Berkeley CS61B Data Structures, Spring 2021

https://sp21.datastructur.es/about.html

Instructor: Josh Hug (me)      [hug@cs.berkeley.edu](mailto:hug@cs.berkeley.edu)

**Please post administrative issues to Ed or send an email to** [**cs61b@berkeley.edu**](mailto:cs61b@berkeley.edu)

<https://sp21.datastructur.es/>

About the Course

## 1 Course Structure

Small minority of your learning:

* **Introduction to new material**: Lectures / reading.

The vast majority of your learning:

* **Theory**: Discussion sections, study guides, theory homework.
* **Programming, Tool Usage, Problem Decomposition:** Labs, coding HW, projects.
* **Design:** Projects 2 and 3.

## 2 Course Phases

Phase 1: Programming Intensive Introduction to Java.

* Weeks 1-4.
* One browser-based programming HW (this HW0 is optional).
* Four labs to introduce you to various tools (starting this week).
* Two projects (proj0 and proj1).
* Midterm 2/10 at time TBD.

Phase 2: Data Structures.

* Weeks 5-10.
* Incredibly important and foundational material: Expect an CS job interview to lean heavily on this part of the course.
* One programming HWs (HW1) and one exam-prep theory HW (HW2).
  + Applications and deeper insight into data structures.
* One very large solo project (Proj 2), due 4/2. Checkpoint due 3/12.
* Labs:
  + Lab 5: Peer review on project 1.
  + Two labs that implement data structures (hash table and BST).
  + Remaining labs are focused on project 2.
* Midterm 3/17 at time TBD.

Phase 3: Algorithms and Software Engineering.

* Weeks 10-14
* Project:
  + Proj 3: Build Your Own World: An open ended project where you and a partner build a 2D world with physics according to your own design. Due during lab in the last week of the class.
* Labs devoted to project.
* One theory homework due 5/3.

See calendar at <http://datastructur.es> for more.

## 3 Learn to Use the Terminal (Optional)

If you already know how to open and use a terminal, skip this section.

The terminal is an application that allows you to run all sorts of programs, as well as manipulate files in your own computer. It is a powerful but also dangerous tool, so please be careful with using some of these commands. On Unix-like operating systems, the Terminal application will provide you with everything that you need. On macOS, for example, you can use Spotlight to search for the Terminal application.

Here are some important ones that you may find useful in this course:

* cd: change your working directory
* cd hw

This command will change your directory to hw.

* pwd: present working directory
* pwd

This command will tell you the full absolute path for the current directory you are in if you are not sure where you are.

* .: means your current directory
* cd .

This command will change your directory to the current directory (aka. do nothing).

* ..: means one parent directory above your current directory
* cd ..

This command will change your directory to its parent. If you are in /workspace/day1/, the command will place you in /workspace/.

* ls: list files/folders in directory
* ls

This command will list all the files and folders in your current directory.

ls -l

This command will list all the files and folders in your current directory with timestamps and file permissions. This can help you double-check if your file updated correctly or change the read-write- execute permissions for your files.

* mkdir: make a directory
* mkdir dirname

This command will make a directory within the current directory called dirname.

* rm: remove a file
* rm file1

This command will remove file1 from the current directory. It will not work if file1 does not exist.

rm -r dir1

This command will remove the dir1 directory recursively. In other words, it will delete all the files and directories in dir1 in addition to dir1 itself. Be careful with this command!

* cp: copy a file
* cp lab1/original lab2/duplicate

This command will copy the original file in the lab1 directory and and create a duplicate file in the lab2 directory.

* mv: move or rename a file
* mv lab1/original lab2/original

This command moves original from lab1 to lab2. Unlike cp, mv does not leave original in the lab1 directory.

mv lab1/original lab1/newname

This command does not move the file but rather renames it from original to newname.

There are some other useful tricks when navigating on a command line:

* Your shell can complete file names and directory names for you with *tab completion*. When you have an incomplete name (for something that already exists), try pressing the tab key for autocomplete or a list of possible names.
* If you want to retype the same instruction used recently, press the up key on your keyboard until you see the correct instruction. This saves typing time if you are doing repetitive instructions.

Defining and Using Classes

## （1）Managing Complexity

a good foundational computer science course should primarily teach you to properly manage complexity.

* This philosophy drives nearly all aspects of this 61B’s design.

helper methods.

* Using helper methods lets you formalize the decomposition of large problems into small ones.
* By focusing mental effort on a single task, there’s less room to make mistakes.

Project 0 2048Game

完成了前三个测试后。开始跟着Hug的video写public boolean tilt(Side side)的探索代码时，需要把电脑系统语言设置成英文，否则无论怎么按方向键，tile都无法移动。

Testing

## 1 Testing Philosophy

### （1）Correctness Tool #1: Autograder

Berkeley CS61B’s autograder is in fact based on JUnit plus some extra custom libraries.

Autograder has some benefits.

However, autograders don't exist in the real world and relying on autograders can build bad habits.

### （2）Correctness Tool #2: JUnit Tests

JUnit testing, as we have seen, unlocks a new world for you. Rather than relying on an autograder written by someone else, you write tests for each piece of your program.

We refer to each of these pieces as a unit.

This allows you to have confidence in each unit of your code - you can depend on them. This also helps decrease debugging time as you can isolate attention to one unit of code at a time (often a single method). Unit testing also forces you to clarify what each unit of code should be accomplishing.

There are some downsides to unit tests, however. First, writing thorough tests takes time. It's easy to write incomplete unit tests which give a false confidence to your code. It's also difficult to write tests for units that depend on other units (consider the addFirst method in your LinkedListDeque).

注意：在IntelliJ裏面運行和Junit相關的項目時，要自己去關聯IntelliJ自帶的的Junit library：

比如D:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2021.3.1\libjunit4.jar

***Test-Driven Development (TDD)***

TDD is a development process in which we write tests for code before writing the code itself. The steps are as follows:

1. Identify a new feature.
2. Write a unit test for that feature.
3. Run the test. It should fail.
4. Write code that passes the test. Yay!
5. Optional: refactor code to make it faster, cleaner, etc. Except now we have a reference to tests that should pass.

Test-Driven Development is not required in this class and may not be your style but unit testing in general is most definitely a good idea.

### （3）Correctness Tool #3: Integration Testing

Unit tests are great but we should also make sure these units work properly together. Integration testing verifies that components interact properly together.

JUnit can in fact be used for this.

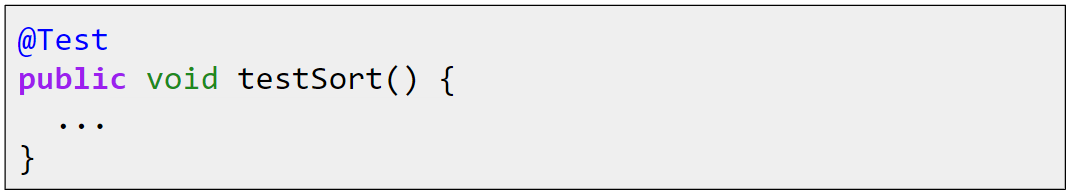
You can imagine unit testing as the most nitty gritty, with integration testing a level of abstraction above this.

The challenge with integration testing is that it is tedious to do manually yet challenging to automate. And at a high level of abstraction, it's easy to miss subtle or rare errors.

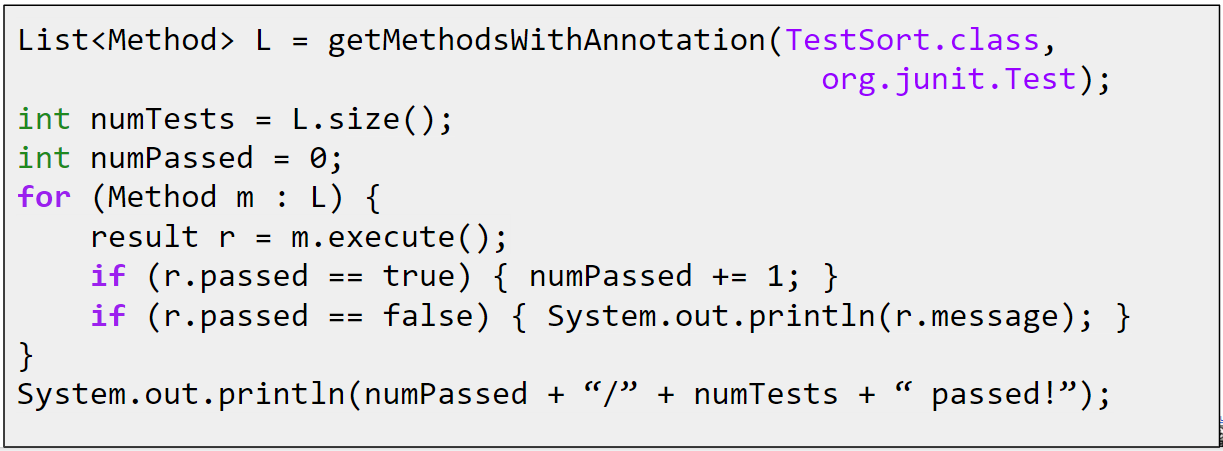
As a summary, you should **definitely write tests but only when they might be useful**! Taking inspiration from TDD, writing your tests before writing code can also be very helpful in some cases.

## 2 What is an Annotation?

Annotations (like org.junit.Test) don’t do anything on their own.



Runner uses reflections library to iterate through all methods with “Test” annotation.

