Victoria University
of Wellington, New Zealand
Te Whare Wananga o te
Upoko o te Ika a Maui
Aotearoa

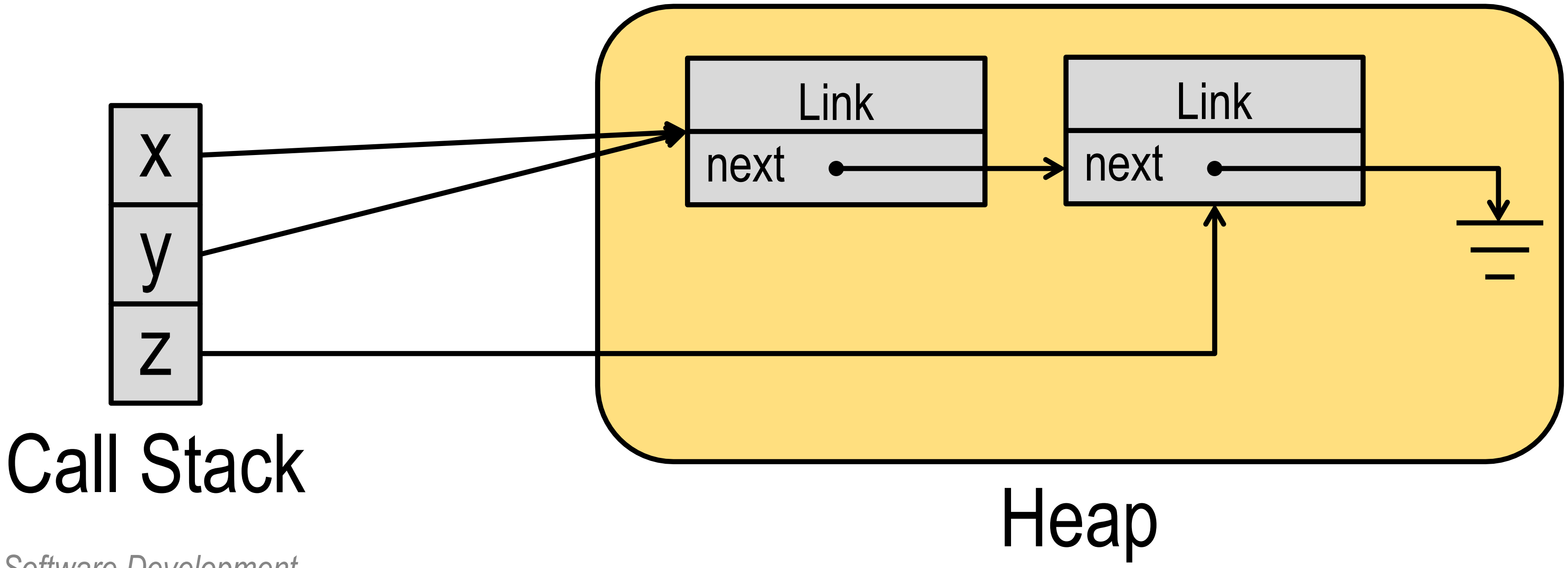
SWEN221:
Software Development

23: Memory Management and Garbage Collection

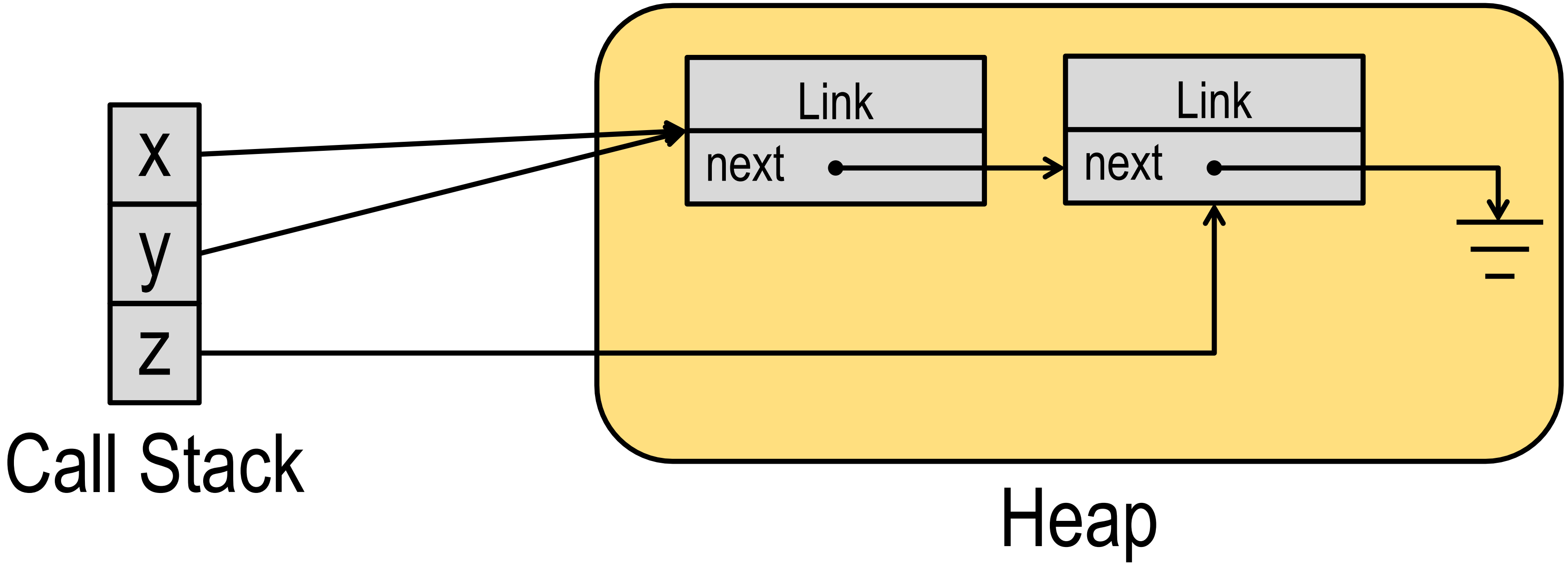
David J. Pearce & Nicholas Cameron & James Noble
Engineering and Computer Science, Victoria University

Objects and Memory (recap)

```
class Link {
  private Link next;
  public Link(Link next) { this.next = next; }
  public static void main(String[] args) {
    Link z = new Link(null);
    Link x = new Link(z);
    Link y = x;
}}
```



