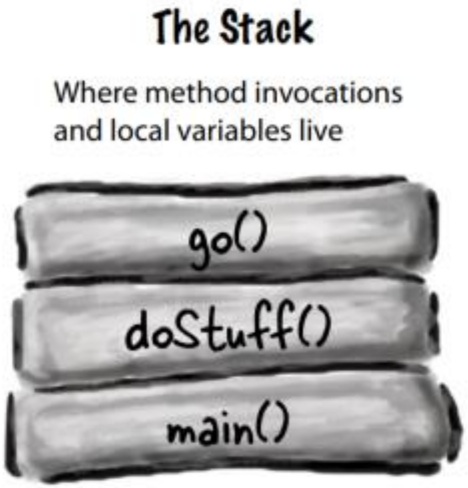
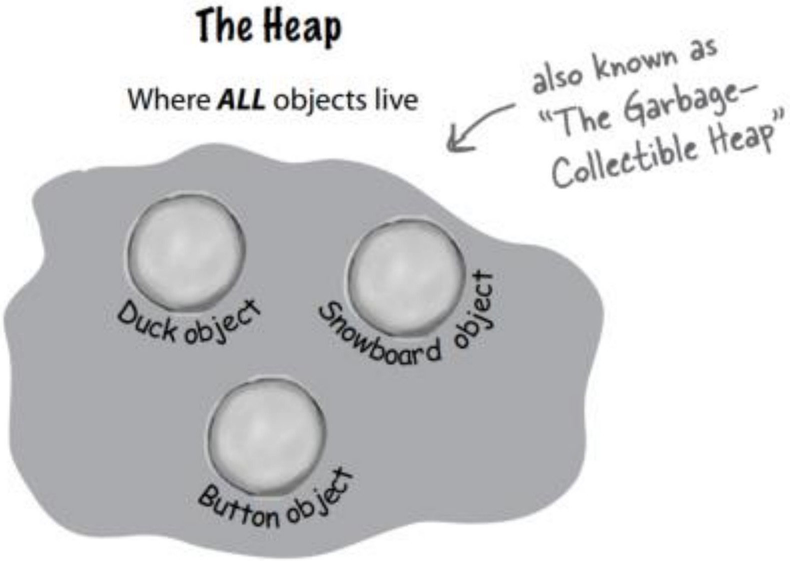
Chapter 9

Life and Death of an Object

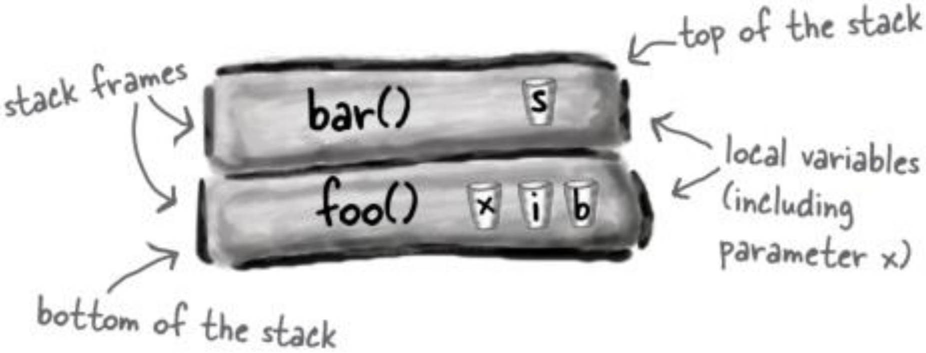
- You're in charge of an object's lifecycle
- You decide when and how to construct it. You decide when to destroy it
- You don't actually destroy the object yourself, you simply abandon it.
- Once it's abandoned, the Garbage Collector (GC) can vaporize it, reclaiming the memory that object was using

- In Java, we (programmers) care about two areas of memory—the one where objects live (the heap), and the one where method invocations and local variables live (the stack)

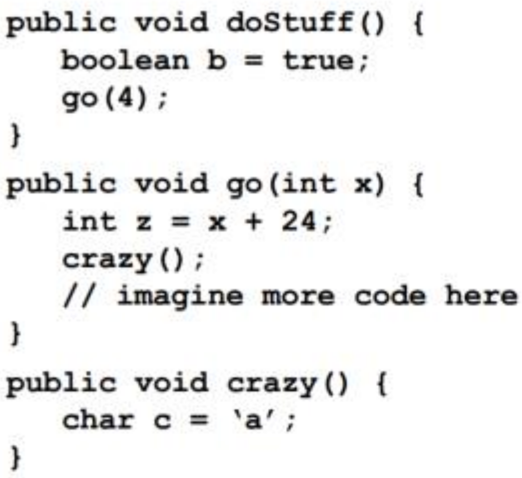




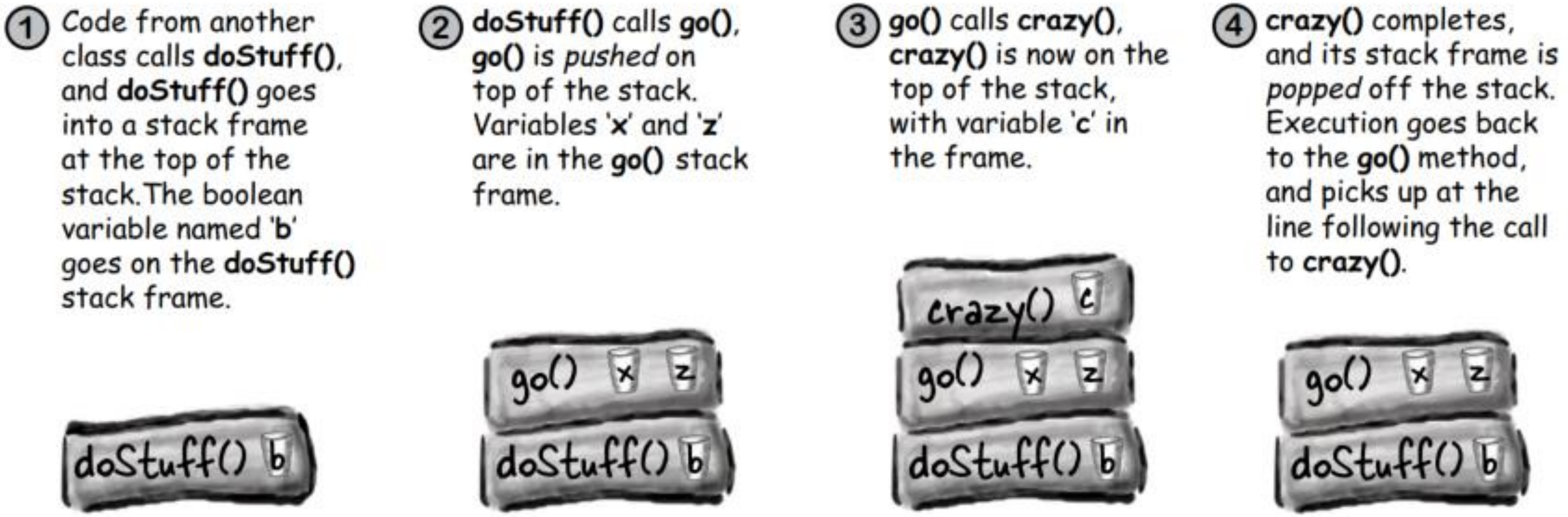
- When you call a method, the method lands on the top of a call stack
- The method at the top of the stack is always the currently-running method for that stack
- If method foo() calls method bar(), method bar() is stacked on top of method foo().



What about an object which is created *locally*?



