5COSC001W - OBJECT ORIENTED PROGRAMMING

Lecture 4: Heap vs Stack - Garbage Collector - The static keyword - The final keyword - The Java class hierarchy

Dr Dimitris C. Dracopoulos
email: `d.dracopoulos@westminster.ac.uk`

1 Stack vs Heap

Memory in Java is divided into two areas, the *stack* and the *heap* (Figure 1).

- Local variables (primitive types variables inside methods), data related to method calls and returns (e.g. arguments passed to a function, return address), and intermediate calculations are allocated in the *stack*.
- Objects are allocated in the *heap*.

The amount of heap size can be controlled via the `-Xms` and `-Xmx` command options of the `java` utility. The `java` utility starts an instance of the Java Virtual Machine and executes a program within it.

Data allocated in the heap (objects) live independent of the scope in which they were allocated. For example, an object created inside a method, exists even when the execution of the method terminates.

Example:

```
public class ObjectLifetime {
    StringBuffer st;

    public int foo() {
        int i = 8;

        /* object created here exists outside the method, as long
           as there is still a reference to it */
```

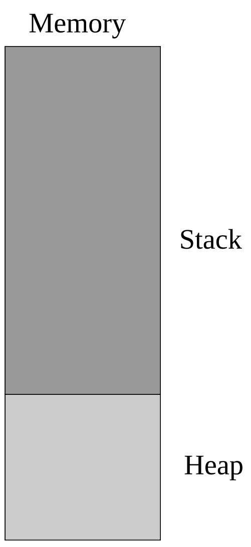


Figure 1: Memory in Java is allocated either in the stack (local variables which are primitive types) or in the heap (objects).

```
        StringBuffer a = new StringBuffer("b1");
        st = a;

        return i;
    }

    public static void main(String[] args) {
        ObjectLifetime ol = new ObjectLifetime();

        // call method foo, on object referenced by ol
        int k = ol.foo();

        // update the value in object pointed to by ol
        ol.st.append("5");

        System.out.println("Object referenced by st: " + ol.st);
    }
}
```

In the above example, the `StringBuffer` object created inside method `foo()` exists even when the method returns. This is because a reference to an object (via the `st` variable), still exists outside the method. As long as there are still references to an object, the object exists in the heap.

Local variable `i` defined inside the method is allocated in the stack, and it does not exist outside the method, even if the value of `i` is returned by the method. Recall that primitive type variables are copied by value, while object reference variables are copied by reference. What happens in the above example is illustrated in Figure 2.

When the `ObjectLifetime` code is run, it displays:

```
Object referenced by st contains: b15
```

