Memory Diagrams — JAVA
Winter 2023
version 1.0

These notes are being created in conjunction with the teaching of CPSC 101 in the Winter 2023 term at the University of Northern British Columbia.

These notes are a work in progress, and copyright belongs exclusively to David Casperson.

Contents

1 Architecture **2**
- 1.1 Physical Components . . . 2
  - 1.1.1 Buses . . . 2
- 1.2 The OS View of Memory . . . 4

2 RAM, Boxes and Arrows **4**
- 2.1 Random Access Memeory . . . 4
- 2.2 Boxes for Primitive Types . . . 6
- 2.3 re-assignment . . . 8
- 2.4 Arrays and Pointers . . . 8
- 2.5 Objects and Pointers . . . 11
- 2.6 Pictures of Objects . . . 13

3 Memory Regions in a Java Program **14**
- 3.1 Lifetime . . . 14
- 3.2 The stack . . . 15
- 3.3 Stack frames . . . 15
- 3.4 The Heap . . . 19
- 3.5 Quadrant diagrams . . . 20
- 3.6 Code memory . . . 20
- 3.7 Static (once) memory . . . 22

4 To Come 22

1 Architecture

1.1 Physical Components

A typical computer can be decomposed into a number of kinds of subsystems: input/outout (I/O) devices, permanent storage, the CPU(s) and ALU(s), and random access memory (RAM), all inter-connected by information transfer buses.

I/O devices include things like cameras, keyboards, touch-pads, touch-sensitive screens, and the like.

Permanent storage means storage that remains even if you re-boot your computer. It typically manifests as files and directories. Verrrry approximately (depending on the decade) it is 1000 times larger and 1000 times slower than random access memory.

The **CPU** and **ALU** are where decision making, instruction interprettation, arithmetic and logic happens. These typically have tiny pieces of memory called *registers*.

`Random access memory` is characterized by being fast regardless of the order in which it is access, and non-permanent. On modern computers the situation is complicated by *cache memory* which is even faster than RAM buth much small, and which sits between the CPU and RAM.

1.1.1 Buses

The various subsystems are connected by *buses*, which transport information. Typically, there are three kinds of buses:

1. Data buses, which carry the the actual information being moved between components;
2. Address buses, which say *where* the information is coming from or going to, and
3. Control buses, which determine what is happening.

