Programming and Algorithms II

Lecture 3: Classes and Objects

Nicolas Wu

nicolas.wu@bristol.ac.uk

Department of Computer Science
University of Bristol

Classes and Objects

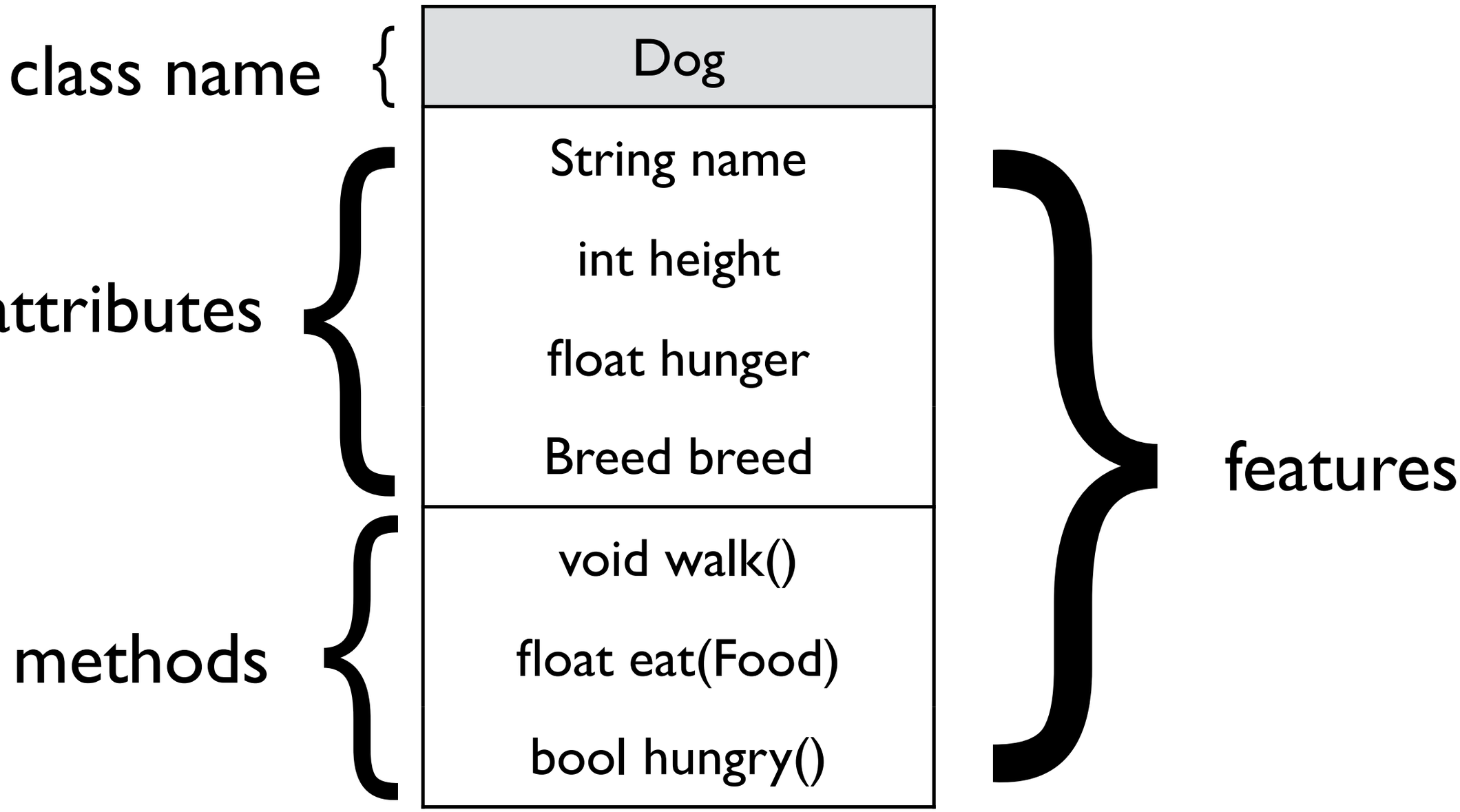
Classes

class
compile time
static
blueprint
abstract
