### 数组：

招式

定义：

静态定义

|  |
| --- |
| int[] intArr;  intArr = new int[]{1, 2, 3, 4, 5, 6, 9, 22, 10, 89, -1, -99, 0};  //此外还可以简化写法  String[] objArr2 = {"we","are","learning","JAVA", "."}; |

动态定义

|  |
| --- |
| //声明数组变量是固定长度  int[] intArr2 = new int[5];  for(int i = 0; i < intArr2.length; i++) {  intArr2[i] = i + 1;  } |

数组常用的遍历方法

|  |
| --- |
| for(int i = 0; i < intArr2.length; i++)  System.out.println(intArr2[i]);    for(int a : intArr2)  System.out.println(a); |

数组标识符实际就是指向真实对象的一个引用，那些对象存在于内存中的堆中。

心法：

java中不存在多维数组，在内存中都是以线性数组的形式存在。

数组与集合的区别：

1. 集合只能装载引用类型，数组可以装备基本类型
2. 正因为原因1，如果装载基本类型，集合只能使用对应的封装类型，在访问效率上，由于数组的元素在内存中是连续的，所以效率高高于使用集合。
3. 数组功能弱，定义时必须固定长度

练习：

<https://www.hackerrank.com/challenges/camelcase/problem>

<https://www.hackerrank.com/challenges/angry-professor/problem?h_r=profile>

<https://www.hackerrank.com/challenges/beautiful-days-at-the-movies/problem>

<https://www.hackerrank.com/challenges/arrays-ds/problem>

<https://www.hackerrank.com/challenges/sparse-arrays/problem>

<https://www.hackerrank.com/challenges/mars-exploration/problem>

<https://www.hackerrank.com/challenges/strong-password/problem>

<https://www.hackerrank.com/challenges/2d-array/problem>

### 类：

字符：人类可以读懂的语言

字节：机器可以识别的“语言”，专业术语-机器指令

编译：可以类似认为人类的语言翻译成机器指令

在eclipse/STS项目目录下，有src存放的是源文件，bin则放的是编译后的二进制文件

Java的程序编译后才能够运行

类的封装规则：

priavte/default/protected/public

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **private** | **default** | **protected** | **public** |
| 同一个类中 | ○ | ○ | ○ | ○ |
| 同一个包中 |  | ○ | ○ | ○ |
| 子类中 |  |  | ○ | ○ |
| 全局范围 |  |  |  | ○ |
|  | 类中 | 包中 | 子类中 | 全部 |

方法重写： 子类覆盖父类的方法

方法重载： 相同方法名，不同参数列表，列如

|  |
| --- |
| public void print() {  System.out.println("I'm a sub-class");    }    public void print(String abc) {  System.out.println(abc);  }    public void print(int abc) {  System.out.println(abc);  }    public void print(String abc, int bcd) {  System.out.println(abc + bcd);  } |

变量的作用域与生命周期：

变量的生命周期

|  |
| --- |
| {  int a = 5;//变量a的作用域仅限在花括号里  }//变量a的生命结束 |

子类构造器的执行顺序

|  |
| --- |
| /\*\*  \* 1. 调用父类的构造器HelloWorld()  \* 2.调用Object的构造器 Object()  \* 3. 执行完Step2  \* 4.执行完Step1  \* 5.执行子类的构造器 HelloWorld2()  \* 6.初始化对象变量 Integer abc = 1  \*  \* 简单一句话，生成子类对象前要先生成父类对象  \*  \*/  HelloWorld2 w2 = new HelloWorld2(); |
|  |

### final

final修饰的类不能被继承，同样父类中的方法如果被final修饰，子类不能够重写

心法：

final 修饰的变量必须在使用时已经赋值了。

所以final修饰的类的成员变量，可以在**1.声明时赋值2.类的构造器内赋值**。 以上两点都保证了类的对象生成时，final的成员变量也完成赋值了。

但是static final 修饰的类的成员变量，必须在**1.声明时赋值2.类中static代码块赋值**，因为static 的静态变量在类的加载时（对象生成前）就调用了。

final修饰的引用变量，例如Map, 在Map<> map = new HashMap<>(),后可以对map随意put()/remove()等，因为map的对象没变，只是内部引用数据改变。但是不能之后再 map = new HashMap<>(); 因为这样map的对象就变化了。

所以private final Map<String, String> map = new HashMap<>();

之后，还可以进行map的put/remove,如何禁止呢？可以使用Collections的内部类UnmodifiableMap

final Map<> map = Collections.unmodifiableMap(values)

其实就是继承Map，并重写put/remove方法，使其抛出异常。

抽象类： 可以含有抽象方法，及只有方法签名没有方法的具体实现,抽象类要用abstract关键字

接口：类似抽象类，可以含有抽象方法，但是可以不用abstract修饰符

一个类只能继承一个父类，但是可以实现多个接口

归并排序：

<https://blog.csdn.net/qq_28081081/article/details/80603251>

Enum枚举类

1.枚举类默认使用final修饰符，即不能够诞生子类

2.成员变量使用final修饰符，即一般在内部调用构造器时进行赋值

3. 构造器使用private修饰符，即只能在类的内部通过构造器生成枚举对象

HashSet

hashSet集合判断两个**元素相等**的标准是通过两个对象的equals()和hashCode()方法比较，只有全部相等才视为同一元素。所以对自定义类设计时应注意保持equals()和hashCode()一致。

即，equals返回true，并且hashCode相等。

Set实际上就是通过将Map的value设为null,保留key.

