

CSE 12:

Basic data structures and object-oriented design

Jacob Whitehill
jake@mplab.ucsd.edu

Lecture Three
3 Aug 2011

Review from Lecture 2

Abstract Data Types (ADTs)

- Some of the most important data structures in computer programming are **collections**.
- One of the most common collections is the **list**, which can be implemented (among other ways) as an **ArrayList** or a **LinkedList**.

Abstract data types (ADTs)

- Usually the **user** of the collection will not care about, and not want to be bothered with, the task of **implementing** the data structure.
- This yields a natural division of labor..

User versus implementor

List interface

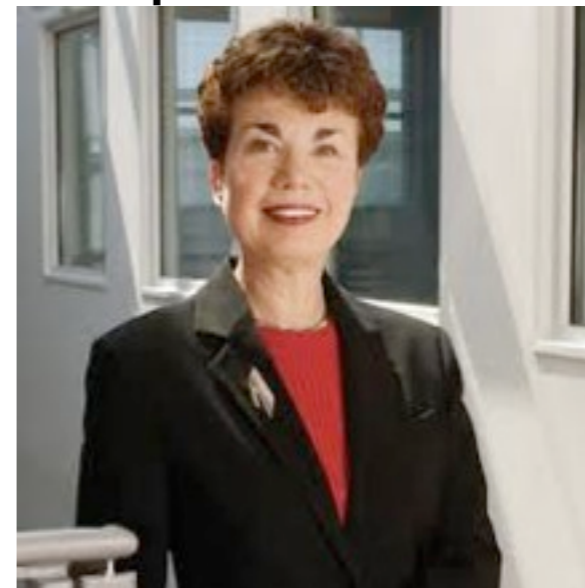
User



writes:

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

Implementor



writes:

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

Wall of abstraction

Photos courtesy of Google Image Search.

Interfaces

- The user and implementor must agree on the **interface** of the ADT.
- The interface is a collection of **method signatures**, which specify:
 - what methods the ADT provides.
 - what the methods do.
 - what parameters they take.
 - what they return
 - which exceptions they throw

Interfaces

- The interface may also specify:
 - *how* the user may call the methods, e.g.:
 - “method **A** () *must* be called before method **B** () ”

Interface as a contract

- The interface is a **contract** between the user and the implementor:
- It specifies which methods the implementor must write, and what they do.
- It specifies how the user may call them

Interfaces in Java

- In Java, interfaces are created with the `interface` keyword. E.g., in file `MyInterface.java`:

```
interface MyInterface {  
    void method1 (Object o);  
    ...  
    Integer[] method5 (int a, int b);  
}
```

Interfaces in Java

- Before an interface can be used, it must be implemented by a concrete class, e.g., in

MyImplementation.java:

Must explicitly write
“implements...”!

```
class MyImplementation implements MyInterface {
    Object _myObject;
    void method1 (Object o) {
        int a = 5;
        if (o == _myObject) {
            // blah blah
        }
    }
    ...
}
```

List interface

```
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;  
}
```

Interface implementations

- An interface may be implemented by multiple classes, e.g.:

```
List list = new ArrayList();  
...  
list = new LinkedList();
```

The Java Virtual Machine (JVM).

Why study the Java VM?

- Before studying the more complicated data structures, it is important to understand the relationship between a **computer program** and the **memory** on which it operates.
- In the case of Java, this entails a discussion of the Java Virtual Machine (JVM).

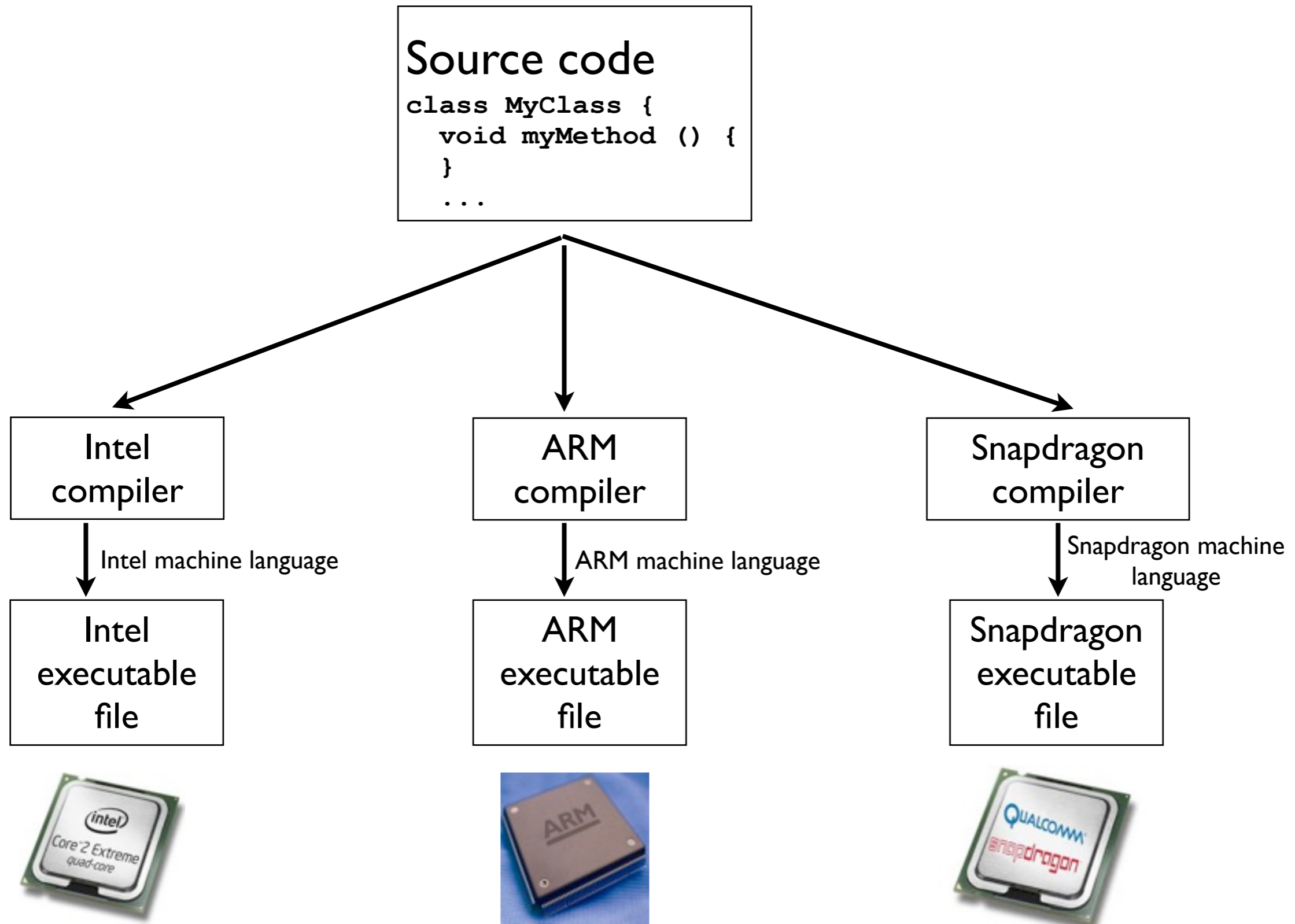
“Real machines”

- To appreciate “virtual machines”, let’s first consider how a source program is converted to an executable program on a “real machine”:
 - The programmer writes some **source code**.
 - He/she then **compiles** the source code into **machine instructions** that are specific to the particular hardware platform.

“Real machines”

- If the programmer wants her program to run on 5 different hardware platforms, she needs to:
 - Write the source code only once (thank goodness!).
 - Compile the source code 5 times.

“Real machines”



Java

- In the 1990s, Sun Microsystems developed a new programming language called “Java”.
- Motto: “Write once, run everywhere.”
- This might also be described as, “Compile once, run everywhere.”
- Once Java code is compiled, it can run on any platform, irrespective of CPU type.
 - How is this possible?

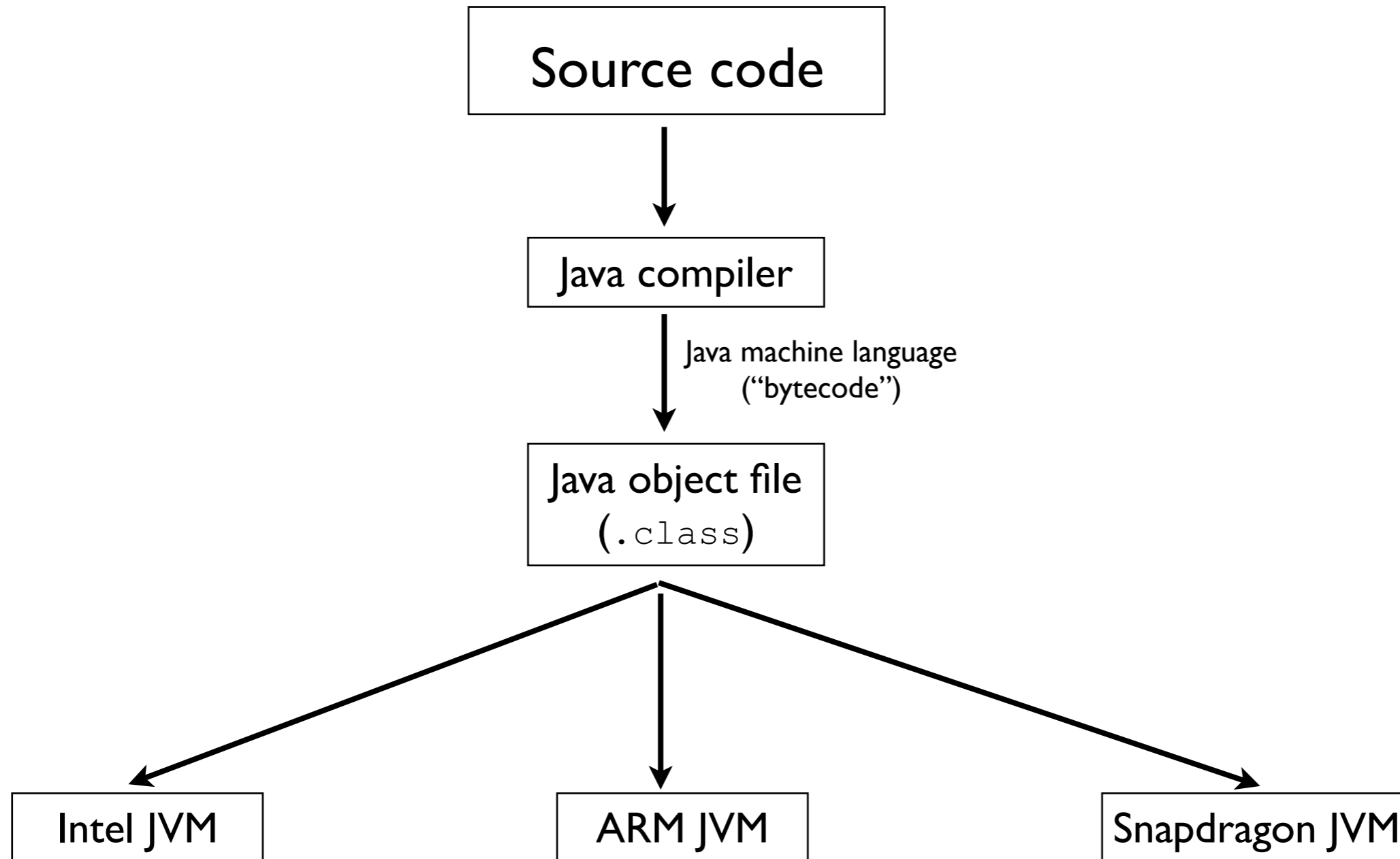
Java Virtual Machine

- The designers of Java inserted a layer of *abstraction* between the Java compiler and the hardware CPUs.
- This abstraction is called the Java Virtual Machine (JVM).
- The JVM provides a convenient “abstract machine language” that can run on *any CPU*.
- This means that a Java program need only be compiled *once*, and it can run on *any* hardware platform.

Java Virtual Machine

- The JVM was also designed from the ground-up to provide *security*, e.g.:
- Bounds checking: it won't let you access the 9th element of an 8-element array.
- Type safety: it won't let you treat an `Integer` as a `String`.

Java Virtual Machine



JVMs translate from bytecode into native machine code.



Java Virtual Machine

- *Every* compiled Java `.class` file will run on every Java Virtual Machine for every hardware platform in the *same way*.
- This *convenience through abstraction* comes at a price:
 - A new JVM must be created for every hardware platform.
 - The burden of **portable** code has shifted from software programmer to operating system designer + hardware manufacturer.

Java Virtual Machine

- The Java Virtual Machines are, themselves, software programs.
- The JVMs simulate a “Java CPU”:
 - They read the “bytecode” from the .class files, and then convert the *abstract* “Java instructions” to *real* CPU instructions.

Java Virtual Machine

```
x += 10;
```

Java source code

Java compiler

```
iinc 1, 10
```

Java machine language
(bytecode)

Intel JVM

JVM must convert from
bytecode to native machine
language in real time.

```
addl $10, -4(%rbp)
```

Intel machine language



Java Virtual Machine

- Every instruction of bytecode must be converted into a *real* instruction on the *actual* hardware CPU.
 - This incurs a cost -- Java is typically slower than C.
- The JVM plays an analogous role to the memory controller from last lecture -- it *implements an abstraction*.

