CSE 12:
Basic data structures and object-oriented design

Jacob Whitehill
jake@mplab.ucsd.edu

Lecture Two
2 Aug 2011

Scheduler (demo)

Review from last lecture

- In computer science, all data must ultimately be represented as a binary sequence.
- Data structures are necessary to encode useful information in binary sequences.
- Data structures may vary in their time complexity, space complexity, and “code complexity” (human effort).

Review from last lecture

- It is important to learn the fundamental data structures of computer science so you don't keep having to "rediscover the wheel".
- The fundamental data structures covered in this course include: **lists**, **stacks**, **queues**, **heaps**, **trees**, **hash tables**, and **graphs**.

Fundamental data structures

- 5 of these structures (list, stack, queue, heap, hash table) are useful as **collections** to support *add*/*find*/*remove* operations.
  - In coarse English, a collection is useful if the programmer wants to "put data in it", and later "pull data out of it."
    - E.g., you're writing a program to manage the financial aid of all UCSD students. You want "some structure" (collection) to hold all the **`UCSDStudent`** objects while the program is running -- you don't want to manage the data yourself.

Fundamental data structures

- Different collections have different time and space costs for the add/retrieve/remove operations.
  - Which collection is best depends on which operations your code calls most often.

Fundamental data structures

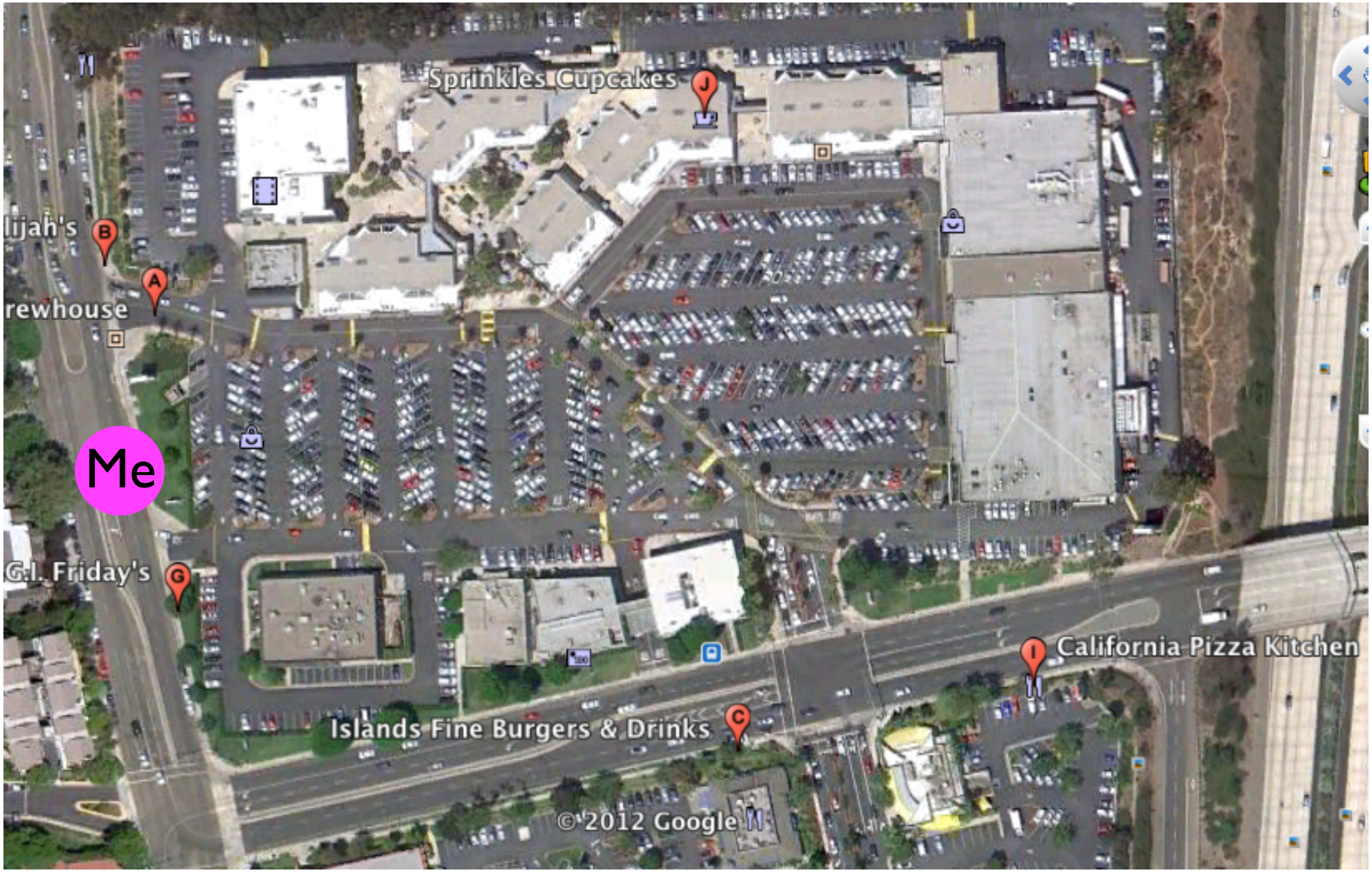
- 2 of these structures (tree, graph) are useful to represent connectivity relationships among data:
  - Trees can represent hierarchical relationships (e.g., heredity).
  - Graphs can represent arbitrary relationships between pairs of data (e.g., Facebook friends).

Fundamental data structures

- In this course we will develop all of these data structures as Abstract Data Types (ADTs).
- In this lecture I hope to:
  - Explain abstraction from a computer system's perspective.
  - Motivate building data structures as ADTs.
  - Introduce our first ADT of the course: the **list**.

Review of
Unannounced Quiz 0

Which markers are close to me?



Unannounced Quiz 0

- Given:

```
class Location {
    // ...
}
class Marker {
    public boolean isCloseTo (Location location) {
       // ...
    }
}
```

Unannounced Quiz 0

- Objective was to implement `GooglePlanet` method:

```
// Return an array of Marker objects that are
// "close to" the specified location.
Marker[] findLocalMarkers (Location location) {




}
```

Unannounced Quiz 0

- Objective was to implement **`GooglePlanet`** method:

```
// Return an array of Marker objects that are
// "close to" the specified location.
Marker[] findLocalMarkers (Location location) {
  Create empty list localMarkers
  For each marker in _markers:
    If marker.isCloseTo(location):
        Add marker to localMarkers
  Return localMarkers
}
```

- In actual Java code, this becomes surprisingly tedious...

```
public Marker[] findLocalMarkers (Location location) {
   Marker[] localMarkers = new Marker[128];  // initialize to some small size
   int idx = 0;
   for (Marker marker : _markers) {
      if (marker.isCloseTo(location)) {
         if (idx == localMarkers.length) {  // Array is full
            // Allocate a new array twice as big as the last one
            Marker[] newLocalMarkers = new Marker[2*localMarkers.length];
            // Copy the old array into the new array
            for (int i = 0; i < localMarkers.length; i++) {
               newLocalMarkers[i] = localMarkers[i];
            }
            // Now, make localMarkers point to that *new* array -- the "old"
            // version of localMarkers will be swept away by the garbage collector.
            localMarkers = newLocalMarkers;
         }
         // Now, we know we definitely have enough room to store one more marker
         localMarkers[idx] = marker;
         idx++;
      }
   }
   // We still have to "trim down" the localMarkers array to the
   // exact number of Marker objects that we actually added --
   // this is recorded in idx. Let's allocate one more Marker[]
   // to store the correct number of objects.
   Marker[] newLocalMarkers = new Marker[idx];
   for (int i = 0; i < idx; i++) {
      newLocalMarkers[i] = localMarkers[i];
   }
   localMarkers = newLocalMarkers;
   return localMarkers;
}
```

```java
public Marker[] findLocalMarkers (Location location) {
Marker[] localMarkers = new Marker[128];  // initialize to some small size
int idx = 0;
for (Marker marker : _markers) {
    if (marker.isCloseTo(location)) {
        if (idx == localMarkers.length) {  // Array is full
            // Allocate a new array twice as big as the last one
            Marker[] newLocalMarkers = new Marker[2*localMarkers.length];
            // Copy the old array into the new array
            for (int i = 0; i < localMarkers.length; i++) {
                newLocalMarkers[i] = localMarkers[i];
            }
            // Now, make localMarkers point to that *new* array -- the "old"
            // version of localMarkers will be swept away by the garbage collector.
            localMarkers = newLocalMarkers;
        }
        // Now, we know we definitely have enough room to store one more marker
        localMarkers[idx] = marker;
        idx++;
    }
}
// We still have to "trim down" the localMarkers array to the
// exact number of Marker objects that we actually added --
// this is recorded in idx. Let's allocate one more Marker[]
// to store the correct number of objects.
Marker[] newLocalMarkers = new Marker[idx];
for (int i = 0; i < idx; i++) {
    newLocalMarkers[i] = localMarkers[i];
}
localMarkers = newLocalMarkers;
return localMarkers;
}
```

All this code is “nuisance” code.