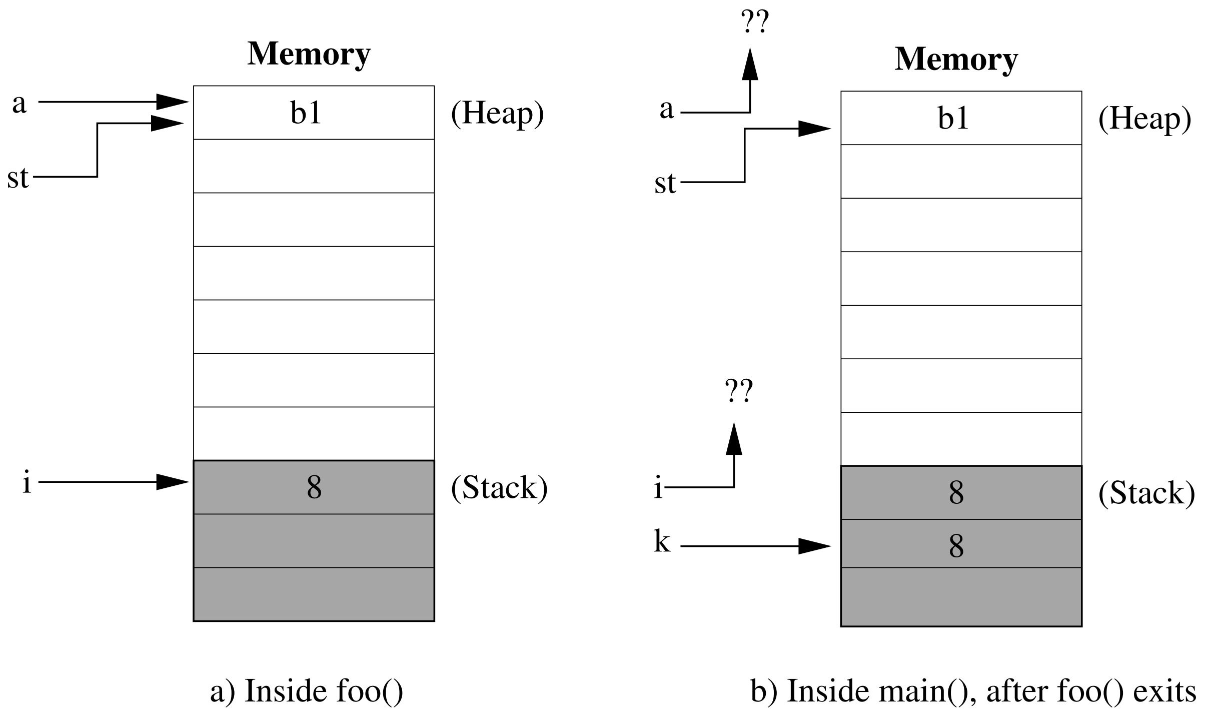


Figure 2: The lifetime of objects, as shown in the `ObjectLifetime` example. Objects created inside methods still exist outside of them, assuming there are references to the object outside the method.

2 The Garbage Collector

It is necessary to free data which are no longer needed from memory. This will ensure that memory will not become full, and there is always available space to allocate more data. In programming languages there are two ways to deallocate (free) memory:

- The programmer deallocates memory explicitly (C, C++).
- The system is responsible to free (recycle) unused memory. This is done using a process called as *garbage collector* (Java).

The Java garbage collector frees an object from the heap, only when there are no more references to that object.

- The caveat is that the programmer cannot control when the garbage collector starts its execution. This implies that even objects which have no references will remain in memory (heap), until the Java Virtual Machine (JVM) decides to execute the garbage collector.

  There is no guarantee that the garbage collector will run during the lifetime of a program. The JVM will run the garbage collector only when it thinks it is appropriate, e.g. when the available heap size goes low.

Example:

```
class Book {
    private String colour;
}

public class GarbageCollectorExample {
    public static void main(String[] args) {
        Book b1 = new Book();  // Book object created
        Book b2 = b1;  // 2 references to the same Book object
        Book b3 = b2;  // 3 references to the same Book object

        /* After the following statement:
           3 references (b1, b2, b3) to the first Book object -
           1 reference to the second book object (b4) */
        Book b4 = new Book();

        b1 = null;  // 2 references to first object
        b2 = null;  // 1 reference to first object
        b3 = null;  // 0 references to first object - candidate for garbage collection
    }
}
```

Figure 3 shows what happens in the above code. As soon as there are zero references to the `Book` object created in the first line of `main()`, the object becomes a candidate to be recycled (freed) by the garbage collector. This will happen (the memory addresses occupied by the object will be released to the pool of free heap space), when the garbage collector runs next.

3 Static Methods

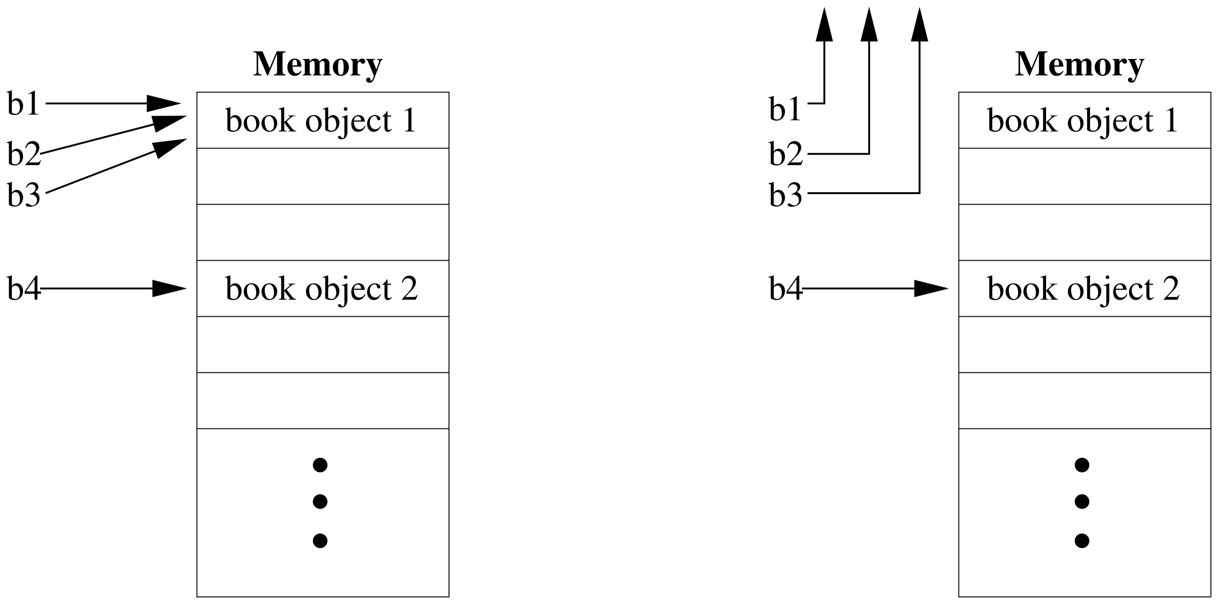
A method declared as `static` can be called on a class, without the need to create an object of a class. Static methods are sometimes called *class methods*.

```
class A {
    static int bar() {
        System.out.println("bar() called!");

        return 0;
    }
}

public class StaticMethodsExample {
    public static void main(String[] args) {
        A.bar();  // OK. No need to create an object

        A a1= new A();
        a1.bar();  // OK too!
```



