**Lab 2 setup: Setting Up IntelliJ**

主要就是Intellij IDEA 的配置

**在新版 IntelliJ IDEA 中添加 JDK ：**

添加 JDK 到 IntelliJ 项目中

1.打开 IntelliJ IDEA：

启动 IntelliJ IDEA 并打开你的项目。

2.打开项目设置：

3.如果你的项目已经打开，可以直接点击顶部菜单栏的 File（文件） -&gt; Project Structure…（项目结构…）

4.如果你的项目尚未打开，可以在欢迎窗口选择你的项目，然后点击 Configure（配置） -&gt; Project Structure…（项目结构…）

5.选择 JDK 设置：

在弹出的项目结构窗口中，选择 Project（项目）面板，然后在右侧 Project SDK 下拉菜单中选择 New…。

6.选择 JDK 类型：

在弹出的窗口中，选择 JDK。通常情况下，如果你已经在系统中安装了 JDK，IntelliJ 会自动检测到可用的 JDK。如果没有自动检测到，可以手动浏览找到 JDK 的安装目录。

7.确认 JDK 选择：

选择正确的 JDK 版本（例如 JDK 11、JDK 8 等），然后点击 OK。

8.应用设置：

在项目结构窗口中，确保 Project SDK 下拉菜单显示选择的 JDK 版本。然后点击 Apply（应用）或 OK（确定）保存设置并关闭窗口。

9.确认设置：

确保项目结构窗口已关闭，并且项目设置已成功应用。现在你的项目应该已经配置了正确的 JDK。

***（设置默认 JDK***

*如果你想设置默认的 JDK（即新建项目时默认使用的 JDK 版本），可以在 IntelliJ IDEA 的全局设置中配置：*

*10.打开全局设置：*

*点击顶部菜单栏的 File（文件）-&gt; Settings（设置）。*

*11.选择 JDK 设置：*

*在弹出的设置窗口中，选择 Build, Execution, Deployment（构建、执行、部署）-&gt; Java Compiler（Java 编译器）。*

*12.配置默认 JDK：*

*在右侧 Project bytecode version（项目字节码版本）下拉菜单旁边的 Project SDK 选项中选择默认的 JDK 版本。*

*13.应用设置：*

*点击 Apply（应用）或 OK（确定）保存设置。）*

我的JDK地址： C:\Users\njsal\.jdks\openjdk-22.0.1

Getting Java Libraries

git submodule update --init

用于初始化并更新 Git 仓库中的子模块

介于我们导入的文件来自github，我们需要稳定的开启代理服务器。  
**Importing Libraries and Running Code**The trouble is that we haven’t told IntelliJ where to find the CS61B libraries we just pulled.  
1、Click **File -> Project Structure** in the top left of IntelliJ. A window should pop up, and you should click **Libraries** on the left panel of this window.

2、Click on the javalib folder (which is in library-sp18) and press OK. If you get a message that says something like **“IDEA cannot determine what kind of files the chosen items contain”** then you probably selected the **wrong folder.** Make sure you’ve selected the “javalib” folder and not the “library-sp18” folder.

那个加号点过后，要选择Java，再选路径。  
**Embedded Terminal (Optional)**

Type in “terminal” in the search bar. Once there, type in "C:\Program Files (x86)\Git\bin\sh.exe" --login -i into the Shell Path field. Click ok.

**Lab 2: Unit Testing with JUnit and IntLists**

左键想要运行的程序，即可debug

点击行号可以设置断点

which means we have now set a breakpoint. If we run the program in debug mode again it’ll stop at that line.

