INFO1113 Object-Oriented Programming

Week 4A: Binary IO and Memory
Interpreting data and understanding the memory model

Topics

Binary IO

In the previous week we saw how we could read data from text files. We will start this lecture by reading a binary file and how we can store data in a non-readable format.

What info do we need?

Like text data we can employ patterns but they are not easily obvious to inspect.

When reading a binary file there is a specific layout to interpret it correctly. This is typically bundled with a **file format specification**.

What info do we need?

Like text data we can employ patterns but they are not easily obvious to inspect.

When reading a binary file there is a specific layout to interpret it correctly. This is typically bundled with a **file format specification**.

For example, a **poke.dex** file format.

pokedex_header: 40 bytes	seen_pokemon: seen*4 bytes
magic: 4 bytes : int owner: 32 bytes : char no_pkmn: 2 bytes : short seen: 2 bytes : short	pkmn_id: 2 bytes : short type_1 : 1 byte : byte type_2 : 1 byte : byte

What info do we need?

We also need to know about the **endianness** of the data as well.

Without this information we may not interpret the data correct and therefore get strange values.

In the following examples we will assume **machine default.**

Binary I/O

Let’s read some files!

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

public class BinaryReader {

    public static void main(String[] args) {
        try {
            FileInputStream f = new FileInputStream(“some_file.bin”)
            byte[] buffer = new byte[4]; //We will read 1 integer
            f.read(buffer);
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Binary I/O

Let's read some files!

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException

public class BinaryReader {

    public static void main(String[] args) {
        try {
            FileInputStream f = new FileInputStream("some_file.bin")
            byte[] buffer = new byte[4];
            f.read(buffer);
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Okay we have read the data but what can we do with it?

Let’s read some files!

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException

public class BinaryReader {

    public static void main(String[] args) {
        try {
            FileInputStream f = new FileInputStream(“some_file.bin”)
            byte[] buffer = new byte[4];
            f.read(buffer);
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

We need to **interpret** the data.

We need to convert the data

Binary I/O

Let’s write some files!

```
import java.io.FileInputStream;

public class BinaryReader {

    public static int convert(byte[] b) {
        return (b[3] & 0xFF) |
          ((b[2] & 0xFF) << 8) |
          ((b[1] & 0xFF) << 16) |
          ((b[0] & 0xFF) << 24);
    }
}
```

Considering we have captured the data in a **byte[]** array

We need to transform this to an integer.

Shifting and bitwise operations

Binary I/O

Let’s read some files!

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

public class BinaryWriter {
    //<snip> Assume we have convert here
    public static void main(String[] args) {
        try {
            FileInputStream f = new FileInputStream(“some_file.bin”)
            byte[] buffer = new byte[4];
            f.read(buffer);
            int v = convert(buffer);
            System.out.println(v); //Output the integer
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

How about writing?

Binary I/O

Let's write some files!

```
import java.io.FileOutputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

public class BinaryWriter {
    //<snip> Assume we have convert here
    public static void main(String[] args) {
        try {
            FileOutputStream f = new FileOutputStream("some_file.bin")
            int v = 50;
            byte[] buffer = convert(v);
            f.write(buffer); //Write out the bytes
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Binary I/O

Let’s write some files!

```
import java.io.FileOutputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

public class BinaryWriter {
    //<snip> Assume we have convert here
    public static void main(String[] args) {
        try {
            FileOutputStream f = new FileOutputStream(“some_file.bin”)
            int v = 50;
            byte[] buffer = convert(v);
            f.write(buffer); //Write out the bytes
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

We write the byte[] array to the file.

Binary I/O

Let's write some files!

```
import java.io.FileInputStream;

public class BinaryReader {

    public static byte[] convert(int v) {
        byte[] b = new byte[4];
        b[0] = (byte) (v >> 24);
        b[1] = (byte) (v >> 16);
        b[2] = (byte) (v >> 8);
        b[3] = (byte) v;
        return b;
    }
}
```

From the integer we need to transform this to a byte array for writing.

Sidenote: How are classes encoded?

.class files are not human readable, just like image formats and executables but how does the **JVM** understand and interpret this file?

Simply, there is some method of interpreting binary files and being able to read them. By following the specification on the by Oracle[1] we can design a program to interpret and understand the class.

However, for demonstration purposes, let's start off with something that is a little more digestible.

Let’s read and write some data

There must be a better way!

Of course!

We just showed the hard way of reading and writing. We can do away with these conversions entirely using **ByteBuffer** and **ByteOrder** classes to manipulate the data with its inbuilt methods.

We can even go a **step further** and use a **DataInputStream** and **DataOutputStream** and use methods such as **writeInt** and **readChar.**

Applications

Why would we want to do this?

There are going to be applications where the data itself will not be presented in a textual representation.