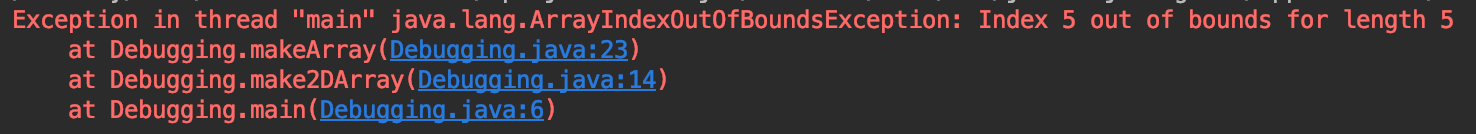


## 3 Debugging Guide

Author: Anjali Kantharuban

<https://sp21.datastructur.es/materials/guides/debugging-guide.html>

### （1）Understanding The Stack Trace



When you see an exception like this, this means your computer caught a mistake while it was running your program. To be helpful, before it exits out, your computer tells you all the information you need to know to find out where the error happened. The first line follows this format:

[Which thread errored] [What error occurred] [Any more information about the error]

What we want to focus on is the last two segments. In this case, that looks like:

Exception in thread “main” java.lang.ArrayIndexOutOfBoundException: Index 5 out of bounds for length 5

This gives you two pieces of information:

1. The error was an Array Index Out of Bounds Exception.
2. We tried to access item 5 in an array of length 5.

This tells you what happened, but not where it happened. That’s where the stack trace is useful. The three (or however many) lines under the header describe where the computer was in the code when it errored. The top trace line is what the computer was executing when it crashed and the list describes what functions called each other in reverse order. Here, we see that our main function called make2DArray on line 6. Then, make2DArray called makeArray on line 14. Finally, the code errored on line 23 within makeArray. Clicking on the blue links will jump you to that part of the code, so you don’t have to spend time scrolling. Knowing this, you can start debugging the specific sequence of calls that caused the error.

### （2）What Does This Error Mean

Perhaps you have read the stack trace, but you don’t understand what the error means. For the most part, Java errors are named such that they are understandable without prior knowledge, but in case you come across something you don’t recognize, here’s a cheat sheet:

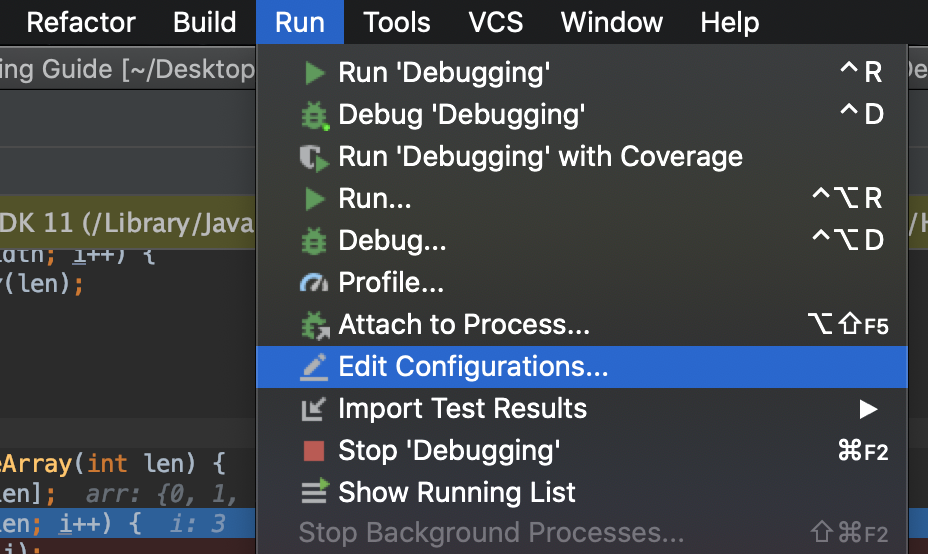
| **Error** | **What it Usually Means** |
| --- | --- |
| \_\_\_ expected | The parser can’t make sense of the line because there’s a character that it doesn’t understand or a missing character. |
| cannot find \_\_\_ | You are calling a method or class that the computer doesn’t have access to. |
| Illegal start of expression | You are missing a closing brace somewhere before this line. |
| Illegal start of type | You wrote code outside of a function body that shouldn’t be there. |
| Incompatible types – expected \_\_\_ | You are trying to assign something to a variable that is not the same type. |
| Missing method body | Your function declaration line has a semicolon. |
| Missing return statement | You should be returning something in this method but you aren’t. |
| Non-static method cannot be called from a static context | You called a method on the class itself instead of an instance of the class. |
| \*Program Freezes\* | You’re likely stuck in some sort of logical loop. |

### （3）Stepping Through With the Debugger

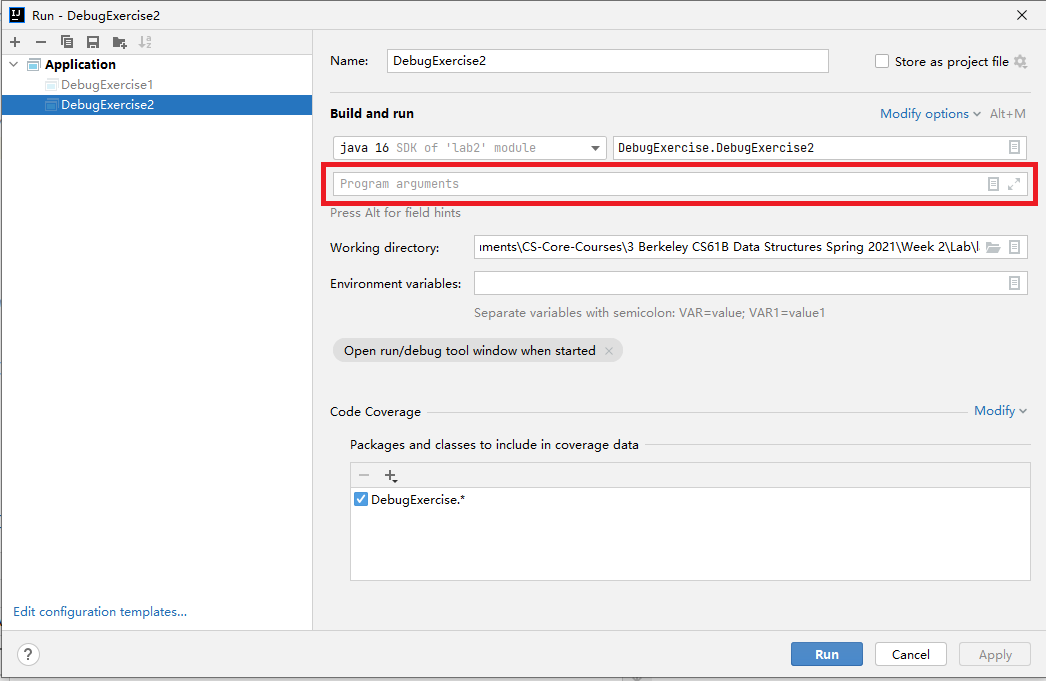
#### ① Running Code Through IntelliJ

To use the debugger, you need to run your code through IntelliJ instead of through the console.

1. From the main menu, go to Run > Edit Configurations.



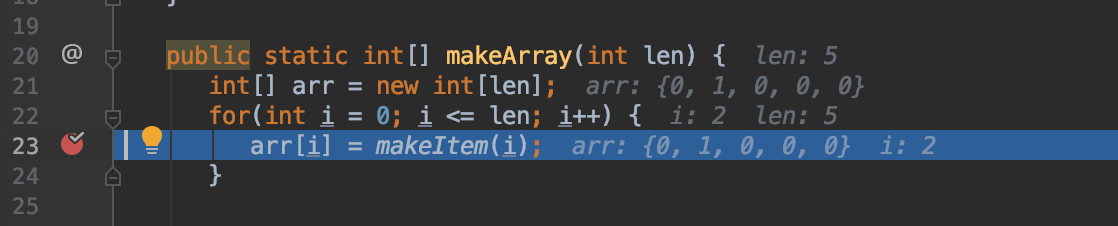
1. Enter any arguments for your program into the field marked Program Arguments. These are the arguments you would pass into the command line if you were to run it in the console.