Memory management in the JVM

Your data inside the JVM

- Compiled Java programs execute within the Java Virtual Machine.
- All Java programs need to store some **data**.
- Data are manipulated in Java using **variables**.
- It is helpful, when learning Java in general and data structures in particular, to understand *how the Java Virtual Machine manages these variables and data in memory.*

Variables in Java

- In Java, all variables are either of *primitive* or *reference* type.

- Primitive types:

`boolean, byte, char, short, int, long, float, and double.`

E.g., `float myHeight = 178.0; // cm`

- Reference types:

References to `Object`, subclasses of `Object`, and all *arrays*:

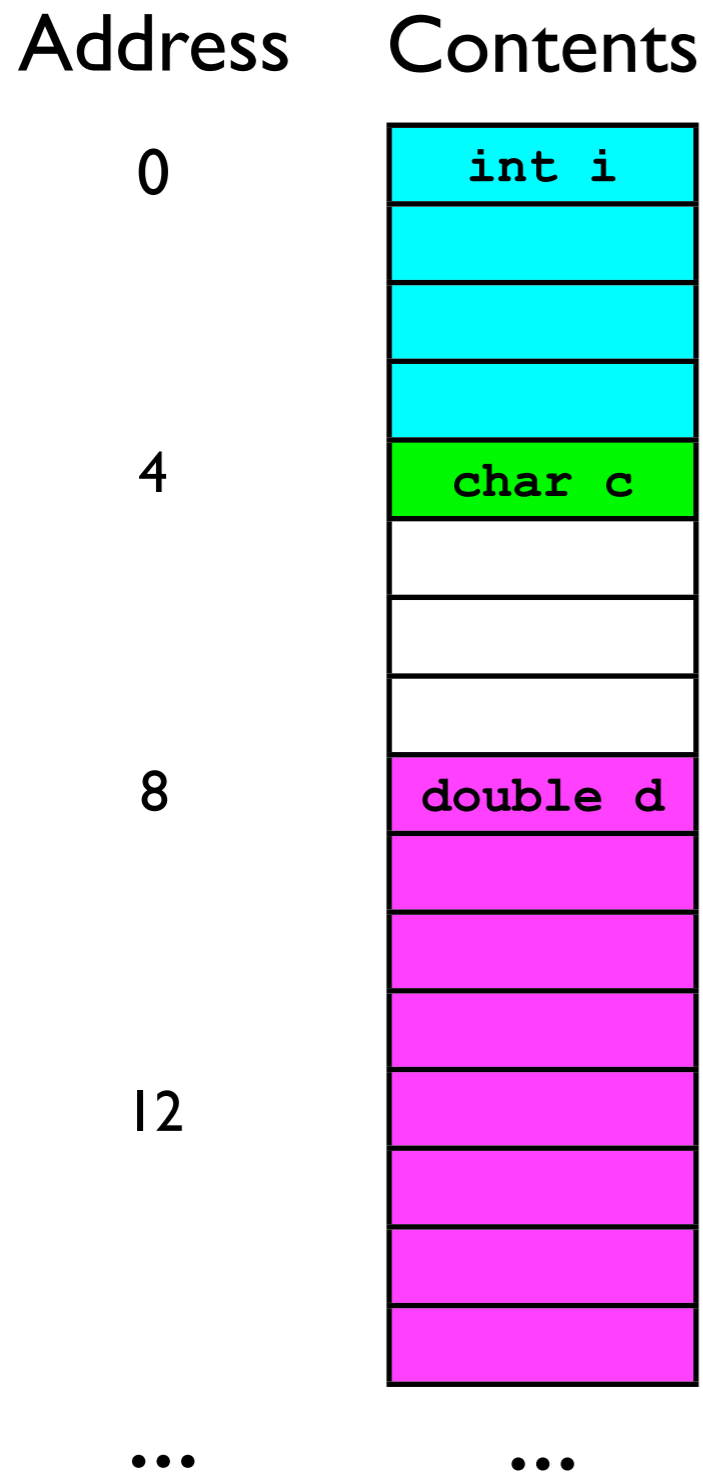
```
String s = "test";  
int[] arrayOfInts = new int[16];
```

Variables in Java

Address	Contents
0	11110110
	11001001
	01010001
	01011000
4	11101000
	11100000
	01000100
	11001110
8	01100101
	00101001
	01101111
	00010111
...	...

- The JVM's "system memory" can be viewed as a column of bytes, each with its own address.
- All data (`int`, `float`, `Object`, etc.) are stored somewhere in this memory column.
- The location of each variable/object is called its *address*.

Variables in JVM Memory

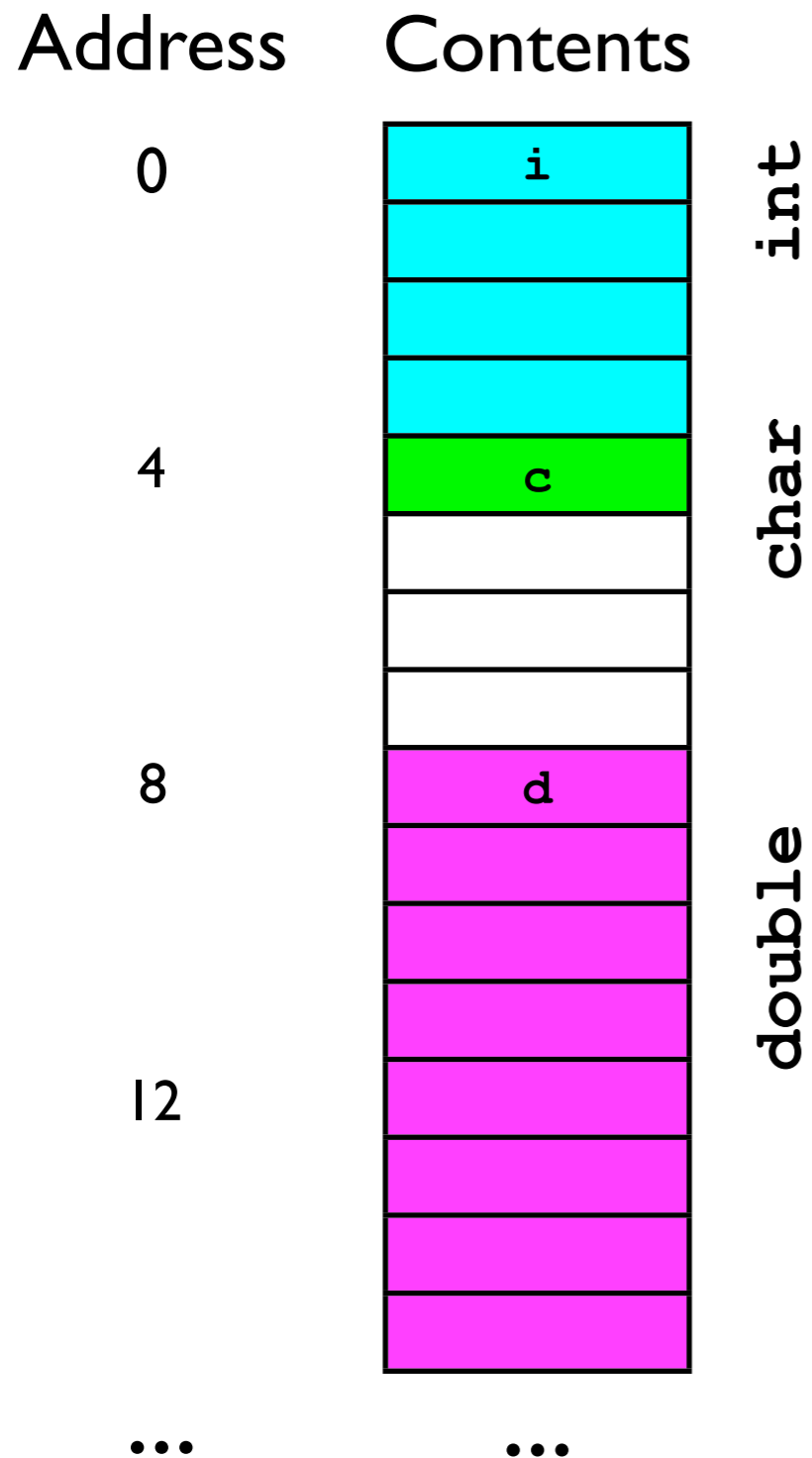


- Consider a method that declares three variables of primitive type:

```
void myMethod () {  
    int i;  
    char c;  
    double d;  
}
```

- These might be stored in memory as shown to the left.

Variables in JVM Memory



- Different variables require different numbers of bytes for storage:

byte, char: 1

short: 2

int, float: 4

long, double: 8

These numbers would correspond to a 32-bit JVM.

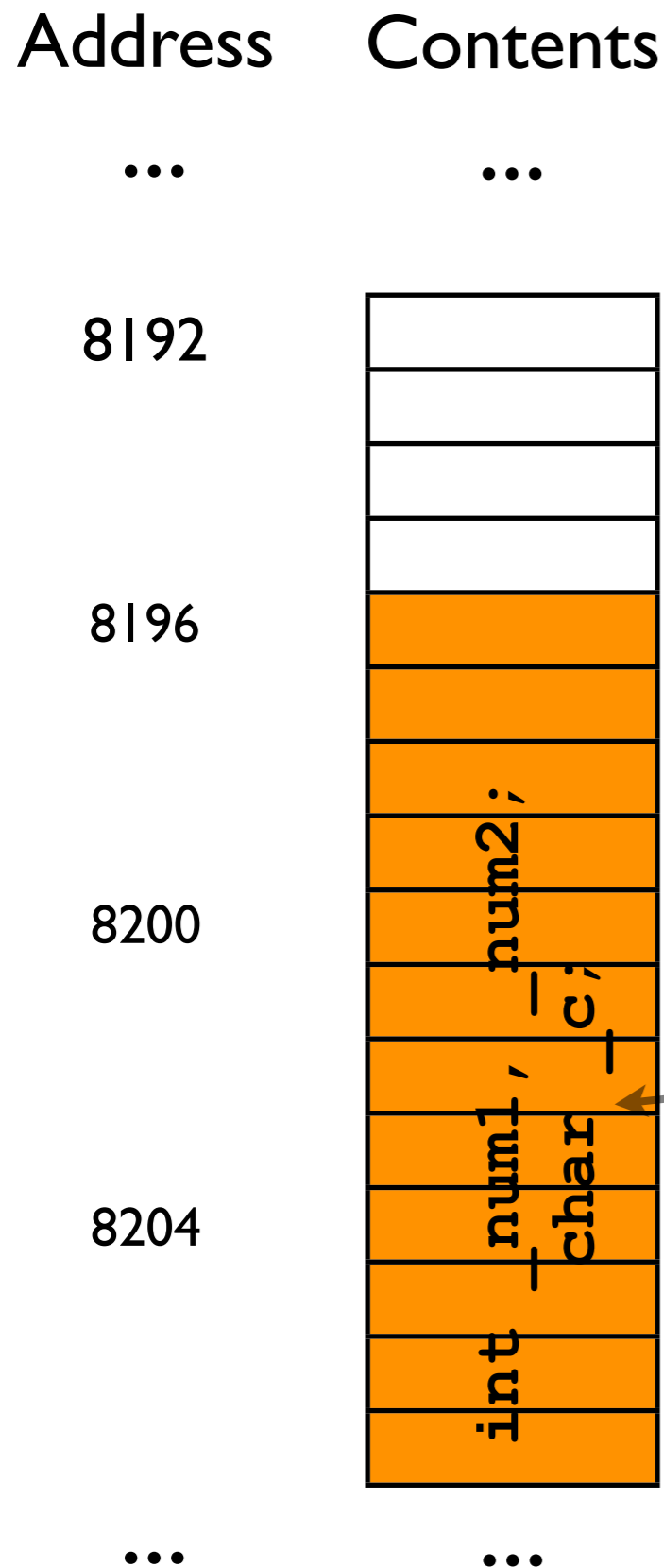
Objects in JVM Memory

- Objects also (of course) take up memory.
- How much memory they need depends on the **instance variables** stored in them.

```
class MyObject {  
    int _num1, _num2;  
    char _c;  
}
```

- `2 ints + 1 char = 9 bytes` (The JVM probably rounds up to nearest multiple of 4.)
- There is also some amount of overhead (even if your class defines no instance variables).

Variables in JVM Memory



- An object of type `MyObject` may require about 12 bytes.
- The exact amount depends on the particular JVM.
- Somewhere within those bytes are `_num1`, `_num2`, and `_c`.
- The exact location depends on the particular JVM.