Unannounced Quiz 0

- Suppose there already existed a class called **`ArrayList`** that allowed us to:
  - *add* data to it
  - *retrieve* data from it using an index; and
  - would *resize* itself automatically?
- Our own code becomes much simpler...

```
public ArrayList findLocalMarkers (Location location) {
    ArrayList localMarkers = new ArrayList();

    for (Marker marker : _markers) {
        if (marker.isCloseTo(location)) {
            localMarkers.add(marker);
        }
    }

    return localMarkers;
}
```

Unannounced Quiz 0

- What would this hypothetical **`ArrayList`** class look like?
- It would certainly need an **`add`** method:

```
class ArrayList {
    void add (Object o) { ... }
    // ...
}
```

- All of the "nuisance" code would go into these methods.

Unannounced Quiz 0

- In writing the **`findLocalMarkers`** method, we could then just *use* this **`ArrayList`** class.
  - We are the **user** of this **`ArrayList`**.
- Someone else would then have to **implement** the class by writing the actual implementation of **`ArrayList.add(o)`**.
  - They are the **implementor** of the class.

Unannounced Quiz 0

- Separating the user from the implementor facilitates an elegant division of labor in writing software.

Data structures you're already familiar with.

Data structures you already know

- In prior coursework you have already worked with some simple data structures:
  - **Arrays**:

```
int[] numbers = new int[100];
...
numbers[5] = 16;
```

Data structures you already know

- In prior coursework you have already worked with some simple data structures:
  - **Arrays**: collection of related variables specified by an index:

```
int[] numbers = new int[100];
...
numbers[5] = 16;
```

More convenient than declaring 100 variables!

```
int number1, number2, number3, ...number100;
```

Data structures you already know

- **Strings**:

```
String firstName = "Jimmy";
String lastName = "Carter";
String fullName = firstName + " " + lastName;
System.out.println("Hello, " + fullName);
```

Data structures you already know

- **Strings**: a finite sequence of characters:

```
String firstName = "Jimmy";
String lastName = "Carter";
String fullName = firstName + " " + lastName;
System.out.println("Hello, " + fullName);
```

String data structure allows you to "add" strings together.

Data structures you already know

- In other languages (e.g., C), a string is simply an array of characters:

```
char str1[32] = "angry";  // str of max len 32
char str2[32] = "bird";
```

- You can't concatenate two strings simply by "adding" them:

```
char str3[64] = str1 + str2;
```

Data structures you already know

- **Records**:

```
class Customer {
  String _name;
  int _age;
  float _accountBalance;
}
```

Data structures you already know

- **Records**: a group of related variables:

```
class Customer {
  String _name;
  int _age;
  float _accountBalance;
}
```

Records alleviate the burden of maintaining "whose name goes with whose age and whose balance?"

Data structures you already know

- Simple data structures like arrays, strings, and records provide conveniences to the programmer.
- However, these structures are not physically present anywhere in the computer.
  - They are *not real*; they are **abstract**.

    Merriam-Webster: existing in thought or as an idea but not having a physical or concrete existence
- In contrast, **bits** (0/1) are physically present -- they encode whether a particular transistor is on/off.

Abstraction for convenience.

Memory abstraction

- Even the "one long sequence of 1's and 0's" from last lecture is abstract:
  - In fact, computers typically have *multiple* long sequences of 0's and 1's -- one for each *memory chip* in the machine.

0010011110001001110100101101100111001100100001010000...

2GB

1010001110101110011100110100010111101101010010111...

2GB

2GB

110101000011000110011111001010000101010110...

Memory abstraction

- Hence, if we want to write to or read from a particular byte of memory, we must specify both which chip (A, B, or C) and which location on that chip (anywhere from 0 to 2147483647).

A

0010011110001001110100101101100111001100100001010000...

2GB

B

1010001110101110011100110100010111101101001011 1...

2GB

C

2GB

11010100001100011001111100101000010101 10...

Memory abstraction

- How is this related to computer programming?
  - Every variable in every program you write must be stored in memory *somewhere*.

A

0010011110001001110100101101100111001100100001010000...

2GB

B

1010001110101110011100110100010111101101001011 1...

2GB

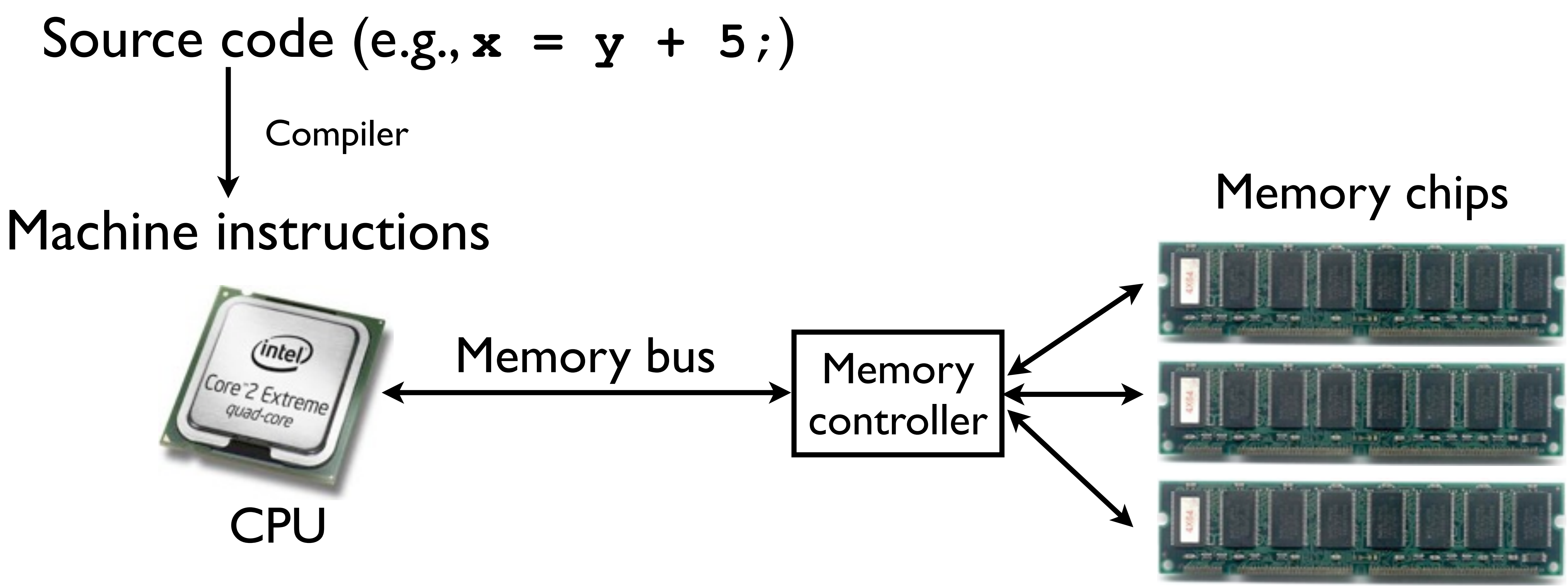
C

2GB

110101000011000110011111001010000101 0110...

Foray into computer architecture

- Somewhere between your source code and the memory chips, the determination of "which memory chip" must be made...