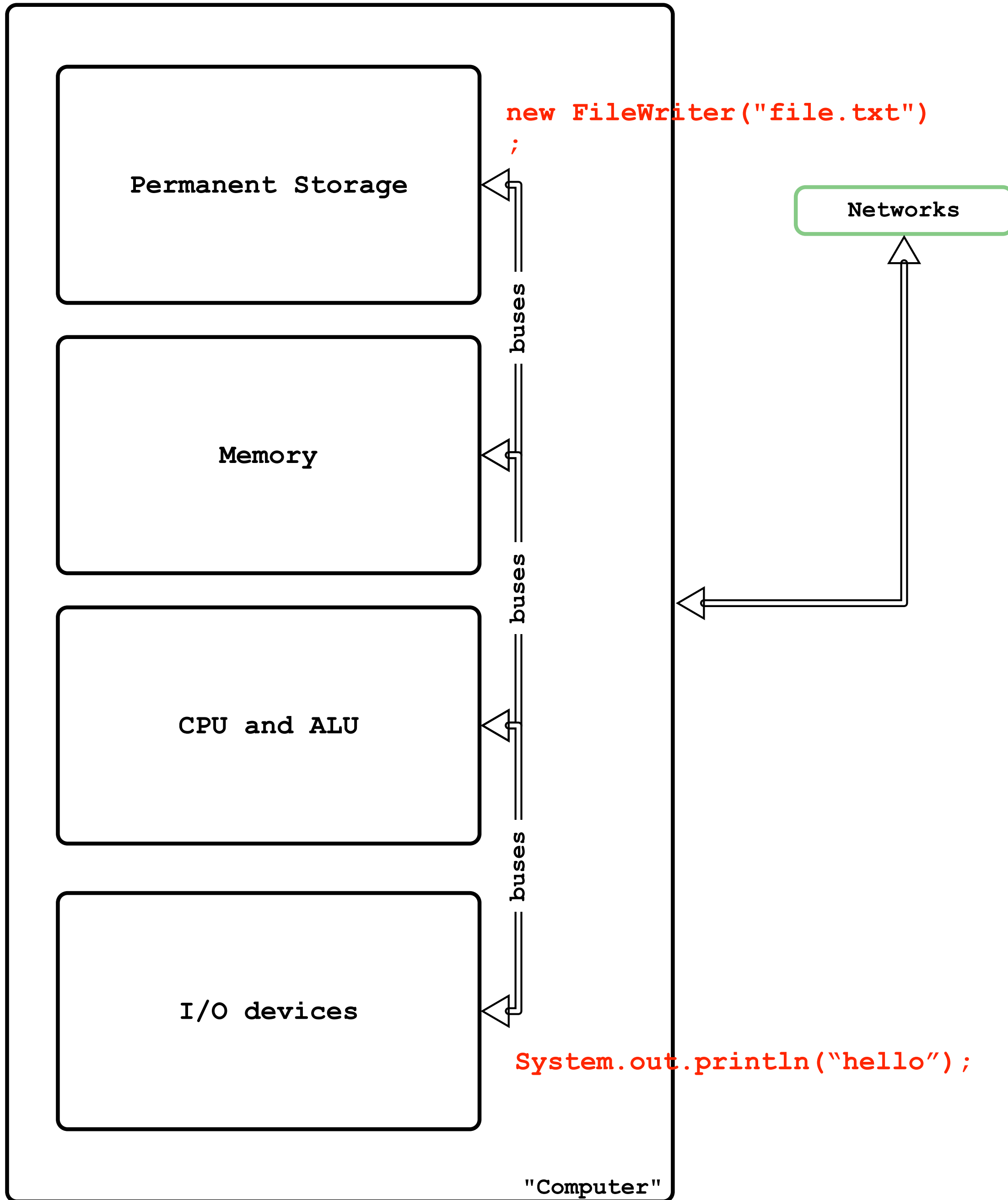

Figure 1: Physical hardware subsystems

1.2 The OS View of Memory

Applications and other user programs access physical hardware through the *operating system*. The operating system ensures the orderly behaviuor of multiple user processes.

Common operating systems include Linux, Mac OS, and various Microsoft products.

The operating system gives each program the illusion of having its own random access memory space that is completely disconnected from other running programs. See Figure 2.

2 RAM, Boxes and Arrows

From now on, we take a Java program point of view. That is, we (mainly) ignore the operating system. Some physical subsystems are accessed through appropriate Java libraries. Simple input and output typically happens through `System.in`, `System.out`, and `System.err`. Permanent storage is typically access through the `java.io` classes like `InputStream` and `FileWriter`. The CPU/ALU is implictly involved in executing all Java code. Pieces of code like arithmetic operators (*e.g.*, `+`, `%`, `|`) and control statements (*e.g.*, `return`, `for`-loops), directly involve use of the CPU/ALU.

Random access memory as seen by Java is typically through variables. The rest of this document is about random access mmeory (RAM).

2.1 Random Access Memeory

Modern computers use a *von Newcommand* architecture, an architecture first proposed by John von Heumann. A key feature of this architecture is that machine language instructions (that tell the CPU what to do) are stored in the same memory as numbers and text and video images.

In other words, random access memory is a sequence of boxes with no pre-determined purpose. Typically the smallest sized box to have its own address is an 8-bit byte.

