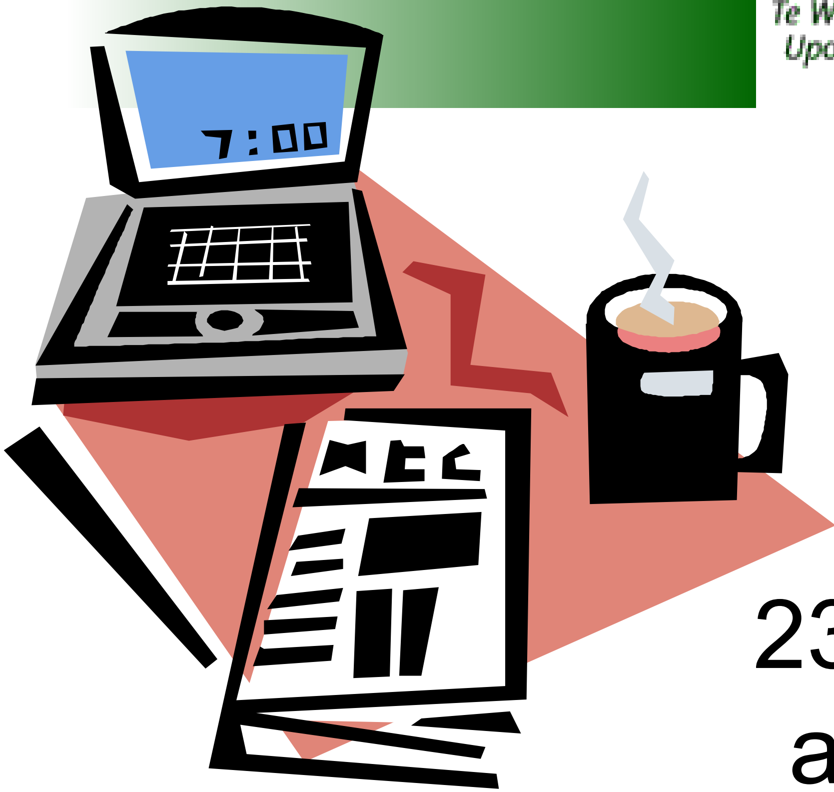
Victoria University of Wellington, New Zealand
Te Whare Wananga o te Upoko o te Ika a Maui
Aotearoa



SWEN221: Software Development

23: Memory Management and Garbage Collection

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

Putting knowledge together

```
abstract class Tree<E>{
  static final class Empty<E> extends Tree<E>{/*...*/}
  static final class Node<E> extends Tree<E>{/*...*/}

  public abstract boolean isEmpty();

  public abstract Node<E> get();

  //Using lambdas to match and avoid instanceof/casts
  public abstract <T> T match(Function<Node<E>,T> forNode, T forEmpty);

  private Tree(){}//sealing Tree: only its nested can have instances

  static private final Empty<?> empty=new Empty<>();
  @SuppressWarnings("unchecked")
  static public <E>  Empty<E> empty() {return (Empty<E>)empty;}

  static public <E>  Node<E> node(E elem, Tree<E> left, Tree<E> right){
    return new Node<E>(elem,left,right);
  }
}
```

Putting knowledge together

```
static final class Empty<E> implements Tree<E>{

  private Empty(){}

  public boolean isEmpty() {return true;}

  public Node<E> get() {throw new Error("..");}

  public <T> T match(Function<Node<E>,T> forNode, T forEmpty){
    return forEmpty;
  }
}
```