Objects and Memory (recap)



- Notes:
  - Variables x,y and z are **references**
  - Variables x and y **point** to same object
  - Two instances of Link exist in **heap**

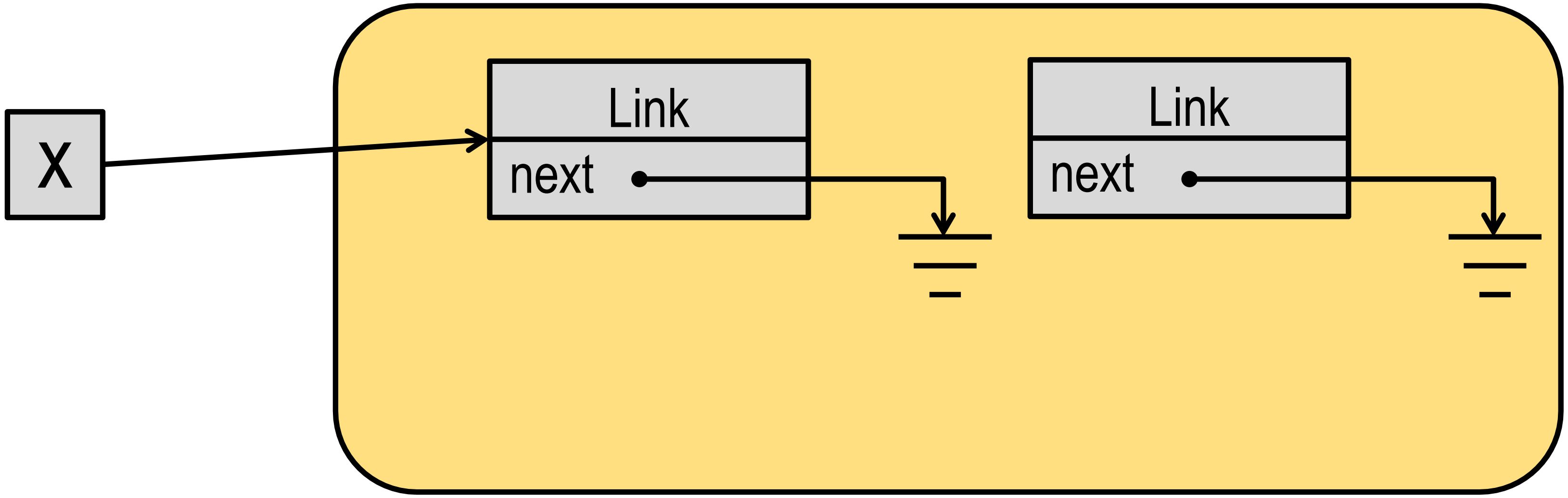
Objects and Memory (recap)

- More:
  - All objects are created on the **heap**
  - Variables and fields are **references** to objects on the heap
  - Don't need to **delete** objects on Java (unlike C/C++)

Q) What happens when the **heap** gets full?

Objects and Memory (recap)

```java
class Link {
  private Link next;
  public Link(Link next) { this.next = next; }
  public static void main(String[] args) {
    Link x = new Link(null);
    x = new Link(null);
}}
```



- In this case, first object created becomes **unreachable**

Reachability

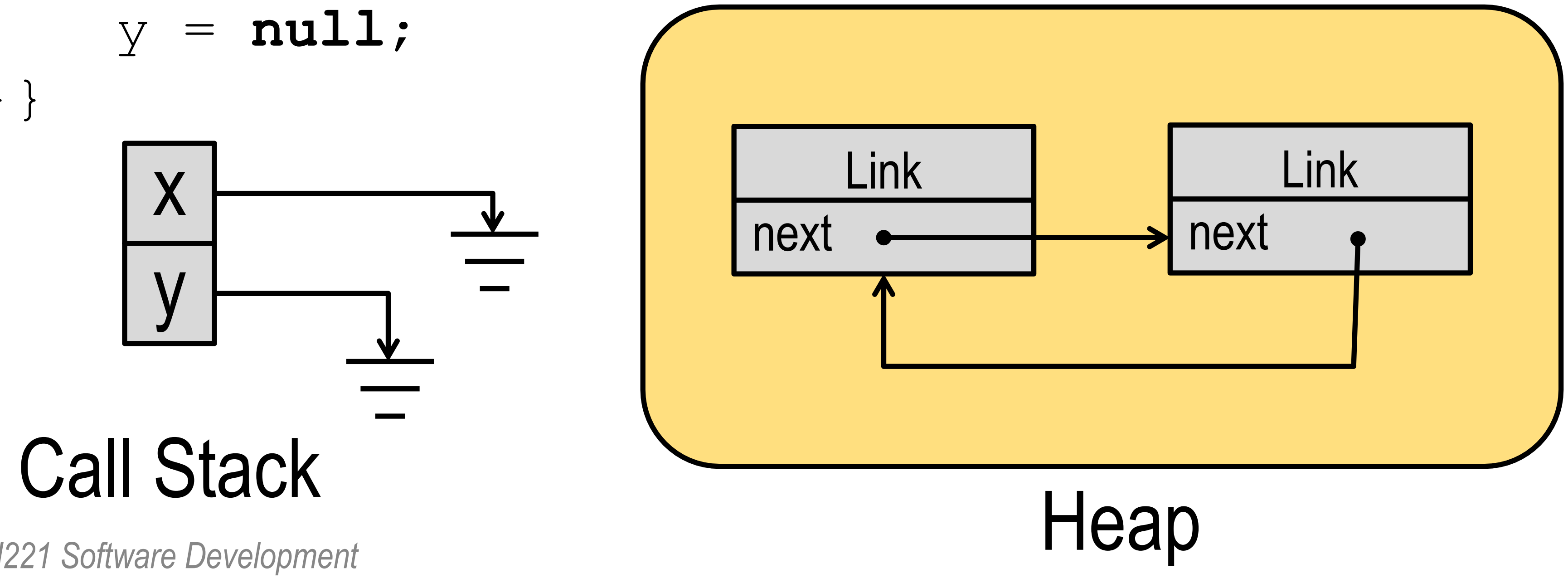
Defintion: *Reachable Object*

An object is reachable if a reference to it is stored in a local or static variable **or** it is stored in a field or array element of a reachable object.