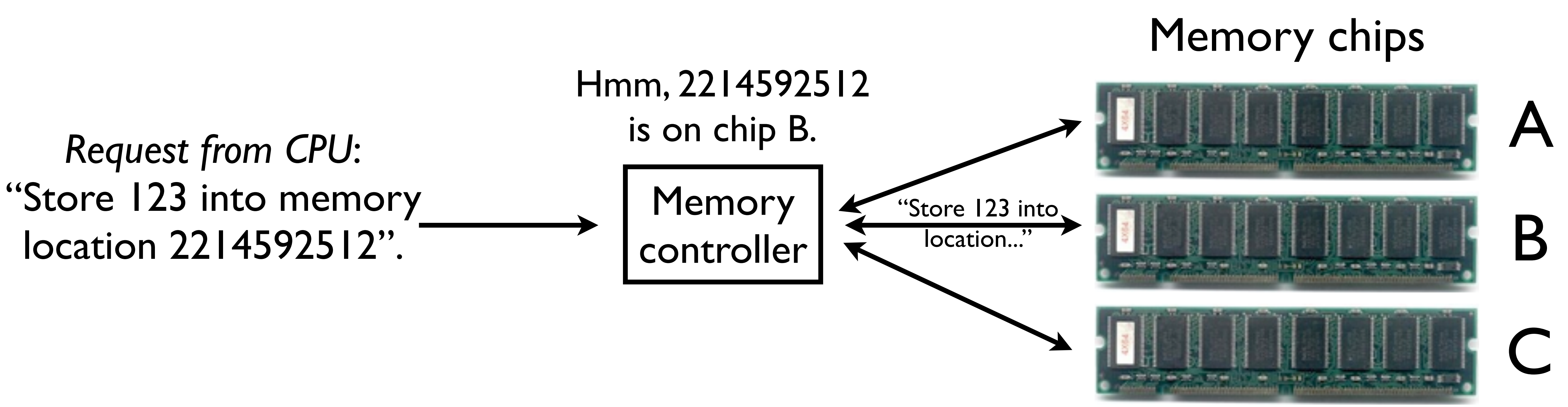


Memory abstraction

- The memory controller provides a "convenient illusion":
  - It allows the CPU, compiler, and ultimately our Java code to "pretend" there's only one large bank of memory of size 6GB.
  - No need to specify "memory chip A, B, or C".
  - Just specify the byte location you're interested in (anywhere from 0 to 6442450943).
- This illusion is called an "abstraction".

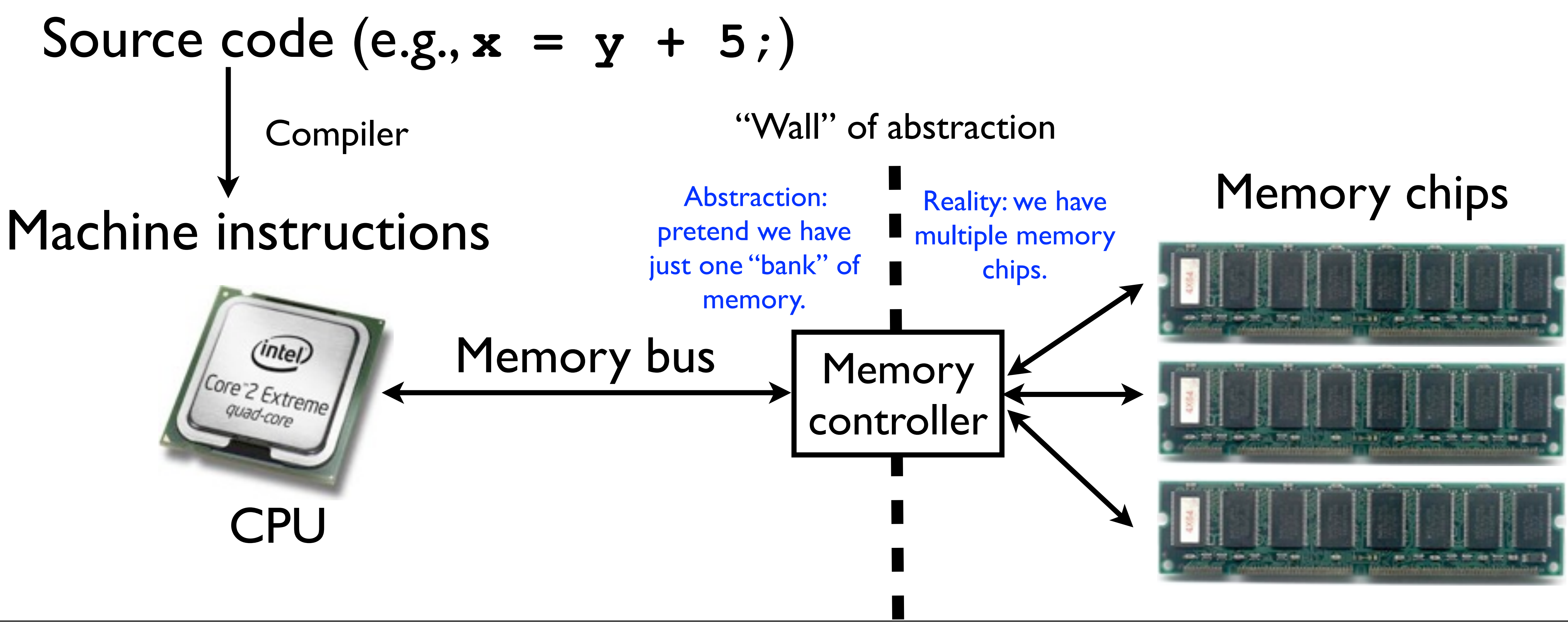
Memory abstraction

- Memory controller must "translate" between "abstract" requests of the CPU and "reality" of multiple memory chips.



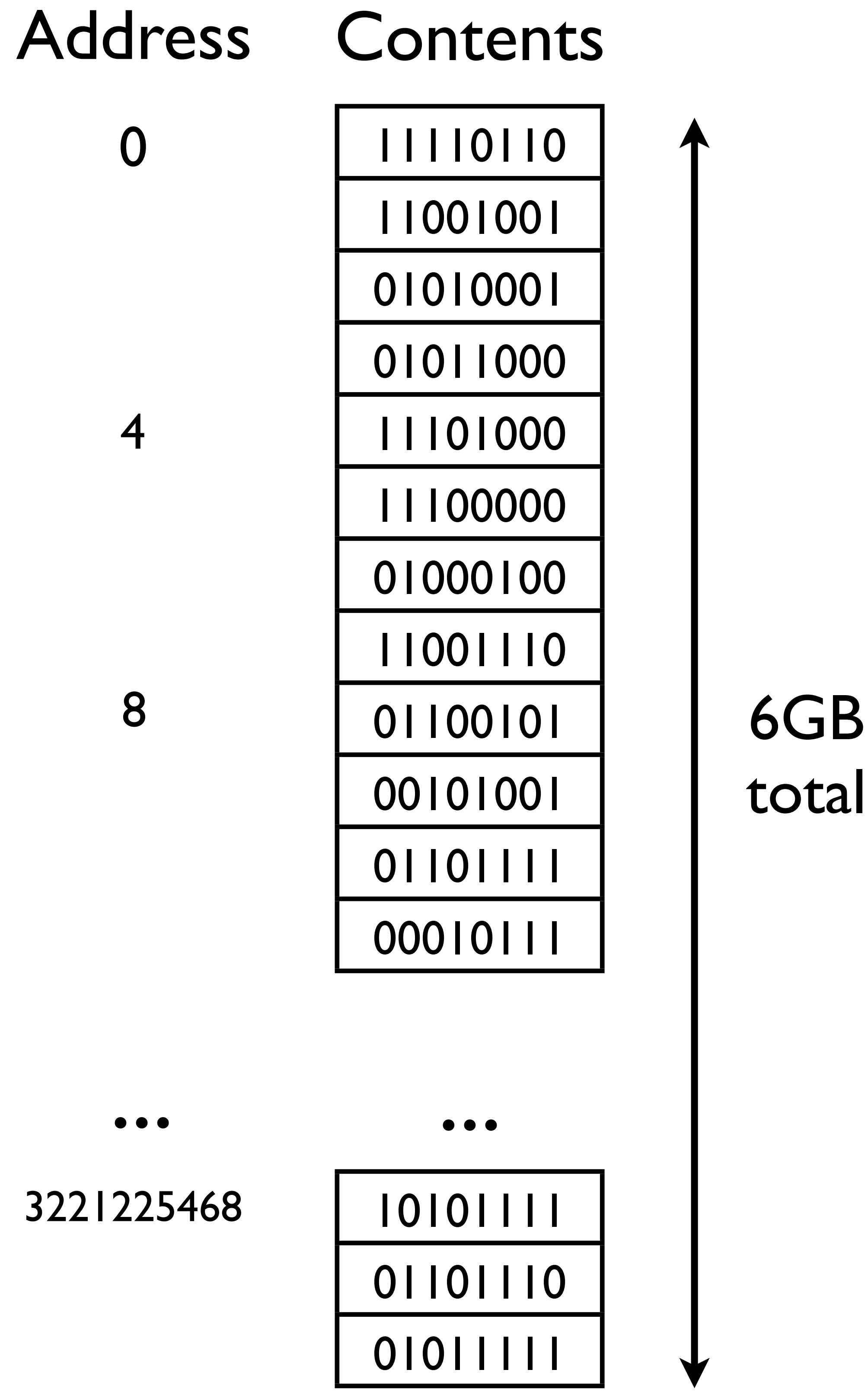
Memory abstraction

- Thanks to this "memory abstraction", the CPU, operating system, Java compiler, and ultimately you-the-programmer don't have to worry about which memory chip your variables are stored.