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



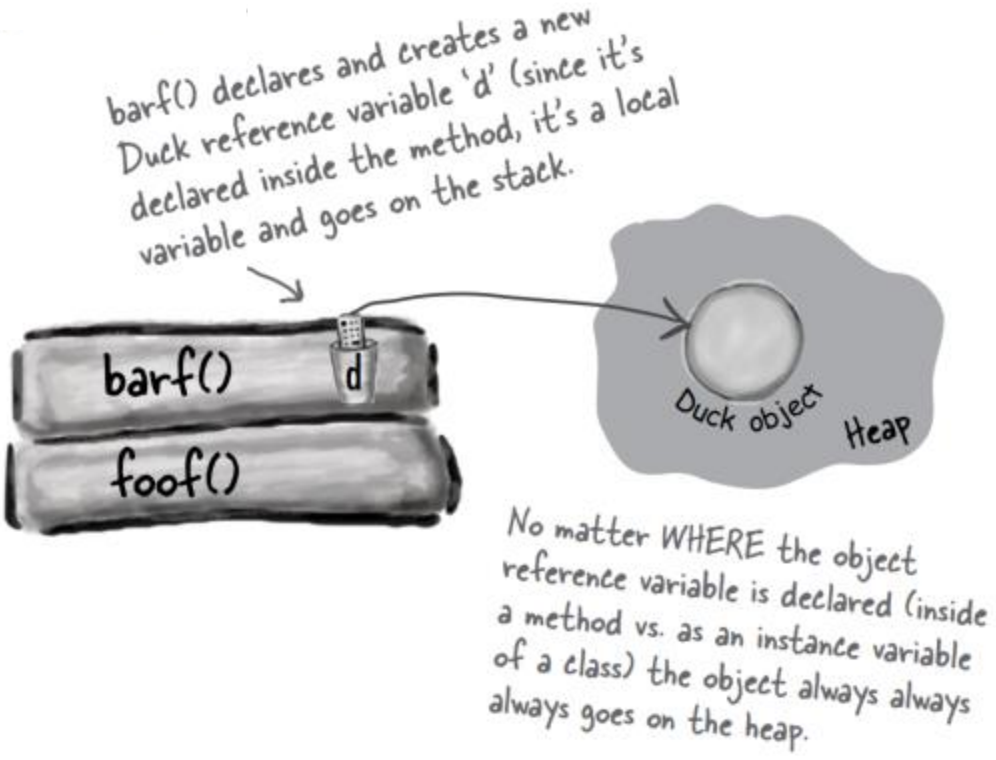
1. Code from another class calls **doStuff()**, and **doStuff()** goes into a stack frame at the top of the stack. The boolean variable named '**b**' goes on the **doStuff()** stack frame.

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()**.

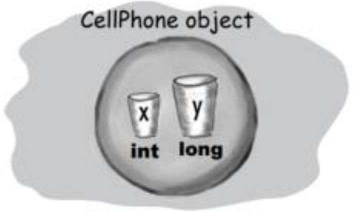
- If the local variable is a reference to an object, only the variable (the reference) goes on the stack. The object itself still goes in the heap



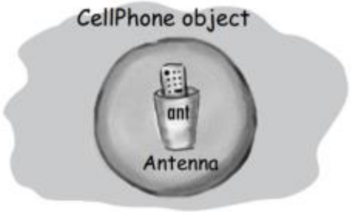
barf() declares and creates a new Duck reference variable 'd' (since it's declared inside the method, it's a local variable and goes on the stack.

No matter WHERE the object reference variable is declared (inside a method vs. as an instance variable of a class) the object always always always goes on the heap.

- Instance variables live on the Heap, inside the object they belong to
- If CellPhone has an instance variable declared as the non-primitive type Antenna, Java makes space within the CellPhone object only for the Antenna's reference variable

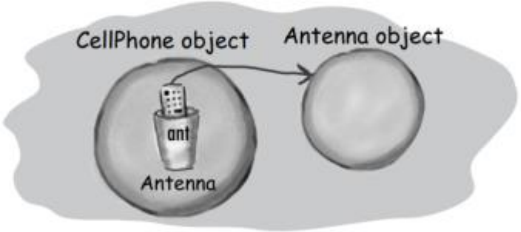


Object with two primitive instance variables. Space for the variables lives in the object.



Object with one non-primitive instance variable— a reference to an Antenna object, but no actual Antenna object. This is what you get if you declare the variable but don't initialize it with an actual Antenna object.

```
public class CellPhone {
  private Antenna ant;
}
```

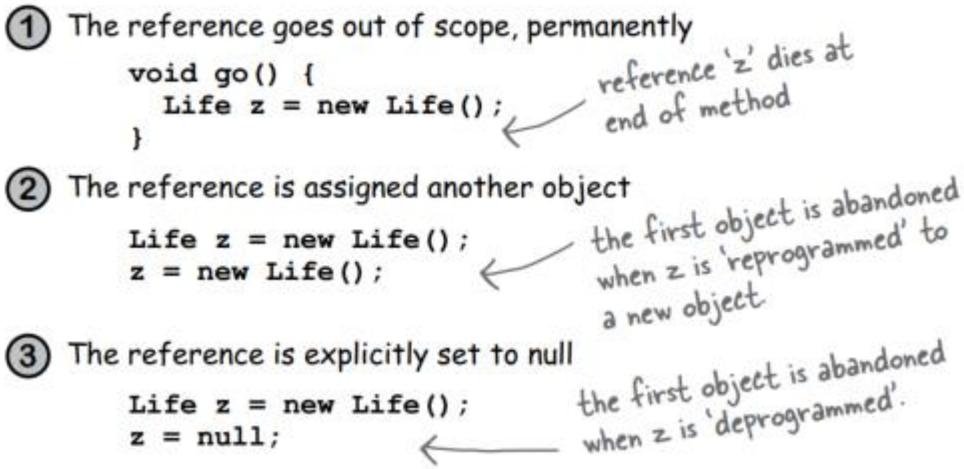


Object with one non-primitive instance variable, and the Antenna variable is assigned a new Antenna object.

```
public class CellPhone {
  private Antenna ant = new Antenna();
}
```

- What should you do when you no longer need to use an object?
- An object is alive as long as there are live references to it. If a reference variable goes out of scope but is still alive, the object it refers to is still alive on the Heap.
- If an object is unreachable(there’re not any variables refer to it), the Garbage Collector will figure that out. Sooner or later, that object is going down.
- Once an object is eligible for garbage collection (GC), you don’t have to worry about reclaiming the memory that object was using. If your program gets low on memory, GC will destroy some or all of the eligible objects, to keep you from running out of RAM
- Your job is to make sure that you abandon objects (i.e, make them eligible for GC) when you’re done with them, so that the garbage collector has something to reclaim

- Three ways to get rid of an object’s reference



1. The reference goes out of scope, permanently

```
void go() {
  Life z = new Life();
}
```

reference ‘z’ dies at end of method

2. The reference is assigned another object

```
Life z = new Life();
z = new Life();
```

the first object is abandoned when z is ‘reprogrammed’ to a new object

3. The reference is explicitly set to null

```
Life z = new Life();
z = null;
```

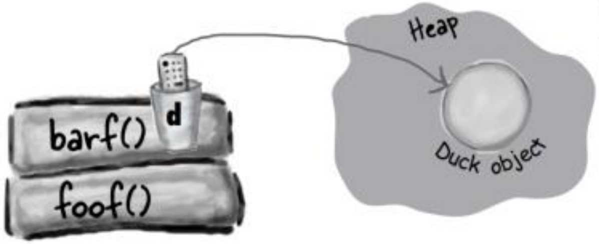
the first object is abandoned when z is ‘deprogrammed’.

- Reference goes out of scope

```
public class StackRef  {
   public void foof() {
      barf();
   }

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

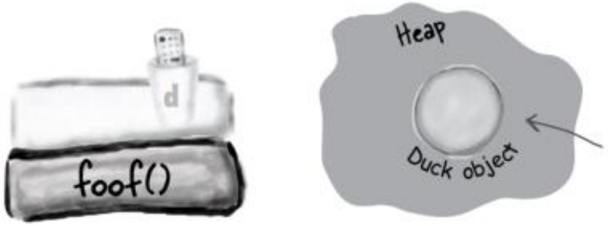
1 *foof()* is pushed onto the Stack, no variables are declared.

2 *barf()* is pushed onto the Stack, where it declares a reference variable, and creates a new object assigned to that reference. The object is created on the Heap, and the reference is alive and in scope.



The new Duck goes on the Heap, and as long as barf() is running, the 'd' reference is alive and in scope, so the Duck is considered alive.

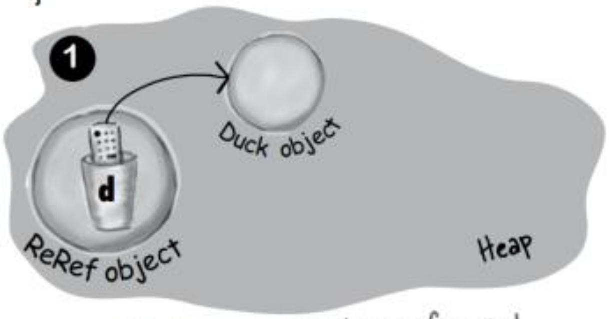
3 *barf()* completes and pops off the Stack. Its frame disintegrates, so 'd' is now dead and gone. Execution returns to *foof()*, but *foof()* can't use 'd' .



Uh-oh. The 'd' variable went away when the barf() Stack frame was blown off the stack, so the Duck is abandoned. Garbage-collector bait.

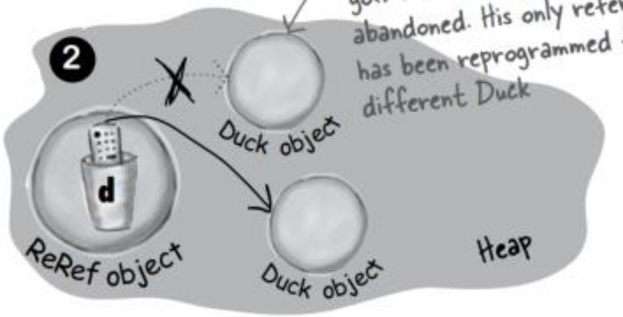
- Assign the reference to another object

