面向对象第六次实验指导书

实验仓库

> 公共信息发布区：exp6_public。

> 个人仓库：oo_homework_2022 / homework_2022你的学号exp_6。

> 请同学们作答完成后，将json文件与代码**提交到个人仓库**，并将json文件内容**填入内容提交区**,再在本页面下方选择对应的 `commit`，最后**点击最下方的按钮进行提交**。详细的提交过程请参见提交事项章节。

实验目标

1. 进一步训练根据 JML 补充代码及撰写 JML 的能力
2. 了解 Java 的垃圾回收机制

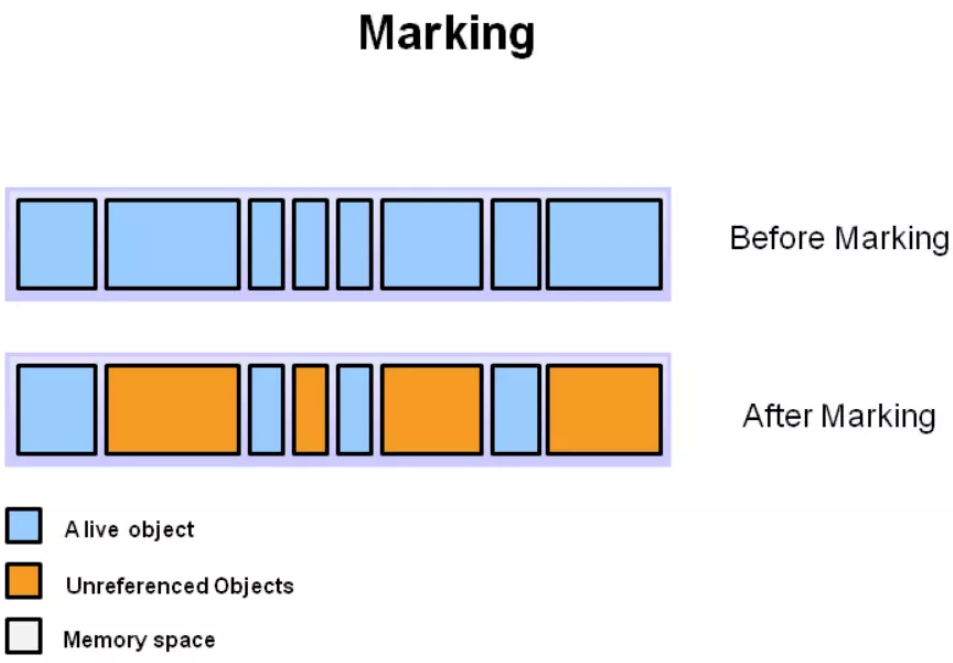
背景引入

与 C++ 程序设计语言相比，Java 程序设计语言拥有一个独特的语言特性——自动垃圾回收机制 (Garbage Collection)。在 Java 和 C++ 中，新创建一个对象都需要使用 `new` 运算符。然而，在 C++ 中，程序员需要人工管理内存，对于不再使用的对象使用 `delete` 运算符显式地回收内存；在 Java 中，程序员无需人工管理内存，JVM 会自动触发垃圾回收，将没有被引用的对象占据的内存空间释放。