immutable
type

Objects

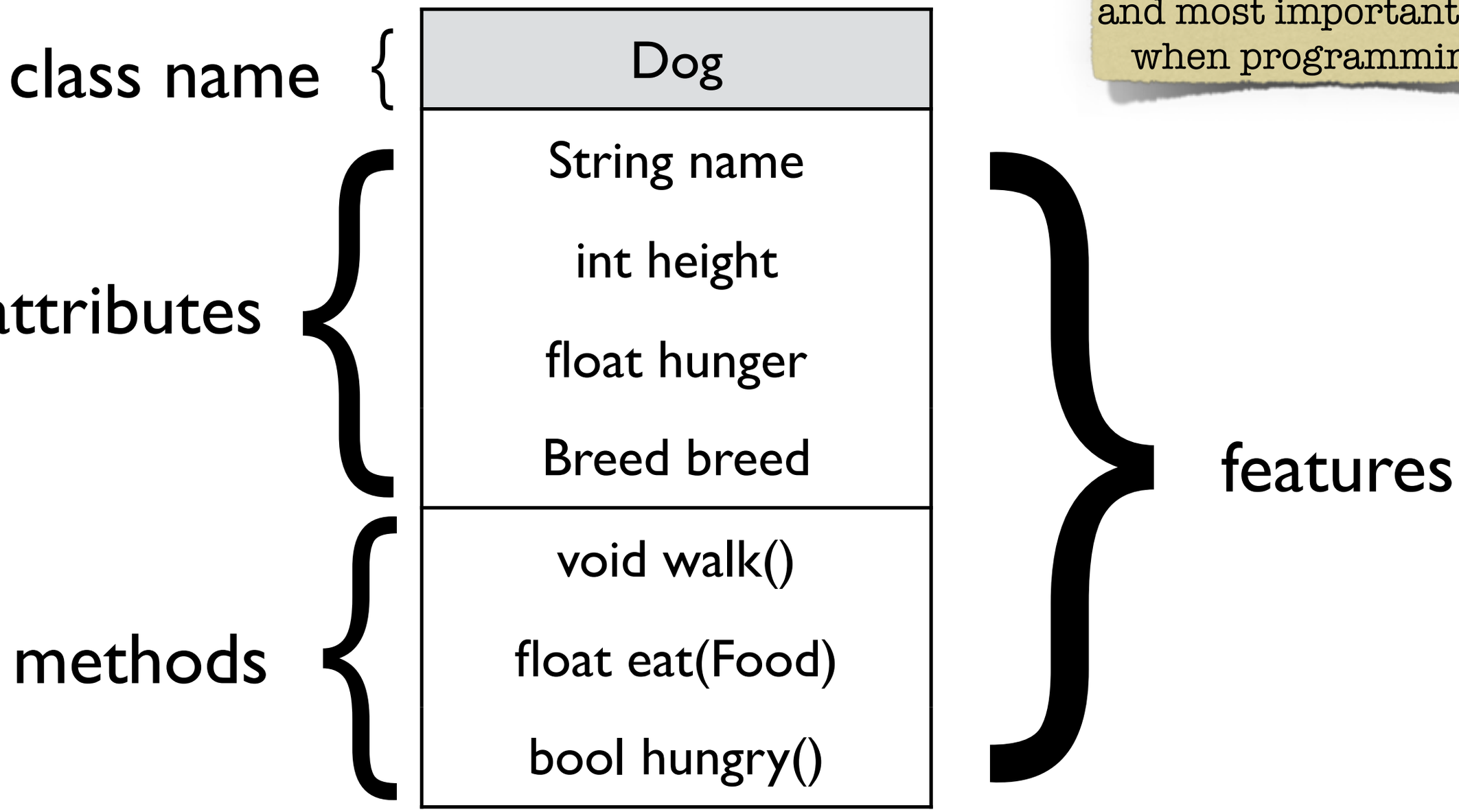
object
run time
dynamic
instance
concrete
stateful
value

Class Anatomy

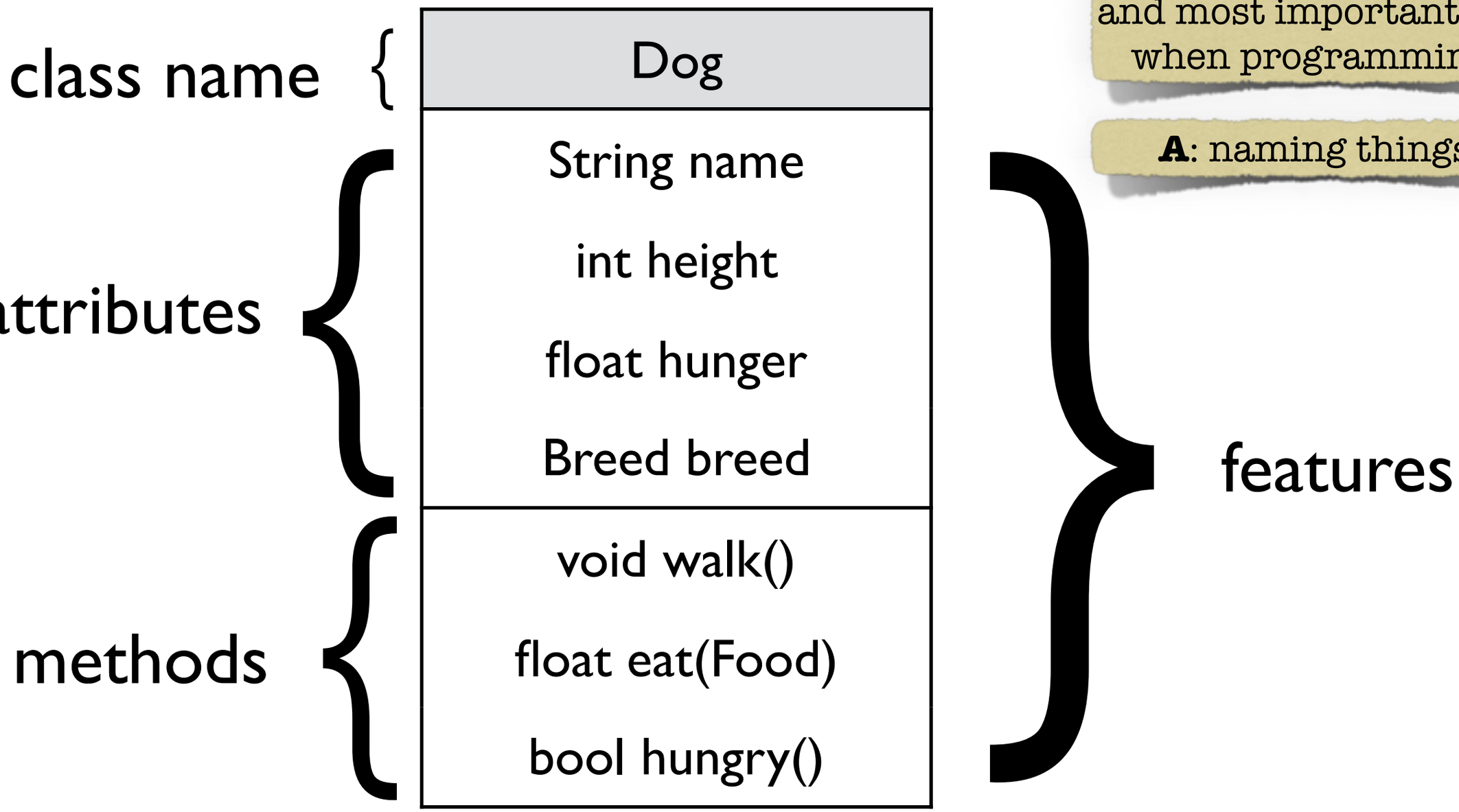


Class Anatomy

Q: What is the hardest and most important task when programming?