再debug即可。  
We can advance the program one step by clicking on the “step into” button, which is an arrow that points down and to the left as shown on the next line: step_into

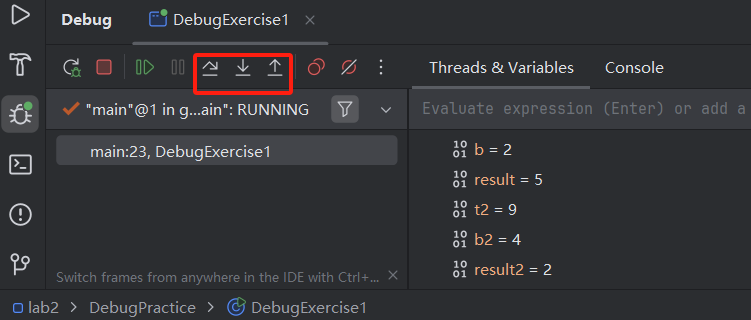
1、进入调试模式后，你可以使用以下快捷键来逐步执行代码：

F7：Step into（步入）。这个快捷键可以让程序在遇到方法调用时进入该方法内部执行。

F8：Step over（步过）。这个快捷键可以让程序执行当前行，不进入当前行的方法内部，直接到达下一行。

Shift + F8：Step out（步出）。这个快捷键可以让程序执行完当前方法内的所有代码，直到方法返回到调用处。

\*其实我们新版的图标在：



右移 (>>)：

右移操作将一个数的二进制表示向右移动指定的位数。移出右边界的位会被丢弃。**对于无符号数，左边用0填充；对于有符号数，根据符号位来填充左边。**

例如，假设 b = -7，其二进制表示为 1111 1001（假设是一个8位有符号整数）。如果我们执行 b >> 2，即右移两位：

b 的二进制 1111 1001 右移两位变成 1111 1110。

结果是 1111 1110，对应十进制为 -2。

位与：& 位或：| 位反：~ 异或：^

取a, b最大值：

public static int max(int a, int b) {  
 int w = (b - a) >> 31;//常见标记形式，b>=a, w=0;b<a，w=-1,二进制，全1  
 int z = ~(b - a) >> 31; //反过来  
 int max = a & w | b & z; //&w就是0，&z就是本身  
 return max;  
}

两个数求和：

public static int add(int a, int b) {  
 int x = a, y = b;  
 int xor, and, temp;  
 and = x & y; //需要进位的  
 xor = x ^ y; //不用进位的  
 while (and != 0) {  
 and <<= 1; //左移1位，模拟进位  
 temp = xor ^ and;//temp代表现在不用进位的，之后赋值给xor  
 and &= xor; //原xor和新的and来看有没有新的进位，!0则继续加  
 xor = temp;  
 }  
 return xor; //这个不用进位的就是最终的和  
}

这个仍然处理不了大整数，具体写法到原来学C的代码里找（字符串、进位）

An even faster approach is to make our breakpoint conditional. To do this, **right (or two-finger) click on the red breakpoint dot**. Here, you can set a condition for when you want to stop. In the condition box, enter “newTotal < 0”, stop your program, and try clicking “debug” again. You’ll see that you land right where you want to be.

int newTotal = totalTurnips + numAvailable;

totalTurnips = newTotal;

这种逻辑在多次累加时可能会导致整数溢出，从而使 totalTurnips 变成负数。  
为了避免整数溢出，可以采取以下改进：  
**使用长整型 (long)**：将 totalTurnips 和 newTotal 的类型改为 long，这样可以支持更大范围的整数值。  
**检查溢出条件**：在计算 newTotal 之前，可以添加条件检查，确保不会导致整数溢出。

public class DebugExercise3 {  
 public static long countTurnips(In in) {  
 long totalTurnips = 0;  
 while (!in.isEmpty()) {  
 String vendor = in.readString();  
 String foodType = in.readString();  
 double cost = in.readDouble();  
 int numAvailable = in.readInt();  
 if (foodType.equals("turnip")) {  
 // Check if adding numAvailable will not cause overflow  
 if (numAvailable > 0 && Long.*MAX\_VALUE* - totalTurnips < numAvailable) {  
 // Handle overflow scenario here, if needed  
 // For now, assume it doesn't occur in realistic input  
 System.*err*.println("Warning: Integer overflow may occur. Adjusting calculation.");  
 // Handle overflow by skipping this addition or logging an error  
 continue; // Skip this iteration or handle as per your requirement  
 }  
 for (int i = 0; i < numAvailable; i++) {  
 totalTurnips++;  
 }  
 }  
 in.readLine();  
 }  
 return totalTurnips;  
 }  
 public static void main(String[] args) {  
 In in = new In("foods.csv");  
 System.*out*.println(*countTurnips*(in));  
 }  
}

