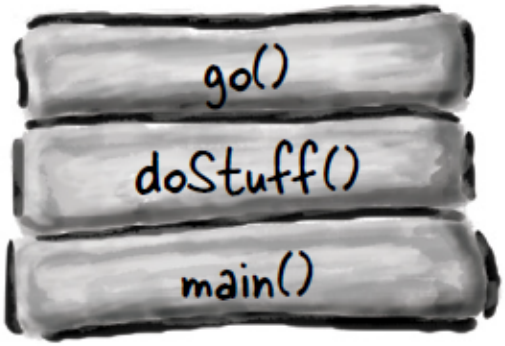
Constructors and garbage collection

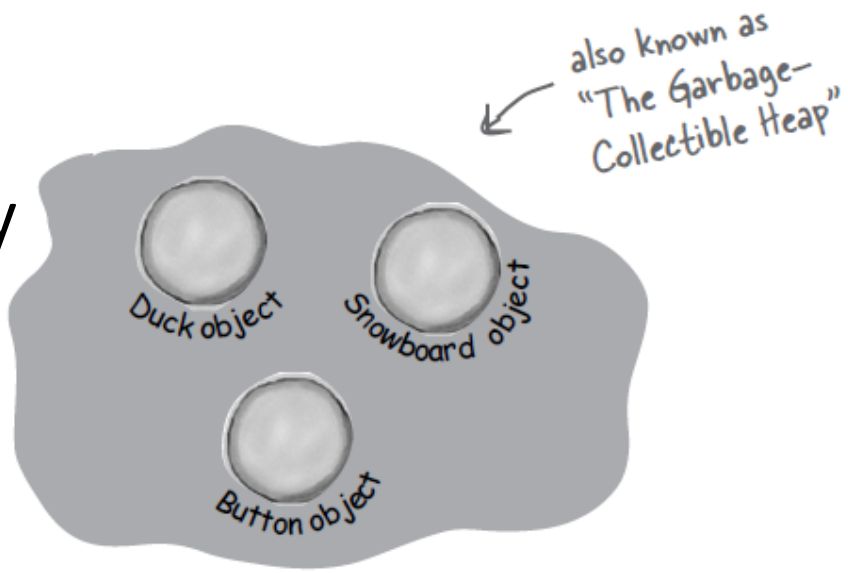
Overview

- Where Java stores stuff
  - The Heap
  - The Stack
  - Breaking both
  - Java Garbage Collector (GC)
- Creating objects
  - Copy constructors
  - Methods creating new objects

Stack and Heap

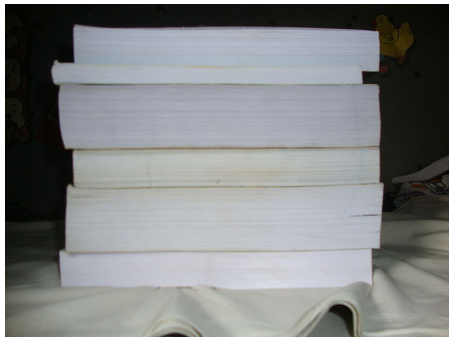
- Stack
  - Where method invocations live
  - Where local variables live
  - A chunk of memory
- Heap
  - Where all objects live
  - Another chunk of memory





The Stack

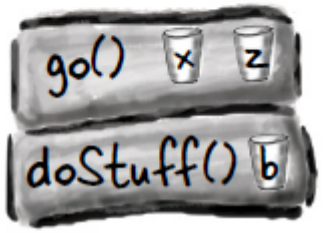
- Methods are stacked
  - When you call a method, the method lands on top of a call stack
  - Stack frame
    - Contains the state of the method
    - What line it is executing
    - Values of the method's local variables (including parameters)
  - Method stays on stack until it finishes executing



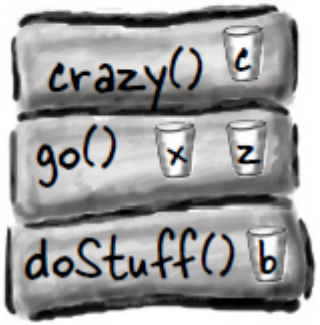
```
public void doStuff()
{
    boolean b = true;
    go(4);
}

public void go(int x)
{
    int z = x + 24;
    crazy();
}

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

1. Code from another class calls **doStuff()**, and **doStuff()** goes into a stack frame at the top of the stack. The boolean variable named '**b**' goes on the **doStuff()** stack frame.

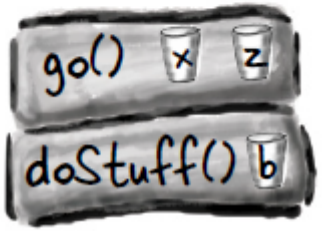
2. **doStuff()** calls **go()**, **go()** is *pushed* on top of the stack. Variables '**x**' and '**z**' are in the **go()** stack frame.



3. **go()** calls **crazy()**, **crazy()** is now on the top of the stack, with variable '**c**' in the frame.



4. **crazy()** completes, and its stack frame is *popped* off the stack. Execution goes back to the **go()** method, and picks up at the line following the call to **crazy()**.



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

public void go(int b)
{
    int c = b + 24;
    crazy();
}

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

1