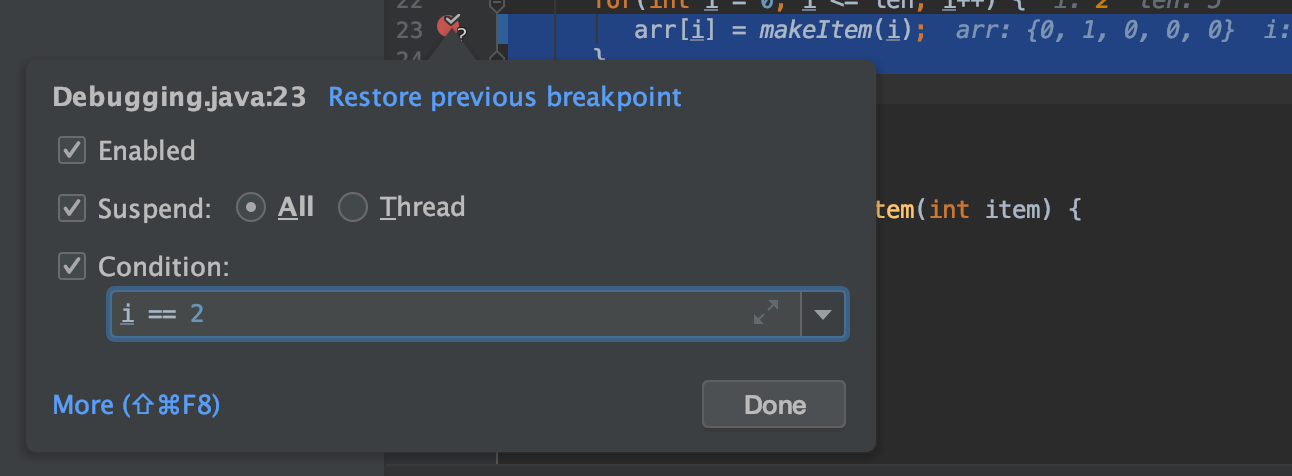
1. To run the debugger, either click Run > Debug or right click on the green arrow next to the function and select Debug

#### ② Setting Breakpoints

To examine how the code operates at runtime, we set breakpoints. Breakpoints pause your code at the line they are set so that you can see the state of all the variables around where an error occurred. To set a breakpoint, click on the space between a line number and the code:



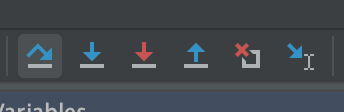
This kind of breakpoint just pauses the code the first time your computer comes across this line. If you are in a situation where the error only occurs when a variable is set to a certain value, you can set a conditional breakpoint.



To do this, set a normal breakpoint and then right click the red circle that appears. Now, you can set the breakpoint condition in the given field. Your condition can be any True/False statement that would compile at this point in the code. This means you have to use variables that already exist in the current frame, but they don’t necessarily have to be referenced in the current line.

#### ③ Stepping Through Code

##### a) Top Toolbar



To step through code, you need to understand the toolbar at the top of the debug view. To see their names, hover over the icons. From left to right, they are:

| **Button** | **What It Does** |
| --- | --- |
| Step Over | This allows you to execute the current line of code and move on to the next line in this frame. |
| Step Into | This allows you to step into any function calls in the current line given the functions are yours. |
| Force Step Into | This allows you to step into any function calls in the current line even if they are from some third party library. You should not need to do this. |
| Step Out | If you are in a function or loop, this allows you to skip the rest of the frame, essentially bringing you out to wherever this function was called. |
| Drop Frame | This allows you to reset the current frame by returning to the previous frame where it was called. This is useful if you missed the part of a function you were trying to see by essentially letting you rewind time. |
| Run to Cursor | If you are running in debug mode, you can quickly jump to areas of interest in the code by clicking run to cursor. This will act as if there is a temporary breakpoint set wherever your cursor is. |

##### b) Left Toolbar



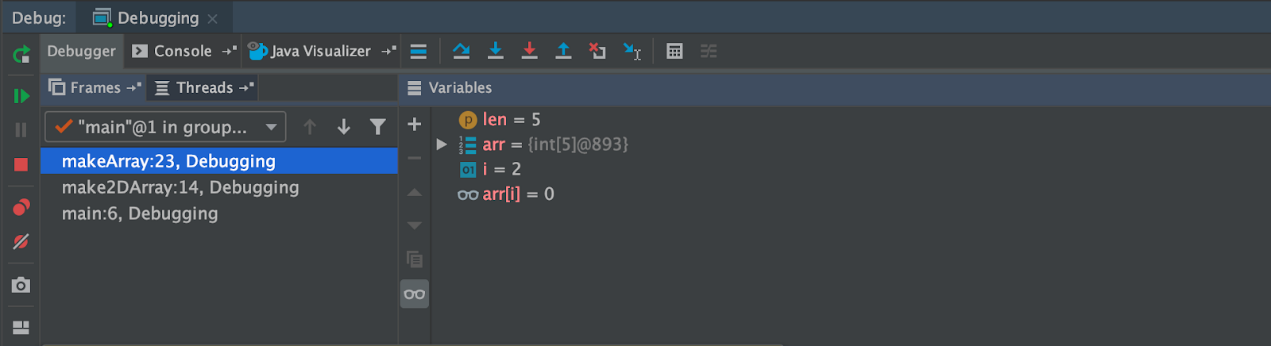
There is also a second toolbar along the left side of the debug menu. This menu is for more general controls. From top to bottom they are:

| **Button** | **What It Does** |
| --- | --- |
| Rerun | Rerun the debugger with the same settings as the current run. |
| Resume Program | Continues the program until it hits the next breakpoint. |
| Pause Program | If your program seems like it’s freezing, run it in debug mode with no breakpoints and click pause when your program freezes. It will most likely pause within whatever logical loop is causing the freeze. |
| Stop Program | If you’re done debugging, you can click this to end the program early. |
| View Breakpoints | This opens up a window that displays all your current breakpoints. Here, you can edit their settings and toggle them on/off. |
| Mute Breakpoints | Toggles all breakpoints on/off. |

#### ④ Analyzing the Current State

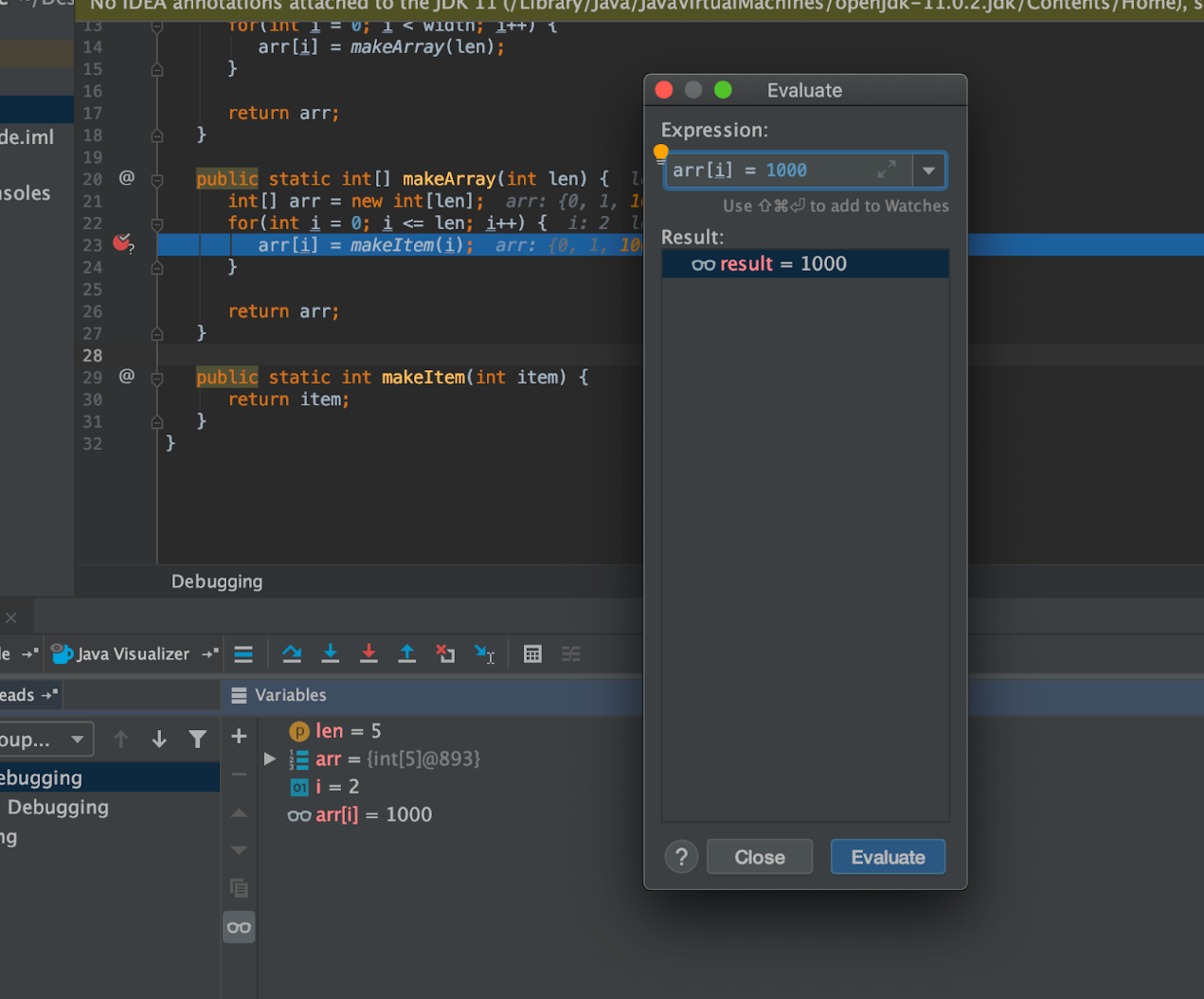
There are two places where you can get information about the state of your program while in debug mode.

##### a) Debug View

Firstly, in the debug view there are two columns:

1. The first column shows the stack trace up until this point. Each line describes a frame, in order of narrowest to widest. The name of the frame is first and refers to the name of the function called. Then, we see the line number we are currently on for that frame. In this example,  make2DArray called makeArray on line 14, so it remains on that line until makeArray finishes running.
2. The second column lists all the variables in the current frame, which **includes any global variables**. You can’t see variables that don’t exist in the current frame. Here you can see what value they hold as of the line you are on.

If you want more information than just the values of each variable, there’s a button called evaluate expression in the toolbar. Pressing this allows you to essentially insert lines on the fly to the program.



