run it !



If you see the following message from console, congratulations!



**4. If something goes wrong…**

If there is anything wrong during the installation steps or running your first Java project, please feel free to let me know(an email with screenshot of your problem sent to [laioffer.java@gmail.com](mailto:laioffer.java@gmail.com) would be sufficient :)).

# 

# Class 1 - Hello Java & Array

**Why** do we have this class?

- 对编程语言的理解和熟悉程度

- 必须要掌握的概念和思想

- 编程方式和技巧

- 正确的格式

对于Java, 我至少需要那些知识?

什么样的代码才能达到面试官的要求?

**What** do you expect from this class?

- learn how to write Java code

- basic data structures

- 概念, 语言的特性, 编程思想

- 编程过程中常用的API

- 常见面试中的考点

- FAQ & Homeworks

Do you **Need** to attend this class?

**Q & A**

??

## Java “Hello World”

public class HelloWorld {

public static void main(String[] args) {

System.out.println("Hello World!");

}

}

what can we learn from HelloWorld?

* Everything starts from a class - Java is Object-Oriented Programming Language!
  + what is class?
* public static void main(String[] args) is the main entrance of Java program.
  + What is “static” ?
  + Why the main is “static” ?

public class AnotherHelloWorld {

private String to;

public AnotherHelloWorld(String to) {

this.to = to;

}

public void sayHello(String from) {

System.out.println("Hello World! From: " + from + " To: " + to);

}

public static void main(String[] args) {

String to = "Jack";

String from = "Laioffer";

AnotherHelloWorld helloWorld = new AnotherHelloWorld(to);

helloWorld.sayHello(from);

}

}

what are the differences?

* construct an object of the class AnotherHelloWorld in the main method
  + what is object?
  + how to construct an object?
* call the object’s sayHello() method
  + field vs. method

### 

### Object-Oriented Paradigm

Everything in the world is object

* each object has a type.
* each object is defined by two kinds of information
  + state: the “field”s - what things it maintains
  + behavior: the “method”s - what it can do

In programming world, software objects are often used to abstract and model the objects in the real world to solve the problems.

### Class, Object, Reference

**class** - is like the blueprint for a house. Using this blueprint, you can build as many houses as you like.

**object** - each house you build (or instantiate, in OO lingo) is an **object**, also known as an **instance**.

**reference** - each house also has an address, of course. If you want to tell someone where the house is, you give them a card with the address written on it. That card is the object's **reference**.

If you want to visit the house, you look at the address written on the card. This is called **dereferencing**.

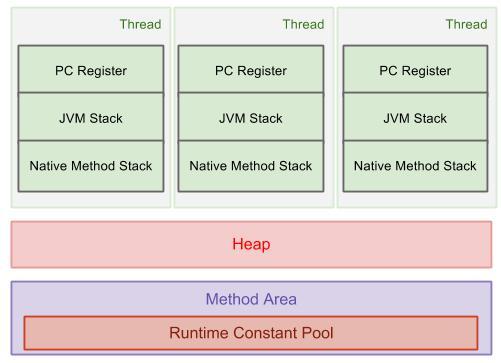
AnotherHelloWorld helloWorld = new AnotherHelloWorld(to);

* “AnotherHelloWorld” is the class
* “helloWorld” is a reference of “AnotherHelloWorld” class
* “new AnotherHelloWorld(to)” will create a new object of “AnotherHelloWorld” class, and “helloWorld” is referenced to the object.

**null,** means the reference is not associated with any object**.**

### “new”

Java Memory Areas:



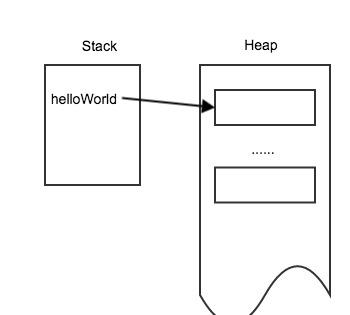
JVM Stack - frames, local variables, controlling the method invocation.

Heap - all the objects, with garbage collection mechanism.

Method Area - methods, constructors, class meta data.

“new”

* all the objects are allocated on Heap
* “new” means allocate memory on Heap, create the object, and return the reference of the object



### Static vs. non-static

class level vs. object level state.

Another more complicated HelloWorld…

public class ComplicatedHelloWorld {

private static int numOfInstances = 0;

private String to;

public ComplicatedHelloWorld(String to) {

this.to = to;

numOfInstances++;

}

public void print(String from) {

System.out.println("Hello, from " + from + " to " + to);

}

public static void printNumInstances() {

System.out.println("Totally " + numOfInstances + " Objects Created!");

}

public static void main(String[] args) {

ComplicatedHelloWorld helloWorld = new ComplicatedHelloWorld(args[0]);

helloWorld.print("laioffer");

printNumInstances();

helloWorld = new ComplicatedHelloWorld(args[1]);

helloWorld.print("laioffer");

printNumInstances();

}

}

what are the differences?

* (String[] args) - takes the command line arguments as an array of Strings
* “numberOfInstances” is static field - it is associated with the class rather than associated with the instances.

### Access Modifiers

public - everyone can access

private - only myself can access

protected - only my children and same package can access

default - only the same package can access

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Modifier | Class | Package | Subclass | World |
| public | Y | Y | Y | Y |
| protected | Y | Y | Y | N |
| no modifier | Y | Y | N | N |
| private | Y | N | N | N |

### 

### Data Types

Generally, two types of data

* primitive data types - the smallest granularity of representing the state
* object types - objects

|  |  |  |  |
| --- | --- | --- | --- |
| **Data Type** | **Default Value (for fields)** | **Bits:** | **Range:** |
| byte | 0 | 8 | [-27, 27 - 1] |
| short | 0 | 16 | [-215, 215 - 1] |
| int | 0 | 32 | [-231, 231 - 1] |
| long | 0L | 64 | [-263, 263 - 1] |
| float | 0.0f | 32 |  |
| double | 0.0d | 64 |  |
| char | '\u0000' | 16(Unicode) | [0, 216 - 1] |
| boolean | false | unknown? | [false, true] |
| String(or any object) | null |  |  |

### Array

int[] array = new int[3];

int[] array = {1, 2, 3};

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

int array[] ? → array can be declared this way, but is strongly not recommended for readability and easier maintenance.

**two-dimensional array**: an array of arrays, each of the arrays could has different length.

int[][] matrix = new int[5][2];

int matrix[][]

int[] matrix[]

int[][] matrix = {{1,2}, {1,2}, {1,2}, {1,2}, {1,2}};

int[][] matrix = new int[5][];

for (int row = 0; row < 5; row++) {

matrix[row] = new int[row];

}

How to access array:

int[] array = {1, 2, 3};

0, 1, 2

int value = array[1]; → value = 2

How to get the length of the array?

int length = array.length;

for (int index=0; index<array.length; index++) {

System.out.println(array[index]);

}

Notice: ArrayIndexOutOfBoundException!

If the index specified in not in the valid range of the array.

for (int number : array) {

System.out.println(number);

}

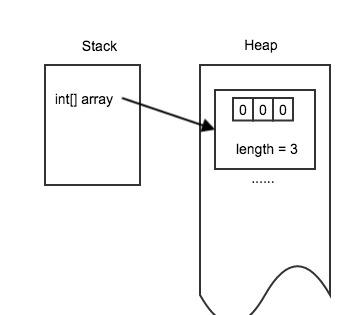
int[] array = null;

array[1]; // NullPointerException.

What are arrays in Java? How are they stored in memory?

* arrays are Objects
* I can not find the class for arrays…
  + True, the class declaration is not visible, but there still exists a class inside JVM.
  + each of the arrays created is an instance of that class.
  + remember “new” - it implies that there is a new object created.
* the elements in the array still occupy consecutive memory space on the heap
* “length” is a field of the array object
* array length can not be changed after the array object is created.

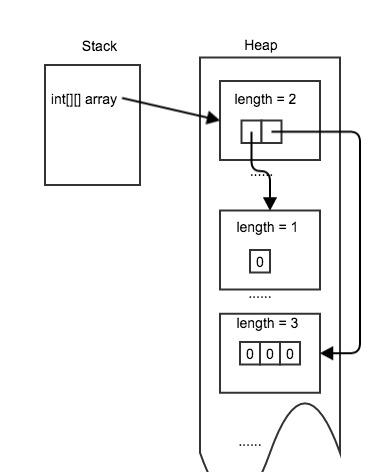
int[] array = new int[3];



int[][] array = new int[2][];

array[0] = new int[1];

array[1] = new int[3];



## Practice Of Homeworks

Selection Sort

[solution](https://docs.google.com/document/d/1Oaq3kQMCZNZZvAcgkN9kgAzmDRfGo3KQI-r2Yf0aRQ4/edit#heading=h.utzx1tfrud6)

Merge Sort

[solution](https://docs.google.com/document/d/1Oaq3kQMCZNZZvAcgkN9kgAzmDRfGo3KQI-r2Yf0aRQ4/edit#heading=h.yw5bu9aajbry)

Quick Sort

[solution](https://docs.google.com/document/d/1Oaq3kQMCZNZZvAcgkN9kgAzmDRfGo3KQI-r2Yf0aRQ4/edit#heading=h.hlj84p2tts46)

# Class 2 - Array & List

## Array vs. Linked List Comparison

* Memory Layout
  + Array: consecutive allocated memory space, no overhead
  + Linked List: non-consecutive, overhead of multiple objects with the “next” reference
* (Random) access time? - get ith element
  + Array: O(1)
  + Linked List: O(n) worst case
* Search time (non-sorted)?
  + Array: O(n)
  + Linked List: O(n)
* Search time (sorted)
  + Array: O(logn)
  + Linked List: O(n)

Examples of Linked List Operations:

int length(ListNode head) {

int length = 0;

while (head != null) {

length++;

head = head.next;

}

return length;

}

1 → 2 → 3 → null

ListNode get(ListNode head, int index) {

while (head != null && index > 0) {

head = head.next;

index--;

}

return head;

}

1 → 2 → 3 → null

0 1 2

ListNode appendHead(ListNode head, int value) {

ListNode newHead = new ListNode(value);

newHead.next = head;

return newHead;

}

1 → 2 → 3 → 4 → null

p

ListNode appendTail(ListNode head, int value) {

ListNode newNode = new ListNode(value);

if (head == null) {

return newNode;

}

// we know head is not null here, and we need to know the last non-null node so that we

// can append after it. so we can use while(head.next != null)

ListNode prev = head;

while (prev.next != null) {

prev = prev.next;

}

// now prev is the last non-null node.

prev.next = newNode;

return head;

}

**head could be changed - we need to return the new head if changed**

**usually need to consider**

* **head == null**
* **head.next == null**

**while loop - we need to understand what is the terminate condition**

**use head/head.next in the condition**

ListNode search(ListNode head, int value) {

while (head != null) {

if (head.value == value) {

return head;

}

head = head.next;

}

return null;

}

ListNode remove(ListNode head, int value) {

// 1.

if (head == null) {

return head;

}

// 2.

if (head.value == value) {

return head.next;

}

// 3.

ListNode prev = head;

while (prev.next != null && prev.next.value != value) {

prev = prev.next;

}

if (prev.next != null) {

prev.next = prev.next.next;

}

return head;

}

1 → 2 → 3 → null

p

## 

## About Objects, References, null again

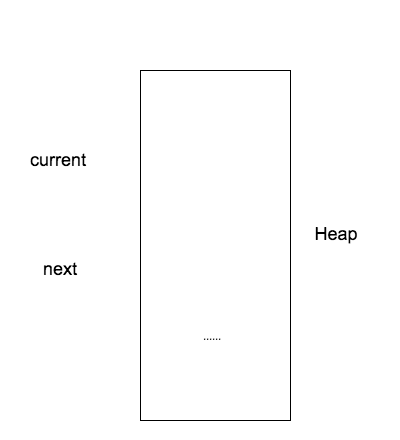
null - the reference is not pointed to any Object on the Heap.

* a special value for reference

…

ListNode current = null;

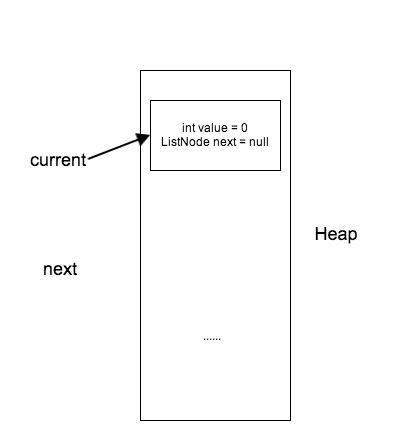
ListNode next; // initialize to null



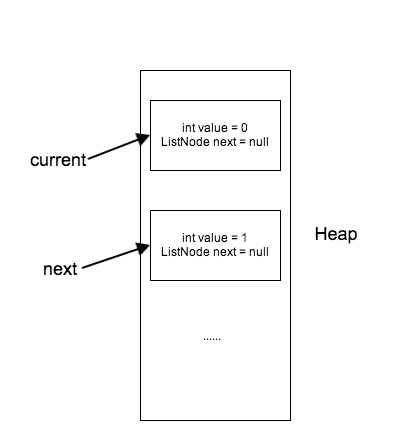
current = new ListNode(0);

// 1. allocate space on heap and create a new ListNode object with {value = 0

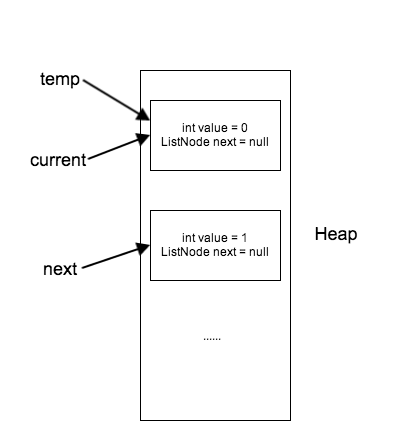
// next = null}  
 // 2. return the object’s memory address as the value of reference.



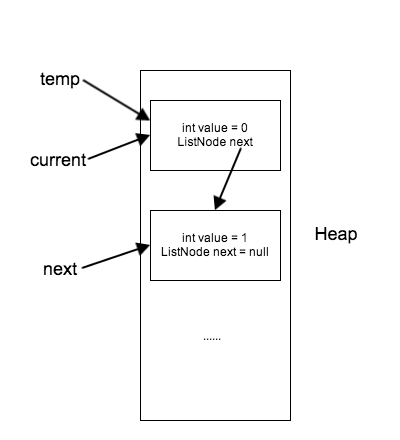
next = new ListNode(1);



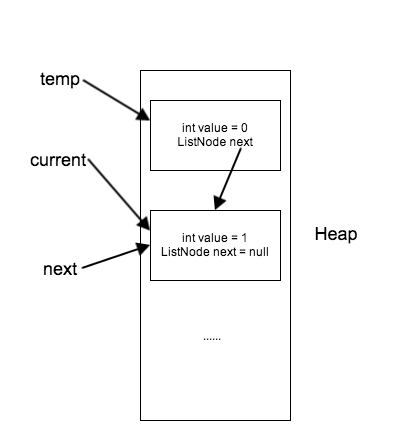
ListNode temp = current;



current.next = next;



current = current.next;



### 

## Pass by value? Pass by reference?

primitive type - pass by value

* primitive type are always value

public void testPassByValue() {

int value = 0;

changeValue(value);

System.out.println(value); // What is the expected output here? 0

}

public void changeValue(int value) {

value = 1;

}

// What if you want to change the value in above example?

public void testPassByValue() {

int value = 0;

value = changeValue(value);

System.out.println(value); // What is the expected output here? 1

}

public int changeValue(int value) {

value = 1;

return value;

}

**Pass by value** - make a copy of the value of the original variable, and pass the copy to the method as argument.

object type - pass by reference?

* what does it mean by pass by reference?
* what is “reference”?