```
doStuff()
boolean b → true
```

2

```
go()
int b → 4
int c → 28
```

```
doStuff()
boolean b → true
```

3

```
crazy()
char c → 'a'
```

```
go()
int b → 4
int c → 28
```

```
doStuff()
boolean b → true
```

4

```
go()
int b → 4
int c → 28
```

```
doStuff()
boolean b → true
```

A brief intro to recursion…

- Recursion
  - A method can call itself!
  - A extremely useful technique
  - But it can go wrong…

0! = 1
1! = 1
2! = 1 x 2 = 2
3! = 1 x 2 x 3 = 6
4! = 4 x 3 x 2 x 1 = 24

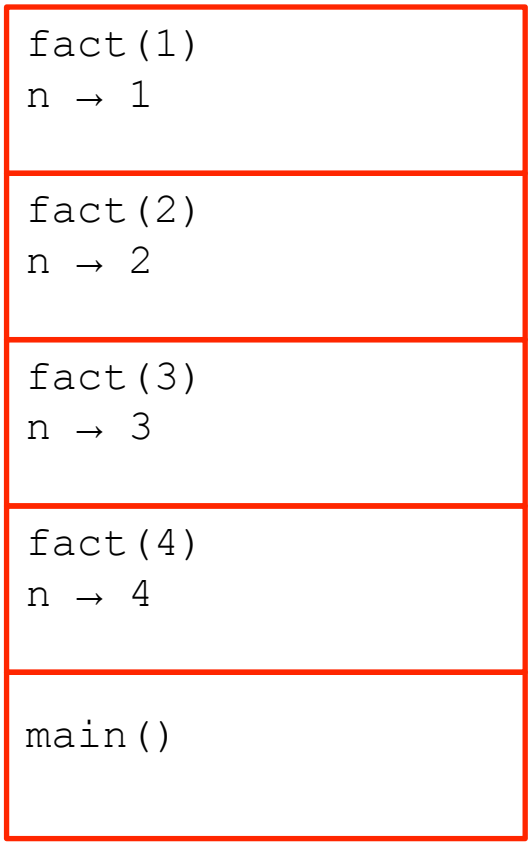
Some example factorials.

Computing a factorial recursively

```
public class Factorial
{
   public static long fact(long n)
   {
      if (n <= 1)
        return 1;
      return n * fact(n - 1);
   }

   public static void main(String [] args)
   {
      System.out.println("4! = " + fact(4));
   }
}
```

0! = 1
1! = 1
2! = 1 x 2 = 2
3! = 1 x 2 x 3 = 6
4! = 4 x 3 x 2 x 1 = 24

Some example factorials.



Computing a factorial recursively

```
public class Factorial
{
   public static long fact(long n)
   {
      if (n <= 1)
        return 1;
      return n * fact(n - 1);
   }

   public static void main(String [] args)
   {
      System.out.println("4! = " + fact(4));
   }
}
```

0! = 1
1! = 1
2! = 1 x 2 = 2
3! = 1 x 2 x 3 = 6
4! = 4 x 3 x 2 x 1 = 24

Some example factorials.

```
fact(1)
n → 1
return 1
```

```
fact(2)
n → 2
return 2 * fact(1)
```

```
fact(3)
n → 3
return 3 * fact(2)
```

```
fact(4)
n → 4
return 4 * fact(3)
```

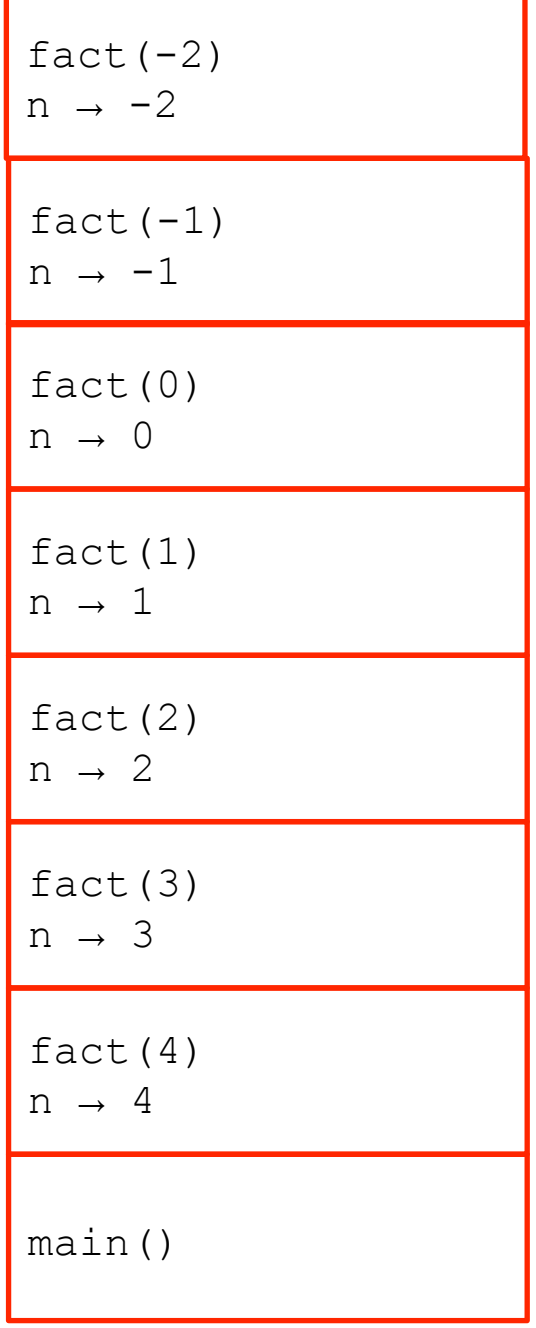
```
main()
fact(4) = 24
```

Breaking the stack