溢出检查和处理：在 if 语句中，首先检查 numAvailable 是否为正数，并且检查 Long.MAX\_VALUE - totalTurnips 是否小于 numAvailable，这可以避免将 totalTurnips 累加到一个超过 Long 类型最大值的不安全的范围内。如果发现可能会溢出，代码中给出了一个简单的警告输出，并通过 continue 语句跳过了这次迭代，这样可以在溢出情况下避免对 totalTurnips 的不安全操作。**这也意味着，有些值没加进去！！**

**Application: IntLists**

Destructive:

public static void dSquareList(IntList L) {

while (L != null) {

L.first = L.first \* L.first;

L = L.rest;

}

}

IntList origL = Intlist.list(1, 2, 3)

dSquareList(origL);

// origL is now (1, 4, 9)

Non-destructive:

迭代(iterative)：

public static IntList squareListIterative(IntList L) {  
 if (L == null) {  
 return null;  
 }  
 IntList res = new IntList(L.first \* L.first, null);  
 IntList ptr = res;  
 L = L.rest;  
 while (L != null) {  
 ptr.rest = new IntList(L.first \* L.first, null);  
 L = L.rest;  
 ptr = ptr.rest;  
 }  
 return res;  
}  
//递归（recursive）：  
*/\*\*  
 \* Returns a list equal to L with all elements squared. Non-destructive.  
 \*/*public static IntList squareListRecursive(IntList L) {  
 if (L == null) {  
 return null;  
 }  
 return new IntList(L.first \* L.first, *squareListRecursive*(L.rest));  
}

创建：

1、简易

IntList myList = IntList.of(0, 1, 2, 3);

// Creates the IntList 0 -> 1 -> 2 -> 3 -> null

2、正规

*/\*First element of list.\*/*public int first;  
*/\*Remaining elements of list.\*/*public IntList rest;  
*/\*A List with first FIRST0 and rest REST0.\*/*public IntList(int first0, IntList rest0) {  
 first = first0;  
 rest = rest0;  
}  
*/\*A List with null rest, and first = 0.\*/*public IntList() {  
/\* NOTE: public IntList () { } would also work. \*/  
 this(0, null);  
}

### Implementing Destructive vs. Non-destructive Methods

 catenate(IntList A, IntList B) and dcatenate(IntList A, IntList B) both result in an IntList which contains the elements of A followed by the elements of B.The only difference between these two methods is that dcatenate modifies the original IntList A (i.e. it’s destructive) and catenate does not.

public static IntList dcatenate(IntList A, IntList B) {  
 //*TODO: fill in method* if (A == null){  
 return B;  
 }  
 else {  
 IntList ptr = A;  
 while (ptr.rest != null){ //指针  
 ptr = ptr.rest;  
 }  
 ptr.rest = B;  
 }  
 return A;  
}

public static IntList catenate(IntList A, IntList B) {  
 //*TODO: fill in method* if (A ==null){  
 return B;  
 }  
 IntList res = new IntList(A.first, null);  
 IntList ptr = res;  
 A = A.rest;  
 while (A != null) {  
 ptr.rest = new IntList(A.first, null);  
 A = A.rest;  
 ptr = ptr.rest;  
 }  
 ptr.rest = B;  
 return res;  
}

上面都是迭代的方法，现在用递归看一下：

public static IntList dcatenate(IntList A, IntList B) {

if (A == null) {

return B; // 如果A为空，直接返回B链表

} else {

// 递归调用，找到A链表的末尾，并将B链表接在末尾

A.rest = dcatenate(A.rest, B);

return A;

