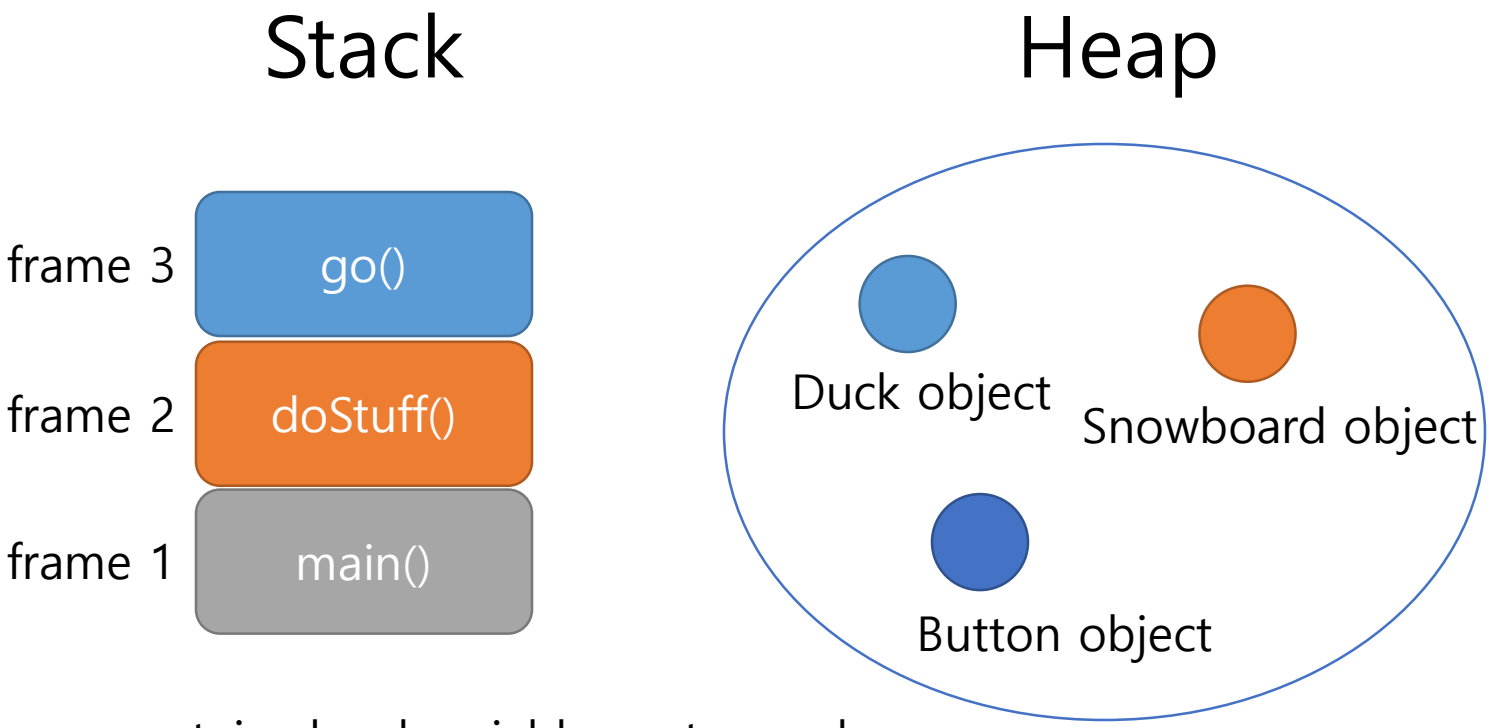
Constructor & Garbage Collection

Table of Contents

Java Memory Sections

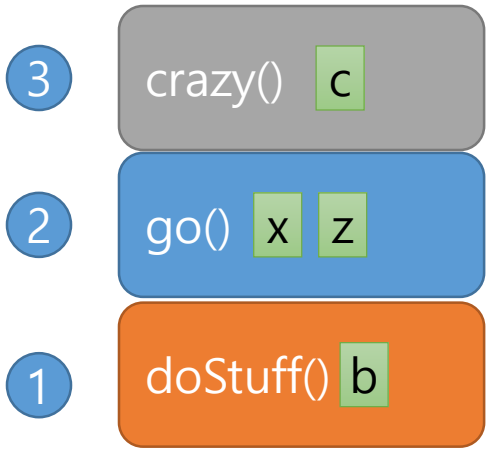
- Heap
  - The heap section contains objects.
- Stack
  - The stack section contains method registers, execution point, local primitive and reference variables, and parameters.
- Code & Static
  - The code section contains the program byte code.
  - The static section contains static variables.

Stack and Heap (1)



* Each frame contains local variables, return value, operand stack, constant pool resolution, exception dispatch, etc.

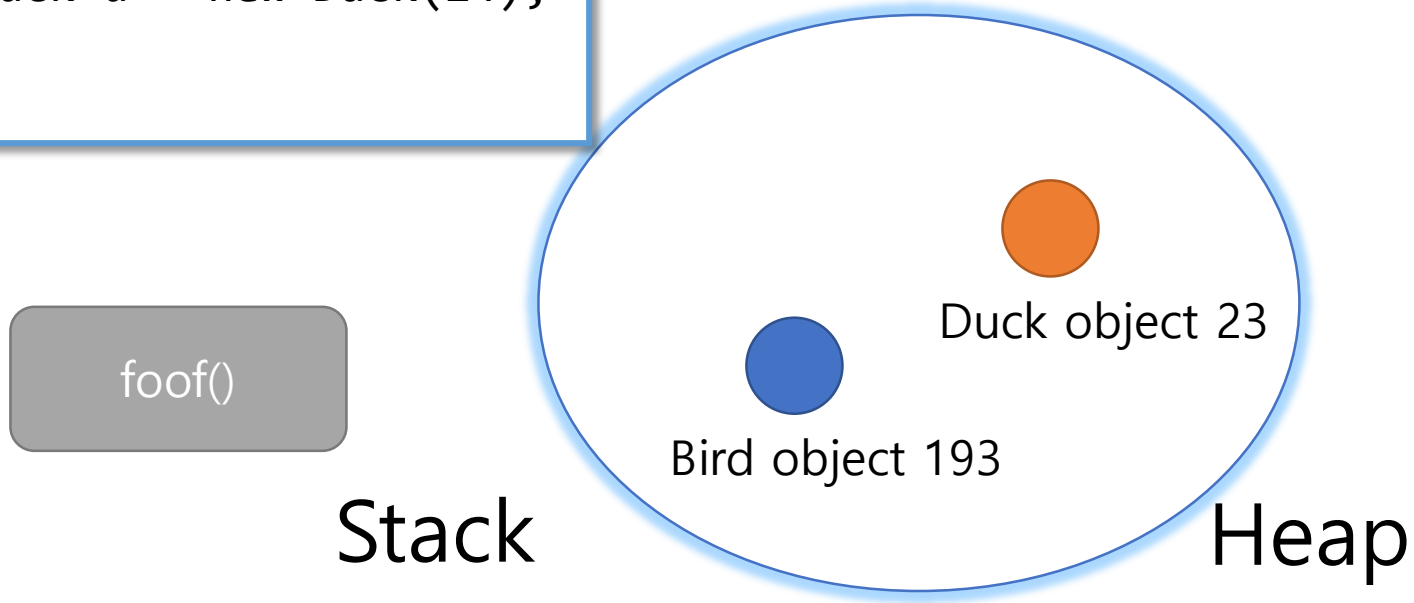
Stack and Heap (2)




```
public void doStuff() {
    boolean b = true;
    go(4);
}

public void go(int x) {
    int z = x + 24;
    crazy();
    // more code here
}

public void crazy() {
    char c = 'a';
}
```

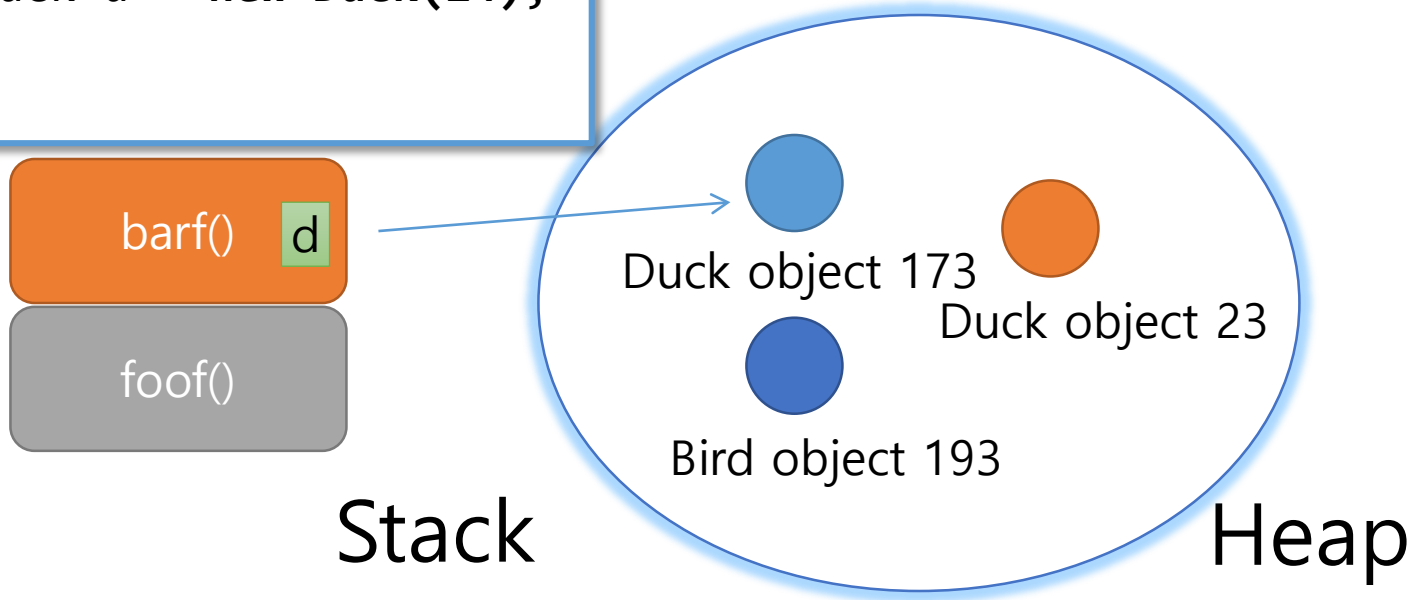
Stack and Heap (3)

```
public class StackRef {
    public void foof() {
        barf();
    }
    public void barf() {
        Duck d = new Duck(24);
    }
}
```