```
public class Factorial
{
   public static long fact(long n)
   {


      return n * fact(n - 1);
   }

   public static void main(String [] args)
   {
      System.out.println("4! = " + fact(4));
   }
}
```

0! = 1
1! = 1
2! = 1 x 2 = 2
3! = 1 x 2 x 3 = 6
4! = 4 x 3 x 2 x 1 = 24

Some example factorials.



Breaking the stack

```
public class Factorial
{
    public static long fa
    {


        return n * fact(n
    }

    public static void ma
    {
        System.out.println
    }
}
```

```
fact(-2)
n → -2

fact(-1)
```

```
Exception in thread "main"
java.lang.StackOverflowError
at Factorial.fact(Factorial.java:6)
at Factorial.fact(Factorial.java:6)
at Factorial.fact(Factorial.java:6)
at Factorial.fact(Factorial.java:6)
at Factorial.fact(Factorial.java:6)
at Factorial.fact(Factorial.java:6)
at Factorial.fact(Factorial.java:6)
at Factorial.fact(Factorial.java:6)
at Factorial.fact(Factorial.java:6)
at Factorial.fact(Factorial.java:6)
at Factorial.fact(Factorial.java:6)
at Factorial.fact(Factorial.java:6)
at Factorial.fact(Factorial.java:6)
at Factorial.fact(Factorial.java:6)
at Factorial.fact(Factorial.java:6)
at Factorial.fact(Factorial.java:6)
at Factorial.fact(Factorial.java:6)
at Factorial.fact(Factorial.java:6)
...
```

0! = 1
1! = 1
2! = 1 x 2 = 2
3! = 1 x 2 x 3 = 6
4! = 4 x 3 x 2 x 1 = 2

Some example facto

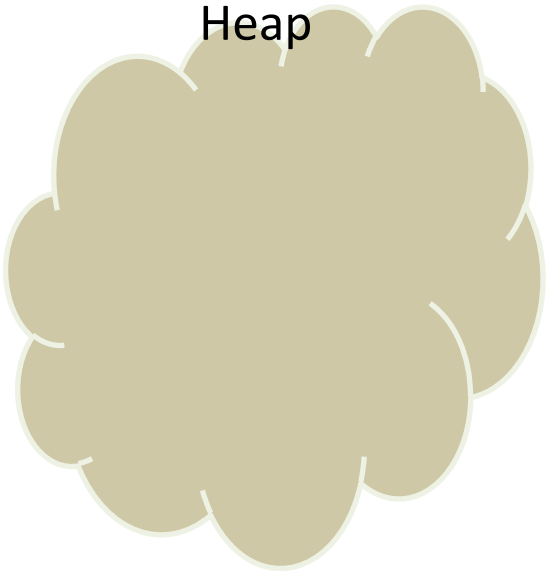
Where things live

- Local variables live on the ***stack***
  - Local variables in a method
  - Parameters passed to a method
  - Reference variables
    - Only the remote control lives on the stack
- Objects live on the ***heap***
  - When you create objects, they live on the heap
  - Instance variables of an object
    - Stored as part of the object living on the heap

The miracle of object creation

- Creating an object in 3 easy steps:
  1) Declare a reference variable

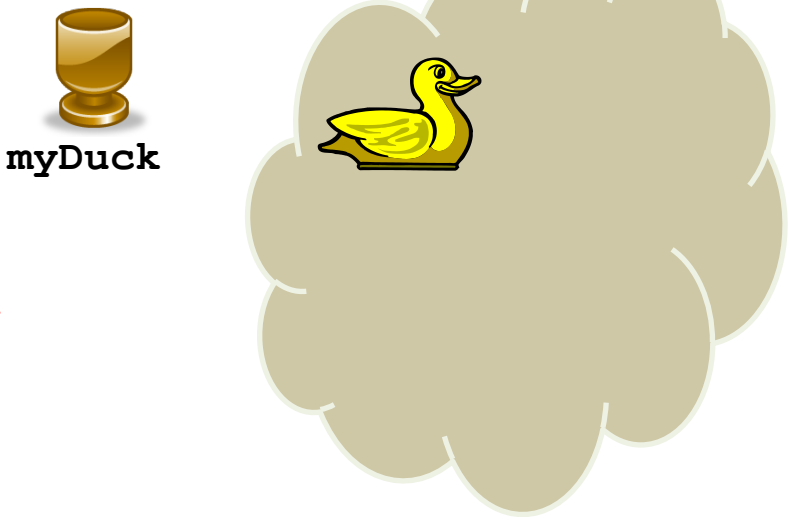
myDuck




