Java classes and objects in memory

How are classes and objects represented in memory?

In Computer Science we often use abstraction - focussing on what things do, not how they are done. But it sometimes help to understand things by seeing actual concrete forms, not general abstract versions. So - how?

No-one knows

Of course that is not true. First - who decides? Oracle. How do they tell us? In the JVM Specification. One version is here:

https://docs.oracle.com/javase/specs/jvms/se13/html/index.html

The point is that 'the JVM' is not just one piece of software. Wiki lists more than 70 known JVMs, for different versions, different operating systems and hardware, and written by different people. Anyone can write a JVM, so long as it complies with the rules in the JVM spec.

Some things are fixed. For example the format of a .class file, of Java source code compiled to bytecode. This must be true, since the same bytecode must run on any platform with a JVM - *this is the whole point of Java*. So a lot of the JVM spec is taken up with how the contents of a .class file are organised. Also the meaning of the different op-codes - so the JVM writer knows how to execute bytecode.

Then at 2.7, the JVM Spec says

> *The Java Virtual Machine does not mandate any particular internal structure for objects.*

In other words, there are no rules about how objects are represented in memory. Any way will do, so long as the other rules in the JVM spec work.

Compare that with the language C. A C string will consists of a sequence of ASCII characters codes, one character per byte (ignoring 'wide chars') and with a 0 at the end. So byte by byte we know what a C string looks like in memory. Not so for Java objects.

But 2.7 goes on to say:

> *In some of Oracle's implementations of the Java Virtual Machine, a reference to a class instance is a pointer to a handle that is itself a pair of pointers: one to a table containing the methods of the object and a pointer to the `Class` object that represents the type of the object, and the other to the memory allocated from the heap for the object data.*

In other words, this says (without much explanation) of how *some of* Oracle's JVM implementation do it. But it makes no promises - it might be like this, or not, and it might change in the future anyway.

If it might not be like this - how can we write Java source code that will still work? Because it *does not matter*. The Java Language Specification (JLS) sets out the rules of Java, as source code. Those rules matter. But when compiled into bytecode, when a JRE loads it, it can set up objects in memory however it likes. It makes no difference to the person who writes Java source code.

Oracle's hint

But can we understand Oracle's version? It says

> *a reference to a class instance is a pointer to a handle that is itself a pair of pointers..*

Suppose our Java program says:

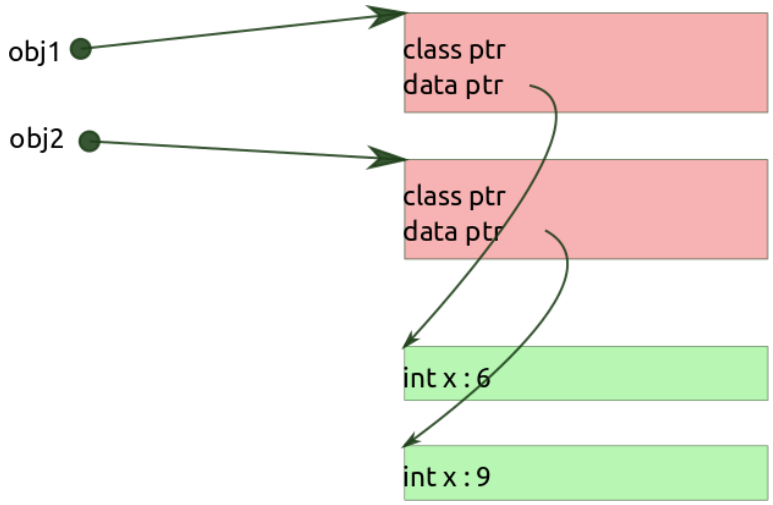
```
class Class1
{
  int x;
  Class1(int x)
  {
    this.x=x;
  }
}
```

and code to use it

```
   Class1 obj1 = new Class1(6);
   Class1 obj2 = new Class1(9);
```

What do we get in memory?

In heap memory we will have ..

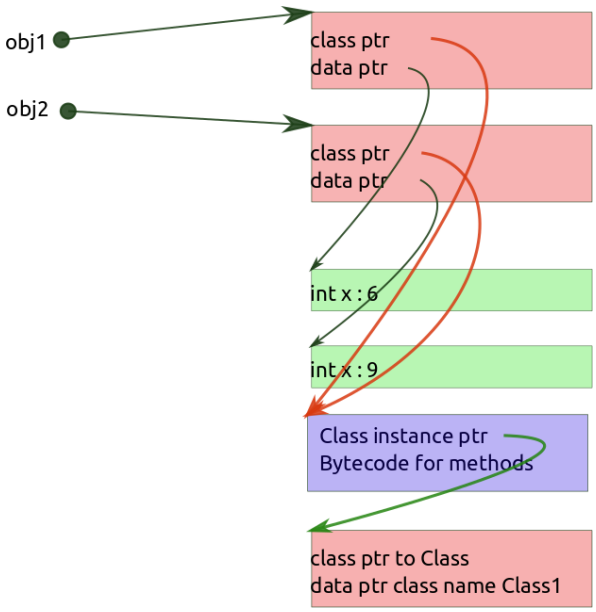


obj1 and obj2 are reference types - in other words, pointers to objects. The two pink blocks are those objects, in effect - each one, as Oracle says, *a pair of pointers: one to a table containing the methods of the object and a pointer to the* `Class` *object that represents the type of the object* (not yet shown) *, and the other to the memory allocated from the heap for the object data.*

The green blocks are the data of the two objects - here just ints, to keep it simple.

The pointers are C style - in other words, actual addresses of where this is in RAM.

Now we add a bit more:



The blue block is where Oracle says *a table containing the methods of the object and a pointer to the `Class` object that represents the type of the object.* Both obj1 and obj2 are the same class, Class1, so the two red arrows point to the same block. This block contains the bytecode for the class methods. In this example there are none, but there is a constructor, which in bytecode is treated as a special <init> method.

Why does this make sense? It separates the per class data (in blue) from the per object data (in green). We might have 1000 instances of class Class1. Each will have its own data (green block). But there is no point in storing 1000 copies of the byte code (blue block).

What about the green pointer? This is *a pointer to the `Class` object that represents the type of the object*. Suppose we add to the source code.

```
Class1 obj1 = new Class1(6);
Class1 obj2 = new Class1(9);
Class c= obj1.getClass();
System.out.println(c);  // class Class1
```

This is *reflection*. Here c is an object reference. What type is it? Type Class. It is an instance of the class Class. What does it represent? The class of obj1 - which is Class1.

So c also points to the pink block which the green arrow points to.