}

}

public static IntList catenate(IntList A, IntList B) {

if (A == null) {

return B; // 如果A为空，直接返回B链表

} else {

// 将A链表的第一个节点和剩余节点连接到结果链表中

return new IntList(A.first, catenate(A.rest, B));

}

}

**Project 0: NBody Simulation**

Your goal for this project is to write a program simulating the motion of N objects in a plane, accounting for the gravitational forces mutually affecting each object as demonstrated by Sir Isaac Newton’s [Law of Universal Gravitation](http://en.wikipedia.org/wiki/Newton%27s_law_of_universal_gravitation).  
Ultimately, you will be creating a program NBody.java that draws an animation of bodies floating around in space tugging on each other with the power of gravity.

暑期不考虑project的完成。

**2.1~2.5 Notes**

int x;

double y;

We'd end up with boxes of size 32 and 64 respectively, as shown in the figure below:



8 primitive types: byte, short, int, long, float, double, boolean, char

**Exercise 2.1.1**: Suppose we have the code below:

public class PassByValueFigure {

public static void main(String[] args) {

Walrus walrus = new Walrus(3500, 10.5);

int x = 9;

doStuff(walrus, x);

System.out.println(walrus);

System.out.println(x);

}

public static void doStuff(Walrus W, int x) {

W.weight = W.weight - 100;

x = x - 5;

}

}

Does the call to doStuff have an effect on walrus and/or x?

1. **对象和引用（Walrus）：**在Java中，对象是按引用传递的，但是引用本身是按值传递的。这意味着当你将一个对象（例如 Walrus）传递给一个方法时，实际上传递的是对该对象的引用。对对象状态的更改（如修改 Walrus 的 weight）会影响原始对象。
2. **基本类型变量（int x）：**基本类型变量（例如 int）是按值传递的。这意味着方法中会传递该变量值的副本。在 doStuff 方法中对参数 x 的修改不会影响 main 方法中的原始 x 变量。

在调用 doStuff(walrus, x); 后：walrus 的 weight 减少了100个单位。main 方法中的 x 保持为9，因为 doStuff 内部对 x 的修改是局部的，不会影响 main 中的原始 x 变量。

The most reasonable base case is that rest is null, which results in a size 1 list.

/\*\* Return the size of the list using... recursion! \*/

public int size() {

if (rest == null) {

return 1;

}

return 1 + this.rest.size();

}

/\*\* Return the size of the list using no recursion! \*/

public int iterativeSize() {

IntList p = this;

int totalSize = 0;

while (p != null) {

totalSize += 1;

p = p.rest;

}

return totalSize;

}

**SLLists 单链表**

单链表是一种基本的数据结构，它由一个节点的序列组成，每个节点包含数据和一个指向下一个节点的引用（或指针）。这种数据结构非常适合用于需要频繁插入和删除操作的情况，因为在单链表中，每个节点只存储对下一个节点的引用，而不需要像数组那样预先分配固定大小的内存空间。

单链表的基本操作包括：

插入（Insertion）：在链表的任意位置插入一个新节点。

删除（Deletion）：从链表中删除任意位置的节点。

访问（Access）：通过遍历链表来访问特定位置的节点或者遍历整个链表。

搜索（Search）：查找链表中是否存在某个特定的元素。

创建（每一个链表元素）：

public class IntNode {

public int item;

public IntNode next;

public IntNode(int i, IntNode n) {

item = i;

next = n;

}

}

接下来是链表创建：

public class SLList {

//前面写的

public/private IntNode first;

public SLList(int x) {

first = new IntNode(x, null);

}

/\*\* Adds an item to the front of the list. \*/

public void addFirst(int x) {

first = new IntNode(x, first);

}

}

在Java中，使用 private 关键字可以限制类中的变量或方法只能被同一个类内部的代码访问，其他类无法直接访问这些私有成员。例如，当我们将单链表（SLList）类中的 first 变量声明为 private 时：

public class SLList {

private IntNode first;