(a) After:
```
Book b1 = new Book();
b2 = b1;
b3 = b2
Book b4 = new Book();
```

(b) After:
```
b1 = null;
b2 = null;
b3 = null;
```

Figure 3: What happens during the execution of the `GarbageCollectorExample` code. The object created in the first line of `main` will become a candidate to be released, as soon as there are no references to it, i.e. in stage (b).

```
    }
}
```

As the example indicates, a `static` method can also be called on an object of the class.

Methods should be static when the function they perform is not associated with an individual object, but rather with the class itself. It is common to have static methods in utility classes. For example, consider a class `Calculator` providing methods `add(int, int)`, `substract(int, int)`, etc. Such methods do not depend on objects of the class, but they are used to perform arithmetic operations between their arguments. Therefore, they should be declared as static.

4 Static Data

Class fields which are declared `static`, are shared among all objects of the class. This means, that a single instance of the field will be created, independent of the number of objects of the class. This is illustrated in the example below, and shown in Figure 4.

```
class Book {
    static int numberOfBooks;
    int numberOfPages;

    Book(int pages) {
        ++numberOfBooks;
        numberOfPages = pages;
    }
}

public class StaticFieldsExample {
    public static void main(String[] args) {
        Book b1 = new Book(10);
        Book b2 = new Book(20);
        Book b3 = new Book(5);

        System.out.println("b1.numberOfBooks: " + b1.numberOfBooks);
        System.out.println("b2.numberOfBooks: " + b2.numberOfBooks);
        System.out.println("b3.numberOfBooks: " + b3.numberOfBooks);
        System.out.println("Book.numberOfBooks: " + Book.numberOfBooks);

        System.out.println("b1.numberOfPages :" + b1.numberOfPages);
        System.out.println("b2.numberOfPages :" + b2.numberOfPages);
        System.out.println("b3.numberOfPages :" + b3.numberOfPages);
        // System.out.println(Book.numberOfPages);  // Error!
    }
}
```

The output of the `StaticFieldsExample` is:

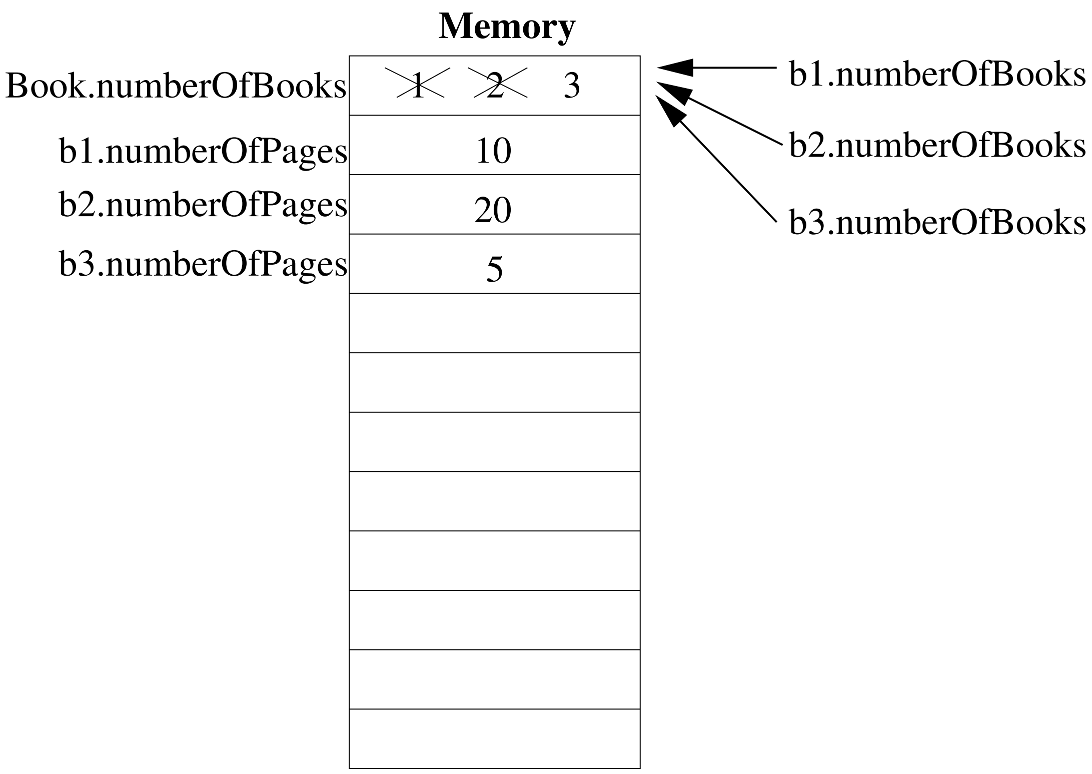


Figure 4: Static fields are shared among all objects of a class. Non-static fields (instance fields) are unique for each object of the class.

```
b1.numberOfBooks: 3
b2.numberOfBooks: 3
b3.numberOfBooks: 3
Book.numberOfBooks: 3
b1.numberOfPages :10
b2.numberOfPages :20
b3.numberOfPages :5
```

5 Final Classes and Methods

The keyword `final` can be used for the following:

- *final variables*: these are constants, i.e. their value cannot be changed.
- *final classes*: these classes cannot be used as a base for another class. You cannot inherit from a final class.
- *final methods*: these cannot be overridden in a subclass.

Example:

```
final class Patent {
}

class MyPatent extends Patent { }  // Error! Cannot inherit from final class

class Calculator {
    final int increaseByOne(int x) {
        return x+1;
    }
}


class MyCalculator extends Calculator {
    int increaseByOne(int x) {  // Error! Cannot override final method
        return x+5;
    }
}


class MyInteger {
    public int i;
}


public class FinalExample {
    public static void main(String[] args) {
        final MyInteger m1 = new MyInteger();
        m1 = new MyInteger();  // Error! m1 is constant
        m1.i = 5; // OK
    }
}
```

As illustrated in the example above, a `final` object reference variable cannot be changed so as to point to a different variable. However, the contents of the object that the variable is pointing to, can be changed (assuming that the object itself is mutable).

6 The Java class hierarchy

The parent of every Java class (whether user defined or library) is class `Object`. This means that all the methods found in `Object` are inherited by every class.

It should be noted that as Java does not support multiple inheritance (i.e. a class can only inherit from a single class directly), the above implies that `Object` is the direct or indirect parent class of any Java class. For example, in Figure 5 `Object` is an indirect parent class of `ArrayList` (its direct parent class is `AbstractList`).