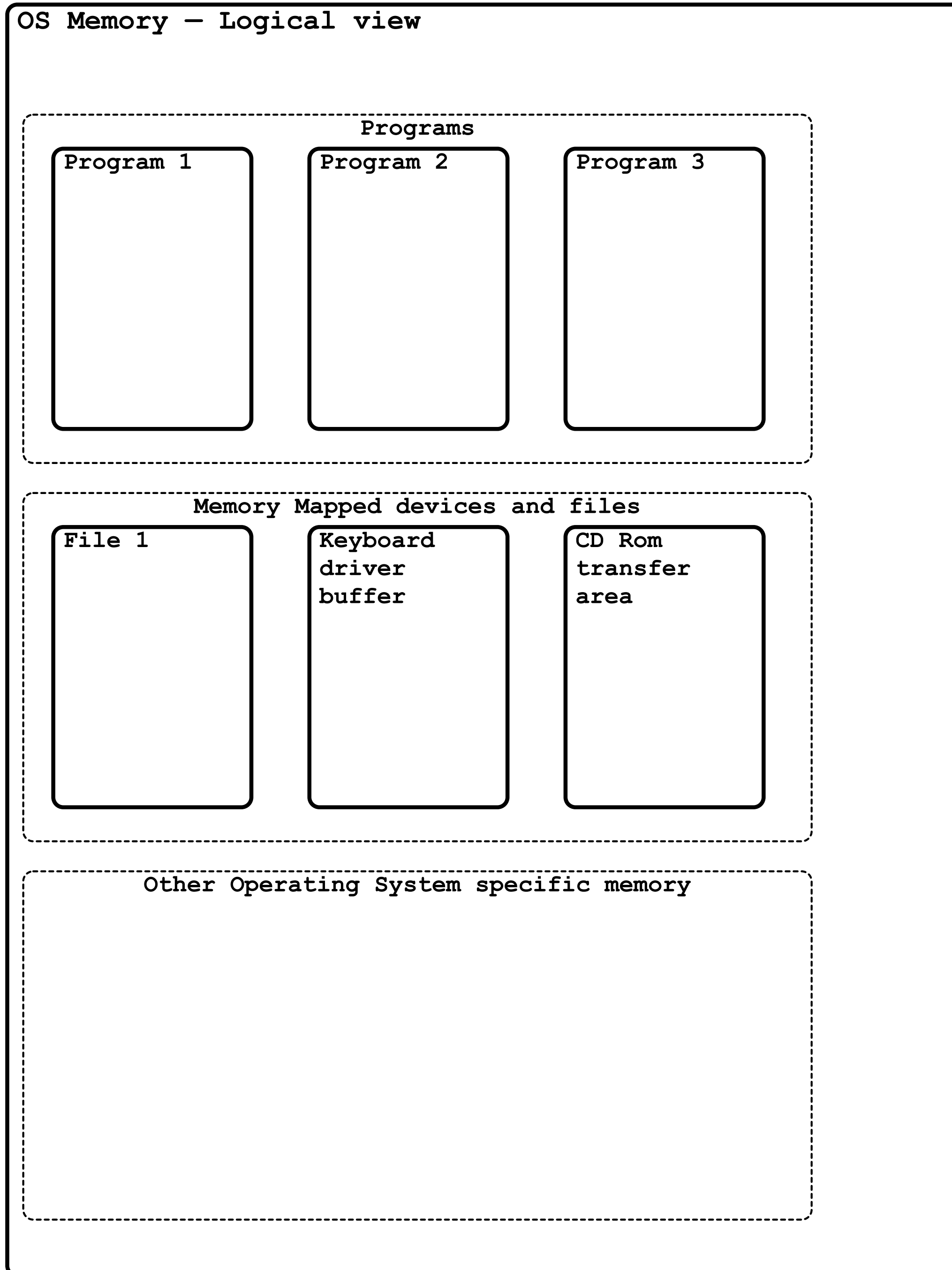

Figure 2: OS view of memory

In a von Neumann architecture, the meaning of the contents of a box depend on the mechine instructions that use it. See Figure 3. In particular, a region of memory might store a string of characters at one time, and nachine instructions later.

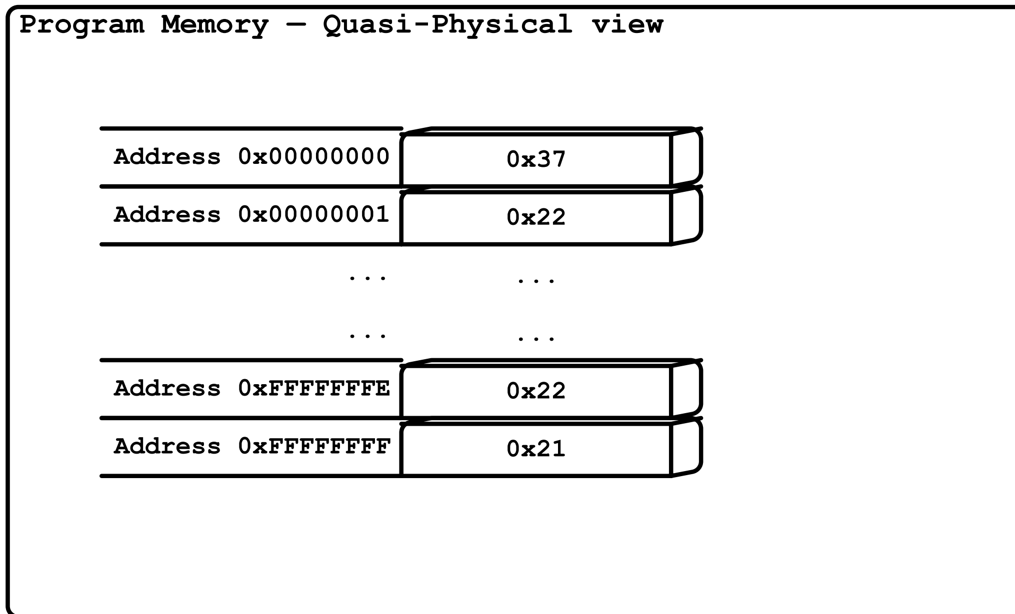

Figure 3: Memory as a sequence of addressable boxes

However, a programming language like Java imposes rules on how memory is used, and a picture like Figure 3 is not particularly useful.

2.2 Boxes for Primitive Types

Instead we draw boxes (and sometimes clouds) to represent a region of memory that is used for one purpose. For intance, a Java statement like

```
int x = 0x01021304; // 16911108
```

might be represented as shown in Figure 4 or Figure 5 (depending on how the machine stores multi-byte integers). However, Figure 6 gives a representation more suited to our purposes. Note that Figure 6 does not show the precise byte size of the object, nor its exact memory location,

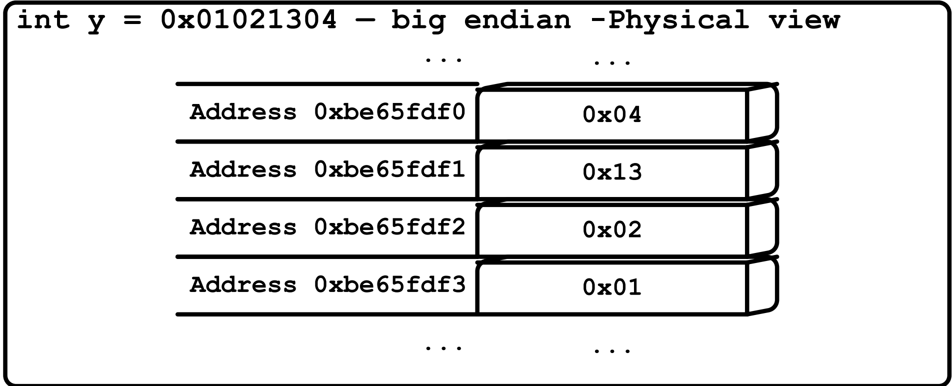

Figure 4: Assignment (big-endian)

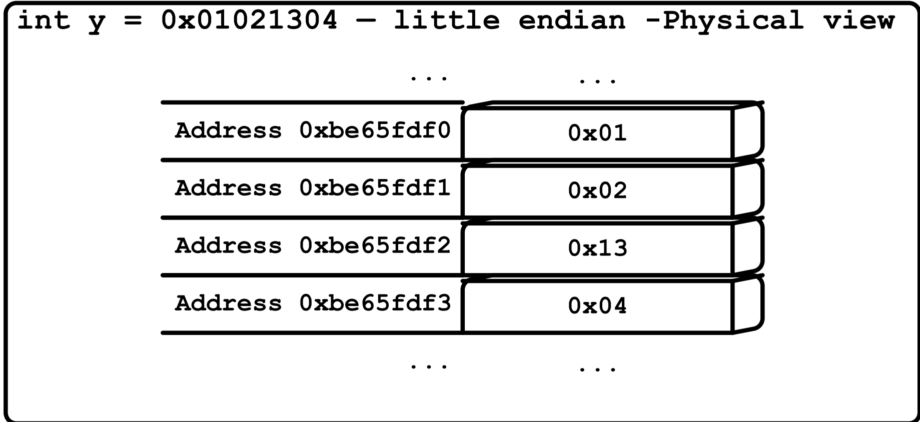

Figure 5: Assignment (little-endian)