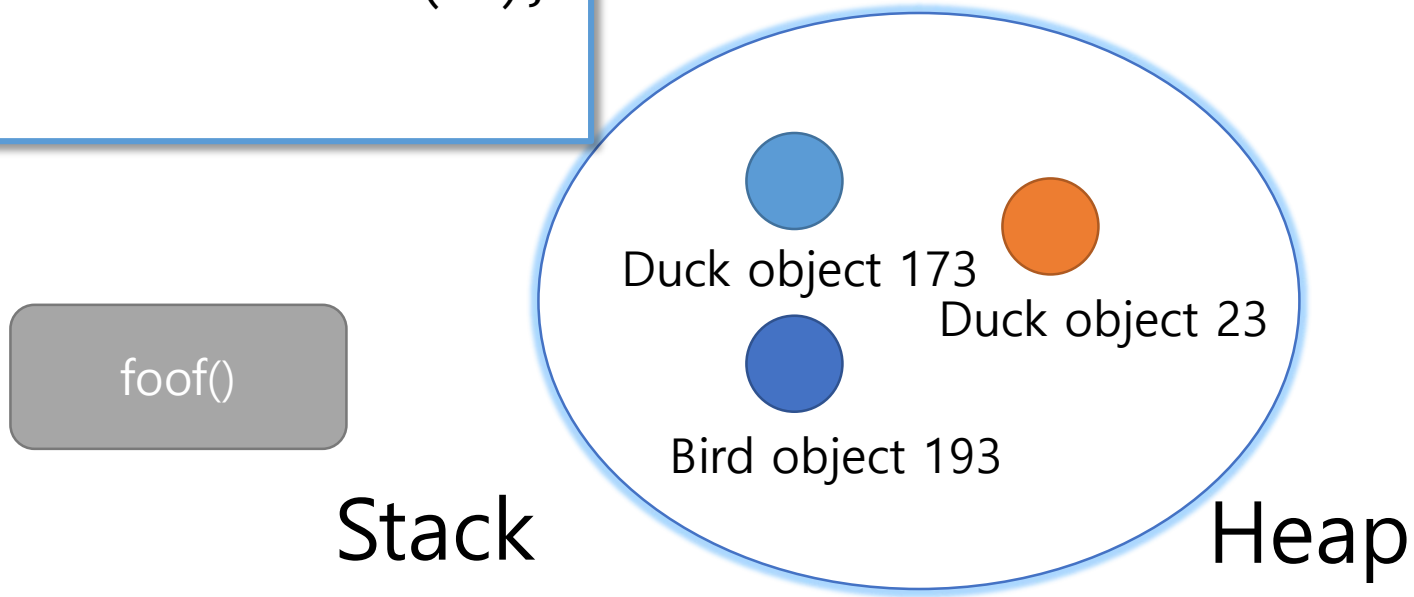
Stack and Heap (3)

```
public class StackRef {
    public void foof() {
        barf();
    }
    public void barf() {
        Duck d = new Duck(24);
    }
}
```



Stack and Heap (3)

```
public class StackRef {
    public void foof() {
        barf();
    }
    public void barf() {
        Duck d = new Duck(24);
    }
}
```



Stack and Heap (4)

```
public class Cellphone {
    int x;
    long y;
    Antenna ant;
}
```

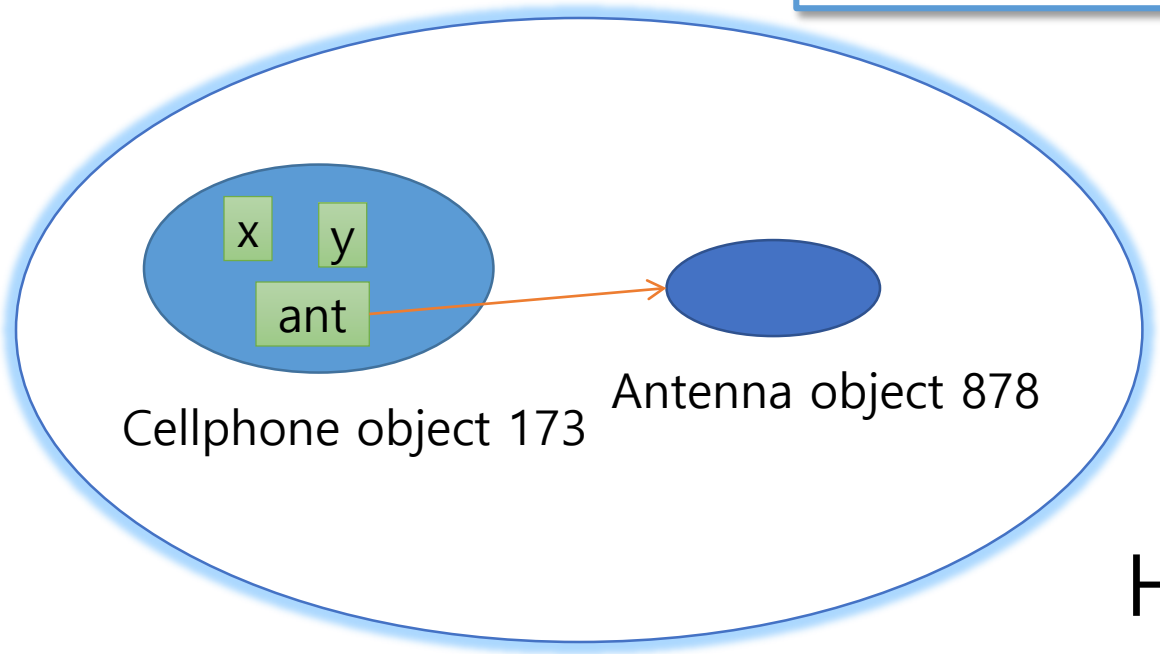


Table of Contents

Constructor

- Object creation

```
Duck myDuck = new Duck();
```

- Constructor
  - You cannot invoke a **constructor** without **new**.

```
Duck myDuck = new Duck();
```

- Constructor Definition
  - Constructors do not have a return type. Do not write *void*!

```
public Duck() { }
```

// default constructor

Construct a Duck

- Duck.java

```
public class Duck {
    int size;



}
```

- DuckAUse.java

```
public class UseADuck {
    public static void main (String[] args) {
        Duck d = new Duck();
    }
}
```

Construct a Duck

- Duck.java

```
public class Duck {
    int size;
    public Duck() {

    }
}
```

- DuckAUse.java

```
public class UseADuck {
    public static void main (String[] args) {
        Duck d = new Duck();
    }
}
```

Construct a Duck

- Duck.java

```
public class Duck {
    int size;
    public Duck() {
        System.out.println("Quack");
    }
}
```

```
% java UseADuck
Quack

%
```

- DuckAUse.java

```
public class UseADuck {
    public static void main (String[] args) {
        Duck d = new Duck();
    }
}
```

Initializing object state (1)

```
public class Duck {
    int size;

    public Duck() {
        System.out.println("Quack");
    }

    public void setSize(int newSize) {
        size = newSize;
    }
}
```

```
public class UseADuck {
    public static void main(String[] args) {
        Duck d = new Duck();
        d.setSize(42);
    }
}
```

Initializing object state (2)

```
public class Duck {
    int size;

    public Duck(int duckSize) {
        System.out.println("Quack");
        size = duckSize;
    }



}
```

```
public class UseADuck {
    public static void main(String[] args) {
        Duck d = new Duck(42);

    }
}
```

Overloaded Constructor (2)

```
public class Duck {
    int size;

    public Duck(int duckSize) {
        System.out.println("Quack");
        size = duckSize;
    }

}
```

If a class has a constructor, the Java compiler does not write a default constructor automatically.

```
public class UseADuck {
    public static void main(String[] args) {
        Duck d = new Duck(42);
        Duck d2 = new Duck();
    }
}
```

Overloaded Constructor (1)

```
public class Duck {
    int size;

    public Duck() {
        size = 27;
    }

    public Duck(int duckSize) {
        size = duckSize;
    }
}
```

```
public class UseADuck {
    public static void main(String[] args) {
        Duck d = new Duck(42);
        Duck d2 = new Duck();
    }
}
```

Overloaded Constructor (3)

- You can overload constructors.

```
public class Mushroom {
    public Mushroom(int size) { .. }
    public Mushroom() { .. }
    public Mushroom(boolean isMagic) { .. }
    public Mushroom(boolean isMagic, int size) { .. }
    public Mushroom(int size, boolean isMagic) { .. }
}
```