Memory addresses

- The memory controller provides us with the "abstraction" of viewing memory as **one, long sequence of bytes** (8 bits each).
- Each location in the memory bank is called an **address**.



Memory controller implements `OneLongBinarySequence` abstraction

- The memory controller is responsible for *implementing* this abstraction.
- The memory controller must handle requests/ **messages** from the CPU and respond to them appropriately.
- Example requests:
    - “Store value 123 into address 2152420584.”
    - “Fetch the value stored at address 2152420584.”

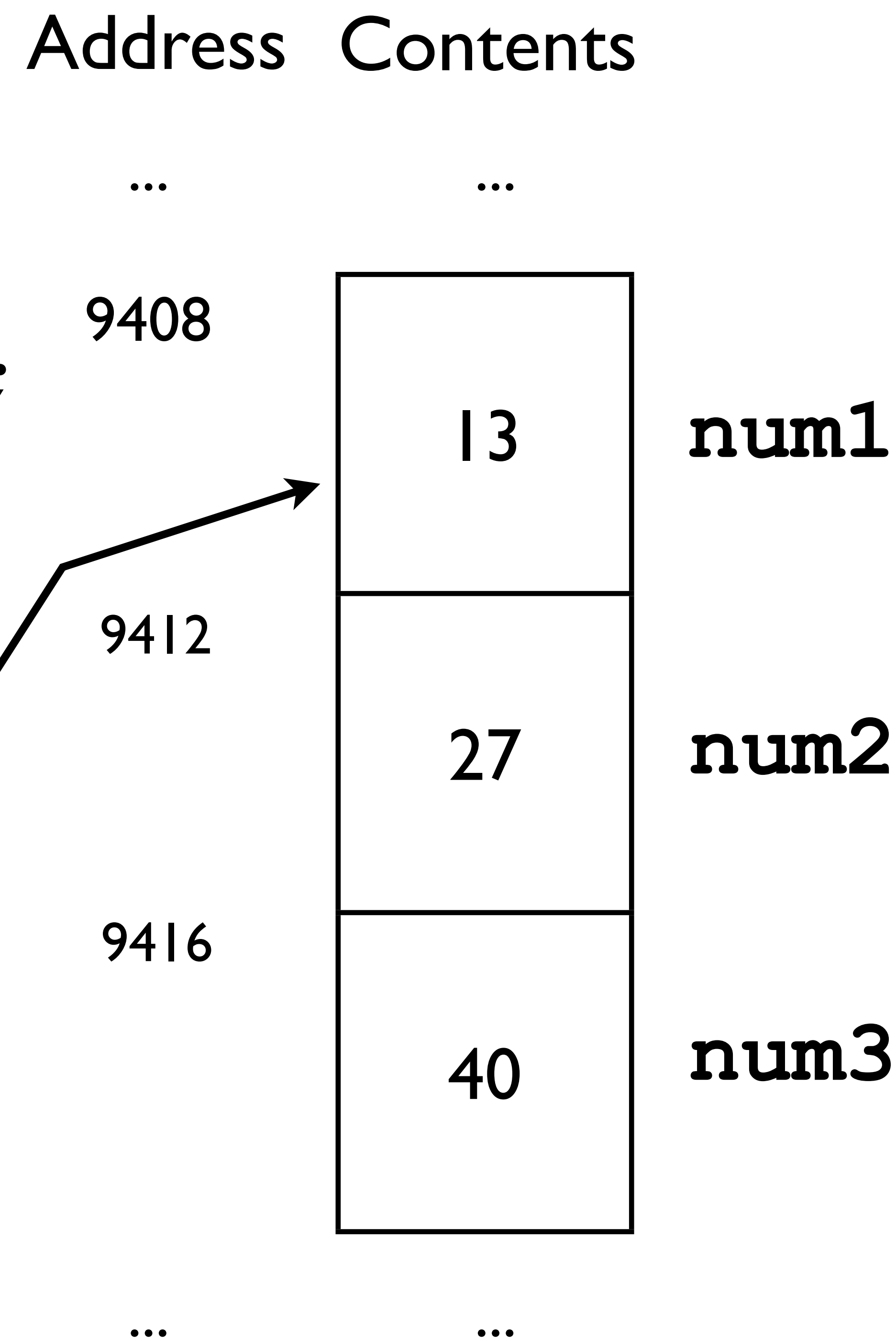
Programming language abstractions

- In this course, we will deal with abstractions primarily at the **programming language** and **data structure** level.
- Programming languages allow us to refer to data using meaningful variable names, e.g.,
  `int imageWidth;`
  instead of referring to particular memory addresses, e.g., **`4938248`**.

Programming language abstractions

- Example:

```
void addNumbers () {
  int num1 = 13, num2 = 27;
  int num3 = num1 + num2;
}
```



The compiler/interpreter implements the abstraction, i.e., translates between variable names and memory addresses.

Point to emphasize

- Abstraction provides a *convenient illusion*:
  - The **`OneLongBinarySequence`** is more convenient than having to know on which memory card a particular byte is stored.
  - A variable name is easier to remember than an integer memory address.

Point to emphasize

- Abstractions are not "real":
  - The `OneLongBinarySequence` is actually divided across several memory chips.
  - A variable is actually just a region of computer memory starting at a particular address.

Abstraction to hide details.

Data structure abstractions

- In this course, we will study some of the fundamental data structures of computer science: **list**, **stack**, **queue**, **heap**, **tree**, **hash table**, and **graph**.
- Each of these provides a convenient **abstraction** to the programmer.
- We implement these data structures as **abstract data types** (**ADT**s).

Abstract data types

- An abstract data type (ADT) provides the programmer with a convenient "container" for storing data.
  - For instance, a **list** is an abstraction for a container of ordered elements that can grow as we add more elements to it.
- The programmer interacts with the ADT by calling various **methods** on it.

Abstract data types

- The details of how the methods are implemented are generally not visible to the "user".
  - The "user" is the programmer who wants to use the ADT to manage his/her data.
  - The user doesn't necessarily care how the ADT is implemented, as long as the methods work according to the **interface specification**.
  - This allows flexibility in the **implementation** of the ADT.

ADT example

- This discussion of abstract data types may be getting “abstract”.
- Let’s concretify things by introducing one of the classics: a **list**.

Lists

- Sometimes you need to manage a collection of variables:
  - Students enrolled at UCSD.
  - Customers who buy stuff from your company.
  - List of programs currently running on your machine.

Lists

- So...just use an array:

```
Student[] ucsdStudents = new
Student[28000];
```

Linked lists

- But what if the number of students is not known ahead of time?
  - We could just allocate a really big array with room to spare.

```
ucsdStudents = new Student[100000];
```

Why not use an array?

- There are two problems with this:
  - It is wasteful -- many elements of **`ucsdStudents`** will never be used.
  - If we try to allocate too big an array, then the initialization may *fail*, due to:
    - Lack of free memory; or
    - Lack of *contiguous* free memory (i.e., available in one big block).

Why not use an array?

- Ok, fine -- start out with a small array, and make it bigger when it's full.
  - But it's annoying for the programmer to have to keep "enlarging the array".
- What we want is an object that manages the array for us.
  - We don't really care how it's done, as long as it works.
    - *We're not concerned with the details.*

What we want

- What we want is some data structure that has the following capabilities:
  - We can add elements (e.g., `Students`) to it, and it will store them.
    - The data structure should automatically "grow" itself as needed in an "efficient" manner (much more later).