For instance, here we have inserted the expression arr[i] = 1000 and clicked evaluate, which is reflected in the Result Panel and in the Variables Panel. Essentially, it is as if we inserted the line arr[i] = 1000 before arr[i] = makeItem(i).

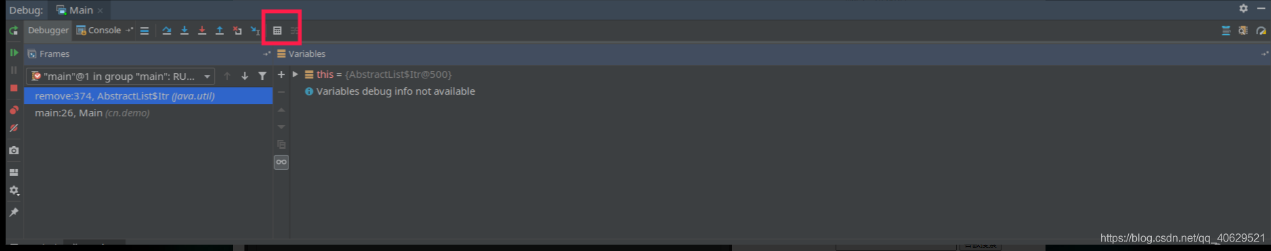
A good use of this functionality is to ensure that two objects in your code are equal (rather than being two instances of the same class) by evaluating a == b which can be difficult to tell from the Variables Panel alone.

This has the added benefit of not changing any values in the code, so you don’t have to worry about accidentally modifying the behavior you trying to observe.

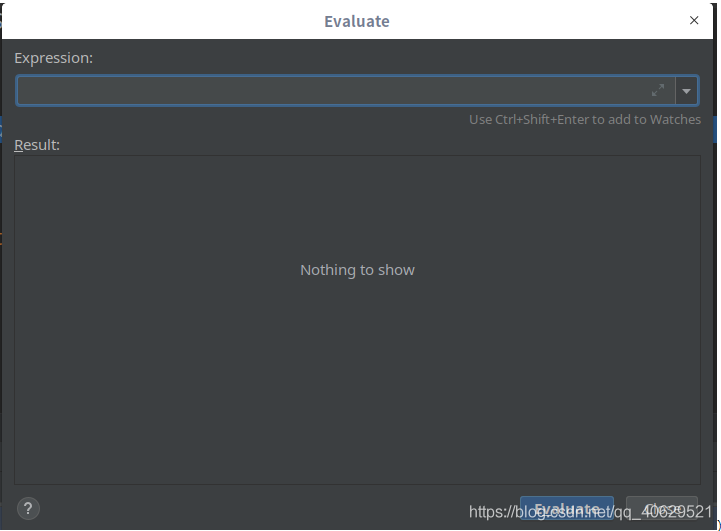
###### evaluate expression

分享一個IDEA的DEBUG神器——Evaluate Expression

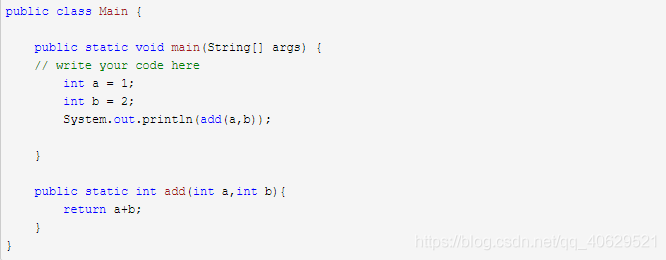
evaluate expression是idea一個在debug模式下可以動態執行代碼的工具，甚至還能動態修改該次代碼運行時變量的值。有了這工具，debug時可執行代碼中的方法，查看變量的值。



打開evaluate expression窗口，在Expression的下面input裡寫java代碼。

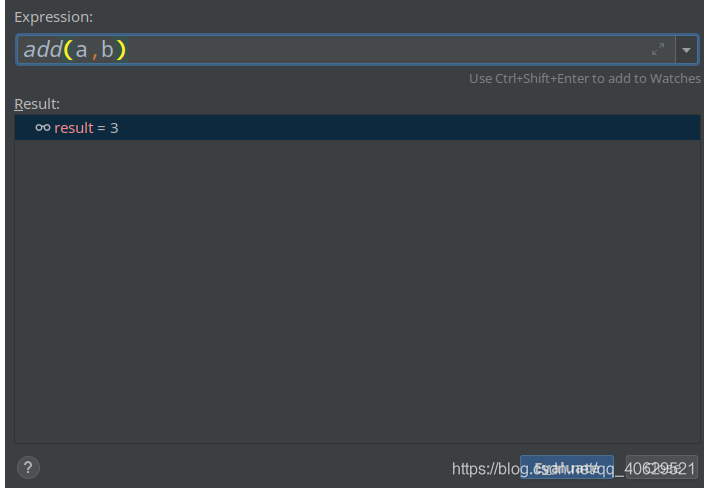


Demo

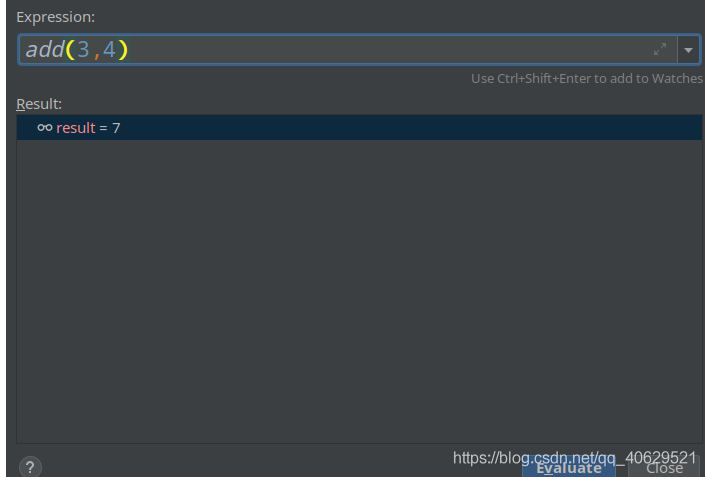


上面代碼計算a+b的值並且打印出來。實際開發中，基本不大使用Println這種控製臺打印方式。常使用log4j這類日誌組件來打印運行日誌。debug時，想知道add這個方法的返回內容。因為直接調用方法，debug時無法通過變量的方式查看結果值，使用打印日誌的方式打印出來再去看日誌文件又麻煩，此時evaluate expression有了用武之地。

輸入add(a,b)方法和參數a、b（此時根據上面的代碼a=1，b=2），點擊右下角的evaluate按鈕，可在Result中看到結果：



也可以把a, b換成其他參數，比如3和4：

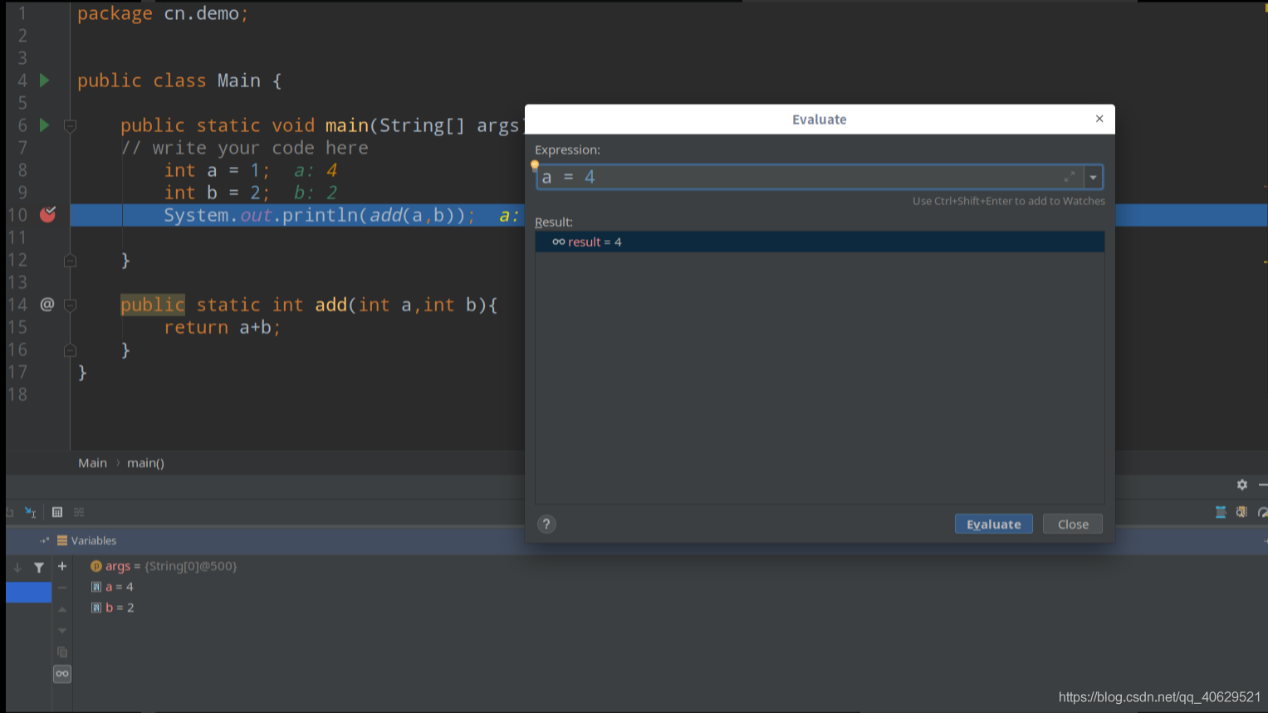


這裏填入的參數可以是當前運行方法所允許的任何變量。只要在原代碼基礎上能夠執行，這裏都可以得到結果。

原代碼運行後會在控製臺打印出3。

現在第10行打個斷點，打開evaluate expression窗口，然後把a的值改成4。

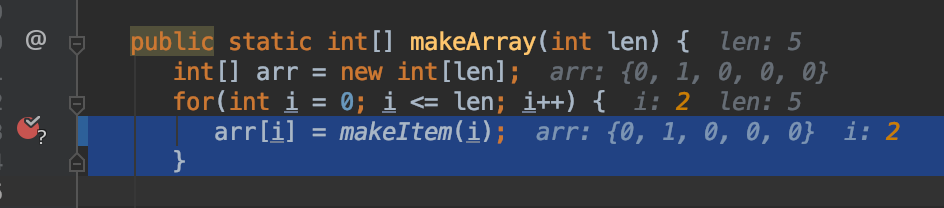
接下來點擊“step over”或者“step into”按鈕，把代碼執行到最後，打印出來的值是6。