Overloaded Constructor (4)

- More complicated example

```
class Animal { }
class Dog extends Animal { }
class Puppy extends Dog { }

class Alpha {
    Alpha() { System.out.println("Alpha()"); }
    Alpha(Animal a) { System.out.println("Alpha(Animal a)"); }
    Alpha(Dog d) {System.out.println("Alpha(Dog d)"); }
    Alpha(Animal a, Dog d) {
        System.out.println("Alpha(Animal a, Dog d)");
    }
    Alpha(Dog d, Animal a) {
        System.out.println("Alpha(Dog d, Animal a)");
    }
}
```

Overloaded Constructor (5)

- More complicated example

```
Puppy p = new Puppy();
Dog d = p;
Animal a = d;

Alpha ap1 = new Alpha(a);
Alpha ap2 = new Alpha(d);
Alpha ap3 = new Alpha(p);
Alpha ap4 = new Alpha(a, d);
Alpha ap5 = new Alpha(d, a);
```

```
class Animal { }
class Dog extends Animal { }
class Puppy extends Dog { }
class Alpha {
    Alpha() { ... }
    Alpha(Animal a) { ... }
    Alpha(Dog d) { ... }
    Alpha(Animal a, Dog d) { ... }
    Alpha(Dog d, Animal a) { ... }
}
```

```
Alpha(Animal a)
Alpha(Dog d)
Alpha(Dog d)
Alpha(Animal a, Dog d)
Alpha(Dog d, Animal a)
```

Overloaded Constructor (6)

- More complicated example

```
Puppy p = new Puppy();
Dog d = p;
Animal a = d;

Alpha ap1 = new Alpha(a);
Alpha ap2 = new Alpha(d);
Alpha ap3 = new Alpha(p);
Alpha ap4 = new Alpha(a, d);
Alpha ap5 = new Alpha(d, a);
Alpha ap6 = new Alpha(p, p);
```

```
class Animal { }
class Dog extends Animal { }
class Puppy extends Dog { }
class Alpha {
    Alpha() { ... }
    Alpha(Animal a) { ... }
    Alpha(Dog d) { ... }
    Alpha(Animal a, Dog d) { ... }
    Alpha(Dog d, Animal a) { ... }
}
```

```
Alpha(Animal a)
Alpha(Dog d)
Alpha(Dog d)
Alpha(Animal a, Dog d)
Alpha(Dog d, Animal a)
```

Overloaded Constructor (6)

- More complicated example

```
Puppy p = new Puppy();
Dog d = p;
Animal a = d;

Alpha ap1 = new Alpha(a);
Alpha ap2 = new Alpha(d);
Alpha ap3 = new Alpha(p);
Alpha ap4 = new Alpha(a, d);
Alpha ap5 = new Alpha(d, a);
Alpha ap6 = new Alpha(p, p);
```

```
class Animal { }
class Dog extends Animal { }
class Puppy extends Dog { }
class Alpha {
    Alpha() { ... }
    Alpha(Animal a) { ... }
    Alpha(Dog d) { ... }
    Alpha(Animal a, Dog d) { ... }
    Alpha(Dog d, Animal a) { ... }
}
```

```
reference to Alpha is ambiguous, both method Alpha(Animal, Dog) in
alpha and method Alpha(Dog, Animal) in Alpha match
              Alpha ap6 = new Alpha(p, p);
1 error
```

Overloaded Constructor (7)

- Invoking one overloaded constructor from another

```
public class Duck {
    int size;

    public Duck() {
        size = 27;
    }

    public Duck(int duckSize) {
        size = duckSize;
    }
}
```

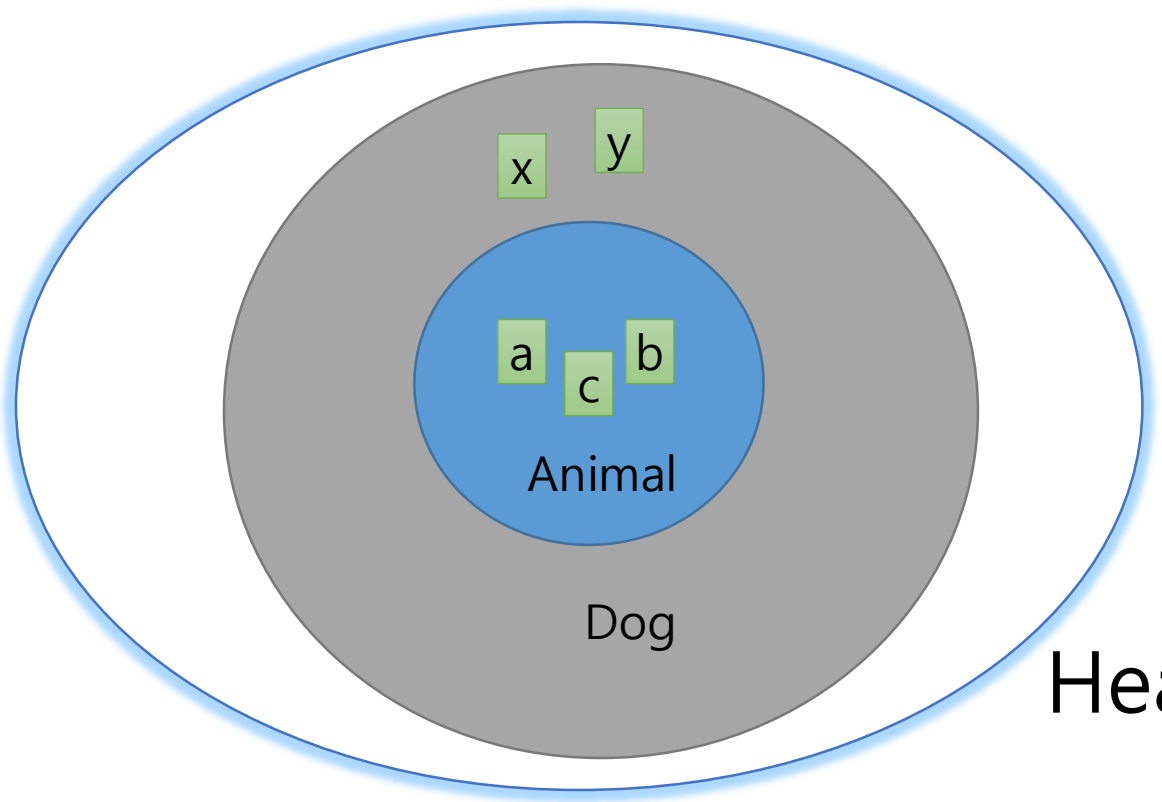
Overloaded Constructor (7)

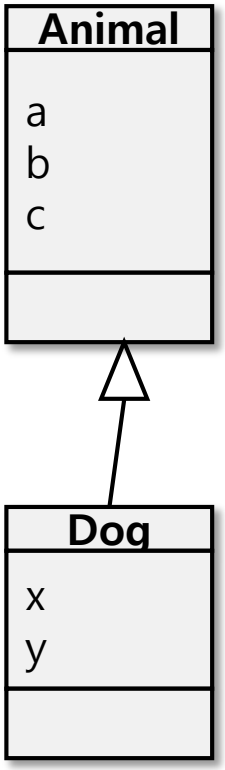
- Invoking one overloaded constructor from another
  - **this()**

```
public class Duck {
    int size;

    public Duck() {
        this(27);
    }

    public Duck(int duckSize) {
        size = duckSize;
    }
}
```

Superclass Constructor (1)

- Constructor Chaining
  - If you want to make a Dog, you must also make an Animal.





Programmer A

Programmer B

Superclass Constructor (2)

```
public class Animal {
    public Animal() {
        System.out.println("Making an Animal");
    }
}
```

```
public class Hippo extends Animal {
    public Hippo() {
        System.out.println("Making an Hippo");
    }
}
```

```
public class TestHippo {
    public static void main(String[] args) {
        System.out.println("Starting...");
        Hippo h = new Hippo();
    }
}
```

```
$ java TestHippo
Starting...
Making an Animal
Making a Hippo
```

```
INVOKE: main()
 OUTPUT: Starting...
 INVOKE: Hippo()
  INVOKE: Animal()
   OUTPUT: Making an Animal
  RETURN: Animal()
  OUTPUT: Making an Hippo
 RETURN: Hippo()
RETURN: main()
```

Superclass Constructor (3)

```
public class Animal {
    public Animal() {
        System.out.println("Making an Animal");
    }
}
```

```
public class Hippo extends Animal {
    public Hippo() {
        super();
        System.out.println("Making an Hippo");
    }
}
```

```
public class TestHippo {
    public static void main(String[] args) {
        System.out.println("Starting...");
        Hippo h = new Hippo();
    }
}
```

```
$ java TestHippo
Starting...
Making an Animal
Making a Hippo
```

```
INVOKE: main()
 OUTPUT: Starting...
 INVOKE: Hippo()
  INVOKE: Animal()
   OUTPUT: Making an Animal
  RETURN: Animal()
  OUTPUT: Making an Hippo
 RETURN: Hippo()
RETURN: main()
```

Superclass Constructor (4)

```
public Boop() {

}
```

```
public Boop() {
    super();
}
```

```
public Boop(int i) {

    size = i;
}
```

```
public Boop(int i) {
    super();
    size = i
}
```

```
public Boop(int i) {
    size = i;
    super(); // ERROR
}
```

Superclass Constructor (5)

```
public abstract class Animal {
    private String name;
    public String getName() {
        return name;
    }
    public Animal() {
        name = null;
    }
}
```

```
public class Hippo extends Animal {
    public Hippo(String name) {

    }
}
```

```
public class MakeHippo {
    public static main(String[] args) {
        Hippo h = new Hippo("Buffy");
        System.out.println(h.getName());
    }
}
```

Successfully compiled !!