Example 1:

public void testPassByRef() {

ListNode node = new ListNode(0);

System.out.println(node.value);

change(node);

System.out.println(node.value); // What is the expected value here? 1

}

public void change(ListNode node) {

node.value = 1;

}

Example 2:

public void testPassByRef() {

ListNode node = new ListNode(0);

System.out.println(node.value);

change(node);

System.out.println(node.value); // what is the expected value here? 0

}

public void change(ListNode node) {

node = new ListNode(1);

}

node → ListNode{value:0, next:null}

node’ → ListNode{value:1, next:null}

In Example 1,

the original reference is copied and the copy of the original reference is pointing to the same object, the copy is passed to the method as argument;

In the method, it changes the content of the object pointed by the reference.

In Example 2,

the original reference is copied and the copy of the original reference is pointing to the same object, the copy is passed to the method as argument;

In the change() method, what we change is the copy, we point the copy to another object, but this change only has the scope inside the method, so the original reference is not changed.

reference - it is actually a value(you can think of it as a int type), the value is the memory address of the pointed object(int type)

How can we modify Example 2 to make the “node” pointing to the new ListNode?

public void testPassByRef() {

ListNode node = new ListNode(0);

System.out.println(node.value);

node = change(node);

System.out.println(node.value); // what is the expected value here?

}

public ListNode change(ListNode node) {

node = new ListNode(1);

return node;

}

**Java is always “pass by value”**

## ArrayList & LinkedList in Java

### List Interface

* An ordered collection (sometimes called a *sequence*).
* Lists can contain duplicate elements.

The user of a List generally has precise control over where in the list each element is inserted and can access elements by their integer index.

Provided functionalities(interfaces)

* Random Access:

set(int index)

get(int index)

add(int index, E e), add(E e)

remove(int index)

* Search
* Iterator
* Range-View
* Other useful methods:

isEmpty();

size();

Most popular implementation class: ArrayList, LinkedList.

### ArrayList

ArrayList is regarded as a resizable array.

* maintain an array with potential unused cells
* will expanded the array if there is no unused cells available
  + by replacing the maintained array with a new larger array (1.5 times larger by default)

class ArrayList<E> {

E[] array; // the maintained array, the current maximum capacity is array.length;

int size; // the number of actually used cells in the maintained array

…...

}

get(int index): array[index]

set(int index, E e): array[index] = e

size(): size // size means the number of used cells, not the capacity.

when call add(), remove(), size is modified as well.

Example:

array = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}, size = 10, capacity = 10(array.length).

call add(11):

1. check if it is full size == capacity
2. if it is full, create a new array1, array1’s size = 1.5 \* array’s size, copy all the element in array to array1, and change the maintained array to array1.

array1 = new Integer[size \* 1.5]

array1 = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, null, null, null, null, null}

size = 10, capacity = 15

array = array1

1. set the 11th element’s value to 11, size++.

array = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, null, null, null, null}

size = 11, capacity = 15

What if you call add(int index, E e)?

1. expand if necessary
2. then right move all the elements after index by 1,
3. set the element at index.

Example:

array = {1, 3, 4, 5, 6, null, null, null, null, null}, size = 5, capacity = 10

call add(1, 2) - add Integer 2 at index 1

1. do not need to expand since size < capacity
2. move all the elements of {3, 4, 5, 6} right by 1 position, size++

array = {1, null, 3, 4, 5, 6, null, null, null, null}, size = 6, capacity = 10

1. set index 1 as Integer 2

array = {1, 2, 3, 4, 5, 6, null, null, null, null}

call remove(1) - remove the element at index 1

1. check if empty first.
2. move all the elements of {3, 4, 5, 6} left by 1 position, size--

ArrayList<Integer> list = new ArrayList<Integer>();

list.size(); // 0

list.add(1); // add at end, {1}

list.size(); // 1

list.add(0, 2); // add at head, {2, 1}

list.size(); // 2

list.add(1, 3); // add at middle, {2, 3, 1}

list.get(1); // 3

list.remove(1); // {2, 1}

list.set(0, 4); // {4, 1}

### LinkedList

backend by a doubled linked list, wrap the size, head and tail information.

class ListNode<E> {

E e;

ListNode<E> prev;

ListNode<E> next;

}

class LinkedList<E> {

ListNode<E> head;

ListNode<E> tail;

int size;

}

get(int index) - from the head/tail, traverse the list to get the element at index

set(int index, E e) - from the head/tail, traverse the list to the get the element at index, and change it

add()

* at tail - append new node after tail and update tail to the new added node
* at head - insert in front of the head, and update head to the new added node
* at middle -

remove()

* at head - update the head to head.next
* at tail - update the tail to tail.prev
* at middle - from head/tail, traverse the list to get the element at index, and remove it.

size()

isEmpty()

LinkedList<Integer> list = new LinkedList<Integer>();

list.size(); // 0

list.add(1); // add at end, {1}

list.size(); // 1

list.add(0, 2); // add at head, {2, 1}

list.size(); // 2

list.add(1, 3); // add at middle, {2, 3, 1}

list.get(1); // 3

list.remove(1); // {2, 1}

list.set(0, 4); // {4, 1}

Time Complexity Analysis:

|  |  |  |
| --- | --- | --- |
| Operation | ArrayList | LinkedList |
| get(int index) at head/tail | O(1) | O(1) |
| get(int index) in middle | O(1) | O(n) |
| set(int index) at head/tail | O(1) | O(1) |
| set(int index) in middle | O(1) | O(n) |
| add(int index, E e) in middle | O(n) | O(n) |
| add(int index, E e) at head | O(n) | O(1) |
| add(E e) at tail | worst O(n)  amortized O(1) | O(1) |
| remove(int index) at head | O(n) | O(1) |
| remove(int index) at tail | O(1) | O(1) |
| remove(int index) at middle | O(n) | O(n) |
| size() | O(1) | O(1) |
| isEmpty() | O(1) | O(1) |

***When to choose what? ArrayList or LinkedList?***

1. If you need stack/queue, use LinkedList (or ArrayDeque, which is even better)
2. If you have a lot of random access operations, use ArrayList.
3. If you always add/remove at the end, use ArrayList
   1. When the time complexity is similar for using ArrayList and LinkedList, use ArrayList. (there is extra memory cost for LinkedList since each ListNode has to maintain the prev/next references, and memory space has to be allocated for each node in the list.)
4. Stack and Vector class are not recommended, whenever you need a Vector, use ArrayList, whenever you need a Stack, use LinkedLIst.

## Practice Of Homework

**Binary Search Variant 1.3**

public int closest(int[] array, int target) {

if (array == null || array.length == 0) {

return -1;

}

int left = 0;

int right = array.length - 1;

// remain two elements, the two elements closest to target,

// largest smaller than target, smallest larger than target

// or two smaller closest / two larger closest

while (left < right - 1) {

int mid = left + (right - left) / 2;

if (array[mid] == target) {

return mid;

} else if (array[mid] < target) {

left = mid;

} else {

right = mid;

}

}

// left >= right -1, possible array.length == 1 (left == right)

if (Math.abs(array[left] - target) <= Math.abs(array[right] - target)) {

return left;

}

return right;

}

# Class 3 - PriorityQueue

## null & empty collections, array

They are DIFFERENT!

null - there is no array object associated with the reference != there is an array object, but the array object does not contains anything(length == 0).

null - there is no list object associated with the reference != there is an list object, but the list does not contains anything(size == 0).

**Example 1**

// the reference “list” is not pointed to any object at all.

ArrayList<Integer> list = null;

// NullPointerException will be thrown for all below calls,

// because there is no object.

list.isEmpty();

list.size();

list.get(0);

**Example 2**

// the reference “list” is pointed to one ArrayList object,

// but the ArrayList object does not contain anything.

ArrayList<Integer> list = new ArrayList<Integer>();

// these two calls will be fine.

list.isEmpty();

list.size();

// IndexOutOfBoundException (index >= size)

list.get(0);

**Example 3**

// the reference “array” is not pointed to any object at all.

int[] array = null;

// NullPointerException will be thrown for all below calls,

// because there is no object.

array[0];

**Example 4**

// the reference “array” is pointed to one array object,

// but the array object does not contains anything.

int[] array = new int[0];

// IndexOutOfBoundException (index >= size), but

// there is an array object.

array[0];

### Check null and/or empty

public void reverse(ArrayList<Integer> list) {

// null check has to be prior to any other checks.

// the first check is to check if there is no list object.

if (list == null) {

...

}

// the second check is to check if the list object does not contains anything.

// they have different meanings!

if (list.isEmpty()) {

...

}

// using “OR” to put the check together, null check has to be the first one

// the sequence of examining the logical expressions is from left to right.

// if null check is not the leftmost one, it is possible to throw NullPointerException.

if (list == null || list.isEmpty()) {

...

}

}

public void reverse(int[] array) {

if (array == null || array.length == 0) {

...

}

}

### 

### Check corner cases

Check null

Check length/size

Check index out of bound

What is the termination condition of for/while loop?

Example

public ListNode middle(ListNode head) {

// TODO fill in the solution.

// 1. check null

if (head == null) {

return head;

}

// 2.

// Tips: make some small examples to see what it should be like

// 1 -> null

// 1 -> 2 -> null

// 1 -> 2 -> 3 -> null

ListNode slow = head;

ListNode fast = head;

while (fast.next != null && fast.next.next != null) {

// fast forward two steps each time, so we need to guarantee two nodes not null

fast = fast.next.next;

slow = slow.next;

}

return slow;

}

A few points

* The null/length/out of bound check is always prior to other checks.

if (index > 0 && array[index - 1] > 0)

* Make a concrete example to see
  + what you want to guarantee within the loop.
  + when you want to stop the loop.

**Unknown sized binary search**

public class UnknownSizeBinarySearch {

/\*

\* Wrapper class for an unknown sized int array. The length() method is not

\* provided to outside the class.

\*/

public static class Dictionary {

private int[] array;

public Dictionary(int[] array) {

this.array = array;

}

public Integer get(int index) {

if (array == null || index >= array.length) {

return null;

}

return array[index];

}

}

public int unknownSizeBinarySearch(Dictionary dictionary, int target) {

if (dictionary == null) {

return -1;

}

int left = 0;

// initialize right index as 2 ^ 0 - 1

int right = 0;

while (dictionary.get(right) != null && dictionary.get(right) < target) {

// 1. move left to right,

// 2. double right index, since the array's indices are started from 0,

// we use right = 2 ^ n - 1

left = right;

right = 2 \* right + 1;

}

// not very necessary, but can return earlier.

if (dictionary.get(right) != null && dictionary.get(right) == target) {

return right;

}

return binarySearch(dictionary, target, left, right);

}

private int binarySearch(Dictionary dict, int target, int left, int right) {

// classical binary search

while (left <= right) {

int mid = left + (right - left) / 2;

if (dict.get(mid) == null || dict.get(mid) > target) {

right = mid - 1;

} else if (dict.get(mid) < target) {

left = mid + 1;

} else {

return mid;

}

}

return -1;

}

**Shifted Sorted Array Binary Search**

2 3 4 5 1

4 5 1 2 3

1

1 2

2 1

public class ShiftedSortedArrayBinarySearch {

public int search(int[] array, int target) {

if (array == null) {

return -1;

}

int left = 0;

int right = array.length - 1;

// case 1. array[mid] is the target

// case 2. array's left part is monotonic increasing

// case 3. array's right part is monotonic increasing

while (left <= right) {

int mid = left + (right - left) / 2;

if (array[mid] == target) {

return mid;

} else if (array[left] <= array[mid]) {

if (target >= array[left] && target < array[mid]) {

right = mid - 1;

} else {

left = mid + 1;

}

} else {

if (target > array[mid] && target <= array[right]) {

left = mid + 1;

} else {

right = mid - 1;

}

}

}

return -1;

}

## linked list questions, when to use dummy node?

it is not a must, but can make your life much easier.

* the head could be changed when solving the problem

Example - insert a node into sorted list

1). without dummy node

public ListNode insertNode(int value, ListNode head) {

// has to deal with the situation when we need to insert the node before head.

if (head == null || head.value >= value) {

ListNode newHead = new ListNode(value);

newHead.next = head;

return newHead;

}

ListNode prev = head;

while (prev.next != null && prev.next.value < value) {

prev = prev.next;

}

ListNode newNode = new ListNode(value);

newNode.next = prev.next;

prev.next = newNode;

return head;

}

2). with dummy node

public ListNode insertNode(int value, ListNode head) {

// TODO complete the implementation

ListNode dummy = new ListNode(0);

dummy.next = head;

ListNode prev = dummy;

while (prev.next != null && prev.next.value < value) {

prev = prev.next;

}

ListNode newNode = new ListNode(value);

newNode.next = prev.next;

prev.next = newNode;

return dummy.next;

}

* not sure yet which node will be head

Example - merge two sorted list

1). without dummy node

public ListNode merge(ListNode headOne, ListNode headTwo) {

// TODO complete the solution.

// which one is the head.

ListNode head = null;

if (headOne == null && headTwo == null) {

return head;

}

if (headOne == null) {

return headTwo;

} else if (headTwo == null) {

return headOne;

} else if (headOne.value <= headTwo.value) {

head = headOne;

headOne = headOne.next;

} else {

head = headTwo;

headTwo = headTwo.enxt;

}

ListNode current = head;

while (headOne != null && headTwo != null) {

if (headOne.value <= headTwo.value) {

current.next = headOne;

headOne = headOne.next;

} else {

current.next = headTwo;

headTwo = headTwo.next;

}

current = current.next;

}

if (headOne != null) {

current.next = headOne;

} else {

current.next = headTwo;

}

return head;

}

2). with dummy node

public ListNode merge(ListNode headOne, ListNode headTwo) {

ListNode dummyHead = new ListNode(0);

ListNode current = dummyHead;

while (headOne != null && headTwo != null) {

if (headOne.value <= headTwo.value) {

current.next = headOne;

headOne = headOne.next;

} else {

current.next = headTwo;

headTwo = headTwo.next;

}

current = current.next;

}

if (headOne != null) {

current.next = headOne;

} else {

current.next = headTwo;

}

return dummyHead.next;

}

**Reverse LinkedList**

1 > 2 > 3 > null

null < 1 < 2 < 3

p c

public ListNode reverseIterative(ListNode head) {

if (head == null) {

return head;

}

ListNode current = head;

ListNode prev = null;

while (current != null) {

ListNode next = current.next;

current.next = prev;

prev = current;

current = next;

}

return prev;

}

**Recursion:**

1. assume the smaller problem is already solved, can we use the result directly?

1 > 2 > 3 > null

for the list starting from “2”, it is already reversed:

null < 1 < 2 < 3

public ListNode reverseRecursive(ListNode head) {

if (head == null || head.next == null) {

return head;

}

ListNode result = reverseRecursive(head.next);

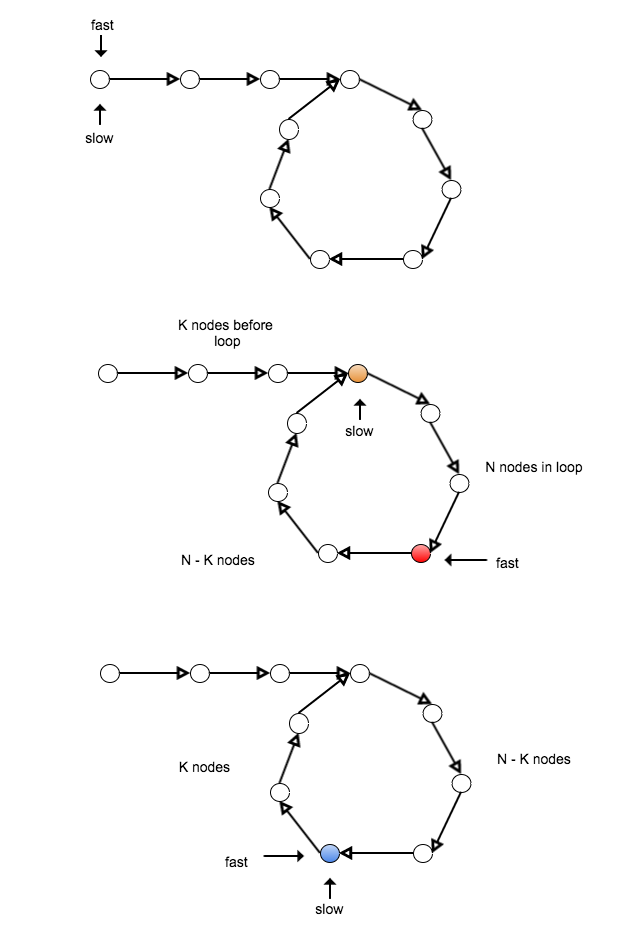
head.next.next = head;

head.next = null;

return result;

}

**Has Cycle**



public ListNode hasCircle(ListNode head) {

if (head == null || head.next == null) {

return null;

}

ListNode slow = head;

ListNode fast = head;

while (fast != null && fast.next != null) {

fast = fast.next.next;

slow = slow.next;

if (fast == slow) {

break;

}

}

if (fast == null || fast.next == null) {

return null;

}

fast = head;

while (fast != slow) {

fast = fast.next;

slow = slow.next;

}

return fast;

}

## PriorityQueue

It is a heap with the same Queue interface, for example, we can use offer(), peek(), poll().