這種修改變量的方法在這一次debug下會一直生效。開發web項目時，使用debug模式啟動了一次tomcat容器，則使用evaluate expression修改的變量值在這一次tomcat運行下都會生效。

##### b) Code View

The second place to get information is on the code itself.

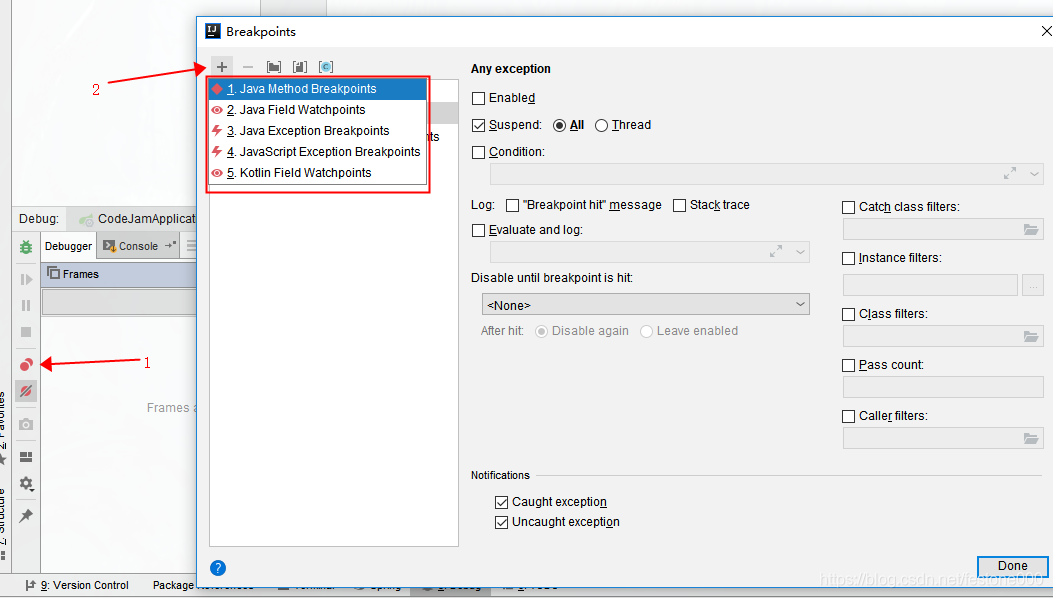


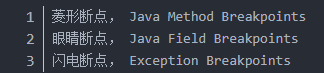
In debug mode, IntelliJ shows the value of every variable referenced in each line of code next to the line. In addition, it highlights the value of whatever variable was most recently changed.

## 4 IDEA的菱形斷點是什麽意思

在Idea中點擊1處，出現對話框。

點擊2處，可見紅框中內容。





## 5 Execution Breakpoints

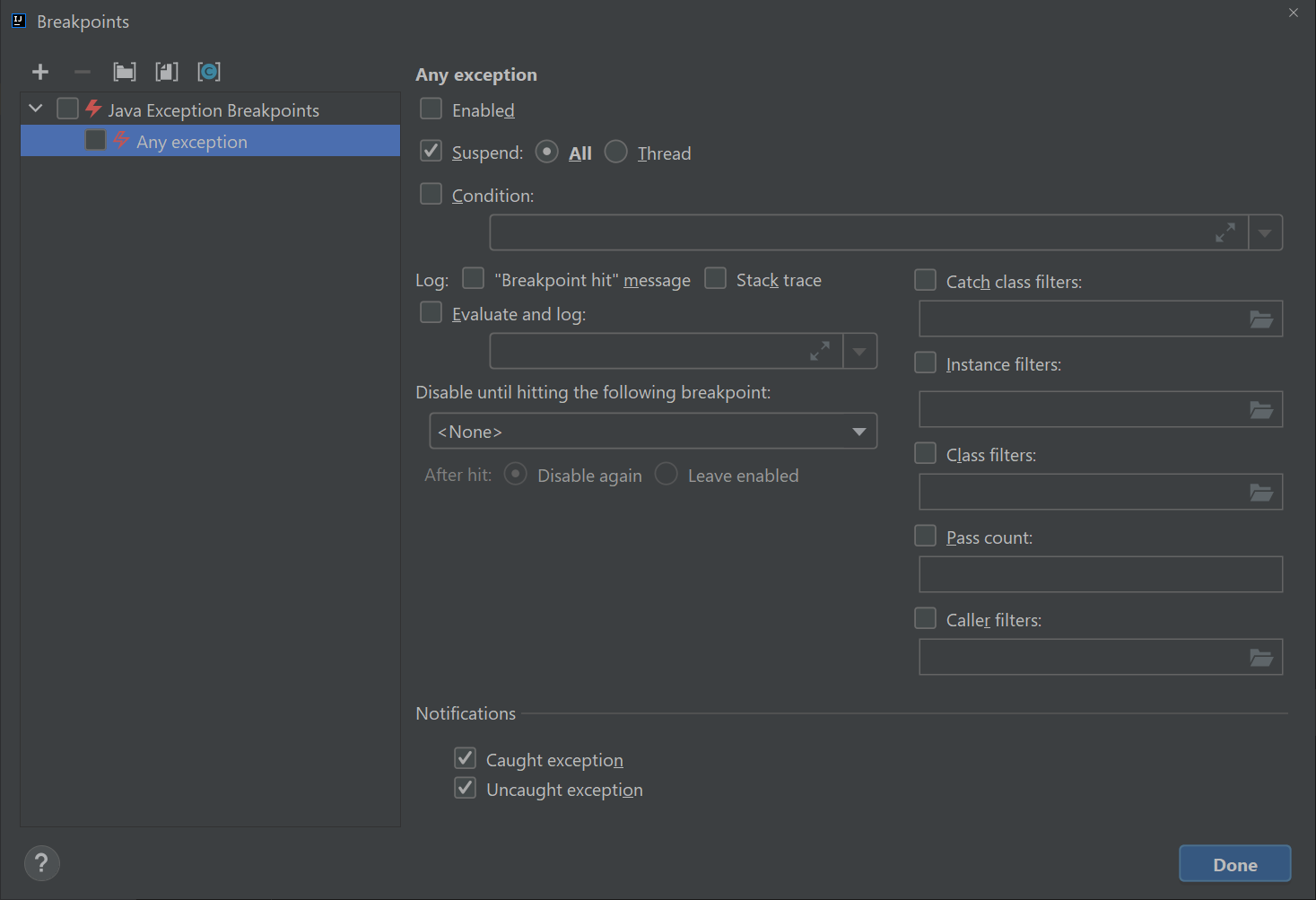
You’ll notice that every time the test fails, the message that we get is something like:

java.lang.ArrayIndexOutOfBoundsException: Index 7 out of bounds for length 7

at randomizedtest.BuggyAList.resize(BuggyAList.java:31)

set up an “Execution Breakpoint” so that we can stop the code and visualize what’s going on when your code crashes.