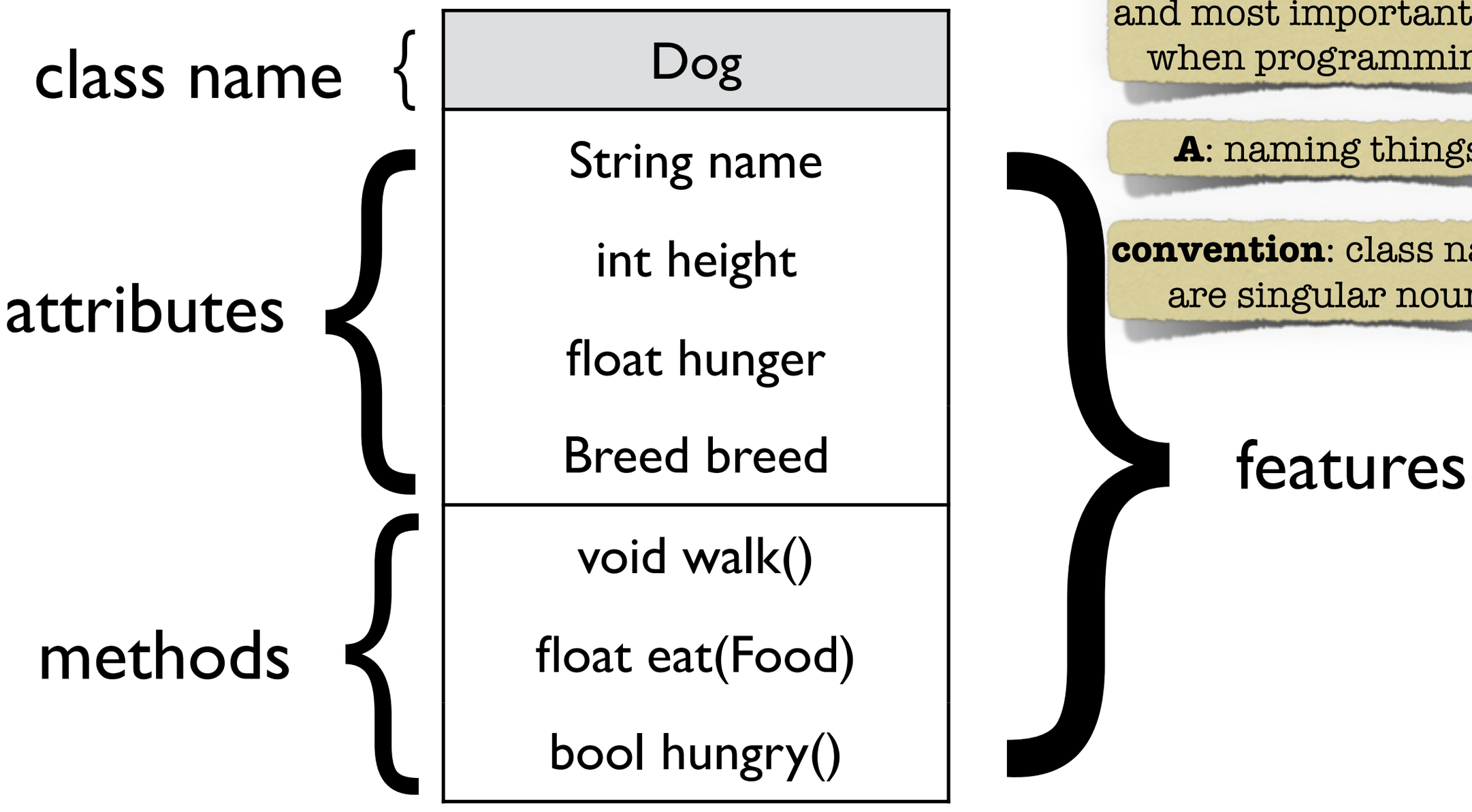
Class Anatomy



Q: What is the hardest and most important task when programming?

A: naming things!

Class Anatomy



Q: What is the hardest and most important task when programming?

A: naming things!

convention: class names are singular nouns

Class Implementation

Dog
String name int height float hunger Breed breed
void walk() float eat(Food) bool hungry()

```
class Dog {
  String name;
  Color  color;
  float  hunger;
  Breed  breed;

  void walk () {
    System.out.println(name + “walked”);
    hunger = hunger + 0.2;
  }

  int eat(Food food) {
    System.out.println(name + “ate”);
    hungry = 0;
  }

  bool hungry() {
    return (hunger > 1.0);
  }
}
```

Object Instantiation

- Creating new objects can be done with the **new** keyword:

```
Dog scooby = new Dog();
```

- Without parameters, this creates a default object, which can then be populated with data:

```
scooby.name  = “Scooby”;
scooby.color = new Color(139, 69, 19);
```

convention:
classes: upper case;
objects: lowercase

Attributes

Attributes

- Attributes capture what objects can **be**

```
String name;
Color  color;
float  hunger;
Breed  breed;
```

convention: attribute names are usually singular nouns (but plural if the attribute is a collection), or adjectives for boolean values