References

- Consider the following code:
`MyObject obj = new MyObject();`
- This code actually refers to **two** “things” in memory:
 - The newly created *object* of type `MyObject` (about 12 bytes).
 - The *reference* `obj` to that new object (just 4 bytes -- enough to store a memory address on a 32-bit machine).
- **A *reference* in Java stores the location (address) in memory of an object or array.**

References

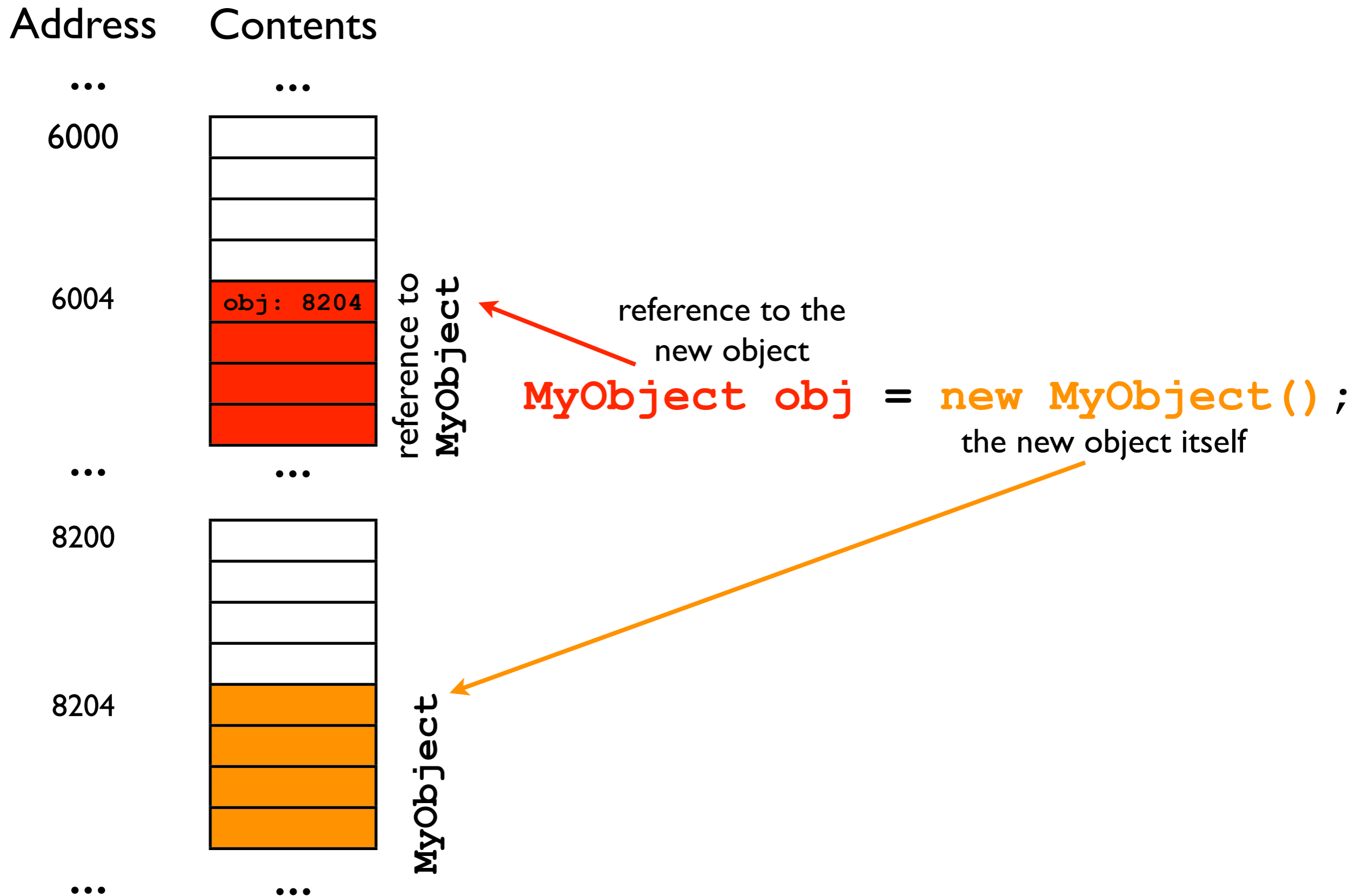
- The call to `new MyObject()` causes the Java Virtual Machine to instantiate a new object of type `MyObject`.
- Memory is *allocated* (“set aside”) for the new object.
- The result of the “new” call is the *address* of the newly created object.
- In `MyObject obj = new MyObject()`, this address is then stored in the `MyObject`-reference `obj`.

References

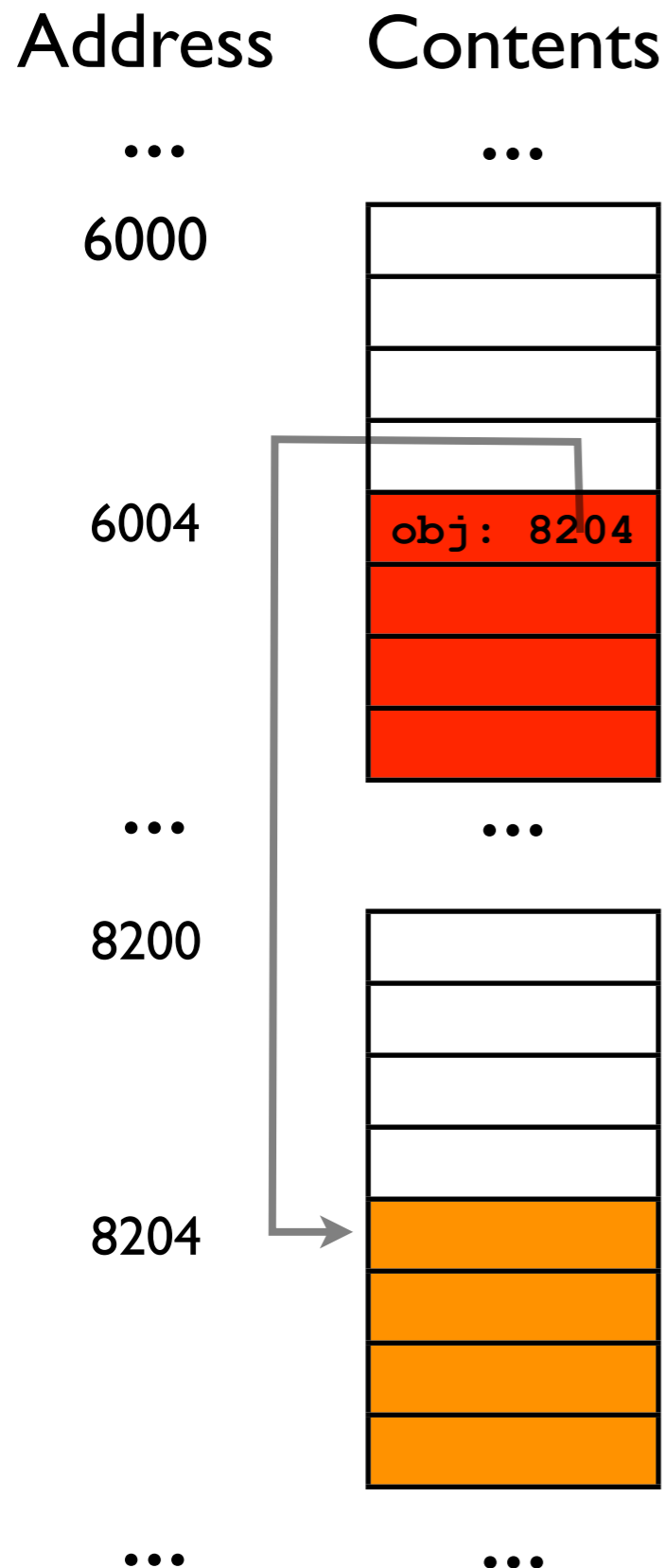
- If you “forget” to store the address returned by `new MyObject()`, then the newly created object will be essentially “forgotten” -- there’s no way to know where in memory it is stored.
- Example:

```
new Object(); // Creates a new object, then
               // promptly forgets where it is.
```
- (Eventually, the garbage collector will remove it.)

Objects, and references to objects



Objects, and references to objects



- In effect, the reference “points to” the newly created object by storing its address.
- This is why references are sometimes called **pointers**.

reference to the new object

```
MyObject obj = new MyObject();
```

the new object itself

JAKE -- show PrintAddr demo

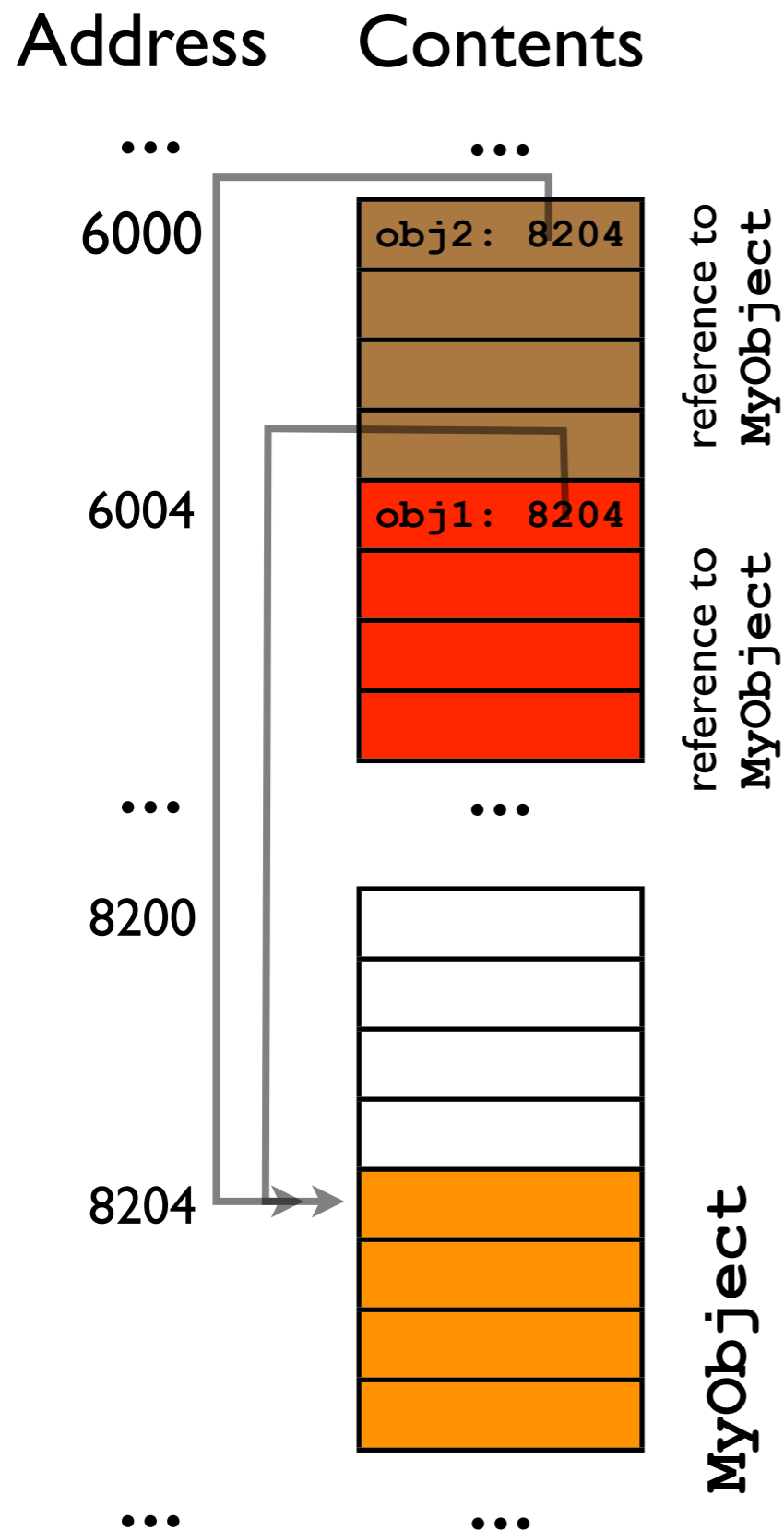
Multiple references to same object

- Consider:

```
MyObject obj1 = new MyObject();  
MyObject obj2 = obj1;
```

- Object-references `obj1` and `obj2` now point to the *same*, newly created object.
- If you modify `obj1._num1`, this will also affect `obj2._num1`

Multiple references to same object



```
MyObject obj1 = new MyObject();  
MyObject obj2 = obj1;
```

- All you're doing in line 2 is setting the *address* stored in variable `obj2` to the same *address* stored in variable `obj1`.
- This causes `obj2` to “point to” the same object that `obj1` points to.