```
public class ReRef {

    Duck d = new Duck();

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



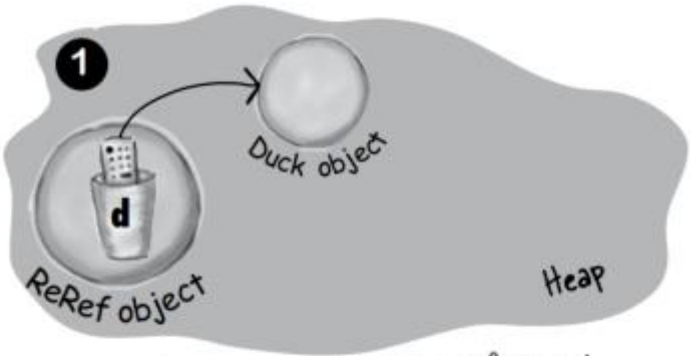
The new Duck goes on the Heap, referenced by 'd'. Since 'd' is an instance variable, the Duck will live as long as the ReRef object that instantiated it is alive. Unless...

When someone calls the go() method, this Duck is abandoned. His only reference has been reprogrammed for a different Duck

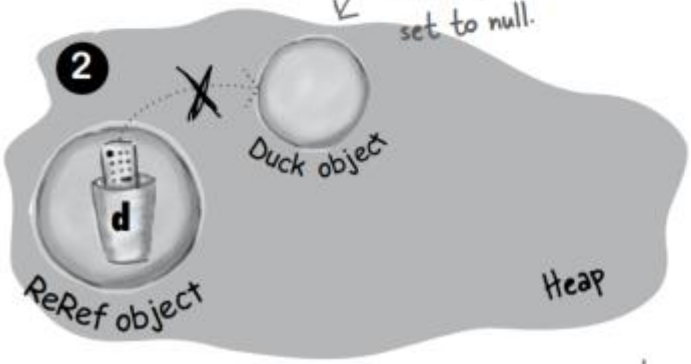


'd' is assigned a new Duck object, leaving the original (first) Duck object abandoned. That first Duck is now as good as dead.

- Explicitly set the reference to null



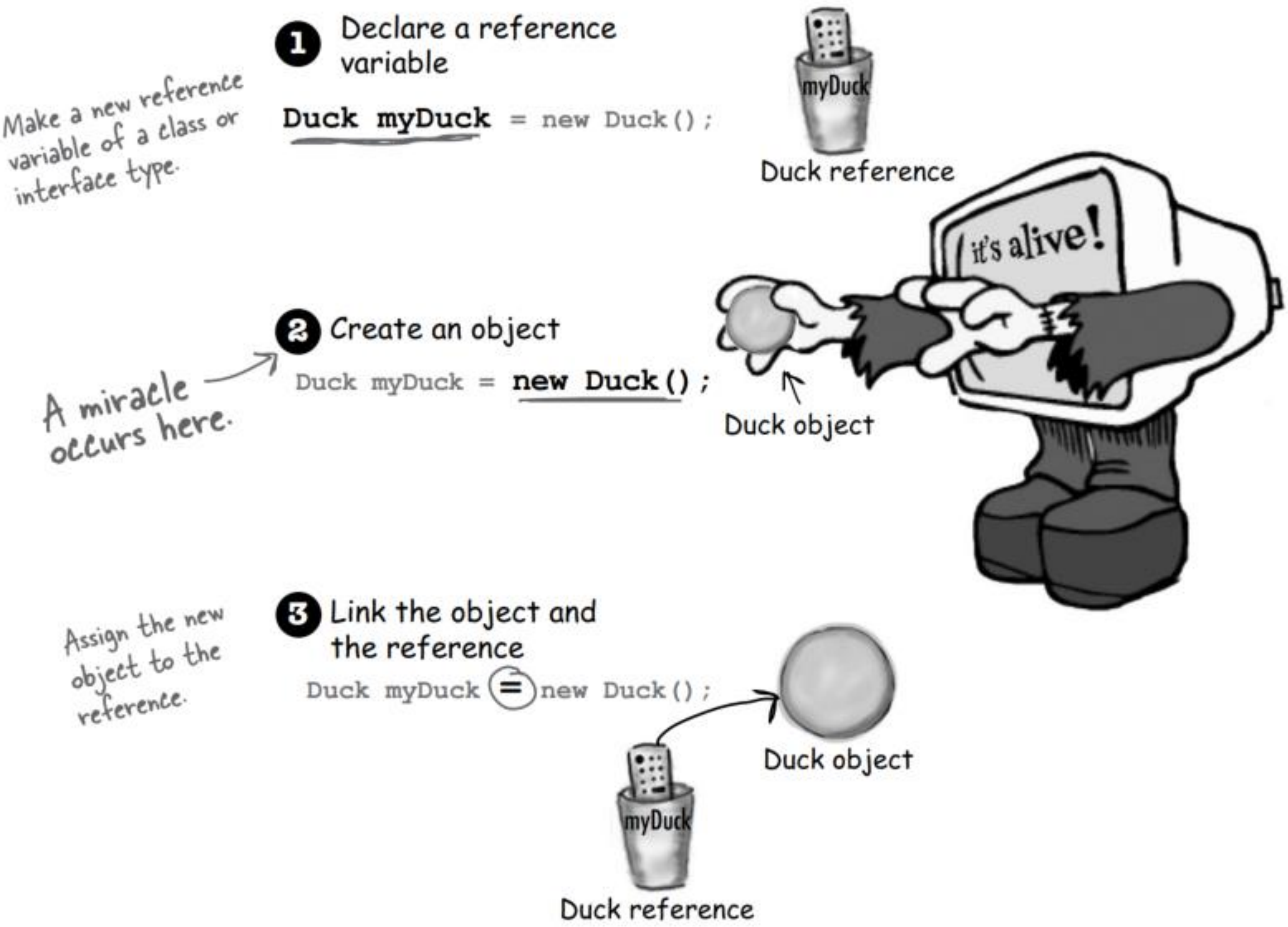
The new Duck goes on the Heap, referenced by 'd'. Since 'd' is an instance variable, the Duck will live as long as the ReRef object that instantiated it is alive. Unless...



This Duck is abandoned. His only reference has been set to null.

'd' is set to null, which is just like having a remote control that isn't programmed to anything. You're not even allowed to use the dot operator on 'd' until it's reprogrammed (assigned an object).

- The 3 steps of an object: declaration, creation and assignment



1. Declare a reference variable

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

2. Create an object

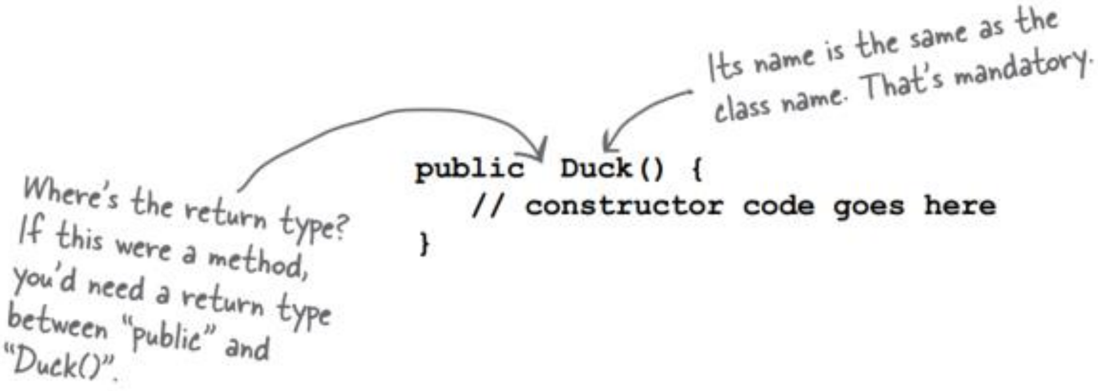
```
Duck myDuck = new Duck();
```

3. Link the object and the reference

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

Make a new reference variable of a class or interface type.
myDuck
Duck reference
it's alive!
A miracle occurs here.
Duck object
Assign the new object to the reference.
Duck object
myDuck
Duck reference

- Are we calling a method named Duck()? No. We’re calling the Duck ***constructor***.
- The only way to invoke a constructor is with the keyword new followed by the class name
- You can write a constructor for your class, but if you don’t, the compiler writes one for you!