- At a given point in time, the reachable objects:
  - Are those which can potentially be still used
  - Require space allocated in the heap
  - Cannot be deleted from the heap

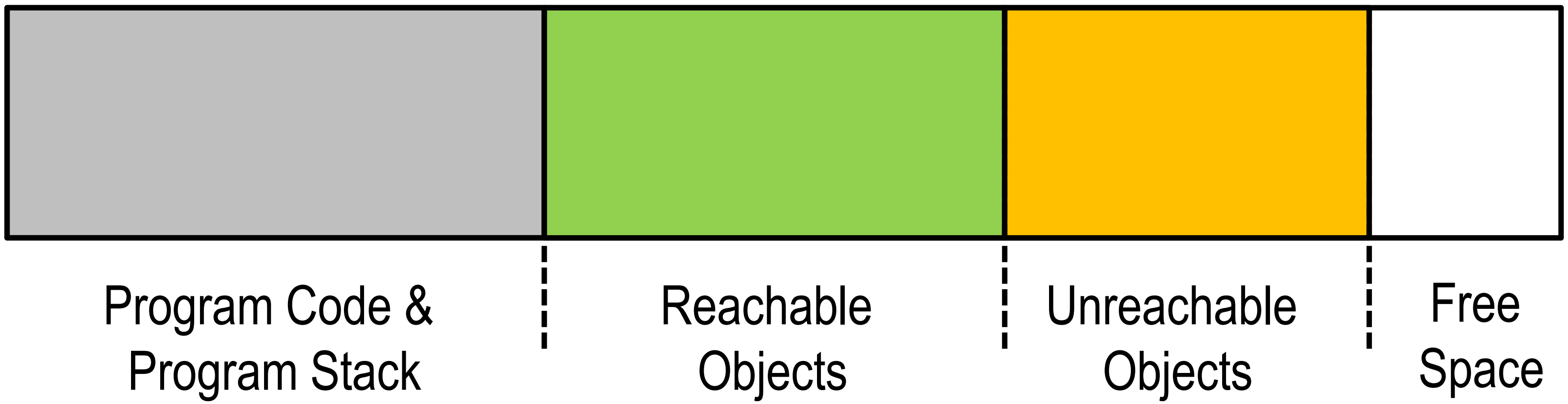
Q) Are these objects **reachable**?

```
class Link {
  private Link next;
  public Link(Link next) { this.next = next; }
  public static void main(String[] args) {
    Link x = new Link(null);
    Link y = new Link(x);
    x.next = y;
    x = null;
    y = null;
}}
```



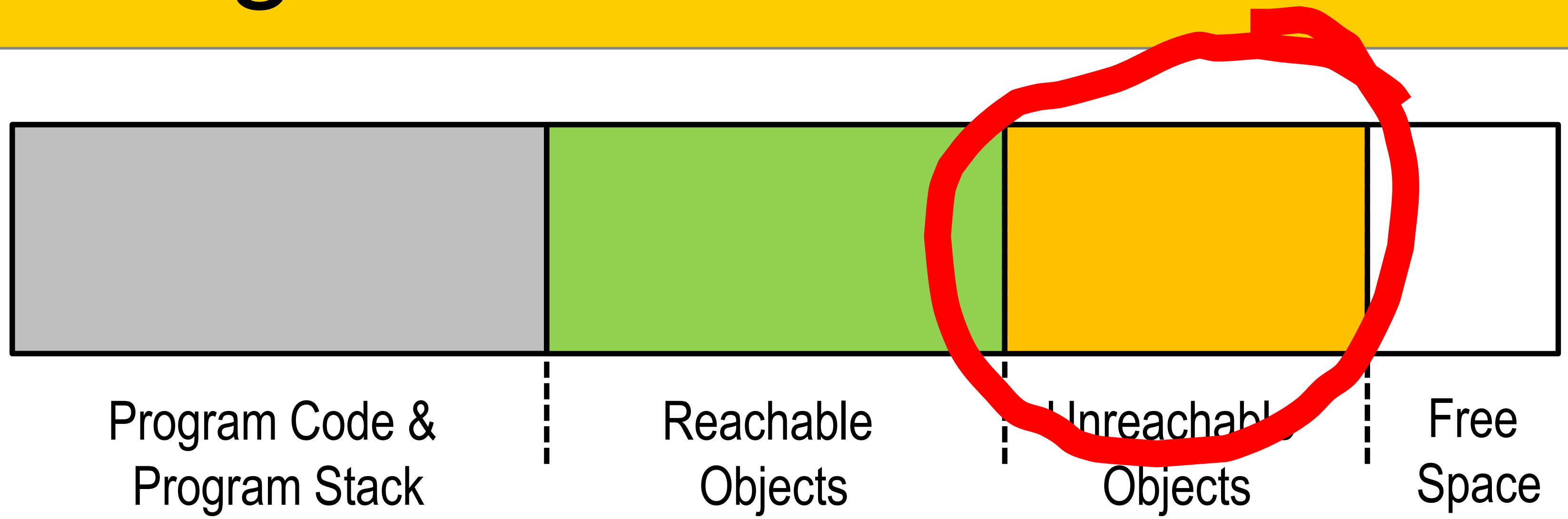
Breakdown of Memory Usage

- A rough breakdown of memory usage for a running program:



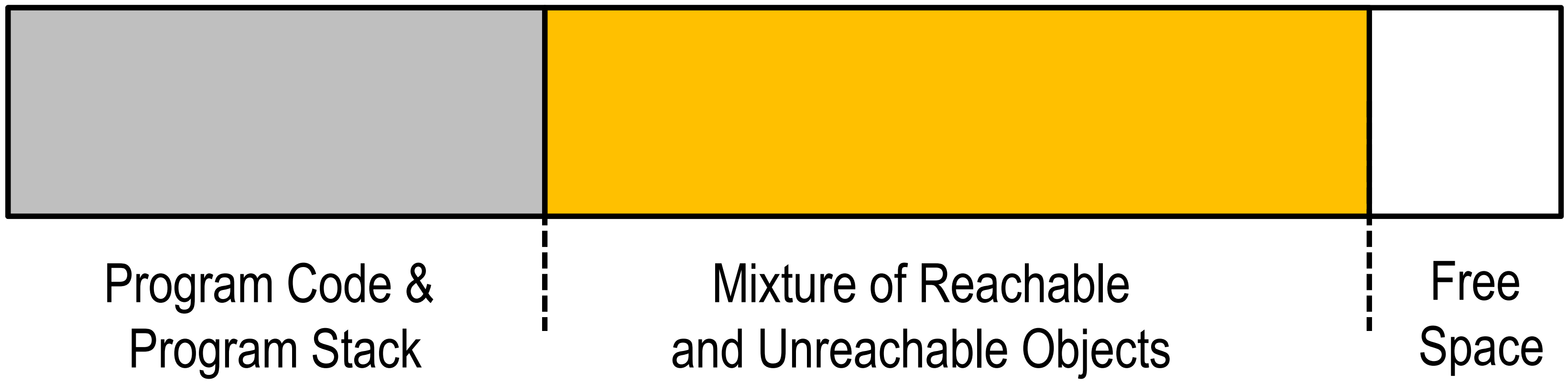
  - A running program has a **finite** amount of memory storage it can use
  - When memory is exhausted, program **halts** with `OutOfMemory` exception
  - Want to make most **efficient** use of memory …