Superclass Constructor (5)

```
public abstract class Animal {
    private String name;
    public String getName() {
        return name;
    }
    public Animal(String theName) {
        name = theName;
    }
}
```

```
public class Hippo extends Animal {
    public Hippo(String name) {
        super();
    }
}
```

```
public class MakeHippo {
    public static main(String[] args) {
        Hippo h = new Hippo("Buffy");
        System.out.println(h.getName());
    }
}
```

Compilation error !!

Superclass Constructor (5)

```
public abstract class Animal {
    private String name;
    public String getName() {
        return name;
    }
    public Animal(String theName) {
        name = theName;
    }
}
```

```
public class Hippo extends Animal {
    public Hippo(String name) {
        super(name);
    }
}
```

```
public class MakeHippo {
    public static main(String[] args) {
        Hippo h = new Hippo("Buffy");
        System.out.println(h.getName());
    }
}
```

```
$ java MakeHippo
Buffy
```

Superclass Constructor (6)

```
import java.awt.Color;
class Mini extends Car {
    Color color;

    public Mini() {
        this(Color.RED);
    }

    public Mini(Color c) {
        super("Mini");
        color = c;
    }

    public Mini(int size) {
        this(Color.RED);
        super(size);
    }
}
```

```
$ javac Mini.java
Mini.java:16 call to super must be
first statement in constructor
    super()
```

Table of Contents

Object Lifespan (1)

- Local variable

```
public void read() {
    int s = 42; // Stack
}
```

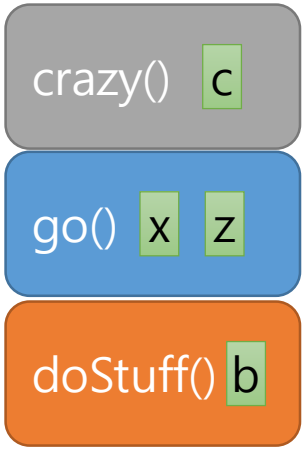
- Instance variable (member variable)

```
public class Life {
    int size; // Heap
    public void setSize(int s) { // Stack
        size = s;
    }
}
```

Object Lifespan (2)

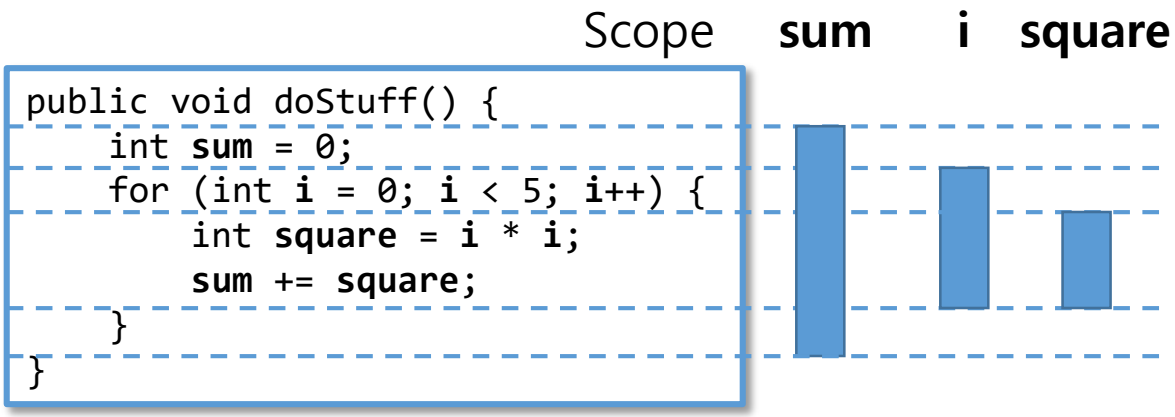
- Life
  - A local variable is alive until the method completes.
- Scope
  - You can use a variable only when it is in scope.

```
public void doStuff() {
    boolean b = true;
    go(4);
}
public void go(int x) {
    int z = x + 24;
    crazy();
}
public void crazy() {
    char c = 'a';
}
```



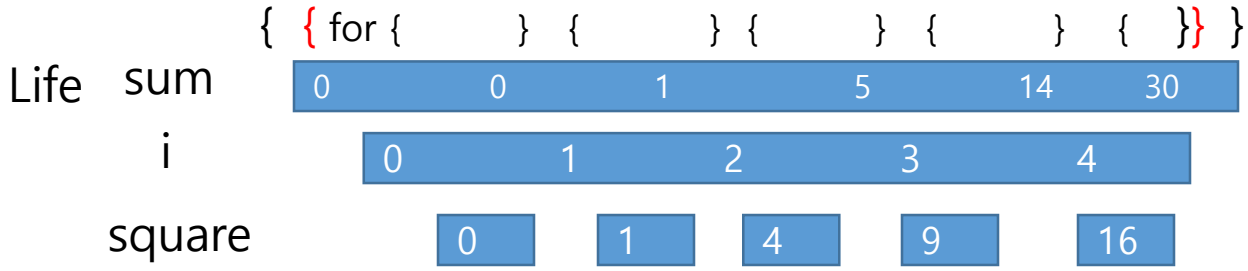
Object Lifespan (3)

- Scope
  - You can use a variable only when it is in scope.



Object Lifespan (4)

- Life
  - A local variable is alive until the method completes.




```
public void doStuff() {
    int sum = 0;
    for (int i = 0; i < 5; i++) {
        int square = i * i;
        sum += square;
    }
}
```

Object Lifecycle (1)

- Garbage Collection(GC)
  - An object becomes *eligible for Garbage Collection* when its last live reference disappears.
  - If your program gets low on memory, Garbage Collection will destroy some or all of the eligible objects.

Object Lifecycle (2)

- Ways to get rid of an object's reference:

```
void go() {
    Life z = new Life();
    // reference 'z' dies at the end of method.
}
```

```
Life z = new Life();
z = new Life(); // the first object is abandoned,
                // when z is referring to a new object.
```

```
Life z = new Life();
z = null; // the first object is abandoned,
          // when z is referring to null.
```

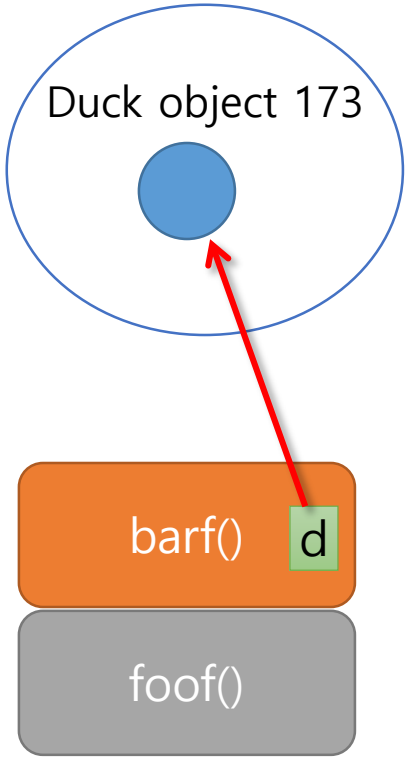
```
if (k > 3) {
    Life z = new Life();
    // reference 'z' dies at end of block.
}
```

Object Lifecycle (3)

This object is abandoned. It becomes a garbage, and will be collected later.

- Example #1

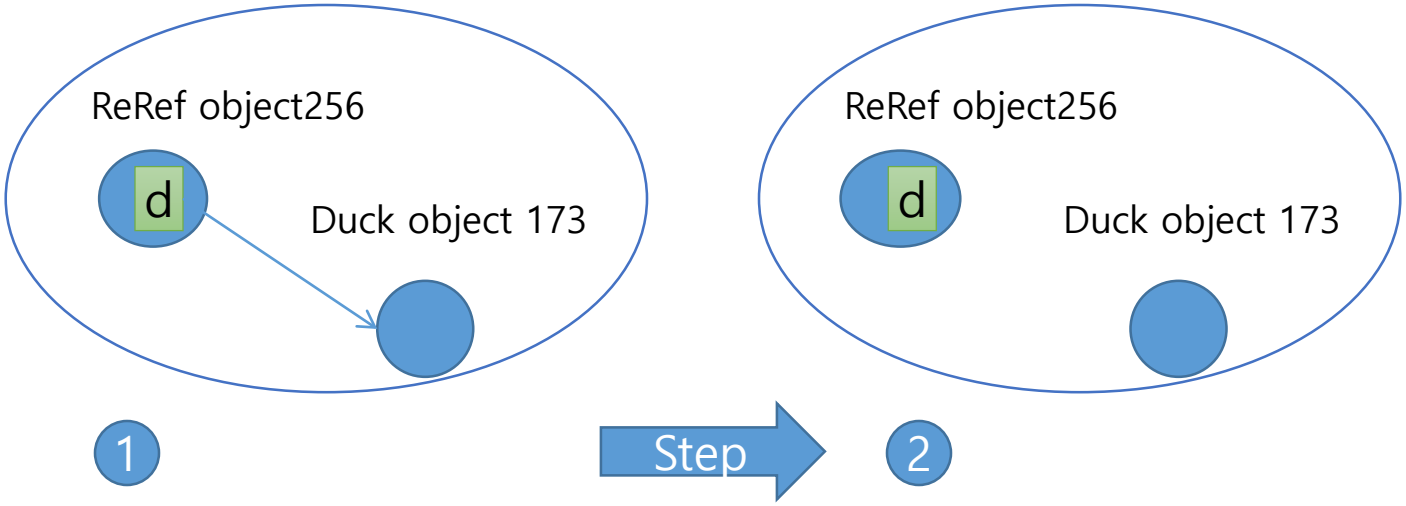
```
public class StackRef {
    public void foof() {
        barf();
    }
    public void barf() {
        Duck d = new Duck();
    }
}
```