// 其他代码

}

这意味着只有在同一个 .java 文件中的 SLList 类才能直接访问 first 变量。这种访问权限的设定有助于控制和保护类的内部状态，防止外部类或程序意外地修改关键的类成员变量，从而提高了代码的安全性和可靠性。

例如，如果有另一个类 SLLTroubleMaker 试图访问 SLList 类中的 first 变量，如下所示：

public class SLLTroubleMaker {

public static void main(String[] args) {

SLList list = new SLList(); //这里是新建单链表

list.first = new IntNode(10); // 这里会产生编译错误

}

}

在这个例子中，当我们尝试访问 list.first 时，编译器会报错并显示类似 "first has private access in SLList" 的错误信息。这是因为 first 变量是 private 的，无法从 SLList 类外部直接访问或修改。

因此，通过使用 private 关键字，我们可以有效地控制和保护类中的成员变量和方法，只允许它们在类内部进行操作和修改，提高了代码的封装性和安全性。

We can do this by creating a special node that is always there, which we will call a **sentinel node**. The sentinel node will hold a value, which we won't care about.

A SLList with a sentinel node has at least the following invariants:

* The sentinel reference always points to a sentinel node.
* The front item (if it exists), is always at sentinel.next.item.
* The size variable is always the total number of items that have been added.

addLast and getLast will be fast, but removeLast will be slow. That's because we have no easy way to get the second-to-last node, to update the last pointer, after removing the last node.

**完整版（基础完整+基础函数）：**

public class SLList {

    public class IntNode {

        public int item;

        public IntNode next;

        public IntNode(int i, IntNode n){

            item = i;

            next = n;

        }

    }

    public IntNode first;

    public SLList(int x){

        first = new IntNode(x, null);

    }

    // add a node at the front

    public void addFirst(int x){

        first = new IntNode(x, first);

    }

    // ptint the SLList

    public void printList(){

        IntNode current = this.first; //a pointer

        while (current != null){

            System.out.print(current.item);

            if (current.next != null){

                System.out.print("->");

            }

            current = current.next;

        }

        System.out.println();

    }

    // add an item to the end of the list

    /\* It's ok to create a special node that is always there, which we will call a sentinel node\*/

    public void addLast(int x){

        if (first == null){

            first = new IntNode(x, null);

            return; // end this method

        }

        IntNode current = first;

        while (current.next != null){

            current = current.next;

        }

        current.next = new IntNode(x, null);

    }

    // renturn the size of the SLList(recursive)

    public static int sizeOf (IntNode first){

        if (first == null){

            return 0;

        }

        return 1 + sizeOf(first.next);

    }

    public int size(){

        return sizeOf(first);

    }

    // iterative

    public int sizeIterative() {

        int count = 0;

        IntNode current = first;

        while (current != null) {

            count++;

            current = current.next;

        }

        return count;

    }

    public static void main(String[] args){

        SLList A = new SLList(15); // create a new SLList

        A.addFirst(10);

        A.addFirst(5);

        A.addLast(20);

        A.printList(); // 5->10->15->20

        int x = A.size();

        System.out.println(x); //4

        int y = A.sizeIterative();

        System.out.println(y); //4

    }

}

**DLLists 双向链表**

对比单链表，One fix is to add a second sentinel node to the back of the list.

A generic DLList that can hold any type would look as below:

public class DLList<BleepBlorp> {

private IntNode sentinel;

private int size;

.. . . ..

Here, BleepBlorp is just a name I made up, and you could use most any other name you might care to use instead, like GloopGlop, Horse, TelbudorphMulticulus or whatever.

Now that we've defined a generic version of the DLList class, we must also use a special syntax to instantiate this class. To do so, we put the desired type inside of angle brackets during declaration, and also use empty angle brackets during instantiation. For example:

DLList<String> d2 = new DLList<>("hello");

d2.addLast("world");

Since generics only work with reference types, we cannot put primitives like int or double inside of angle brackets, e.g. <int>. Instead, we use the reference version of the primitive type, which in the case of int case is Integer, e.g.