Multiple references to same object

- Example:

```
MyObject obj1 = new MyObject();  
MyObject obj2 = obj1;  
obj1._num1 = 5;  
obj2._num1 = 6;  
System.out.println(obj1._num1);  
// Prints out 6 !
```

== versus equals()

- The discussion of references brings up the issue of “equality” between objects.
- `String s1 = new String("test");`
`String s2 = new String("test");`

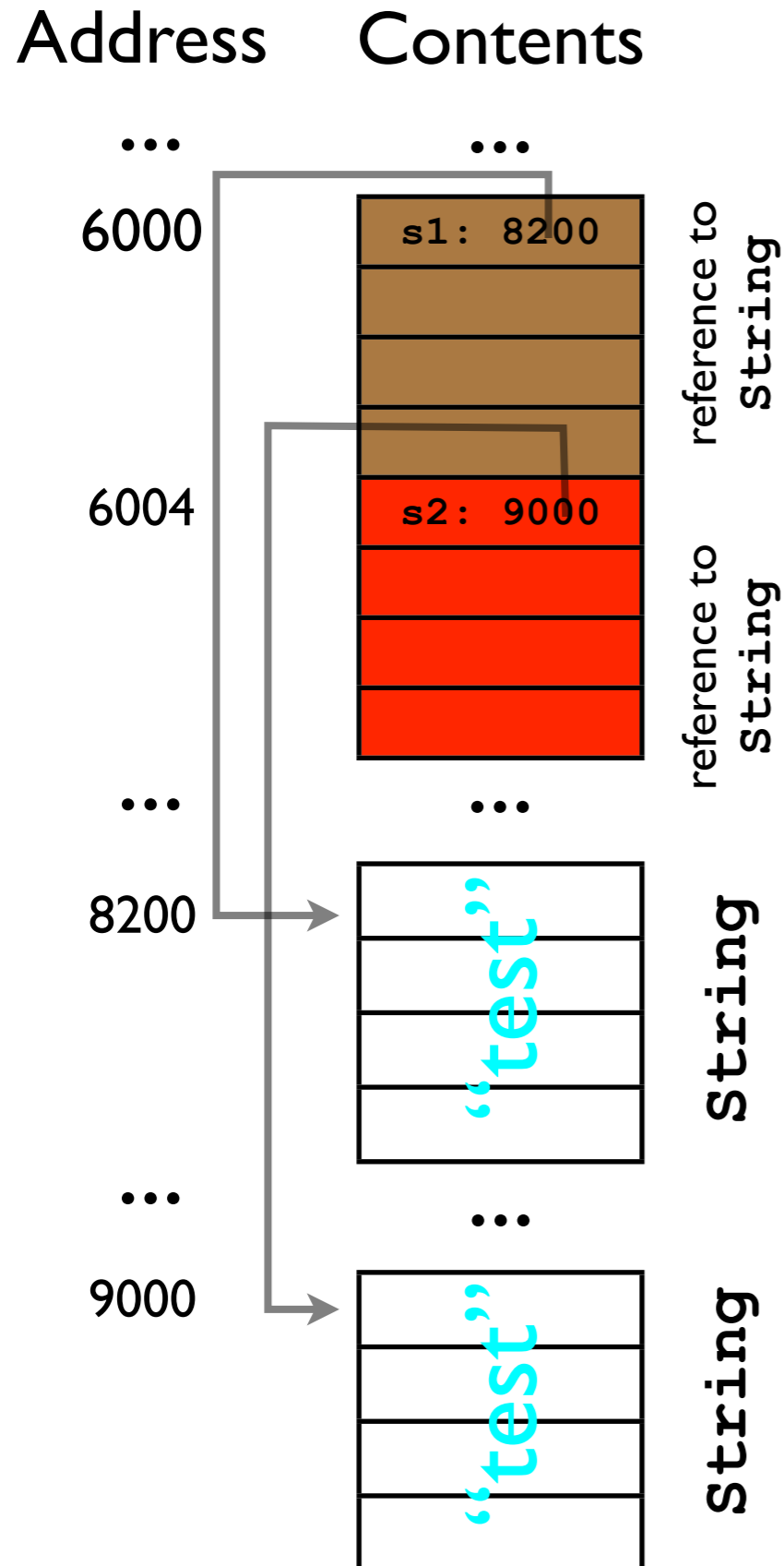
```
s1.equals(s2)  
s1 == s2
```

== versus equals()

- The discussion of references brings up the issue of “equality” between objects.
- ```
String s1 = new String("test");
String s2 = new String("test");

s1.equals(s2) // true
s1 == s2 // false! -- why?
```
- `s1.equals(s2)` compares the *contents* of the objects pointed to by `s1` and `s2`.
- `s1 == s2` compares the *addresses* stored in reference-variables `s1` and `s2`!

# == versus equals()



- `s1` and `s2` point to two different `String` objects.
- The *contents* of the two `String`s happens to be the same.

# Dereferencing operator

- You're already well-familiar with the dereferencing operator `.` (dot).

```
MyObject obj = new MyObject();
```

```
obj._num1 = 3;
```

What does this really mean  
in terms of memory?

Left side: valid (non-`null`) reference

Right side: name of  
instance/class variable.

# Dereferencing operator

Address      Contents

...                  ...

8192                  obj: 8196

8196

8200                  \_num1: 1

8204

...                  ...

reference to  
MyObject

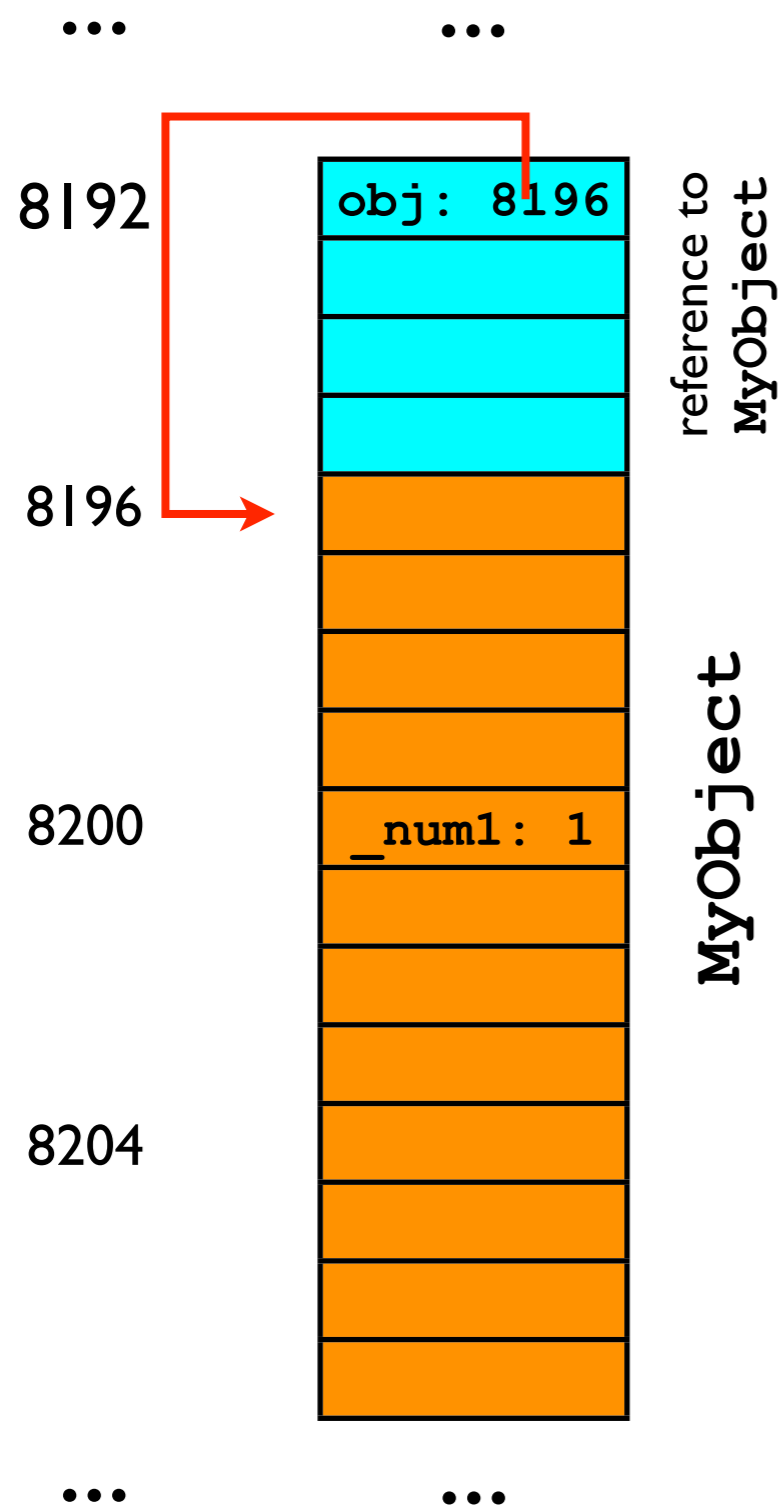
MyObject

```
obj._num1 = 3;
```

- Step 1: read the address stored in obj (8196).

# Dereferencing operator

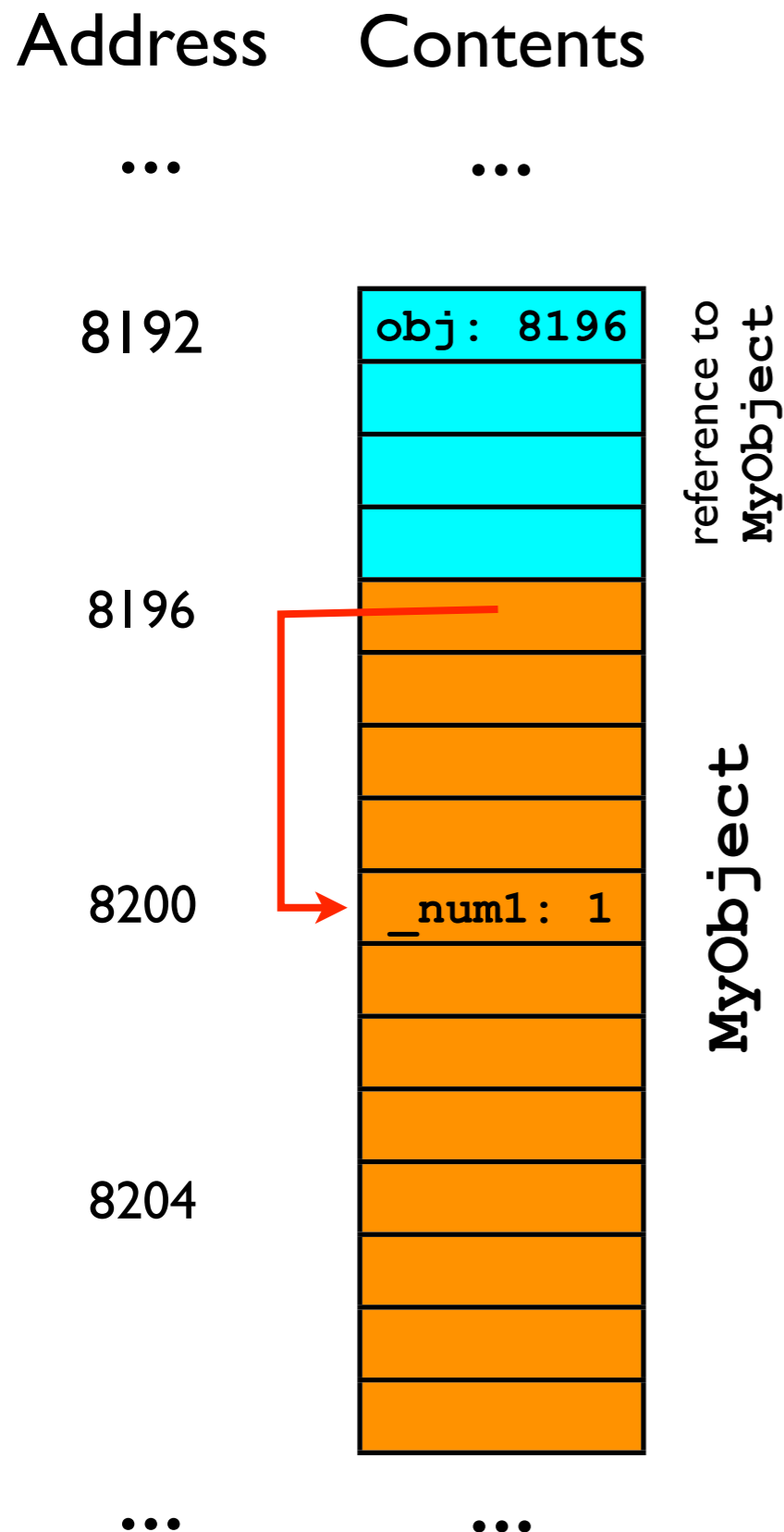
Address    Contents



```
obj._num1 = 3;
```

- Step 1: fetch the address stored in obj (8196).
- Step 2: dereference (“go to”) that address.

# Dereferencing operator



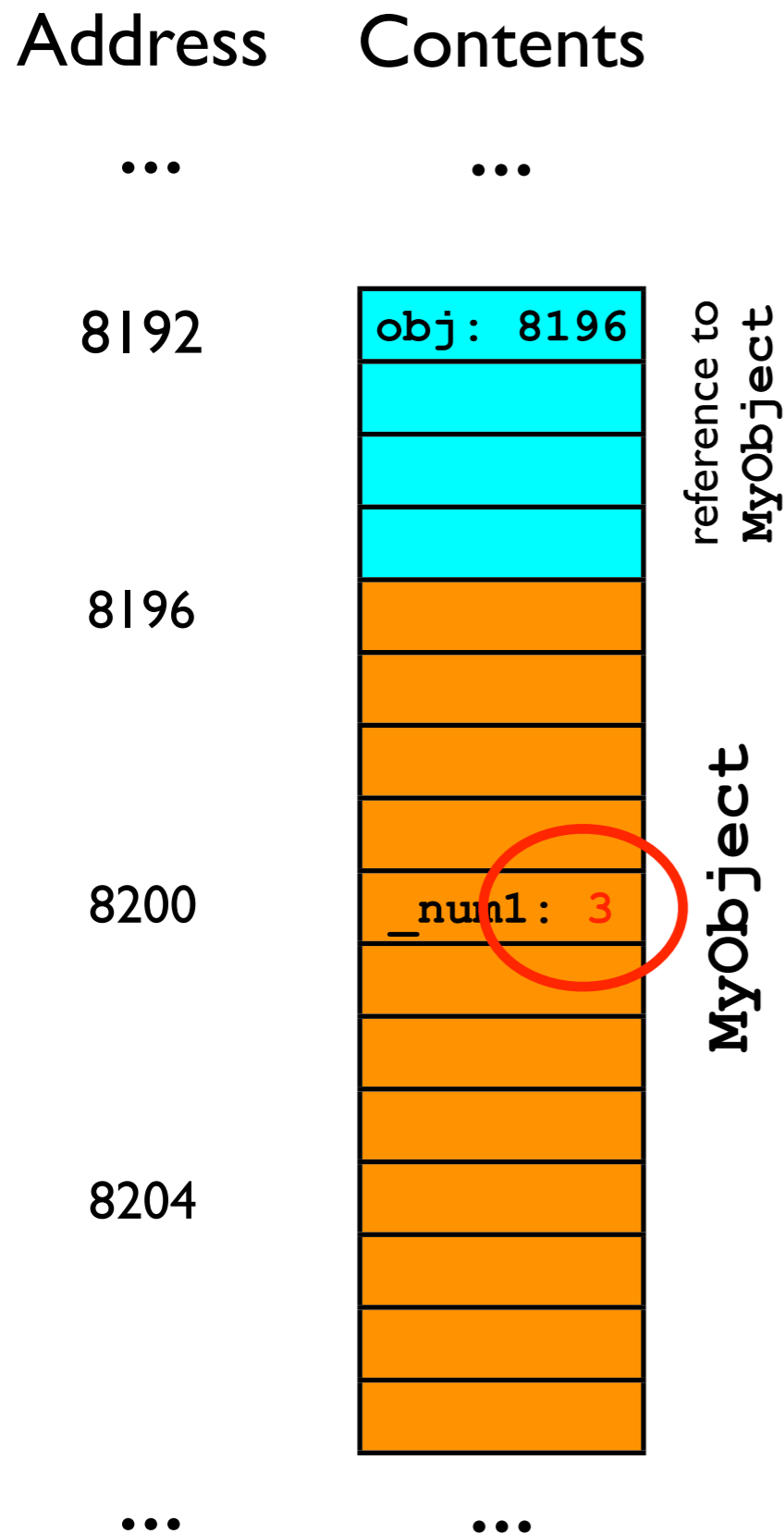
```
obj._num1 = 3;
```

- Step 1: fetch the address stored in obj (8196).
- Step 2: dereference (“go to”) that address.
- Step 3: find where in that MyObject the instance variable \_num1 is stored.