Object Lifecycle (4)

- Example #2

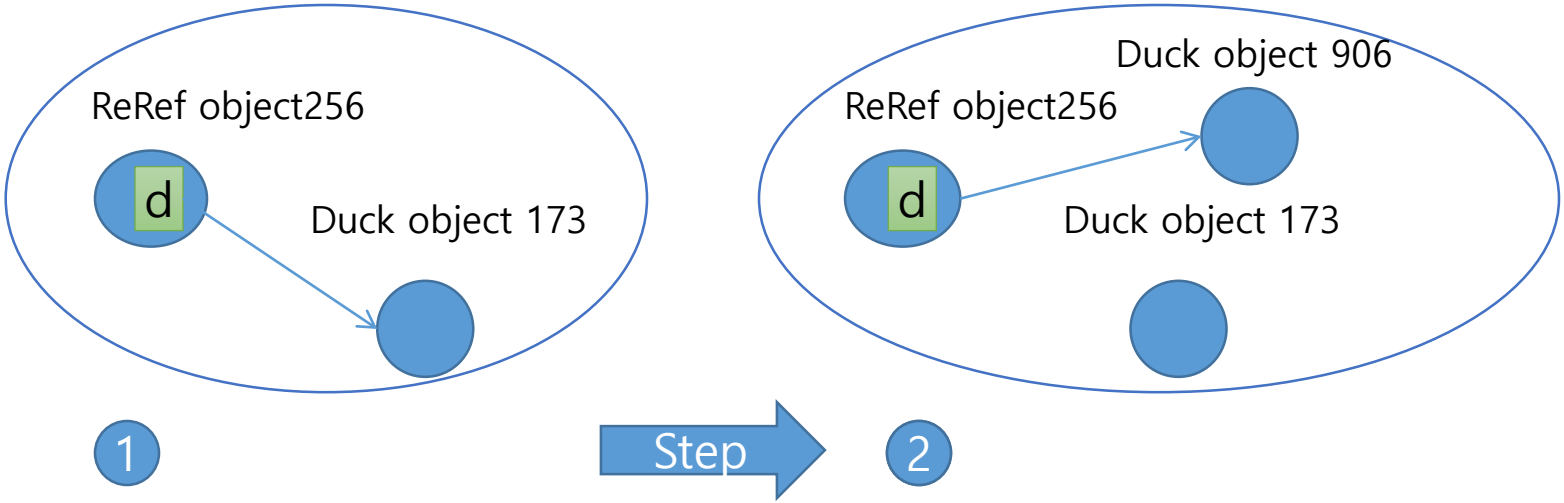
```
public class ReRef {
    Duck d = new Duck();
    public void go() {
        d = null;
    }
}
```



Object Lifecycle (5)

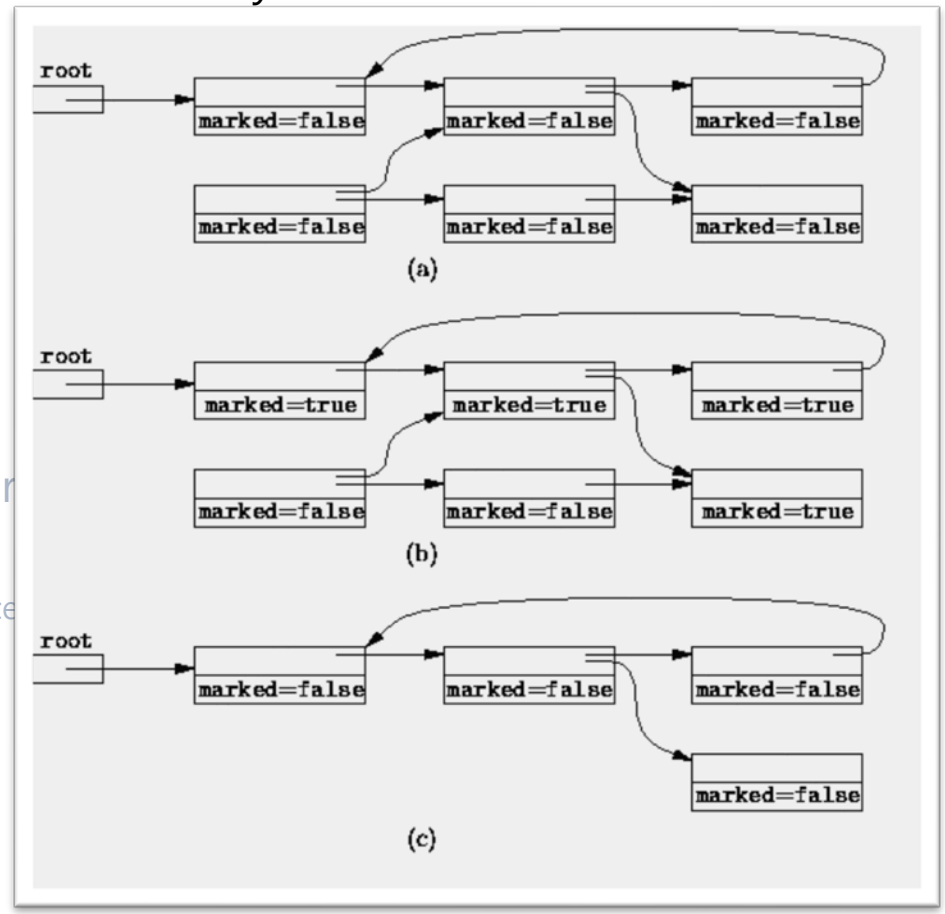
- Example #3

```
public class ReRef {
    Duck d = new Duck();
    public void go() {
        d = new Duck();
    }
}
```



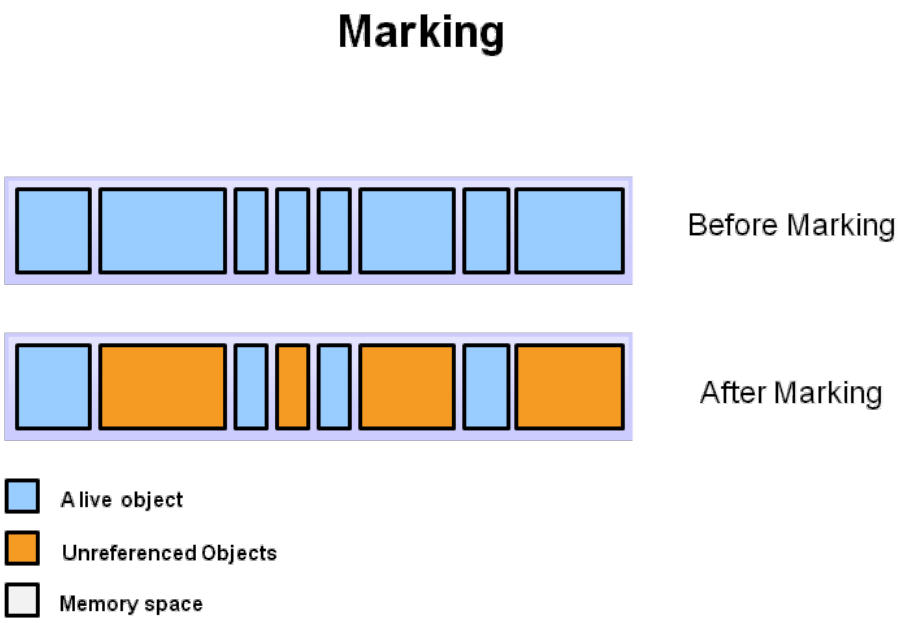
Garbage Collection Techniques

- Reference Counting Collectors
  - cannot handle "cycling". not used for Java any more.
- Tracing Collectors
  - called "mark-and-sweep"
  - Reducing heap fragmentation
    - Compacting
    - Copying
- Generational Collectors
  - Three generations of heap memor
    - Young generation
      - Eden, Survivor To, Survivor From space
    - Tenured or old generation
    - Permanent generation