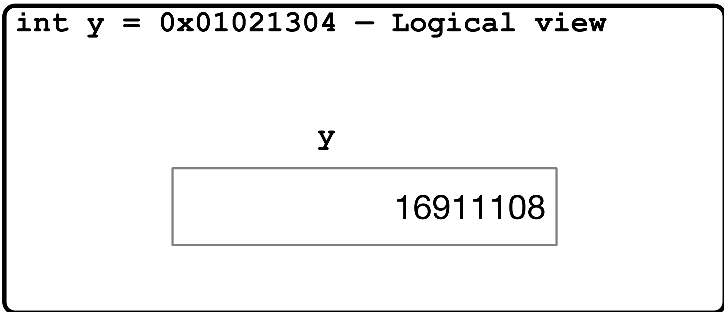

Figure 6: Assignment (logical view)

and it depicts the value in a human-readable format. Also note that we put variable name on the picture, although the actual variable name is not likely known at runtime.

This particular example used the `int` type. We use the same convention for all of the primitive types (`boolean`, `byte`, `char`, `double`, `float`, `int`, `long`, `short`). Note that in Java each box has a fixed *type*. That is, the following code is illegal:

```
int x = 16911108;
    x = false ;
```

In Java, once a box is created, it serves a fixed purpose (has a fixed type) during its lifetime.

We say that Java is a *statically typed* language.

2.3 re-assignment

The statements "`int x=5`" and "`x=5`" look quite similar but differ in one important point; the former tells the compiler to ensure that there is storage for the for the variable, and then give it an *initial* value; whereas the latter *modifies* an existing value. We know that

```
1   final int x = 5 ;
2   x = 5 ;
```

is illegal. In order to be clear, we will try to refer to the first kind of assignment as initialization, and the second as re-assignment.

In Java, re-assignment always means modifying the contents of exactly one variable box. (In C/C++ assignment can cause the copying of arbitrarily large chunks of memory.)

2.4 Arrays and Pointers

All non-primitive types in Java are objects, including arrays. However, arrays are sufficiently special that we treat them separately from other objects.

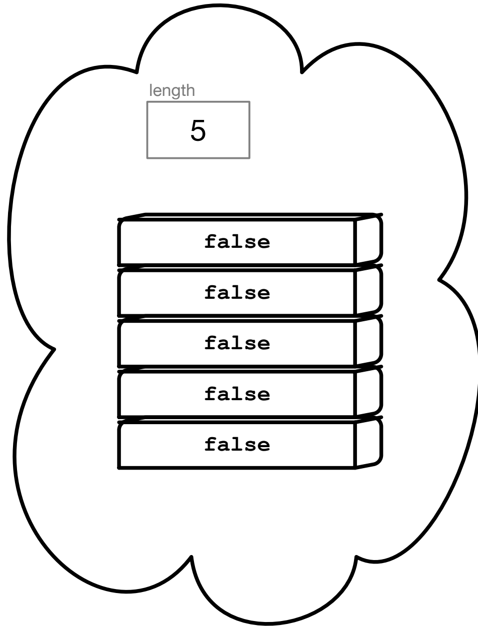

Figure 7: Array obejct created by “`new boolean[5]`”

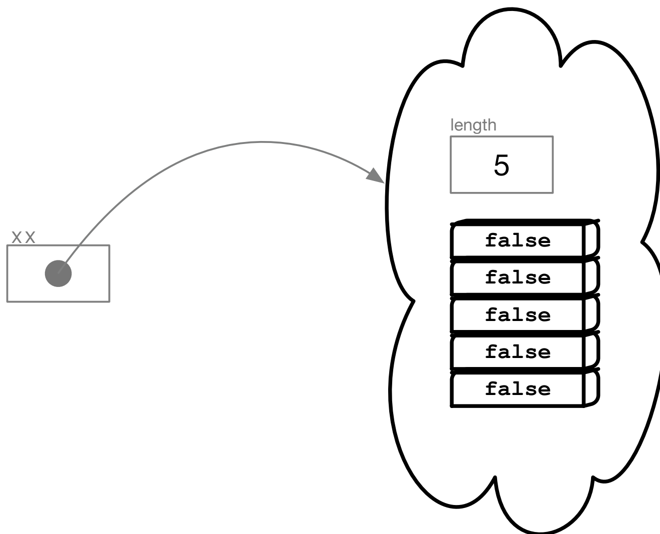

Figure 8: After executing “`boolean [] xx = new boolean[5]`”

Unlike with primitive types, we need to be very careful with arrays to distinguish between array **variables** and array **objects**. Let us start with the latter. Array objects are created by evaluating expressions of the form

`new` *type* `[`*n*`]`.

where *n* is an expression that evaluates to a non-negative integer. The *effect* of executing "`new boolean[5]`" is to create the object shown in Figure 7. It contains 5 identical slots for storing `boolean` values, and information about its size. The *value* of the expression "`new boolean[5]`" is the memory address of the array object it creates. This value can be stored in an array variable.