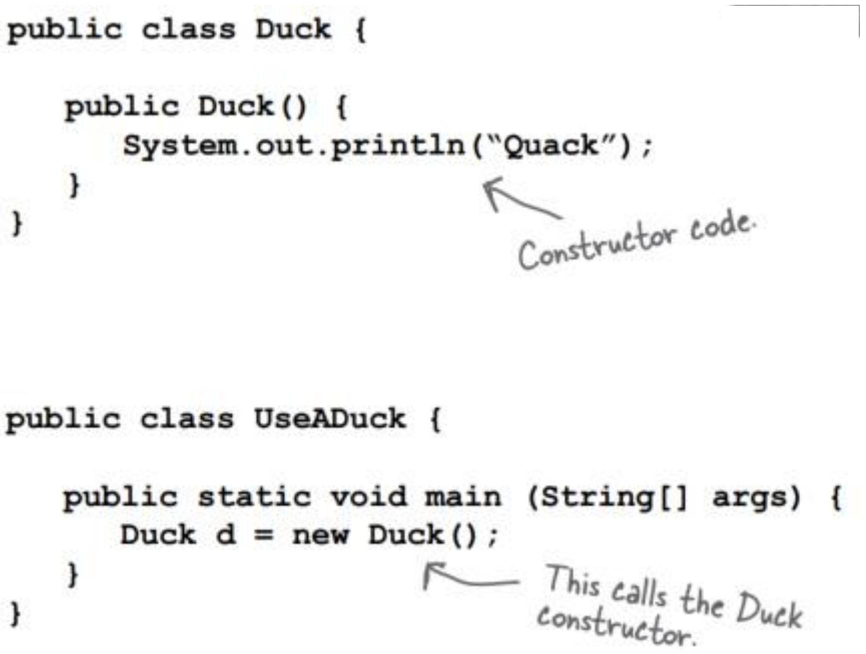


```
public Duck() {
    // constructor code goes here
}
```

Its name is the same as the class name. That’s mandatory.

Where’s the return type? If this were a method, you’d need a return type between “public” and “Duck()”.

- The key feature of a constructor is that it runs ***before*** the object can be assigned to a reference.
- You get a chance to step in and do things to get the object ready for use
- Can you imagine conditions where that would be useful?



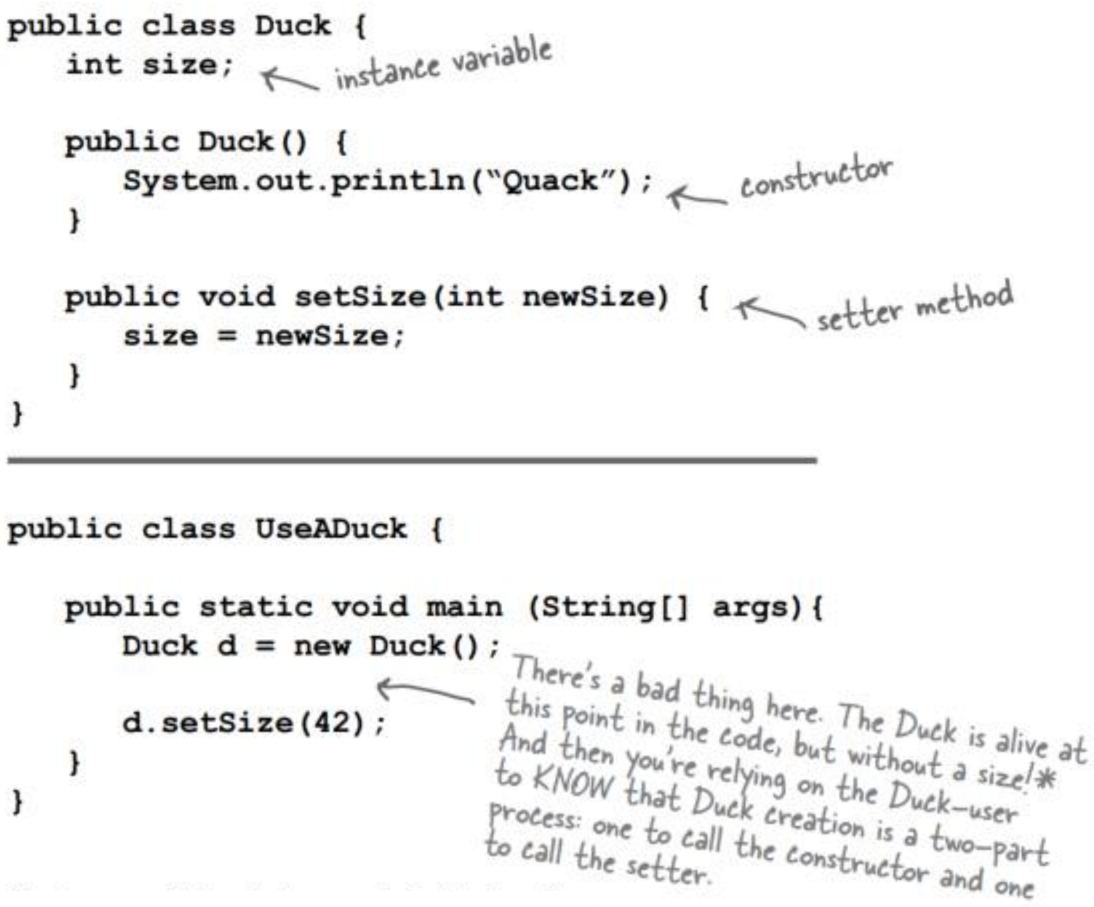
```
public class Duck {

    public Duck() {
        System.out.println(“Quack”);
    }
}
```

Constructor code.

```
public class UseADuck {

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

This calls the Duck constructor.

- Most people use constructors to ***initialize*** the state of an object.
- In other words, to make and assign values to the object's instance variables
- Quiz: recall the setter method we mentioned before. We can set values to instance variables after we instantiate it. Why we need a constructor?
- That leaves the Duck temporarily without a size*, and forces the Duck user to write two statements—one to create the Duck, and one to call the setSize() method



```
public class Duck {
   int size;

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

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

instance variable

constructor

setter method

```
public class UseADuck {

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

      d.setSize(42);
   }
}
```

There's a bad thing here. The Duck is alive at this point in the code, but without a size!* And then you're relying on the Duck-user to KNOW that Duck creation is a two-part process: one to call the constructor and one to call the setter.

- The best place to put initialization code is in the constructor. And all you need to do is make a constructor with ***arguments***.

```
public class Duck {
   int size;

   public Duck(int duckSize) {
      System.out.println(“Quack”);

      size = duckSize;

      System.out.println(“size is ” + size);
   }
}
```

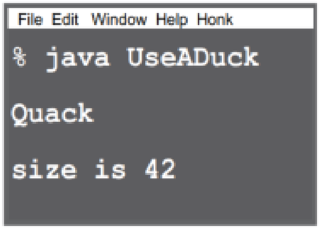
Add an int parameter to the Duck constructor.