Putting knowledge together

boring but needed

interesting

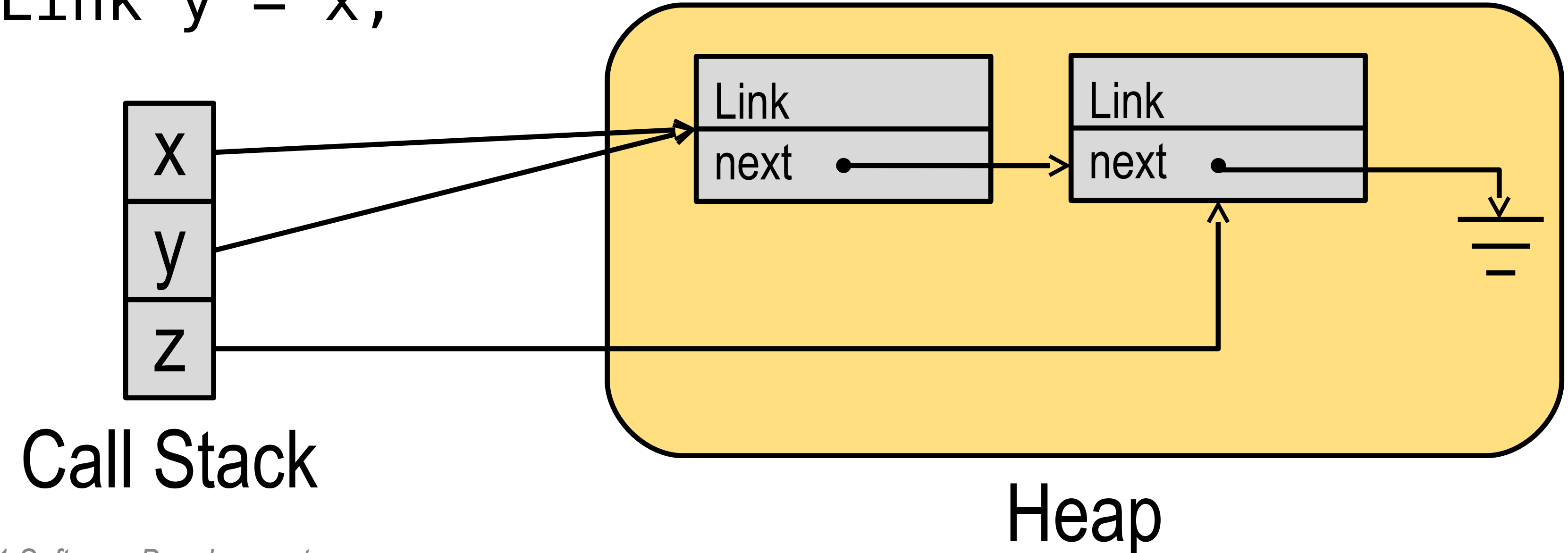
```
static final class Node<E> implements Tree<E>{

  private Tree<E> left;//never null
  private Tree<E> right;//never null
  private E elem;//never null

  public Tree<E> getLeft(){return left;}
  public Tree<E> getRight(){return right;}
  public E getElem(){return elem;}

  public void setLeft(Tree<E> left){assert left!=null;this.left=left;}
  public void setRight(Tree<E>right){assert right!=null;this.right=right;}
  public void setElem(E elem) {assert elem!=null;this.elem=elem;}

  private Node(E elem,Tree<E> left,Tree<E> right){
    setElem(elem);setLeft(left);setRight(right);
  }
  public boolean isEmpty(){return false;}
  public Node<E> get() {return this;}
  public <T> T match(Function<Node<E>,T> forNode, T forEmpty) {
    return forNode.apply(this);
  }
}
```