(This is hidden from the programmer, but the JVM knows where it is.)



# Dereferencing operator



```
obj._num1 = 3;
```

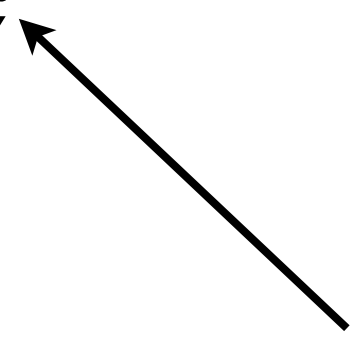
- Step 1: fetch the address stored in obj (8196).
- Step 2: dereference (“go to”) that address.
- Step 3: find where in that MyObject the instance variable \_num1 is stored.
- Step 4: set its value to 3.

# References inside objects

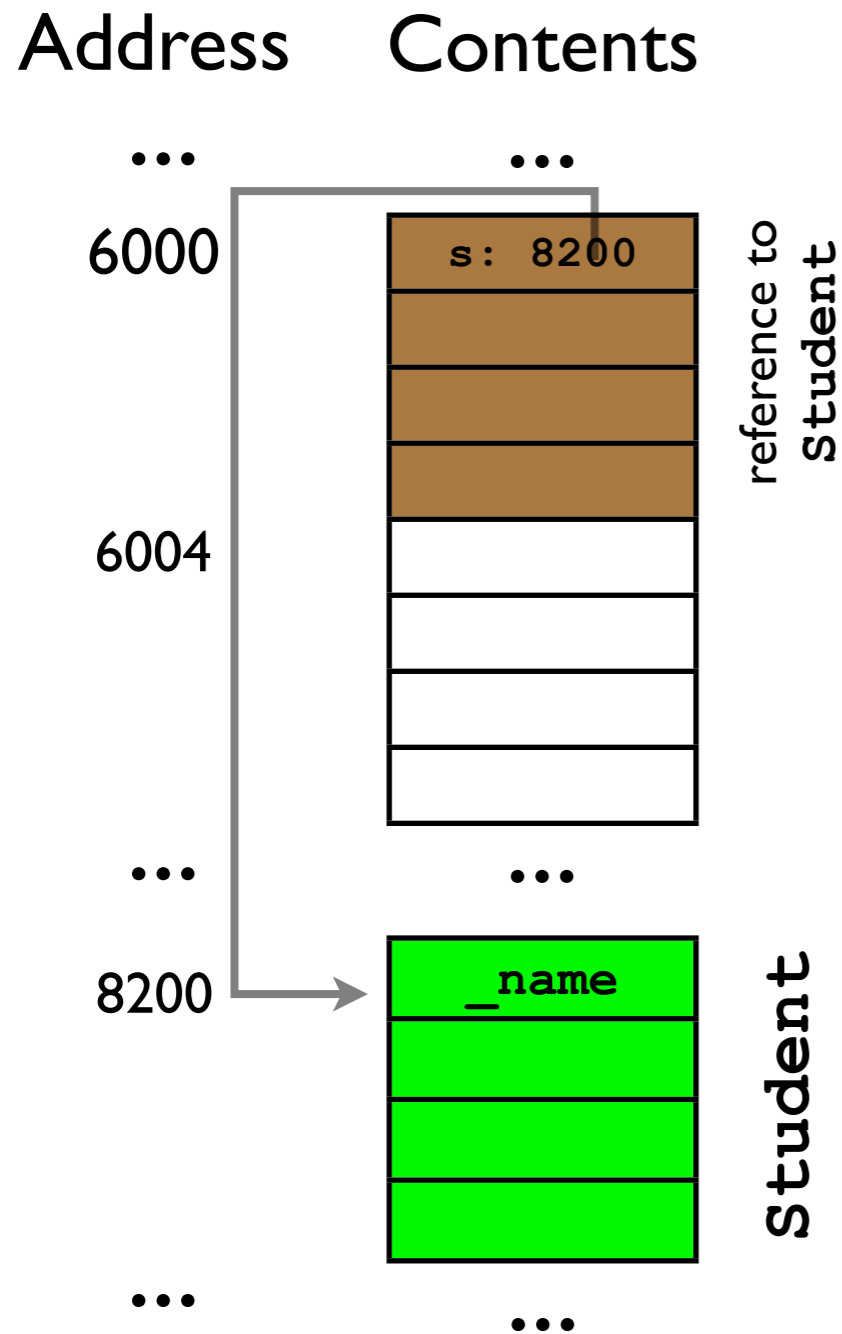
- It is commonplace for objects to contain instance variables that are references to other objects.

```
class Student {
 String _name;
 int _age;
}
```

The `_name` instance variable of a `Student` object is a reference to a `String` object.

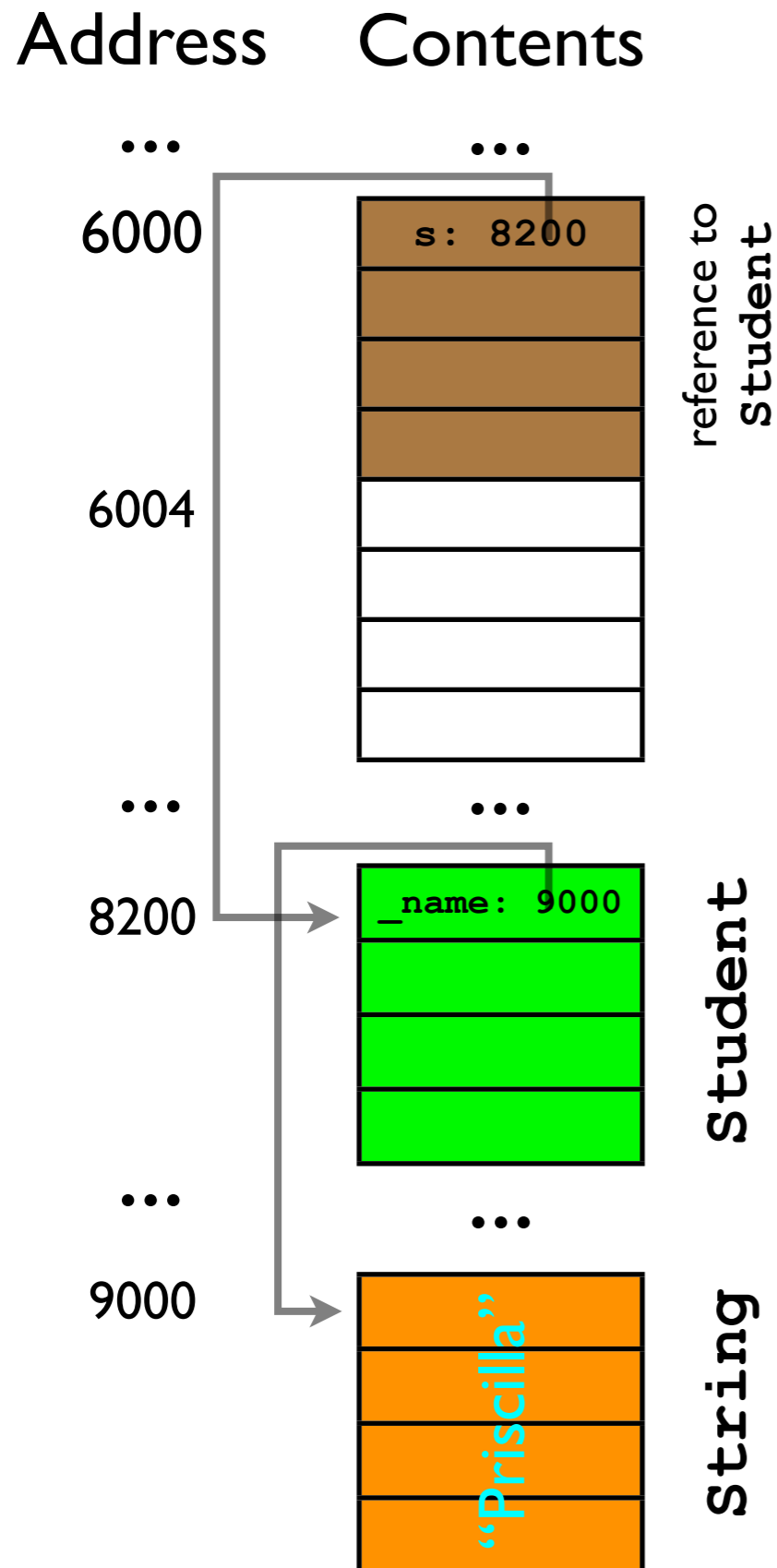


# References inside objects



```
Student s = new Student();
```

# References inside objects

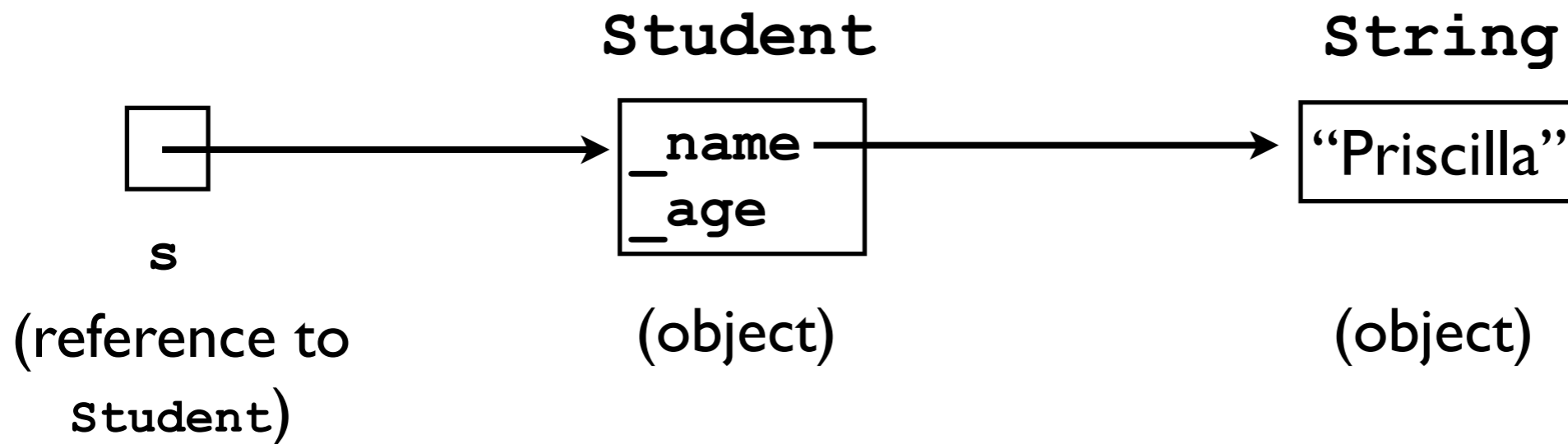


```
Student s = new Student();
s._name = "Priscilla";
```

# Simplified memory diagrams

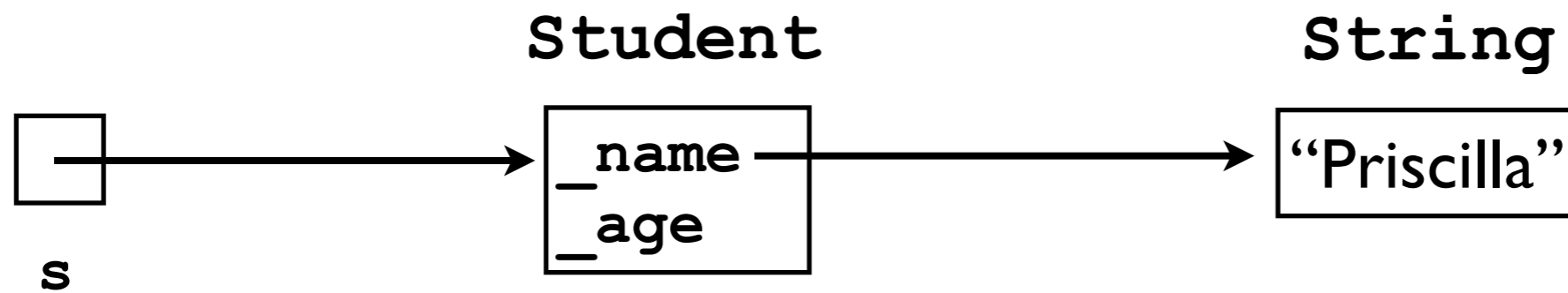
- We've seen these columnar memory diagrams a lot now.
- Let's take a more "abstract" perspective on memory and not worry about the particular memory addresses as much.
- Let's also move the objects out of the memory column to illustrate their relationships better.
- In reality, of course, the objects are all stored somewhere on some memory DIMM (chip) as a sequence of 1's and 0's...
  - Yada yada yada...