      Time cost
    - It should not use memory wastefully.

      Space cost

What we want

- We can retrieve a particular element specified by index *i*.
- We can remove a particular element specified by index *i*.

List interface specification

- Here's a Java specification of what we want:

For now, just take this to mean "quickly".

```
class List {

  ...

  // Adds the specified element to the end of the list.
  // Takes O(1) time.
  void add (Object element) { ... }

  // Returns the element contained in the list at index
  // i if it exists. Else, throws NoSuchElementException.
  Object get (int i) throws NoSuchElementException { ... }

  // Removes the element contained in the list at index
  // i if it exists. Else, throws NoSuchElementException
  void remove (int i) throws NoSuchElementException { ... }
}
```

List interface specification

- Notice the things we **don't** care about:

```
class List {

  ...

  // Adds the specified element to the end of the list.
  // Takes O(1) time.
  void add (Object element) { ... }

  // Returns the element contained in the list at index
  // i if it exists. Else, throws NoSuchElementException.
  Object get (int i) throws NoSuchElementException { ... }

  // Removes the element contained in the list at index
  // i if it exists. Else, throws NoSuchElementException
  void remove (int i) throws NoSuchElementException { ... }
}
```

Don't care about the **instance variables**.

Don't care *how* the methods work.

List interface specification 

- Notice the things we **do** care about:

```
class List {
  ...
    // Adds the specified element to the end of the list.
    // Takes O(1) time.
    void add (Object element) { ... }

    // Returns the element contained in the list at index
    // i if it exists. Else, throws NoSuchElementException.
    Object get (int i) throws NoSuchElementException { ... }

    // Removes the element contained in the list at index
    // i if it exists. Else, throws NoSuchElementException
    void remove (int i) throws NoSuchElementException { ... }
}
```

We care about what the methods **return**.

We care about the **parameters** we must pass in.

We care what the methods do.

We care what exceptions it might throw (more later).

List specification

- A description of methods...
  - What the methods do.
  - What parameters they take.
  - What they return.
  - What exceptions they might throw.
- ...is known as an **interface**.
- An **`interface`** in Java contains:
  - No instance variables.
  - No method bodies.

List interface

An interface consists of method **signatures**.

```
interface List {
  // Adds the specified element to the end of the list.
  // Takes O(1) time.
  void add (Object element);

  // Returns the element contained in the list at index
  // i if it exists. Else, throws NoSuchElementException.
  Object get (int i) throws NoSuchElementException;

  // Removes the element contained in the list at index
  // i if it exists. Else, throws NoSuchElementException
  void remove (int i) throws NoSuchElementException;
}
```

A method signature consists of the method name, parameters, return type, and exceptions thrown.

Using a list

- Before we can use a **`List`** object, we first need some class that implements the **`List`** interface.
- The **`List`** is an abstraction -- we can't create **`List`** objects by writing:

```
List list = new List();  // Won't compile
```

  - The reason is that **`List`** is just a description of what a list *should do* -- not how it would *actually work*.

Implementing the `List` interface

- In order to create an instance of `List`, you must first create a (concrete) *class* that *implements* the (abstract) `List` *interface*.
- What does this mean?
  - It means that we must implement the **body** of every method whose signature was defined in the interface.

Implementing the `List` interface

```
class ListImpl implements List {
  private Object[] _array;
  private int _numElements;

  void add (Object element) {
    ...
    _array[_numElements++] = element;
  }

  Object get (int i) throws NoSuchElementException {
    ...
  }

  void remove (int i) throws NoSuchElementException {
    ...
  }
}
```

Tell the compiler explicitly that `ListImpl` implements the `List` interface.

Creating a `List`

- Now that we (hypothetically) have a **`ListImpl`** implementation of **`List`**, we can create a List object:

```
List list = new ListImpl();  // ok!
list.add(new Student("Bertha", 18));
...
```

Abstraction for good software design.

Why separate interface from implementation?

- So far, creating a `List` interface and a `ListImpl` implementation hasn't bought us very much.
- Why is it useful?

Why separate interface from implementation?

1. Separating interface from implementation facilitates a *division of labor* among members of a software development team.

I'll work on the graphical front-end to manage a list of UCSD students.

User

Fabulous. I'll create the List implementation itself.

Implementor

Photos courtesy of Google Image Search.

Why separate interface from implementation?

1. Separating interface from implementation facilitates a *division of labor* among members of a software development team.

    1. Both the implementors and users of the ADT agree on the interface.

    2. The implementor **implements** the interface (writes the ADT method bodies).

    3. The user **calls** the interface methods.

Why separate interface from implementation?

List *interface*

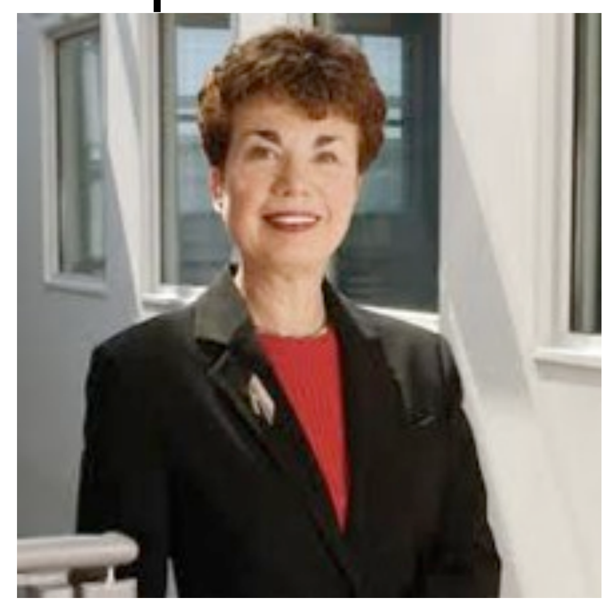
User

writes:

```
List list = new ListImpl();
list.add(new Student());
...
```

Wall of abstraction

Implementor



writes:

```
class ListImpl implements List {
  ...
  void add (Object o) {
    _array[_numElements++] = o;
  }
}
```

Photos courtesy of Google Image Search.

Why separate interface from implementation?

2. Programming an application that uses objects of an interface type is more flexible.

   - If a new, better implementation comes out, you can switch by changing one line of code.

Why separate interface from implementation?

```
// Create the list
List list = new ListImpl();

// Do lots of stuff with the list
list.add(new Student("Maurice", 16));
list.add(someOtherStudent);
...
Student s = (Student) list.get(15);
...
```

Why separate interface from implementation?

Substitute a different implementation.

```
// Create the list
List list = new ListImplImproved();

// Do lots of stuff with the list
list.add(new Student("Maurice", 16));
list.add(someOtherStudent);
...
Student s = (Student) list.get(15);
...
```

None of the remaining code has to change at all!

Why *not* an ADT?

- There are a few situations where you would *not* want to implement a data structure as an ADT.
  - Encapsulating a data structure into an ADT incurs a small amount of time cost and space cost.
  - In performance-critical programs (e.g., real-time systems, small-memory systems), this overhead might be a real problem.
- However, in the vast majority of programming scenarios, using data structures as ADTs is the right choice.

Implementing a List ADT.