The PriorityQueue will arrange the elements in the array based on the order.

Internally it is implemented using an array.

offer(E e) - insert one element into the heap

peek() - peek the top element in the heap

poll() - remove the top element from the heap, and return it

size()

isEmpty()

time complexity:

offer(E e) - O(logn)

peek() - O(1)

poll() - O(logn)

size() - O(1)

isEmpty - O(1)

**Order - You have to tell the PriorityQueue how to compare the objects**

* **Comparable**

class Coordinate implements Comparable<Coordinate> {

int x;

int y;

int value;

@Override

public int compareTo(Coordinate another) {

if (this.value == another.value) {

return 0;

} else if (this.value < another.value) {

return -1;

} else {

return 1;

}

}

}

PriorityQueue<Coordinate> minHeap = new PriorityQueue<Coordinate>();

* **Comparator**

class Coordinate {

int x;

int y;

int value;

}

class MyComparator implements Comparator<Coordinate> {

@Override

public int compare(Coordinate o1, Coordinate o2) {

if (o1.value == o2.value) {

return 0;

} else if (o1.value < o2.value) {

return -1;

} else {

return 1;

}

}

}

PriorityQueue<Coordinate> minHeap = new PriorityQueue<Coordinate>(2 \* k, new MyComparator());

Comparable/Comparator use the returned int value to determine the order of the two objects

* < 0 less than (the left argument will be considered of higher priority)
* == 0 equals
* > 0 greater than (the right argument will be considered of higher priority)

How to **change the default behavior** of a class E already implements Comparable

PriorityQueue<Integer> minHeap = new PriorityQueue<Integer>();

class Integer implements Comparable<Integer> {

int value;

@Override

public int compareTo(Integer another) {

if (this.equals(another)) {

return 0;

} else if (this.intValue() < another.intValue()) {

return -1;

} else {

return 1;

}

}

}

Example,

PriorityQueue<Integer> minHeap = new PriorityQueue<Integer>();

// I want a max heap here…

// So I need to provide a new Comparator overriding the natural order provided by Integer class

class ReverseComparator implements Comparator<Integer> {

@Override

public int compare(Integer i1, Integer i2) {

if (i1.equals(i2)) {

return 0;

} else if (i1.intValue() > i2.intValue()) {

return -1;

} else {

return 1;

}

}

}

PriorityQueue<Integer> maxHeap = new PriorityQueue<Integer>(16, new ReverseComparator());

Or, there is a utility method Collections.reverseOrder(), it will return a comparator that reverses the natural order.

PriorityQueue<Integer> maxHeap = new PriorityQueue<Integer>(16, Collections.reverseOrder());

**Note: if both Comparator<E> is provided and the E implements Comparable<E>, PriorityQueue will choose the order specified in Comparator.**

Most frequently used constructors

PriorityQueue<E> heap = new PriorityQueue<E>();

* initialize the internal array with default capacity(11)
* class E must implements Comparable<E> !

PriorityQueue<E> heap = new PriorityQueue<E>(16);

* initialize the internal array with specified capacity(16)
* class E implements Comparable<E> !

PriorityQueue<E> heap = new PriorityQueue<E>(16, new MyComparator());

* initialize the internal array with specified capacity(16)
* class MyComparator implements Comparator<E> !

ArrayList<E> list = new ArrayList<E>();

…...

PriorityQueue<E> heap = new PriorityQueue<E>(list);

* initialize the internal array with heapify()
* class E implements Comparable<E> !

**heapify() - convert an array to heap in O(n) time**

* it is not exposed to outside in PriorityQueue class, it is a private method.
* the only way you can utilize the heapify(), is to use the constructor

[**PriorityQueue**](https://docs.oracle.com/javase/7/docs/api/java/util/PriorityQueue.html#PriorityQueue(java.util.Collection))([**Collection**](https://docs.oracle.com/javase/7/docs/api/java/util/Collection.html)<? extends [**E**](https://docs.oracle.com/javase/7/docs/api/java/util/PriorityQueue.html)> c)

class PriorityQueue<E> {

private E[] heapArray;

private int size;

public PriorityQueue(Collection<? extends E> c) {

heapArray = c.toArray();

heapify(); // the only place heapify is called.

}

private void heapify() {

// heapify the heapArray.

for (int i = size / 2 - 1; i >= 0; i--) {

SiftDown(i);

}

}

}

**Smallest K Elements In Unsorted Array**

public int[] smallestK(int[] array, int k) {

// 1. sanity check.

assert array != null && k > 0 && array.length >= k;

// 2. maxHeap

PriorityQueue<Integer> maxHeap = new PriorityQueue<Integer>(k, Collections.reverseOrder());

// 3. insert the elements into the maxHeap.

for (int i = 0 ; i < array.length(); i++) [

if (i < k) {

maxHeap.offer(array[i]);

} else if (array[i] < maxHeap.peek()) {

maxHeap.poll();

maxHeap.offer(array[i]);

}

}

// 4. maxHeap-> k smallest elements,

return toArray(maxHeap);

}

private int[] toArray(PriorityQueue<Integer> maxHeap) {

int[] array = new int[maxHeap.size()];

for (int i = array.length - 1; i >= 0; i--) {

array[i] = maxHeap.poll();

}

return array;

}

# Class 4 - PriorityQueue, Queue & Stack Implementation

## Implement PriorityQueue(Heap)

### Definition

* Heap is a (binary) tree based data structure
* Across the entire tree, the relation between a parent node and a child node stays consistent.
  + Example: **min heap**

1. the parent node is always <= its two child nodes (parent node is the smallest node in the subtree rooted at itself).

2. the relation between the two child nodes can differ.

* The common implementation of a heap is using a complete binary tree.
  + A “**complete binary tree**” is a binary tree in which every level, except possibly the last level, is completely filled, and all nodes are as far left as possible.
* Property:
  + **O(1) access min (min heap), max(max heap).**
  + Height (N nodes in total): logN.

**Time Complexity:**

peek() - O(1)

poll() - O(logn)

offer() - O(logn)

### Representation

* it can also be represented as an array, since it is a complete binary tree.
  + Why? - if it is a complete binary tree, the matching between the nodes and the index of the array is determined, and the relation of parent and child nodes can be well transferred to the relation between two indices.
* Example:
  + index of parent = i, what is the index of the two child nodes?
    - left child = 2 \* i + 1
    - right child = 2 \* i + 2
  + the root of the tree is at index 0.

Min Heap Example:

1

/ \

3 10

/ \ / \

5 4 13 19

/ \ /

11 8 6

{1, 3, 10, 5, 4, 13, 19, 11, 8, 6 }

0 1 2 3 4 5 6 7 8 9

Max Heap Example:

19

/ \

13 6

/ \ / \

8 11 5 4

/ \ /

3 7 1

{19, 13, 6, 8, 11, 5, 4, 3, 7, 1}

0 1 2 3 4 5 6 7 8 9

### Basic Heap Internal Operations

**SiftUp()**

* when to use?
  + the element need to be moved up to maintain the heap’s property, for example, when offering a new element into the heap.
* how?
  + compare the element with its parent, move it up when necessary. do this until the element does not need to be moved.

offer():

1

/ \

3 10

/ \ / \

5 4 13 19

/ \ / \

11 8 6 2

append the new element 2 at the end of the heap.

compare 2 with its parent 4, since 2 < 4, we need to move it up.

1

/ \

3 10

/ \ / \

5 2 13 19

/ \ / \

11 8 6 4

compare 2 with its parent again, since 2 < 3, we need to move it up.

1

/ \

2 10

/ \ / \

5 3 13 19

/ \ / \

11 8 6 4

compare 2 with its parent again, since 2 >= 1, we can stop here.

after SiftUp(), the heap’s property is maintained.

**SiftDown()**

* when to use?
  + the element need to be moved down to maintain the heap’s property, for example, when poll the root element from the heap.
* how?
  + compare the element with its two children, if the smallest one of the two children is smaller than the element, swap the element with that child. do this until the element does not need to be moved.

poll()

1

/ \

3 10

/ \ / \

7 4 13 19

/ \ /

11 8 6

remove root and move the last element of the heap to root.

6

/ \

3 10

/ \ / \

7 4 13 19

/ \

11 8

compare 6 with the smallest of its two children, swap with 3

3

/ \

6 10

/ \ / \

7 4 13 19

/ \

11 8

compare 6 with the smallest of its two children, swap with 4

3

/ \

4 10

/ \ / \

7 6 13 19

/ \

11 8

Time Complexity = O(Height) = O(logn).

**Heapify()**

* convert an array into a heap in O(n) time.
* how?
  + for each node that has at least one child, we perform SiftDown() action, in the order of from the last such node to the root.
  + **the time complexity of heapify() is O(n)**

10

/ \

11 7

/ \ / \

2 8 4 6

/ \

13 3

all the red nodes are the nodes has at least one child. perform SiftDown() on each of them following the order: 2, 7, 11, 10:

SiftDown() on 2:

10

/ \

11 7

/ \ / \

2 8 4 6

/ \

13 3

SiftDown() on 7:

10

/ \

11 4

/ \ / \

2 8 7 6

/ \

13 3

SiftDown() on 11:

2

/ \

3 4

/ \ / \

10 8 7 6

/ \

13 11

SiftDown() on 10:

2

/ \

3 4

/ \ / \

10 8 7 6

/ \

13 11

Other Operations?

**update()**

* if you know the position of the element you want to update, it will take O(logn).
* how?
  + either you need SiftUp(), or SiftDown() on that element.
* what if you do not know the position of the element?
  + **you need to find the position of the element first, O(n)**

Examples:

1

/ \

2 4

/ \ / \

3 8 7 6

/ \

13 11

update 3 to 9 : SiftDown() is needed.

2

/ \

8 4

/ \ / \

10 9 7 6

/ \

13 11

update 10 to 1 : SiftUp() is needed.

1

/ \

2 4

/ \ / \

8 9 7 6

/ \

13 11

/\*\*

\* An example implementation of capacity limited min heap with

\* the capability to do update and poll at a specific index.

\*

\* This is for demonstration of shiftUp/shiftDown methods and

\* how to utilize these methods to do basic heap operations.

\*

\* The public methods provided are:

\* size()

\* isEmpty()

\* isFull()

\* peek()

\* poll()

\* offer()

\* peekAt(int index) - get the value at a given index

\* pollAt(int index) - get the value at a given index and remove it from the heap

\* update(int index, int value) - update the index to a given new value

\*

\*/

public class MinHeap {

private static final int DEFAULT\_CAPACITY = 128;

private int[] array; // the heap array, length is capacity

private int size; // the current heap’s size, <= capacity at any time

public MinHeap() {

this(DEFAULT\_CAPACITY);

}

public MinHeap(int capacity) {

assert capacity > 0;

this.array = new int[capacity];

this.size = 0;

}

public MinHeap(int[] array, int capacity) {

assert array != null && array.length >= 0;

assert array.length <= capacity;

size = array.length;

this.array = Arrays.copyOf(array, capacity);

heapify(); // O(n)

}

// Check if empty first.

// Just return the element at index 0.

public int peek() {

return peekAt(0);

}

/\*

\* Check if empty first.

\* size--

\* return the element at index 0, move the last element to index 0 and do shiftDown(0).

\*/

public int poll() {

return pollAt(0);

}

/\*

\* Check if full first.

\* size++

\* put the element to last position(array[size - 1]), and do shiftUp(size - 1).

\*/

public boolean offer(int element) {

if (!fullCheck()) {

return false;

} else {

size++;

update(size - 1, element);

}

}

/\*

\* Check if index is in the range first.

\* update to the new value at the desired index.

\* if can do shiftUp(index), do shiftUp(index), else do shiftDown(index).

\* return the previous element.

\*/

public int update(int index, int newElement) {

boundryCheck(index);

int result = array[index];

array[index] = newElement;

//if (array[index] < array[pIndex]) {

shiftUp(index);

//} else {

shiftDown(index);

//}

return result;

}

/\*

\* Check if index is in the range.

\* size--

\* move array[size] to array[index], do update().

\*/

public int pollAt(int index) {

boundryCheck(index);

size--; // array[size - 1] → array[size]

return update(index, array[size]);

}

public int peekAt(int index) {

boundryCheck(index);

return array[index];

}

public int size() {

return size;

}

public boolean isEmpty() {

return size == 0;

}

public boolean isFull() {

return size == array.length;

}

private void emptyCheck() {

}

private void fullCheck() {

}

private void boundryCheck(int index) {

}

// shiftDown for all the non-leaf nodes (size / 2 - 1);

0

1 2 size = 3, non-leaf node last index = 0

0

1 size = 2, non-leaf node last index = 0;

private void heapify() {

for (int index = size / 2 - 1; index >= 0; index--) {

shiftDown(index);

}

}

// from current index, current index's left child,

// current index's right child, pick the one with smallest value,

// if the smallest value's index is not current index,

// we need to do swap and shiftDown on the smallest value's index.

private void shiftDown(int index) {

while (index < size) {

int minIndex = index;

int leftIndex = index \* 2 + 1;

int rightIndex = index \* 2 + 2;

if (leftIndex < size && array[leftIndex] < array[minIndex]) {

minIndex = leftIndex;

}

if (rightIndex < size && array[rightIndex] < array[minIndex]) {

minIndex = rightIndex;

}

if (minIndex == index) {

break;

}

swap(index, minIndex);

index = minIndex;

}

}

// if parent index's value is larger than current index's value,

// we need to do swap and shiftUp on parent's index.

private void shiftUp(int index) {

while (index > 0) {

int pIndex = (index - 1) / 2;

if (array[pIndex] <= array[index]) {

break;

}

swap(index, pIndex);

index = pIndex;

}

}

private void swap(int indexOne, int indexTwo) {

int temp = array[indexOne];

array[indexOne] = array[indexTwo];

array[indexTwo] = temp;

}

}

## Queue & Stack

In Java, it is really Queue and Deque.

A collection used to hold multiple elements prior to processing.

Contains a buffer and specifies a mechanism to choose which is the next element to process.

* Queue - FIFO (exception: PriorityQueue)
  + offer() - offer at the tail
  + poll() - poll at the head
  + peek() - look at the head without polling it out
* Deque - FIFO & LIFO (both queue and stack)
  + provide both offer at head/tail, poll at head/tail, and peek at head/tail
  + so it can be used as a queue, or a stack.

Queue

|  |  |  |
| --- | --- | --- |
| Type of operation | throw exception | return special value(null) |
| Insert | add(e) | offer() |
| Remove | remove() | poll() |
| Examine | element() | peek() |

Deque

|  |  |  |
| --- | --- | --- |
| Type of operation | First element | Last element |
| Insert | addFirst(e)  offerFirst(e) | addLast(e)  offerLast(e) |
| Remove | removeFirst()  pollFirst() | removeLast()  pollLast() |
| Examine | getFirst()  peekFirst() | getLast()  peekLast() |

Other useful methods:

isEmpty();

size();

**All the operations’ cost is O(1).**

Most popular implementation class: LinkedList, ArrayDeque(Java 6).