DLList<Integer> d1 = new DLList<>(5);

d1.insertFront(10);

If you need to instantiate a generic over a primitive type, use Integer, Double, Character, Boolean, Long, Short, Byte, or Float instead of their primitive equivalents.

完整版：

public class DLList{

    public class IntNode{

        public int item;

        public IntNode prev;

        public IntNode next;

        public IntNode (int i){

            this.item = i;

            this.prev = null;

            this.next = null;

        }

    }

    IntNode head;

    IntNode tail;

    public DLList(){

        this.head = null;

        this.tail = null;

    }

    public void addFirst(int i){

        IntNode newNode = new IntNode (i);

        if (head == null){

            head = newNode;

            tail = newNode;

        }

        else {

            newNode.next = head;

            head.prev = newNode;

            head = newNode;

        }

    }

    public void addLast(int i){

        IntNode newNode = new IntNode (i);

        if (tail == null){

            head = newNode;

            tail = newNode;

        }

        else {

            newNode.prev = tail;

            tail.next = newNode;

            tail = newNode;

        }

    }

    public void printForward(){

        IntNode current = head;

        while(current != null){

            System.out.print(current.item);

            if(current.next != null){

                System.out.print("->");

            }

            current = current.next;

        }

        System.out.println();

    }

    public void printBackward(){

        IntNode current = tail;

        while(current != null){

            System.out.print(current.item);

            if(current.prev != null){

                System.out.print("->");

            }

            current = current.prev;

        }

        System.out.println();

    }

    public void removeAt(int pos){

        if (head == null) {

            System.out.println("链表为空，无法删除元素。");

            return;

        }

        if (pos < 1) {

            System.out.println("无效的位置。");

            return;

        }

        IntNode current = head;

        int count = 1;

        while (current != null && count < pos){

            current = current.next;

            count = count + 1;

        }

        if (current == null) {

            System.out.println("位置超出链表长度。");

        }

        if (current == head && current == tail) {

            // Case 1: Single node in the list

            head = null;

            tail = null;

        }

        else if (current == head) {

            // Case 2: Deleting the head node

            head = current.next;

            head.prev = null;

        }

        else if (current == tail) {

            // Case 3: Deleting the tail node

            tail = current.prev;

            tail.next = null;

        }

        else {

            // Case 4: Deleting a node in the middle

            current.prev.next = current.next;

            current.next.prev = current.prev;

        }

        System.out.println("成功删除位置 " + pos + " 的元素。");

    }

    public static void main(String[] args) {

        DLList L = new DLList();

        L.addFirst(15);

        L.addFirst(10);

        L.addFirst(5);

        L.addLast(20);

        System.out.print("正向打印链表： ");

        L.printForward();

        System.out.print("反向打印链表： ");

        L.printBackward();

        L.removeAt(3);

        System.out.println("删除元素后链表：");

        L.printForward();

    }

}

**Arrays**

Creation:  
x = new int[3];  
fill each of the 3 boxes with the default int value 0  
y = new int[]{1, 2, 3, 4, 5};  
int[] z = {9, 10, 11, 12, 13};

System.arraycopy(b, 0, x, 3, 2); 这行代码展示了一种将一个数组的信息复制到另一个数组的方法。System.arraycopy 接受五个参数：

1. 源数组：要复制数据的数组。
2. 源数组起始位置：从源数组的哪个位置开始复制数据。
3. 目标数组：要将数据复制到的数组。
4. 目标数组起始位置：从目标数组的哪个位置开始粘贴数据。
5. 要复制的项数：要复制的元素个数。

就是说，b[0],b[1]复制到a[3],a[4]  
**2D Arrays in Java**

易错：

int[][] pascalsTriangle;

pascalsTriangle = new int[4][];

int[] rowZero = pascalsTriangle[0];//就是null

pascalsTriangle[0] = new int[]{1};