- Each object has its own copy of its variables
- Attributes can be plain old datatypes (PODs):

```
bool  char  int  float  double
```

- Attributes can be references to other objects

Methods

Methods

- Methods capture what objects can **do**

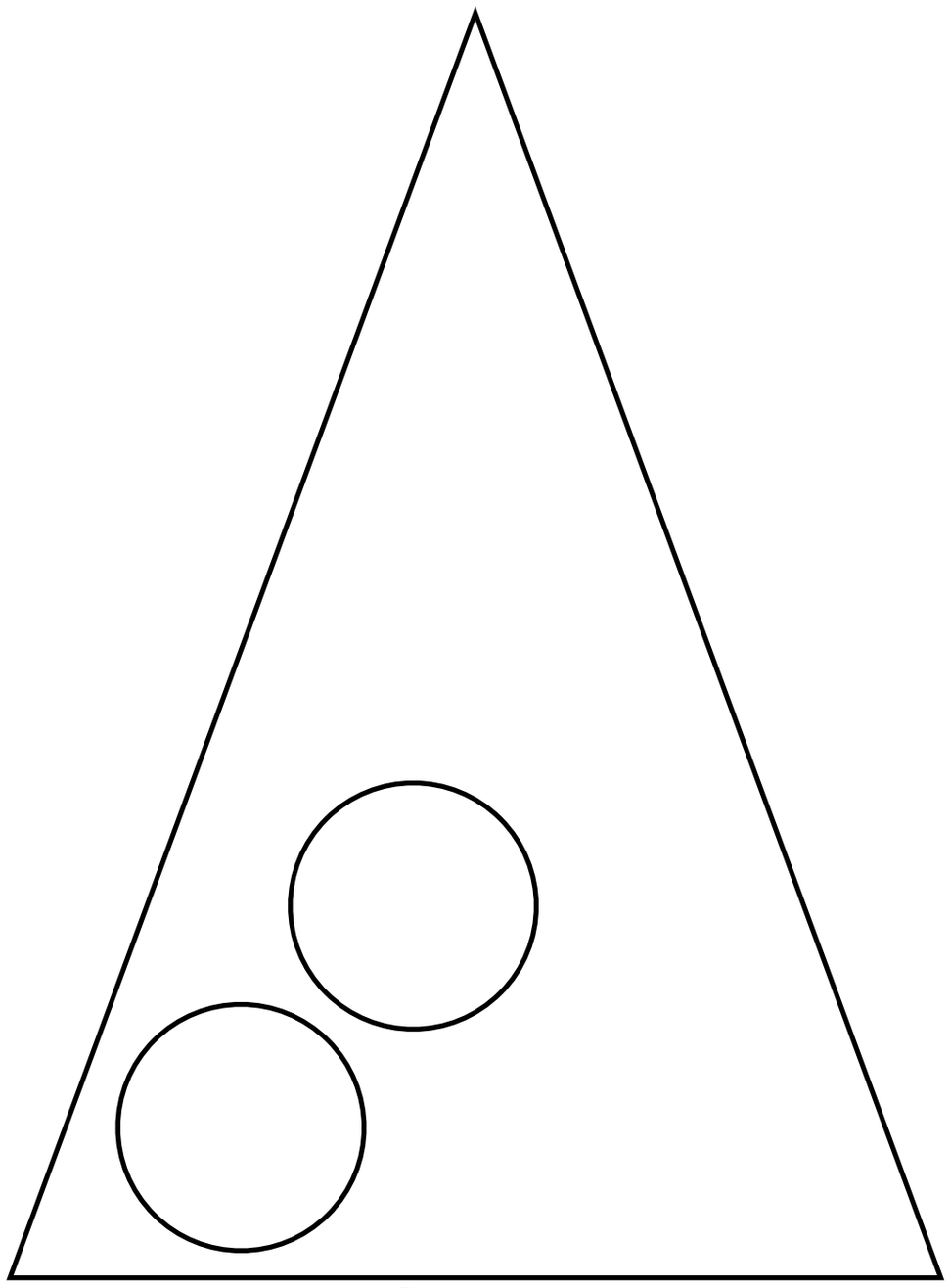
```
scooby.eat(“Scooby Snack”);
scooby.walk();
```

- Methods take parameters
- Methods return values
- Methods can be *overloaded* (don’t confuse that with *overriding* … see later)
- They are the main way to communicate with an object: sometimes we call them messages