```
Duck myDuck
```

The miracle of object creation

- Creating an object in 3 easy steps:
  1) Declare a reference variable
  2) Create an object

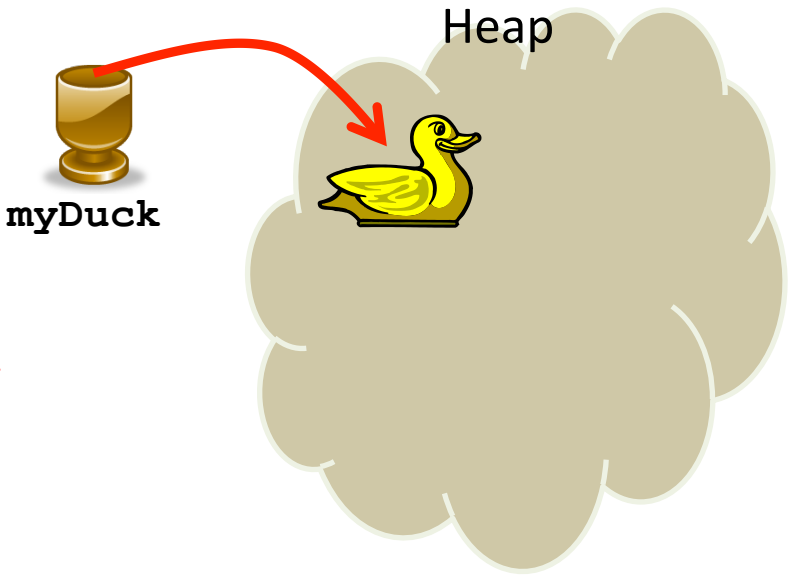



```
new Duck();
```

The miracle of object creation

- Creating an object in 3 easy steps:
  1) Declare a reference variable
  2) Create an object
  3) Link the object and the reference

You have now consumed memory on the heap based on how much state a Duck requires to be stored (i.e. the number and type of its instance variables




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

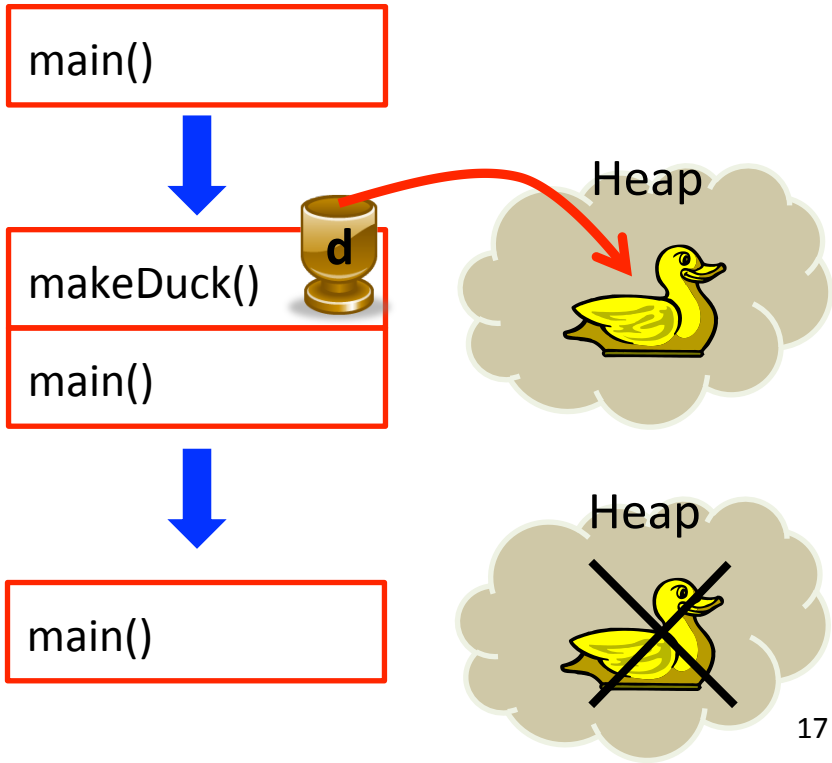
Java Garbage Collector

- Java tracks # of variables referring to an object
  - Once no one refers to an object, its memory can be freed
  - Java's automatic Garbage Collector (GC) periodically handles this for you
  - Enjoy it (while it lasts)

Killing objects

- Object-killer #1
  - Reference goes out of scope permanently

```
public class DuckKiller1
{
    public static void makeDuck()
    {
        Duck d = new Duck();
    }

    public static void main(String [] args)
    {
        makeDuck();
        while (true)
        {
            // Do something
        }
    }
}
```



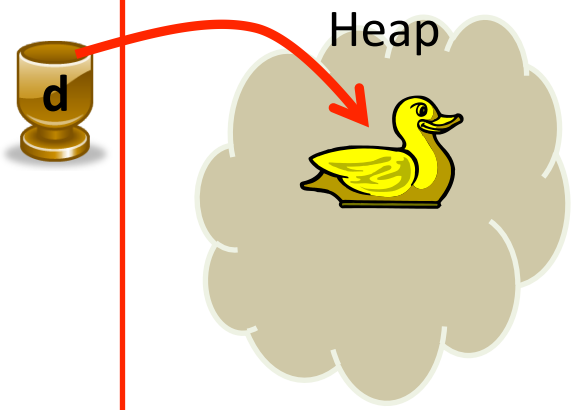
Killing objects

- Object-killer #1b
  - Variable inside curly braces goes out of scope