Use the argument value to set the size instance variable.

```
public class UseADuck {

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

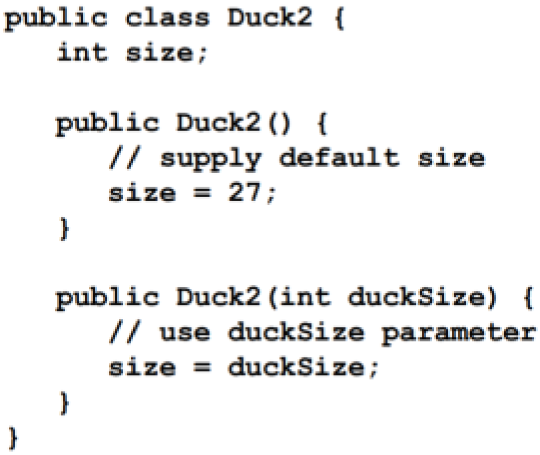
This time there's only one statement. We make the new Duck and set its size in one statement.

Pass a value to the constructor.



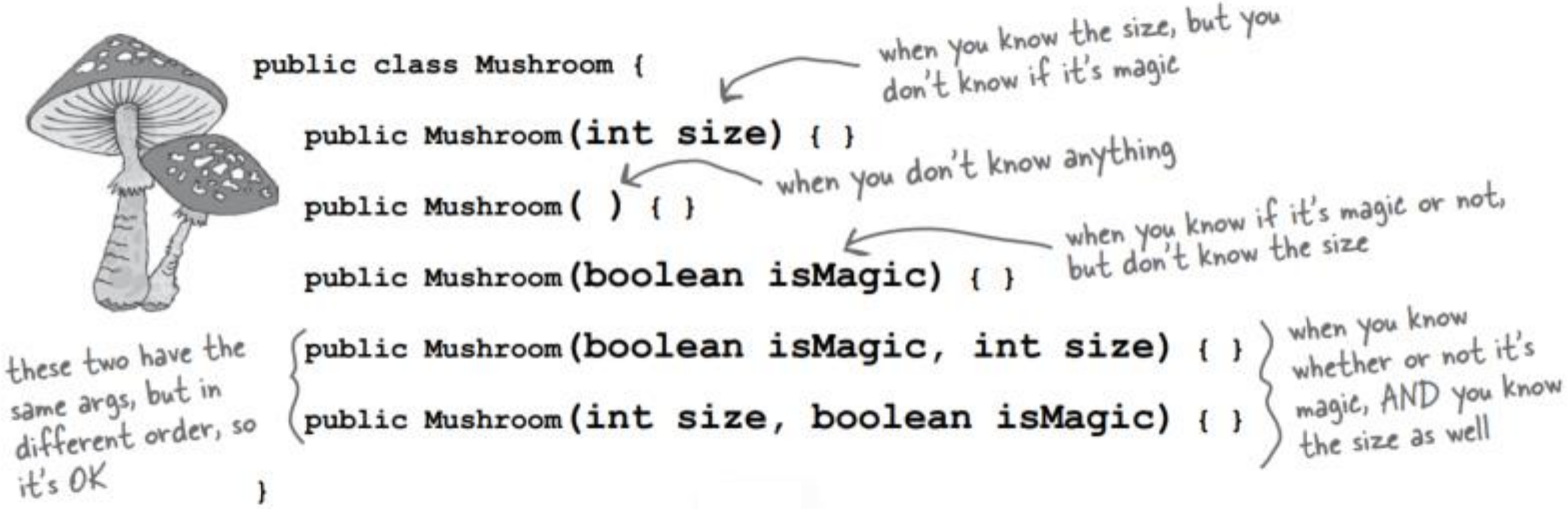
```
% java UseADuck
Quack
size is 42
```

- Can you have multiple constructors in a class?
- Imagine that you want Duck users to have TWO options for making a Duck—one where they supply the Duck size (as the constructor argument) and one where they don't specify a size and thus get your default Duck size
- So we need two constructors. One that takes an int and one that doesn't. If you have more than one constructor in a class, it means you have ***overloaded*** constructors.



```
public class Duck2 {
   int size;

   public Duck2() {
      // supply default size
      size = 27;
   }

   public Duck2(int duckSize) {
      // use duckSize parameter
      size = duckSize;
   }
}
```

- Quiz: If I made an constructor that takes arguments, does the compiler make a no-arg constructor for you?
- No. The compiler gets involved with constructor making only if you don't say anything at all about constructors.
- Constructor overloading - each constructor MUST have a different argument list



```
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) { }
}
```

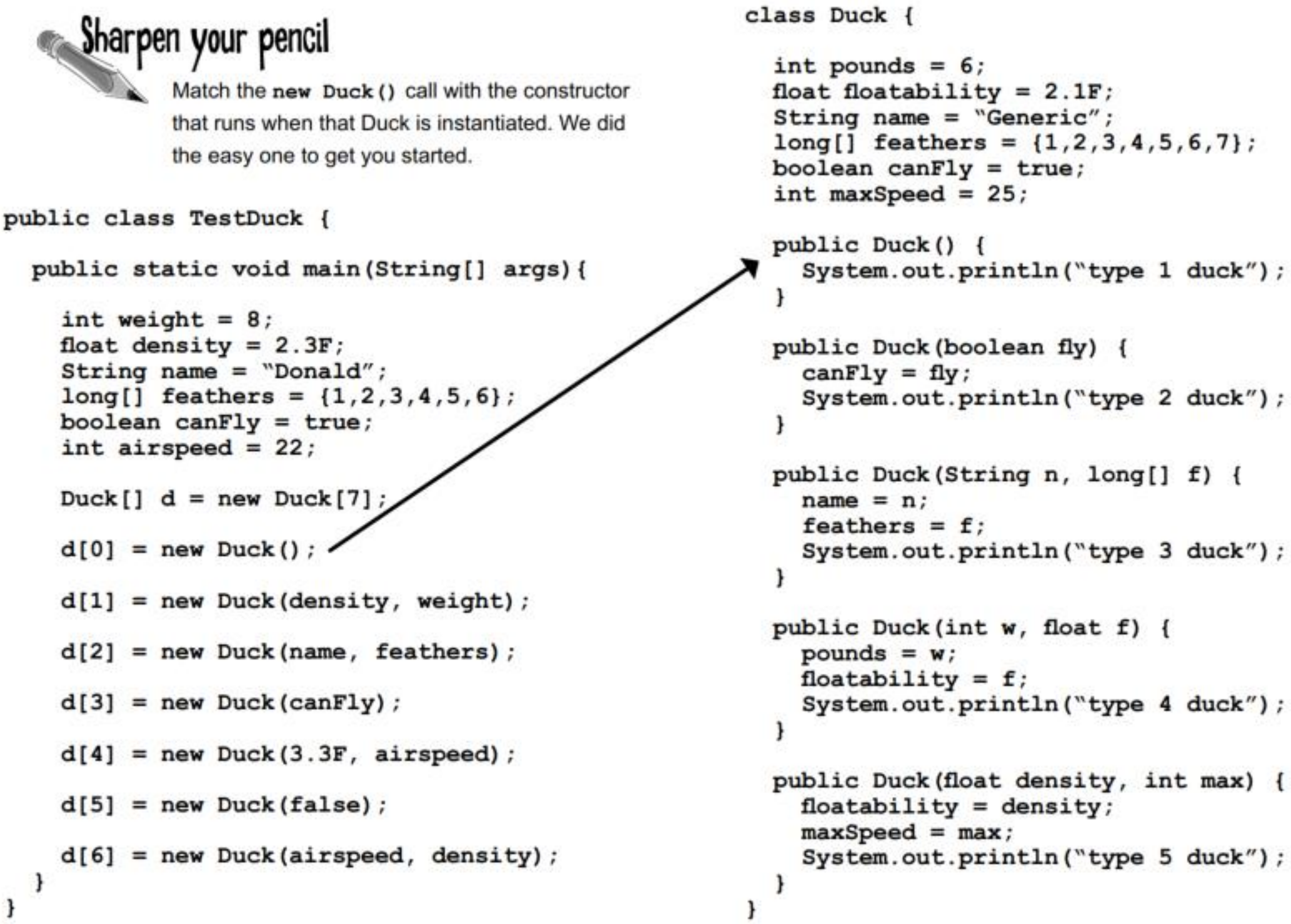
when you know the size, but you don't know if it's magic

when you don't know anything

when you know if it's magic or not, but don't know the size

when you know whether or not it's magic, AND you know the size as well

these two have the same args, but in different order, so it's OK



Sharpen your pencil

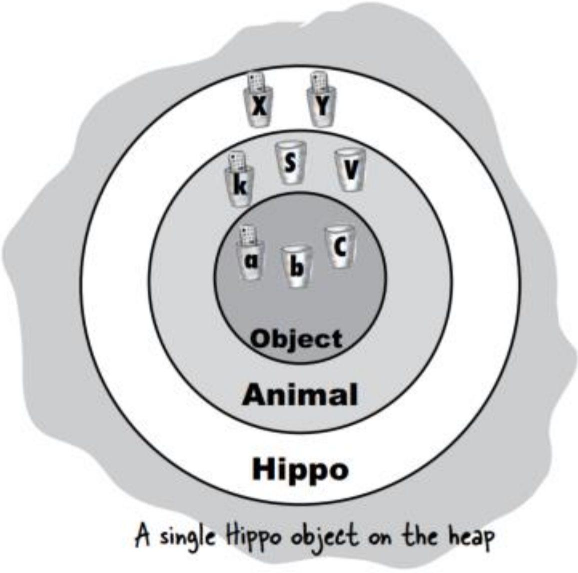
Match the **new Duck()** call with the constructor that runs when that Duck is instantiated. We did the easy one to get you started.