Putting knowledge together

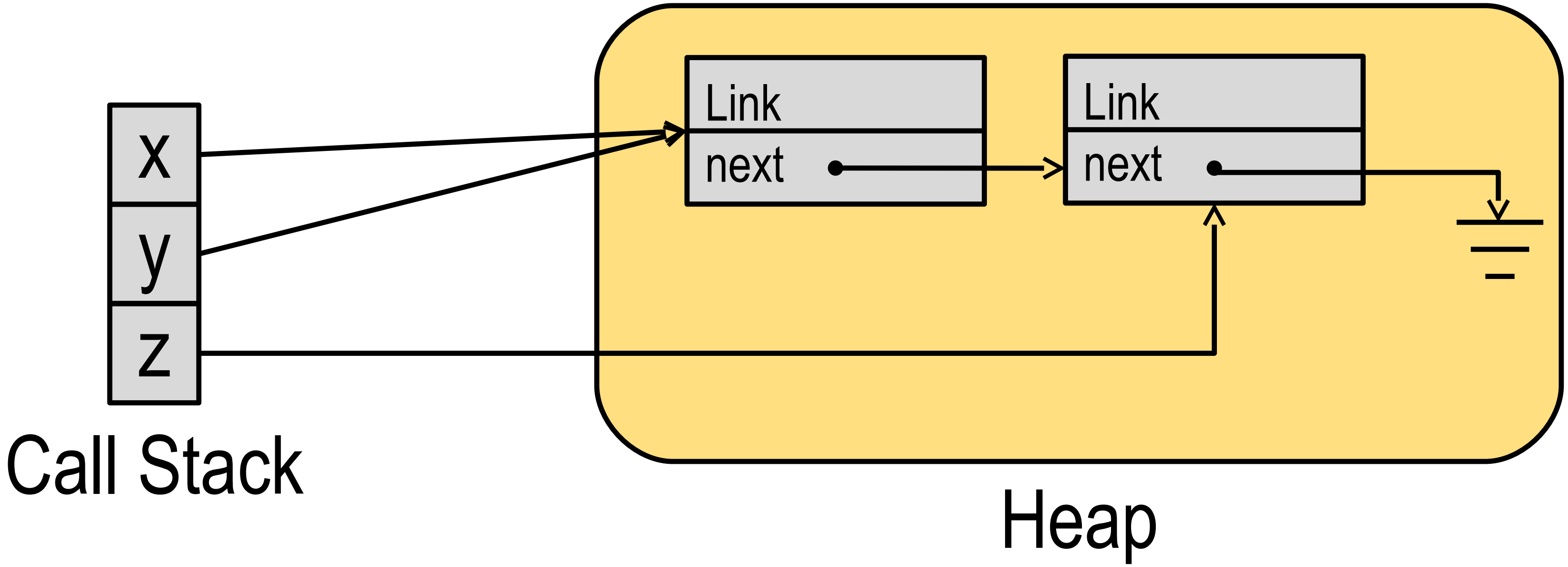
```java
public static void main(String[] args){

  Tree<String> hello=Tree.node("hello",Tree.empty(),Tree.empty());

  Tree<String> world=Tree.node("world",Tree.empty(),Tree.empty());

  Tree<String> hi=Tree.node("hi",hello,world);

  System.out.println(toString(hi));
  }

public static String toString(Tree<String> t){
  return t.match(
    //case Node<String>
    n->" "+n.getElem()+toString(n.getLeft())+toString(n.getRight()),
    //case Empty
    "");
  }
```

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**

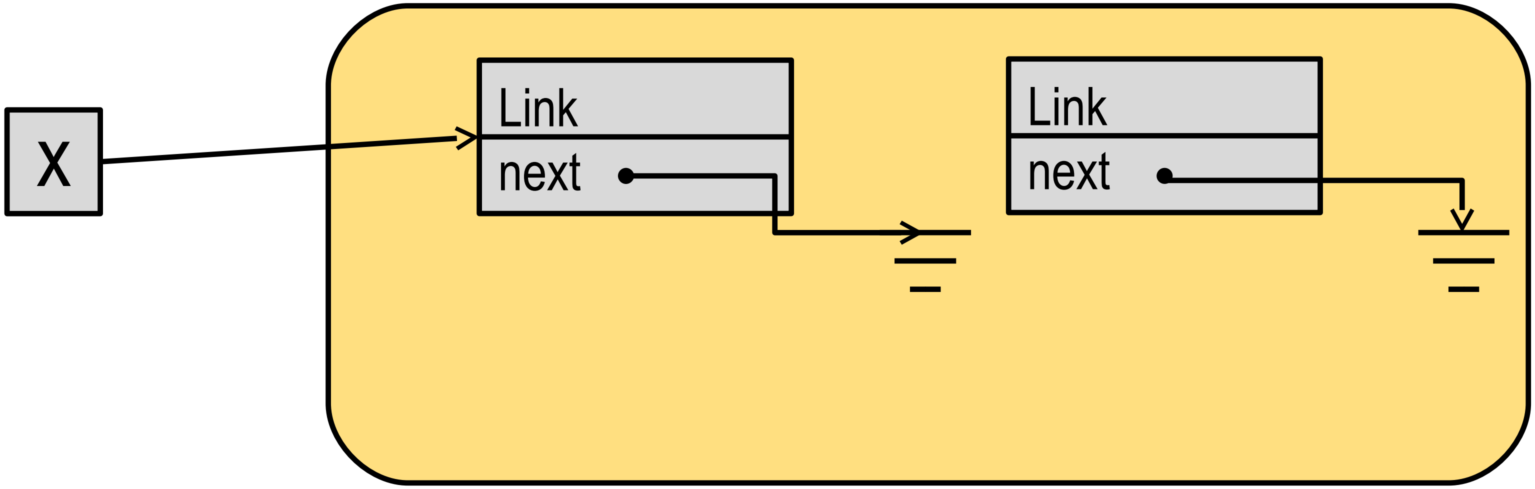
Objects and Memory (recap)