When you need a queue or stack, you can just use LinkedList, as it implements both Queue and Deque.

Queue<Integer> queue = new LinkedList<Integer>();

queue.offer(1);

System.out.println(queue.peek());

System.out.println(queue.poll());

Deque<Integer> stack = new LinkedList<Integer>():

stack.offerFirst(1);

System.out.println(stack.peekFirst());

System.out.println(stack.pollFirst());

……

### Implementing Queue/Stack.

In general, two ways:

* use array
* use linked list

**Use Linked List:**

class ListNode {

int value;

ListNode next;

public ListNode(int value) {

this.value = value;

}

}

class Queue {

ListNode head;

ListNode tail;

public Queue() {

head = tail = null;

}

public boolean offer(int value) {

if (tail == null) {

head = tail = new ListNode(value);

} else {

tail.next = new ListNode(value);

tail = tail.next;

}

return true;

}

public Integer peek() {

if (head == null) {

return null;

}

return head.value;

}

public Integer poll() {

if (head == null) {

return null;

}

int result = head.value;

head = head.next;

if (head == null) {

tail = null;

}

return result;

}

}

class Stack {

ListNode head;

public Stack() {

head = null;

}

public boolean push(int value) {

ListNode newHead = new ListNode(value);

newHead.next = head;

head = newHead;

return true;

}

public Integer pop() {

if (head == null) {

return null;

}

int result = head.value;

head = head.next;

return result;

}

public Integer top() {

if (head == null) {

return null;

}

return head.value;

}

}

**Use Array:**

**Circular Array**

class Queue {

int[] array;

int head;

int tail;

//int size;

public Queue(int capacity) {

// santity check on capacity

array = new int[capacity];

head = 0;

tail = 1;

}

public boolean offer(int value) {

if (head == tail) {

return false;

}

array[tail] = value;

tail = (tail + 1) % array.length;

return true;

}

public Integer poll() {

if ((head + 1) % array.length == tail) {

return null;

}

head = (head + 1) % array.length;

return array[head];

}

public Integer peek() {

if ((head + 1) % array.length == tail) {

return null;

}

return array[(head + 1) % array.length];

}

}

class Stack {

int[] array;

int head;

public Stack(int capacity) {

// sanity check on capacity

array = new int[capacity];

head = 0;

}

public boolean push(int value) {

if (head == array.length) {

return false;

}

array[head++] = value;

return true;

}

public Integer pop() {

if (head == 0) {

return null;

}

return array[--head];

}

public Integer top() {

if (head == 0) {

return null;

}

return array[head - 1];

}

}

# Class 5 - HashMap

## practice of queue/stack

**Implement Queue With Two Stacks**

Notice: stack is LIFO, how we can make it FIFO?

two stacks, one for input, the other for output.

1. when offer(), always push input stack.

2. when poll(), always pop output stack.

\*3. when is the element in input stack moved to output stack?

offer(1), offer(2), offer(3), peek()

input: {1, 2, 3}

output: {}

move all the elements to output stack

input: {}

output: {4}

\* only move the elements in input stack to output stack when output stack is empty.

public class TwoStackQueue {

LinkedList<Integer> input;

LinkedList<Integer> output;

public TwoStackQueue() {

input = new LinkedList<Integer>();

output = new LinkedList<Integer>();

}

public int size() {

return input.size() + output.size();

}

public boolean isEmpty() {

return size() == 0;

}

public Integer poll() {

// 1. empty

if (isEmpty()) return null;

// 2. output is empty

move();

output.pollFirst();

}

public Integer peek() {

// 1. empty

if (isEmpty()) return null;

// 2. output is empty

move();

output.peekFirst();

}

private void move() {

if (output.isEmpty()) {

while (!input.isEmpty()) {

output.offerFirst(input.pollFirst());

}

}

}

public void offer(Integer element) {

input.offerFirst(element);

}

}

time complexity:

offer() → O(1)

peek() → worst case O(n), average O(1)

poll() → worst case O(n), average O(1)

**Binary Tree Level Order Traversal**

1

/ \

2 3

/ \ \

4 5 7

/ \

8 9

BFS: use queue

queue state:

**{1} - level 1, we know how many nodes in level 1 by queue.size()**

**{2, 3} - level 2, we know how many nodes in level 2 by queue.size()**

{3 | 4, 5} - poll 2,

**{4, 5, 7} - level 3, we know how many nodes in level 3**

{5, 7 | } - poll 4

{7 | 8} - poll 5

**{8, 9} - level 4,**

{9 |} - poll 8

{} - poll 9

result:

1

23

457

89

public void BFS(TreeNode root) {

assert root != null;

Queue<TreeNode> queue = new LinkedList<TreeNode>();

queue.offer(root);

while (!queue.isEmpty()) {

int size = queue.size();

for (int i = 0 ; i < size; i++) {

TreeNode cur = queue.poll();

System.out.print(cur.key);

if (cur.left != null) {

queue.offer(cur.left);

}

if (cur.right != null) {

queue.offer(right);

}

}

System.out.println();

}

}

**Is Complete Binary Tree**

what method we should use to tackle this problem?

1. what is complete binary tree?

1

/ \

2 4

/ \ / \

6 3 5 9

/ \

7 8 YES

1

/ \

2 4

/ \ / \

6 3 5 9

/ \ \

7 8 11 NO

2. BFS - once we find a node that does not have some child node, all the nodes afterwards should not have any child node at all.

public boolean isComplete(TreeNode root) {

/\*

\* Thought:

\* use a flag to mark when the rest of the nodes in the queue

\* should not have any child nodes.

\* when offering the node to the queue, check the flag first.

\*/

assert root != null;

boolean flag = false;

Queue<TreeNode> queue = new LinkedList<TreeNode>();

queue.offer(root);

while (!queue.isEmpty()) {

TreeNode cur = queue.poll();

// if left is null → flag = true;

// if left is not null:

// if flag is true, return false

// if flag is false, offer the left node to the queue.

if (cur.left == null) {

flag = true;

} else if (flag) {

return false;

} else {

queue.offer(cur.left);

}

if (cur.right == null) {

flag = true;

} else if (flag) {

return false;

} else {

queue.offer(cur.right);

}

}

return true;

}

## Set

A collection that can not contain duplicate elements.

The Java platform contains three general-purpose Set implementations:

HashSet, TreeSet, and LinkedHashSet.

* **HashSet**: which stores its elements in a hash table, is the best-performing implementation; however it makes no guarantees concerning the order of iteration
* **TreeSet**: which stores its elements in a red-black tree(balanced binary search tree), orders its elements based on their values
* **LinkedHashSet**: it is a HashSet and also it is a LinkedList, it maintains the order when each of the elements is inserted into the HashSet.

**Map**

An collection that maps keys to values. A Map cannot contain duplicate keys; each key can map to at most one value.

The Java platform contains three general-purpose Map implementations:

HashMap, TreeMap, and LinkedHashMap.

## HashMap

**Time Complexity**

|  |  |  |
| --- | --- | --- |
| Operation | Average | Worst |
| search(contains) | O(1) | O(n) |
| insert(put) | O(1) | O(n) |
| delete(remove) | O(1) | O(n) |

**Common API**

<http://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html>

put(K key, V value)

get(K key)

remove(K key)

containsKey(K key)

entrySet() - get the set view of all the entries in the hashmap

keySet() - get the set view of all the keys in the hashmap

values()

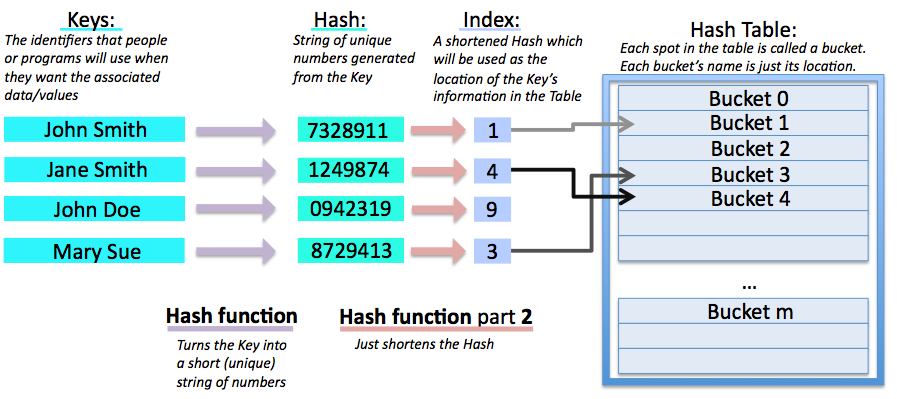
clear()

size()

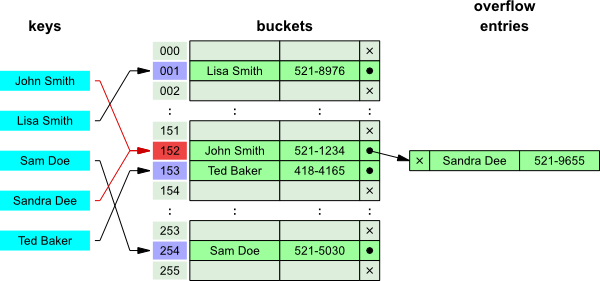
isEmpty()

**A hash table implementation.**

* A table of buckets(an array), each bucket has its own bucket number(index).
* For each <Key, Value>, it goes to one of the buckets, the bucket index is determined by a hash function applied on Key and the number of buckets.



* **Collision Control**
  + **Separate Chaining**(Close Addressing) - the element of each of the buckets is actually a single linked list.
  + If different Keys are determined to use the same bucket(collision), they will be chained in the list.



* What are the elements in the buckets?

class Entry<K, V> {

K key;

V value;

Entry<K, V> next;

}

<http://docs.oracle.com/javase/8/docs/api/java/util/Map.Entry.html>

**HashMap operations**

* **put(Key, Value)**

1. int hash = hash(Key.hashCode()) -- hash
2. int index = hash % table\_size -- index
3. search for the list resides in the bucket to see if the key already exists
   1. if not find the same key already existed, add a new Entry node to the list
   2. if find an Entry with the same key, update the value of that Entry

* **get(Key)**

1. int hash = hash(Key.hashCode()) -- hash
2. int index = hash % table\_size -- index
3. search for the list resides in the bucket to see if the key already exists

**load factor**

* default 0.75
* number of Entries / number of buckets
* If the ratio is too large, there will be too many collisions and the performance of the hash map will be degraded dramatically, so if the load ratio is exceeding a certain factor, the hash map needs to be expanded(**rehashing**).

HashMap can only has one null key, it is always mapped to bucket 0.

**In HashMap, we can only have one key.**

**\*\*HashMap use equals() to determine if two keys are the same key, and use hashCode() to determine which bucket the key should go to.**

**==, equals(), hashCode()**

How do you define “the same key”?

* **equals**()
  1. defined in Object class
  2. Object is the root class for any Java class
  3. any Java class implicitly extends Object class
  4. the default implementation is check if the two references are pointed to the same object “==”

class Object {

public boolean equals(Object obj) {

return this == obj;

}

public int hashCode() {

// here it returns the object’s memory address, unique for each of the objects.

}

}

* 1. sometimes we do not want to compare the reference… we want to compare the “value”

Example:

// s1 and s2 are different String objects

String s1 = “abc”;

String s2 = new String(“abc”);

// this will return false, “==” always compare if they are the same object.

s1 == s2;

// this will return true, equals() for String class actually checks the “value”

s1.equals(s2);

class String {

char[] array;

int offset;

int length;

@Override

public boolean equals(Object obj) {

if (obj == this) return true;

if (!(obj instanceof String)) return false;

String another = (String) obj;

// check each of the characters of the two String object if they are the same

// return true if they match

}

}

* other class has to override equals(), hashCode() if you want to have a different meaning other than the ones defined in Object class.

class MyObject (extends Object) {

int value;

public MyObject(int value) {

this.value = value;

}

}

if in your own class, the equals() and hashCode() methods are not been overridden, it will inherit the methods from Object class.

MyObject one = new MyObject(1);

MyObject another = new MyObject(1);

one == another; // false

System.out.println(one.equals(another)) // false

class MyObject (extends Object) {

int value;

public MyObject(int value) {

this.value = value;

}

@Override

public boolean equals(Object obj) {

//… compare value.

}

}

MyObject one = new MyObject(1);

MyObject another = new MyObject(1);

one == another; // false

System.out.println(one.equals(another)) // true

* **hashCode**()
  1. defined in Object class, the default implementation is the reference value(which essentially is the memory address for the object)

**When you want to override equals(), please override hashCode() as well**

* Usually when you have your own class and you have your own way of define “the same”, you have to override equals()
* If you override equals(), usually you will have to override hashCode() as well, to guarantee “the same” Keys will be mapped to the same bucket.
* Example

**not override equals(), hashCode()**

class Coordinate {

int x;

int y;

public Coordinate(int x, int y) {

this.x = x;

this.y = y;

}

@Override

public String toString() {

return “(” + x +”, ” + y + “)”;

}

}

HashMap<Coordinate, Integer> map = new HashMap<Coordinate, Integer>();

map.put(new Coordinate(1, 2), 5);

map.put(new Coordinate(1, 2), 6);

System.out.println(map);

**override equals(), not override hashCode()**

class Coordinate {

int x;

int y;

public Coordinate(int x, int y) {

this.x = x;

this.y = y;

}

@Override

public boolean equals(Object obj) {

if (this == obj) {

return true;

}

if (!(obj instanceof Coordinate)) {

return false;

}

Coordinate another = (Coordinate) obj;

return this.x == another.x && this.y == another.y;

}

@Override

public String toString() {

return “(” + x +”, ” + y + “)”;

}

}

HashMap<Coordinate, Integer> map = new HashMap<Coordinate, Integer>();

map.put(new Coordinate(1, 2), 5);

map.put(new Coordinate(1, 2), 6);

System.out.println(map);

**override equals(), override hashCode()**

class Coordinate {

int x;

int y;

public Coordinate(int x, int y) {

this.x = x;

this.y = y;

}

@Override

public boolean equals(Object obj) {

if (this == obj) {

return true;

}

if (!obj instanceof Coordinate) {

return false;

}

Coordinate another = (Coordinate) obj;

return this.x == another.x && this.y == another.y;

}

@Override

public int hashCode() {

return x \* 101 + y;

}

@Override

public String toString() {

return “(” + x +”, ” + y + “)”;

}

}

HashMap<Coordinate, Integer> map = new HashMap<Coordinate, Integer>();

map.put(new Coordinate(1, 2), 5);

map.put(new Coordinate(1, 2), 6);

System.out.println(map);

**hashCode() is VERY important!**

1. The performance of HashMap solely depends on how good the hashCode() is.
2. easy, fast, efficient
3. minimize collision, as evenly distributed as possible

**Common hashCode() implementation**

class Combo {

A a;

B b;

C c;

public Combo(A a, B b, C c) {

this.a = a;

this.b = b;

this.c = c;

}

@Override

public int hashCode() {

return a.hashCode() \* 31 \* 31 + b.hashCode() \* 31 + c.hashCode;

}

}

use a prime as the feed product → to minimize collisions

a0 \* p ^ k + a1 \* p ^ (k - 1) + a2 \* p ^ (k - 2) + … + ak \* p ^ 0

**How to iterate each of the key, value pairs in a HashMap?**

HashMap<String, Integer> map = new HashMap<String, Integer>();

map.put(“google”, 1);

map.put(“yahoo”, 2);

…...

1. // This is not efficient, do not do this!

for (String key : map.keySet()) {

System.out.println(key);

System.out.println(**map.get(key))**;

}

1. // This is good!

for (Entry entry : map.entrySet()) {

System.out.println(entry.getKey());

System.out.println(entry.getValue());

}

## HashSet

It is backed up by a HashMap instance. Only care about the Key here.

Common API

<http://docs.oracle.com/javase/7/docs/api/java/util/HashSet.html>

add(E e)

remove(Object o)

contains(Object o)

clear()

size()

isEmpty()

class HashSet<K> {

private HashMap<K, Object> map;

private static final Object PRESENT = new Object(); // special object used for all the keys

public HashSet() {

map = new HashMap<K, V>();

}

public boolean contains(K key) {

return map.containsKey(key);

}

public boolean add(K key) {

return map.put(key, PRESENT);

}

…...