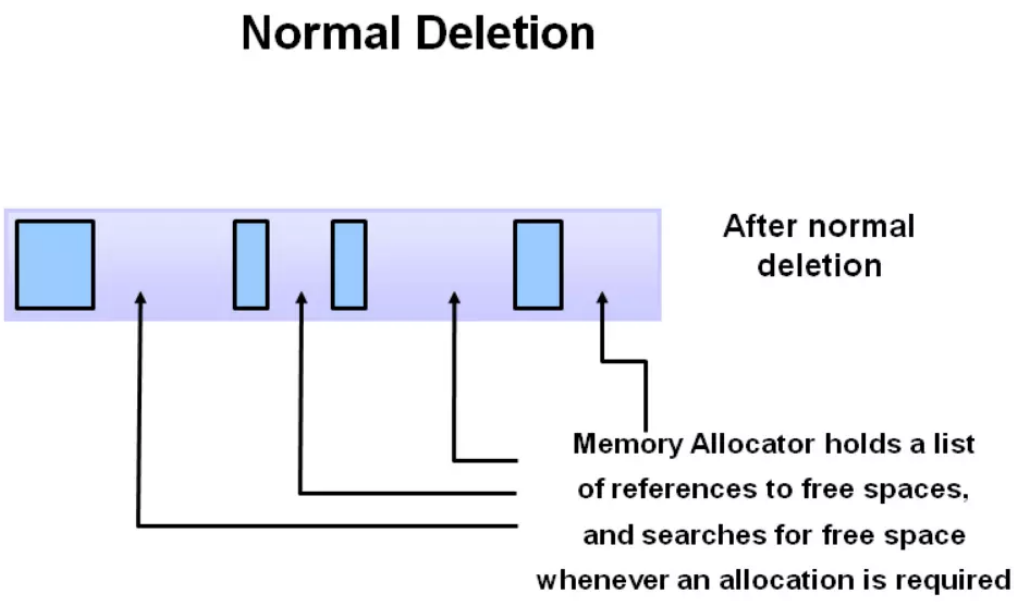
基本垃圾回收机制

基本的 Java 垃圾回收机制如下：

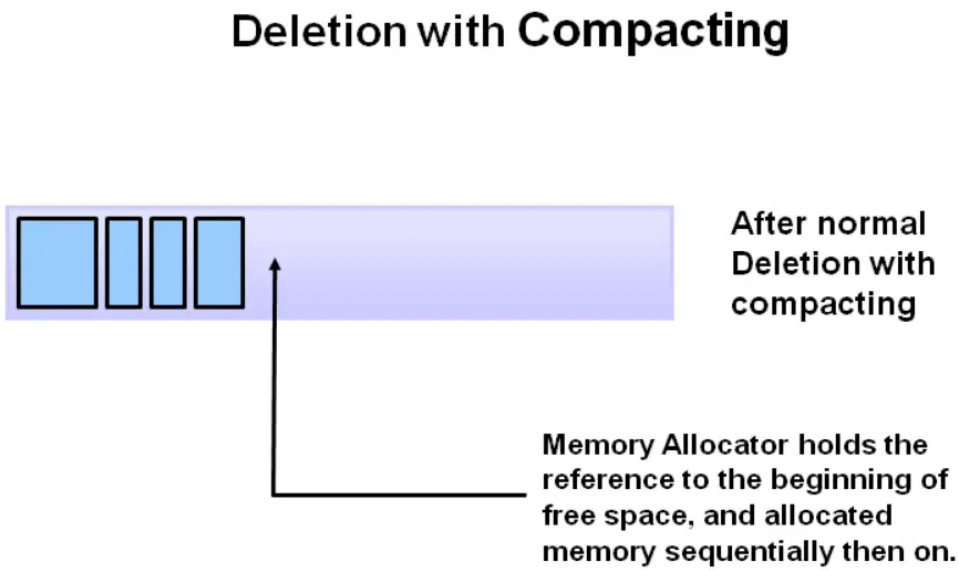
首先，垃圾回收器会找出当前哪些对象是正在使用中的，并将其标记为存活对象；以及哪些对象是没有被引用的，并将其标记为未引用对象，这一步称为**标记** 。下图显示了一个标记前后的内存图的样式：