convention: methods that do things are ideally verbs

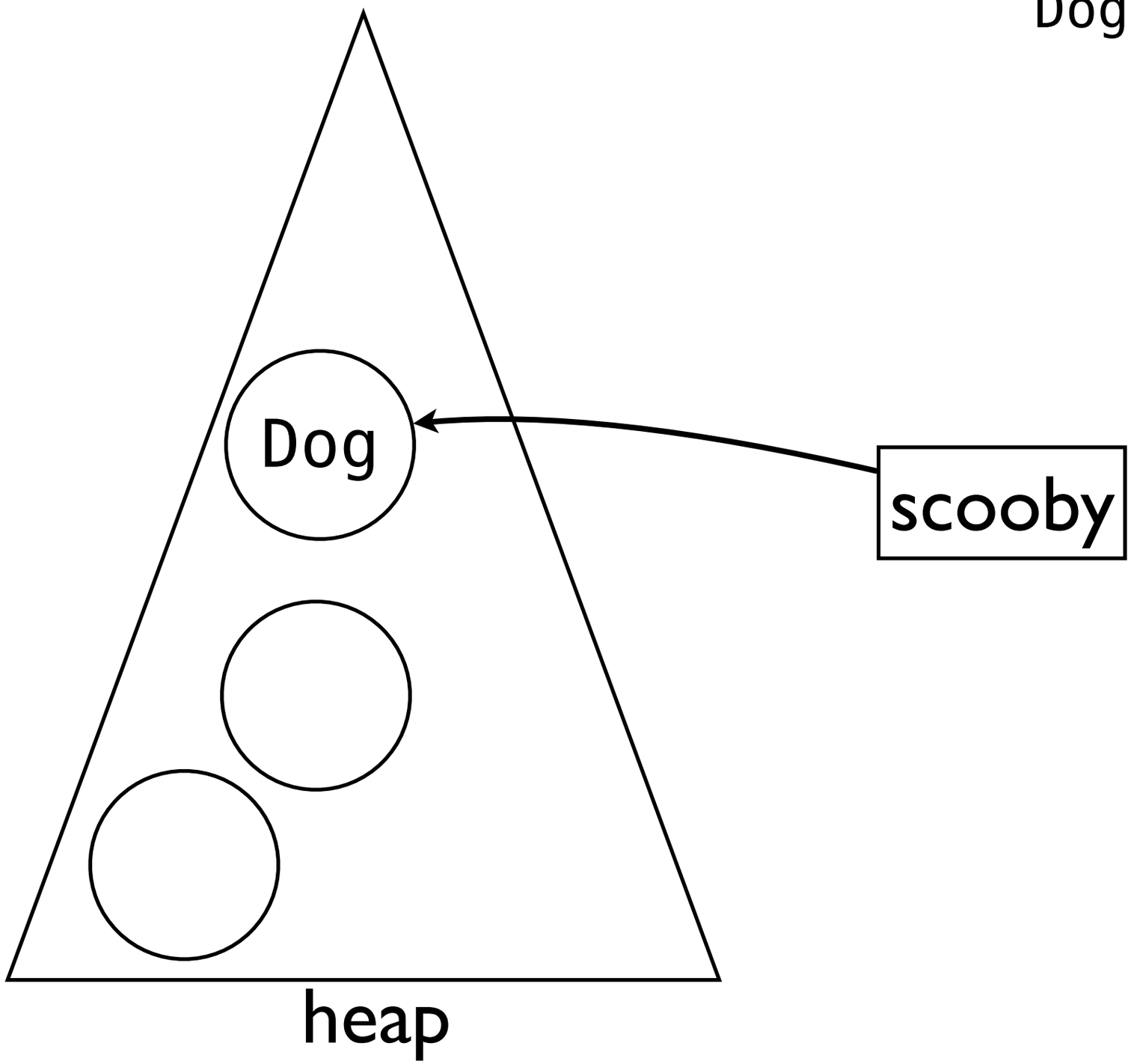
The Heap

The Life of Objects



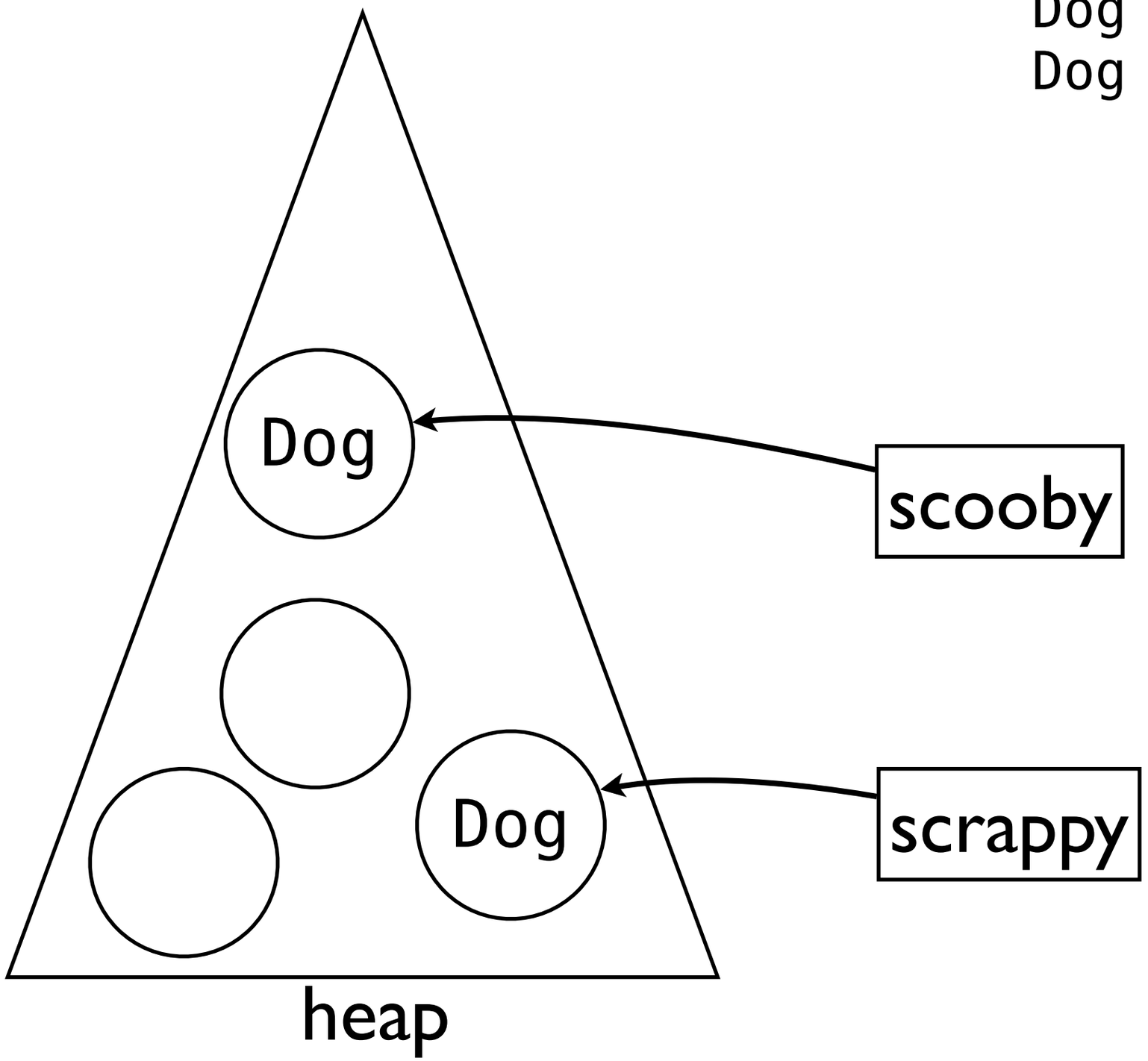
The Life of Objects

```
Dog scooby  = new Dog();
```