An array variable, for instance, `boolean [] xx`, is a fixed size memory box that contains either `null` or the memory address of an array object of the same type. When we execute code like

```
boolean [] xx = new boolean [5] ;
```

we get a memory picture as showin in Figure 8. A memory address as data is called a *pointer* (C/C++) or *reference* (Java). We draw this as a an arrow whose tail is at the location storing the memory address, and whose head points to the memory location in question.

Consider the code

```
1  boolean [] xx = new boolean [5] ;
2  boolean [] yy = xx ;
3  int [] zz = xx ; // compile time error
```

This code illustrates two points

- Creation of array variables is logical independet from the creation of array objects.
- Array variables (and pointers and references in general) also have types that must be respected.

The memory picture after line 2 is shown in Figure 9. An important point here is that although assignment to `xx` does not affect `yy`, assignment to the object pointed at by `xx` does affect `yy`.

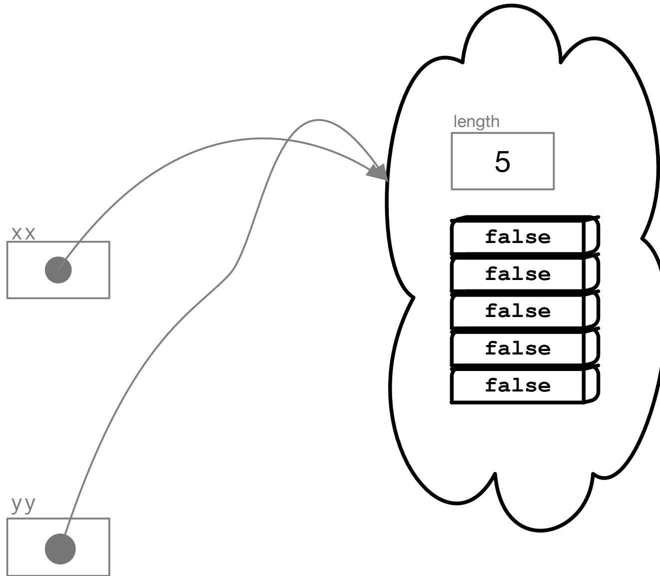

Figure 9: Two variables: one object

```
4    xx[2] = true;
5    System.out.println("yy[2] is "+yy[2]);
```

shows that `yy[2]` is now `true`. However, modifying the *variable* `xx` does not affect the *object* that `yy` is pointing to.

```
6    xx = new boolean[12] ; // re-assignment
7    System.out.println("yy[2] is "+yy[2]); // no change here.
```

\+ **Question 1.** Draw a memory picture explaining what is happening.

2.5 Objects and Pointers

As with array types, we need to be careful to distinguish between object variables and objects themselves. (Non-array) objects are created by evaluating expressions of the form

`new` *Type* `(...)`.

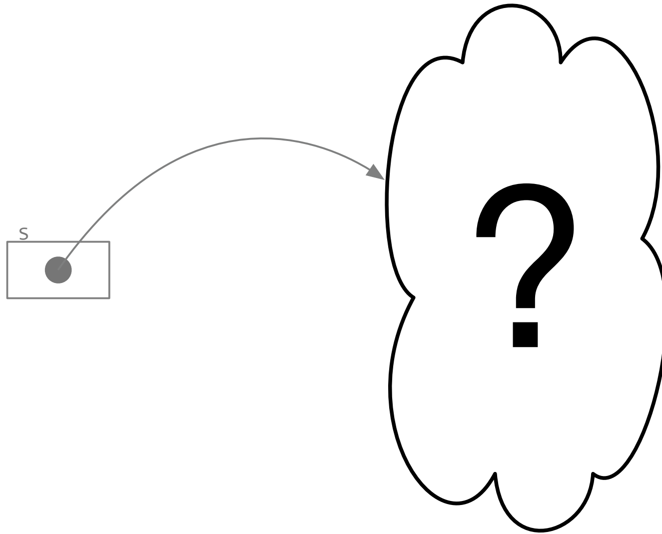

Figure 10: After executing "`Scanner s = new Scanner(r);`"

where *Type* is a class name and "`...`" represents possible constructor arguments. Suppose that `r` is a variable of type `java.io.Reader`. The *effect* of executing "`new Scanner(r)`" is to

1. allocate memory space for a `java.util.Scanner` object;
2. call the constructor function `java.util.Scanner.Scanner(Readable x)` with value `r`, which
3. initializes the member variables inside the `Scanner` object.

The *value* of the expression "`new Scanner(r)`" is the memory address of the object created. See Figure 10. In the case of the `java.util.Scanner` class, we don't what is contained inside the object because the member variables are all private and undocumented.