其次，垃圾回收器会将当前所有未引用对象删除，也就是上图中橙色的部分。



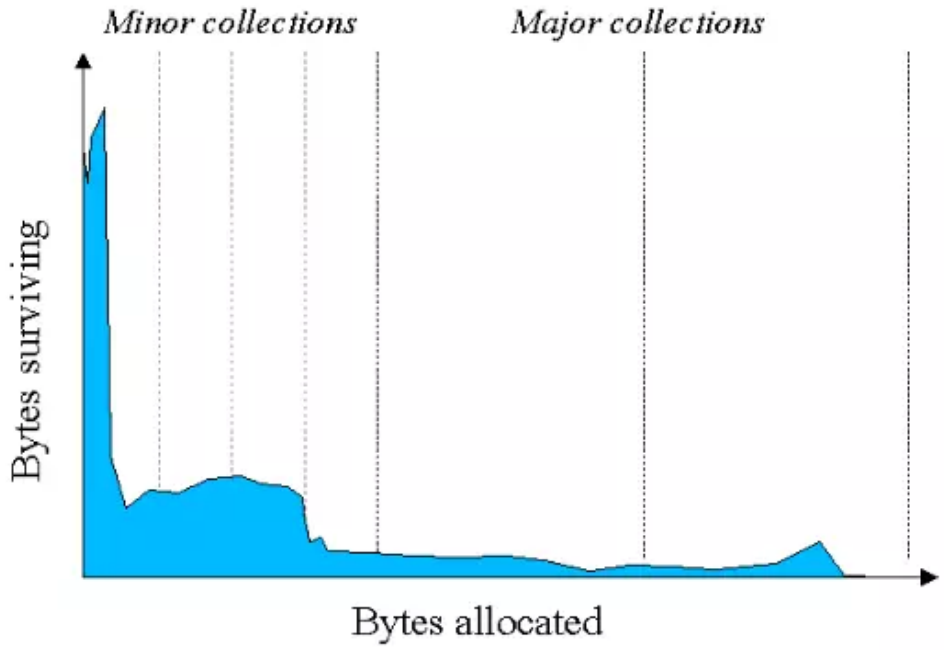
最后，为了提升性能，在删除完未引用对象后，通常还会采取**压缩**操作，将内存中的存活对象放置在一起，以便后续能够更加高效快捷地分配新的对象。



分代垃圾回收机制

在实际的程序中，如果完全采用上面的基本垃圾回收机制，会导致垃圾回收非常低效，这是因为每一次垃圾回收都需要标记所有的对象并进行删除和压缩；垃圾回收的耗时与分配的对象数量成正相关的联系。

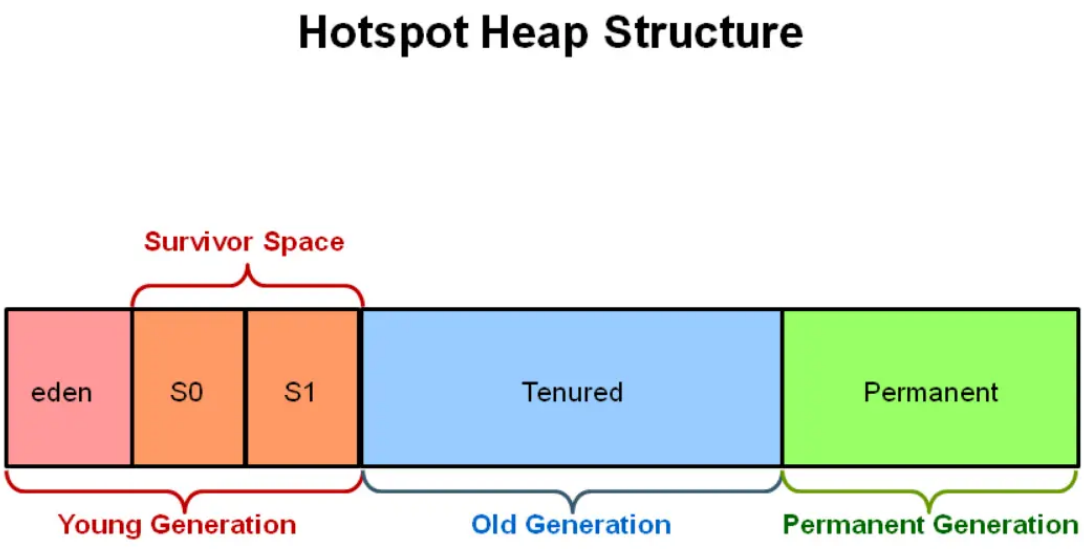
实际上，对一个程序运行过程中所有对象的存活时间进行统计，可以得到下面的图：



横轴代表程序运行时间，纵轴代表分配的字节

从图中我们可以看出，大部分对象的存活时间都比较短(聚集在左侧)，存活的对象随着程序的运行逐渐减少，因此，利用对象存活时间的规律对内存中的对象进行分代，可以加快垃圾回收的效率。