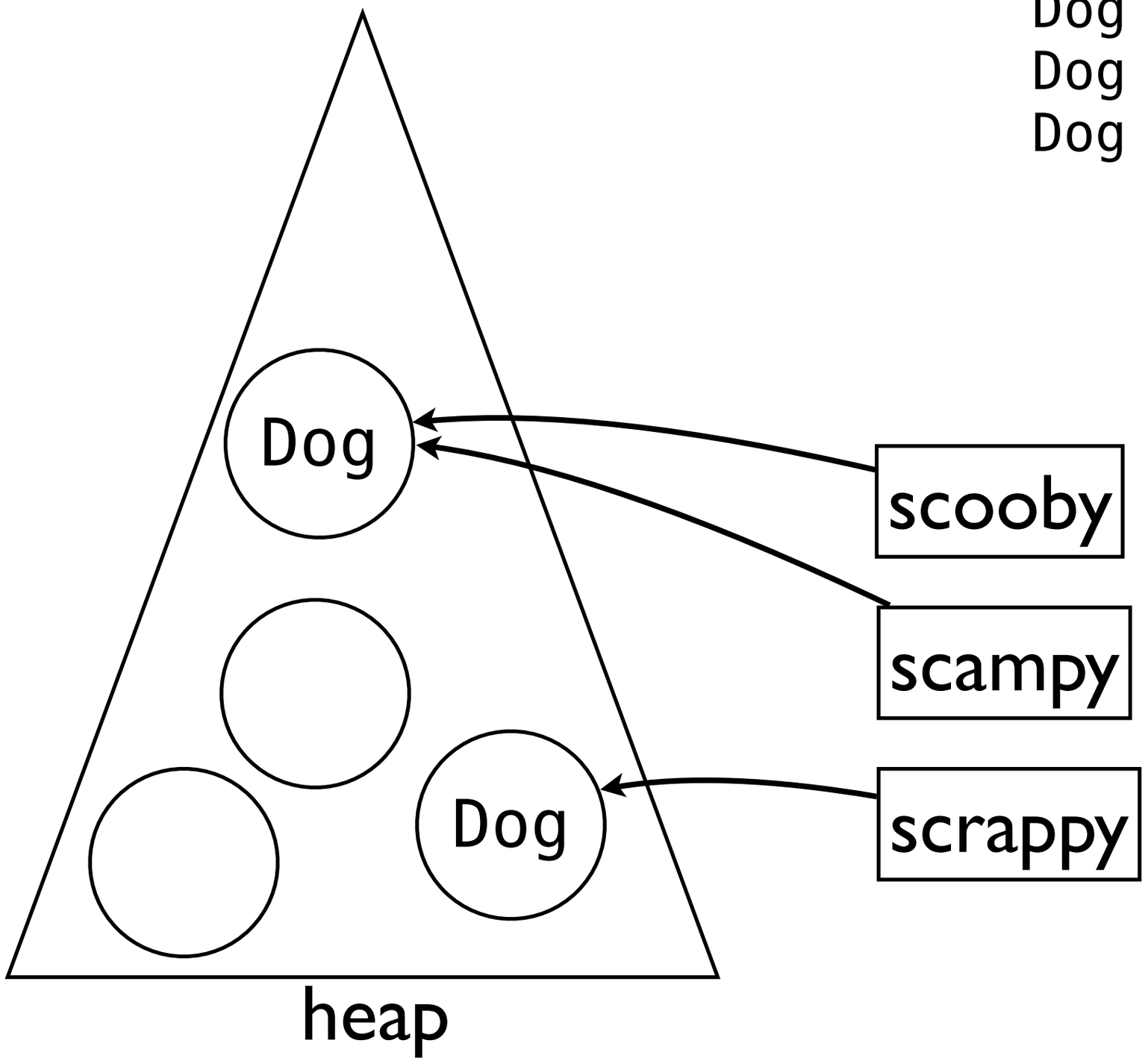
The Life of Objects

```
Dog scooby  = new Dog();
Dog scrappy = new Dog();
```