```
00 public class DuckKiller1b
01 {
02   public static void main(String [] args)
03   {
04      if (args.length >= 0)
05      {
06         Duck d = new Duck();
07      }
08      while (true)
09      {
10         // Do something
11      }
12   }
13 }
```

main()

line 04
line 05
line 06



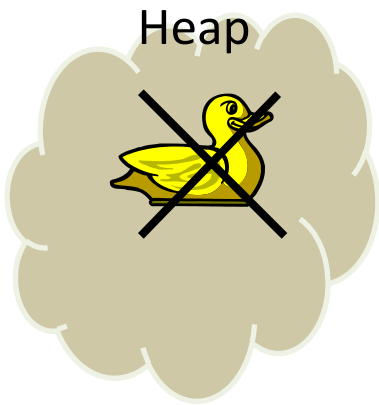
Killing objects

- Object-killer #1b
  - Reference goes out of scope permanently

```
00 public class DuckKiller1b
01 {
02   public static void main(String [] args)
03   {
04      if (args.length >= 0)
05      {
06         Duck d = new Duck();
07      }
08      while (true)
09      {
10         // Do something
11      }
12   }
13 }
```

main()

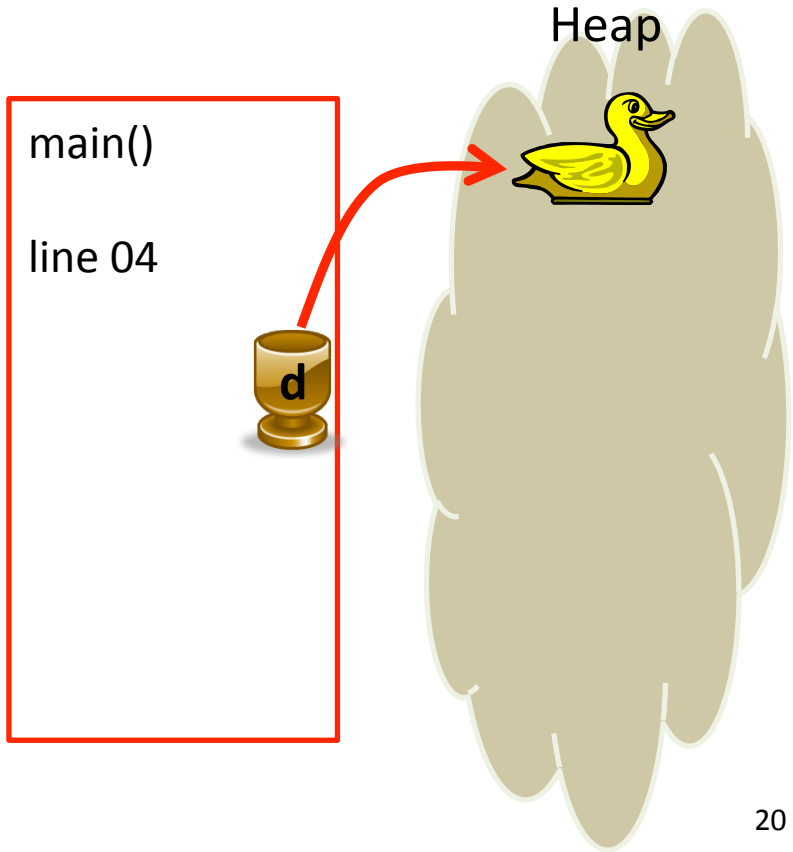
line 04
line 05
line 06
line 07
line 08



Killing objects

- Object-killer #2
  - Assign the reference to another object

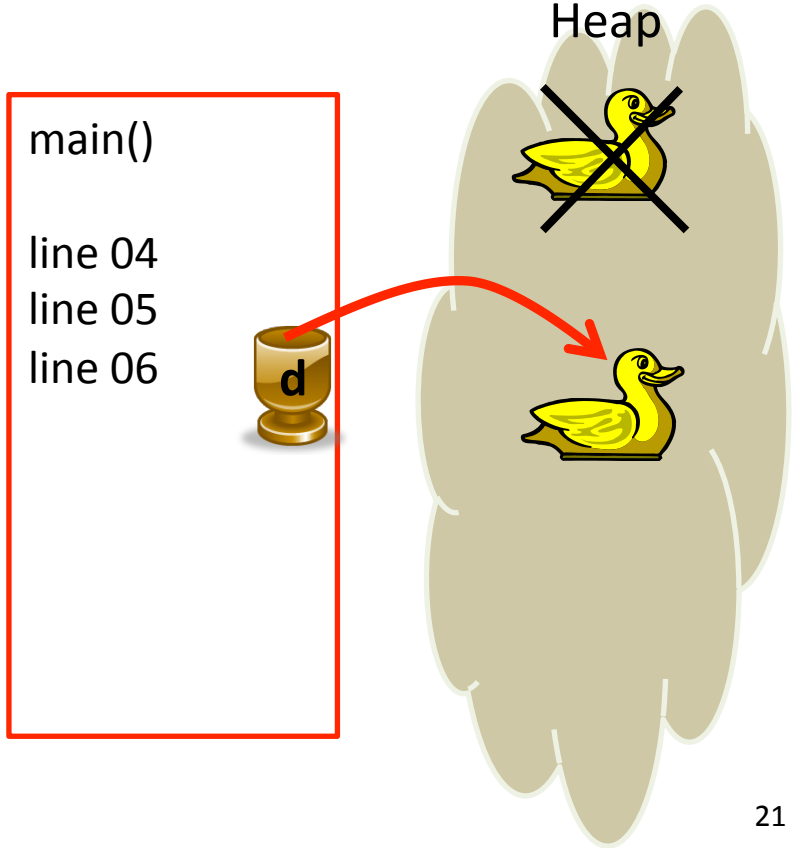
```
00 public class DuckKiller2
01 {
02    public static void main(String [] args)
03    {
04       Duck d = new Duck();
05       System.out.println("quack!");
06       d = new Duck();
07       System.out.println("quack!");
08       d = new Duck();
09       System.out.println("quack!");
10    }
11 }
```



Killing objects

- Object-killer #2
  - Assign the reference to another object