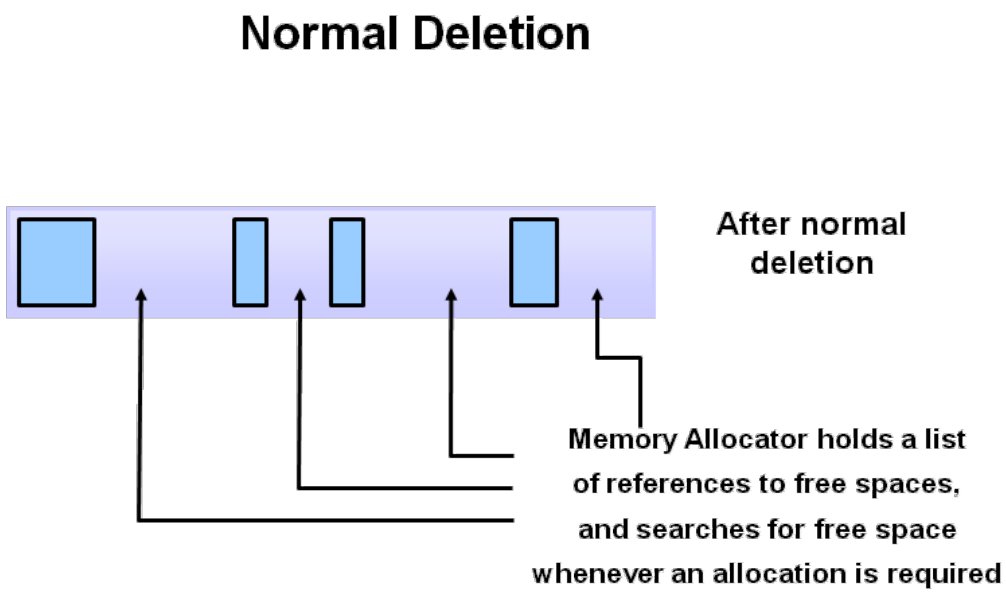
Java Garbage Collector

- Step 1: Marking



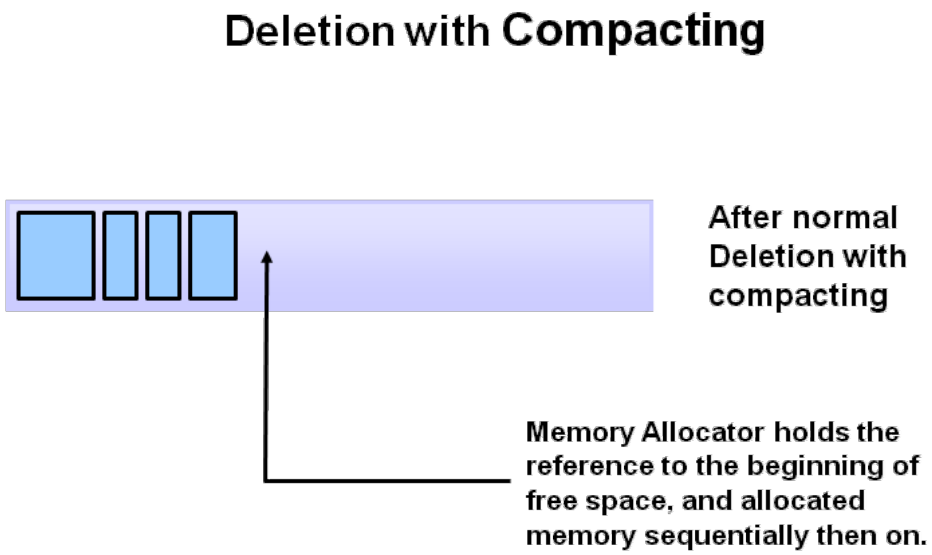
Java Garbage Collector

- Deletion

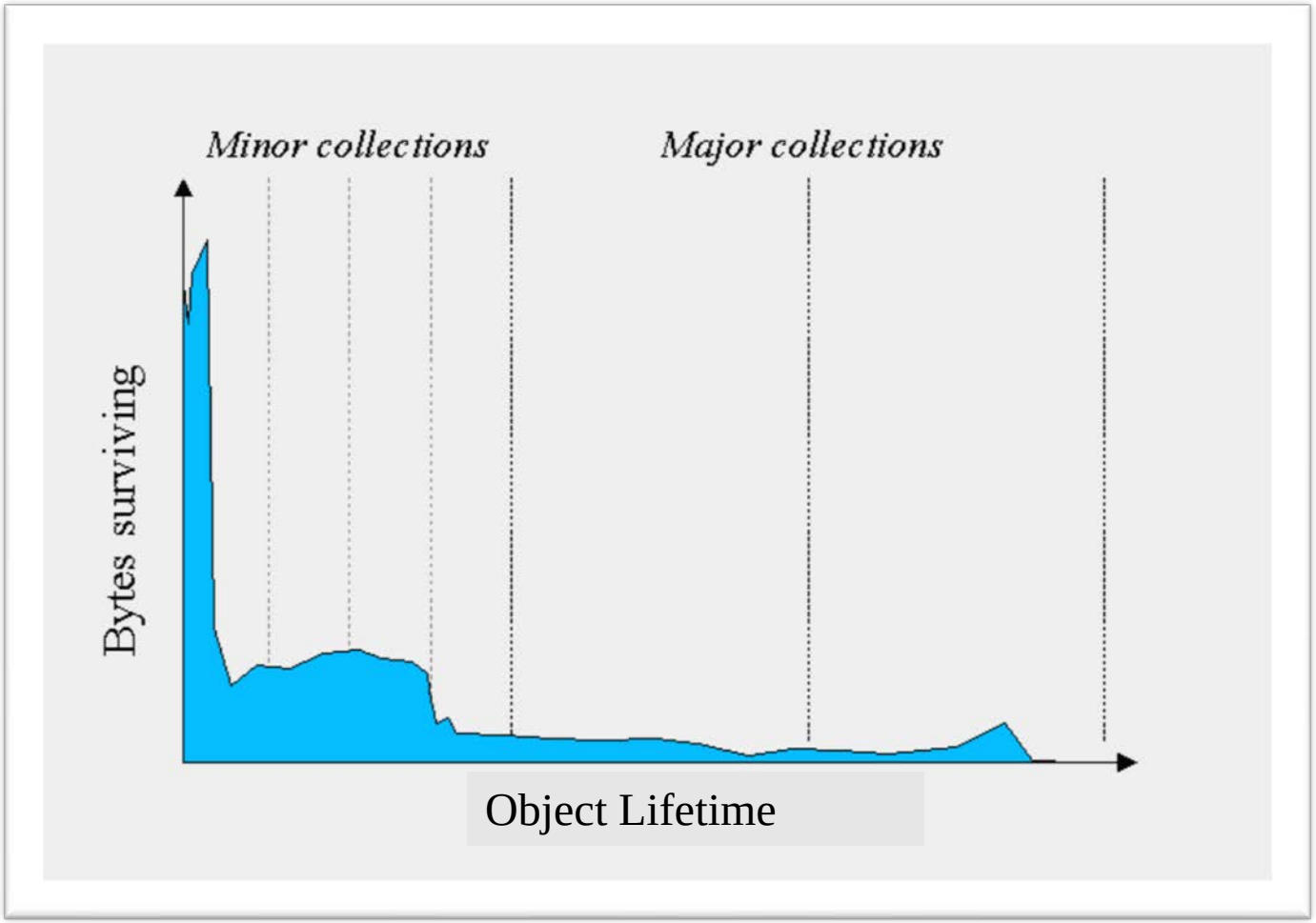


Java Garbage Collector

- Deletion

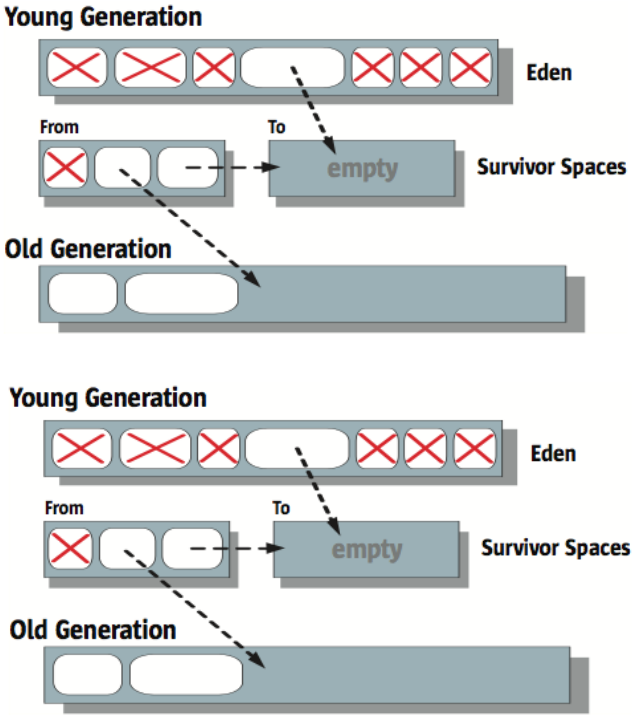


Generational Garbage Collection



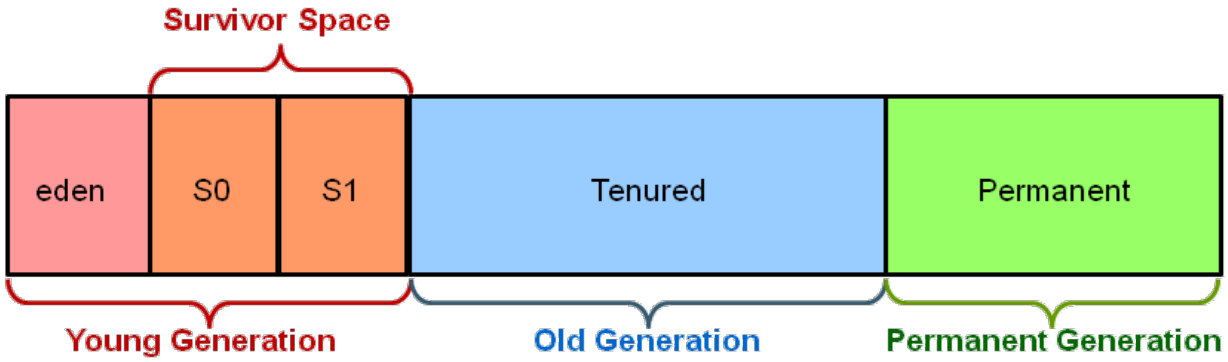
Garbage Collection Techniques

- Reference Counting Collectors
  - cannot handle "cycling". not used for Java any more.
- Tracing Collectors
  - called "mark-and-sweep"
  - Reducing heap fragmentation
    - Compacting
    - Copying
- Generational Collectors
  - Three generations of heap memory
    - Young generation
      - Eden, Survivor To, Survivor From spaces.
    - Tenured or old generation
    - Permanent generation