Garbage Collection



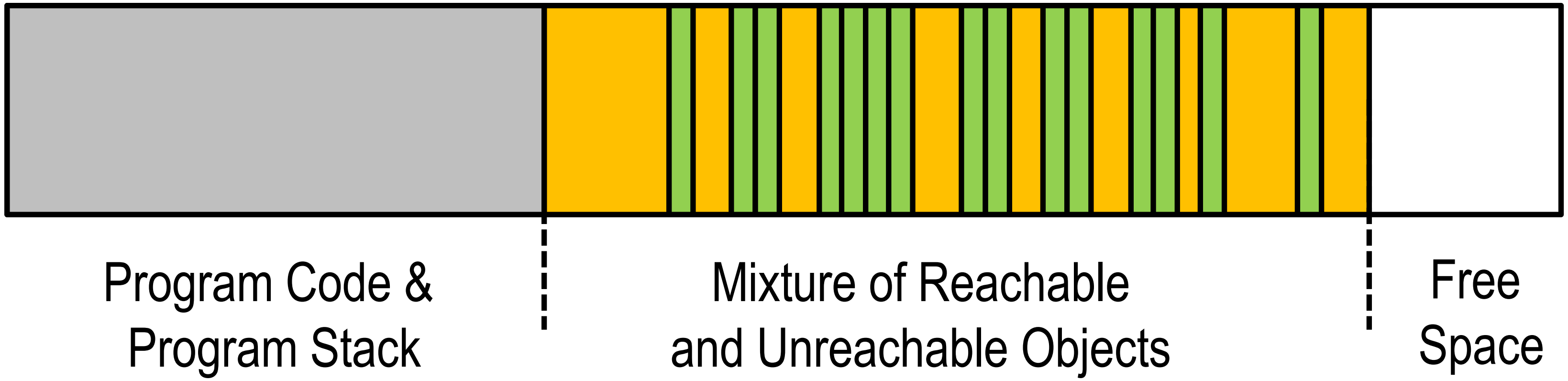
- Key Ideas:
  - Unreachable objects cannot **affect** program execution
  - Therefore, memory occupied by them can be **safely reclaimed**
  - Reclamation process is called **garbage collection**

Mark 'n Sweep Garbage Collection



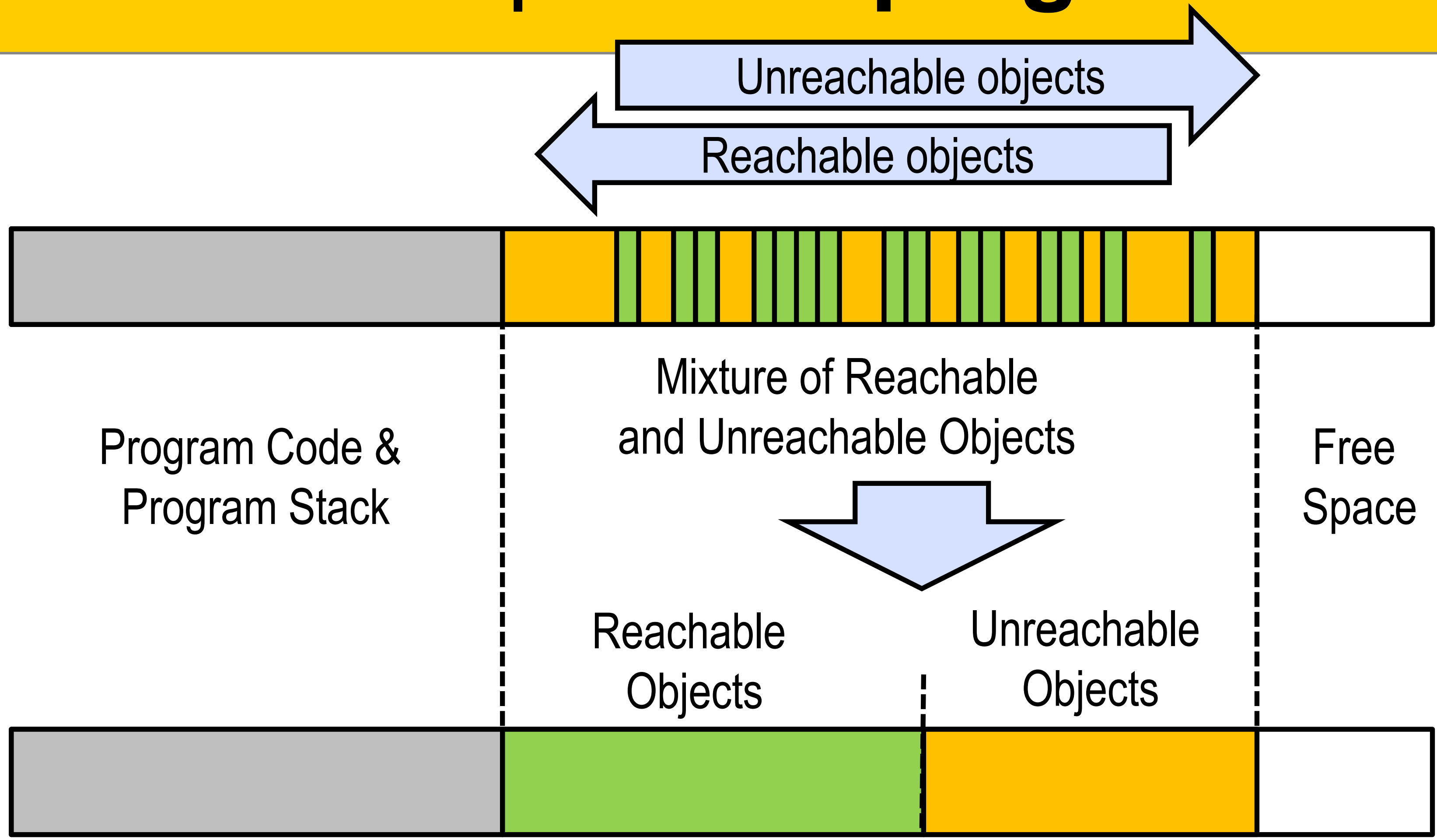
- Notes:
  - During execution, unreachable objects are **mixed up** with reachable objects
  - Must first **identify** unreachable objects, then we can reclaim them
  - **Basic algorithm** for this is called "mark and sweep"