- 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)

```
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

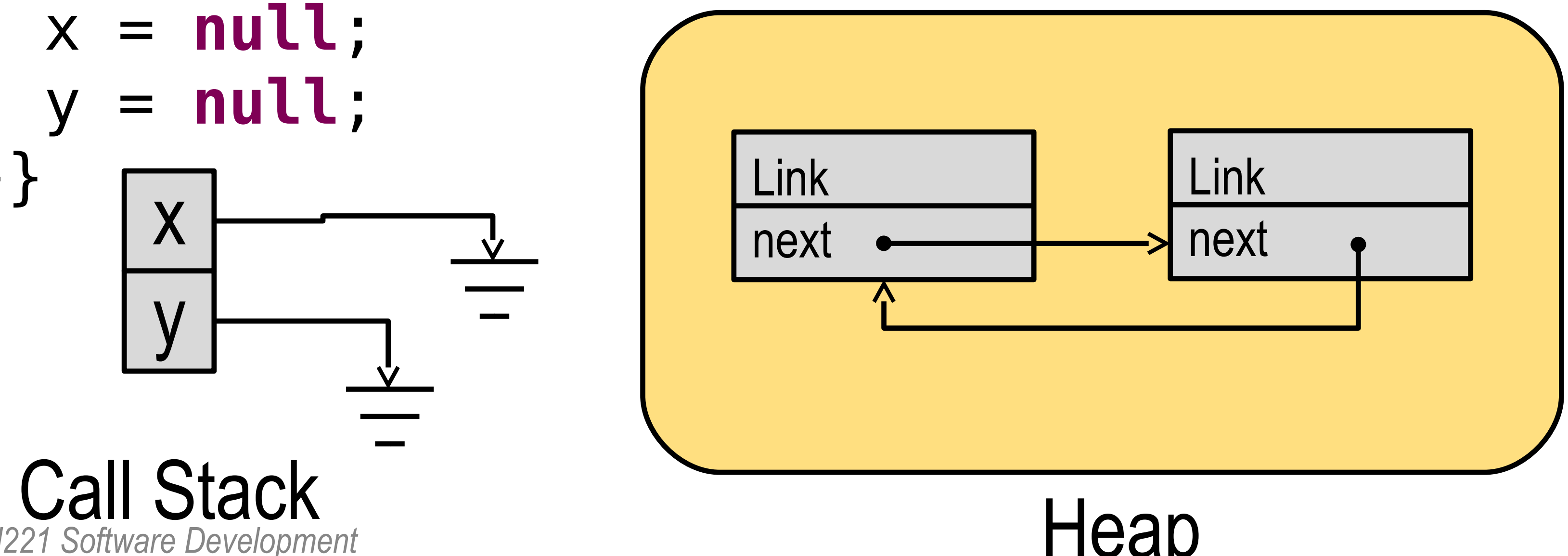
Defintion: *Reachable Object*

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

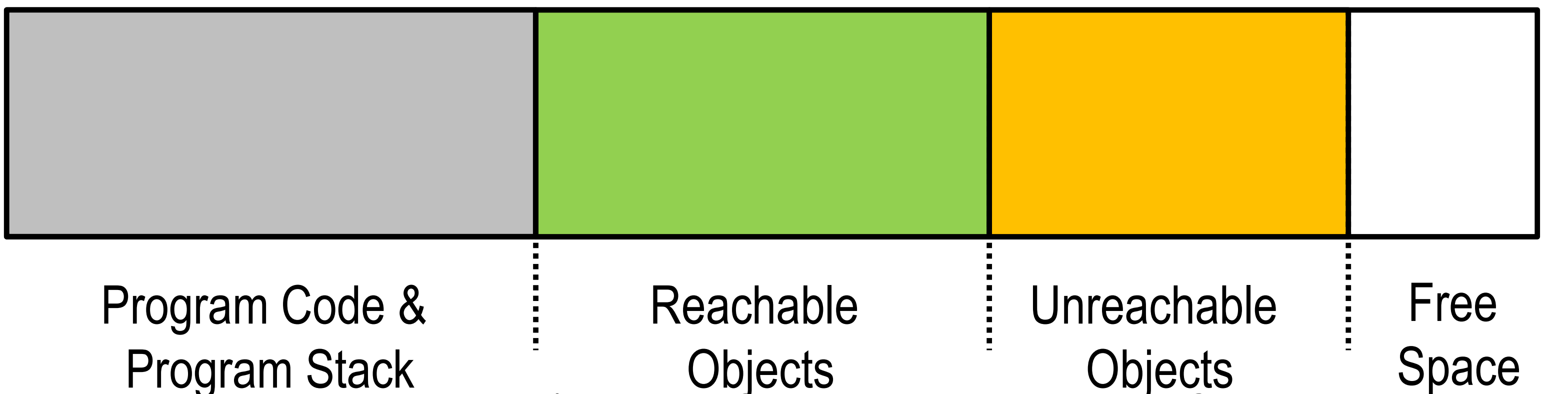
Q) Are these objects **reachable**?