# Simplified figure for `class Student`

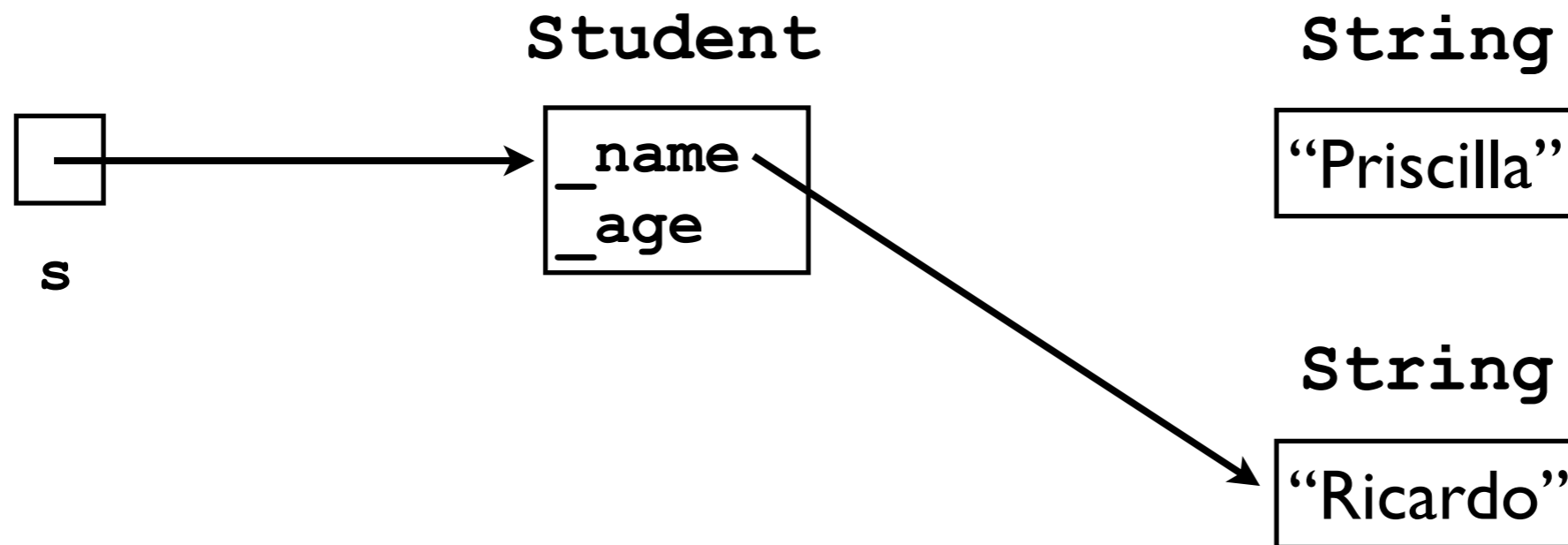


**Inside the boxes:** Sometimes I will write the *names* of instance variables and sometimes their *values*; it should be clear from the context.

# Even simpler figure for `class Student`



# Changing `s._name`



```
s._name = "Ricardo";
```



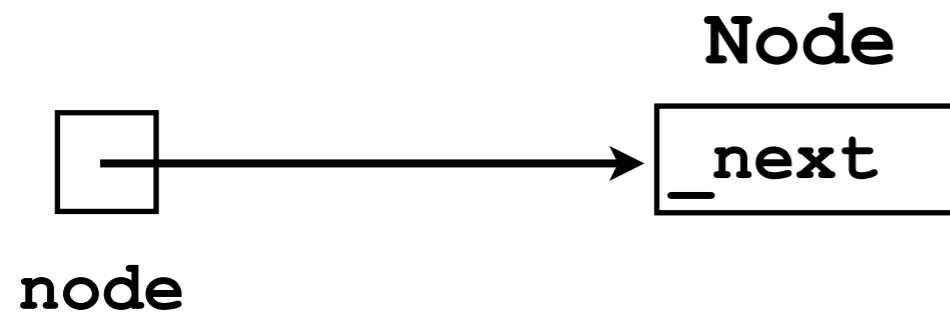
# Here's where things get fun...

- It is also (sometimes) useful for an object to contain a reference to *another* object of the *same class*.
- In this way, we can “chain” together multiple objects.
- Example:

```
class Node {
 Node _next;
}
```

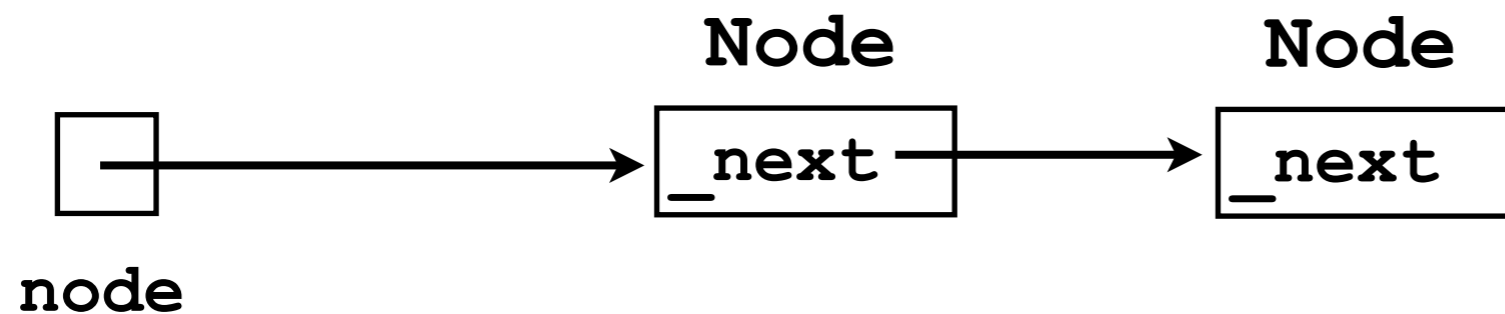
# Chain of Nodes

```
Node node = new Node();
```



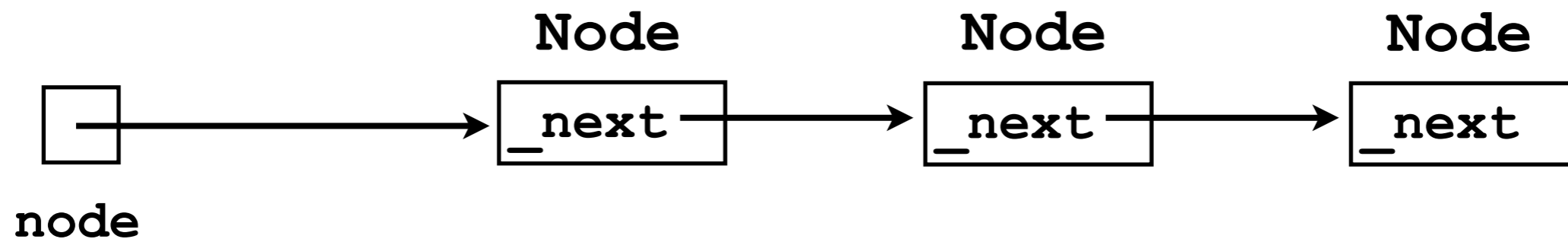
# Chain of Nodes

```
Node node = new Node();
node._next = new Node();
```



# Chain of Nodes

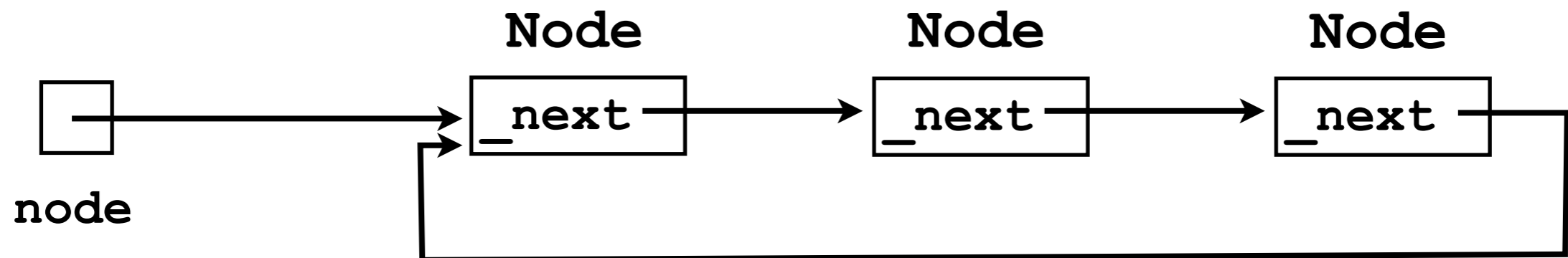
```
Node node = new Node();
node._next = new Node();
node._next._next = new Node();
```



# Loop of Nodes

We can even create a “loop”:

```
node._next._next._next = node;
```



# Node chains and loops

- Why would we want to build these bizarre structures?
- They are sometimes useful in implementing ADTs.

# Linked lists.

# ArrayLists

- Recall from last lecture that we discussed how to implement a reasonable `List` interface using an array.
- We called this implementation an `ArrayList`.
- The `ArrayList.add(o)` method would automatically resize its internal `_underlyingStorage` array whenever it got full.
- This is more convenient for the user than having to manage the array him/herself.

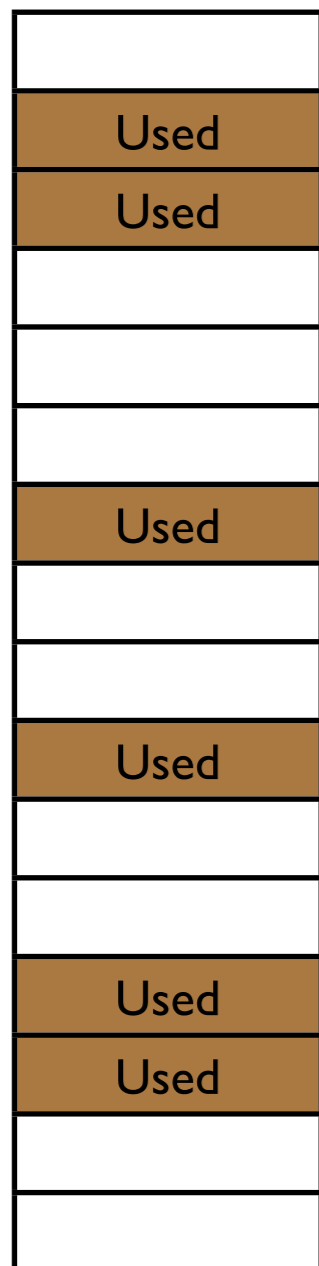


# Problems with `ArrayLists`

- However, the `ArrayList` is unsatisfying in a few ways:
  - It is still wasteful in memory -- after doubling the size of the `_underlyingStorage`, about half of the array elements are unused.
  - It does not solve the *contiguity problem*.

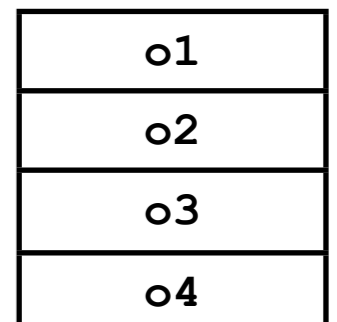
# Contiguity problem

Memory



- Sometimes the pool of free memory can become “fragmented” -- split into small chunks.
- In this case, it may not be possible to allocate one large, contiguous array.

? ← I can't store myself anywhere!