```
public class TestDuck {

  public static void main(String[] args){

    int weight = 8;
    float density = 2.3F;
    String name = “Donald”;
    long[] feathers = {1,2,3,4,5,6};
    boolean canFly = true;
    int airspeed = 22;

    Duck[] d = new Duck[7];

    d[0] = new Duck();

    d[1] = new Duck(density, weight);

    d[2] = new Duck(name, feathers);

    d[3] = new Duck(canFly);

    d[4] = new Duck(3.3F, airspeed);

    d[5] = new Duck(false);

    d[6] = new Duck(airspeed, density);
  }
}
```

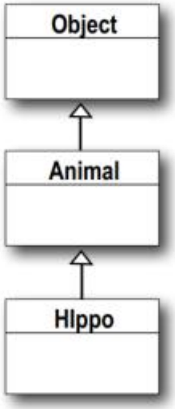
```
class Duck {

  int pounds = 6;
  float floatability = 2.1F;
  String name = “Generic”;
  long[] feathers = {1,2,3,4,5,6,7};
  boolean canFly = true;
  int maxSpeed = 25;

  public Duck() {
    System.out.println(“type 1 duck”);
  }

  public Duck(boolean fly) {
    canFly = fly;
    System.out.println(“type 2 duck”);
  }

  public Duck(String n, long[] f) {
    name = n;
    feathers = f;
    System.out.println(“type 3 duck”);
  }

  public Duck(int w, float f) {
    pounds = w;
    floatability = f;
    System.out.println(“type 4 duck”);
  }

  public Duck(float density, int max) {
    floatability = density;
    maxSpeed = max;
    System.out.println(“type 5 duck”);
  }
}
```

- Quick review about constructors
- A constructor is the code that runs when somebody says new on a class type
- A constructor must have the same name as the class, and no return type
- If you don't put a constructor in your class, the compiler puts in a default constructor. The default constructor is always a no-arg constructor
- (*Recommended*) Always provide a no-arg constructor if you can
- You can have more than one constructor in your class, as long as the argument lists are different. Having more than one constructor in a class means you have overloaded constructors
- Quiz: What about superclasses? When you make a Dog, should the Canine (or Animal) constructor run too?

- All the constructors in an object's inheritance tree MUST run when you make a new object.



A single Hippo object on the heap



- A new Hippo object also IS-A Animal and IS-A Object. If you want to make a Hippo, you must also make the Animal and Object parts of the Hippo. This is called Constructor Chaining.

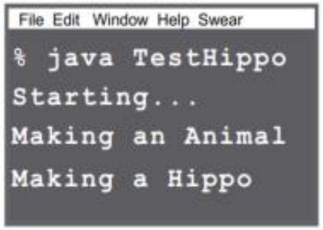
```
public class Animal {
   public Animal() {
      System.out.println(“Making an Animal”);
   }
}
```

```
public class Hippo extends Animal {
   public Hippo() {
      System.out.println(“Making a Hippo”);
   }
}
```

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

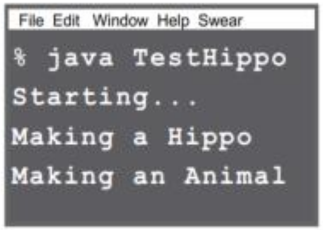
Which one is the correct output?

A



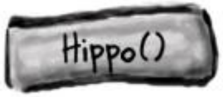
```
% java TestHippo
Starting...
Making an Animal
Making a Hippo
```

B

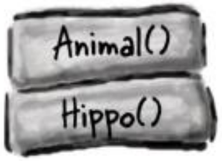


```
% java TestHippo
Starting...
Making a Hippo
Making an Animal
```

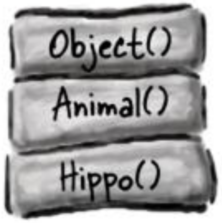
① Code from another class says **new Hippo()** and the **Hippo()** constructor goes into a stack frame at the top of the stack.



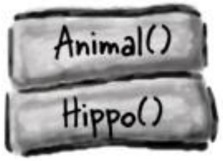
② **Hippo()** invokes the superclass constructor which pushes the **Animal()** constructor onto the top of the stack.



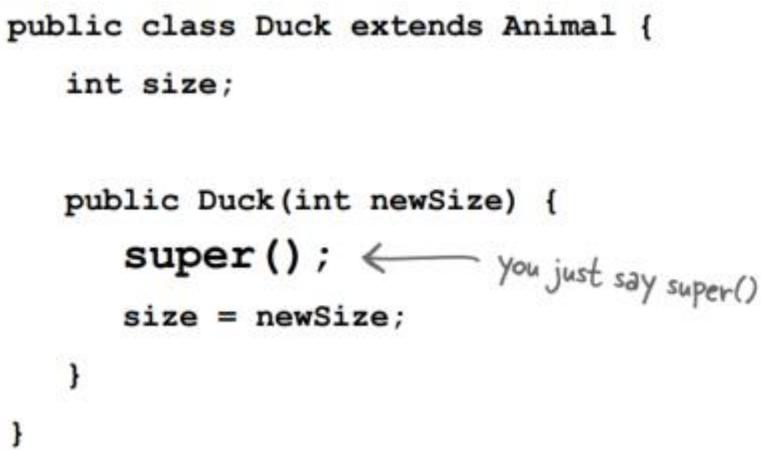
③ **Animal()** invokes the superclass constructor which pushes the **Object()** constructor onto the top of the stack, since Object is the superclass of Animal.



④ **Object()** completes, and its stack frame is *popped* off the stack. Execution goes back to the **Animal()** constructor, and picks up at the line following Animal's call to its superclass constructor



- How do you invoke a superclass constructor?
- The only way to call a super constructor is by calling super().

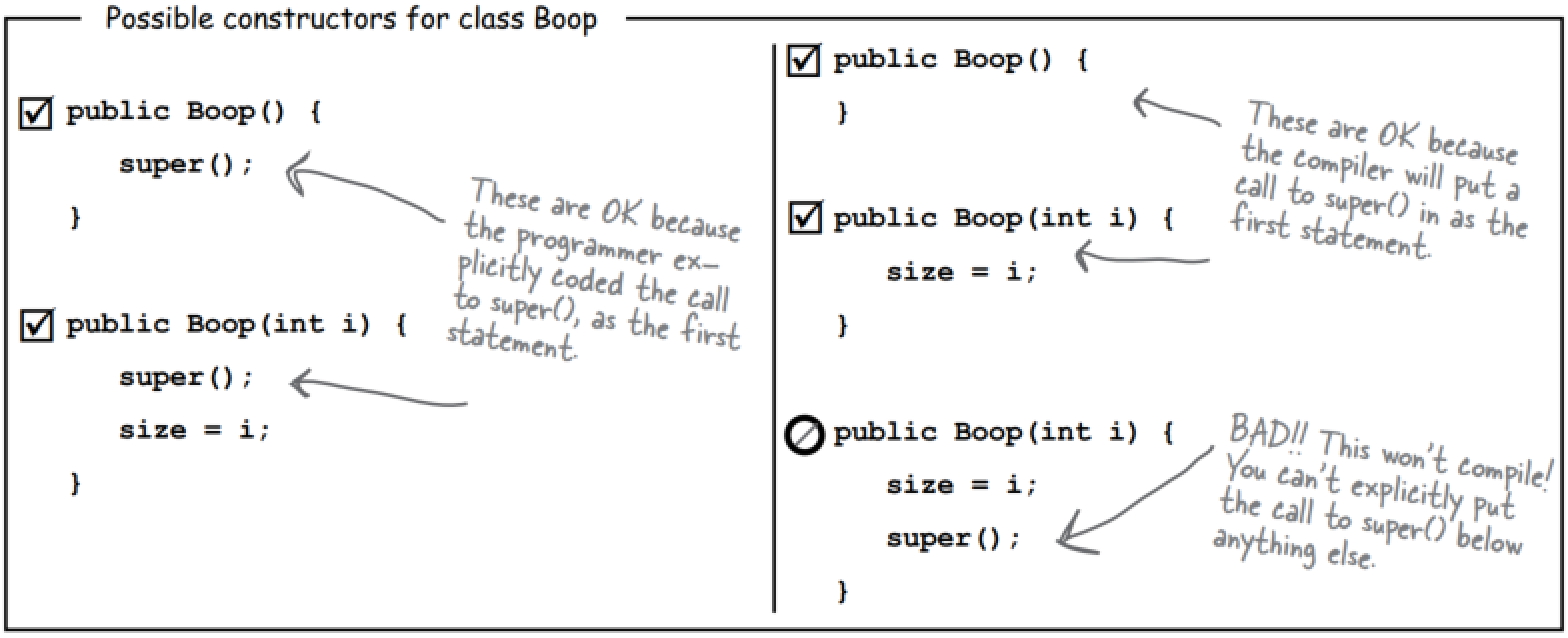