The actual boxes inside an object correspond to the non-static member variables of its class. For instance, if we have

Time1.java
```
public class Time1 {
  public  final static short SECONDS_PER_MINUTE=60;
  private int mySecond;
  public  int myMinute;  // ewww!!
  private int myHour;
  public Time1() { myHour=myMinute=mySecond=3;}
```

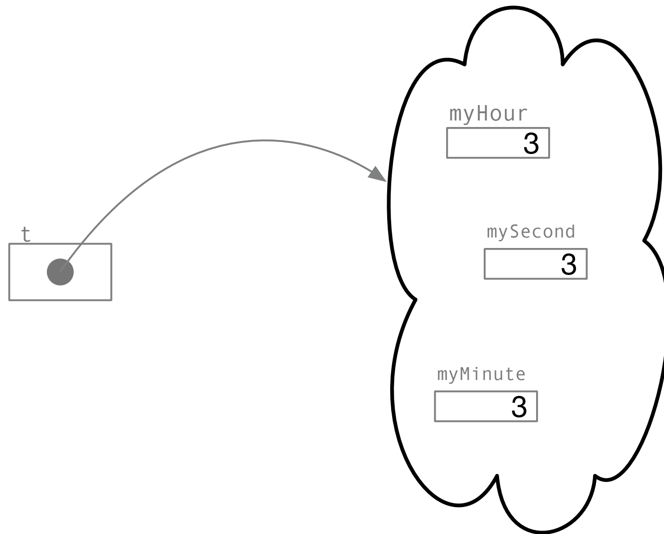

Figure 11: After executing "`Time1 t = new Time1();`"

```
  // ... member functions
}
```

and we execute elesewhere "`Time1 t=new Time1()`", we get the result shown in Figure 11. Note that:

1. `static` member variables are not part of the object created.
2. Methods (`static` or not) are not part of the object created.
3. Access keywords (`public`, `private`, `protect`) do *not* appear in run-time memory. (They *do* affect compile time; illegally attempting to access a `private` member variable results in a compile-time error.)

\+ **Question 2.** Continuing the above example, consder the code

```
1   Time1 u = t ;
2   t.myMinute = -3 ; // a good class would not allow this
3   t = null ;
```

and draw memory pictures showing what happens.

2.6 Pictures of Objects

In Figures 7–11 we have drawn objects as clouds. We want to emphesize that objects are aggregates of storage lcoations. In Java every object is an

instance of a named class.[1] Every object from the same class has the same *shape*. Figuratively, htis means that we draw each object of the same class with the same shape in a diagram. Literally, it means that each object in the same class has the same size, and the same internal layout of storage.

People sometimes use the *cookie cutter* metaphor, where classes are cookie cutters and objects are cookies. This communicates the point that different classes may have different shaped objects, but objects from the same class have the same shape. The metaphor also correctly suggests that objects (cookies) never change shape onece they are cut (constructed).[2]

If you want a memory diagram to explicitly indicate that there are two different classes of objects in use using non-cloud shaped cookies is appropriate.

Also note that it is the cookies (objects) that sit on the cookie sheet (memory diagram, heap memory), not the cookie cutter. The answer to the question "where do classes live in memory?" is complicated.

3 Memory Regions in a Java Program

We now look in more detail at how the Java runtime organizes memory.

3.1 Lifetime

We have said that (a) in the von Neumann architecture, random access memory has no pre-determined rôle, but, (b) in Java, a box has a fixed type during its lifetime. We now elaborate on the idea of *lifetime* of a box. As a program runs, the Java runtime allocates contiguous chunks of random access eemory (boxes!) for some purpose. Later, when it is

[1] Java supports *anonymous inner* classes that are extensions of named interfaces or classes and that do not have a name in the text of the program. In particular, λ-expressions exploit the abillty to create such classes. However, even though there is no name in the text of the program, the compiler creates a name and a corresponding `.class` file. Thus, at *run-time* every object belongs to a particular named class.

[2] Although `ArrayList` objects may appear to be able to change size and shape, at the memory-diagram level they don't.

finished with that purpose, the runtime deallocates the box. The lifetime of the box is the span of time between its allocation and deallocation.

During its lifetime, a box may have its contents change multiple times (for instance, consider a loop variable), but it hever changes *type*. That is, if it starts out as a box for storing `double` values, it is only ever used to store `double` values. Sometimes it may appear that we are storing in integer in double box (imagine "`double x = 3;`"). However, the quantity that is stored in the box is always in double format.

After a box's lifetime, the underlying memory is reclaimined, and the next time that it is allocated it may be used for a different purpose.

In the descriptions that follow, pay attention to the lifetimes of various kinds of storage.

3.2 The stack

The word **stack** has two meanings in computer science, one general, and one more specific. In general, a stack is a data structure that contains multiple items, but with quite restricted access. Think of a stack of plates at a smorgasbord restaurant. Only the top of the stack is accessible.

More specifically, a stack is a region of memory used for keeping track of variuables contained in methods. From now on we use stack in this sense.

3.3 Stack frames

The stack (in the program memory senxe) consists of a stack (in the general sense) of **stack frames**, where each stack frame corresonds corresponds to a method call. A stack frame for a method `m` is put on the stack when `m` is called. The frame is taken off the stack when the mehtod `m` returns. Between the time when `m` is called and when it returns, it is possible that `m` itself may call other methods, whose stack frames stack on top of the frame for `m`. It is even possible that the method `m` may be called again before it returns (we call such a mehtod *recursive*). That is, there may be more than one stack frame in existence for a particular method.

A stack frame contains memory for the following:

(i) method parameters,

```
public static double oddThing(int n)
    {
    if (n%2==0) ++n ;
    if (n==1)
        {
        return 1 ;
        }
    else
        {
        double t1 = oddThing(n-2) ;
        double t2 = Math.log(t1) ;
        double answer = n * t1 ;
        return answer ;
        }
    }
```

Figure 12: Recursive routine

(ii) for non-`static` methods, the `this` pointer,
(iii) variables declared inside the body of the method,
(iv) temporaries, and
(v) control information.