Total free memory: 10 slots

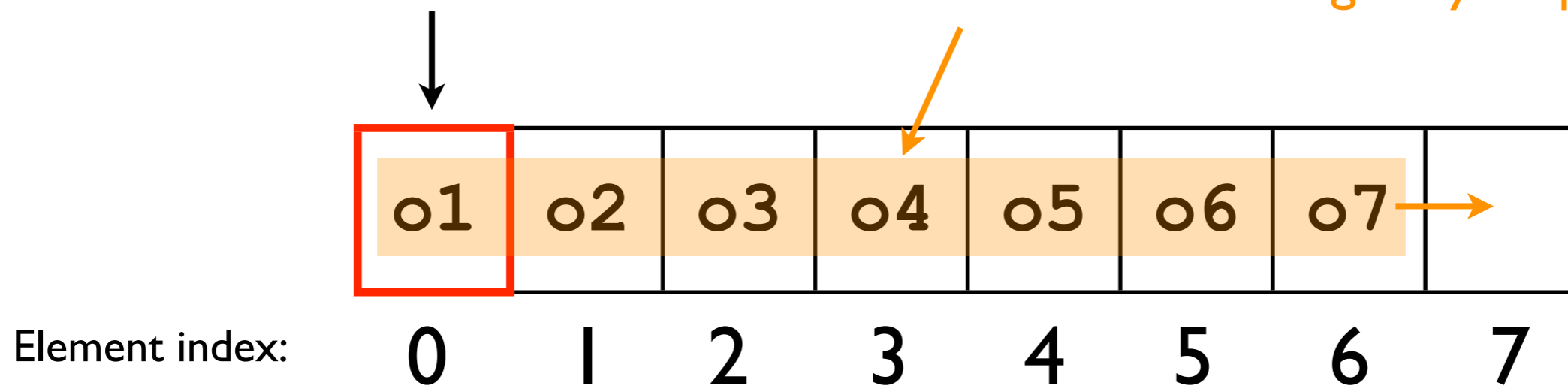
Maximum contiguous memory: 3 slots

# Problems with ArrayLists

- Another disadvantage of `ArrayLists` arises when you want to add an object to the *front* of the list: `arrayList.addToFront(o8);`

Where to insert o8

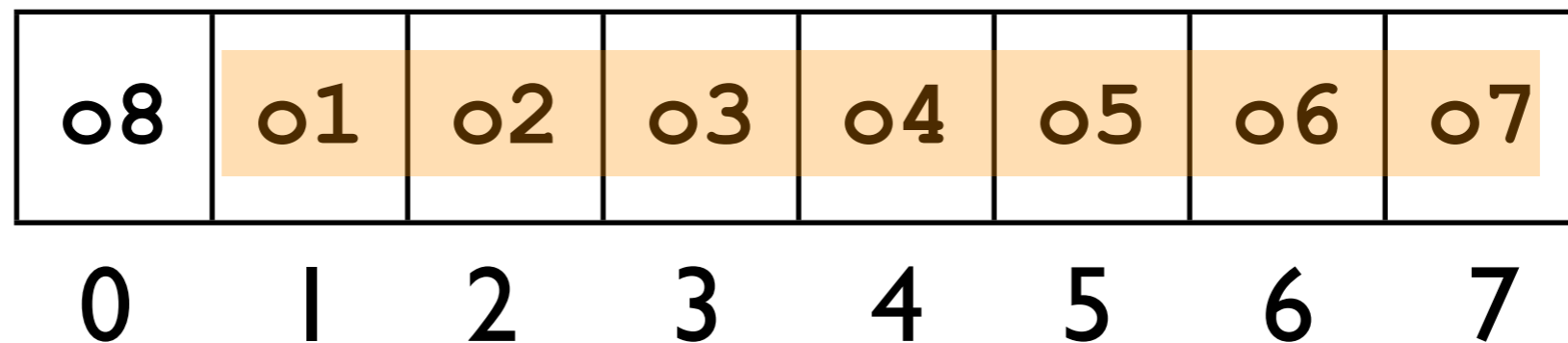
Move all these elements right by 1 space.



- We have to move all the other elements first!

# Problems with ArrayLists

- Another disadvantage of `ArrayLists` arises when you want to add an object to the *front* of the list: `arrayList.addToFront(o8) ;`



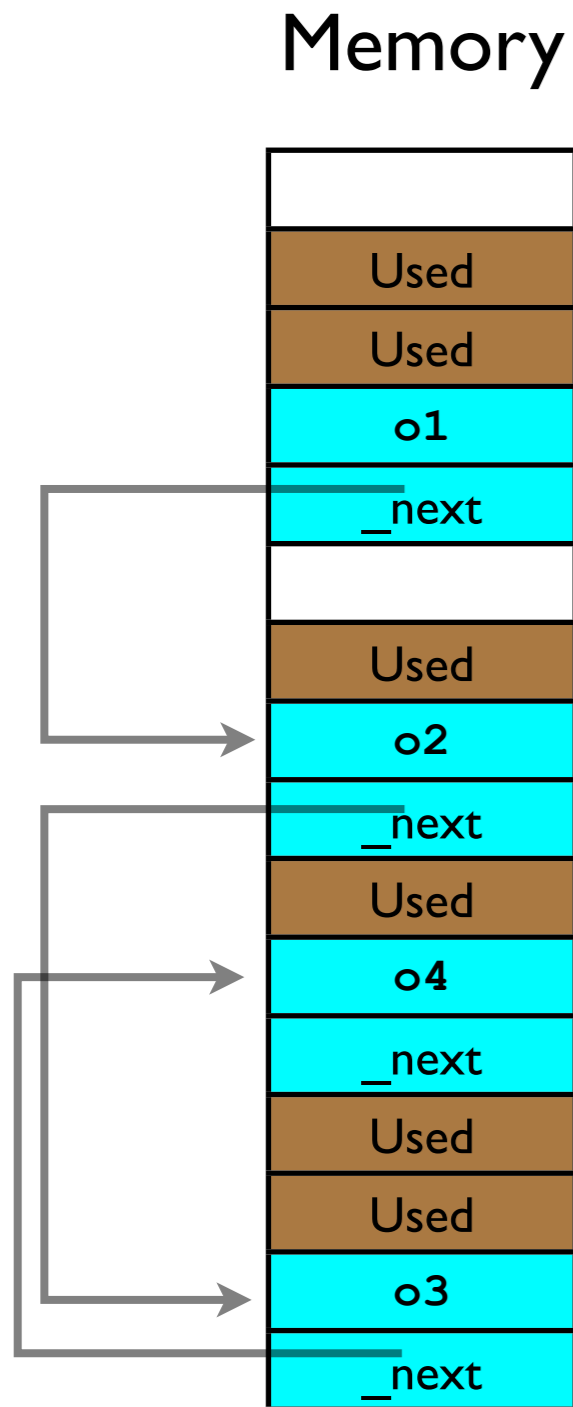
Element index:

- We have to move all the other elements first!
- This is expensive!

# Linked lists

- Linked lists provide a convenient ADT for storing ordered data.
- Linked lists store exactly as many elements as are needed -- no “wasted space”.
- They can be easily resized.
- Linked lists do not suffer from the “contiguity problem”.

# Contiguity problem



- Linked lists can be stored **non-contiguously** in memory by “chaining” **nodes** together.

Total free memory: 10 slots

Maximum contiguous memory: 3 slots

# Linked lists

- Let's conceptualize a linked list by considering one of the fundamental operations of the `LinkedList` ADT:

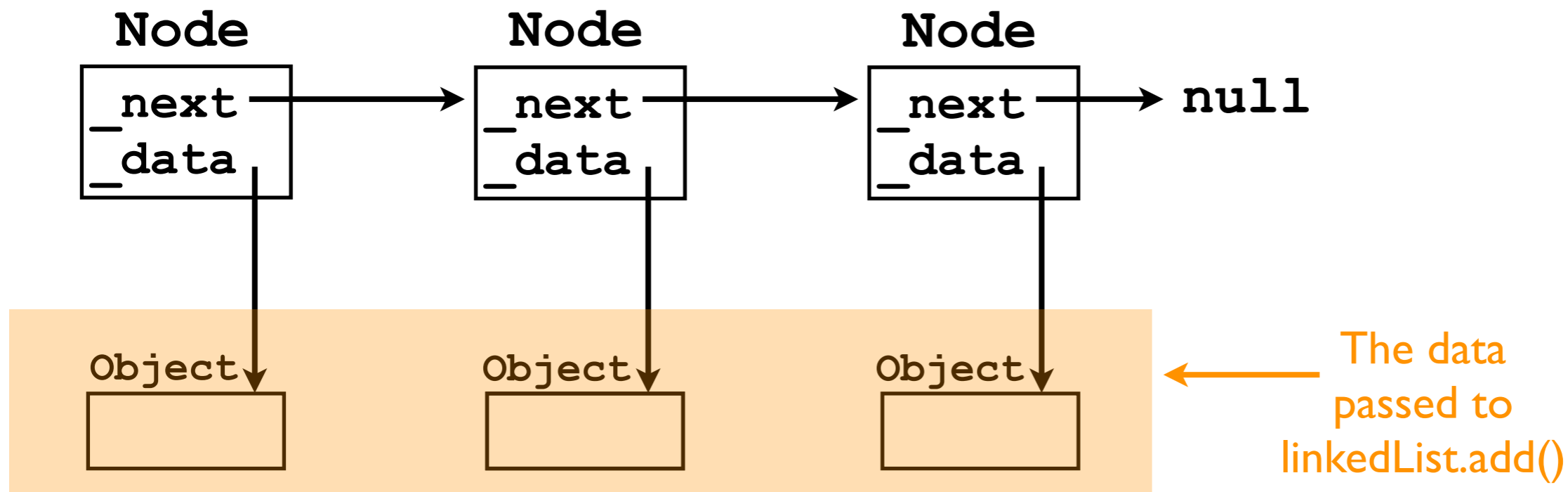
```
void add (Object o);
```

# Adding a new data element

- General strategy:

I. Store the user's data in "nodes on a chain":

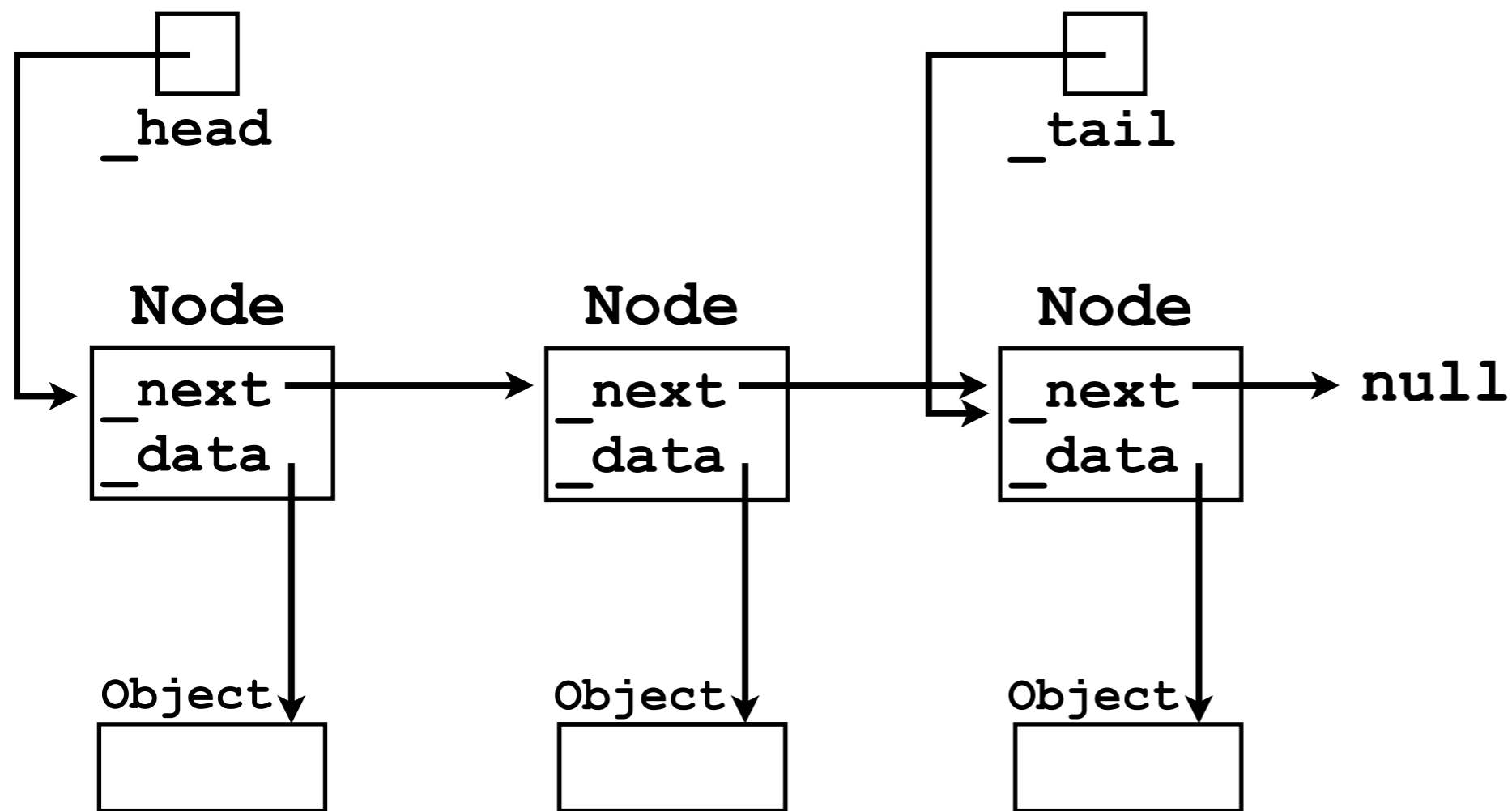
```
class Node {
 Node _next;
 Object _data;
}
```





# Adding a new data element

- General strategy:
  2. We also maintain pointers to the first (“head”) and last (“tail”) node in the chain.