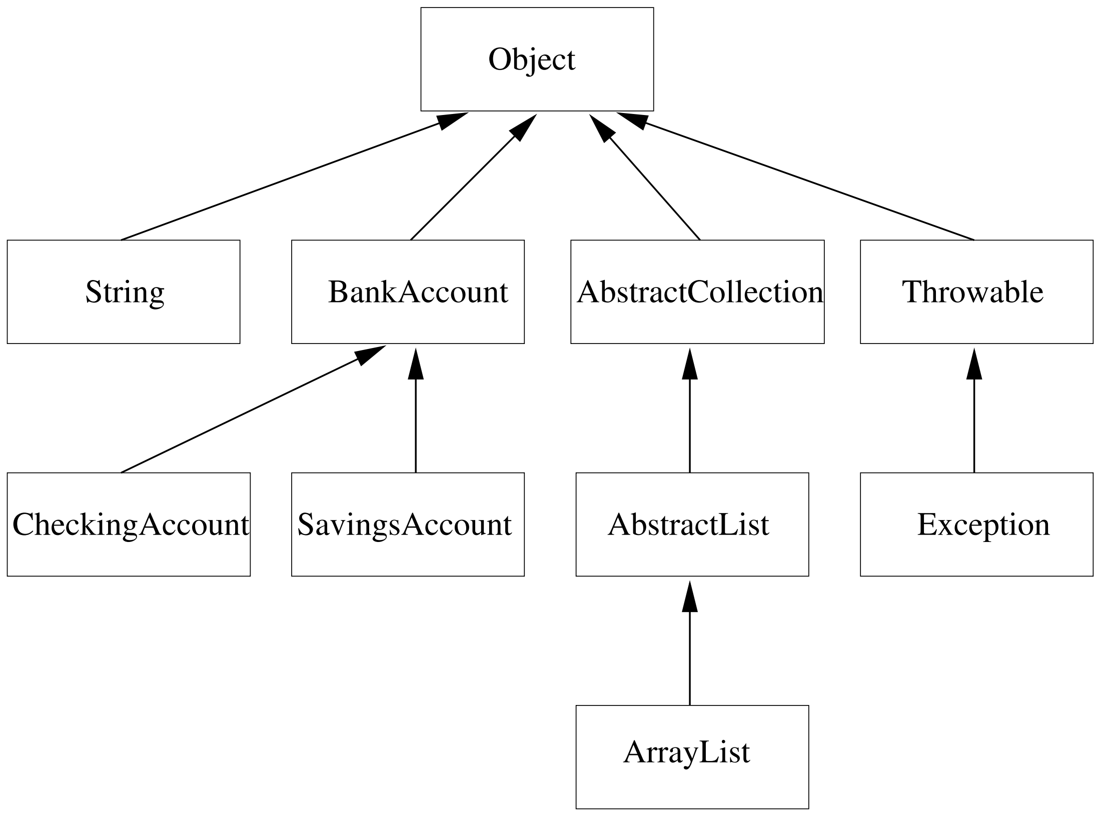


Figure 5: The parent of every Java class is class `Object`.

7 Order of Object Initialisation

The following series of events occur during the creation of an object:

1. Sufficient memory is allocated in the heap to hold the object. (including all instance variables specific to the object and all instance variables of its superclass - and since the superclass inherits all instance fields from its superclass enough memory is allocated to include the data of its super-superclass, etc.). It should be noted that any user specified initialisation values are not assigned to the fields at this point (see Step 4).
2. All instance variables of the object are initialised to their default values, i.e. all field objects to nulls, primitive numerics to zero and booleans to false.
3. The default constructor (i.e. the constructor with no arguments) of the direct superclass of the object is called. The constructor of the superclass will invoke the default constructor of its own superclass and so on, until the constructor of the parent class of all classes `java.lang.Object` is called.
4. The user specified initialisation values of the instance variables are assigned to them and any initialisation blocks are executed.
5. The actual body of the constructor is executed.

Example:

The following example illustrates the sequential execution of the above steps during the creation of an object of class `TimeTravel`:

```
class SpaceTravel extends Travel {
    private float distance = 5000;

    SpaceTravel() {
        System.out.println("SpaceTravel() constructor!");
    }
}

class TimeTravel extends SpaceTravel {
    private String timeElapsed = "0 years";

    // initialisation block
    {
        System.out.println("Initialisation block");
    }


    TimeTravel() {
        System.out.println("TimeTravel() constructor!");
    }
}

public class Travel {
    Travel() {
        System.out.println("Travel() constructor!");
    }

    public static void main(String[] args) {
        TimeTravel t = new TimeTravel();
    }
}
```

When the example is run it produces the output:

```
Travel() constructor!
SpaceTravel() constructor!
Initialisation block
TimeTravel() constructor!
```

The sequence of events which take place when the statement `TimeTravel t = new TimeTravel()` is encountered:

1. Memory is allocated in the heap to hold an object of class `TimeTravel`.

2. The object's instance variables, i.e. `timeElapsed` are initialised to their default values, so `timeElapsed` is assigned the `null` value.
3. The constructor of `TimeTravel` is called. Before doing anything else, this calls the default constructor of its superclass `SpaceTravel`.
4. The instance variable of `SpaceTravel` `distance` is initialised to its default value 0.
5. The constructor of `SpaceTravel` calls its superclass default constructor `Travel` and the body of that constructor is executed printing the string "Travel() constructor!".
6. The constructor of `Travel` returns and the instance variable `distance` is initialised to 5000.
7. The body of the constructor `SpaceTravel` is executed, printing the message "SpaceTravel() constructor!".
8. The constructor of `SpaceTravel` returns, and the instance variable of `TimeTravel` timeElapsed is initialised to the string "0 years".
9. The initialisation block in `TimeTravel` is executed, printing the message "Initialisation block".
10. The body of the constructor of `TimeTravel` is executed, producing the string "TimeTravel() constructor!".