Consider the code shown in Figure 12, and suppose that we call `oddThing(5)`. The stack frames look as shown in Figure 13 at various times during execution of the code.

Note that parameters and local variables are very similar in stack memory diagrams. The lifetime of parameters is the duration of the call; the lifetime of a variable is from its declaration to the end of the block of code that contains it.[3] For simplicity, we often ignore the fact that some variables are declared in a smaller block than the whole function.

[3] Variables declared in the "`()`"-part of an `for`-loop live until the loop is exited.

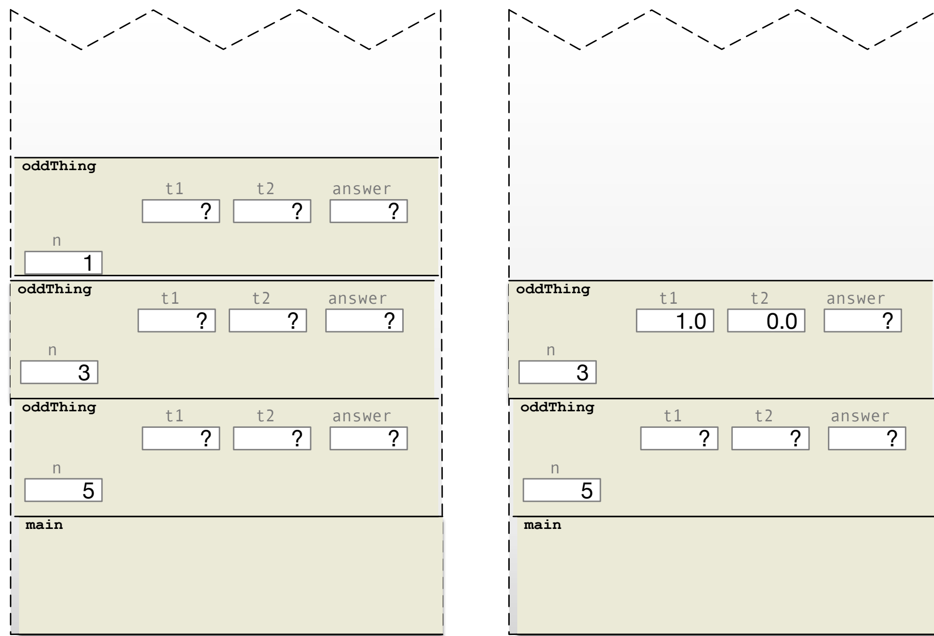

Figure 13: Stack diagrams

```
root1 =
    (- b
     - Math.sqrt
            (b*b-4*a*c)
    ) / (2*a) ;
```

```
double t1 = b*b;
double t2 = 4*a ;
double t3 = t2*c;
double t4 = t1 - t3 ;
double t5 = Math.sqrt(t4);
double t6 = - b - t5;
double t7 = t6 / 2 ;
double t8 = t7 / a ;
root1 = t8;
```

Figure 14: Mathematics verseus Explicit temporaries

Temporaries are in effect variables created by the compiler. Suppoese for instance, we change

```
14        {
15        double t1 = oddThing(n-2) ;
16        double t2 = Math.log(t1) ;
17        double answer = n * t1 ;
18        return answer ;
19        }
```

to

```
14        {
15        double t1 = oddThing(n-2) ;
16        double t2 = Math.log(t1) ;
17        return n * t1 ;
18        }
```

The quantity that we used to call `answer` still needs to be computed and stored somewhere. It is quite possible that the Java compiler may create a temporary to store it. In general, a complicated expression with lots of nested sub-expressions can be rewritten into a sequence of assignments involving less nested expressions. However, this is something that the compiler can manage on its own. See Figure 14 for another example.

Control information is other information that the Java virtual machine (JVM) needs to manage method calls and returns. In CPSC 101 we are not

concerned too much with what this information is, just an awareness that it exists.

3.4 The Heap

The word **heap** also has two meanings in computer science. As a data structure, a heap is a particular way to implmenent a *priority queue*. The other use of *heap* is as a region of program memory, like the stack. There is no connection between the data structure meaning and the region of program memory meaning.

Memory in the stack is automatically and implicitly controlled by method calls and returns. By contrast memory in the heap is partly under the explicit control of the programmer. Executing a `new` expression (whether for an array or a class object) creates a memory region in the heap that contains that object (see Sections 2.4 and 2.5). Contained in the memory region for an object are:

1. Slots for variables for all of the *non*-`static` member variables in its class declaration.

2. Memory for its super-class object.

   (We will discuss super class objects more when we discuss inheritance. In effect an object region can be thought of as a nested Russian doll. The ooutermost doll (the *actual* object type), corresponds to the class in the `new`-expression that created this memory region; the innermost doll is an object of class `Object` type.)

3. Various pointers (that is memory addresses) pointing to other object in the heap or in the static region. Some of these only occur in advanced programming situations:

   (a) a pointer to a **class** object in the static region that describes properties common to all of the objects in this class, such as the class name, a list of its methods, slots for all of the `static` member variables in its class declaration.

   (b) if the object is an object of a non-static inner class, a pointer to the object of the surrounding class that created this object.

4. Note that `static` member variables do *not* have slots inside an object. This is consisntent with each static variable occurring exactly once in a program.

In Java only the beginning half (allocation) of an object's lifetime is under explicity programmer control. Objects are removed from meory by the *garbage collector* when it determines that an object can no longer be used because there are no poniters to it.

