CENG 443

Introduction to Object-Oriented Programming Languages and Systems

Java Garbage Collection

What is GC?

- The Java virtual machine's heap stores all objects created by a running Java application
- Objects are created by the program through *new* keyword, but never freed explicitly by the program
  - No need to call free()
- Garbage collection is the process of automatically freeing objects that are no longer needed
- An object is determined to be “no longer needed” when there is no other object referencing to it
  - Each object has a reference counter - when it becomes 0, it means there is no other object referencing to it

Advantages of GC

- Programmer is free from memory management
  - Less error prone code
- System cannot crash due to memory management
  - More reliable application
  - Memory-leak is still possible
  - You can use memory profiler to find out where memory-leaking code is located

Disadvantages of GC

- GC could add overhead
  - Many GC schemes are focused on minimizing GC overhead
- GC can occur in an non-deterministic way

When Does GC Occur?

- JVM performs GC when it determines the amount of free heap space is below a threshold
  - This threshold can be set when a Java application is run
- You explicitly request garbage collection

How Does JVM Perform GC?

- Find the objects no longer reachable

  -
    ```
    MyObject obj = new MyObject();
    obj = null;
    ```
- Find the objects references copied to other reference

  -
    ```
    MyObject obj1 = new MyObject();
    MyObject obj2 = new MyObject();
    obj2 = obj1;
    ```

the instance (object) pointed by (referenced by) obj2 is not reachable and available for garbage collection.

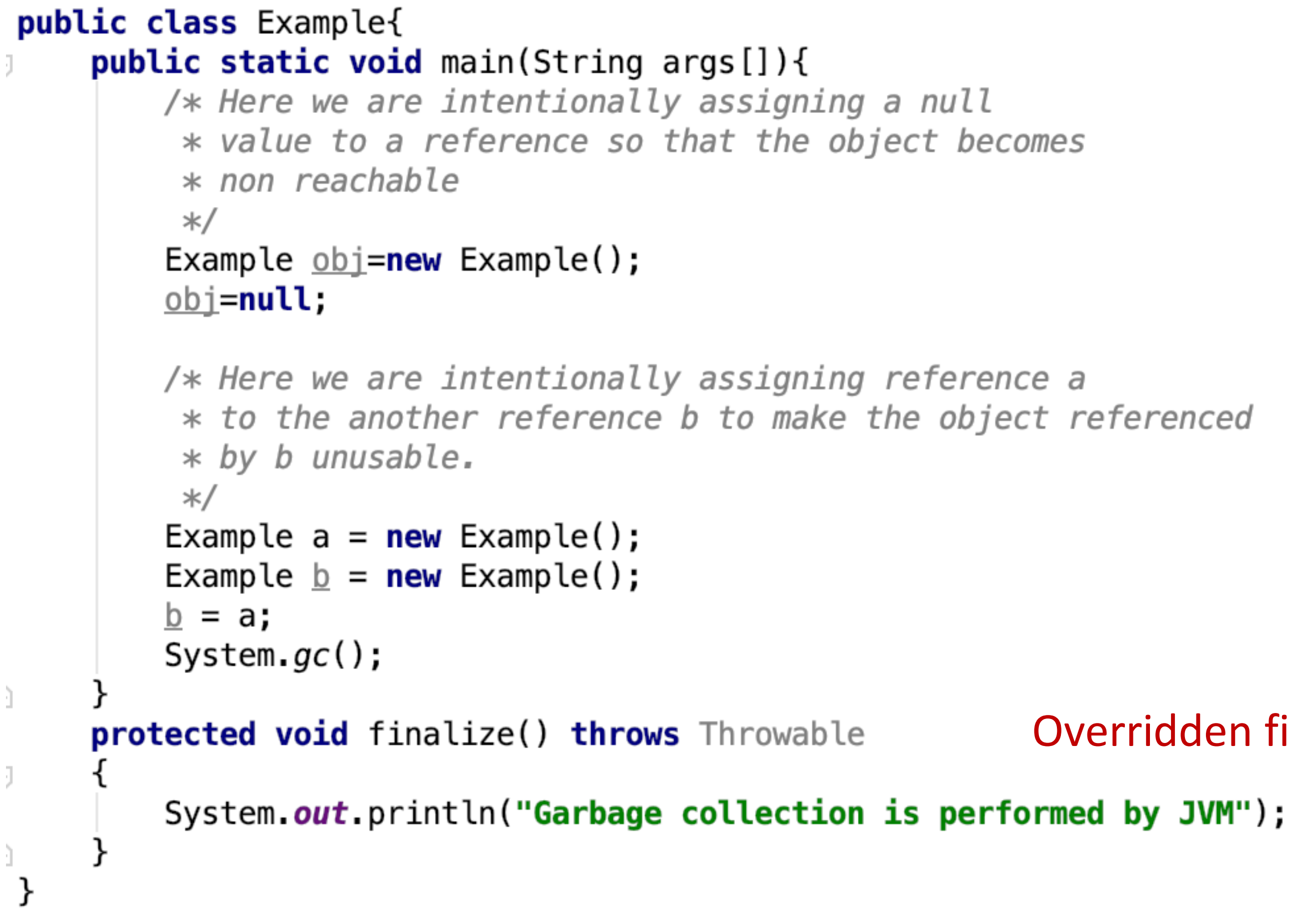
How Does JVM Perform GC?