- Interacting with devices and peripherals (gamepads, midi devices)
- Modifying executables
- Encoding images
- Writing your own file format for your programs
- Writing software for automotive applications
- Video and audio streams
- Networking application

Let’s simplify it

Stack

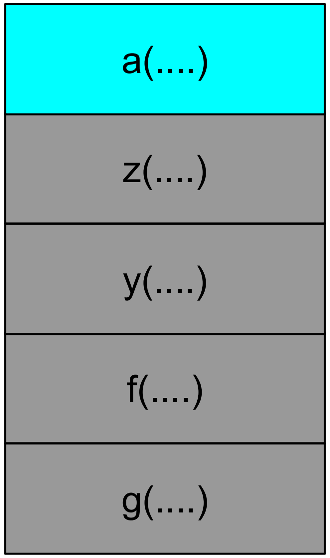
We're revisiting the **stack**.

Everytime we invoke a method it will be placed on the stack.

We need to understand the fixed limit of the stack within the **JVM**. By default the stack limit is **1MB**.

A **stack frame** can be thought of as an **instance** of a method. Each method has a size which is the combination of its **instructions, variables and argument data**.

Each stack frame has a **return address.**



Stack variables

Heap

The heap is separate memory space for which objects are **dynamically** allocated.

Whenever we use the **new** keyword, memory for the object is allocated and an address is returned.

Unlike the stack, heap allocated object lifetimes are a little more complicated.

Stack based objects exist in lexicographical scope. (**ie the {} braces**).

While heap objects can be aliased and their lifetimes are dictated when all references are no longer in scope.

Heap

The heap is separate memory space for which objects are **dynamically** allocated.

Whenever we use the **new** keyword, memory for the object is allocated and an address is returned.

Unlike the stack, heap allocated object lifetimes are a little more complicated.

Stack based objects exist in lexicographical scope. (**ie the {} braces**).

While heap objects can be aliased and their lifetimes are dictated when all references are no longer in scope.

```
int[] array = new int[16];
```

Let's consider the following

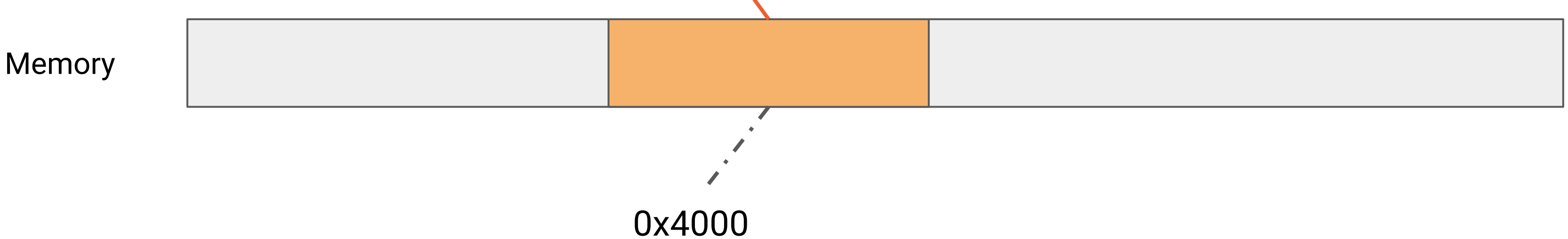
```java
public static void main(String[] args) {
  int x = 5;
  int[] array = new int[16];
  int[] alias = array;
  int y = x;
}
```

Memory

Heap

Let’s consider the following

```
public static void main(String[] args) {
  int x = 5;
  int[] array = new int[16];
  int[] alias = array;
  int y = x;
}
```

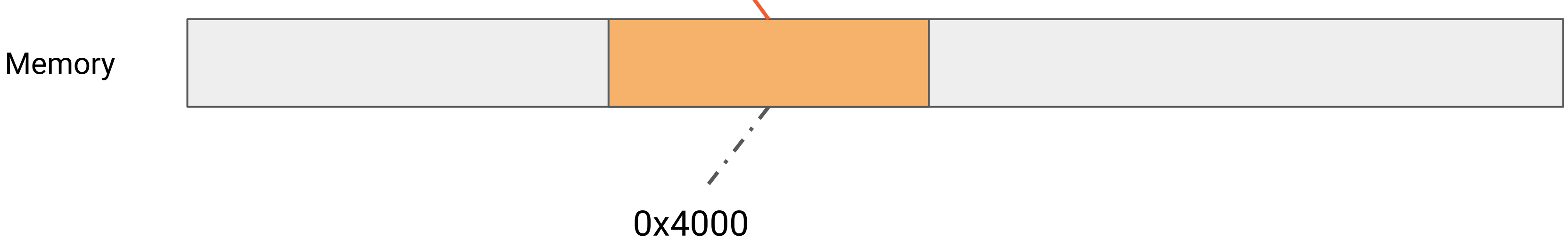


Heap

Let's consider the following

```
public static void main(String[] args) {
    int x = 5;
    int[] array = new int[16];
    int[] alias = array;
    int y = x;
}
```

We can see that **array** has its own memory space which of the size of 16 integers. What would occur in the following assignment with **alias**.

