# Memory representation

As mentioned, the heap in our VM is represented as an array of unsigned integers[[1]](#footnote-2). Each object allocated on the heap has a two word header containing the size, GC bits and a class pointer. To overcome the bootstrapping problem of needing a pointer to the Class class before it has been loaded a number of “known” classes are bound to negative values during initialization and subsequently replaced when negative class pointers are encountered. The size recorded in the header is only used during GC and is not guaranteed to be the actual size of the object; see the section on our mark-sweep implementation. The pointer returned from the memory manager when allocating points to the first word after the header to make it easier for the actual object implementations to keep their offsets straight. Because the header is not part of the objects own representation it is also not part of the size given to the memory manager when allocating, and as such it would be illogical for the objects to have to take it into account when accessing their contents.

We have six different object layouts representing application defined objects, strings, arrays, classes and two forms of message handlers.

The two **message handler** representations share a two word header. The first word encodes a string pool index, visibility, whether it is an external handler and whether it is marked as an entry point. The second word contains a pointer to the class defining it. The representation for **external message handlers** takes up further two words: one for the external name used to resolve the actual .NET method and one for the number of arguments the method expects. **Internal message handlers**’ takes up a lot more: one word for local variable, argument and instruction counts (eight, eight and sixteen bits respectively) and an additional word per instruction.

**Classes** contain a lot more information and thus have a more elaborate representation. The first word encodes visibility and name (as a string pool index), the second is a pointer to the parent class, the third points to a string containing the name of the file defining the class (to be used for debugging purposes). The fourth contain field and message handler counts and the fifth contain inner class and super class counts. The sixth word contains a pointer to the class’ linearization if computed. From the seventh word forward the contents is guided by the counts in word four and five. First are the names of super classes, followed by message handlers and lastly inner classes. Because fields are only accessible from with a class there is no need to store their names as all loads are converted to offsets during class loading. Message handlers and inner classes take up two words each as we copy their header to make resolution faster.

The representation of **strings** is quite straight forward: one word for length in characters, one for hash code and one for every two characters[[2]](#footnote-3). Going forward we would have liked to make more efficient representations for substrings and concatenations but we simply did not have the time.

The **array** representation uses one word for the element count and two words per element to store a class pointer and object pointer. We store integers directly and so a way to distinguish between references and integers was needed. Our first approach allocated a number of words in the beginning of the array for a map indicating which entries were references and which were integers. This representation was doubtless more efficient but was too susceptible to race conditions: if two threads were to update two different elements that used the same word as map the risk of losing one of the updates is quite big. While we didn’t test it the overhead locking the array on each update would incur was deemed too great. Obviously a race condition can still occur if two threads update the same element but that is to be expected. Although the problems that can occur in our VM compared to e.g. .NET are a bit more serious. If one thread updates an element with an integer and another thread updates the same element to some object, it is possible that the integer will be treated as an object subsequently.

Finally we have the **application object representation**. The class of these objects is already recorded in the common object header so no need to store that once more[[3]](#footnote-4). Because we support multiple-inheritance[[4]](#footnote-5) we cannot calculate the final field offsets locally for each class but need to do it for the specific class we are instantiating. But even though we can calculate the final field offsets when we know the exact class we cannot actually update the message handlers so instead we store an additional offset which will be added to the locally calculated offset on each field access. These offsets are stored in the beginning of the object[[5]](#footnote-6). The first word is used to store an offset to the first field and the second forward stores the aforementioned offsets followed by two words per field (one for class and one for object reference).

Overall the current design represents an attempt at making a tradeoff between generality and efficiency. Because we have several different object layouts we cannot uniformly scan each reference in an object, but if we had a completely uniform object layout we would have terrible locality in the layout of classes and message handlers. The object header could have been limited to one word leaving the class pointer to be part of the representation of application objects and identifying the rest using some hardcoded bit pattern in the remaining header word. This would limit the maximum size of objects but not by so much as to make it a problem – currently the maximum size is 230 would be lowered to something like 226 which is still more than enough to be acceptable. This layout did however allow us to more directly reuse the objects representing classes and message handlers as mirrors in our reflection API. Generally any object on heap can be treated like an Object (the base type in our language) with no special cases.

1. Technically we have a Word struct wrapping [↑](#footnote-ref-2)
2. Conversion from .NET strings to VM strings are done by the built Unicode encoding found in System.Text.Encoding.Unicode. [↑](#footnote-ref-3)
3. An interesting side note is the fact that because the class stored in the object header is determined by the type parameter given to the allocation method and thus the class of application objects cannot be recorded this way. Instead the class is updated by the helper method used to create application object instances. [↑](#footnote-ref-4)
4. And because super classes are not resolved until the first instantiation. [↑](#footnote-ref-5)
5. This also represents a bit of wasteful object layout design: the list of offsets might as well be stored in each class instead of in each object. [↑](#footnote-ref-6)