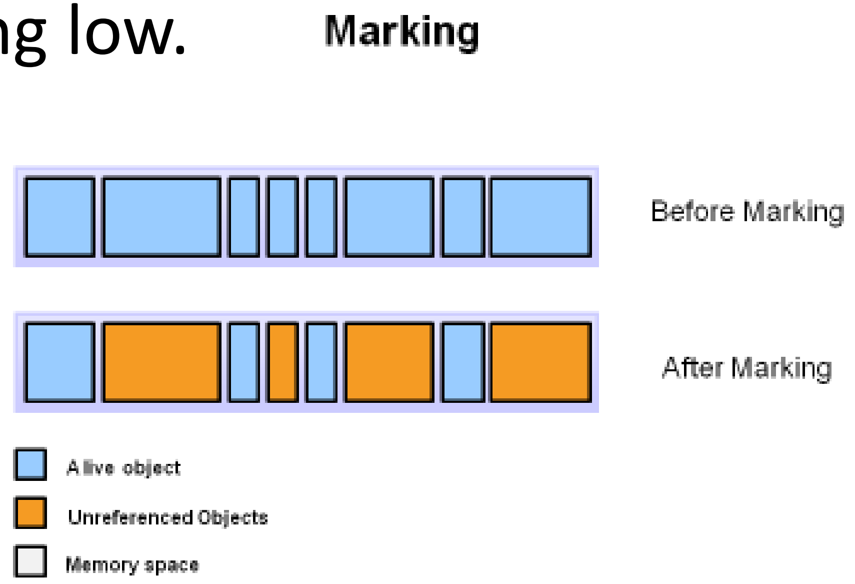
Garbage Collection

And Memory Leaks

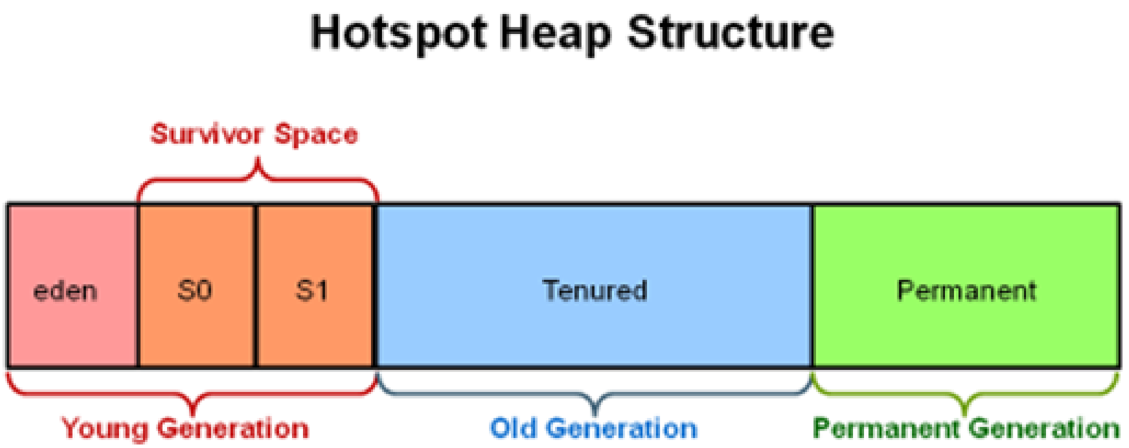
Automatic Garbage Collection

- Java automatically allocates and deallocates memory
- The garbage collector evaluates heap memory, identifies which objects are in use, and deletes unused object
  - If there is some reference to an object, that object is *in use*
- This process happens when memory is running low.



Generational Garbage Collection

- Most objects are created and quickly discarded
- Objects that are not quickly discarded are likely to stick around for a long time
- Taking advantage of this means the garbage collector does not have to check *all* objects for references during *every* garbage collection



Generational Garbage Collection

- New objects are put in the Young Generation part of the heap.
  - When this is full, there is a *minor* collection.
  - Some surviving objects are moved to the Old Generation.
- Long surviving objects are put into Old Generation.
  - These collections are *major* or *full* garbage collections and take longer.
  - Thus, these should be minimized.
- These are *stop the world* events (all threads are stopped).

- The Permanent Generation contains metadata, class definitions, etc.

Tuning

- You can customize the heap size (initial and maximum), the size of young generation, and the size of permanent generation.
- You can specify different garbage collectors, for example:
  - Serial
  - Parallel
  - Concurrent

- Often, tuning is done to reduce the *stop-the-worlds.*

MEMORY LEAKS

Memory Leaks

- Java does automatic garbage collection so there is no need to worry about memory management, right?
  - Not so fast!

- A *memory leak* can occur when we are done with an object but still have a reference to it.
  - In this case, the object will never be garbage collected.
  - It is an *obsolete reference*

Practice

- Review at the example from Effective Java. Can you spot the memory leak?

```java
import java.util.*;

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;
}
```

```java
public Object pop() {
     if (size == 0)
        throw new
          EmptyStackException();

     return elements[--size];
}


private void ensureCapacity() {
   if (elements.length == size)
      elements = Arrays.copyOf(
          elements, 2*size + 1);
}
}
```

Possible Signs of a Memory Leak

- Runs fine at first then slows
- Runs fine with small inputs, slows with larger inputs
- Expanding old-generation memory usage
- OutOfMemory errors

Monitoring Garbage Collection

- You can use –verbose:gc to have verbose garbage collection.
  - Add this as a runtime argument in your IDE.
- VisualVM comes with JDK
  - https://docs.oracle.com/javase/8/docs/technotes/guides/visualvm/intro.html
  - In the Java bin folder: jvisualvm
- Many other tools

Practice

- Review the MemoryLeakExample example.

Avoiding Memory Leaks

- Make sure objects are de-referenced when they are not needed
  - Assigning them to null
  - Assigning them to another reference
  - Letting objects fall out of scope
- However: nulling objects should be the exception, rather than the rule. (from Effective Java, Item 5)
  - Best to let objects become naturally dereferenced when they are out of scope
  - This is why you should declare objects to the narrowest scope possible!
- Whenever your class manages its own memory, be on the lookout for memory leaks.
- Be on the lookout for static variables.
  - Especially static variables that are collection classes!