```
00 public class DuckKiller2
01 {
02    public static void main(String [] args)
03    {
04       Duck d = new Duck();
05       System.out.println("quack!");
06       d = new Duck();
07       System.out.println("quack!");
08       d = new Duck();
09       System.out.println("quack!");
10    }
11 }
```



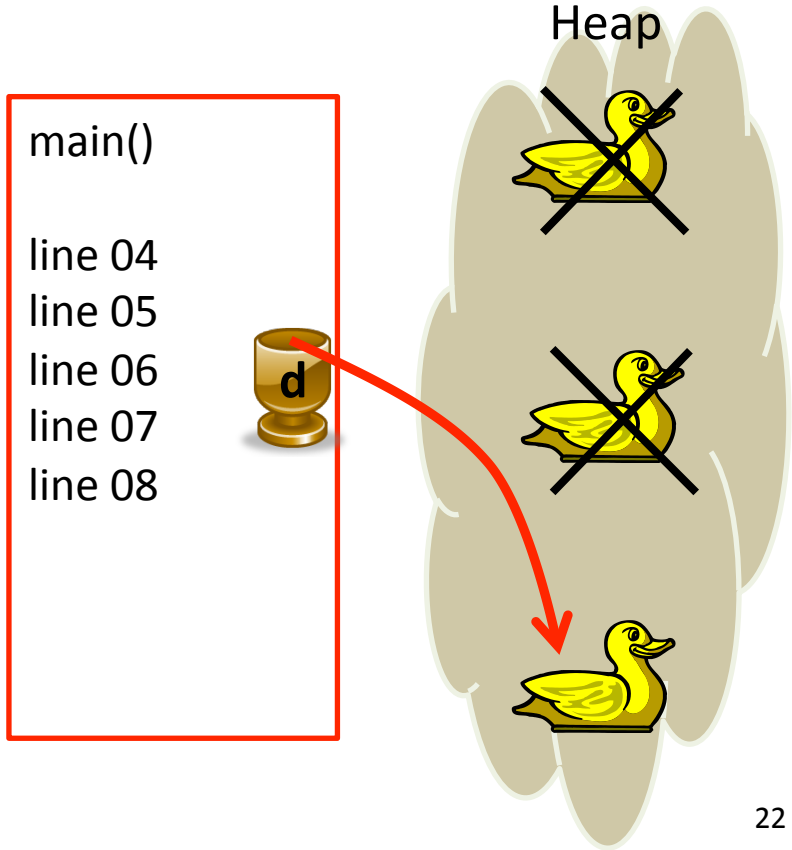
After executing line 06, no one is referring to the first Duck object anymore. It is now subject to garbage collection.

Killing objects

- Object-killer #2
  - Assign the reference to another object