}

/\*\*

\* A hashtable implementation of map.

\*

\* supported operations:

\* size()

\* isEmpty()

\* clear()

\* put(K key, V value)

\* get(K key)

\* containsKey(K key)

\* containsValue(V value)

\* remove(K key)

\*/

public class MyHashMap<K, V> {

// the entry in the bucket is singled linked list

// each node contains a pair of key-value.

static class Node<K, V> {

// the key is not supposed to be changed once the Entry

// for the key is constructed.

final K key;

V value;

Node<K, V> next;

Node(K key, V value, Node<K, V> next) {

this.key = key;

this.value = value;

this.next = next;

}

public K getKey() {

return key;

}

public V getValue() {

return value;

}

public V setValue(V value) {

V old = this.value;

this.value = value;

return old;

}

// for pretty print purpose

@Override

public String toString() {

return String.valueOf(key) + ": " + String.valueOf(value);

}

}

private static final int DEFAULT\_CAPACITY = 16;

private static final float DEFAULT\_LOAD\_FACTOR = 0.75f;

private int size; // how many entries are in the table

private Node<K, V>[] table;

private final float loadFactor;

public MyHashMap() {

this(DEFAULT\_CAPACITY, DEFAULT\_LOAD\_FACTOR);

}

public MyHashMap(int capacity) {

this(capacity, DEFAULT\_LOAD\_FACTOR);

}

@SuppressWarnings("unchecked")

public MyHashMap(int capacity, float loadFactor) {

assert capacity > 0;

assert loadFactor > 0f && loadFactor < 1f;

this.table = (Node<K, V>[]) new Node[capacity];

this.loadFactor = loadFactor;

}

public int size() {

return size;

}

public boolean isEmpty() {

return size == 0;

}

public void clear() {

Arrays.fill(this.table, null);

}

private Node<K, V> getNode(K key) {

if (isEmpty()) {

return null;

}

// 1. get the bucket number/index.

int bucketIndex = getBucketIndex(key);

// 2. search the key in the according linked list

return findNode(key, table[bucketIndex]);

}

private Node<K, V> findNode(K key, Node<K, V> head) {

while (head != null) {

if (equalsKey(key, head.getKey())) {

return head;

}

head = head.next;

}

return null;

}

private int getBucketIndex(K key) {

return hash(key) % table.length;

}

private int hash(K key) {

// key == null

if (key == null) return 0;

return key.hashCode() & 0x7FFFFFFF; // handle negative integer.

}

public boolean containsKey(K key) {

return getNode(key) != null;

}

// O(n)

public boolean containsValue(V value) {

if (isEmpty()) {

return false;

}

// the only thing we can do to check whether the value

// is in the map is to iterate the whole table.

// O(n) time complexity

for (Node<K, V> node : table) {

while (node != null) {

if (equalsValue(value, node.value)) {

return true;

}

node = node.next;

}

}

return false;

}

public V put(K key, V value) {

// 1. determine which bucket should this key go.

int bucketIndex = getBucketIndex(key);

// 2. find the node in the list if there is already a node containing the key

Node<K, V> node = findNode(key, table[bucketIndex]);

// if the key is not found, add a new node to the head of the list.

if (node == null) {

table[bucketIndex] = new Node<K, V>(key, value, table[bucketIndex]);

size++;

return null;

}

// if the key is found, change the value of the node.

return node.setValue(value);

}

public V get(K key) {

Node<K, V> node = getNode(key);

return node == null ? null : node.getValue();

}

public V remove(K key) {

if (isEmpty()) {

return null;

}

// find the bucket for the key

// remove the node for the key from the linked list.

}

// we use equals() to determine if two values are the same.

private boolean equalsValue(V value1, V value2) {

return value1 == value2 || value1 != null && value1.equals(value2);

}

// we use equals() to determine if two keys are the same.

private boolean equalsKey(K key1, K key2) {

return key1 == key2 || key1 != null && key1.equals(key2);

}

// for pretty print purpose

@Override

public String toString() {

StringBuilder builder = new StringBuilder("{");

for (Node<K, V> node : table) {

while (node != null) {

builder.append(node).append(", ");

node = node.next;

}

}

if (builder.length() > 1) {

builder.delete(builder.length() - 2, builder.length());

}

builder.append("}");

return builder.toString();

}

# Class 6 - Java Collection & Tree

## Data Structures vs. Java Implementations

**Data Structures**

* array new int[5];
* array(sorted) -- search O(n) → O(logn)
  + O(1) random access
  + limitation on size of the array, fixed size or resizable?
* linked list
  + singly
  + doubly
  + O(n) random access
  + O(1) append at head/tail
* queue
  + O(1) offer, poll, peek
  + FIFO
* stack
  + O(1) push, pop, top
  + LIFO
* deque
  + O(1) insert/remove/peek at both end
* heap
  + elements are ordered
  + **O(1) peek smallest/largest**
  + O(logn) offer, poll
* hashtable
  + elements are not ordered
  + O(1) search, insert, remove
* binary tree
  + O(n) search
* balanced binary search tree
  + elements are ordred
  + O(logn) search, insert, remove

**Java Implementation**

* array (int[] array, int[][] matrix, ArrayList)
* linked list
  + singly
  + doubly (LinkedList)
* queue (LinkedList)
* stack (LinkedList)
* deque (LinkedList)
* heap (PriorityQueue)
* hashtable (HashMap, HashSet)
* Balanced Binary Search Tree (TreeMap, TreeSet)
* …...

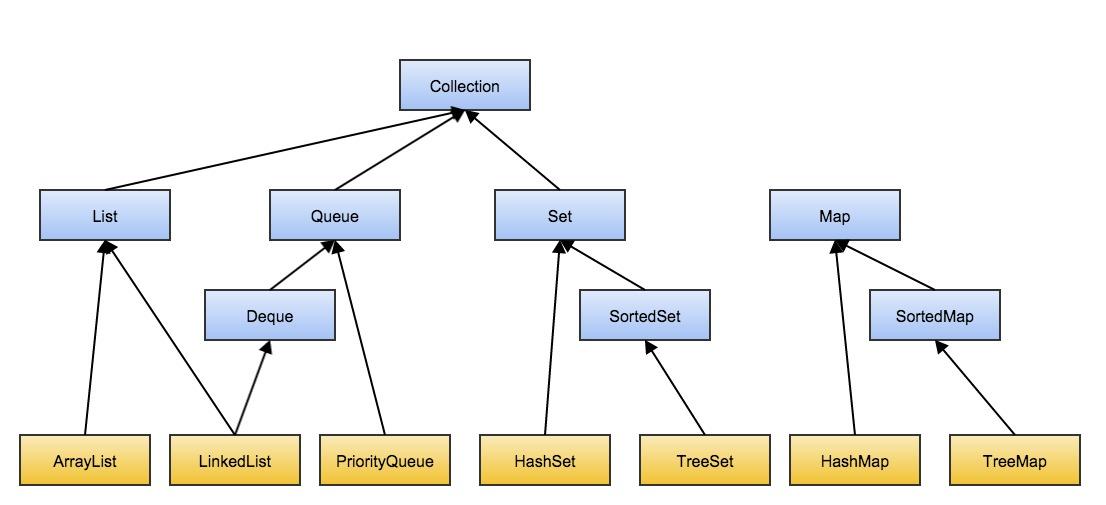
We studied these from bottom level closest to the the real implementation. Can we do it in reverse order?

Lets form a standard library and class hierarchy:

## Java Collections

* The most most frequently used classes and APIs during practices/interviews.
* The representation classes of **data structures**(called collections/containers).
* Collections Framework is available from Java 5 (a set of the classes and interfaces forming a hierarchical structure, following the strong OO paradigm, containing all the interfaces and implementations)
* Official tutorial: <http://docs.oracle.com/javase/tutorial/collections/>

**Interfaces Hierarchy**



interface Collection<E> extends Iterable<E> {

int size();

boolean isEmpty();

void clear();

boolean add(E e);

boolean contains(E e);

boolean remove(E e);

Object[] toArray();

<T> T[] toArray(T[] array);

…...

}

<http://docs.oracle.com/javase/7/docs/api/java/util/Collection.html>

**Defines what operations should be provided for any classes that is a “Collection”. Any class that is a kind of collection should implement the defined method.**

interface Set<E> extends Collection<E> {

…...

}

<http://docs.oracle.com/javase/7/docs/api/java/util/Set.html>

**Defines a group of data structures, that contains no duplicate elements.**

interface SortedSet<E> extends Set<E> {

……

// some additional operations should be supported by SortedSet

E first(); // smallest one

E last(); // largest one

SortedSet<E> headSet(E toElement); // all the elements smaller than toElement

SortedSet<E> tailSet(E fromElement); // all the elements larger than fromElement

…...

}

<http://docs.oracle.com/javase/7/docs/api/java/util/SortedSet.html>

**Define a subtype of Set, that contains no duplicate elements, and the elements are sorted. It is a subtype of Set.**

class HashSet<E> implements Set<E> {

…...

}

<http://docs.oracle.com/javase/7/docs/api/java/util/HashSet.html>

**A concrete implementation of Set using hashtable.**

* It is a Set
* It is a Collection

**It implements all the methods specified in Set and Collection.**

class TreeSet<E> implements SortedSet<E> {

……

}

<http://docs.oracle.com/javase/7/docs/api/java/util/TreeSet.html>

**A concrete implementation of SortedSet using red-black tree.**

* It is a SortedSet
* It is a Set
* It is a Collection

**It implements all the methods specified in SortedSet, Set and Collection.**

Similarly,

interface List<E> extends Collection<E> {

……

// provide random access by index

E get(int index);

E remove(int index);

E set(int index, E element);

...

}

<http://docs.oracle.com/javase/7/docs/api/java/util/List.html>

**Defines a group of data structures, that can contain duplicate elements, maintain the order of insertion of the elements and provide the capability of random access by using index.**

class ArrayList<E> implements List<E> {

……

}

<http://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html>

**A concrete implementation of List by resizable-array.**

* It is a List
* It is a Collection

**It implements all the methods specified in List and Collection.**

interface Queue<E> extends Collection<E> {

// group one operations

boolean add(E element);

E element();

E remove();

// group two operations

boolean offer(E element);

E peek();

E poll();

}

<http://docs.oracle.com/javase/7/docs/api/java/util/Queue.html>

**Defines a group of data structures, that is designated to 1). hold and buffer elements before processing 2). provide a way of choosing which element buffered is the next one to be processed. 3). there are two ends of the Queue, and offer always at tail, poll() always at head. (FIFO)**

interface Deque<E> extends Queue<E> {

……

boolean offerFirst();

boolean offerLast();

E pollFirst();

E pollLast();

E peekFirst();

E peekLast();

…...

}

<http://docs.oracle.com/javase/7/docs/api/java/util/Deque.html>

**Defines a subtype of queue, where it is double ended. (FIFO & LIFO are both provided)**

class LinkedList<E> implements List<E>,Deque<E> {

…...

}

<http://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html>

**A List implementation that backed by a doubly linked list. It also can be used as queue/deque/stack.**

* It is a List
* It is a Queue
* It is a Deque
* It is a Collection

class PriorityQueue<E> implements Queue<E> {

…...

}

<http://docs.oracle.com/javase/7/docs/api/java/util/PriorityQueue.html>

**Defines an special implementation of queue, where it is using priority to determine which element is the next to process.**

Similar to Map, SortedMap, HashMap and TreeMap:

<http://docs.oracle.com/javase/7/docs/api/java/util/Map.html>

<http://docs.oracle.com/javase/7/docs/api/java/util/SortedMap.html>

<http://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html>

<http://docs.oracle.com/javase/7/docs/api/java/util/TreeMap.html>

Pick the correct implementation class when you need some “Collection” to solve the problem:

When a List/Array is needed, ArrayList/LinkedList

* check point, when to use ArrayList and when to use LinkedList?

When a Queue/Deque/Stack is needed, LinkedList

When a Heap is needed, PriorityQueue

When need to search, insert, remove in O(1), HashSet/HashMap

……

## About **Arrays** and **Collections**

They are the placeholders of a set of utility methods for manipulating arrays and collections objects.

* They are NOT the class for array and Collection
* All the methods in Arrays/Collections are static methods, so there is no need to create a Arrays/Collections object to utilize them

**Arrays**

<http://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html>

* Arrays.sort()

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

Arrays.sort(array);

// array = {1, 2, 3};

* Arrays.asList() - **convert an array to a List**

List<Integer> list = Arrays.asList(1, 2, 3);

// list is a List of [1, 2, 3]

* Arrays.copyOf(original, newLength)

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

int[] copy = Arrays.copyOf(array, 1);

// copy = {1}

copy = Arrays.copyOf(array, 5);

// copy = {1, 2, 3, 0, 0}

* Arrays.copyRange(original, from, to)

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

int[] copy = Arrays.copyRange(array, 1, 2);

// copy = {2}

* Arrays.fill(original, value)

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

Arrays.fill(array, 1);

// array = {1, 1, 1}

* Arrays.toString()

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

String s = Arrays.toString(array);

// s = “[1, 2, 3]”;

* Arrays.binarySearch(original, value)

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

Arrays.sort(array);

// the array must be guaranteed to be sorted before using binarySearch.

int i = Arrays.binarySearch(array, 2);

// i = 1

**Collections**

<http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html>

* Collections.sort()
* Collections.binarySearch()
* Collections.swap()
* Collections.fill()
* Collections.reverse()
* **Collections.reverseOrder()**

PriorityQueue<Integer> maxHeap = new PriorityQueue<Integer>(11, Collections.reverseOrder());

**Collection to array**

List<String> list = new ArrayList<String>();

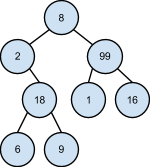
// tell the type information when forming the array.

String[] array = list.toArray(new String[0]);

List<Integer> list = new ArrayLIst<Integer>();

Integer[] array = list.toArray(new Integer[0]);

## Binary (Search) Tree



class TreeNode {

int key;

TreeNode left;

TreeNode right;

// TreeNode parent;

// we can store parent reference as well, this will be useful for solving some problems.

public TreeNode(int key) {

this.key = key;

}

}

Usually, we use the root to represent the binary tree.

General Tree, each node can have arbitrary number of children.

class TreeNode {

int key;

List<TreeNode> children;

// TreeNode parent;

public TreeNode(int key) {

this.key = key;

children = new ArrayList<TreeNode>();

}

}

sounds similar? this is exactly the same representation of directed graph node.

a tree is special kind of directed graph.

class GraphNode {

int key;

List<GraphNode> neighbors;

public GraphNode(int key) {

this.key = key;

neighbors = new ArrayList<GraphNode>();

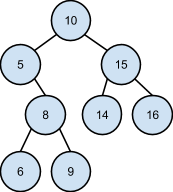
}

}

**Binary Search Tree**

is a binary tree maintains the following property:

for any of the nodes in the binary tree, all the nodes in its right subtree is larger than itself, all the nodes in its left subtree is smaller than itself(compared by the key stored in each of the nodes).



Recursion:

divide and conquer natural of binary tree -- binary tree can be easily divided into three parts

* solve problem for left/right subtree
* solve the problem for root.

Iteration:

Basic Operations of Binary Search Tree:

find() - worst case O(n), average O(logn)

insert() - worst case O(n), average O(logn)

remove() - worst case O(n), average O(logn)

Traversals:

preorder - root, left sub, right sub

inorder - left sub, root, right sub

postorder - left sub, right sub, root

**An important property of Binary Search Tree:**

**all the nodes in right sub is > root**

**all the nodes in left sub is < root**

The most fundamental thing to solve binary search tree problems.

Recursion:

1. root == target

public TreeNode find(TreeNode root, int target) {

if (root == null) {

return root;

}

if (root.key == target) {

return root;

} else if (root.key > target) {

return find(root.left);

} else {

return find(root.right);

}

}

1. return root.
2. return root; (==)
3. return find(root.left); (>)
4. return find(root.right); (<)

**Tail Recursion:**

The recursive call is always the last execution statement.