```
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:

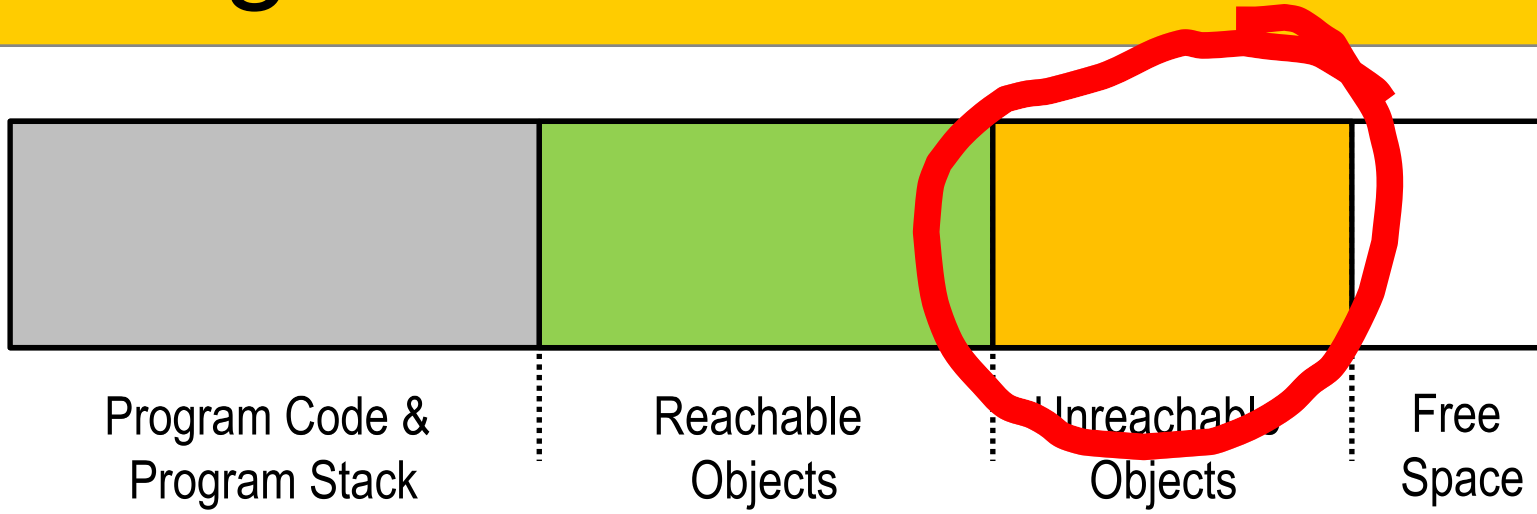


A running program has a **finite** amount of memory storage it can use

When memory is exhausted, program **halts** with OutOfMemory exception

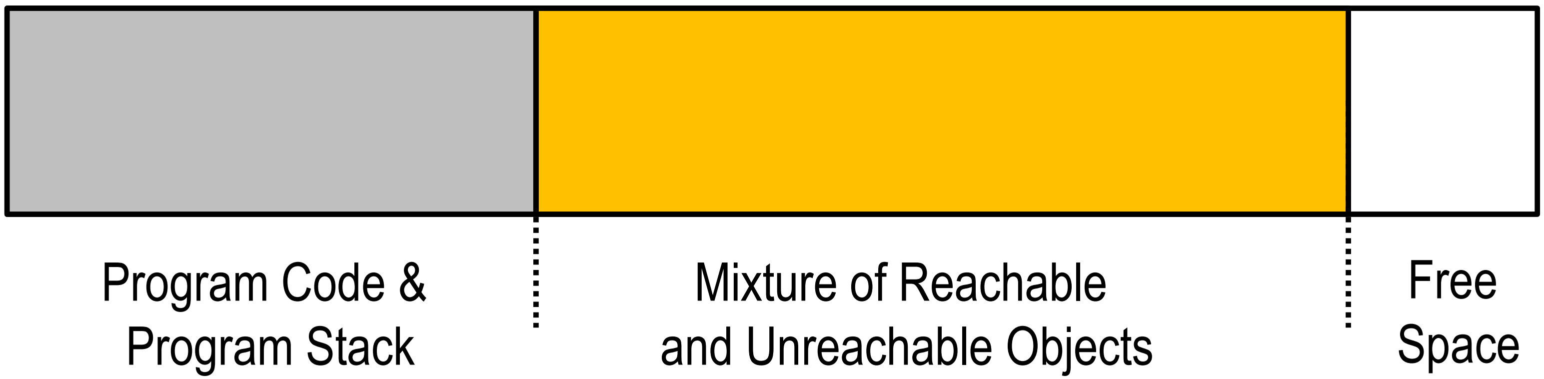