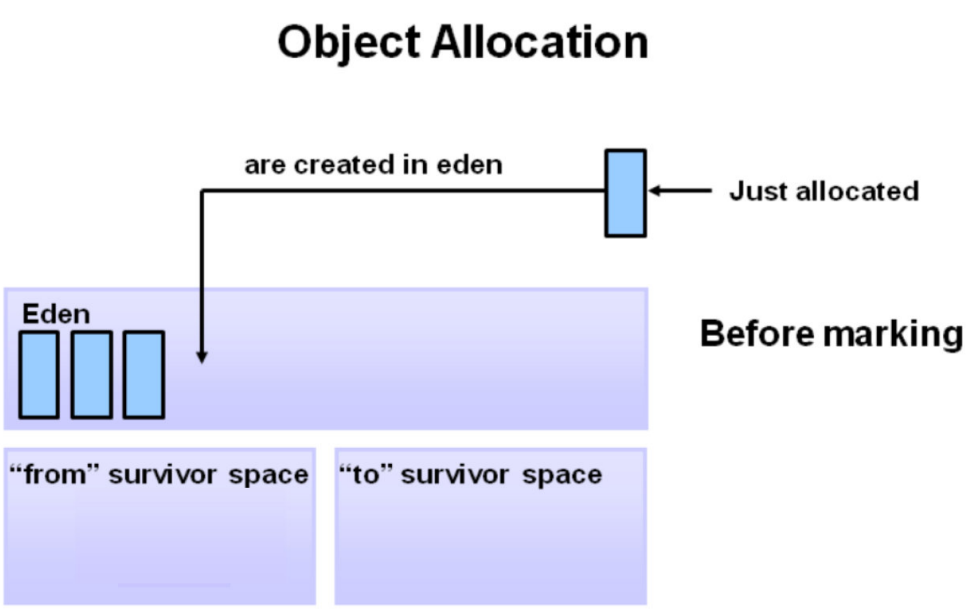
JVM的分代将堆分为如下几个部分：



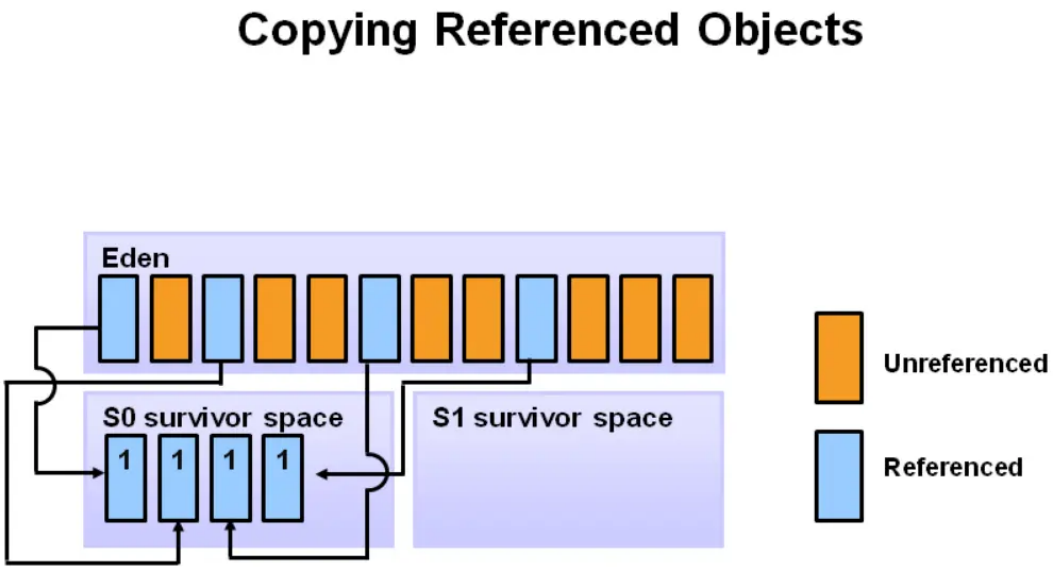
图中红色部分和橙色部分为新生代，用来存储刚分配的对象和分配不久的对象；蓝色部分为老年代，用来存储存活了一定时期的对象；绿色部分为永久代，主要用来存放类和元数据的信息。