3.5 Quadrant diagrams

We have described the *stack* and *heap* regions of memory, and how to draw diagrams for them. Before describing the remaining two regions (the "static" region and method code memory) we dscribe how we fit these four regions into one diagram. See Figure 15. We use a heavy vertical and horizontal line to divide our diagram into four quadrants. Clockwise, from the top left, these quadrants contain:

1. the stack,
2. the heap,
3. code memory, and
4. static (once) memory.

I always draw the quadrants in this order.

Typically when trying to understand how a program is using memory, we only need to look at the top two quadrants, or sometimes, just a few boxes for variables and a couple of objects.

3.6 Code memory

Most Java code is contained inside (non-`abstract`) methods inside classes.[4] Each method can be uniquely identified by its full signature, which con-

[4]There can also be code contained in the initializers for member variables, and specific (possibly `static`) initialzer blocks. These are generally confusing, as the order of their execution is not at all clear. I strongly recommend:

- not using initializer blocks.
- not using initializers in the declaration of non-`static` member variables (put these in constructors instead).

The potential benefits of not repeating common code don't justify the complexity of code outside of methods.

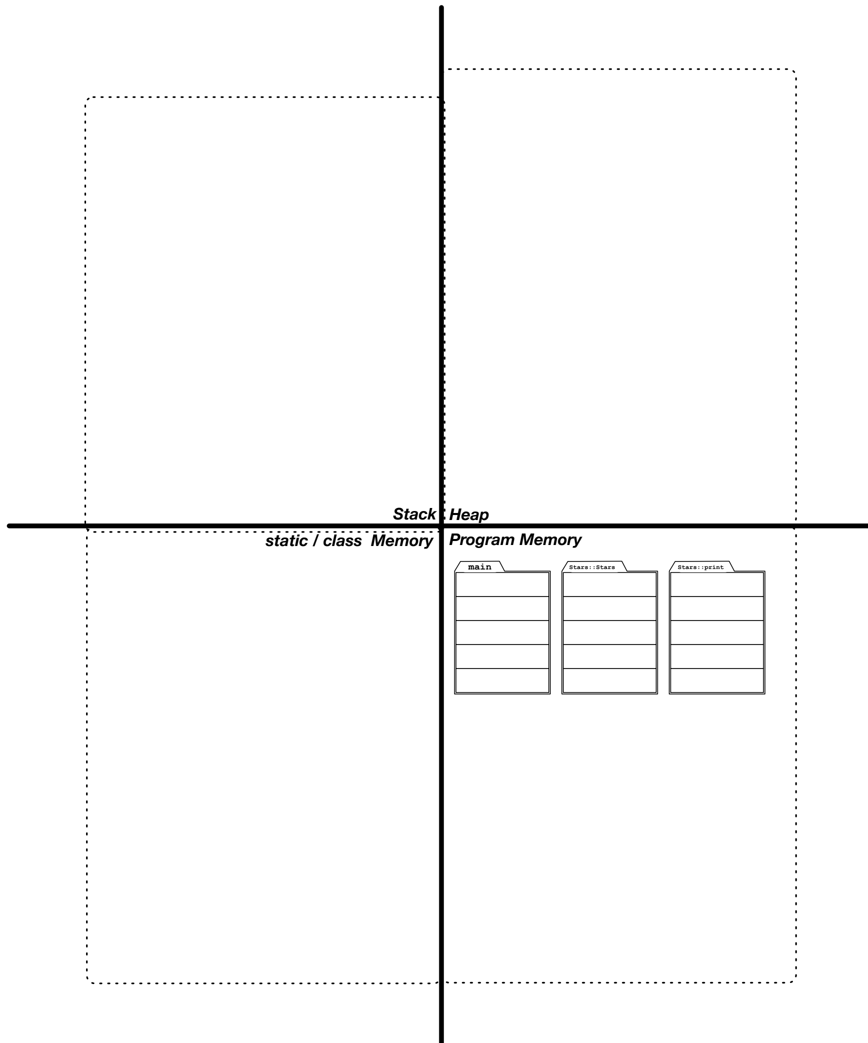

Figure 15: Quadrant Diagram

sists of

- its package (*e.g.*, `java.util`),
- its class name (*e.g.*, `Scanner`),
- its method name (*e.g.*, `hasNext`), and
- its argument pattern (*e.g.*, `(String)` ot `()`),

Each method contains a sequence of JVM instructions, together with information such as its maximum required stack frame size and its parameter pattern. Figure 15 shows how we draw method blocks.

Java code memory contains code for every method used by the program. We can imagine the lifetime of code memory to last for the duration of the program's execution. (Java VM code can also be loaded under programmer control, but that's beyond the scope of this paper.)

3.7 Static (once) memory

The remaining region of memory is characterized by its objects being unique. Many non-Java programming languages have *global* variables (variables that have top-level visibility). Language rules insist that there be exactly one instance of such a variable in a program.

Java does not have global variables, but static member variables have the same property. For instance, there is exactly and only one `System.out` variable. Each class in a program may have zero objects (think `java.lang.Math`) or many objects, but every class has a corresponding unique object of type `java.lang.Class`. Through an object's `Class` object one can find the object's class' `static` member variables, and pointers to its methods.

4 To Come

TRese notes are a work in progress.

Still to be added to these notes:

1. `this` pointers in stack diagrams
2. How superclass constructors work
3. exotic pointers (strong, soft, weak, phantom references) and finalizers.
4. Locks and other supports for concurrency.