实例：

int[][] x = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};

int[][] z = new int[3][];

z[0] = x[0]; //引用

z[1] = x[1];

z[2] = x[2];

z[0][0] = -z[0][0];

int[][] w = new int[3][3];

System.arraycopy(x[0], 0, w[0], 0, 3); //副本

System.arraycopy(x[1], 0, w[1], 0, 3);

System.arraycopy(x[2], 0, w[2], 0, 3);

w[0][0] = -w[0][0];

结果：x[0][0]=-1,z[0][0]=-1,w[0][0]=1

在这里，数组 z 被初始化为一个包含三个元素的二维数组。z[0], z[1], z[2] 分别引用了 x[0], x[1], x[2] 对应的一维数组。此时，z[0] = {1, 2, 3}, z[1] = {4, 5, 6}, z[2] = {7, 8, 9}。z[0][0] = -z[0][0];这行代码将 z[0][0] 的值取反。因为 z[0] = x[0] = {1, 2, 3}，所以 z[0][0] 现在变为 -1。因此，此时 z[0] = {-1, 2, 3}。同时，由于 z[0] 和 x[0] 实际上引用了同一个数组对象，所以 x[0] 也被修改为 { -1, 2, 3 }。

System.arraycopy 在复制数组时是浅拷贝，即复制的是数组的引用，而不是数组的内容。因此，对 w 中的子数组的修改不会影响到 x 中相同位置的子数组。

Array boxes are numbered and accessed using [] notation, and class boxes are named and accessed using dot notation.  
Array boxes must all be the same type. Class boxes can be different types.

**AList 动态数组**

**泛型支持**: ArrayList 支持泛型，因此可以存储特定类型的元素。例如，ArrayList<String> 表示只能存储字符串类型的元素。

**随机访问**: 可以通过索引快速访问 ArrayList 中的元素，这是由于其内部使用数组来存储数据。

增加容量：

public void insertBack(int x) {

if (size == items.length) {

resize(size \* RFACTOR);

}

items[size] = x;

size += 1;

}

泛型AList:\*我们需要创建一个 Object 类型的数组，并进行类型转换：

Glorp[] items = (Glorp[]) new Object[8];

**完整版：**

/\* Invariants:

 addLast: The next item we want to add, will go into position size

 getLast: The item we want to return is in position size - 1

 size: The number of items in the list should be size.

\*/

public class AList{

    public int items[];

    public int size;

     /\*\* Creates an empty list. \*/

     public AList() {

        items = new int[100];

        size = 0;

    }

    /\*\* Inserts X into the back of the list. \*/

    public void addLast(int x) {

        items[size] = x;

        size = size + 1;

    }

    /\*\* Returns the item from the back of the list. \*/

    public int getLast() {

        return items[size - 1];

    }

    /\*\* Gets the ith item in the list (0 is the front). \*/

    public int get(int i) {

        return items[i - 1];

    }

    /\*\* Returns the number of items in the list. \*/

    public int size() {

        return size;

    }

    /\*\* Deletes item from back of the list and

      \* returns deleted item. \*/

    public int removeLast() {

        int x = getLast();

        size = size - 1;

        return x;

    }

    public void addItem(int i){

        int[] a = new int[size + 1];

        System.arraycopy(items, 0, a, 0, size);

        a[size] = i;

        items = a;

        size = size + 1;

        System.out.println(i);

    }

    public static void main(String[] args){

        AList L = new AList();

        L.addLast(5);

        L.addLast(10);

        L.addLast(15);

        int x = L.getLast(); //15

        int y = L.get(2); //10

        int z = L.removeLast(); //15

        int s1 = L.size(); //2

        System.out.println(x+ " " + y + " " + z + " " + s1);

        L.addItem(20); //20

        int s2 = L.size();

        System.out.println(s2); //3

        //the array doesn't actually change size, we are just making a new one that has a bigger size

    }

}