ArrayList

1. 底层就是数组：Object[] elementData;
2. 第一次添加元素的时候，会默认扩容致 private static final int DEFAULT\_CAPACITY = 10;
3. 之后会扩容至1.5倍 oldCapacity + (oldCapacity >> 1);
4. ArrayList中的迭代器，当使用该迭代器时，不能够调用ArrayList的remove方法，否则会发生ConcurrentModificationException，因为modAccount和迭代器的expectedModCount没同步。所以应该使用Iterator的remove方法。

HashMap

扩容发生在第一次put元素，默认大小为16，门槛是75%即12.

newCap = DEFAULT\_INITIAL\_CAPACITY; // DEFAULT\_INITIAL\_CAPACITY = 1 << 4;

newThr = (int)(DEFAULT\_LOAD\_FACTOR \* DEFAULT\_INITIAL\_CAPACITY);

存放元素数组的index是通过(n - 1) & hash， n是数组的长度，hash是hashcode（默认Object的方法）的低16位异或高16位的结果。

如果出现index冲突，则变成链表，在链表末尾添加新的元素。如果链表中的元素大于等于TREEIFY\_THRESHOLD（8） - 1=7，则把链表树化，即变为红黑树结构。

之后，每次扩容都为原来的2倍oldCap << 1

HashSet

创建HashSet对象的时候实际上是创建一个HashMap对象，因为HashSet的一个成员变量是HashMap类型的。

HashSet的添加一个元素e的过程实际上就是往HashMap里面添加(e, new Object);

这种HashMap作为HashSet的一个成员变量的设计模式常称作组合模式，在实际项目中经常运用。

所以HashSet的底层实现就是HashMap

具体案例

[https://www.hackerrank.com/challenges/ctci-bfs-shortest-reach/problem?h\_l=interview&playlist\_slugs%5B%5D=interview-preparation-kit&playlist\_slugs%5B%5D=graphs](https://www.hackerrank.com/challenges/ctci-bfs-shortest-reach/problem?h_l=interview&playlist_slugs[]=interview-preparation-kit&playlist_slugs[]=graphs)

思考

1. 如何能够通过一个点找到和它相连的点。如果存储：

可以使用HashMap<Integer,ArrayList<Integer>>,这样可以高效地找到所有链接的点。

1. 随着往前走，新的起点可能是多个。如何存储：ArrayList<Integer>
2. 如何不走回头路？所以要记载走过的点。如何存储走过的点？走过的点肯定不重复，所以使用HashSet<Integer>

多线程

多线程的创建方式：

**第一种**：

1. 通过继承Thread父类并重写run方法生成线程
2. 通过实现Runnable接口并重写run方法生成具体实现类的对象，然后传递给Thread(Runnable runnable)的构造器生成线程

实际上以上两种方法没有区别，因为Thread**也是实现了Runnable接口、**

**第二种：**

通过Callable的方式创建，可以通过FutureTask的get方法获取线程返回结果

**join()方法**

Thread提供了让一个线程等待另一个线程完成的方法，只要调用需要先执行线程的join方法，主线程则等待其执行完毕再继续执行。

**守护线程**

后台线程有个特征：如果所有的前台线程都死亡，后台线程会自动死亡。

调用Thread对象setDaemon(true)方法可将指定线程设置成后台线程，并且必须在start()方法之前调用。

**线程状态**

public enum State {

/\*\*

\* Thread state for a thread which has not yet started.

\*/

NEW,

/\*\*

\* Thread state for a runnable thread. A thread in the runnable

\* state is executing in the Java virtual machine but it may

\* be waiting for other resources from the operating system

\* such as processor.

\*/

RUNNABLE,

/\*\*

\* Thread state for a thread blocked waiting for a monitor lock.

\* A thread in the blocked state is waiting for a monitor lock

\* to enter a synchronized block/method or

\* reenter a synchronized block/method after calling

\* {@link Object#wait() Object.wait}.

\*/

BLOCKED,

/\*\*

\* Thread state for a waiting thread.

\* A thread is in the waiting state due to calling one of the

\* following methods:

\* <ul>

\* <li>{@link Object#wait() Object.wait} with no timeout</li>

\* <li>{@link #join() Thread.join} with no timeout</li>

\* <li>{@link LockSupport#park() LockSupport.park}</li>

\* </ul>

\*

\* <p>A thread in the waiting state is waiting for another thread to

\* perform a particular action.

