Overview. This worksheet has worked examples for performing memory analysis of small pieces of code.

To get the full benefit, review pages 200-204 from the book. Go through the worked examples below in the same order they are presented and avoid looking at the solutions and explanations until you have tried to find the solutions on your own.

Built-in Types

Question. Using the 64-bit memory cost model from lecture and the textbook, how much memory does each of the following pieces of code use?

Ex. 1

```
private int a = 0;
private int b = 123456789;
private double c = 3.14;
```

Solution. 16 bytes.
Primitive types use the same amount of memory regardless of the value stored in the variable. Each `int` variable uses 4 bytes and a `double` variable uses 8 bytes.

Ex. 2

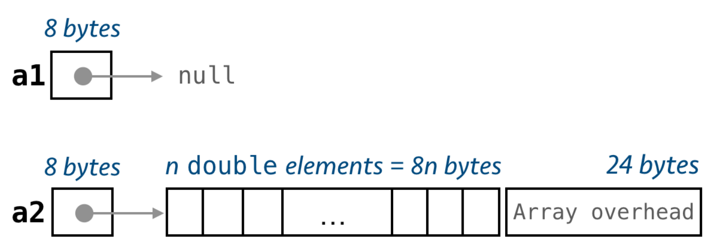
```
private double[] a1;
private String b;
```

Solution. 16 bytes.
This creates a *reference* to an array and a reference a String object. No actual array object or String object is created. A reference in java uses 8 bytes, regardless of what the reference type is.

Ex. 3

```
double[] a2 = new double[n];
```

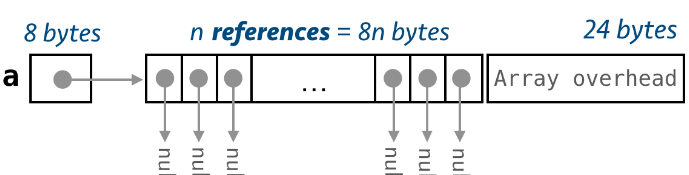
Solution. ~8*n* bytes.
This creates a *reference* to an array and also an actual array object of size n. The reference uses 8 bytes and the array uses $24 + 8n$ bytes. The illustrations compare `a1` in **Ex.2** and `a2` in **Ex.3**:



Ex. 4

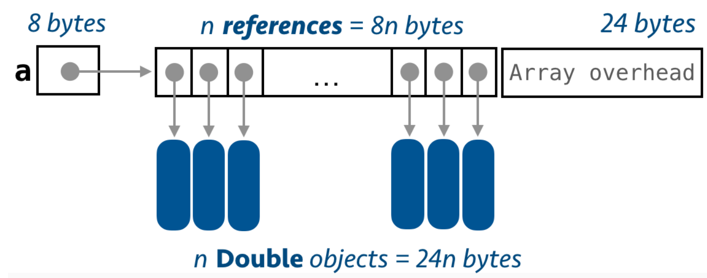
```
Double[] a = new Double[n];
```

Solution. ~8*n* bytes.
This creates a *reference* to an array (8 bytes) and also an actual array of *references* to objects of type `Double` (with capital D), where the references are initially `null` as shown below. Each reference in the array uses 8 bytes.



Ex. 5

```java
Double[] a = new Double[n];
for (int i = 0; i < n; i++)
    a[i] = new Double(Math.random());
// An object of type Double uses
// 24 bytes regardless of the value
// it stores.
```



Solution. ~32*n* bytes.

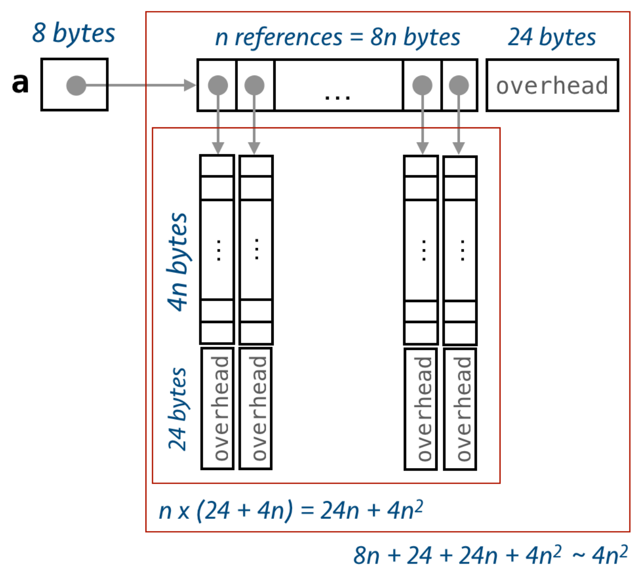
The references in the array are pointing to actual `Double` objects (i.e. they are not `null` as in **Ex. 4**). The total is: $8n$ (references to `Double` objects) $+$ $24n$ (`Double` objects) $+$ 24 (array overhead) $+$ 8 (reference to array) **~** $32n$ bytes.

