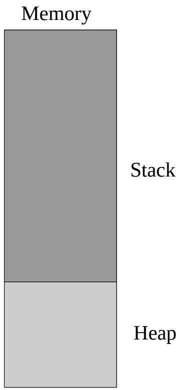
5COSC019W - OBJECT ORIENTED PROGRAMMING
Lecture 4: Heap vs Stack - Garbage Collector - The static keyword - The final keyword - The Java class hierarchy

Dr Dimitris C. Dracopoulos

Stack vs Heap

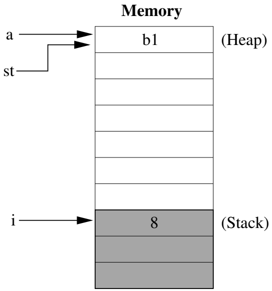
- Local variables (primitive types variables inside methods), data related to method calls and returns (e.g. arguments passed to a function, return address), and intermediate calculations are allocated in the *stack*.
- Objects are allocated in the *heap*.

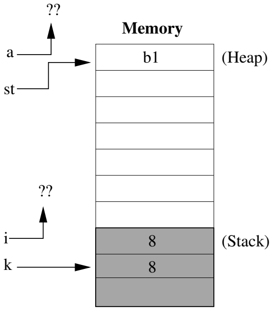


Heap Allocation

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 lon
           as there is still a reference to it */
        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);
    }
}
```





The Garbage Collector

In programming languages there are two ways to deallocate (free) memory:

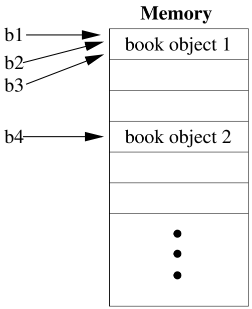
- The programmer deallocates memory explicitly (C, C++).
- The system is responsible to free (recycle) unused memory. This is done using a process called as *garbage collector* (Java).

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

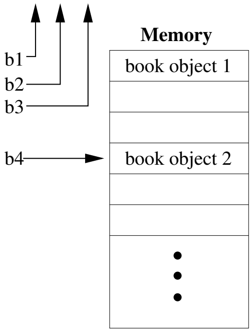
- *The caveat is that the programmer cannot control when the garbage collector starts its execution. This implies that objects which have no references will remain in memory (heap) until the Java Virtual Machine (JVM) decides to execute the garbage collector.*

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
    }
}
```



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



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

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*.

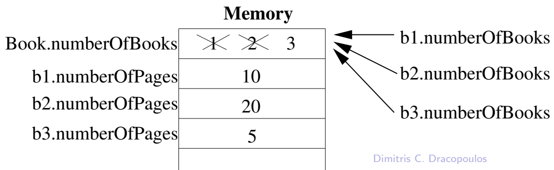
```
class A {
    static int bar() {
        System.out.println("bar() called!");

        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!
    }
}
```

Static Data

Class fields which are `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.

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

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

```java
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.numberOfB

      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:

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

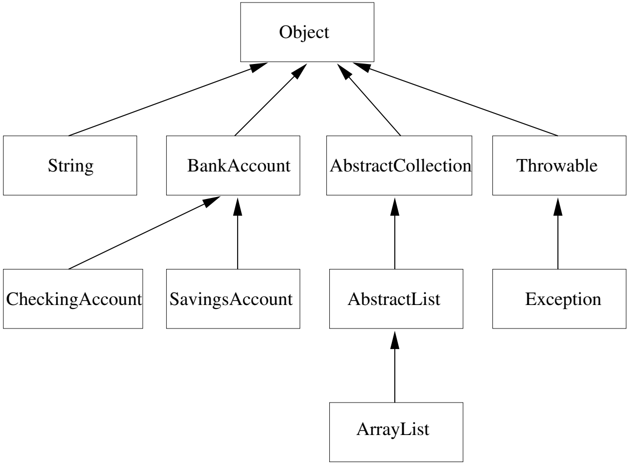
Example:

```
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
        return x+5;             //        final method
    }
}
```

```java
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
    }
}
```

The Java class hierarchy

The parent of every Java class is class `Object`.



Order of Object Initialisation

The following happen during the creation of an object:

1. Sufficient memory is allocated in the heap to hold the object.
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 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:

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

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 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!
```