- The garbage collector must somehow determine which objects are no longer referenced and make available the heap space occupied by such unreferenced objects.
- The simplest and most crude scheme is to keep reference counter to each object
- There are many different schemes - years of research

GC Related Java API

- *finalize() method* in Object class
  - Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
- gc() method in System class
  - Runs the garbage collector.
  - Calling the gc method suggests that the Java Virtual Machine expend effort toward recycling unused objects in order to make the memory they currently occupy available for quick reuse.
  - When control returns from the method call, the Java Virtual Machine has made a best effort to reclaim space from all discarded objects.

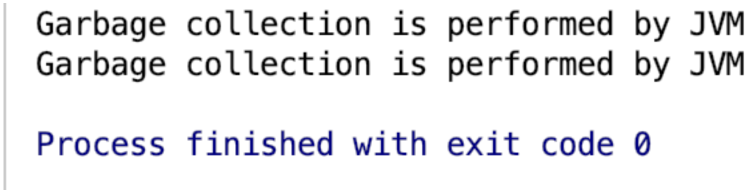
Example



```
public class Example{
    public static void main(String args[]){
        /* Here we are intentionally assigning a null
         * value to a reference so that the object becomes
         * non reachable
         */
        Example obj=new Example();
        obj=null;

        /* Here we are intentionally assigning reference a
         * to the another reference b to make the object referenced
         * by b unusable.
         */
        Example a = new Example();
        Example b = new Example();
        b = a;
        System.gc();
    }
    protected void finalize() throws Throwable
    {
        System.out.println("Garbage collection is performed by JVM");
    }
}
```

Overridden finalize() method



```
Garbage collection is performed by JVM
Garbage collection is performed by JVM

Process finished with exit code 0
```

Item 8: Avoid finalizers and cleaners

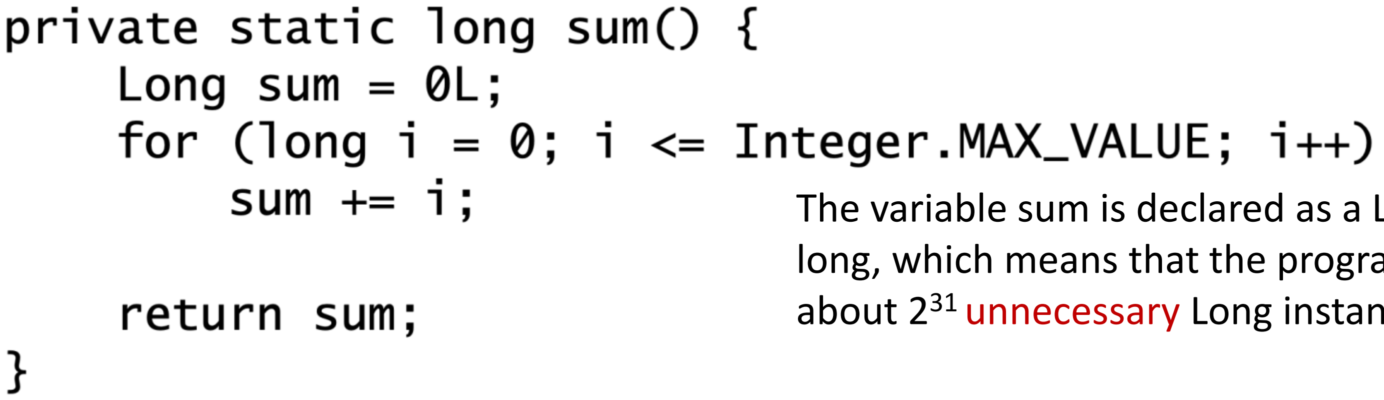
- Finalizers are unpredictable, often dangerous, and generally unnecessary.
- As of Java 9, finalizers have been deprecated, but they are still being used by the Java libraries.
- The Java 9 replacement for finalizers is *cleaners*. **Cleaners are less dangerous than finalizers, but still unpredictable, slow, and generally unnecessary.**
- One shortcoming of finalizers and cleaners is that there is no guarantee they'll be executed promptly
- It can take arbitrarily long between the time that an object becomes unreachable and the time its finalizer or cleaner runs.
    - you should **never do anything time-critical in a finalizer or cleaner.**

Item 6: Avoid creating unnecessary objects

- String s = new String(“hello”);
- String s = “hello”;

The argument to the String constructor (“hello”) is itself a String instance !

Can you spot the problem?



```
private static long sum() {
    Long sum = 0L;
    for (long i = 0; i <= Integer.MAX_VALUE; i++)
        sum += i;

    return sum;
}
```

The variable sum is declared as a Long instead of a long, which means that the program constructs about 2^{31} unnecessary Long instances!

prefer primitives to boxed primitives, and watch out for unintentional autoboxing

Item 7: Eliminate obsolete object references

- GC does not mean that you don’t have to think about memory management
- Memory Leak can happen and in fact happens all the time

Can you find the memory leak?

```
public class Stack {
    private Object[] elements;
    private int size = 0;
    private static final int DEFAULT_INITIAL_CAPACITY = 16;

    public Stack() {
        elements = new Object[DEFAULT_INITIAL_CAPACITY];
    }

    public void push(Object e) {
        ensureCapacity();
        elements[size++] = e;
    }
     public Object pop() {
         if (size == 0)
             throw new EmptyStackException();
         Object result = elements[--size];
         elements[size] = null; // Eliminate obsolete reference
         return result;
     }
    /**
     * Ensure space for at least one more element, roughly
     * doubling the capacity each time the array needs to grow.
     */
    private void ensureCapacity() {
        if (elements.length == size)
            elements = Arrays.copyOf(elements, 2 * size + 1);
    }
}
```