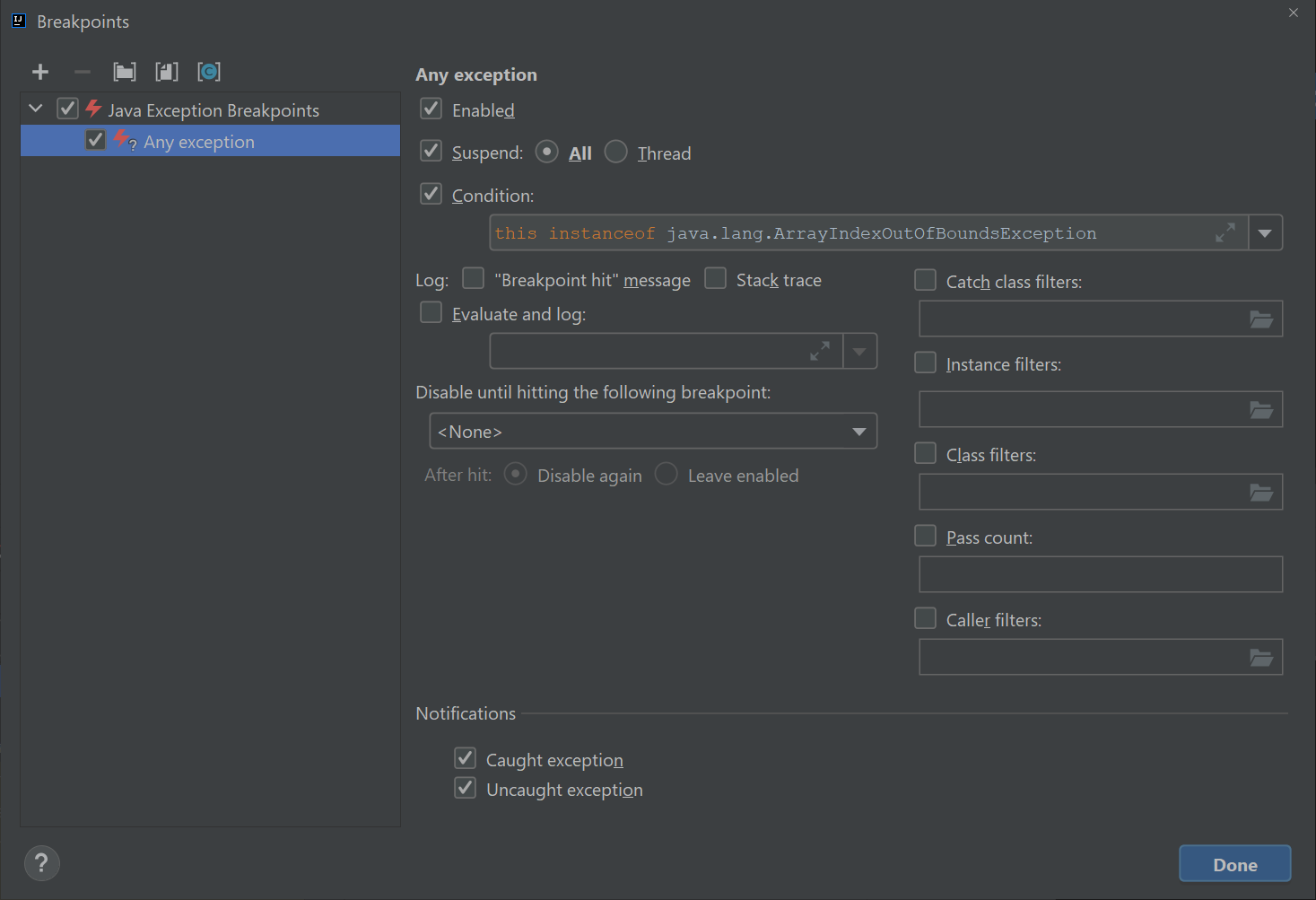
To this, click “Run -> View Breakpoints”. You should see a window like this pop up:



Click on the checkbox on the left that says “any exception” and then click on that says “Condition:” and in the window and enter exactly:

this instanceof java.lang.ArrayIndexOutOfBoundsException

Once you’ve done this, your breakpoints window should look like:



Click the debug button and your code should stop right at the moment the exception is about to occur. Click the visualizer and try to figure out why the code is crashing. Now the real problem solving can start!

**NOTE: If you use the debug feature without specifying a condition, your code will stop in some various mysterious places. Make sure you never have “Any Exception” checked without having a specified condition.**

This is because the process of starting JUnit tests generates a bunch of exceptions that ultimately get ignored. This is well beyond the scope of our class. If you’re done using an execution breakpoint, you should uncheck the “Java Exceptions Breakpoints” box in the top left.

Lists

## 1 IntLists

public class IntList {

public int first;

public IntList rest;

public IntList(int f, IntList r) {

first = f;

rest = r;

}

}

If we want to make a list of the numbers 5, 10, and 15, we can either do:

IntList L = new IntList(5, null);

L.rest = new IntList(10, null);

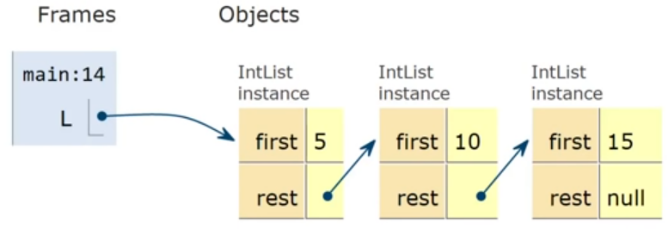
L.rest.rest = new IntList(15, null);

Alternately, we could build our list backwards, yielding slightly nicer but harder to understand code:

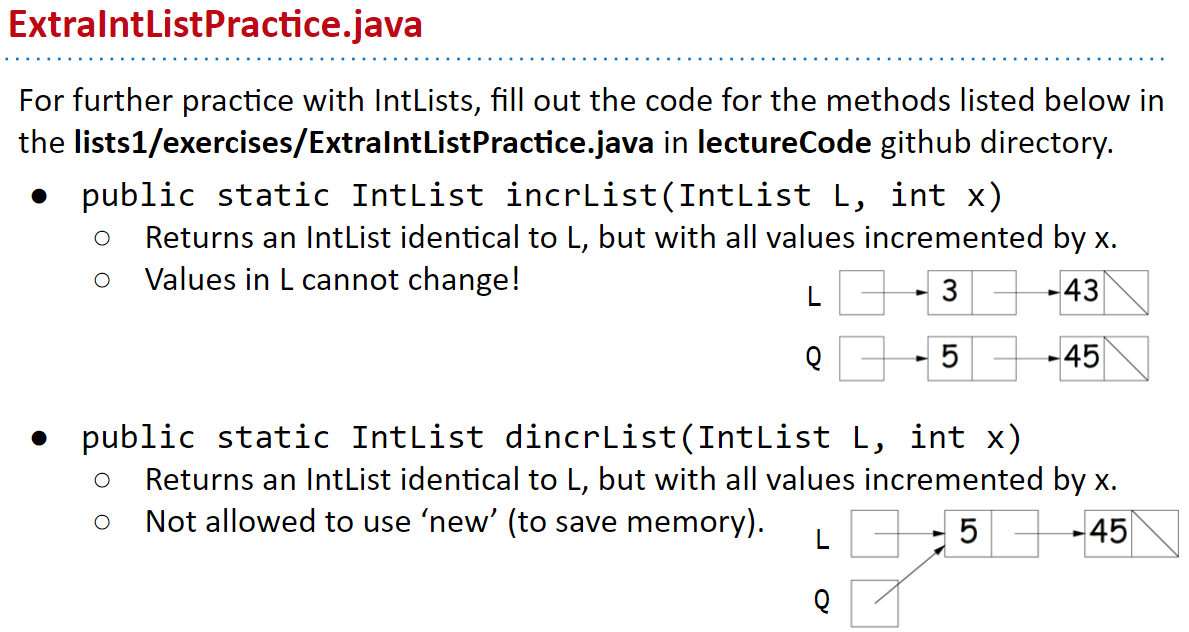
IntList L = new IntList(15, null);

L = new IntList(10, L);

L = new IntList(5, L);

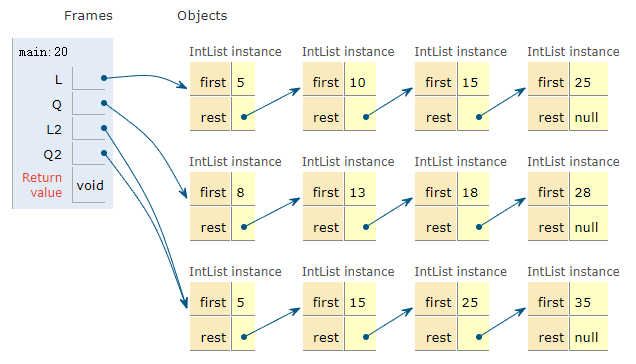


寫了兩個函數，完成了下列練習



incrList是new了一個Q，Q裡面的每個first都比L的first大x（例子裡x=3）。

dincrList是不能new Q2。直接把L2的每個first增加x後，把L2賦值給Q2.



public class IntList {

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

    IntList L = new IntList(25, null);

    L = new IntList(15, L);

    L = new IntList(10, L);

    L = new IntList(5, L);

    IntList Q = incrList(L, 3);

    System.out.print(Q);

    /\*\* ---------- \*/

    IntList L2 = new IntList(33, null);

    L2 = new IntList(23, L2);

    L2 = new IntList(13, L2);

    L2 = new IntList(3, L2);

    IntList Q2 = dincrList(L2, 2);

    System.out.print(Q2);

  }

  public int first;

  public IntList rest;

  public IntList(int *f*, IntList *r*) {

    first = f;

    rest = r;

  }

  /\*\*

   \* Returns an IntList identical to L, but with

   \* each element incremented by x. L is not allowed

   \* to change.

   \*/

  public static IntList incrList(IntList *L*, int *x*) {

    IntList Q = new IntList(L.first + x, null);

    incrListRecursion(L, Q, x);

    return Q;

  }

  public static IntList incrListRecursion(IntList *L*, IntList *Q*, int *x*) {

    if (L.rest != null) {

      Q.rest = new IntList(L.rest.first + x, null);

      incrListRecursion(L.rest, Q.rest, x);

    }

    return Q;

  }

  /\*\*

   \* Returns an IntList identical to L, but with

   \* each element incremented by x. Not allowed to use

   \* the 'new' keyword.