Want to make most **efficient** use of memory …

Garbage Collection



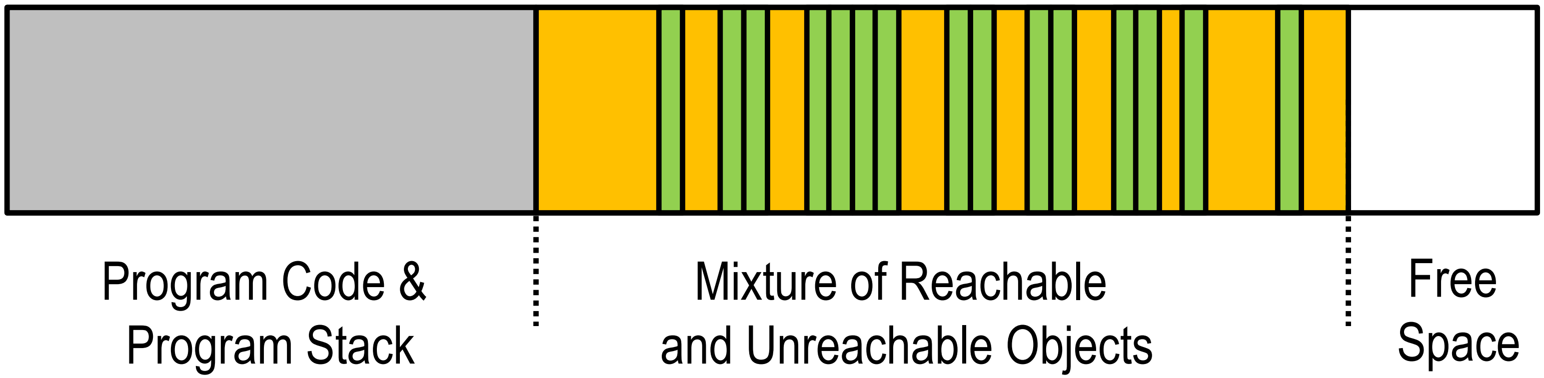
- Key Ideas:
  - Unreachable objects cannot **affect** program execution
  - Therefore, memory occupied by them can be **safely reclaimed**
  - Reclamation process is called **garbage collection**