Mark 'n Sweep: **Marking Phase**



- Notes:
  - Reachable objects are "marked" by **traversing** from object "roots"
  - Could use e.g. **depth-first search** for this
  - Roots are **local** variables and **static** variables

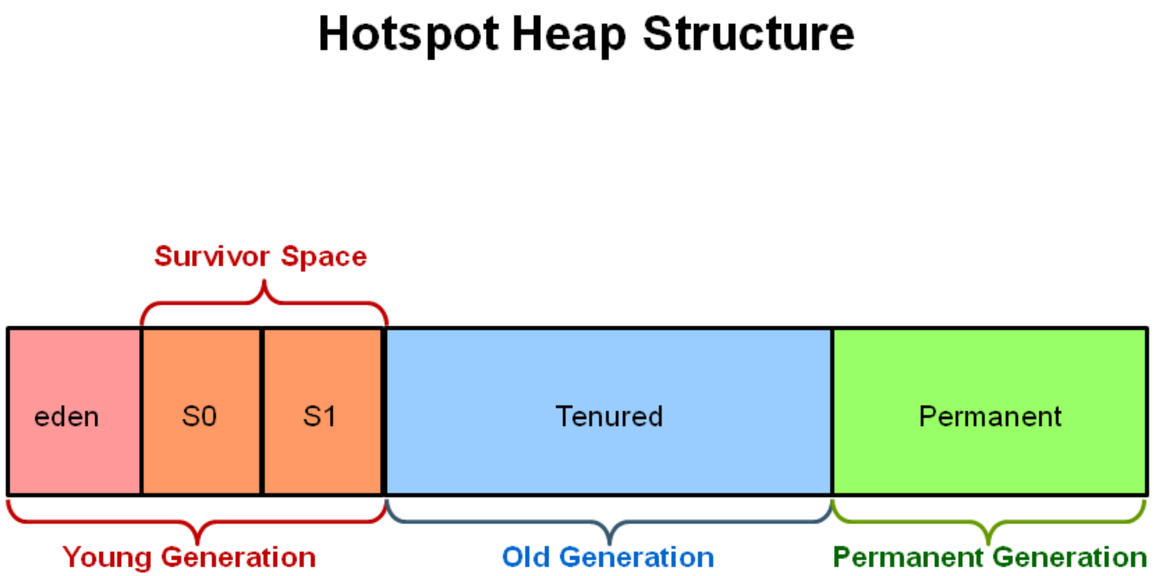
Mark 'n Sweep: **Sweeping Phase**



- Marked objects are "swept" to the left
- Unmarked objects are "swept" to the right
- Then can reclaim the unmarked objects