   \*/

  public static IntList dincrList(IntList *L*, int *x*) {

    L.first += x;

    if (L.rest != null) {

      dincrList(L.rest, x);

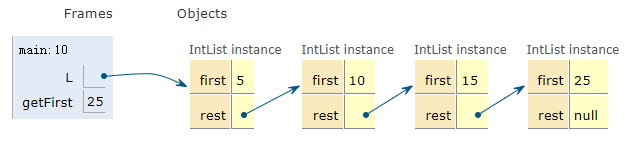
    }

    return L;

  }

}

Get方法獲取某個層級的first



public class IntList {

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

    IntList L = new IntList(25, null);

    L = new IntList(15, L);

    L = new IntList(10, L);

    L = new IntList(5, L);

    int getFirst = L.get(2);

    System.out.println(getFirst);

  }

  public int first;

  public IntList rest;

  public IntList(int *f*, IntList *r*) {

    first = f;

    rest = r;

  }

  public int get(int *i*) {

    if (i == 0) {

      return first;

    } else if (rest != null) {

      return rest.get(i - 1);

    } else {

      return -1;

    }

  }

}

## 2 SLList (Single Linked List)

### (1) public and private

public class SLList {

private IntNode first;

...

Private variables and methods can only be accessed by code inside the same .java file, e.g. in this case SLList.java. That means that a class like SLLTroubleMaker below will fail to compile, yielding a first has private access in SLList error.

public class SLLTroubleMaker {

public static void main(String[] args) {

SLList L = new SLList(15);

L.addFirst(10);

L.first.next.next = L.first.next;

}

}

The public keyword should be thought of as a declaration that a method is available and will work **forever** exactly as it does now.

**When you create a public member (i.e. method or variable), be careful, because you're effectively committing to supporting that member's behavior exactly as it is now, forever.**

### (2) Nested Classes

If the nested class has no need to use any of the instance methods or variables of SLList, you may declare the nested class static, as follows. Declaring a nested class as static means that methods inside the static class can not access any of the members of the enclosing class. In this case, it means that no method in IntNode would be able to access first, addFirst, or getFirst.

public class SLList {

public static class IntNode {

public int item;

public IntNode next;

public IntNode(int i, IntNode n) {

item = i;

next = n;

}

}

private IntNode first;

...

If you don't use any instance members of the outer class, make the nested class static.

### (3) overloaded methods

/\*\* Returns the size of the list starting at IntNode p. \*/

private static int size(IntNode p) {

if (p.next == null) {

return 1;

}

return 1 + size(p.next);

}

public int size() {

return size(first);

}

Here, we have two methods, both named size. This is allowed in Java, since they have different parameters. We say that two methods with the same name but different signatures are **overloaded**.

This is a typical coding pattern.

### (4) Catching

It is possible to rewrite size so that it takes the same amount of time, no matter how large the list.

To do so, we can simply add a **size** variable to the SLList class that tracks the current size, yielding the code below. This practice of saving important data to speed up retrieval is sometimes known as caching.

public class SLList {

... /\* IntNode declaration omitted. \*/

private IntNode first;

private int size;

public SLList(int x) {

first = new IntNode(x, null);

size = 1;

}

public void addFirst(int x) {

first = new IntNode(x, first);

size += 1;

}

public int size() {

return size;

}

...

}

### (5) Sentinel Nodes

One solution to fix addLast is to create a special case for the empty list, as shown below:

public void addLast(int x) {

size += 1;

if (first == null) {

first = new IntNode(x, null);

return;

}

IntNode p = first;

while (p.next != null) {

p = p.next;

}

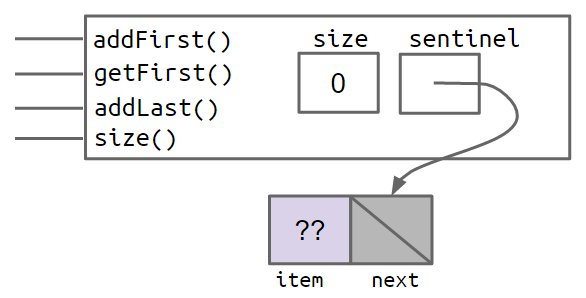
p.next = new IntNode(x, null);

}

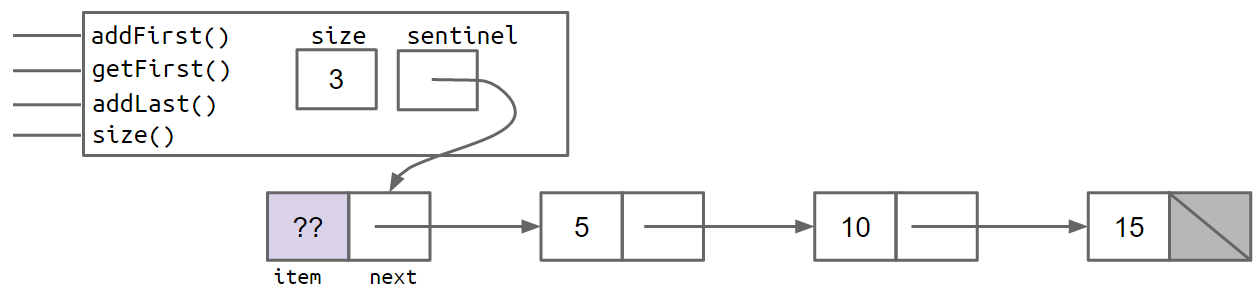
This solution works, but special case code like that shown above should be avoided when necessary. Human beings only have so much working memory, and thus we want to keep complexity under control wherever possible. For a simple data structure like the SLList, the number of special cases is small. More complicated data structures like trees can get much, much uglier.

A cleaner, though less obvious solution, is to make it so that all SLLists are the "same", even if they are empty. 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.

For example, the empty list created by SLList L = new SLList() would be as shown below:



And a SLList with the items 5, 10, and 15 would look like:



In the figures above, the lavender ?? value indicates that we don't care what value is there. Since Java does not allow us to fill in an integer with question marks, we just pick some abitrary value like -518273 or 63 or anything else. Since a SLList with a sentinel has no special cases, we can simply delete the special case from our addLast method, yielding:

public void addLast(int x) {

size += 1;

IntNode p = sentinel;

while (p.next != null) {

p = p.next;

}

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

}

### (6) Invariants

An invariant is a fact about a data structure that is guaranteed to be true (assuming there are no bugs in your code).

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

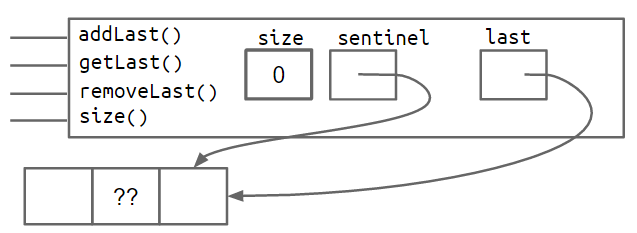
* The sentinel reference always points to a sentinel node.
* The first node (if it exists), is always at sentinel.next.
* The size variable is always the total number of items that have been added.

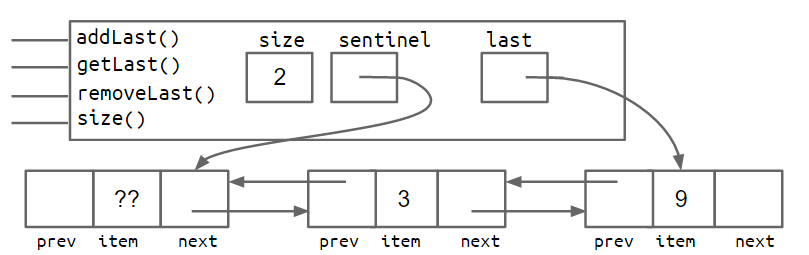
## 3 DLList (Doubly Linked List)

### (1) Looking Back

Our list now has two links for every node.

The box and pointer diagram below shows more precisely what a doubly linked list looks like for lists of size 0 and size 2, respectively.



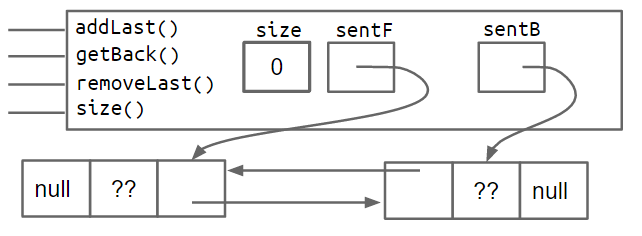


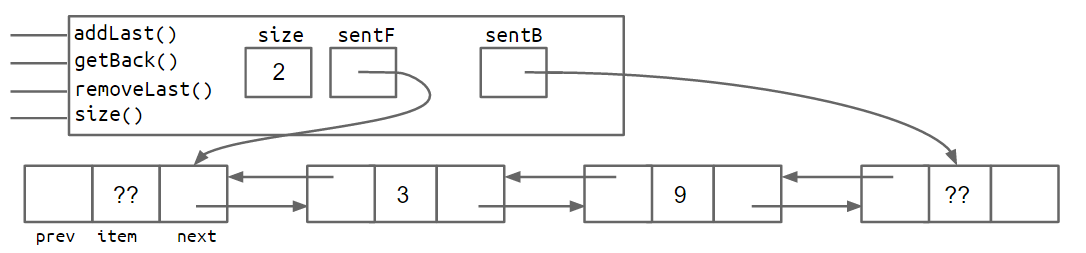
### (2) Sentinel Upgrade

Back pointers allow a list to support adding, getting, and removing the front and back of a list in constant time.