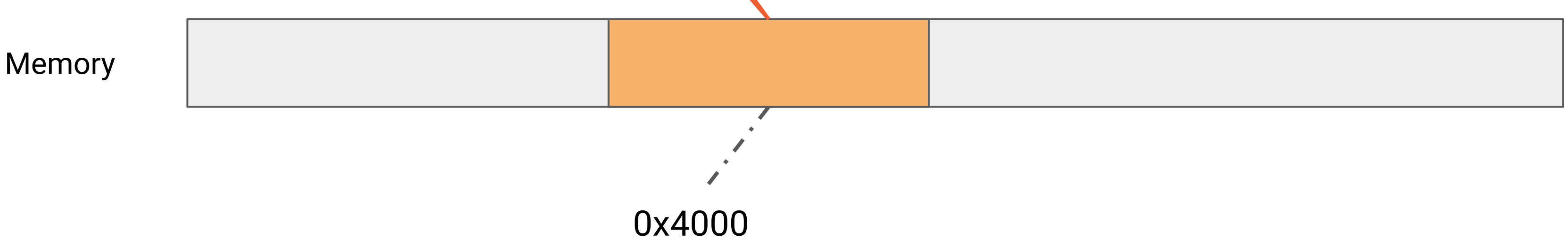


Heap

Let's consider the following

```
public static void main(String[] args) {
    int x = 5;
    int[] array = new int[16];
    int[] alias = array;
    int y = x;
}
```

We can see that **array** has its own memory space which of the size of 16 integers. What would occur in the following assignment with **alias**.

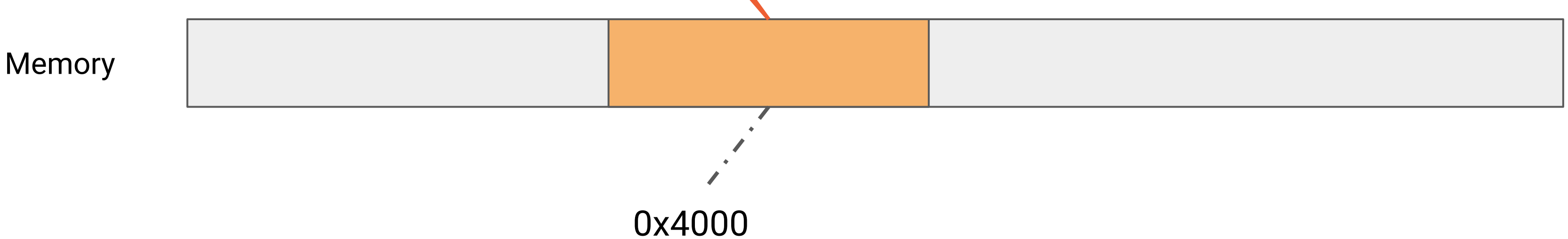


Heap

Let's consider the following

```java
public static void main(String[] args) {
   int x = 5;
   int[] array = new int[16];
   int[] alias = array;
   int y = x;
}
```

Although, **x** and **y** we will compare the **copy-semantics** between a **primitive type** and **a reference type.**