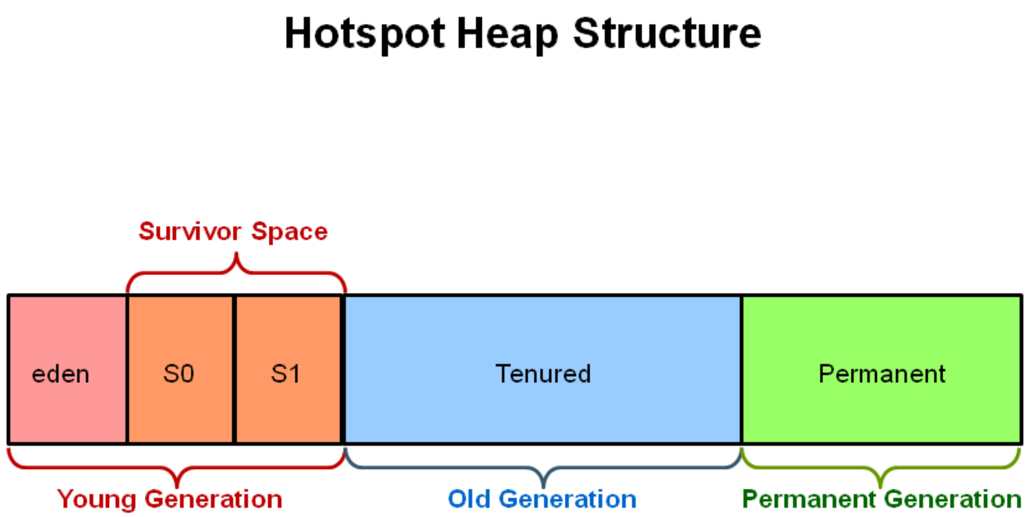
Generational Garbage Collection

- Heap is broken up into smaller generations



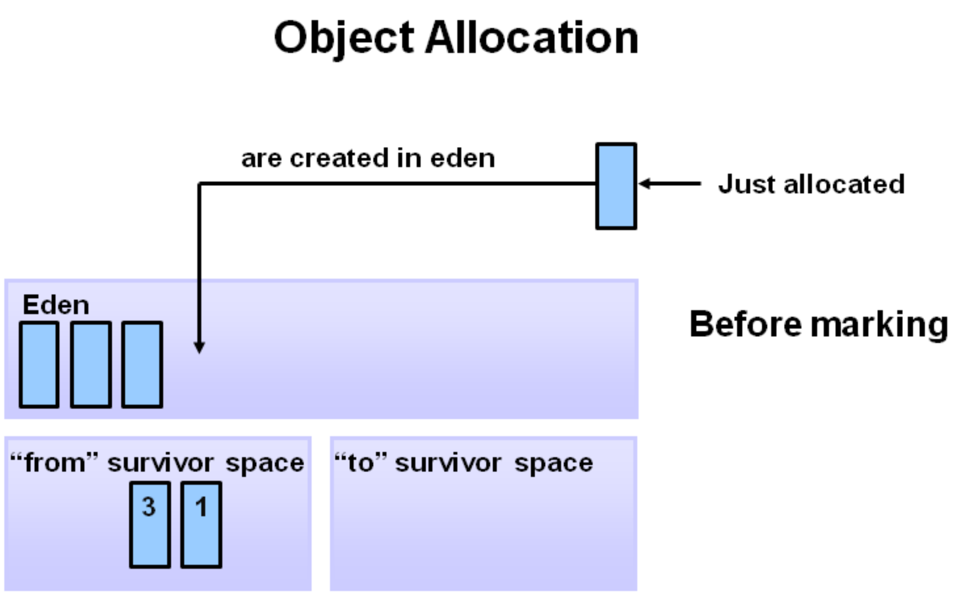
Generational Garbage Collection

- Young generation – minor garbage collection
- Old generation – major garbage collection
- Permanent generation – full garbage collection



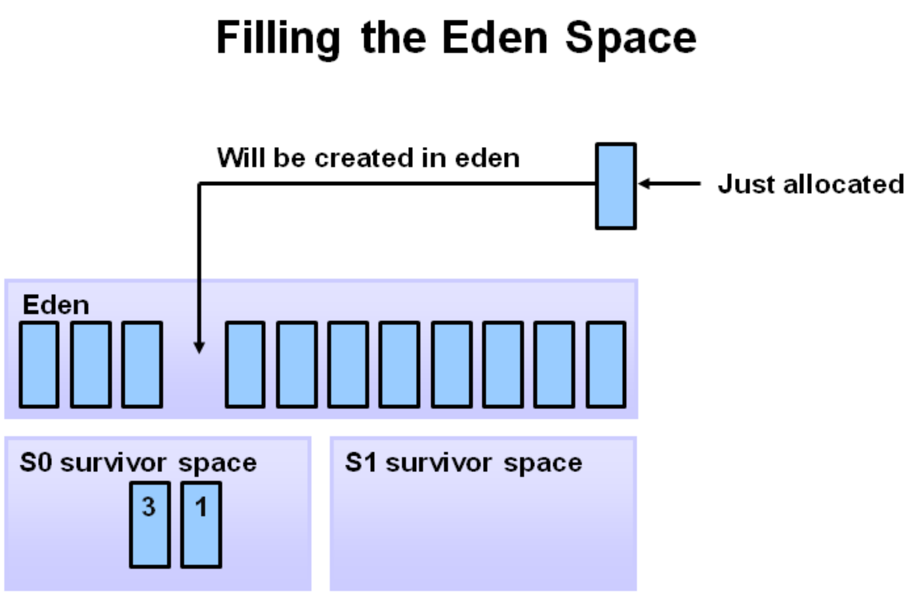
Generational Garbage Collection

- New objects allocated to the eden space. Both survivor spaces are empty.



Generational Garbage Collection

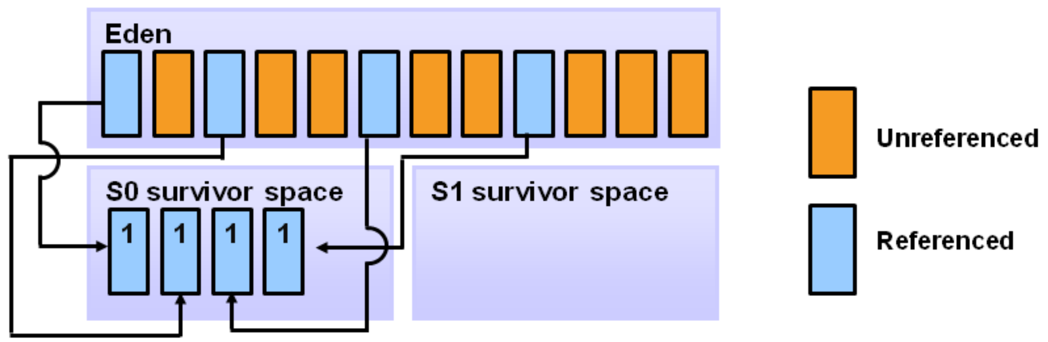
- When the eden space is full, a minor garbage collection is triggered.



Generational Garbage Collection

- Referenced objects to S0 survivor space, unreferenced objects removed.