在 JVM 分代的设计下，垃圾回收被重新设计为如下过程：

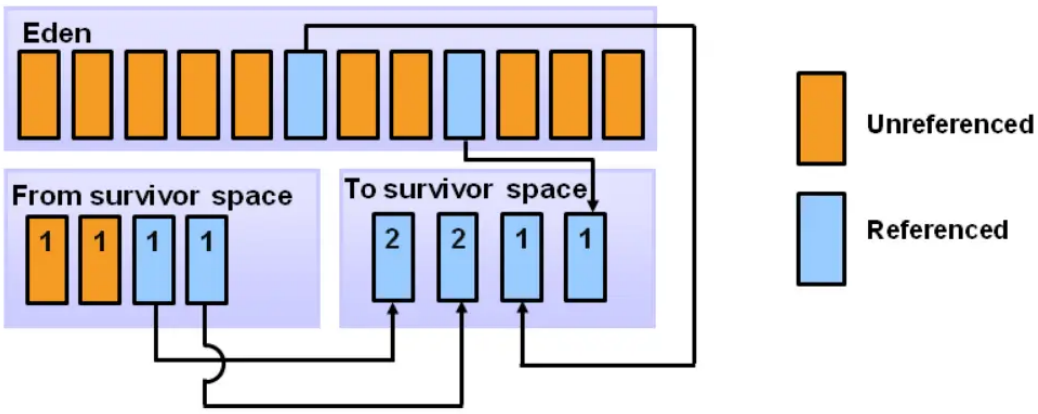
首先，任何新分配的对象都存放于 `eden` 内存中，此时两个 `Survivor` 都是空的。



当新分配的对象达到一定数量时，会将 `eden` 的空间填满，此时会触发**次垃圾回收(小型垃圾回收)**，我们称之为 `MinorGC`。具体地，`MinorGC` 采用的是**标记-复制算法**，首先对 `eden` 和 `FromSurvivorSpace` 中的对象进行标记，然后将存活对象复制到 `ToSurvivorSpace` 中去，随之清空 `eden` 和 `FromSurvivorSpace` 中的对象，并将 `FromSurvivorSpace` 和 `ToSurvivorSpace` 区域调换，如下图所示：