Ex. 6

```java
int[][] a = new int[n][n];
```

Solution. ~4*n*² bytes.

There are n^2 elements in the array each of size 4 bytes. However, the total size is not exactly $4n^2$ bytes. As the illustration to the right shows, 2D arrays in Java are implemented as arrays of arrays.



Ex. 7

```java
Double[][] a = new Double[n][n];
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        a[i][j] = new Double(0.5);
```

Solution. ~32*n*² bytes.

The first line creates n^2 (`null`) references to `Double` objects. Each reference uses 8 bytes. This is exactly like **Ex. 6** with the 4-bytes `int` elements replaced by 8-bytes references. The for loop creates n^2 `Double` objects. Each object uses 24 bytes (see **Ex. 5**). The total is $8n^2 + 24n^2 \sim 32n^2$ bytes.

Ex. 8

```java
char[] a = new char[10];
```

Solution. 56 bytes.

Since each `char` requires 2 bytes, the size of the array (without the array overhead) is 20 bytes. However, in 64-bits memory, objects use blocks that are multiples of 8 bytes. Therefore, 4 bytes of *padding* are added. The total is: 8 bytes (reference to array) $+$ 20 bytes (10 characters) $+$ 4 bytes (padding) $+$ 24 bytes (array overhead) $=$ 56 bytes.

User-defined Types

Ex. 9

```java
public class Complex {
      private double real;
      private double imag;
}
```

Solution. 32 bytes.
Objects use 16 bytes of overhead plus the size of their data members. In `Complex`, there are two data members of type `double` (8 bytes each).

Ex. 10

```java
public class MyType {
      private int a;
      private int[] b;
}
```

Solution. 32 bytes.
In addition to the 16 bytes of object overhead, there are 4 bytes for the `int` data member and 8 bytes for the reference to the array. The total is 28 bytes. Since this is not a multiple of 8, 4 more bytes of padding are added.

Note. We don't have information about the size of the array that will be attached to `b`. Therefore, we considered only the size of the reference. Assuming that an array of size n is attached to `b`, the size of an object of type `MyType` including the referenced memory becomes $32 + 4n \ \sim 4n$ bytes

Ex. 11

```java
public class Queue {
      private Node first, last;

      private static class Node {
            private int item;
            private Node next;
      }
}
```

Solution.
An empty `Queue` uses 16 bytes (object overhead) plus 8 bytes for each of the references to `Node` objects. I.e., the total is 32 bytes.

A `Queue` with n nodes uses 32 bytes $+ \, n \times$ the size of each Node.

An object of type `Node` requires 16 bytes of object overhead + 4 bytes for the `int` item + 8 bytes for the reference to the next `Node`. This totals to 28 bytes, which requires 4 bytes of padding to become a multiple of 8. Hence the total is 32 bytes (class Queue) $+ \ 32 \ \times n$ (n nodes) $\ \sim 32n$ bytes.

Ex. 12

```java
public class Queue {
      private Node first, last;

      private class Node {
            private int item;
            private Node next;
      }
}
```

Solution.
The only difference between this and **Ex. 11** is that the `Node` class is not `static`, which means that it has to store a reference to the `Queue`. Hence, the size of a `Node` object is 32 bytes (as computed in **Ex. 11**) + 8 bytes (reference to the `Queue`) = 40 bytes.
Hence, a `Queue` with n nodes uses $\sim 40n$ bytes.

Ex. 13

```
public class Queue<Item> {
      private Node first, last;

      private class Node {
            private Item item;
            private Node next;
      }
}
```

Solution. ~40*n* bytes.

This differs from **Ex.12** in that the `Queue` is *generic.* The data member `item` is a *reference* to an object whose type is not known until runtime. Therefore, we will consider the size of the reference (8 bytes) without the size of the object attached to it, unless we have information on what the type of this object is.

A `Node` object requires: 16 bytes (object overhead) +
8 bytes (reference to the `Queue`) +
8 bytes (`Item` reference) +
8 bytes (`Node` reference),

which is 40 bytes in total. Adding 32 bytes for class `Queue` (see **Ex. 11**) and multiplying the node size by the number of nodes n gives $32 + 40n \sim 40n$ bytes for a `Queue` of n nodes.

Ex. 14

```
public class Queue<Item> {
      private Node first, last;

      private class Node {
            private Item item;
            private Node next;
      }
}

// assume the Queue has n
// Nodes of Double objects
```

Solution. ~64*n* bytes.

A `Node` object requires: 16 bytes (object overhead) +
8 bytes (reference to the `Queue`) +
8 bytes (`Item` reference) +
24 bytes (`Double` object) +
8 bytes (`Node` reference),

which is 64 bytes for a single node and $\sim 64n$ bytes for a `Queue` of n nodes.