```
public class Duck extends Animal {
   int size;

   public Duck(int newSize) {
      super();
      size = newSize;
   }
}
```

you just say super()

- Quiz: What if you don’t provide a constructor? Or you do provide a constructor but you do not put in the call to super()
- The compiler puts in a call to super() if you don’t.
- The compiler will put a call to super() in each of your overloaded constructors. (The compiler-inserted call to super() is always a no-arg call)

- In general, the call to super() MUST be the first statement in each constructor



Possible constructors for class Boop

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

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

These are OK because the programmer explicitly coded the call to super(), as the first statement.

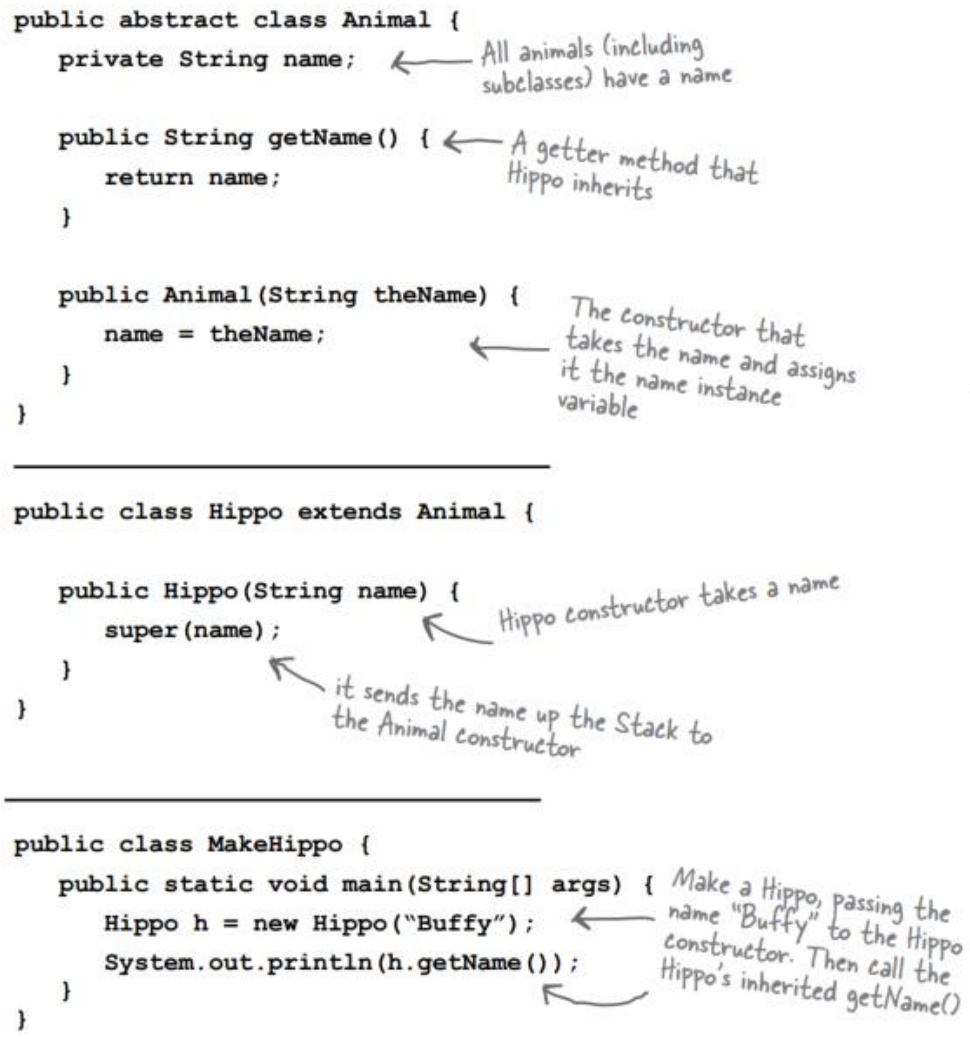
```
☑ public Boop() {
  }

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

These are OK because the compiler will put a call to super() in as the first statement.

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

BAD!! This won't compile! You can't explicitly put the call to super() below anything else.

- What if the superclass constructor has arguments? Can you pass something in to the super() call?



```
public abstract class Animal {
    private String name;

    public String getName() {
        return name;
    }

    public Animal(String theName) {
        name = theName;
    }
}
```

All animals (including subclasses) have a name

A getter method that Hippo inherits

The constructor that takes the name and assigns it the name instance variable

```
public class Hippo extends Animal {

    public Hippo(String name) {
        super(name);
    }
}
```

Hippo constructor takes a name

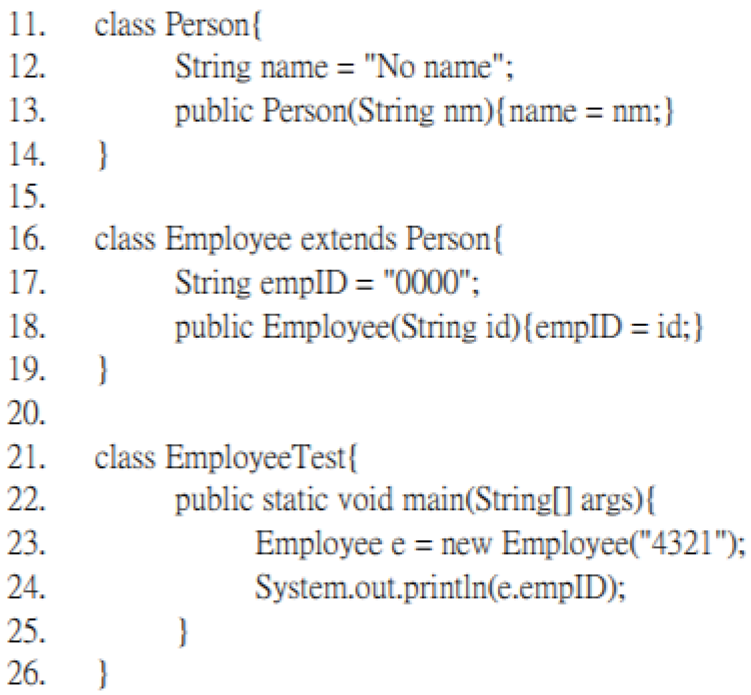
it sends the name up the Stack to the Animal constructor

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

Make a Hippo, passing the name "Buffy" to the Hippo constructor. Then call the Hippo's inherited getName()

- Quiz: If the superclass is ***abstract***, should it even have a constructor?
- Imaging the situation that you have your abstract class extends another class. What will happen if you cannot have constructors in an abstract class?
- Yes, it can have a constructor and it behaves just like a normal class's constructor

- Quiz: what's the result? Will it compile?



```
class Person{
        String name = "No name";
        public Person(String nm){name = nm;}
}

class Employee extends Person{
        String empID = "0000";
        public Employee(String id){empID = id;}
}

class EmployeeTest{
        public static void main(String[] args){
                Employee e = new Employee("4321");
                System.out.println(e.empID);
        }
}
```

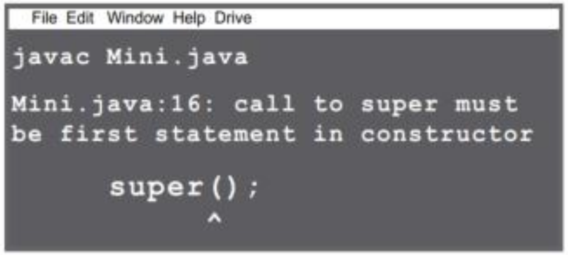
- Invoking one overloaded constructor from another
- What if you have overloaded constructors that, with the exception of handling different argument types, all do the same thing?
- Use the keyword this() (just imagine that the keyword this is a reference to the current object).
- Note: the keyword this() also has to be the first line. That means you can only chain to one constructor.