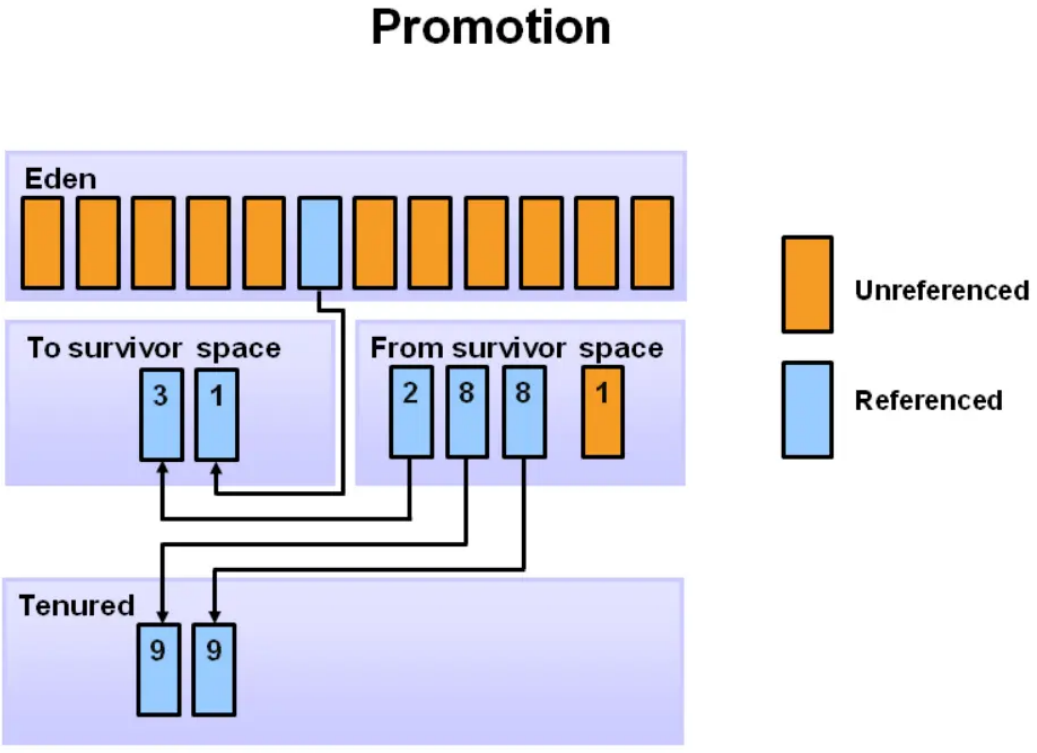


在下一次的 `MinorGC` 时，会重复同样的操作，`Survivor` 区会再次发生交换：



注意到：从 `eden` 区迁移到 `Survivor` 区的对象此时开始有年龄 `Age` 的概念，这里的 `Age` 是用来表示对象的存活时间，每经过一次 `MinorGC`，对象的 `Age` 增加 `1`。

经过了一定次数的 `MinorGC` 后，有些对象的年龄会达到一定的阈值，图中示例为 `8`，此时这些年龄达到阈值的对象在**下一次** `MinorGC` 中会被转移到老年区 `tenured` 中，表示为常使用的对象：



> 实际上不一定只有当对象的年龄达到阈值才会被移动至老年区`tenured`，比如`Survivor`区空间不足等等都可能使得对象在不满足年龄阈值的情况下移动至老年区`tenured`。但是为了简便起见，这里我们不考虑这些情况。

对于老年代中的对象而言，未引用的对象不会在 `MinorGC` 中被回收，而是在**主垃圾回收 (大型垃圾回收)**，我们称之为 `MajorGC` 中被回收。

`MinorGC` 的作用范围是新生代，`MajorGC` 的作用范围是老年代，`MinorGC` 发生的频率高，而 `MajorGC` 发生的频率则较低。老年代中的对象普遍比较稳定，通常会长期存在，所以变化不是特别频繁。`MajorGC` 采用的是**标记-压缩算法**，也就是上面提到的基本垃圾回收机制。

实验任务

本次实验我们将实现一个简单的 JVM 分代垃圾回收机制，也会设置 `eden` 区，`survivor` 区和 `tenured` 区，模拟垃圾回收的工作流程。

仓库中会给出 gcsimulation 文件夹，有如下几个类：

- `MyObject`
  - 模拟创建的对象
- `MyHeap`
  - 普通小顶堆
- `JvmHeap`
  - JVM 中的堆，`eden`、`survivor`、`tenured` 均使用堆来实现，继承自`MyHeap`。
- `MyJvm`
  - 模拟的 JVM，负责管理堆、创建对象、删除对象引用和垃圾回收
- `Main`
  - 模拟程序的输入输出，输入方式为先输入指令名称，换行后再输入参数。
  - 输入有以下几条指令：
    - `CreateObject`：创建新的对象，换行后输入创建对象的个数
    - `SetUnreferenced`：将对象设置为未引用，换行后输入删除引用的对象id，用空格分隔
    - `RemoveUnreferenced`：直接在堆中移除未引用的对象
    - `MinorGC`：小型垃圾回收
    - `MajorGC`：大型垃圾回收
    - `SnapShot`：查看当前 JVM 中堆的快照

注意：虽然在JDK中，所谓的“堆”实际上是一段内存；但在本次实验中，我们使用了小顶堆来实现这一“堆”的概念。

任务清单:

任务分为两大部分，第一部分为 JML 补全，第二部分为代码补全

第一部分：

1. `JvmHeap` 类：按要求补全 [1] 中规格
2. `MyHeap` 类：按要求补全 [2] 中规格
   - Hint：
     - 小顶堆是一棵完全二叉树，父子节点在数组中的下标满足一定的关系，其中零下标为无用下标

       e.g.: 数组 [1, 3, 7, 13, 21, 31] 对应于如下小顶堆：

第二部分：

1. `JvmHeap` 类：按规格实现 `removeUnreferenced` 方法
2. `JvmHeap` 类：按规格实现 `getYoungestOne` 方法

3. `MyJvm` 类：补全 `minorGC` 方法
4. `MyObject` 类：按规格实现 `compareTo` 方法

提交事项：

1. 需要填空的地方在程序中已用 [1]，[2] 等序号和 **// TODO**标注。
2. 对于 JML 中的临时变量的使用，按照 `i`，`j`，`k` 的次序依次使用。
3. 同上一次实验，第一部分任务的答案放在 answer.json 文件中提交，**并同时**需要将作答内容全部写在内容提交区。提交示例如下：

```
{
    "1": "xxxxxx",
    "2": "xxxxxx"
}
```

   注：各答案的结尾无需带分号

4. 对于第二部分任务，直接在官方文件夹 gcsimulation 中作更改，并连同第一部分的 answer.json 文件提交。提交目录应包括子目录 gcsimulation 及 answer.json 文件，即 answer.json 文件不放在 gcsimulation 子目录中。因此，你的仓库布局应当如下所示：

```
homework_2022_你的学号_exp_6 // 仓库目录
| - answer.json              // 第一部分的答案
| - gcsimulation             // 第二部分的答案
| - | - JvmHeap.java
| - | - Main.java
| - | - MyHeap.java
| - | - MyJvm.java
| - | - MyObject.java
```

注意：本次实验要求编译成功

无需关注的部分：

以下部分均由课程组封装好，实验过程中无需关注：

1. Main.java 文件
2. `MyHeap` 类 `add`、`removeFirst` 方法的具体实现，但要清楚成员变量 `size` 的变化时机
3. `MyJvm` 类的 `getSnapShot` 方法

输入输出样例：

输入

```
CreateObject
5
SetUnreferenced
1 4
SnapShot
RemoveUnreferenced
SnapShot
CreateObject
10
SetUnreferenced
2 6 10 15
CreateObject
5
```

```
SnapShot
```

输出

```
Start JVM Garbage Collection Simulation.
Create 5 Objects.
Set id: 1 Unreferenced Object.
Set id: 4 Unreferenced Object.
Eden: 5
0 1 2 3 4
Survive 0: 0

Survive 1: 0

Tenured: 0

---------------------------------
Remove Unreferenced Object.
Eden: 3
0 2 3
Survive 0: 0

Survive 1: 0

Tenured: 0

---------------------------------
Create 10 Objects.
Set id: 2 Unreferenced Object.
Set id: 6 Unreferenced Object.
Set id: 10 Unreferenced Object.
Set id: 15 Unreferenced Object.
Eden reaches its capacity,triggered Minor Garbage Collection.
Create 5 Objects.
Eden: 2
18 19
Survive 0: 13
0 3 5 7 8 9 11 12 13 14 15 16 17 , the youngest one 0's age is 1
Survive 1: 0

Tenured: 0

---------------------------------
End of JVM Garbage Collection Simulation.
```

关于实验测试数据：

1. 不需要同学们考虑异常的情况
2. 不需要同学们考虑数据很大的情况
3. 不需要同学们考虑性能相关实现

参考链接

1. 图解 Java 垃圾回收机制

提示

本题需要同学们在短时间内完成Java垃圾回收机制的阅读理解、JML规格与Java代码的阅读以及对应题目的填写。

Java垃圾回收机制是整个代码的关键；请确保充分理解该机制之后再进行填空。

关于JML与代码，如果感觉时间吃紧，可以试着从方法名上猜测该方法的具体行为，再与JML对照来加快代码阅读速度。此外，在填写代码时可以考虑如何使用代码中已经提供的方法。

此外，如果有一定的空余时间，建议测试一下你所补完的程序，在熟悉Java垃圾回收机制的同时检测代码中的错误。