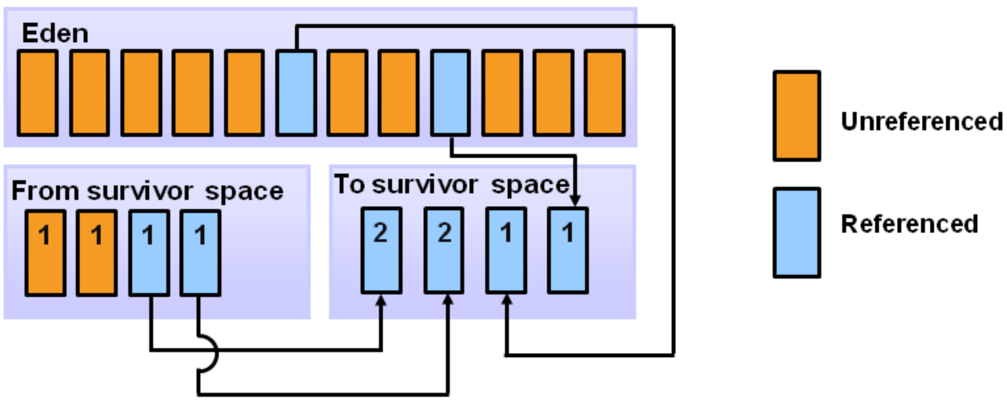
Copying Referenced Objects



Generational Garbage Collection

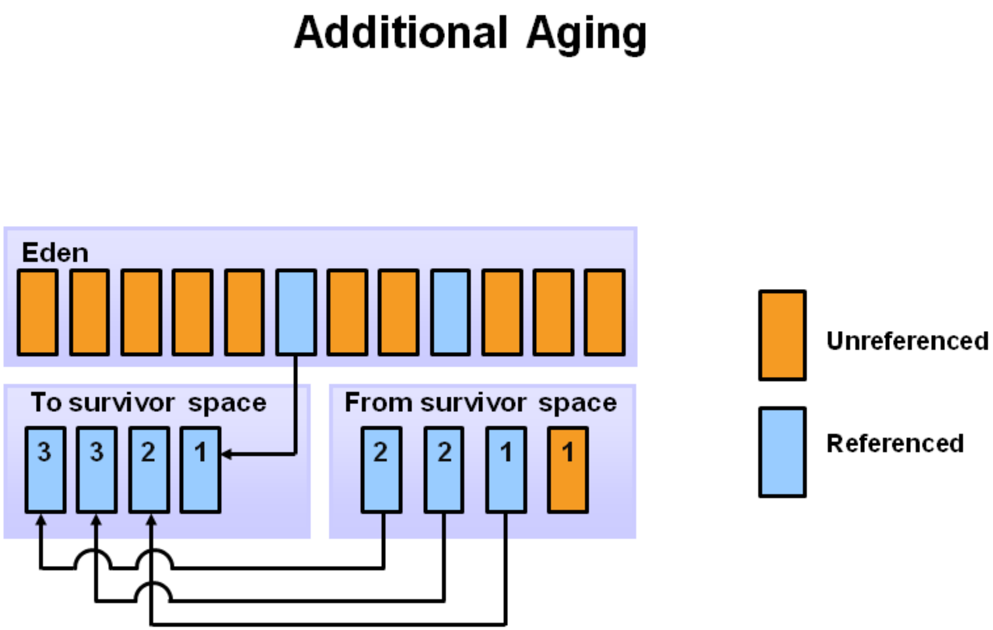
- When next minor garbage collection, referenced objects move to S1 survivor space.

Object Aging



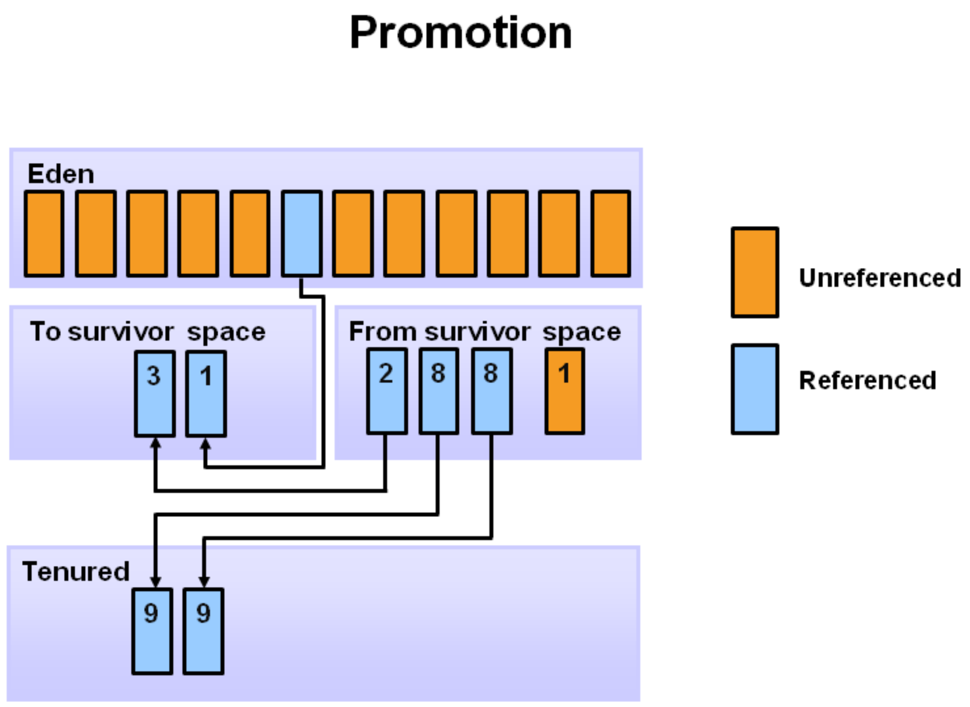
Generational Garbage Collection

- As minor garbage collection goes on, switch the from/to survivor spaces, increment aging.



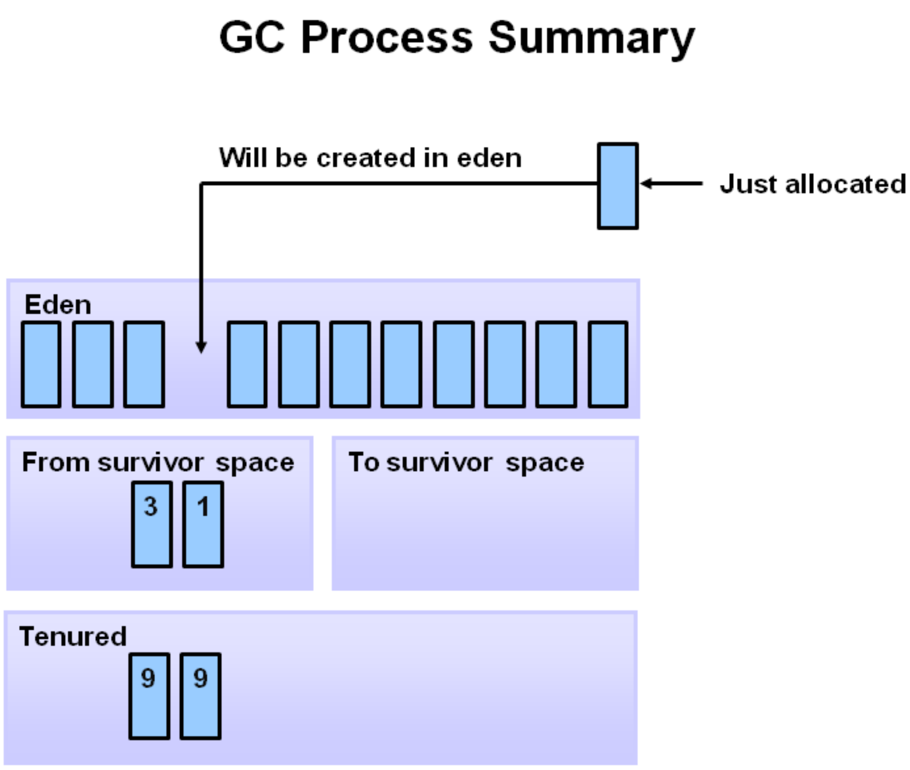
Generational Garbage Collection

- When age of objects reach threshold (8 in this case), move to old/tenured generation.



Generational Garbage Collection

- Eventually, when old/tenured generation is filled up, a major garbage collection is triggered.