We can easily transfer the tail recursion to iterative solution.

public TreeNode find(TreeNode root, int target) {

while (root != null) {

if (root.key == target) {

break;

} else if (root.key > target) {

**root = root.left;**

} else {

**root = root.right;**

}

}

return root;

}

Recursion:

public **TreeNode** insert(TreeNode root, int target) {

if (root == null) {

return new TreeNode(target);

}

if (root.key > target) {

**root.left = insert(root.left, target);**

} else if (root.key < target) {

root.right = insert(root.right, target);

}

return root;

}

Iterative:

public TreeNode insert(TreeNode root, int target) {

if (root == null) {

return new TreeNode(target);

}

TreeNode temp = root;

while (temp.key != target) {

if (temp.key > target) {

if (temp.left == null) {

temp.left = new TreeNode(target);

break;

} else {

temp = temp.left;

}

} else {

if (temp.right == null) {

temp.right = new TreeNode(target);

break;

} else {

temp = temp.right;

}

}

}

return root;

}

**Delete**

case 1. - the node to be deleted has no children.

3 3

/ \ → /

2 8 2

case 2. - the node to be deleted has no left child.

3 3

/ \ / \

2 8 → 2 10

\

10

case 3. - the node to be deleted has no right child.

3 3

/ \ / \

2 8 → 2 6

/

6

case 4. - the node to be deleted has both left and right child. we need to move the smallest node in the right subtree up(or move the largest node in the left subtree up).

which node to replace 8?

* either from left sub, largest of the left subtree
* or from right sub, smallest of the right subtree

case 4.1 - node.right does not have left child, meaning itself is the smallest node

in this case, we just move node.right up.

3 3

/ \ / \

2 8 → 2 10

/ \ / \

6 10 6 12

\

12

root → 8

root.right → 10

root.right.left → null

root.right.left = root.left;

case 4.2 - node.right has left child, we need to find the smallest node, and move it up.

3 3

/ \ / \

2 8 → 2 9

/ \ / \

6 12 6 12

/ \ / \

11 14 11 14

/ /

9 10

\

10

12 → left → left → left

9.left == null.

public TreeNode delete(TreeNode root, int target) {

// root == null

if (root == null) {

return root;

}

if (root.key > target) {

root.left = delete(root.left, target);

return root;

} else if (root.key < target) {

root.right = delete(root.right, target);

return root;

} else { // root.key == target, root is the node to be deleted.

if (root.left == null) {

return root.right;

} else if (root.right == null) {

return root.left;

}

if (root.right.left == null) {

root.right.left = root.left;

return root.right;

}

// root.right.left != null

// find the smallest in root.right. delete it from the subtree of root.right.

TreeNode newRoot = removeSmallest(root.right);

// link the smallest Nodes’ left and right.

newRoot.left = root.left;

newRoot.right = root.right;

// return the smallest Node.

return newRoot;

}

}

private TreeNode removeSmallest(TreeNode root) {

assert root != null && root.left != null;

TreeNode cur = root;

// find the smallest’s parent.

while (cur.left.left != null) {

cur = cur.left;

}

// cur is the parent of the smallest node.

TreeNode smallest = cur.left;

// link parent’s left.

cur.left = cur.left.right;

return smallest;

}

**Binary Tree Traversals** - too easy to use recursion to solve the traversal problem.

public void inorder(TreeNode root) {

if (root == null) {

return;

}

inorder(root.left);

System.out.println(root.key);

inorder(root.right);

}

Why do we need the Iterative solution, since we already have the elegant, simple, easily implemented recursion solution? In another word, is recursion always good?

* Trees could have very deep level
* The recursion levels needed depends on number of levels of the tree
* **Recursion method calls maintained in the stack, and stack is limited space**

m1() {

…

m1();

...

}

{**m1**, m1, m1, ….., m1}

In Java, by default, a several thousands levels recursion calls would easily throw StackOverFlowException.

That is why, iterative solution is very important for an imperative language…

# Class 7 - Tree ctd.

## **Binary Tree Traversals**

- too easy to use recursion to solve the traversal problem.

public void inorder(TreeNode root) {

if (root == null) {

return;

}

inorder(root.left);

System.out.println(root.key);

inorder(root.right);

}

Why do we need the Iterative solution, since we already have the elegant, simple, easily implemented recursion solution? In another word, is recursion always good?

* Trees could have very deep level
* The recursion levels needed depends on number of levels of the tree
* **Recursion method calls maintained in the stack, and stack is limited space**

m1() {

…

m1();

...

}

{**m1**, m1, m1, ….., m1}

In Java, by default, a several thousands levels recursion calls would easily throw StackOverFlowException.

That is why, iterative solution is very important for an imperative language…

It is not easy to convert all recursion solutions to a “while loop” solution without any other auxiliary data structures, since it is not possible to convert the traversal recursion to tail recursion.

public void inorder(TreeNode root) {

if (root == null) {

return;

}

inorder(root.left);

System.out.println(root.key);

inorder(root.right);

}

In this case, we will need something else to help us:

The recursion is implemented by a stack maintaining the method call levels, we can simulate this ourselves, so a stack will be needed.

Pre-Order:

5

/ \

2 8

/ \

1 3

5, 2, 1, 3, 8

once the root is traversed, we can print it directly and we do not need to store it in the stack any more.

1. We always print root first, then root can be eliminated from stack.
2. We traverse left sub first, so the **right sub should be retained in the stack** before the left sub is done.

Stack: Print:

{5} → pop 5 and push 8 and 2 5

{8, 2} → pop 2 and push 3 and 1 2  
{8, 3, 1} → pop 1 1

{8, 3} → pop 3 3

{8} → pop 8 8

public static void printTree(TreeNode root) {

if(root == null) {

return;

}

LinkedList<TreeNode> stack = new LinkedList<TreeNode>();

**stack.push(root);**

while(!stack.isEmpty()) {

**TreeNode temp = stack.pop();**

System.out.println(temp.value);

if(temp.right != null) {

stack.push(temp.right);

}

if(temp.left != null) {

stack.push(temp.left);

}

}

}

In-Order:

5

/ \

2 8

/ \

1 3

The problem is, we can not throw away the root in the stack before we traversed all the nodes in left subtree. How can we know we have already traversed all the nodes in left sub?

Use a helper node to store the current “visiting” node to help identify when we can go back to the previous level.

1. when helper node is not null, push helper and let helper = helper.left
2. when helper is null, means the left subtree is finished, we can print the top, and let helper = top.right. (traverse the left subtree first, then root, then right subtree)
3. do 1 and 2 until current is null and there is no nodes left in the stack.

Helper: Stack: Print:

{}

5 → 5 is not null, helper = 5.left {5}

2 → 2 is not null, helper = 2.left {5, 2}

1 → 1 is not null, helper = 1.left {5, 2, 1}

**null, helper = 1.right {5, 2} 1**

**null, helper = 2.right {5} 2**

3 → 3 is not null, helper = 3.left {5, 3}

null, helper = 3.right {5} 3

**null, helper = 5.right {} 5**

8 → 8 is not null, helper = 8.left {8}

null, helper = 8.right {} 8

public void inOrder(TreeNode root) {

if (root == null) {

return;

}

LinkedList<TreeNode> stack = new LinkedList<TreeNode>();

TreeNode current = root;

while (current != null || !stack.isEmpty()) {

if (current != null) {

stack.push(current);

current = current.left;

} else {

// stack’s top TreeNode’s left subtree has already been traversed.

TreeNode temp = stack.pop();

System.out.println(temp.value);

current = temp.right;

}

}

}

Post-Order:

5

/ \

2 8

/ \

1 3

1, 3, 2, 8, 5 → reverse → 5, 8, 2, 3, 1

Method 1:

1. left subtree, right subtree, root → reverse order of root, right subtree, left subtree

specialPreOrder(TreeNode root) {

// null

System.out.println(root.value);

specialPreOrder(root.right);

specialPreOrder(root.left);

}

public static void printTree(TreeNode root) {

if(root == null) {

return;

}

LinkedList<TreeNode> stack = new LinkedList<TreeNode>();

LinkedList<Integer> reverseStack = new LinkedList<Integer>();

**stack.push(root);**

while(!stack.isEmpty()) {

**TreeNode temp = stack.pop();**

**//System.out.println(temp.value);**

**reverseStack.push(temp.value);**

if(temp.left != null) {

stack.push(temp.left);

}

if(temp.right != null) {

stack.push(temp.right);

}

}

// print reverseStack

while (!reverseStack.isEmpty()) {

System.out.println(reverseStack.pop());

}

}

What is the drawback of Method 1?

reverseStack → space requirement is O(n).

Method 2:

The problem is, we need to traverse both left and right subtrees first, then we can eliminate the root from the stack. We need an mechanism to know, when we finish visiting all subtrees’ nodes.

5

/ \

2 8

/ \

1 3

Lets examine the status of the stack and traversing path more closely in this case:

{5, 2, 1}

push 5

push 2

push 1 → leaf node.

{5, 2}

**pop 1 → here, we can not pop 2 yet (when we go back to 2, we need to know if 2’s two subtrees have all been visited)**

**push 3**

pop 3

**here we can pop 2, because both the subtrees of 2 are visited.**

**{5}**

push 8

pop 8

**pop 5**

maintain a previous Node, to record the previous visiting node on the traversing path.

-current Node = stack.top  
-when previous Node == current node.left, means current node’s left subtree is done.

when previous Node == current node.right, means current node’s right subtree is also done.

5

/ \

2 8

/ \

1 3

Previous Stack Print → left and right subtree both finished

**null** {5}

5 {5, 2}

2 {5, 2, 1} 1

**1 {5, 2}**

2 {5, 2, 3} 3

**3 {5, 2} 2**

2 {5}

5 {5, 8} 8

8 {5} 5

5 {}

1. **if previous is null → going down (left subtree has priority)**
2. **if previous is current’s parent, → going down(left subtree has priority)**
3. if previous == current.left, → left subtree finished, going current.right
4. if previous == current.right → right subtree finished, pop current, going up

public void postOrder(TreeNode root) {

if (root == null) {

return;

}

// stack, previous

LinkedLIst<TreeNode> stack = new LinkedList<TreeNode>();

TreeNode previous = null;

stack.push(root);

while (!stack.isEmpty()) {

TreeNode current = stack.peek();

//

if (previous == null || current == previous.left || current == previous.right) {

if (current.left != null) {

stack.push(current.left);

} else if (current.right != null) {

stack.push(current.right);

} else {

**System.out.println(current.value);**

**stack.pop():**

}

} else if (previous == current.left && current.right != null) {

stack.push(current.right);

} else {

// previous == current.right

System.out.println(current.value);

stack.pop();

}

prev = current;

}

}

## Autoboxing and Unboxing

* **Primitive type vs. Wrapper class**
  + int : Integer
  + long: Long
  + char: Character
  + double: Double
  + boolean: Boolean
  + ……

A wrapper class is just a wrapper of the corresponding primitive type.

The Object representation of primitive type values.

**Similar to String, all the wrapper class objects are IMMUTABLE - internal values can not be changed after initialization.**

It can help:

1. Generic type can not be primitive type, **List<Integer>**, there can not be **List<int>**
2. It can help utilize the class hierarchy to provide useful functionalities.
3. **How do you represent a “not existed” int value? “null”**

Example:

class Integer implements Comparable<Integer> {

private **final** int value; // wrap the primitive type value inside.

public Integer(int value) {

this.value = value;

}

public int intValue() {

return value;

}

public Integer plus(Integer another) {

…..

}

@Override

public int compareTo(Integer another) {

if (value == another.value) {

return 0;

}

return value < another ? -1 : 1;

}

…...

}

Integer a;

Integer b;

a.compareTo(b); // compare

a.plus(b); // +

a < b;

* - \* / > < >= <= … only applied to primitive type.
* **autoboxing**

***Autoboxing*** is the automatic conversion that the Java compiler makes between the primitive types and their corresponding object wrapper classes.

For example, converting an int to an Integer, a double to a Double, and so on.

Example,

List<Integer> integerList = new ArrayList<Integer>();

for (int **i = 0; i < 50; i++**) {

integerList.**add(i)**; // == > integerList.add(**Integer.valueOf(i)**);

}

* **unboxing**

***Unboxing*** is the reverse operation of autoboxing.

Example,

Integer a = 4; // autoboxing ⇒ Integer a = Integer.valueOf(4);

a += 4; // a = a + 4;

a++;

What happened when integer++?

* Integer is immutable, the int value of the Integer object can never change
* a++:

int temp = a.intValue();

temp++;

a = Integer.valueOf(temp);

a += 4;

int temp = a;

temp + = 4;

a = Integer.valueOf(temp);

More Examples:

Integer a = 5; // autoboxing

int b = 5; // primitive

System.out.println(a > b); // unboxing, ⇒ a.intValue() > b, false

System.out.println(a == b); // unboxing, ⇒ a.intValue == b, true

// == both operand can be Object type.

Even More Examples:

Integer a = 5;

Integer c = 5;

System.out.println(a > c); // ⇒ a.intValue() > c.intValue()

System.out.println(a >= c); // ⇒ a.intValue() >= c.intValue()

System.out.println(a == c); // **true**, compare two references pointing to the same object.

a = 129;

c = 129;

System.out.println(a > c);

System.out.println(a >= c);

System.out.println(a == c); // **false**

* Integer class cache the Integer object with value from -127 to 128, so everytime an Integer object within this range is needed, it will always return the corresponding object.

Integer a = 5; ⇒ a pointed to the object of value 5.

Integer c = 5; ⇒ c pointed to the object of value 5.

Integer a = 129; ⇒ a = new Integer(129);

Integer c = 129; ⇒ c = new Integer(129);

* The unboxing is done only when it is necessary, for example, a > b. if you want to compare the int value of two Integer objects, always use equals() !!

More Examples…

public boolean largerThan(int a, int b) {

return a > b;

}

**public boolean largerThan(Integer a, int b) {**

**return a > b;// NullPointerException → a = null. unboxing will possibly introduce NullPointerException.**

**}**

public boolean largerThan(Integer a, Integer b) {

return a > b;

}

**public boolean same(int a, int b) {**

**return a == b; // compare value;**

**}**

**public boolean same(Integer a, int b) {**

**return a == b; // unboxing, NullPointerException.**

**}**

**public boolean sameValue(Integer a, Integer b) {**

**return a == b;// no unboxing, compare reference equility.**

**return a == b || a != null && a.equals(b);**

**}**

# Class 8 -Bits

## Code Style

* Why it is very important?
  + clear, readable, easily maintainable
  + good impression to your interviewee
* Define readable in writing Java code
  + self-explainable variable name
  + good structure and relatively small grouped logics - methods
  + clear and no redundant code
  + if/else/while/for..
    - 3 - 4 is OK, if you see more levels, we can consider refactoring the code…
* Google Java Style Guide
  + <https://google-styleguide.googlecode.com/svn/trunk/javaguide.html>
  + class name: [UpperCamelCase](https://google-styleguide.googlecode.com/svn/trunk/javaguide.html#s5.3-camel-case)
  + method/variable/argument name: [LowerCamelCase](https://google-styleguide.googlecode.com/svn/trunk/javaguide.html#s5.3-camel-case)

class ListNode {

int myValue;

ListNode theNextNode;

public ListNode(int myValue, ListNode theNext) {

this.myValue = myValue;

this.theNext = theNext;

}

}

* + white space

for (int index = 0; index < 1 + 2; index++) {

if (index == 0) {

……

} else {

……

}

}

* + use braces even it is optional

if (node == null) {

return;

}

int value = node.value;

……

* + naming

Better not use single character variable name.

int o, p, q = 0; // what does it mean?

## Bits

* How numbers are represented in bits in Java?
  + **most significant bit**(the leftmost bit):

the sign bit, determines whether the number is positive or negative.