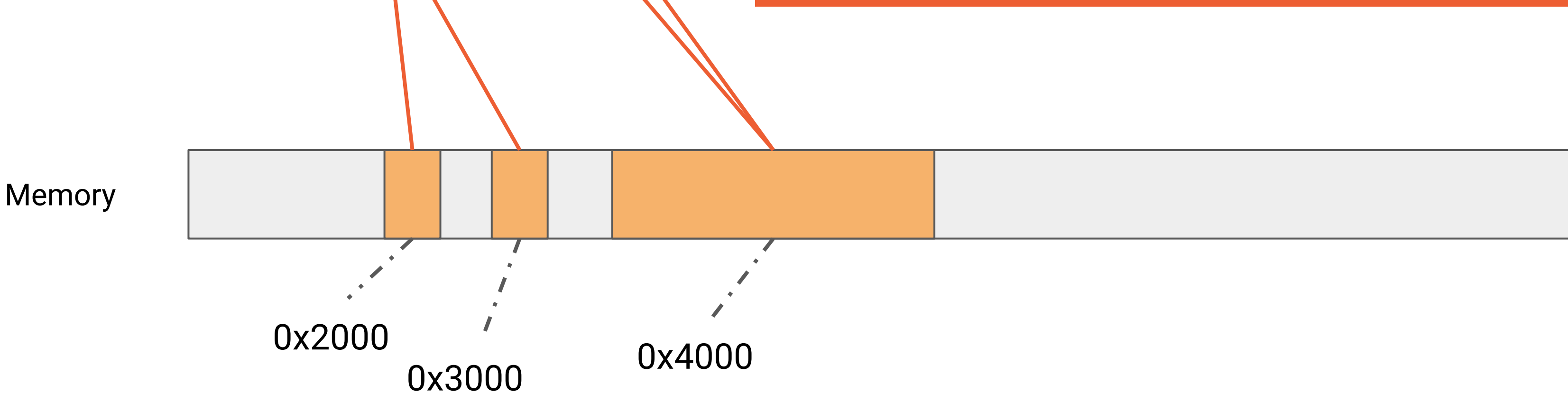


Heap

Let's consider the following

```java
public static void main(String[] args) {
  int x = 5;
  int[] array = new int[16];
  int[] alias = array;
  int y = x;
}
```

Although, **x** and **y are on the stack** we will compare the **copy-semantics** between a **primitive type** and **a reference type.** Primitive types will copy the value between instead of the address.



Garbage Collector

Java employs the use of a **garbage collector** to free memory. Any memory that is allocated using the the **new** keyword will be kept on the heap.

The garbage collector will subsequently free any allocation that no longer has a reference during execution.

The garbage collector will **stop-the-world** and act on allocations without a reference.

All I have known is garbage collecting? What was it like before?

What does manual look like?

Here's some C code!

```
person* create_object() {
  person* p = malloc(sizeof( person ));
  //things
  return p;
}

//<other bits of code>

void delete_person(person* p) {
  free(p);
}
```

Okay! I know you don't know C so here's a java-ized version

What does manual look like?

Here's some ~~C~~ Java code!

```
Person createPerson() {
  Person p = new Person();
  //things
  return p;
}

//<other bits of code>

void deletePerson(Person p) {
  delete p;
}
```

What does manual look like?

Here's some ~~C~~ Java code!

```
Person createPerson() {
  Person p = new Person();
  //things
  return p;
}

//<other bits of code>

void deletePerson(Person p) {
  delete p;
}
```

We have allocated a new Person object

What does manual look like?

Here’s some ~~C~~ Java code!

```
Person createPerson() {
  Person p = new Person();
  //things
  return p;
}

//<other bits of code>

void deletePerson(Person p) {
  delete p;
}
```

We have returned p to the calling method

What does manual look like?

Here's some ~~C~~ Java code!

```
Person createPerson() {
  Person p = new Person();
  //things
  return p;
}

//<other bits of code>

void deletePerson(Person p) {
  delete p;
}
```

Once the allocation has been used and the programmer wants it deleted it is sent to the **deletePerson** method

What does manual look like?

Here's some ~~C~~ Java code!

```
Person createPerson() {
  Person p = new Person();
  //things
  return p;
}

//<other bits of code>

void deletePerson(Person p) {
  delete p;
}
```

At this point allocation is deleted

Scenario 1: Forgot to free

When we forget to deallocate:

```
Person createPerson() {
  Person p = new Person();
  return p;
}

//<other bits of code>

void main(String[] args) {
  Person p = createPerson();
  someWork(p);
  p = createPerson();
```

Uhoh! We didn't deallocate the previous one!

```

}
```

How does the garbage collector handle this?

Scenario 1: Forgot to free

When we forget to deallocate:

```
Person createPerson() {
  Person p = new Person();
  return p;
}

//<other bits of code>

void main(String[] args) {
  Person p = createPerson();
  someWork(p);
  p = createPerson();

}
```

Since the original allocation has no references to it, when the GC passes it, it will be marked for deletion.

Scenario 2: Oops! I thought it was still allocated

```
Person createPerson() {
  Person p = new Person();
  return p;
}

//<other bits of code>

void main(String[] args) {
  Person p = createPerson();
  Person c = p;
  delete p;
  someWork(c);

}
```

Uhoh! If the memory has been freed then potentially we could be using memory that we don’t own anymore

Scenario 2: Oops! I thought it was still allocated

```
Person createPerson() {
  Person p = new Person();
  return p;
}

//<other bits of code>

void main(String[] args) {
  Person p = createPerson();
  Person c = p;
  delete p;
  someWork(c);

}
```

We don't do this anymore

Simply: There is no manual deallocation

Why?

Shouldn’t you just be more careful?

It’s about simplifying writing software for everyone. It allows you as a programmer to be able to write software without needing to worry about memory management.

We can minimise errors and make programming **safer** through this method.