Generational Garbage Collection

Hotspot Heap Structure

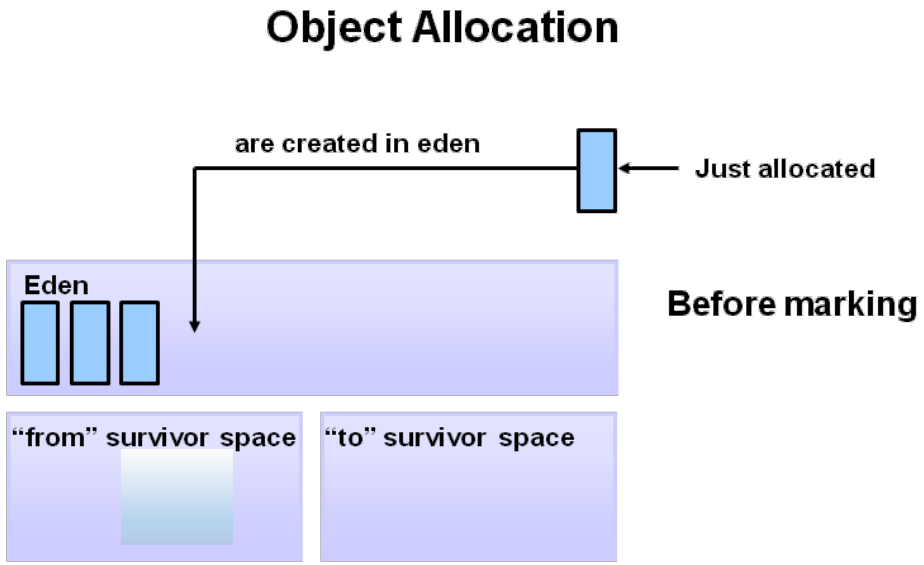


Usual Ratios (from Total Available Heap Memory)	
Young Generation	1/3
Survivor 1 (From)	1/8
Survivor 2 (To)	1/8
Eden	3/4
Older Generation	2/3

Methods (in bytecodes)
Constant pool information
Internal objects created by JVM, etc.

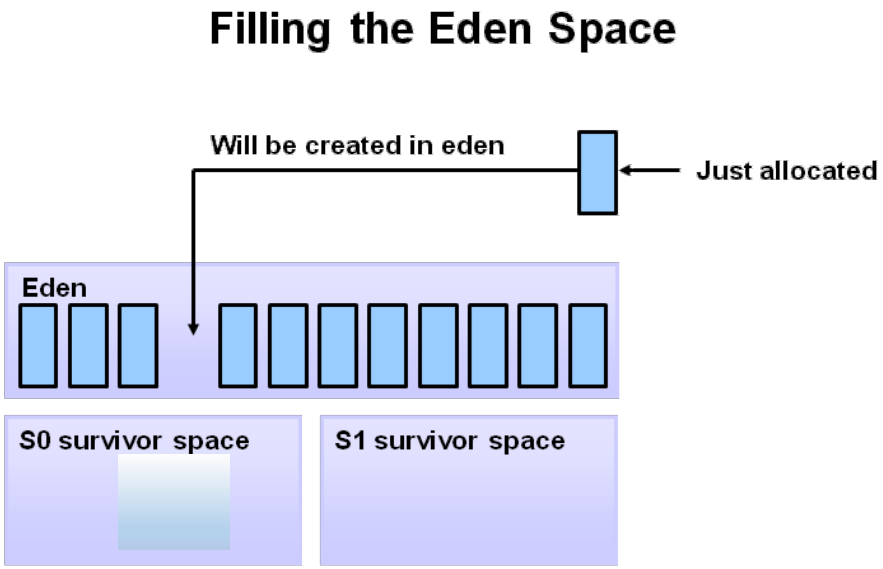
Generational Garbage Collection Process

- First, any new objects are allocated to the eden space. Both survivor spaces start out empty.



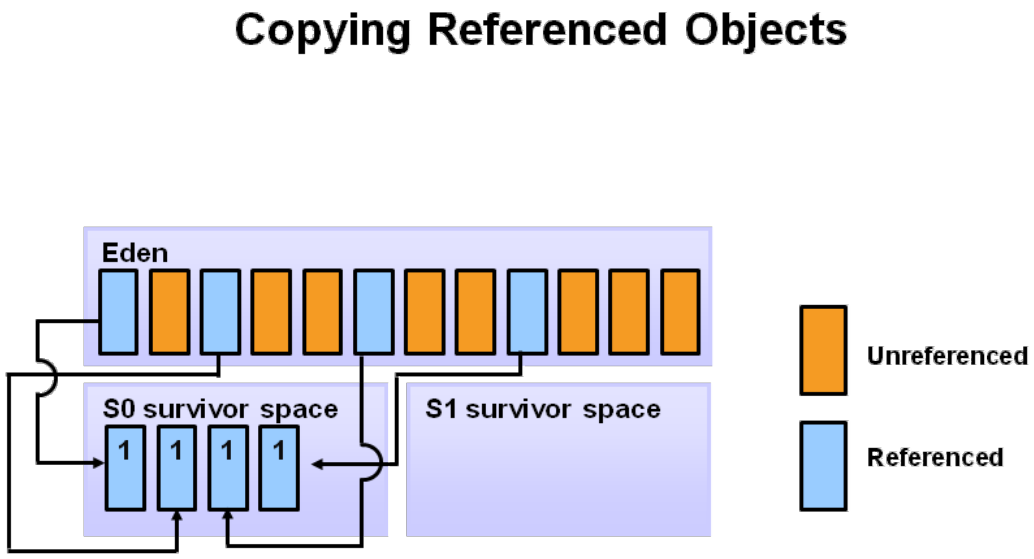
Generational Garbage Collection Process

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



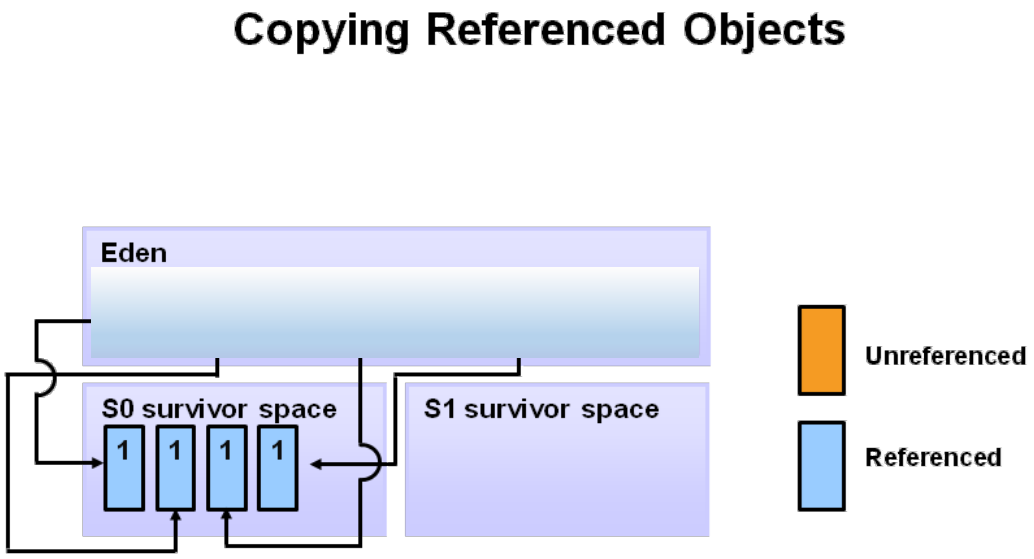
Generational Garbage Collection Process

- Referenced objects are moved to the first survivor space. Unreferenced objects are deleted when the eden space is cleared



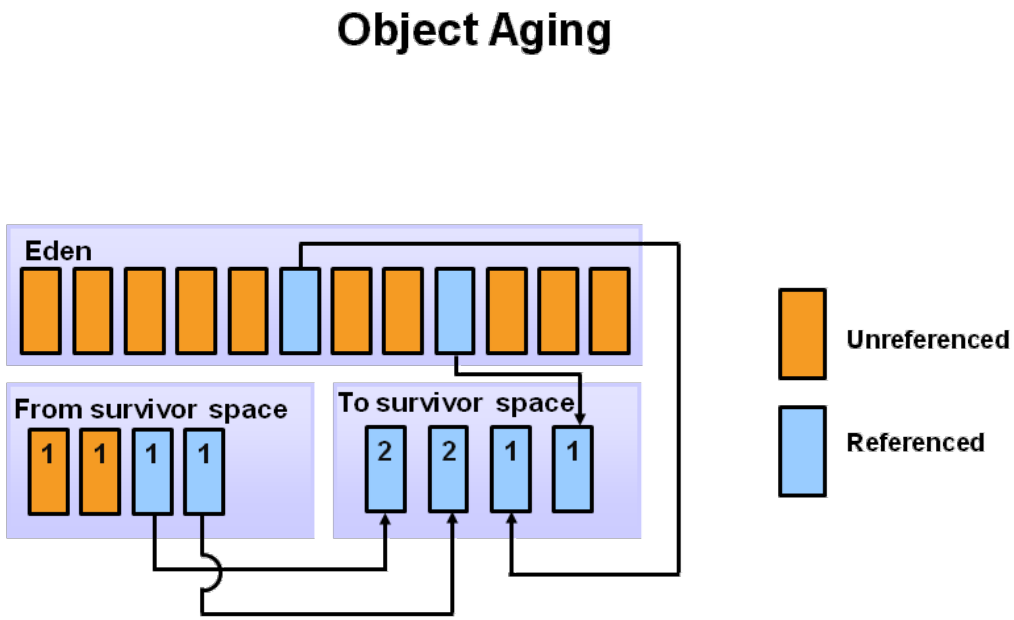
Generational Garbage Collection Process

- Referenced objects are moved to the first survivor space. Unreferenced objects are deleted when the eden space is cleared