**(There is no unsigned type in Java)**

* + **two’s complements**:

**0**: (all zero)

00000000000000000000000000000000

**5**:

sign bit is 0

5 = 1 \* 2^0 + 0 \* 2^1 + 1 \* 2^2

from least significant bit(the rightmost bit) to left

00000000000000000000000000000101

**-5**:

sign bit is 1

“complement of 5” + 1:

11111111111111111111111111111010

+ 1

11111111111111111111111111111011

5 - 4 = 5 + (-4)

00000000000000000000000000000101

+11111111111111111111111111111100

00000000000000000000000000000001

Example:

public class Bit {

public static void printBinary(int value, int shifts) {

System.out.println(value + " : ");

StringBuilder builder = new StringBuilder();

while (shifts > 0) {

builder.append(value & 1);

value >>>= 1;

shifts--;

}

System.out.println(builder.reverse());

System.out.println();

}

public static void main(String[] args) {

// 0

int a = 0;

printBinary(a, 32);

// positive number

a = 5;

printBinary(a, 32);

// negative number

a = -5;

printBinary(a, 32);

a = Integer.MIN\_VALUE;

printBinary(a, 32);

// Integer.MIN\_VALUE = - (MAX\_VALUE + 1)

// 10000000 00000000 00000000 00000000

a = Integer.MAX\_VALUE;

printBinary(a, 32);

// 01111111 11111111 11111111 11111111

// all "1"

a = -1;

printBinary(a, 32);

// 11111111 11111111 11111111 11111111

// signed shift - leftmost bit depends on previous leftmost bit

int b = a >> 5;

printBinary(b, 32);

// unsigned shift - leftmost bit "0"

b = a >>> 5;

printBinary(b, 32);

}

}

* + **“>>>” vs. “>>”**
    - **signed shift “>>”**

The leftmost position after signed shift depends on sign extension(the most significant bit):

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | ... | 1 | 1 | 0 | 1 | 0 | 1 |

>> 3

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ***1*** | ***1*** | ***1*** | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | ... | 1 | 1 | 0 |

* + - **unsigned shift “>>>”**

The signed right shift operator ">>>" shifts a zero into the leftmost position

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | ... | 1 | 1 | 0 | 1 | 0 | 1 |

>>> 3

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ***0*** | ***0*** | ***0*** | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | ... | 1 | 1 | 0 |

### Bit Cheat Sheets

**Basic Operations**

* & bitwise AND
  + 1 & 1 = 1
  + 1 & 0 = 0
  + 0 & 1 = 0
  + 0 & 0 = 0
* | bitwise OR
  + 1 | 1 = 1
  + 1 | 0 = 1
  + 0 | 1 = 1
  + 0 | 0 = 0
* ^ bitwise XOR
  + 1 ^ 1 = 0
  + 1 ^ 0 = 1
  + 0 ^ 1 = 1
  + 0 ^ 0 = 0
* ~ bitwise NOT
  + ~1 = 0
  + ~0 = 1
* << left shift, adding 0 at right end ( \* 2 ^ k)
  + 00000000 00000000 00000000 00000001 << 4

00000000 00000000 00000000 00010000

* >> right shift, padding bit is determined by the most significant bit at left end
  + 10000000 00000000 00000000 00000000 >> 4

11111000 00000000 00000000 00000000

* >>> unsigned right shift, always padding 0 at left end
  + 10000000 00000000 00000000 00000000 >>> 4

00001000 00000000 00000000 00000000

**Generate bit mask**

* 1 << k =
  + 00000000 00000000 00000000 00000001 << 4

00000000 00000000 00000000 00010000

only the kth bit from right end is 1, other bits are all 0

* + to set kth bit to 1
* ~(1 << k) = 11111111 11111111 11101111 1111
  + only the kth bit from right end is 0, other bits are all 1
  + to set kth bit to 0
* ((1 << (k - j + 1)) - 1) << j
  + bit mask with jth to kth bit as 1

Example:

j = 2, k = 5, ⇒ number of bits with 1 is 5 - 2 = 4

First we generate

00000000 00000000 00000000 00001111

=00000000 00000000 00000000 00010000 - 1

By x = (1 << 4) - 1

Then we left shit 2 bits

By x <<= 2

00000000 00000000 00000000 00111100

* ~(((1 << (k - j + 1)) - 1) << j)
  + bit mask with jth to kth bit as 0

**Most Frequently Used**

* check if kth bit is 0/1
  + use & 1 if you want to know what is the value for certain bit. (**x & 1 = x, x & 0 = 0**)
  + **(a >>> k) & 1 == 0 ?**

k = 4

a = 00000000 00000000 11111111 00010000

a >>> 4 = 00000000 00000000 00001111 11110001

& 00000000 00000000 00000000 00000001

* set kth bit to 1
  + use | 1 if you want to set one bit to 1. (**x | 1 = 1, x | 0 = x**)
  + **a |= (1 << k)**

k = 5

a = 00000000 00000000 11111111 00010000

1 << 5 = 00000000 00000000 00000000 00100000 (mask)

| 00000000 00000000 11111111 00110000

* set kth bit to 0
  + **a &= ~(1 << k)**

k = 4

a = 00000000 00000000 11111111 00010000

1 << 4 = 00000000 00000000 00000000 00010000

~(1 << 4) = 11111111 11111111 11111111 11101111(mask)

& 00000000 00000000 11111111 00000000

* XOR
  + a ^ a = 0, (**x ^ x = 0, x ^ (~x) = 1, 0 ^ x = x, 1 ^ x = ~x**)
  + a ^ a ^ b = (a ^ a) ^ b = 0 ^ b = b
  + a ^ b ^ b = a ^ (b ^ b) = a ^ 0 = a
* a & (a - 1)
  + set the rightmost 1 to 0

a = 00000000 11111111 00000000 11110000

a - 1 = 00000000 11111111 00000000 11101111

& = 00000000 11111111 00000000 11100000

* Bit Vector
  + use each of the bits to denote “true”(1) or “false”(0).
  + if using integer(32 bits), we can use it as a boolean array with size 32.
  + to denote boolean array with K size, we can use **((K - 1) / 32) + 1** integers.

Example,

size 255 bit vector need 8 ints

size 256 bit vector need 8 ints

size 257 bit vector need 9 ints

### Examples And Practices

**determine whether a number x is a power of 2 (x == 2^n)**

return x > 0 && x & (x - 1) == 0;

**determine how many bits are 1 for an integer x**

int count = 0;

for (int i = 0; i < 32; i++) {

count += (x >> i) & 1;

}

x % 2

x = x / 2; ⇒ x = x >> 1;

int count = 0;

while (x != 0) {

count++;

x &= (x - 1); // decrease the number of 1s by 1

}

**Some More Examples:**

int x; (x >= 0)

x = x / 16 ⇒ x >>= 4

x = x % 16 ⇒ x & ((1 << 4) - 1)

00000000 11111111 11110000 00001111 = x

**determine if letters in a word are all unique, (assuming we are using ASCII character sets, 256 possible characters in total).**

* We can use a HashSet<Character> to record if a char has been used or not
* Since we know that ASCII has 256 characters, we can just use a bit vector with size 256

Key: character (0 - 255) ⇒ index of the array ⇒ 256 bits in bit vector ⇒ 8 integer.

public boolean allUnique(String word) {

assert word != null;

// ASCII char range: 0 - 255

// we need (256 - 1) / 32 + 1 = 8

int[] vector = new int[8];

// byte[] vector = new byte[32];

for (int i = 0; i < word.length(); i++) {

char cur = word.charAt(i);

// cur = 0 - 255

// cur / 32 ⇒ which integer

// cur % 32 ⇒ which bit of the integer

if (vector[cur / 32] >>> (cur % 32) & 1 != 0) {

return false;

}

vector[cur / 32] |= 1 << (cur % 32);

}

return true;

}

## Practice DFS

Need to be very clear about:

* **What does each state means, what status should be maintained**
* **From one state to a deeper level, what is the status change, what are the states we can reach at the deeper level**
* **When we go back to an upper level(method return), if there is any cleanup work needed**
* **How many levels are there, or, what is the termination condition(When can we stop)?**

**All SubSets I** [**http://www.laicode.com/editor/62/**](http://www.laicode.com/editor/62/)

Examples:

null ⇒ []

“” ⇒ [“”]

“abc” ⇒ [“”, “a”, “ab”, “abc”, “ac”, “b”, “bc”, “c”]

at each state, maintain the subset constructed so far, and determine if the character at current level should be added or not.

when all the characters have been determined, we can stop going further.

public List<String> subSets(String set) {

List<String> result = new ArrayList<String>();

if(set == null) {

return result;

}

StringBuilder sb = new StringBuilder();

helper(set, sb, 0, result);

return result;

}

public void helper(String set, StringBuilder sb, int level, List<String> result) {

if(level == set.length()) {

result.add(sb.toString());

return;

}

helper(set, sb.append(set.charAt(level), level + 1, result);

sb.deleteCharAt(sb.length() - 1);

helper(set, sb, level + 1, result);

}

**All SubSets II** [**http://www.laicode.com/editor/63/**](http://www.laicode.com/editor/63/)

Examples:

“abba” ⇒ [“”, “a”, “aa”, “ab”, “aab”, “abb”, “aabb”, “b”, “bb”]

The only thing need to be changed is to determine the number of occurrence of the character, instead of “should be added or not”.

What is the difference?

level ⇒ unique char

“bb”

“”

“” “b” index = 0

“” “b” “b” “bb” index = 1

level → unique character.

“aabbb” → “a”, “b” ⇒ how many “a” I can add, 0, 1, 2.

set → “abba” → “aabb”

public List<String> subSets(String set) {

List<String> result = new ArrayList<String>();

if(set == null) {

return result;

}

char[] array = set.toCharArray();

Arrays.sort(array);

StringBuilder sb = new StringBuilder();

helper(array, sb, 0, result);

return result;

}

“aabb”

public void helper(char[] array, StringBuilder sb, int index, List<String> result) {

if(level == set.length()) {

result.add(sb.toString());

return;

}

int count = 0;

char cur = array[index];

while (index < array.length && array[index] == cur) {

index++;

count++;

}

// how many “a” in total, and what is the index for next unique character.

// 0 - count “a”s, DFS

helper(array, sb, index, result); // 0 “a”

for (int i = 1; i <= count; i++) {

helper(array, sb.append(cur), index, result);

}

for (int i= 1; i <= count; i++) {

sb.deleteCharAt(sb.length() - 1);

}

**}**

# Class 9 1-2 class review

## Java与C/C++的对比

C/C++:

Text file (.cpp, .h) 源文件是文本文件，人理解，计算机不理解(计算机只能理解二进制的可执行文件)

#include<iostream>

void main() {

int a = 1;

}

C/C++编译器：(比如GCC) 将.cpp, .h文件转换为可执行文件，.o 计算机可理解，人不可读

好处：直接可以运行

坏处：非跨平台(Mac, Linux, Windows, etc.)

Java:

Text file(.java) 是文本文件，人理解，计算机不理解

public class Test {

public static void main(String[] args) {

int a = 1;

System.out.println(a);

}

}

Java编译器：（JDK）将.java文件转换为JVM可理解的文件.class，人不可读，计算机需要借助JVM才能执行

好处：可以跨平台，同样编译好的.class可以在不同操作系统平台下执行（比如分布式系统）

坏处：增加了额外的开销，运行效率稍微低一点（但是现在Java已经是优化了很多，这点开销是可接受的）

区别

1. 多重继承 multi-inheritance
2. 面向对象 Object oriented design
3. 指针 vs. 引用 pointer vs. reference
4. 自动内存管理 memory management, garbage collection
5. etc.

## Class (类) vs. Object (对象)

**class** - is like the blueprint for a house. Using this blueprint, you can build as many houses as you like.

**object** - each house you build (or instantiate, in OO lingo) is an **object**, also known as an **instance**.

**reference** - each house also has an address, of course. If you want to tell someone where the house is, you give them a card with the address written on it. That card is the object's **reference**.

类和对象是相对的，比如人可以是一个类，某一个人是一个对象；哺乳动物是一个类，人类作为哺乳动物的一种也可以是一个对象；

初始化(initialization) 函数

public **class** Person {

public **static** String familyName = “Smith”;

private String name;

public Person (String name) { //与类名一致，不需要返回值

this.name = name;

}

public String getName() {

return name; // 也可以写成 return this.name;

}

public void setName(String name) {

this.name = name; //第一个是属于Person的name，第二个是传入进来的name

}

public static void main(String[] args){

Person pA = new Person(“John”);

Person pB = new Person(“Peter”);

System.out.println(pA.getName()); // John

System.out.println(pB.getName()); // Peter

}

}

## static (静态) vs. non-static

静态表明是属于类本身的，从编译器开始运行就一直存在（但是不一定可以访问, private)

非静态是属于对象本身的，需要等到程序执行到初始化该对象的时候才可以访问，没有初始化的非静态成员，不管是变量还是函数都是不可访问的。

Person person; //或者Person person = null; //Person person = null;

person.getName(); //报错，程序无法继续执行

System.out.println(person.name); //报错，首先name是private的，其次非静态

Person.familyName = “Smith”;

//person.familyName = “Smith”;//不好

System.out.println(Person.familyName);

public static void **main**(String[] args) {

}

问题：Java的main函数为什么是要static的？

答案：在java中，没有static的变量或函数，如果想被调用的话，是要先新建一个对象才可以。而main函数作为程序的入口，需要在其它函数实例化之前就 启动，这也就是为什么要加一个static。main函数好比一个门，要探索其它函数要先从门进入程序。static提供了这样一个特性，无需建立对象， 就可以启动。

why public?

why String[] args:

java Person a b c d 1 0.2 true….

args[0] = “a”, args[4] = “1”, ….

## 传值(pass by value)和传引用(pass by reference)

1. 对象就是传引用(相当于C++中的reference &)

public void funA() {

Person person = new Person(“John”);

System.out.println(person.getName());// “John”

funB(person);

System.out.println(person.getName()); //应该输出什么？”Steven”

}

public void funB(Person p) {

//p = person

p.setName(“Steven”);

}

person -> [name=”John”]

p -> [name=”John”]

2. 原始类型(primitive type: int, long, boolean, float, double, etc. )就是传值

public void funA() {

int a = 1;

funB(a);

System.out.println(a); //应该输出什么？实际上还是1

}

public void funB(int b) {

//b = a = 1;

b=3; //如果希望改变a的值并且传出来，如何做？

}

[a=1]

[b=1] [a=1]

[b=3] [a=1]

3.String类型因为没有提供自身修改的函数，每次操作都是新生成一个对象，所以要特殊对待。更接近primitive type可以认为是传值。

public void funA() {

String a = “John”;

a = funB(a);

System.out.println(a); //Smith

}

public String funB(String b) {

//b = a = “John”;

b=”Smith”; //如果希望改变a的值并且传出来，如何做？

return b;

}

[a=John]

[b=Jonn]

[b=Smith]

[a=Smith]

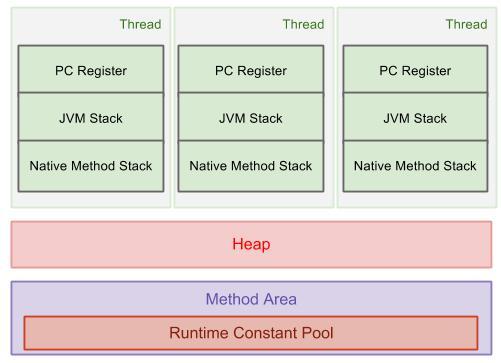
## Memory Management

JVM内存主要有两部分, stack （栈）和heap (堆），基本数据类型，指令（方法）存放在stack中，对象类型则存放于heap中，对象的引用存放在stack中。

stack中的数据基本上属于值传递，heap的对象传递属于引用传递。

stack中的数据可以认为只是拿出来一个copy使用，而heap中的数据是拿出去使用后，再放回来。所以栈数据不会改变，而对象数据会改变。值传递数据不会改变，而引用数据会改变。

code segment, data segment (static)