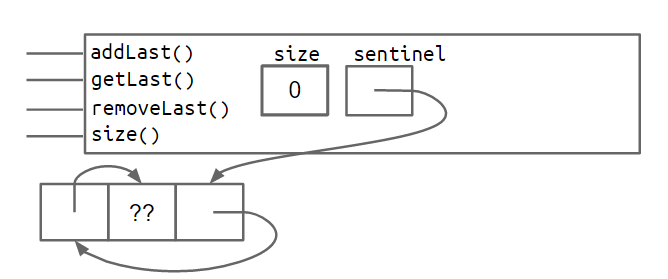
There is a subtle issue with this design where the last pointer sometimes points at the sentinel node, and sometimes at a real node. Just like the non-sentinel version of the SLList, this results in code with special cases that is much uglier than what we'll get after our 8th and final improvement.

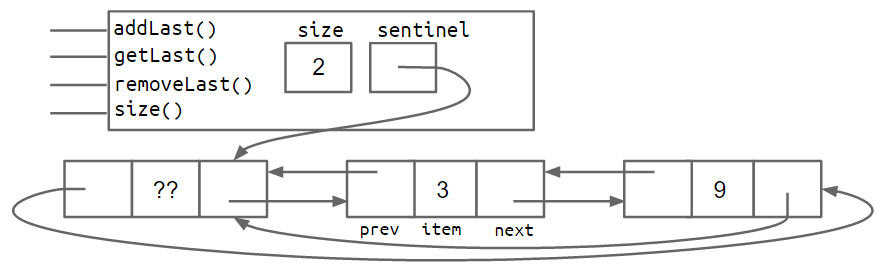
One fix is to add a second sentinel node to the back of the list. This results in the topology shown below as a box and pointer diagram.





An alternate approach is to implement the list so that it is circular, with the front and back pointers sharing the same sentinel node.





Both the two-sentinel and circular sentinel approaches work and result in code that is free of ugly special cases, though I personally find the circular approach to be cleaner and more aesthetically beautiful.

### (3) Generic DLLists

Generics will allow you to create data structures that hold any reference type.

The basic idea is that right after the name of the class in your class declaration, you use an arbitrary placeholder inside angle brackets: <>. Then anywhere you want to use the arbitrary type, you use that placeholder instead. For example, our DLList declaration before was:

public class DLList {

private IntNode sentinel;

private int size;

public class IntNode {

public IntNode prev;

public int item;

public IntNode next;

...

}

...

}

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

public class DLList<BleepBlorp> {

private StuffNode sentinel;

private int size;

public class StuffNode {

public StuffNode prev;

public BleepBlorp item;

public StuffNode next;

...

}

...

}

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);

Some rules of thumb:

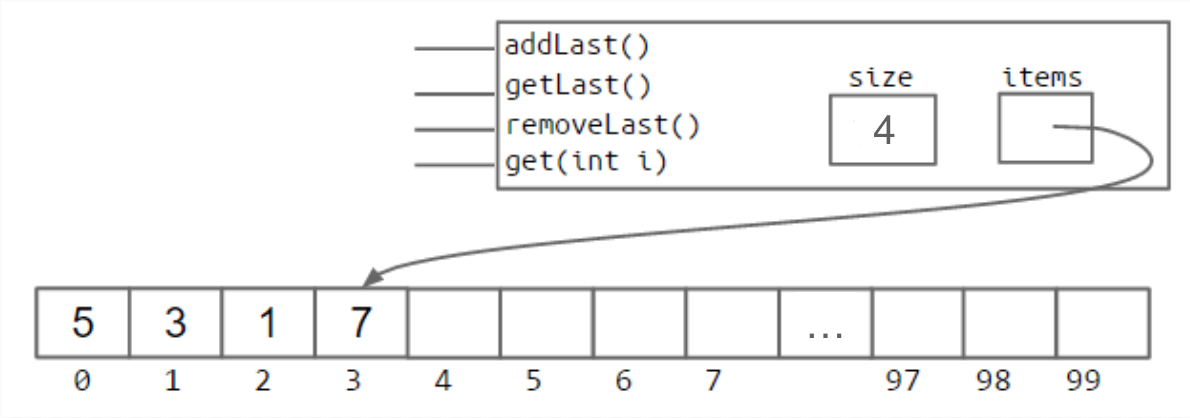
* In the .java file **implementing** a data structure, specify your generic type name only once at the very top of the file after the class name.
* In other .java files, which use your data structure, specify the specific desired type during declaration, and use the empty diamond operator during instantiation.
* 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.

## 4 AList

### （1）Memory Performance

Our AList is almost done, but we have one major issue. Suppose we insert 1,000,000,000 items, then later remove 990,000,000 items. In this case, we'll be using only 10,000,000 of our memory boxes, leaving 99% completely unused.

To fix this issue, we can also downsize our array when it starts looking empty. Specifically, we define a "usage ratio" R which is equal to the size of the list divided by the length of the items array. For example, in the figure below, the usage ratio is 0.04.



In a typical implementation, we halve the size of the array when R falls to less than 0.25.

### （2）Generic ALists

①

Java does not allow us to create an array of generic objects due to an obscure issue with the way generics are implemented. That is, we cannot do something like:

Glorp[] items = new Glorp[8];

Instead, we have to use the awkward syntax shown below:

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

This will yield a compilation warning, but it's just something we'll have to live with. We'll discuss this in more details in a later chapter.

②

The other change we make is that we null out any items that we "delete".

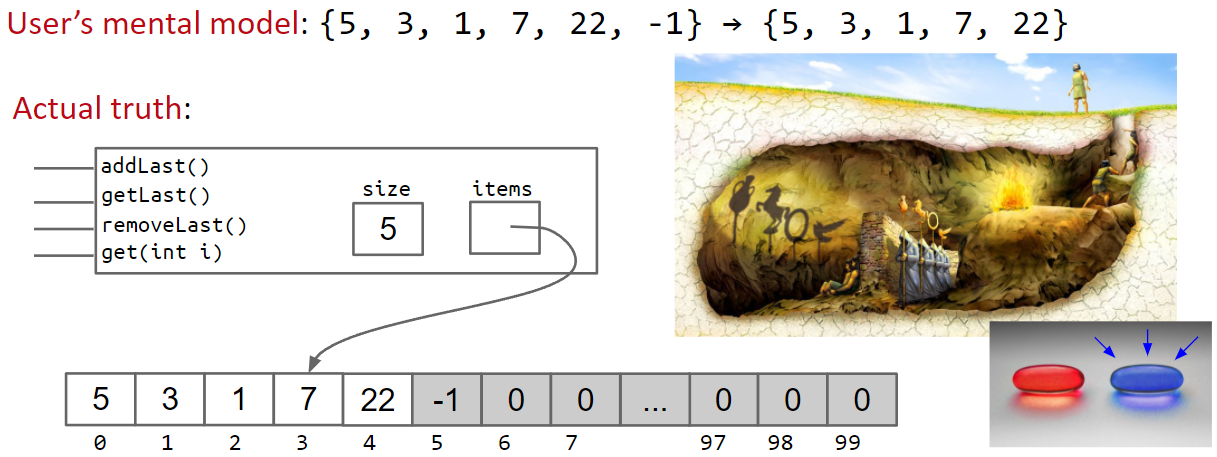
Whereas before, we had no reason to zero out elements that were deleted, with generic objects, we do want to null out references to the objects that we're storing. This is to avoid "loitering". Recall that Java only destroys objects when the last reference has been lost. If we fail to null out the reference, then Java will not garbage collect the objects that have been added to the list.

This is a subtle performance bug that you're unlikely to observe unless you're looking for it, but in certain cases could result in a significant wastage of memory.

### （3）Obscurantism in Java

We talk of “layers of abstraction” often in computer science.

Related concept: obscurantism. The user of a class does not and should not know how it works.



The Java language allows you to enforce this with ideas like private!

A good programmer obscures details from themselves, even within a class.

* Example: addFirst and resize should be written totally independently. You should not be thinking about the details of one method while writing the other. Simply trust that the other works.
* Breaking programming tasks down into small pieces (especially functions) helps with this greatly!
* Through judicious use of testing, we can build confidence in these small pieces, as we’ll see in the next lecture.

## 5 Higher Order Functions

A higher order function is a function that treats other functions as data. For example, take this Python program do\_twice that takes in another function as input, and applies it to the input x twice.

def tenX(x):

return 10\*x

def do\_twice(f, x):

return f(f(x))

A call to print(do\_twice(tenX, 2)) would apply tenX to 2, and apply tenX again to its result, 20, resulting in 200. How would we do something like this in Java?

In old school Java (Java 7 and earlier), memory boxes (variables) could not contain pointers to functions. What that means is that we could not write a function that has a "Function" type, as there was simply no type for functions.

To get around this we can take advantage of interface inheritance. Let's write an interface that defines any function that takes in an integer and returns an integer - an IntUnaryFunction.

public interface IntUnaryFunction {

int apply(int x);

}

Now we can write a class which implements IntUnaryFunction to represent a concrete function. Let's make a function that takes in an integer and returns 10 times that integer.

public class TenX implements IntUnaryFunction {

/\* Returns ten times the argument. \*/

public int apply(int x) {

return 10 \* x;

}

}

At this point, we've written in Java the Python equivalent of the tenX function. Let's write do\_twice now.

public static int do\_twice(IntUnaryFunction f, int x) {

return f.apply(f.apply(x));

}

A call to print(do\_twice(tenX, 2)) in Java would look like this:

System.out.println(do\_twice(new TenX(), 2));