Garbage Collectors

- Setup GC configurations in command line
  - `-Xms`: the initial heap size for when the JVM starts.
  - `-Xmx`: the maximum heap size.
  - `-Xmn`: the size of the Young Generation.
  - `-XX:PermSize`: the starting size of the Permanent Generation.
  - `-XX:MaxPermSize`: the maximum size of the Permanent Generation

Garbage Collectors

- **Serial GC**: both minor and major garbage collections are done serially (using a single virtual CPU)

```
$ java -Xmx12m -Xms3m -Xmn1m -XX:PermSize=20m
-XX:MaxPermSize=20m -XX:+UseSerialGC -jar
JavaDemo.jar
```

Garbage Collectors

- **Parallel GC**: uses multiple threads to perform the young generation garbage collection.

```
$ java -Xmx12m -Xms3m -Xmn1m -XX:PermSize=20m
-XX:MaxPermSize=20m -XX:+UseParallelGC -jar
JavaDemo.jar
```

- Use multiple threads for both young and old generation garbage collection.

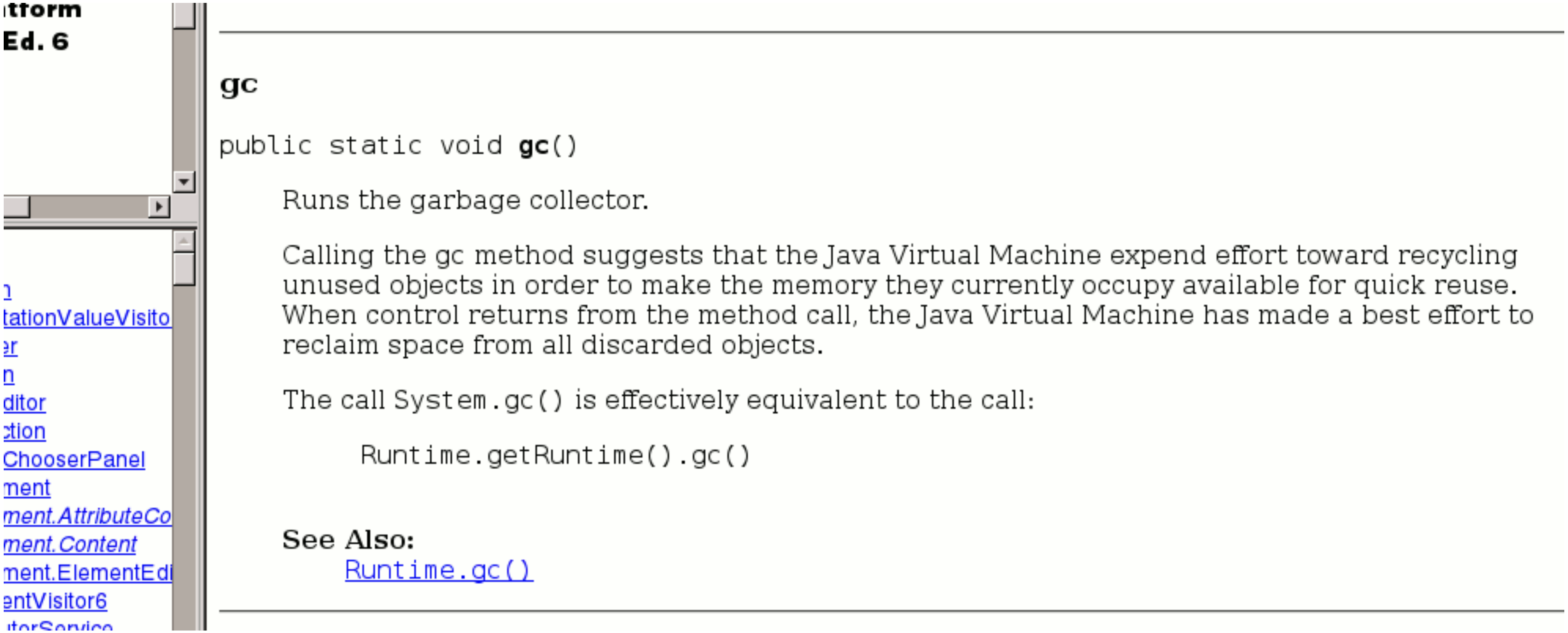
```
$ java -Xmx12m -Xms3m -Xmn1m -XX:PermSize=20m
-XX:MaxPermSize=20m -XX:+UseParallelOldGC -
jar JavaDemo.jar
```

Pros / Cons of Garbage Collection

- **Pros:**
  - Don't have to explicitly **free memory** (as you do in C/C++)
  - Memory **fragmentation** not such an issue
  - Can have **better performance** as active part of heap occupies smaller footprint

- **Cons:**
  - Garbage collection takes time!
  - System **paused** during garbage collection
  - GC pauses are unpredictable
  - Can be a serious problem for **real-time systems**

Forcing Garbage Collection



gc

```
public static void gc()
```

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.

The call `System.gc()` is effectively equivalent to the call:

```
Runtime.getRuntime().gc()
```

See Also:
Runtime.gc()

- Can attempt to force Garbage Collection:
  - Using System.gc()
  - No guarantee that it will do anything!

Weak References

java.lang.ref

Class WeakReference<T>

java.lang.Object
  └ java.lang.ref.Reference<T>
      └ **java.lang.ref.WeakReference<T>**

---

public class **WeakReference<T>**
extends Reference<T>

Weak reference objects, which do not prevent their referents from being made finalizable, finalized, and then reclaimed. Weak references are most often used to implement canonicalizing mappings.

Suppose that the garbage collector determines at a certain point in time that an object is weakly reachable. At that time it will atomically clear all weak references to that object and all weak references to any other weakly-reachable objects from which that object is reachable through a chain of strong and soft references. At the same time it will declare all of the formerly weakly-reachable objects to be finalizable. At the same time or at some later time it will enqueue those newly-cleared weak references that are registered with reference queues.

- Weak References don't prevent garbage collection of objects they refer to (called **referents**)
- Useful for objects which can be reclaimed, but keeping offers some advantage (e.g. a cache)

Notes:

- Not covered (but could be):
  - Fact that references are all updated when objects moved
  - Illustration of memory fragmentation in C/C++
  - Finalisers
  - Generational garbage collection
  - Continuous garbage collection (or similar)
  - Reference counted garbage collection
  - Provide more details of what a call stack is.