\*

\* For example, a thread that has called {@code Object.wait()}

\* on an object is waiting for another thread to call

\* {@code Object.notify()} or {@code Object.notifyAll()} on

\* that object. A thread that has called {@code Thread.join()}

\* is waiting for a specified thread to terminate.

\*/

WAITING,

/\*\*

\* Thread state for a waiting thread with a specified waiting time.

\* A thread is in the timed waiting state due to calling one of

\* the following methods with a specified positive waiting time:

\* <ul>

\* <li>{@link #sleep Thread.sleep}</li>

\* <li>{@link Object#wait(long) Object.wait} with timeout</li>

\* <li>{@link #join(long) Thread.join} with timeout</li>

\* <li>{@link LockSupport#parkNanos LockSupport.parkNanos}</li>

\* <li>{@link LockSupport#parkUntil LockSupport.parkUntil}</li>

\* </ul>

\*/

TIMED\_WAITING,

/\*\*

\* Thread state for a terminated thread.

\* The thread has completed execution.

\*/

TERMINATED;

}

### ReentrantLock 重入锁

举例 java.util.concurrent.ArrayBlockingQueue.put(E) 中的方法

|  |  |
| --- | --- |
| public void put(E e) throws InterruptedException {  Objects.requireNonNull(e);  **final ReentrantLock lock = this.lock;**  **lock.lockInterruptibly();**  try {  while (count == items.length)  notFull.await();  enqueue(e);  } finally {  **lock.unlock();//unlock一定要写到finally里面**  }  } | public void put(E e) throws InterruptedException {  Objects.requireNonNull(e);  Synchronized(this){    while (count == items.length)  notFull.await();  enqueue(e);    }  } |

### Thread的方法中sleep和wait的区别

1、这两个方法来自不同的类分别是Thread和Object

2、最主要是sleep方法没有释放锁，而wait方法释放了锁，使得其他线程可以使用同步控制块或者方法。

3、wait，notify和notifyAll只能在同步控制方法或者同步控制块里面使用，而sleep可以在任何地方使用（使用范围）

4、sleep必须捕获异常，而wait，notify和notifyAll不需要捕获异常

5、sleep是Thread类的静态方法。sleep的作用是让线程休眠制定的时间，在时间到达时恢复，也就是说sleep将在接到时间到达事件事恢复线程执行。wait是Object的方法，也就是说可以对任意一个对象调用wait方法，调用wait方法将会将调用者的线程挂起，直到其他线程调用同一个对象的notify方法才会重新激活调用者

**notify()/notifyAll()和wait()**

notify()/notifyAll()和wait()必须在同一个对象上建立通信并且要被synchronized包裹，如

Object obj = new Object();

synchronized(obj){

obj.wait();

....

}

....

synchronized(obj){

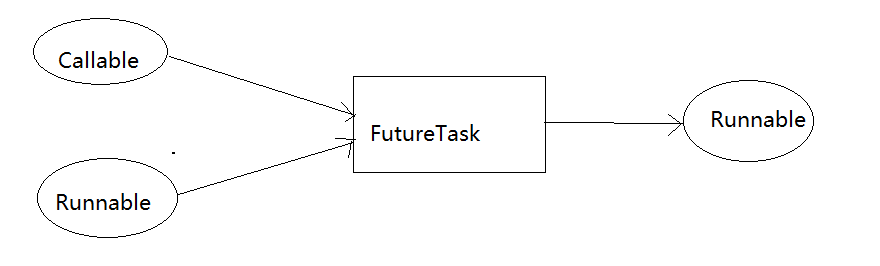
obj.notify();

...

}

### 关于FutureTask

FutureTask实际上继承了Runnable和Future两个接口。 FutureTask可以将Runnable和Callable统一称为Runnable



FutureTask里面有一个成员变量是Callable，通过构造器进来的Callable和Runnable都会赋给这个变量，但是Runnable怎么会赋给Callable变量呢？类型怎么转化呢？

public FutureTask(Runnable runnable, V result) {

this.callable = Executors.callable(runnable, result);

this.state = NEW; // ensure visibility of callable

}

上面的Executors.callable就完成了Runnable向Callable的转化，具体：

通过RunnableAdapter实现的转化：

RunnableAdapter本身是Callable的实现类，又具有Runnable成员变量，在调用Callable的call方法时实际调用Runnable的run方法。

这种本身实现要转化的目的类型接口，又拥有待转化类型的成员变量，通过这个成员变量运行实际的目的方法，就是典型的 - 适配器模式。

|  |
| --- |
| private static final class RunnableAdapter<T> implements Callable<T> {  private final Runnable task;  private final T result;  RunnableAdapter(Runnable task, T result) {  this.task = task;  this.result = result;  }  public T call() {  task.run();  return result;  }  } |