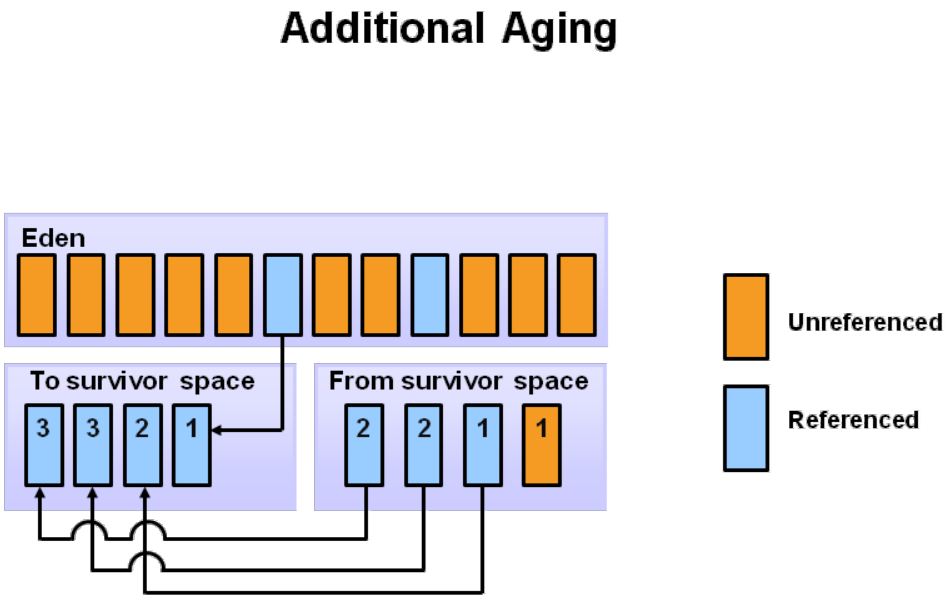
Generational Garbage Collection Process

- Referenced objects are moved to the second survivor space (S1). In addition, objects from the last minor GC on the first survivor space (S0) have their age incremented and get moved to S1.



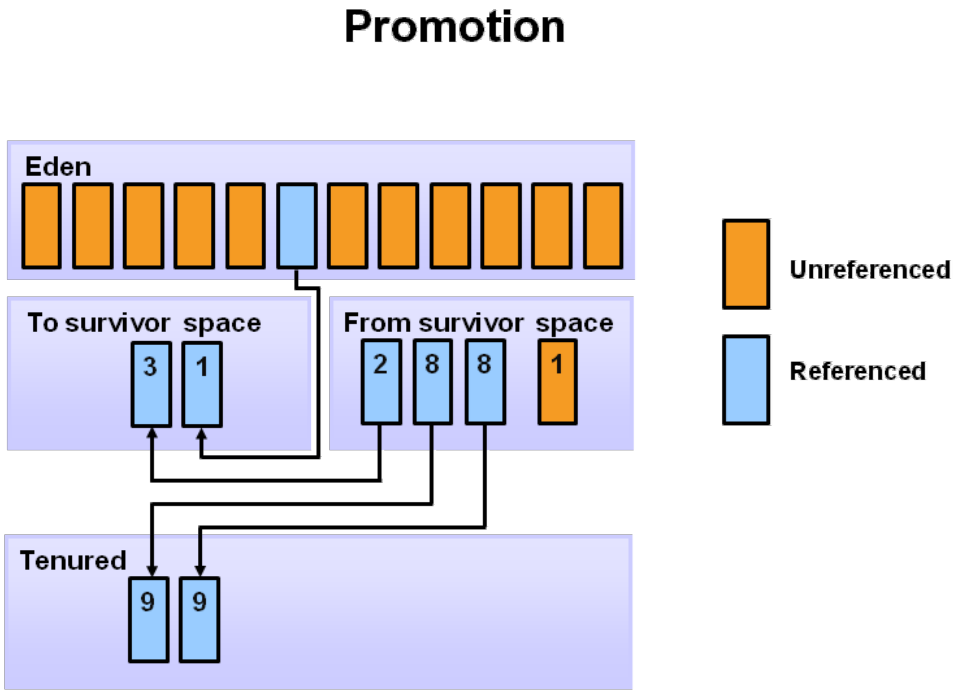
Generational Garbage Collection Process

- The same process repeats. However this time the survivor spaces switch. Referenced objects are moved to S0. Surviving objects are aged. Eden and S1 are cleared.



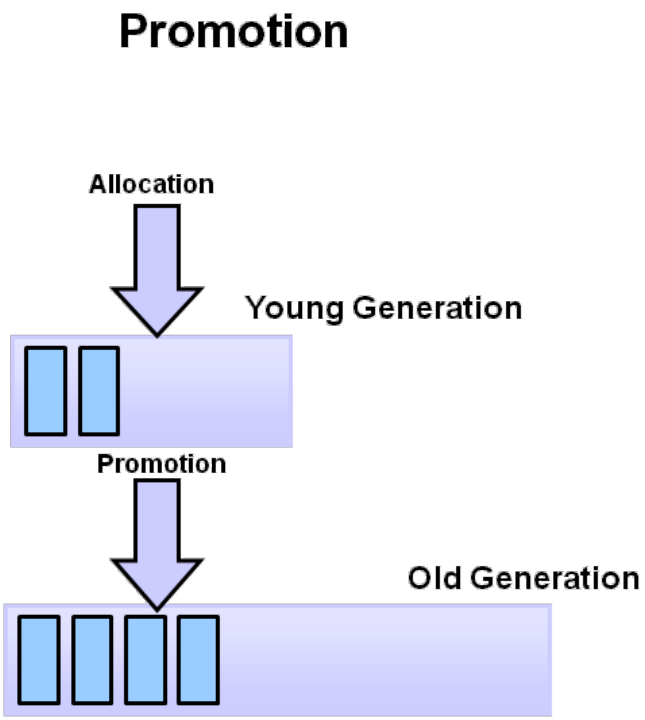
Generational Garbage Collection Process

- After a minor GC, when aged objects reach a certain age threshold (8 in this example) they are promoted from young generation to old generation.



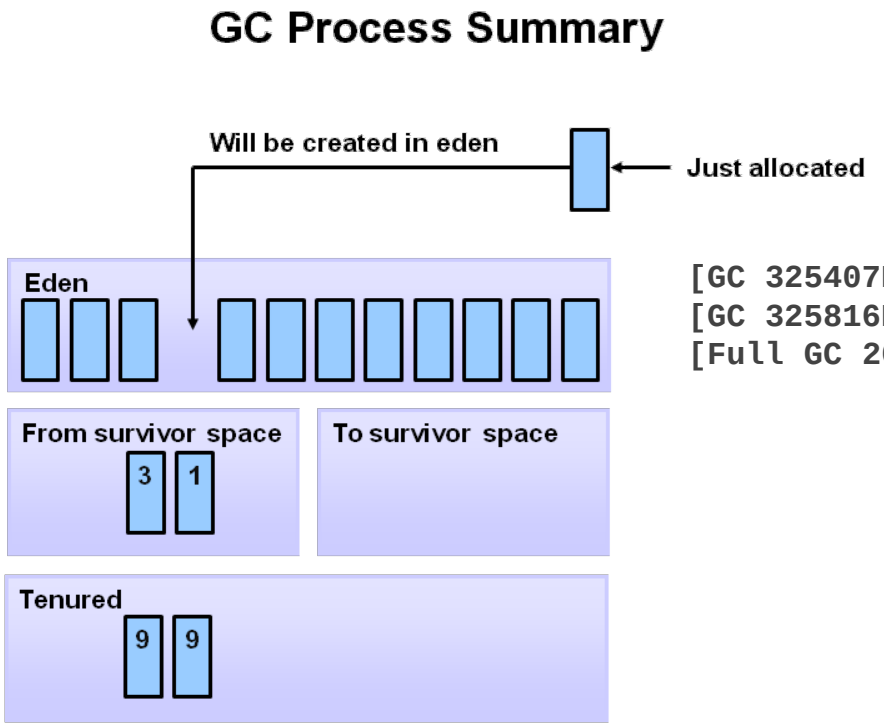
Generational Garbage Collection Process

- As **minor GCs** continue to occur, objects will continue to be promoted to the old generation space.



Generational Garbage Collection Process

- So that pretty much covers the entire process with the young generation. Eventually, a **major GC** will be performed on the old generation which cleans up and compacts that space.




```
[GC 325407K->83000K(776768K), 0.2300771 secs]
[GC 325816K->83372K(776768K), 0.2454258 secs]
[Full GC 267628K->83769K(776768K), 1.8479984 secs]
```

References

- Kathy Sierra and Bert Bates, *Head First Java*, O'Reilly, 2005.
- Java Tutorials
  - http://docs.oracle.com/javase/tutorial/
- Java Platform, Standard Edition 7 API Specification
  - http://docs.oracle.com/javase/7/docs/api/