Mark ‘n Sweep Garbage Collection



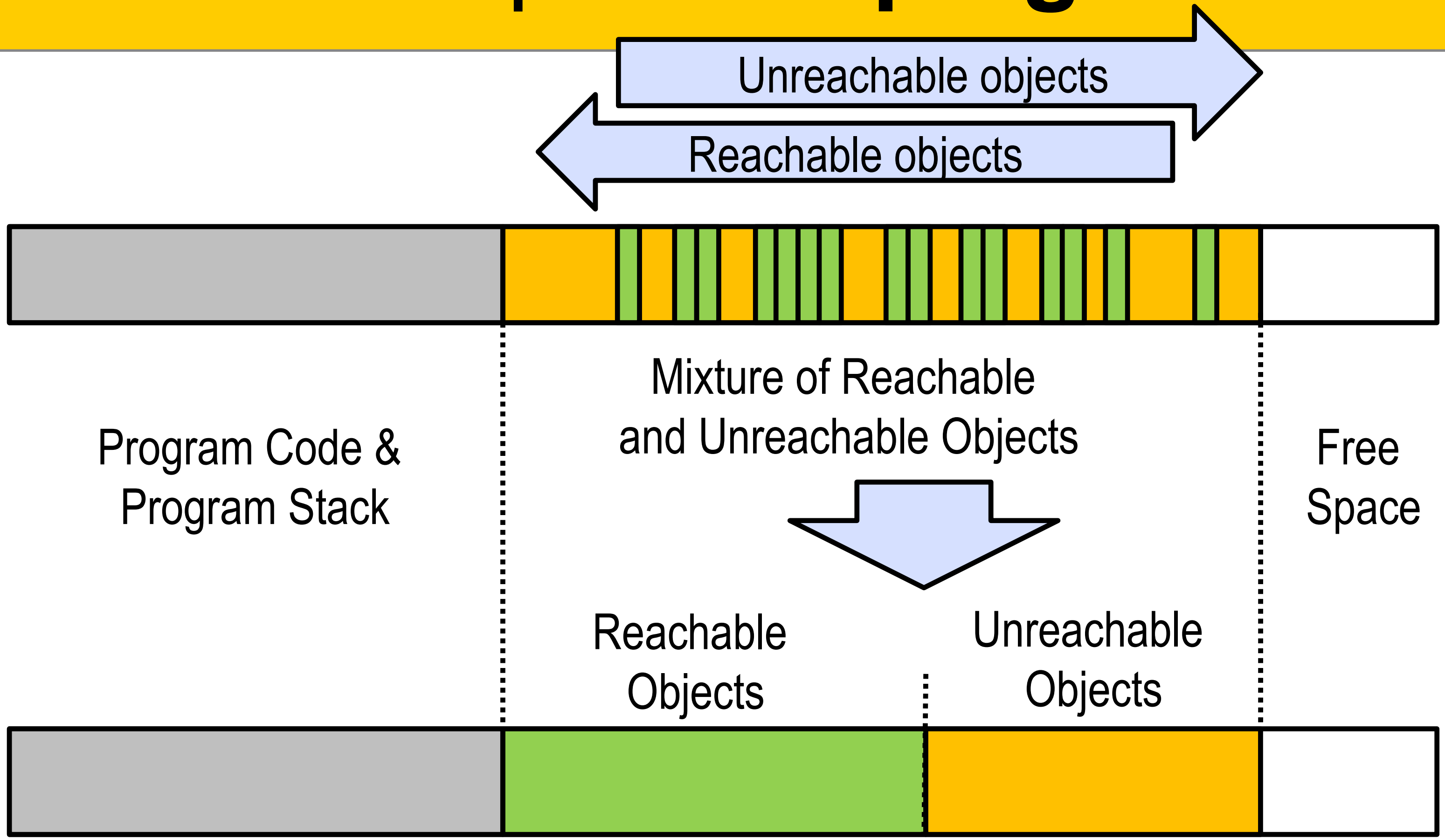
- Notes:
  - During execution, unreachable objects are **mixed up** with reachable objects
  - Must first **identify** unreachable objects, then we can reclaim them
  - **Basic algorithm** for this is called “mark and sweep”

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

Does Java use Mark ‘n Sweep?

Well....

HotSpot uses **also** mark and sweep.

HotSpot combine together another dozen of tecnques...