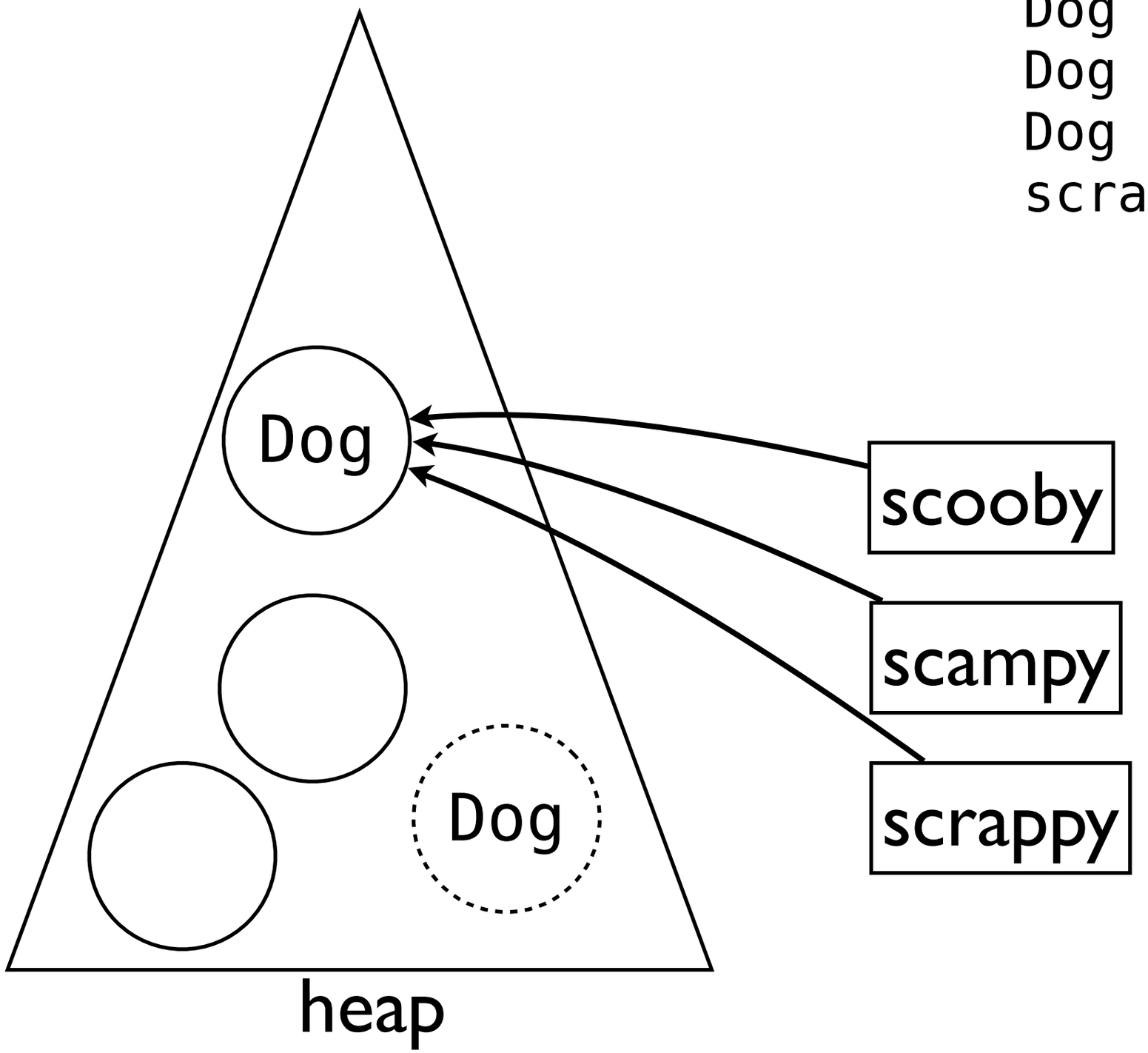
The Life of Objects

```
Dog scooby  = new Dog();
Dog scrappy = new Dog();
Dog scampy  = scooby;
```



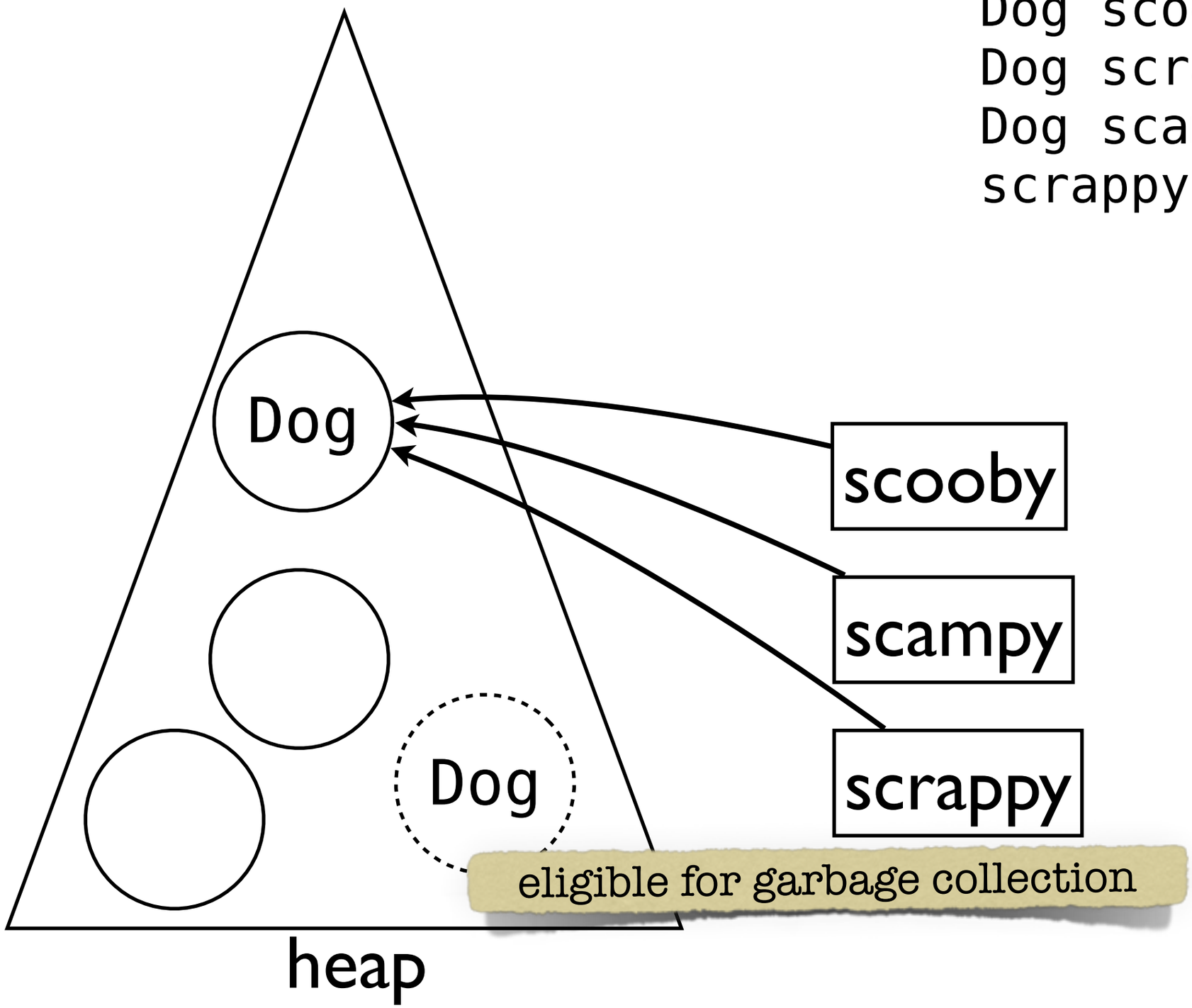
The Life of Objects

```
Dog scooby  = new Dog();
Dog scrappy = new Dog();
Dog scampy  = scooby;
scrappy     = scooby;
```