OutofMemory Error: heap size是说申请不到新的内存了

StackOverflowError: 因为一个线程把stack内存全部耗尽了，一般是递归函数造成的

## inheritance 继承

Java为什么要继承（封装、多态）: 可复用

**Definitions**: A class that is derived from another class is called a subclass (also a derived class, extended class, or child class). The class from which the subclass is derived is called a superclass(also a base class or a parent class).

public class Employee **extends** Person {

private String company;

public void setCompany(String company) { this.company = company;}

public String getCompany() { return company; }

}

//依旧可以调用employee.getName()

Employee e = new Employee(“John”);

String name = e.getName();//John

String company = e.getCompany();

## Override vs. Overload

Override is when you redefine a method that has been defined in a parent class (using the same signature). Resolved at runtime.

public class Employee extends Person {

private String company;

@Override //最好加上，这样编译器会帮你检查

public void setName(String name) {

this.name = this.name + “, ” + this.company;//John, Microsoft

}

public static void main(String[] args){

Employee e = new Employee(“John”);

e.setCompany (“microsoft”);

**e.setName(“Smith”);**

System.out.println(e.getName()); // Smith, Microsoft

}

}

Overload is when you define two methods with the same name, in the same class, distinguished by their signatures (different), Resolved at compile time.

public class Employee extends Person {

private String company;

public void setCompany(String company) {

this.company = company;

}

public void setCompany(String company, String name) {

this.name = name;

this.company = company;

}

public static void main(String[] args) {

Employee e = new Employee (“John”);

e.setCompany(“Microsoft”, “Steven”);

e.setCompany(“Microsoft”);

System.out.println(employee.getName());

}

}

## interface vs. abstract class

A class must be declared abstract when it has one or more abstract methods. A method is declared abstract when it has a method heading, but no body – which means that an abstract method has no implementation code inside curly braces like normal methods do.

in Figure.java

public **abstract** class Figure  
{

pri  
 public **abstract** float getArea() ;

public void print(){

System.out.println(“This is a Figure”);  
}

in Circle.java  
public class Circle **extends** Figure  
{  
 private float radius;  
 public float getArea()  
 {  
 return (3.14 \* (radius \* 2));   
 }  
}

in Rectangle.java  
public class Rectangle **extends** Figure  
{  
 private float length, width;  
 public float getArea()  
 {  
 return length \* width;  
 }  
}

An interface differs from an abstract class has only ‘abstract’ methods.

in Dog.java

public **interface** Dog  
{  
 public boolean isCute();

}

in Husky.java

public class Husky **implements** Dog  
{  
 public boolean isCute{  
 return true;  
 }  
}

Difference: 1. Java does not allow multiple inheritance. In Java, a class can only derive from one class, whether it’s abstract or not. However, a class can implement multiple interfaces.

public class ClassA {

public void printOut() {

System.out.println(“hi, it is classA”);

}

}

public class ClassB {

public void printOut() {

System.out.println(“hi, it is classB”);

}

}

（假设允许类与类之间多继承）

public class ClassC extends ClassA, ClassB {

public static void main(String[] args) {

ClassA classA = new ClassC();

classA.printOut(); // “hi it is classA”

ClassB classB = new ClassB();

classB.printOut(); // “hi it is classB”

ClassC classC = new ClassC();

classC.printOut(); // ???

}

}

但是对于interface没有问题

public interface ClassA {

public void printOut() ;

}

public interface ClassB {

public void printOut() ;

}

（implements多个interfaces是允许的）

public class ClassC implements ClassA, ClassB {

public void printOut() {

System.out.println(“hi it is classC”);

}

public static void main(String[] args) {

ClassC classC = new ClassC();

classC.printOut(); //hi it is classC

}

}

2. An abstract class may provide some methods with definitions – so an abstract class can have non-abstract methods with actual implementation details. An abstract class can also have constructors and instance variables as well. An interface, however, can not provide any method definitions – it can only provide method headings. Any class that implements the interface is responsible for providing the method definition/implementation.

3. abstract : is a ; interface: has a function

Ebay Interview Question: An abstract class is good if you think you will plan on using inheritance since it provides a common base class implementation to derived classes.

An abstract class is also good if you want to be able to declare non-public members. In an interface, all methods must be public.If you think you will need to add methods in the future, then an abstract class is a better choice.

Because if you add new method headings to an interface, then all of the classes that already implement that interface will have to be changed to implement the new methods.

Interfaces are a good choice when you think that the API will not change for a while. Interfaces are also good when you want to have something similar to multiple inheritance, since you can implement multiple interfaces.

## 访问控制 public, private, protected

public - everyone can access

private - only myself can access 但是只是相对于class level的

protected - only my children and same package can access

default - only the same package can access

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Modifier | Class | Package | Subclass | World |
| public | Y | Y | Y | Y |
| protected | Y | Y | Y | N |
| no modifier | Y | Y | N | N |
| private | Y | N | N | N |

public class Person {

private String name;

public Person(String name) { this.name = name; }

public void setName(Person another) {

this.name = another.name;//class level

}

public String getName() { return this.name; }

public static void main(String[] args) {

Person person1 = new Person("John");

person1.name//error private access

Person person2 = new Person("Steven");

person2.setName(person1);

System.out.println(person2.getName());

}

}

Singleton : the Singleton pattern will ensure that there is only one instance of a class is created in the Java Virtual Machine. It is used to provide global point of access to the object.

public class ClassicSingleton {  
 private static ClassicSingleton instance = null;

private String name ;  
 private ClassicSingleton() {  
 // Exists only to defeat instantiation.

this.name = “John”;  
 }  
 public static ClassicSingleton getInstance() {

if (instance == null){

instance = new ...  
 }

return instance;  
 }

public static void main(String[] args) {

ClassicSingleton c = new ClassicSingleton();//wrong no

ClassicSingleton c1 = ClassicSingleton.getInstance();

ClassicSingleton c2 = ClassicSingleton.getInstance();

//wrong implementation

ClassicSingleton c1 = new …

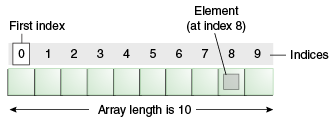
ClassicSingleton c2 = new …

c1.getInstance() != c2.getInstance()

}  
}

## 基本的数据结构及其基本操作

首先什么是内存，可以想象是一排房间，里边存有旅客（数据），如果有门牌号（地址），可以一次性直接访问，视为一次内存操作，不管其位置远近（地址大小）



Array: int[] : 按顺序存储，有索引，读写操作直接作用于存储单元，所以get set（读写）操作都是一次完成，单位时间长度，基本是恒定的

int[] a = new int[10];

a[0] = 100;

a[9] = 200;

a[10] = 300;//error

访问控制 public, private, protected

public - everyone can access

private - only myself can access 但是只是相对于class level的

protected - only my children and same package can access

default - only the same package can access

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Modifier | Class | Package | Subclass | World |
| public | Y | Y | Y | Y |
| protected | Y | Y | Y | N |
| no modifier | Y | Y | N | N |
| private | Y | N | N | N |

public class Person {

protected String name;

public Person(String name) { this.name = name; }

public void setName(Person person) {

this.name = person.name;

}

public String getName() { return this.name; }

public static void main(String[] args) {

Person person1 = new Person("John");

person1.name

Person person2 = new Person("Steven");

person2.setName(person1);

System.out.println(person2.getName());

}

}

public class Animal {

public void fun(Person person) {

String n = person.name //private//error

String n = person.getName() // public //ok

}

}

Project/

Person.java

Animal.java

Singleton : the Singleton pattern will ensure that there is only one instance of a class is created in the Java Virtual Machine. It is used to provide global point of access to the object.

public class ClassicSingleton {  
 private static ClassicSingleton instance = new ClassicSingleton();

private String name ;  
 private ClassicSingleton() {  
 // Exists only to defeat instantiation.

this.name = “John”;  
 }  
 public static ClassicSingleton getInstance() {  
 return instance;  
 }

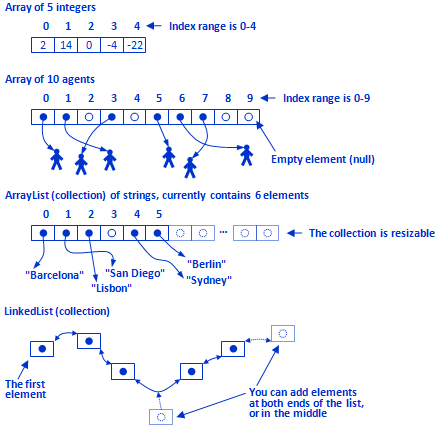
public static void main(String[] args) {

ClassSingleton c = new ClassSingleton();//wrong no

ClassSingleton c1 = ClassSingleton.getInstance();

ClassSingleton c2 = ClassSingleton.getInstance();

}  
}



List: ArrayList

和Array类似，但是大小可自动调整

ArrayList<Integer> list = new ArrayList<Integer>();//int

list.size(); // return 0

list.add(1); // add at end, {1}

list.size(); // 1

//index, value

list.add(0, 2); // add at head, {2, 1}

list.size(); // 2

list.add(1, 3); // add at middle, {2, 3, 1}

list.get(1); // 3

list.remove(1); // {2, 1}

list.set(0, 4); // {4, 1}

Map<K, V> {

K key;

V value;

}

Map<Integer, String> map;

Doubly LinkedList 双向链表

class ListNode<E> {

E e;//int, double, float,

ListNode<E> prev;

ListNode<E> next;

}

class LinkedList<E> {

ListNode<E> head;

ListNode<E> tail;

int size;//

add() {

size ++;

}

remove() {

size --;

}

}

LinkedList<Integer> list = new LinkedList<Integer>();//E=Integer, Double, Boolean, Float, ...

list.size(); // 0

list.add(1); // add at end, {1}

list.size(); // 1

list.add(0, 2); // add at head, {2, 1}

list.size(); // 2

list.add(1, 3); // add at middle, {2, 3, 1}

list.get(1); // 3

list.remove(1); // {2, 1}

list.set(0, 4); // {4, 1}

get(int index) - from the head/tail, traverse the list to get the element at index

set(int index, E e) - from the head/tail, traverse the list to the get the element at index, and change it

add()

* at tail - append new node after tail and update tail to the new added node
* at head - insert in front of the head, and update head to the new added node
* at middle -

remove()

* at head - update the head to head.next
* at tail - update the tail to tail.prev
* at middle - from head/tail, traverse the list to get the element at index, and remove it.

### Single LinkedList, Sort

Selection Sort

[solution](https://docs.google.com/document/d/1Oaq3kQMCZNZZvAcgkN9kgAzmDRfGo3KQI-r2Yf0aRQ4/edit#heading=h.utzx1tfrud6)

Merge Sort

[solution](https://docs.google.com/document/d/1Oaq3kQMCZNZZvAcgkN9kgAzmDRfGo3KQI-r2Yf0aRQ4/edit#heading=h.yw5bu9aajbry)

Quick Sort

[solution](https://docs.google.com/document/d/1Oaq3kQMCZNZZvAcgkN9kgAzmDRfGo3KQI-r2Yf0aRQ4/edit#heading=h.hlj84p2tts46)

Linked List(Singly)

How to represent a singly linked list?

[ next] ->[ next] -> [ next] -> null

class ListNode {

int value;

ListNode next;

//ListNode prev; // Doubly Linked List

public ListNode(int value) {

this.value = value;

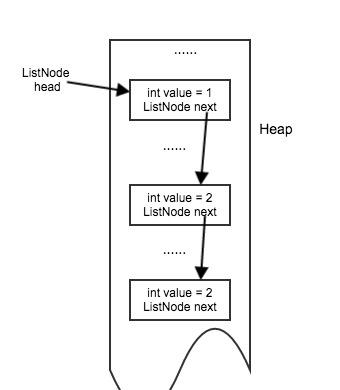
}

}

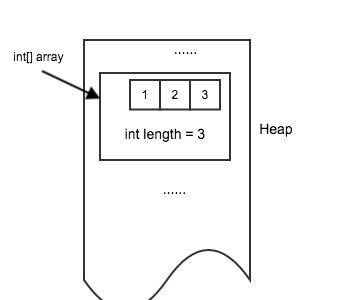
ListNode head = new ListNode(1);

head.next = new ListNode(2);

head.next.next = new ListNode(3);



Array



Examples of Linked List Operations:

int length(ListNode head) {

int length = 0;

while (head != null) {

length++;

head = head.next;

}

return length;

}

1 → 2 → 3 → null

ListNode get(ListNode head, int index) {

while (head != null && index > 0) {

head = head.next;

index--;

}

return head;

}

1 → 2 → 3 → null

0 1 2

ListNode appendHead(ListNode head, int value) {

ListNode newHead = new ListNode(value);

newHead.next = head;

return newHead;

}

1 → 2 → 3 → 4 → null

p

ListNode appendTail(ListNode head, int value) {

ListNode newNode = new ListNode(value);

if (head == null) {

return newNode;

}

// we know head is not null here, and we need to know the last non-null node so that we

// can append after it. so we can use while(head.next != null)

ListNode prev = head;

while (prev.next != null) {

prev = prev.next;

}

// now prev is the last non-null node.

prev.next = newNode;

return head;

}

**head could be changed - we need to return the new head if changed**

**usually need to consider**

* **head == null**
* **head.next == null**

**while loop - we need to understand what is the terminate condition**

**use head/head.next in the condition**

ListNode search(ListNode head, int value) {

while (head != null) {

if (head.value == value) {

return head;

}

head = head.next;

}

return null;

}

ListNode remove(ListNode head, int value) {

// 1.

if (head == null) {

return head;

}

// 2.

if (head.value == value) {

return head.next;

}

// 3.

ListNode prev = head;

while (prev.next != null && prev.next.value != value) {

prev = prev.next;

}

if (prev.next != null) {

prev.next = prev.next.next;

}

return head;

}

1 → 2 → 3 → null

p

## 复杂度 complexity

评价算法的度量：

内存操作时间和数据量的关系可以表达成为一个数学函数，然后取其数量级（考虑在大数据量的情况下的表现）

1. 只考虑最大的；2.不考虑常数

f(n) = 1000, O(f(n)) = O(1)

f(n) = 100n, O(f(n)) = O(n)

f(n)= 10n^2 + 1000n, O(f(n)) = O(n^2)

常用的复杂度关系 logn < n < n^2 < n^3 … < 2^n

好<----------------------------->坏

分析一下ArrayList和LinkedList的各种操作的复杂度

add (int index, int value)

ArrayList has n ints.

0...n/2, n/2+1, n/2+2, ...n-2, n-1 (总共是n)

//set value O(1) at index = n/2

//n/2, n/2 + 1, n/2 + 2… n-1 O(n/2)

O(n/2) + O(1) = O(n/2) = O(n)

LinkedList has n ints,

0...n/2, n/2+1, n/2+2, ...n-2, n-1

head: O(1),

get(n/2), ListNode mid = head.next.next.next (n/2) ; O(n/2)= O(n)

add(0),

head ...

ListNode newHead = new ListNode(100);

newHead.next = head;

if (head != null) {

head.prev = newHead;

}

head = newHead;

O(1)

Time Complexity Analysis:

|  |  |  |
| --- | --- | --- |
| Operation | ArrayList | LinkedList |
| get(int index) at head/tail | O(1) | O(1) |
| get(int index) in middle | O(1) | O(n) |
| set(int index) at head/tail | O(1) | O(1) |
| set(int index) in middle | O(1) | O(n) |
| add(int index, E e) in middle | O(n) | O(n) |
| add(int index, E e) at head | O(n) | O(1) |
| add(E e) at tail | worst O(n)  amortized O(1) | O(1) |
| remove(int index) at head | O(n) | O(1) |
| remove(int index) at tail | O(1) | O(1) |
| remove(int index) at middle | O(n) | O(n) |
| size() | O(1) | O(1) |
| isEmpty() | O(1) | O(1) |

class Person implements Comparable<Person>{

int age;

@Override

int compareTo(Person another){

return this.age - another.age;

}

}

Collections.sort(List<Person> list);

Collections.sort(List<Person> list, new Comparator<Person>()

{

@Override

int compare(Person a, Person b){//

// > 0: a > b

// < 0; a < b

return a.age - b.age;

}

}

);