Contents

1. Introduction
2. Types and Variables
3. Statements and Control Flow
4. Reading Input
5. Classes and Objects
6. Arrays
7. Methods
8. Scope and Lifetime
9. Utility classes
10. Introduction to Object-Oriented Analysis and Design
Chapter 8: Scope and Lifetime

After this chapter you will be able to:

- Apply variable scoping issues
- Describe the function of the stack and heap
- Understand how a garbage collector functions
Method Scope

• Each method has its own local scope

```java
public class Scope {
    public static void main (String [] args) {
        int x = 1;
    }

    public static void foo () {
        System.out.println("x = " + x);
    }
}
```

• Cannot refer to x in foo
Method Scope

• Can declare the same variable name in different methods

```java
public class Scope {
    public static void main (String [] args) {
        int x = 1;
    }

    public static void foo () {
        int x = 2;
        System.out.println("x = " + x);
    }
}
```
Inner Scopes

• Can have nested scopes inside methods

```java
public static void foo () {
    int x = 2;
    System.out.println("x = " + x);
    {
        int y = 4;
        // can refer to y here
    }
    // cannot refer to y here
}
```
Loop Scope

• Variables declared inside a loop are local to the scope of the loop

```java
public static void foo () {
    int x = 2;
    System.out.println("x = " + x);

    for (int i = 1; i < 10; i++) {
        int j = i + 1;
        System.out.println("i = " + i);
        System.out.println("j = " + j);
    }
    // cannot refer to i, j here
}
```
Parameters

- Parameters are only visible in method scope

```java
public class Params {
    public static void main (String [] args) {
        System.out.println("6 * 8 = " + mult(6,8));
        // cannot refer to x, y here
    }

    public static int mult (int x, int y) {
        return x * y;
    }
}
```
Stack

• A stack is a data structure whose behavior is similar to a stack of plates in a cafeteria
• Items can be **pushed** onto the stack
• Items can also be **popped** off of the stack
The Run-Time Stack

• All local variables are *alive* on the **stack** on scope entry. On scope exit local variables are *dead* (inaccessible)

```java
public static int mult (int x, int y) {
    int z = 1;
    for (int i = y; y >= 0; y--){
        z *= x;
    }
    return z;
}
```

- **x, y, z** alive
- **i** alive
- **i** dead
- **x, y, z** dead
The Run-Time Stack

• Management of the stack is handled transparently to the programmer

• The allocation and deallocation happens automatically
Heap

- The **heap** is another area of memory where objects live.
- Objects are allocated in the heap by calling **new**.
- Objects no longer **accessible** by the program are deallocated automatically by the **garbage collector (GC)**.

The GC helps reduce the possibility that a program has a memory leak.
Stack and Heap

- Local variables in stack, objects in heap

```java
{  
    String s = "hello";
    int [] x = {1,2,3};
}
```

![Diagram showing Stack and Heap with variables and objects]
Scope Exit

- On scope exit local variables are no longer alive

```java
{  
    String s = "hello";
    int [] x = {1,2,3};
}
```

Stack

Heap

The GC will later find there are now no references to these two objects

1,2,3

"hello"
Methods Returning Objects

• A method may return an object

```java
public class ReturnObject {

    public static void main (String [] args) {
        getArray();
    }

    public static int [] getArray () {
        int [] x = {1,2,3};
        return x;
    }
}
```
Methods Returning Objects

- Since the object returned by `getArray` is inaccessible it will be reclaimed by the GC

```java
public static void main (String [] args) {
    getArray();
}
```

Stack

Heap

| 1,2,3 |

The GC will later find there are no references to this object
Methods Returning Objects

• Now the returned object is accessible via the variable x

```java
public static void main (String [] args) {
    int [] x = getArray();
}
```

• The GC will not reclaim the object until x is no longer in scope

Stack

Heap

1,2,3
GC

• The GC will not reclaim as long as an object is accessible

```java
public static void main (String [] args) {
    int [] x;
    {
        int [] y = getArray();
        x = y;
    }
}
```

• Question: Is it legal for the GC to reclaim at this point?
Answer

• No, since the object is still accessible via x

```java
public static void main (String [] args) {
    int [] x;
    {
        int [] y = getArray();
        x = y;
    }
}
```

Stack

![Stack Diagram]

Heap

![Heap Diagram]
Calling GC

• The programmer may explicitly call the GC

```java
public static void main (String [] args) {

    int [] y = getArray();
}
System.gc();
}
```

• `System.gc` can be used when the programmer knows there is garbage on the heap
Java™ vs C++

- All objects live in the heap
- GC deallocates unused memory in heap

- Objects live in either stack or heap
- No GC means programmer must deallocate unused memory in heap
References

- Garbage Collection: Algorithms for Automatic Dynamic Memory Management
  by Richard Jones and Rafael Lins
It’s Exercise Time