```
00 public class DuckKiller2
01 {
02    public static void main(String [] args)
03    {
04       Duck d = new Duck();
05       System.out.println("quack!");
06       d = new Duck();
07       System.out.println("quack!");
08       d = new Duck();
09       System.out.println("quack!");
10    }
11 }
```

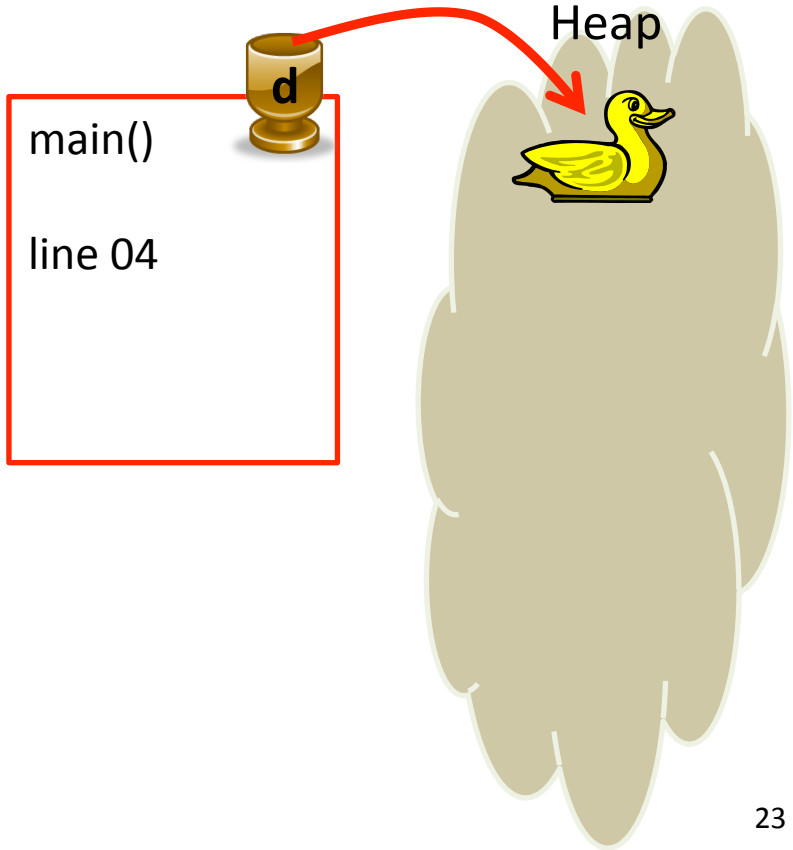
After executing line 08, both the first and second Ducks objects are can be garbage collected.



Killing objects

- Object-killer #3
  - Set the reference variable to null

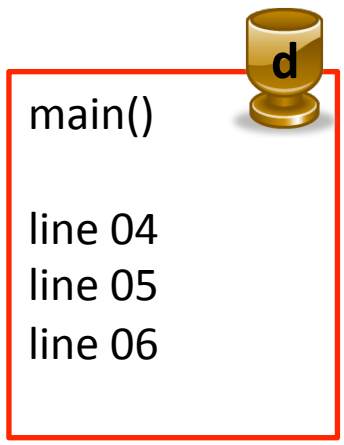
```
00 public class DuckKiller3
01 {
02    public static void main(String [] args)
03    {
04       Duck d = new Duck();
05       System.out.println("quack!");
06       d = null;
07       System.out.println("quack!");
08    }
09 }
```

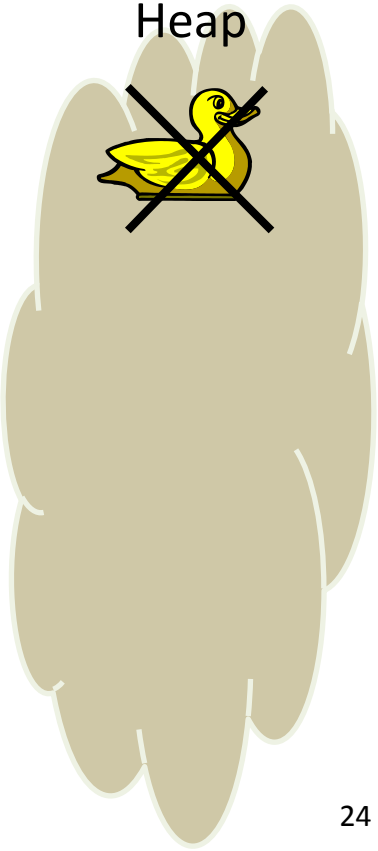


Killing objects

- Object-killer #3
  - Set the reference variable to null

```
00 public class DuckKiller3
01 {
02    public static void main(String [] args)
03    {
04       Duck d = new Duck();
05       System.out.println("quack!");
06       d = null;
07       System.out.println("quack!");
08    }
09 }
```





After executing line 06, no one is referring to the Duck object anymore. The Java garbage collector can now free up the Duck's memory.

Breaking the heap

- Memory is not infinite
  - You can't just keep new'ing objects forever without getting rid of any

```
import java.awt.*;

public class HeapDeath
{
   public static void main(String [] args)
   {
      ArrayList<Color> list = new ArrayList<Color>();

      while (true)
      {
         Color c = new Color(0, 0, 255);
         list.add(c);
      }
   }
}
```

While variable `c` scopes out every loop, the `Color` reference persists inside the `ArrayList` so memory is never freed

```
00 public class HeapPuzzler
01 {
02    public static void main(String [] args)
03    {
04       Duck d = new Duck();
05       Duck [] a = new Duck[4];
06       for (int i = 0; i < a.length; i++)
07          a[i] = new Duck();
08       System.out.println("quack!");
09       a[0] = null;
10       a[1] = d;
11       System.out.println("quack!");
12    }
13 }
```

After executing line	# Ducks on the heap	Variable(s) that point to a Duck object
04		
05		
08		
09		
10		

```
00 public class HeapPuzzler
01 {
02    public static void main(String [] args)
03    {
04       Duck d = new Duck();
05       Duck [] a = new Duck[4];
06       for (int i = 0; i < a.length; i++)
07          a[i] = new Duck();
08       System.out.println("quack!");
09       a[0] = null;
10       a[1] = d;
11       System.out.println("quack!");
12    }
13 }
```

After executing line	# Ducks on the heap	Variable(s) that point to a Duck object
04	1	d
05		
08		
09		
10		