Pros / Cons of Garbage Collection

- **Pros:**
  - Don't have to explicitly **free memory** (as you do in C/C++)
  - A whole class of errors simply disappear.
  - Performance is improved in the general case

- **Cons:**
  - Garbage collection takes time!
  - Performance loss in simple enough programs
  - System **paused** during garbage collection
  - GC pauses are unpredictable,it can be a serious problem for **real-time systems**

On a specific JVM -- JVM tuning

On a specific JVM -- JVM tuning

On HotSpot (the “default one”), we have options to “tune” our Java

- page with all the options

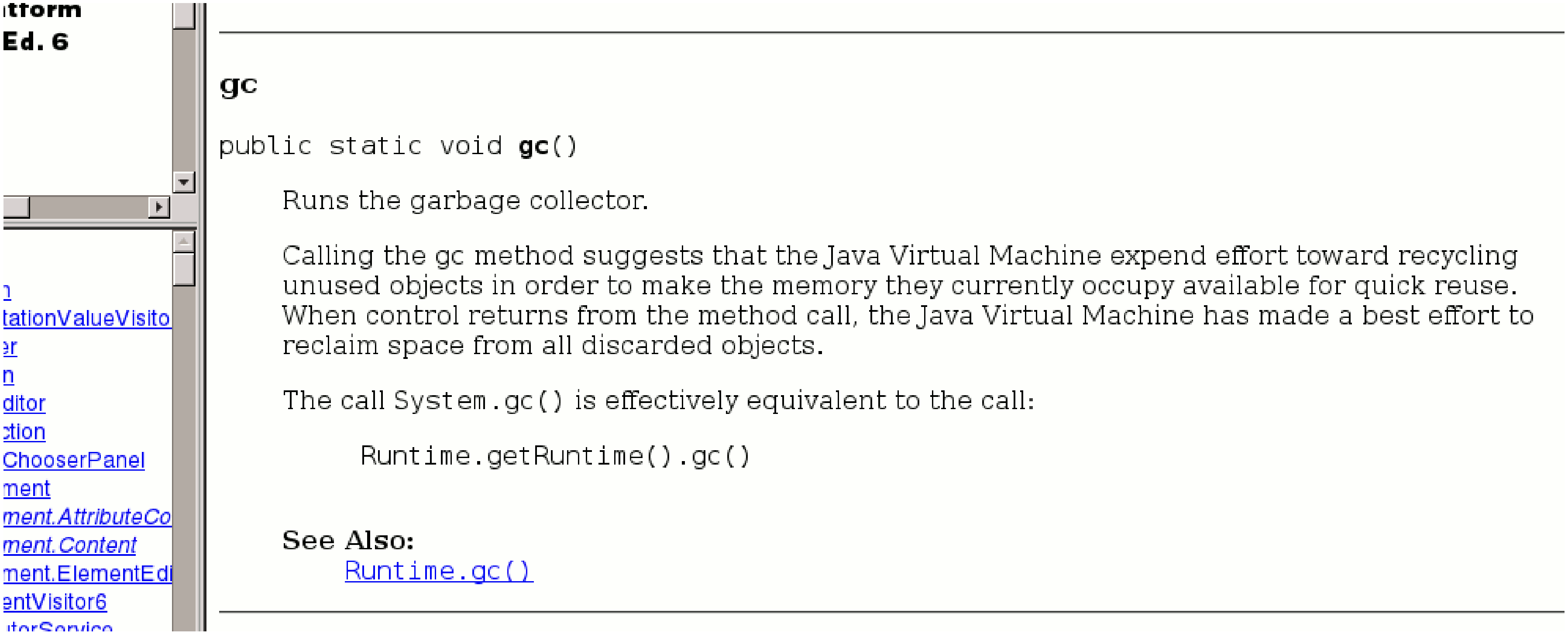
http://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html

- result of a simple search “how to tune JVM”

http://randomlyrr.blogspot.co.nz/2012/03/java-tuning-in-nutshell-part-1.html

On a specific JVM -- JVM tuning

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)

Using WeakReference

```
class MapChunk { CellInfo[][] cells; }
class CellInfo {/* ... */}
/* ... */
WeakReference<MapChunk>[][] chunks
    = new WeakReference[100][100];
/* ..init all 100*100 with an empty WeakRef.. */
MapChunk c = null;
if(chunks[px][py]!=null){c = chunks[px][py].get();}
if (c == null){
  c = loadFromDb(px, py);
  chunks[px][py] = new WeakReference<MapChunk>(c);
  }
/* .. game logic here .. */
```

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.