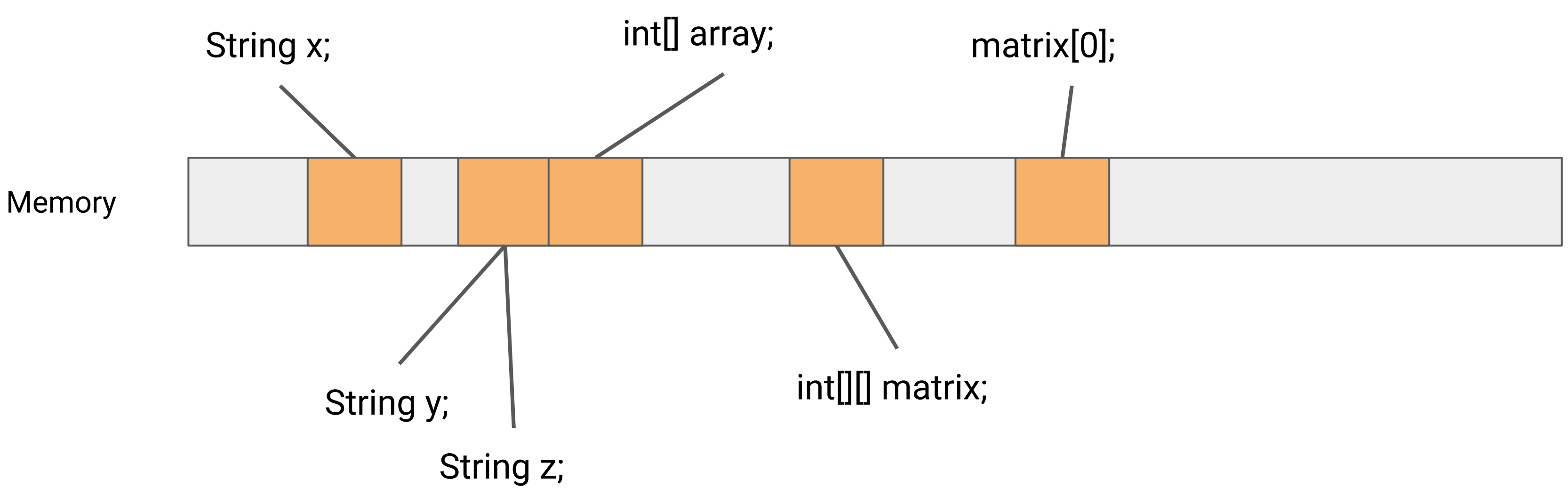
How does the garbage collector clean up?

When an **allocation** no longer has a reference it, the garbage collector will **mark** it for deletion.

How does the garbage collector clean up?

When an **allocation** no longer has a reference it, the garbage collector will **mark** it for deletion. This is when all references have gone out of scope.

Let's see how the following works:

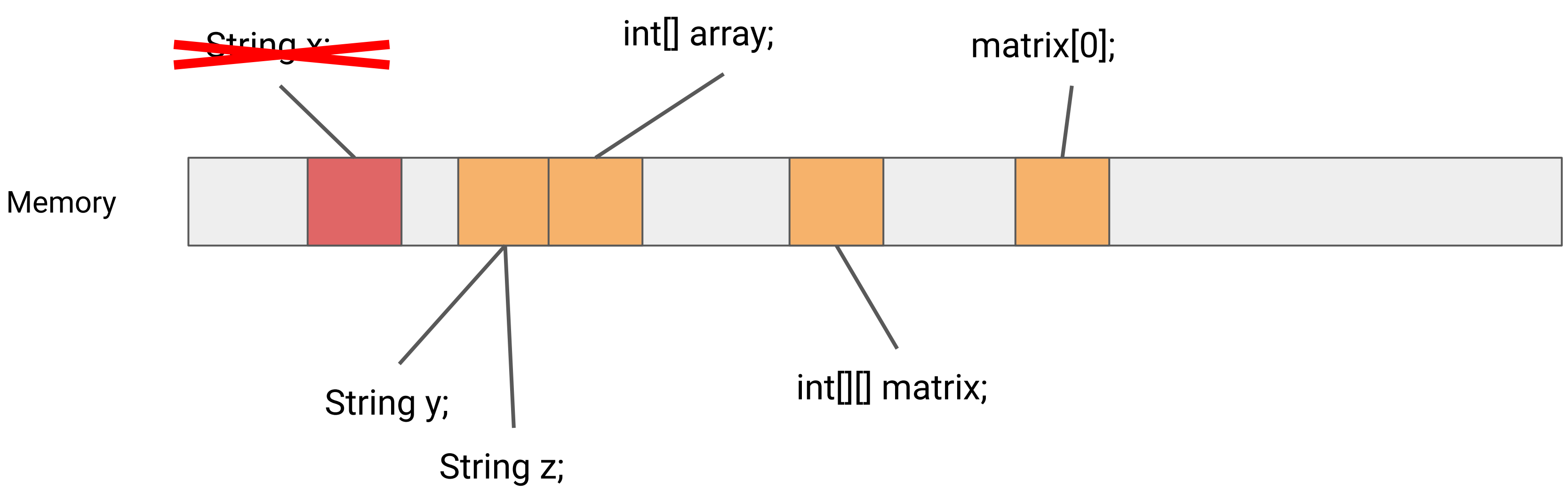


Garbage Collector

How does the garbage collector clean up?