List implementations

- Let's finally talk about how to implement a **`List`** with the three methods **`add`**, **`get`**, and **`remove`**.
- We will cover two kinds of list implementations:
  - **`ArrayList`**
  - **`LinkedList`**

Array lists

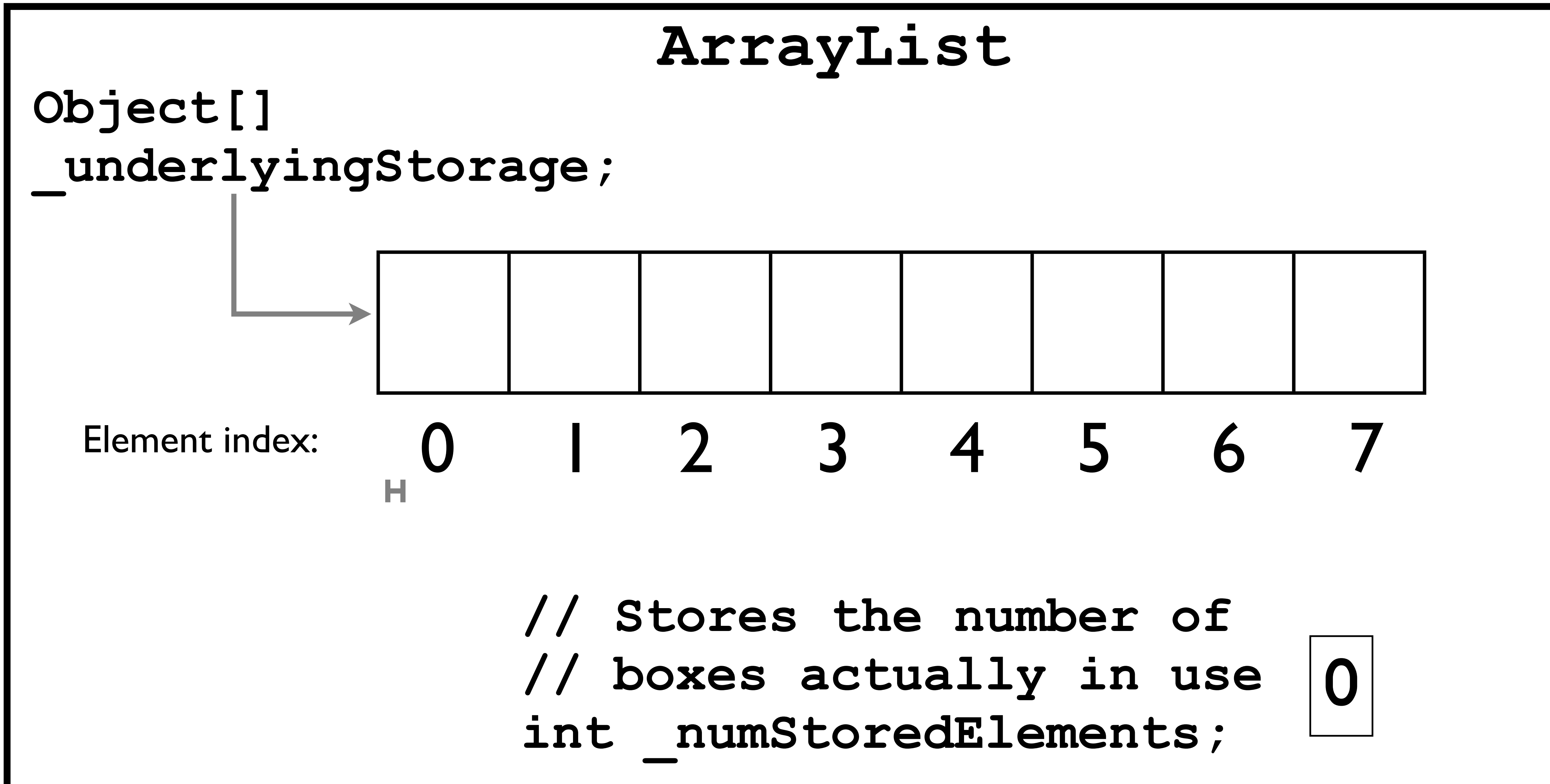
- Let's go back to our "sketch" of how to manage a list that could "grow" when more elements were added:
  - Start with a small array.
  - If it gets full, make the array larger.
  - Hide these details from the "user" -- the programmer using the **`ArrayList`** implementation -- behind the "wall of abstraction" provided by the **`List`** interface.

ArrayLists

- In our `ArrayList` ADT, we will store the data added by the `add(o)` method in an `Object[]`.
- This `Object[]` is the "underlying storage" of the ADT.
  - In 1960s parlance, this is called the "backing store" of the data structure.
  - What would be the "backing store" of the `OneLongBinarySequence` abstraction that the memory controller implements?

ArrayLists

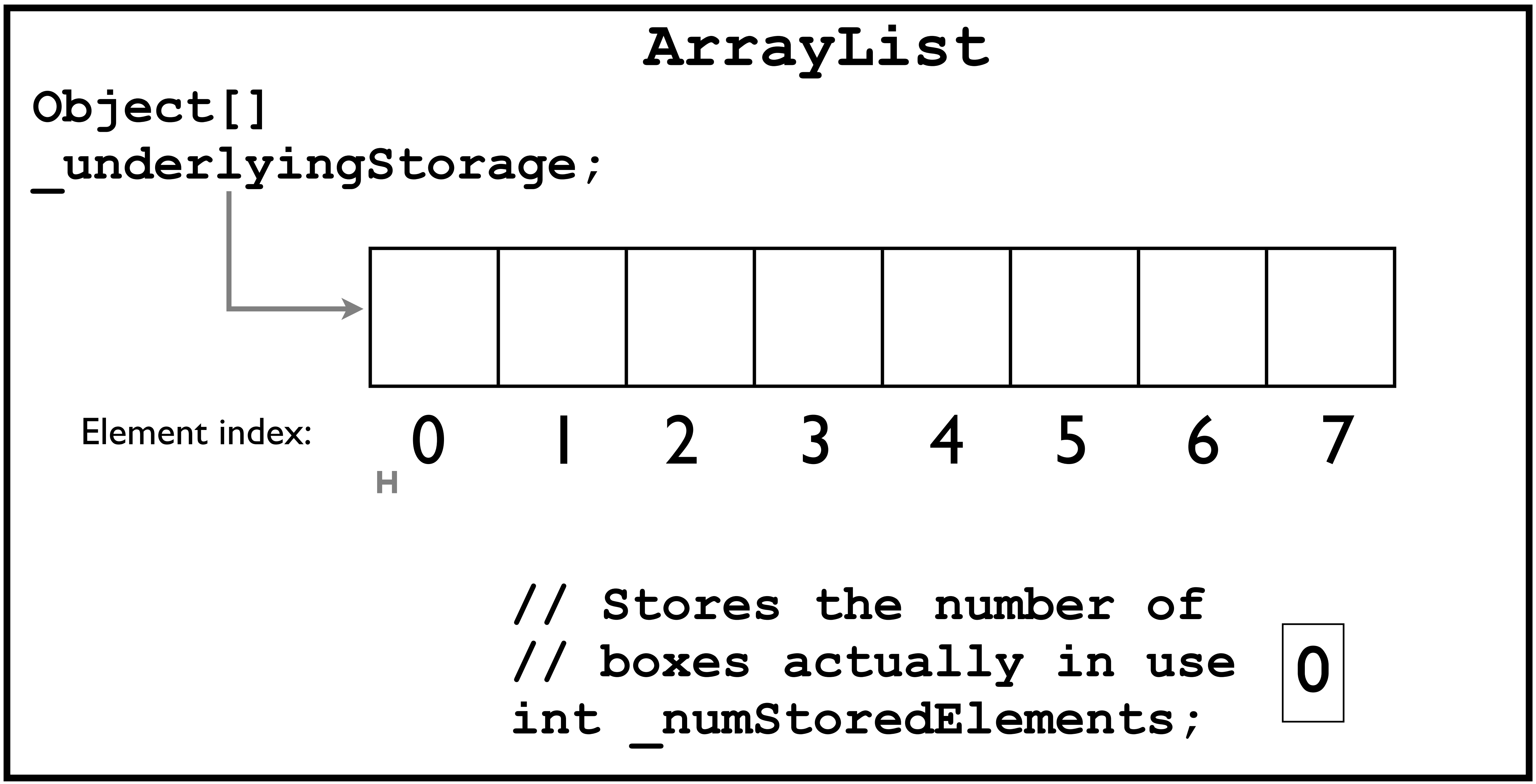
- It is often useful to depict ADTs graphically:



ArrayLists

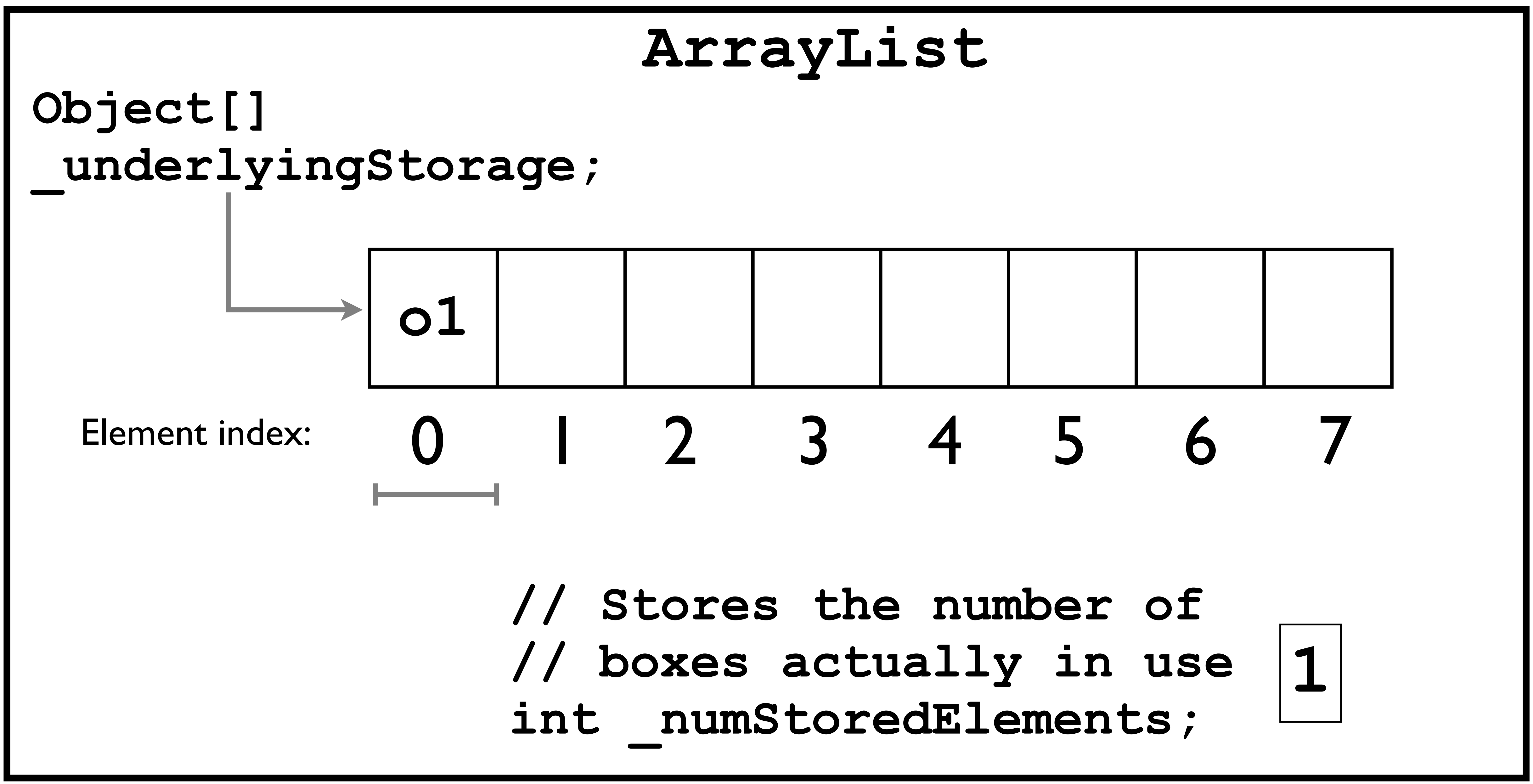
- Consider:

```
Object o1 = "Object1";
Object o2 = "Object2";
Object o3 = "Object3";
```



ArrayLists

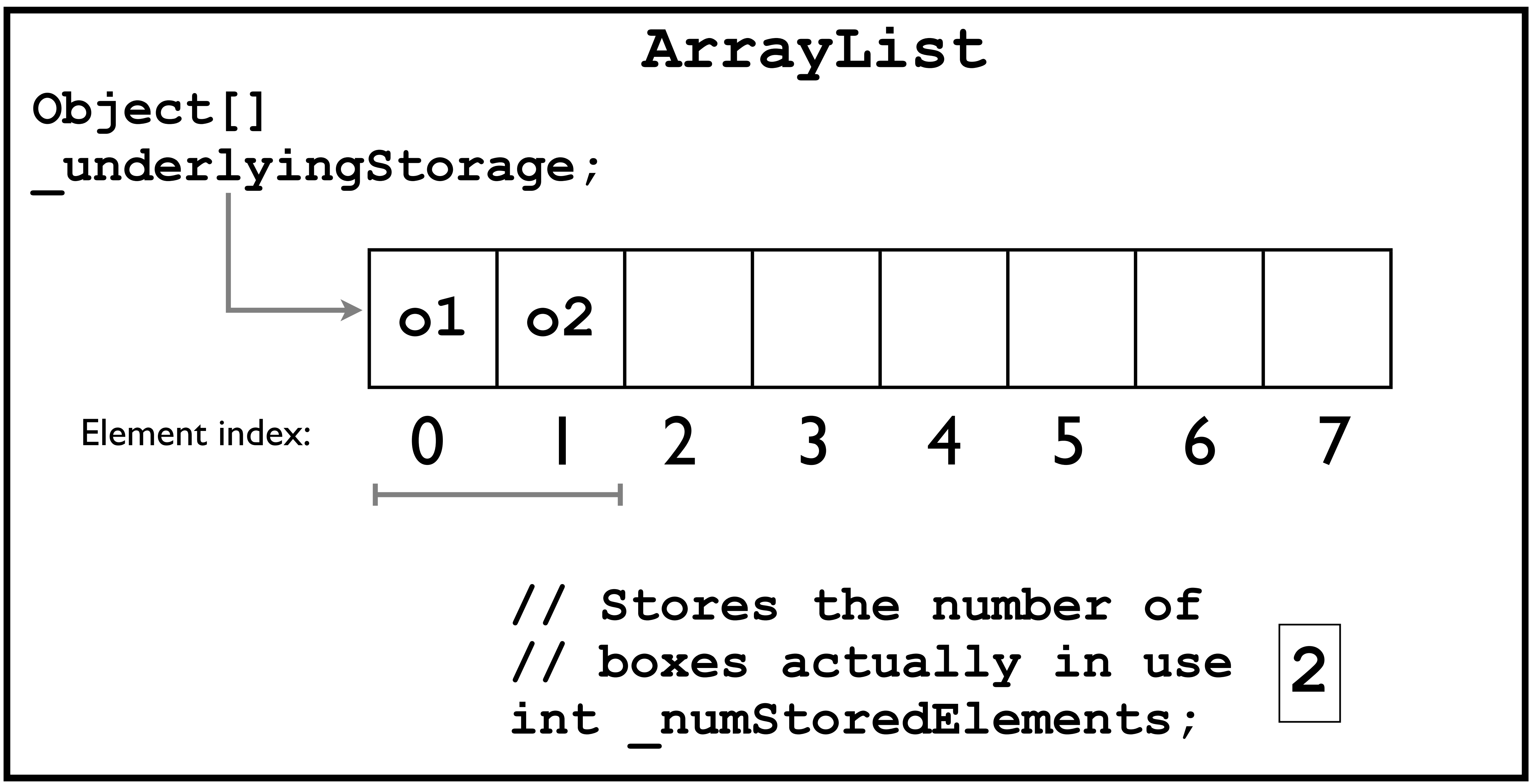
- Consider:
  `arrayList.add(o1);`



ArrayLists

- Consider:

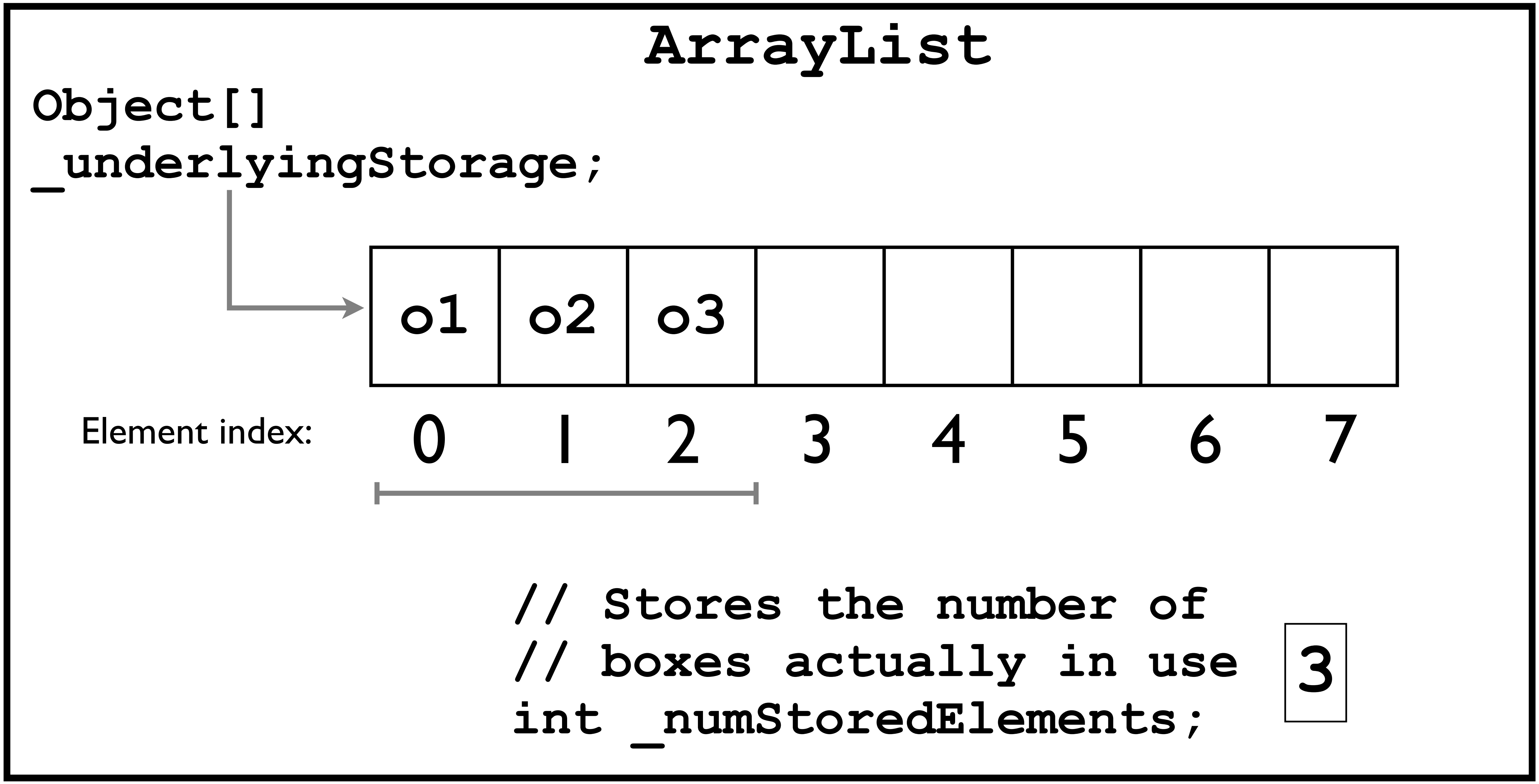
```
arrayList.add(o1);
arrayList.add(o2);
```



ArrayLists

- Consider:

```
arrayList.add(o1);
arrayList.add(o2);
arrayList.add(o3);
```



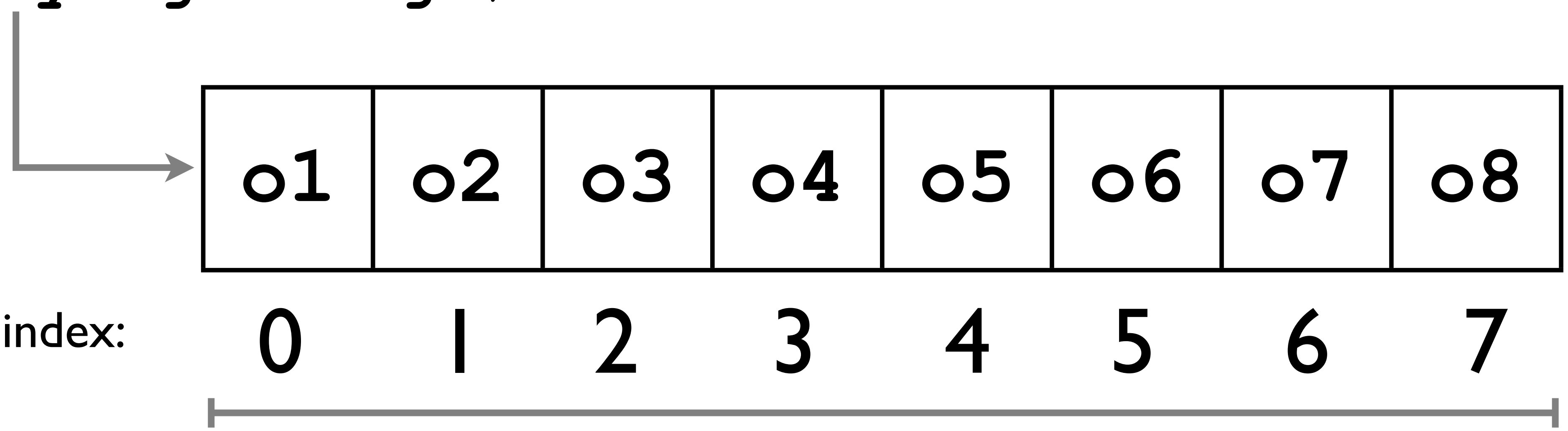
ArrayLists

- Consider:
  // More adds...

- After adding 8 objects to the list, the array is full. (How do we know?)
- If the user calls **add** again, we must enlarge the backing store.

```
Object[]
_underlyingStorage;
```

Element index:




```
// Stores the number of
// boxes actually in use
int _numStoredElements;
```

8

Enlarging an array

- First, what does it mean to "enlarge" an array?
  - In Java, once an array is allocated, its size cannot be changed:

```
Object[] array = new Object[8];
array.length++;  // this is nonsense
```

Enlarging an array

- Instead, we must allocate a *new*, larger array, and *copy* the old array data into the new array.

Enlarging an array

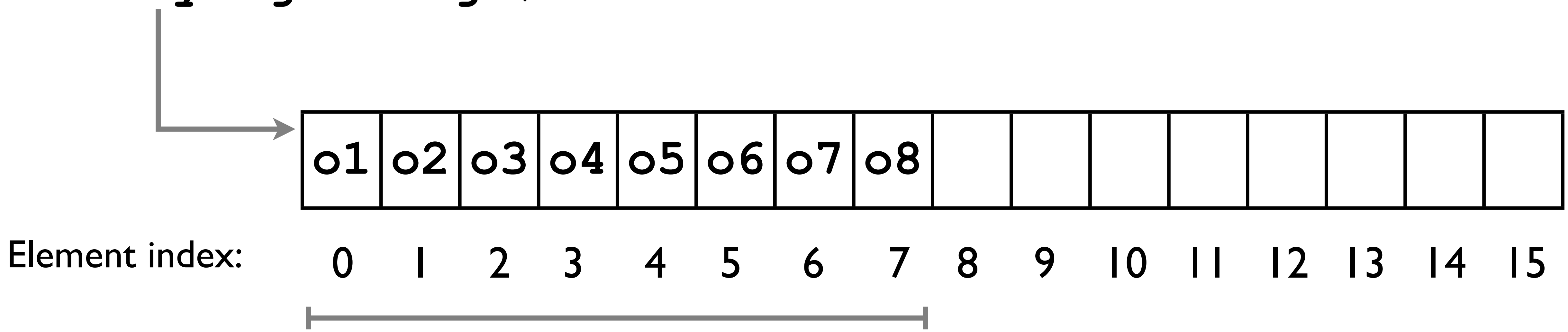
- Instead, we must allocate a *new*, larger array, and *copy* the old array data into the new array:

```
// Allocate initial array
Object[] array = new Object[8];

... // The array gets filled up

// Create a new, larger array
Object[] largerArray = new Object[16];
// Copy the array data into the new array
for (int i = 0; i < array.length; i++) {
  largerArray[i] = array[i];
}
// Replace the old array with the new one
array =  largerArray;
```

Enlarging the array

- After "enlarging" the array, we have:

```
Object[]
_underlyingStorage;
```




```
// Stores the number of
// boxes actually in use
int _numStoredElements;
```

Enlarging an array

- It would be a pain to do this in every application we write in which we need a flexibly-sized array.
- Implementing this "array resizing" in a List ADT once-and-for-all is more efficient and more reliable.

Enlarging the array: implementation issues

- When should we resize the array?
- How do we keep track of how full the current array is?
- By how much should we enlarge the array?

Unannounced quiz 1