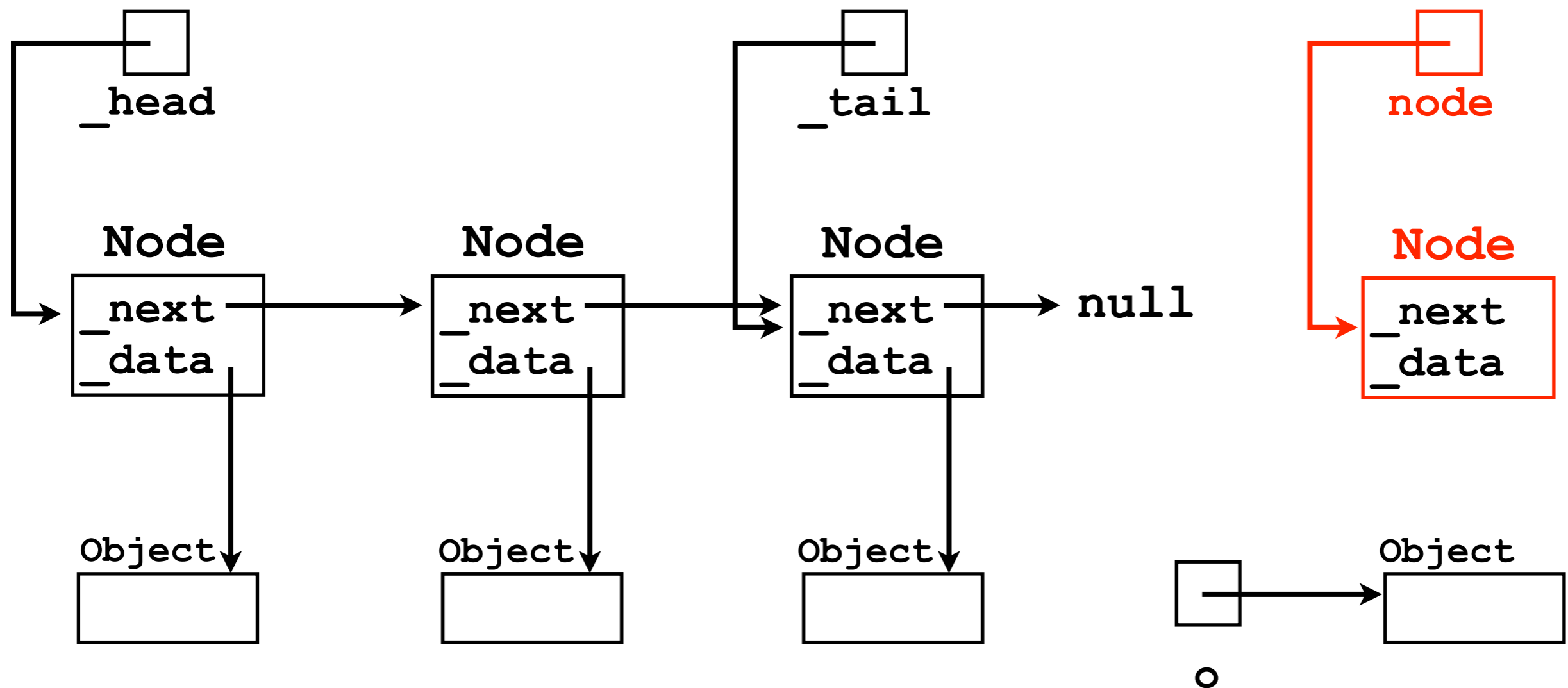


# Adding a new data element

- General strategy:

3. Each time `add(o)` is called, we create a new node.

```
Node node = new Node();
```

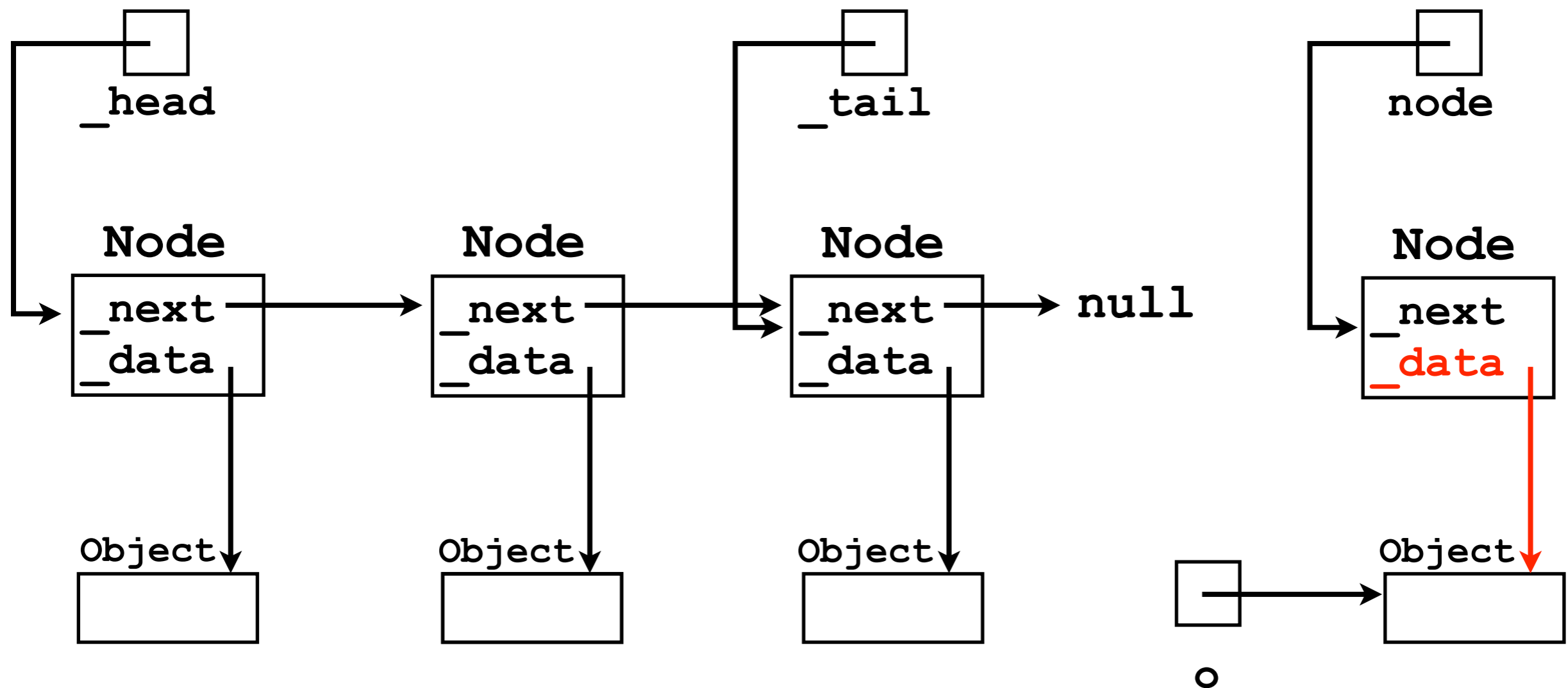


# Adding a new data element

- General strategy:

4. We store `o` “inside” the new Node.

```
node._data = o;
```

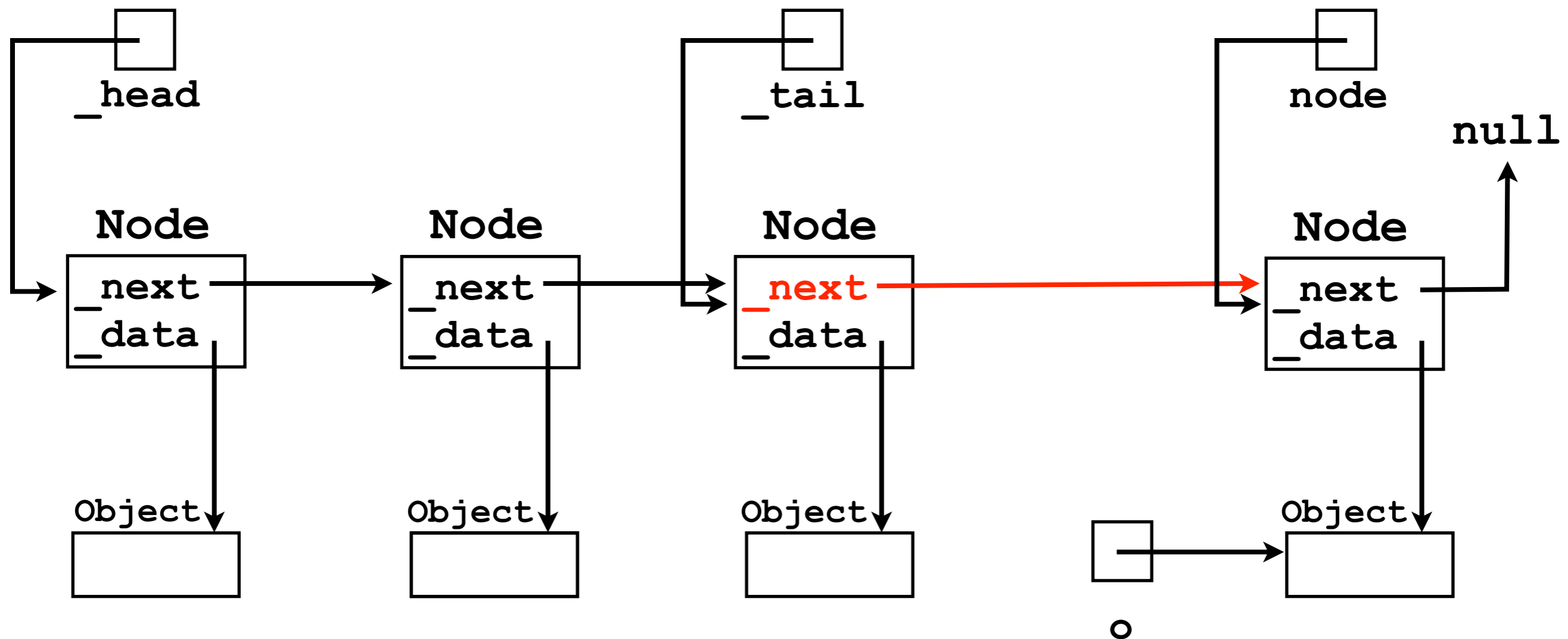


# Adding a new data element

- General strategy:

5. We connect the new Node to the rest of the chain.

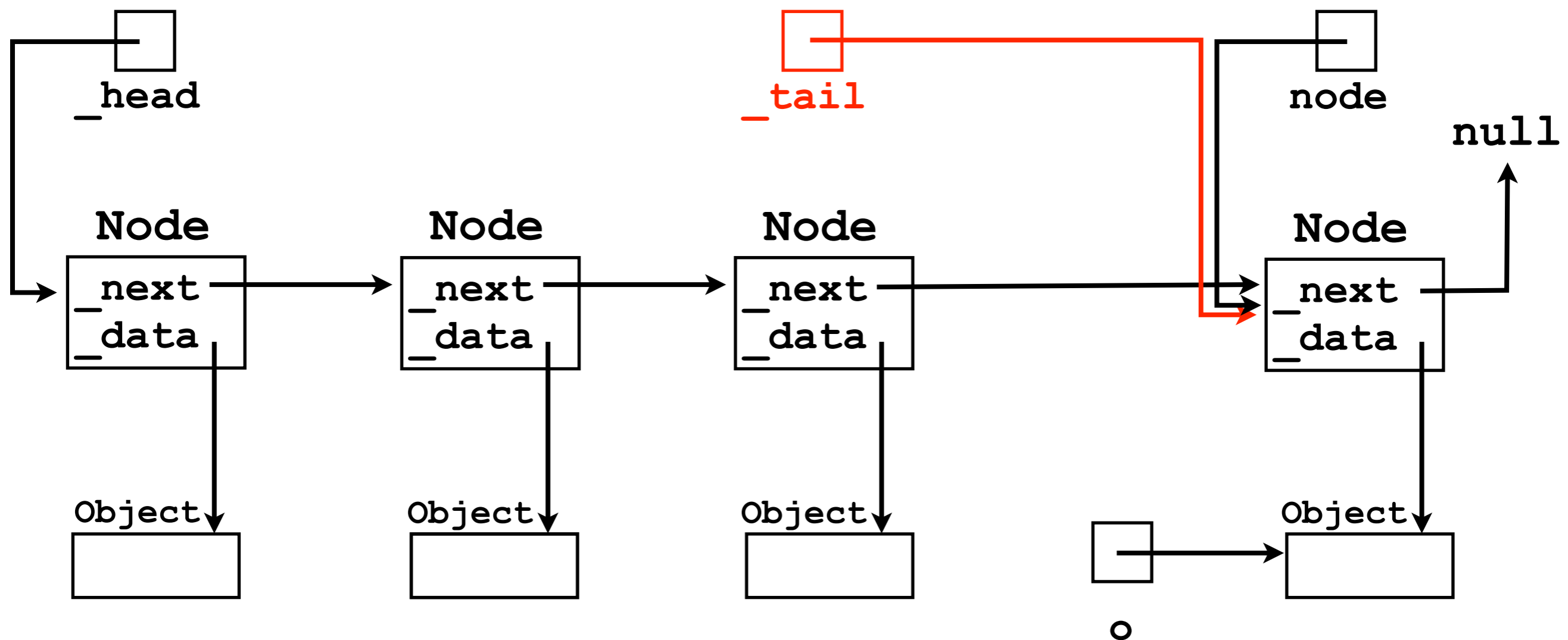
```
_tail._next = node;
```



# Adding a new data element

- General strategy:

6. We update the `_tail` pointer to point to the new node. `_tail = node;`



# Adding a new data element

- General strategy:
  - Done!