```
00 public class HeapPuzzler
01 {
02    public static void main(String [] args)
03    {
04       Duck d = new Duck();
05       Duck [] a = new Duck[4];
06       for (int i = 0; i < a.length; i++)
07          a[i] = new Duck();
08       System.out.println("quack!");
09       a[0] = null;
10       a[1] = d;
11       System.out.println("quack!");
12    }
13 }
```

After executing line	# Ducks on the heap	Variable(s) that point to a Duck object
04	1	d
05	1	d
08		
09		
10		

```
00 public class HeapPuzzler
01 {
02    public static void main(String [] args)
03    {
04       Duck d = new Duck();
05       Duck [] a = new Duck[4];
06       for (int i = 0; i < a.length; i++)
07          a[i] = new Duck();
08       System.out.println("quack!");
09       a[0] = null;
10       a[1] = d;
11       System.out.println("quack!");
12    }
13 }
```

After executing line	# Ducks on the heap	Variable(s) that point to a Duck object
04	1	d
05	1	d
08	5	d, a[0], a[1], a[2], a[3]
09		
10		

```
00 public class HeapPuzzler
01 {
02    public static void main(String [] args)
03    {
04       Duck d = new Duck();
05       Duck [] a = new Duck[4];
06       for (int i = 0; i < a.length; i++)
07          a[i] = new Duck();
08       System.out.println("quack!");
09       a[0] = null;
10       a[1] = d;
11       System.out.println("quack!");
12    }
13 }
```

After executing line	# Ducks on the heap	Variable(s) that point to a Duck object
04	1	d
05	1	d
08	5	d, a[0], a[1], a[2], a[3]
09	4	d, a[1], a[2], a[3]
10		

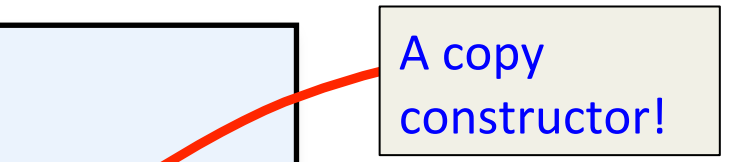
```
00 public class HeapPuzzler
01 {
02    public static void main(String [] args)
03    {
04       Duck d = new Duck();
05       Duck [] a = new Duck[4];
06       for (int i = 0; i < a.length; i++)
07          a[i] = new Duck();
08       System.out.println("quack!");
09       a[0] = null;
10       a[1] = d;
11       System.out.println("quack!");
12    }
13 }
```

After executing line	# Ducks on the heap	Variable(s) that point to a Duck object
04	1	d
05	1	d
08	5	d, a[0], a[1], a[2], a[3]
09	4	d, a[1], a[2], a[3]
10	3	(d, a[1]), a[2], a[3]

Copy constructors

- Copy constructor
  - A special constructor
  - Parameter is another object of the same type
  - Simply copies all the state of the passed in object

```
public class Duck
{
   private String name = "";
   private double weight = 0.0;

   public Duck(Duck otherDuck)
   {
      this.name   = otherDuck.name;
      this.weight = otherDuck.weight;
   }
}
```



Methods giving birth

- An instance method can create a new object and return it

```
public Block move(double x, double y) // Create a new block based on this block
                                      // but at a new (x,y) position

public Block rotate()                 // Create a new block based on this block
                                      // but rotated by 90 degrees
```

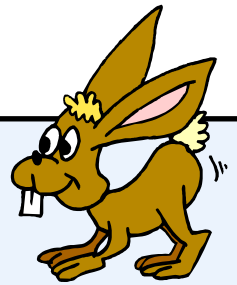
Hey rabbit, what do you know?

"My location and my size!"

Hey rabbit, what can you do?

"I can draw myself and I can tell if somebody clicks on me. Ohh and um, well you know, I can make them there baby rabbits…"

Rabbit class, part 1



```
public class Rabbit
{
   private double x = 0.0;
   private double y = 0.0;
   private static final double size = 0.06;

   public Rabbit(double x, double y)
   {
      this.x = x;
      this.y = y;
   }

   public void draw()
   {
      StdDraw.picture(x, y, "rabbit.png");
   }

   public boolean intersect(double x, double y)
   {
      double deltaX = (this.x - x);
      double deltaY = (this.y - y);
      return (Math.sqrt(deltaX * deltaX + deltaY * deltaY) < size);
   }
   ...
```

Rabbit class, part 2

```
    public Rabbit breed()
    {
        Rabbit baby = new Rabbit(this.x + (0.2 - Math.random() * 0.4),
                                 this.y + (0.2 - Math.random() * 0.4));
        return baby;
    }

    public static void main(String [] args)
    {
        ArrayList<Rabbit> rabbits = new ArrayList<Rabbit>();
        rabbits.add(new Rabbit(0.5, 0.5));
        while (true)
        {
            for (int i = rabbits.size() - 1; i >= 0; i--)
            {
                Rabbit r = rabbits.get(i);
                r.draw();

                if (r.intersect(StdDraw.mouseX(), StdDraw.mouseY()))
                    rabbits.add(r.breed());
            }
            StdDraw.show(100);
        }
    }
}
```

Summary

- How your data is stored
  - Stack
  - Heap
  - Garbage collector
- Creating new objects
  - Copy constructor
  - Other methods creating new object instances