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

    Color color;

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

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

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

    }
}
```

The no-arg constructor supplies a default Color and calls the overloaded Real Constructor (the one that calls super()).

This is The Real Constructor that does The Real Work of initializing the object (including the call to super())

Won't work!! Can't have super() and this() in the same constructor, because they each must be the first statement!



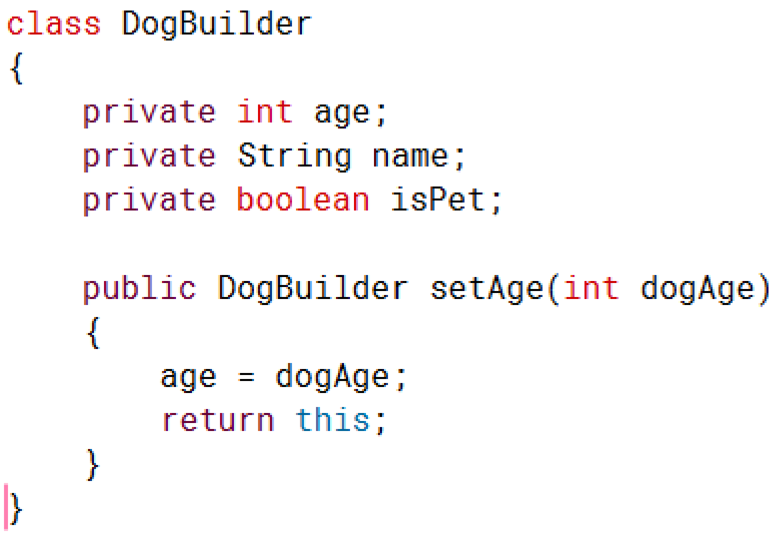
```
javac Mini.java

Mini.java:16: call to super must
be first statement in constructor

      super();
        ^
```

- Extension
- If you have a lot of parameters needed to be initialized, you may have a solution like this:

```
public MyClass(){};
public MyClass(a){};
public MyClass(a,b){};
public MyClass(a,b,c){};
public MyClass(a,b,c,d){};
```

- This is legal but not elegant. It is very difficult for developer to remember the required order of the parameters
- There is a design pattern called The Builder pattern.

- The Builder pattern is an object creation design pattern of which intent is to separate the construction of a complex object from its representation. By doing so, the same construction process can lead to different representations.
- The trick is that we can return the object's reference using the keyword this



```
class DogBuilder
{
    private int age;
    private String name;
    private boolean isPet;

    public DogBuilder setAge(int dogAge)
    {
        age = dogAge;
        return this;
    }
}
```

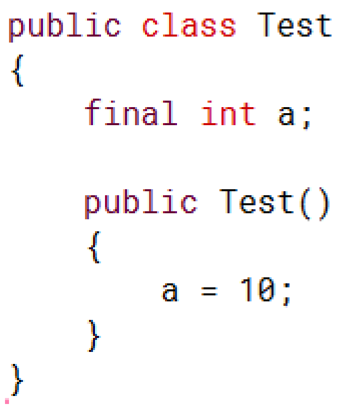
```
class DogBuilder
{
    private int age;
    private String name;
    private boolean isPet;

    public DogBuilder setAge(int dogAge)
    {
        age = dogAge;
        return this;
    }

    public DogBuilder setName(String dogName)
    {
        name = dogName;
        return this;
    }

    public DogBuilder isPet(boolean isAPet)
    {
        isPet = isAPet;
        return this;
    }

    public Dog create()
    {
        return new Dog(age,name,isPet);
    }
}

public class dpTest
{
    public static void main(String[] args)
    {
        Dog d = new DogBuilder().setName("Lucky").setAge(8).isPet(true).create();
    }
}
```

- Quizzes in Chapter 10
- Quiz 1: can you set value to a final variable via a constructor?
- Quiz 2: can you extend a class who only has private constructors?

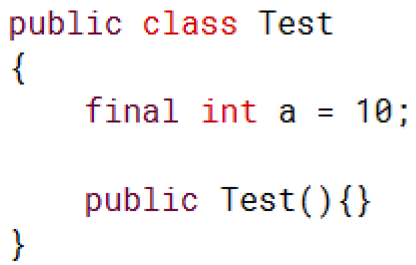
- Quiz 1: can you set value to a final variable via a constructor?
- Ans: yes you can. The keyword final means that you can only set this variable ***once***. There are two ways to do that:

1. In the constructor
2. When you declare it

- Note: you CANNOT do both



```
public class Test
{
    final int a;

    public Test()
    {
        a = 10;
    }
}
```



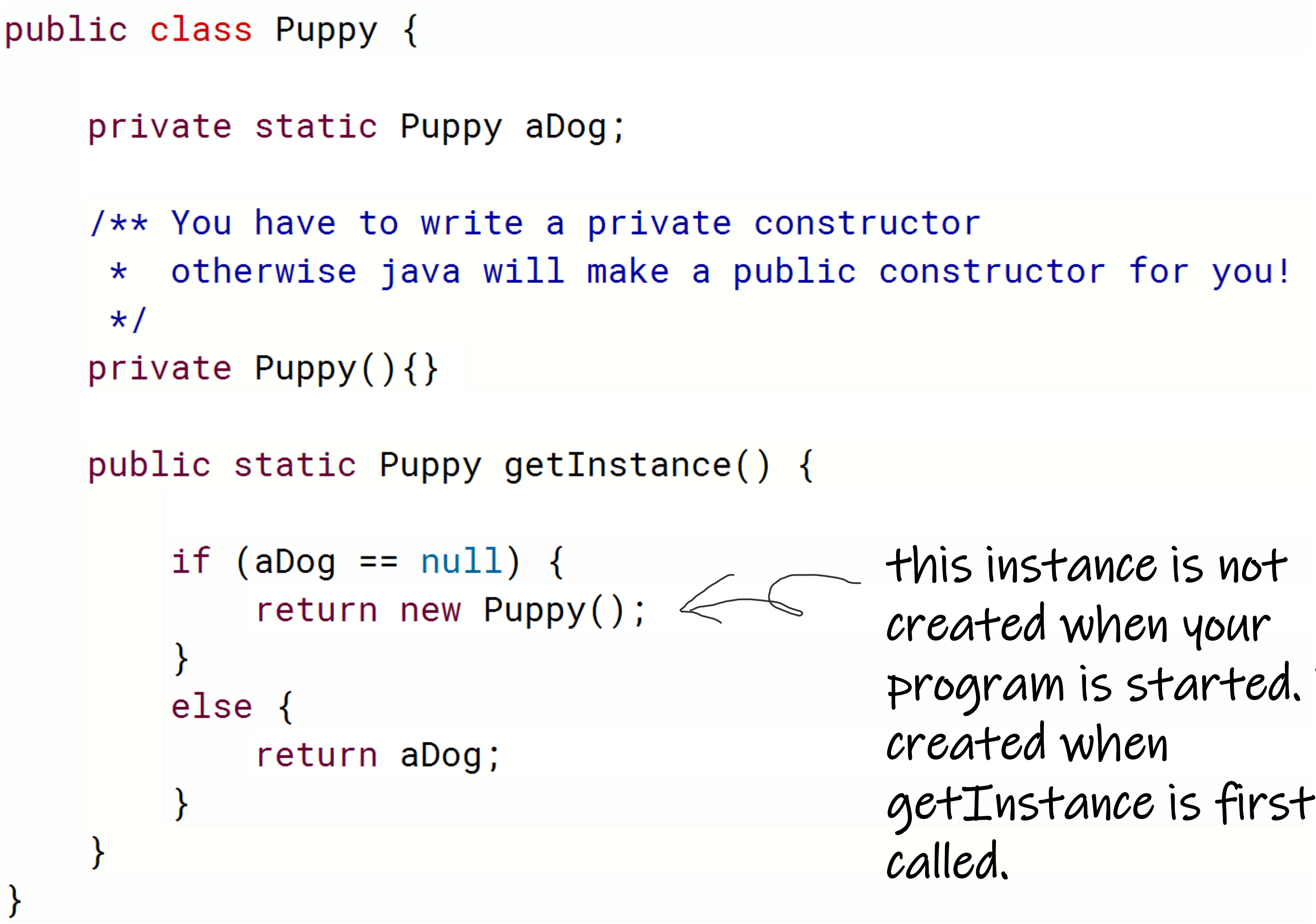
```
public class Test
{
    final int a = 10;

    public Test(){}
}
```

Q: If you do not set value to a final variable in a constructor, can you set it afterwards?

- Quiz 2: can you extend a class who only has private constructors?
- Ans: you can't extend the parent class if it has a private default constructor (because you can’t see them!). So usually we also mark it as a final class.
- In fact, a better question is "should we extend a class having only a private constructor?“

- Quiz: constructor is used for creating objects. What if we don’t need objects?
- Prevent callers from creating instances -> mark the constructor as ***private***
- A class which has private constructors is usually a ***utility*** class or a ***singleton***

- What is a singleton?
- Singleton means "only one". Each class can create one and only one instance no matter how many times it is used.
- The trick is: you can create an instance by yourself and provide a method to let everyone use this instance
- Make this class with a private constructor so that nobody can instantiate it



```java
public class Puppy {

    private static Puppy aDog;

    /** You have to write a private constructor
     *  otherwise java will make a public constructor for you!
     */
    private Puppy(){}

    public static Puppy getInstance() {

        if (aDog == null) {
            return new Puppy();
        }
        else {
            return aDog;
        }
    }
}
```

this instance is not created when your program is started. It is created when getInstance is first called.

- Extension
- In C++, there is another thing called destructor, which is automatically called when an object is destroyed.
- The reason why you need a destructor in C++ because you have to release the pointer by yourself.