When an **allocation** no longer has a reference it, the garbage collector will **mark** it for deletion. This is when all references have gone out of scope.

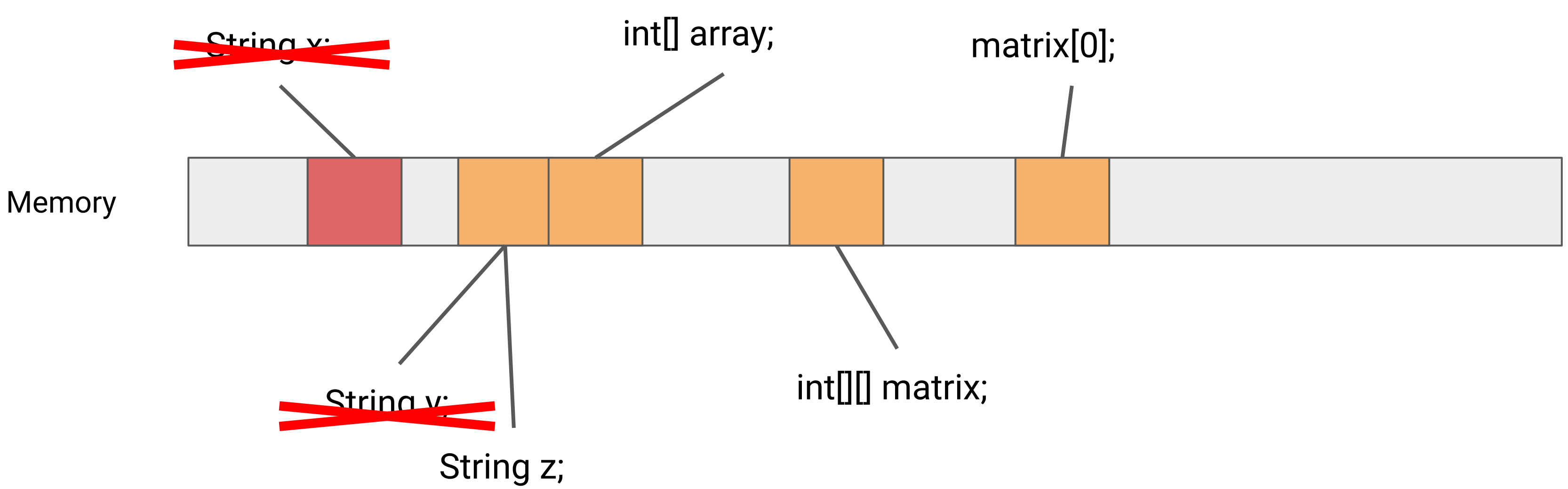
Let's see how the following works:



How does the garbage collector clean up?

When an **allocation** no longer has a reference it, the garbage collector will **mark** it for deletion. This is when all references have gone out of scope.

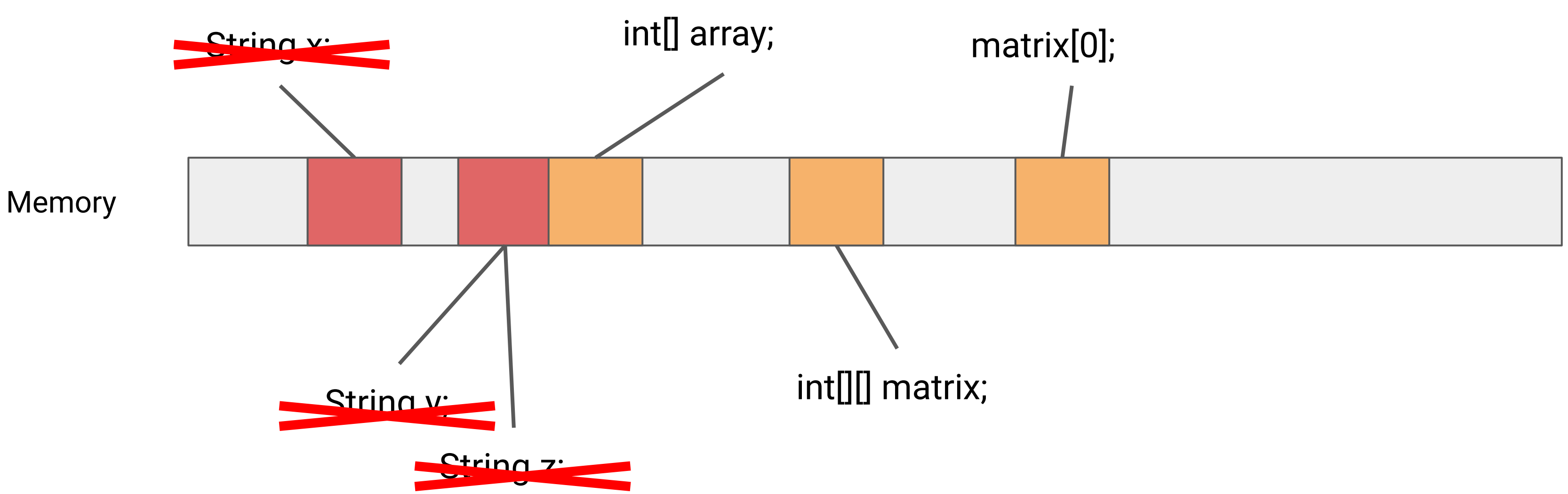
Let's see how the following works:



How does the garbage collector clean up?

When an **allocation** no longer has a reference it, the garbage collector will **mark** it for deletion. This is when all references have gone out of scope.

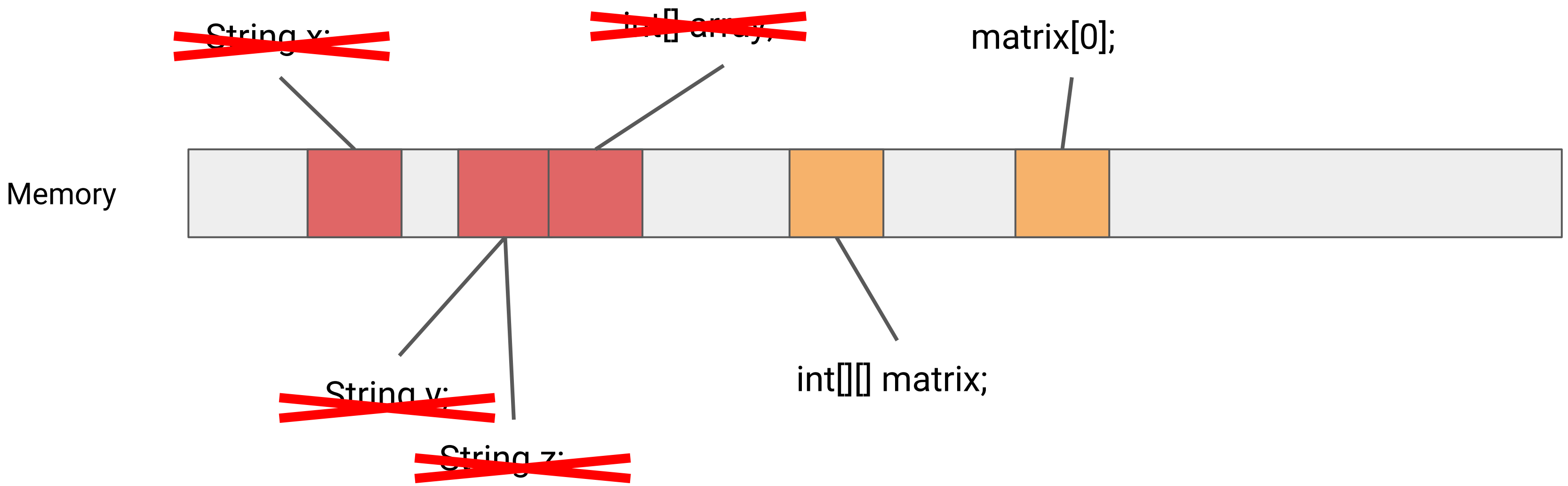
Let's see how the following works:



How does the garbage collector clean up?

When an **allocation** no longer has a reference it, the garbage collector will **mark** it for deletion. This is when all references have gone out of scope.

Let's see how the following works:

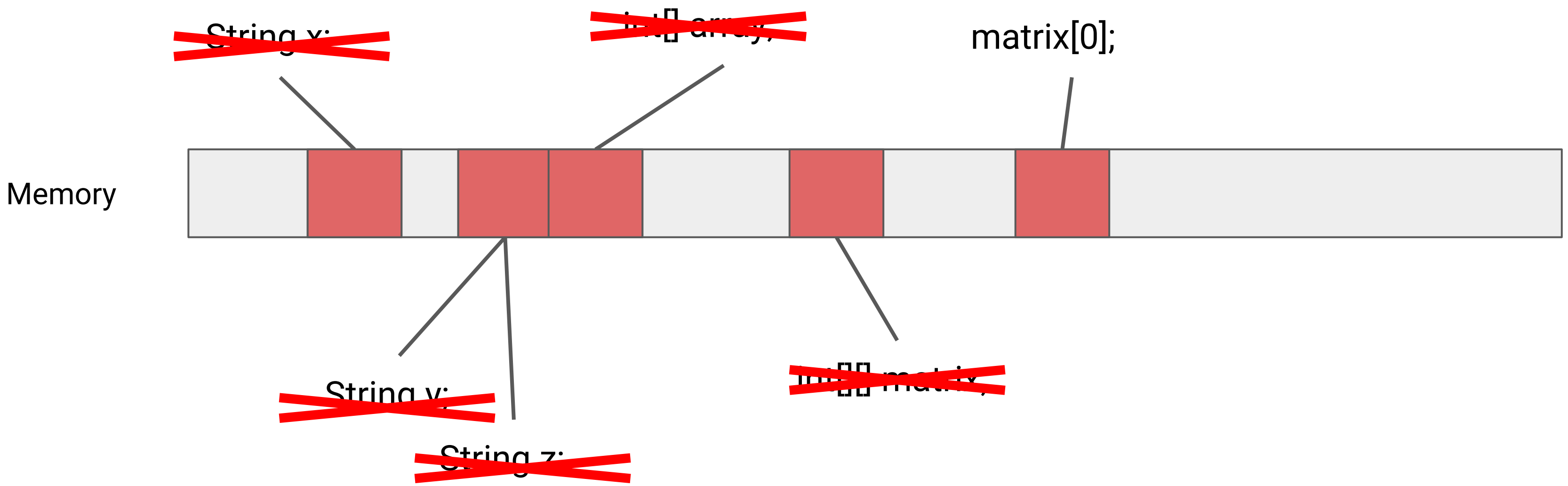


Garbage Collector

How does the garbage collector clean up?

When an **allocation** no longer has a reference it, the garbage collector will **mark** it for deletion. This is when all references have gone out of scope.

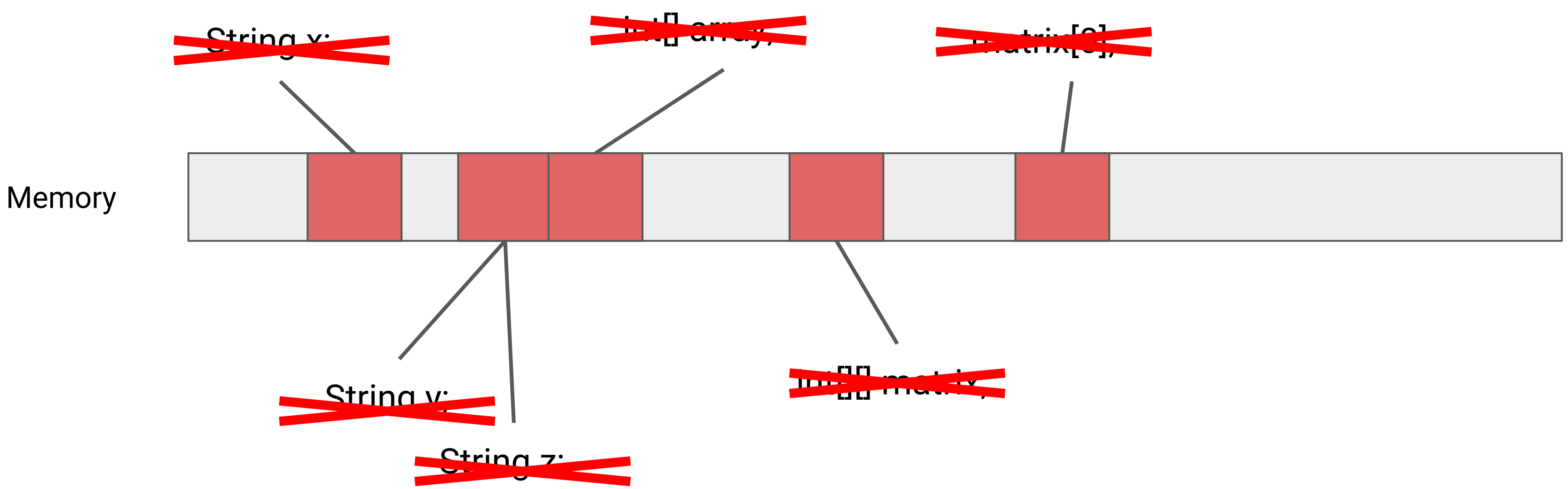
Let's see how the following works:



How does the garbage collector clean up?

When an **allocation** no longer has a reference it, the garbage collector will **mark** it for deletion. This is when all references have gone out of scope.

Let's see how the following works:



Simple references

See you next time!