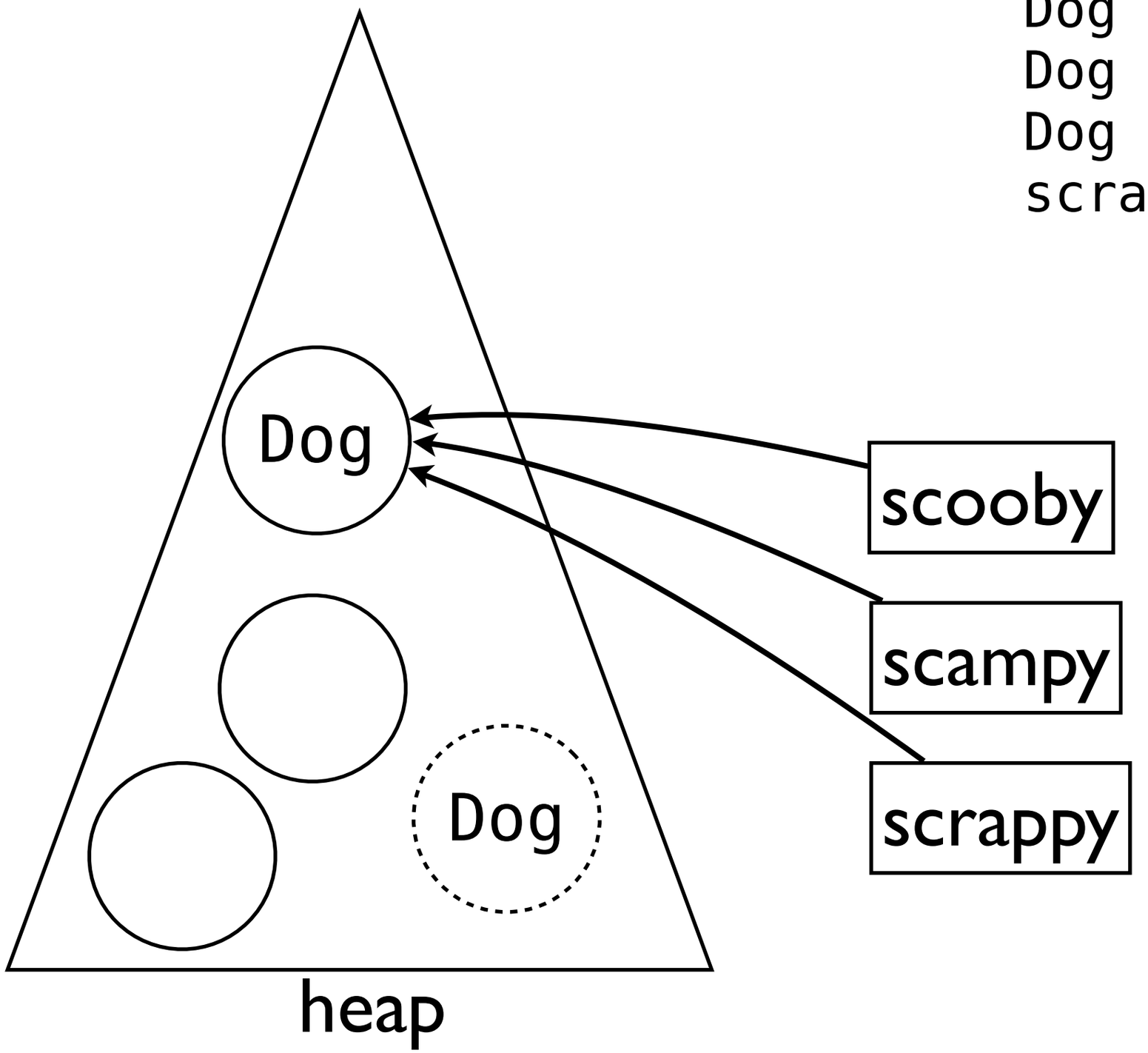
The Life of Objects

```
Dog scooby  = new Dog();
Dog scrappy = new Dog();
Dog scampy  = scooby;
scrappy     = scooby;
```



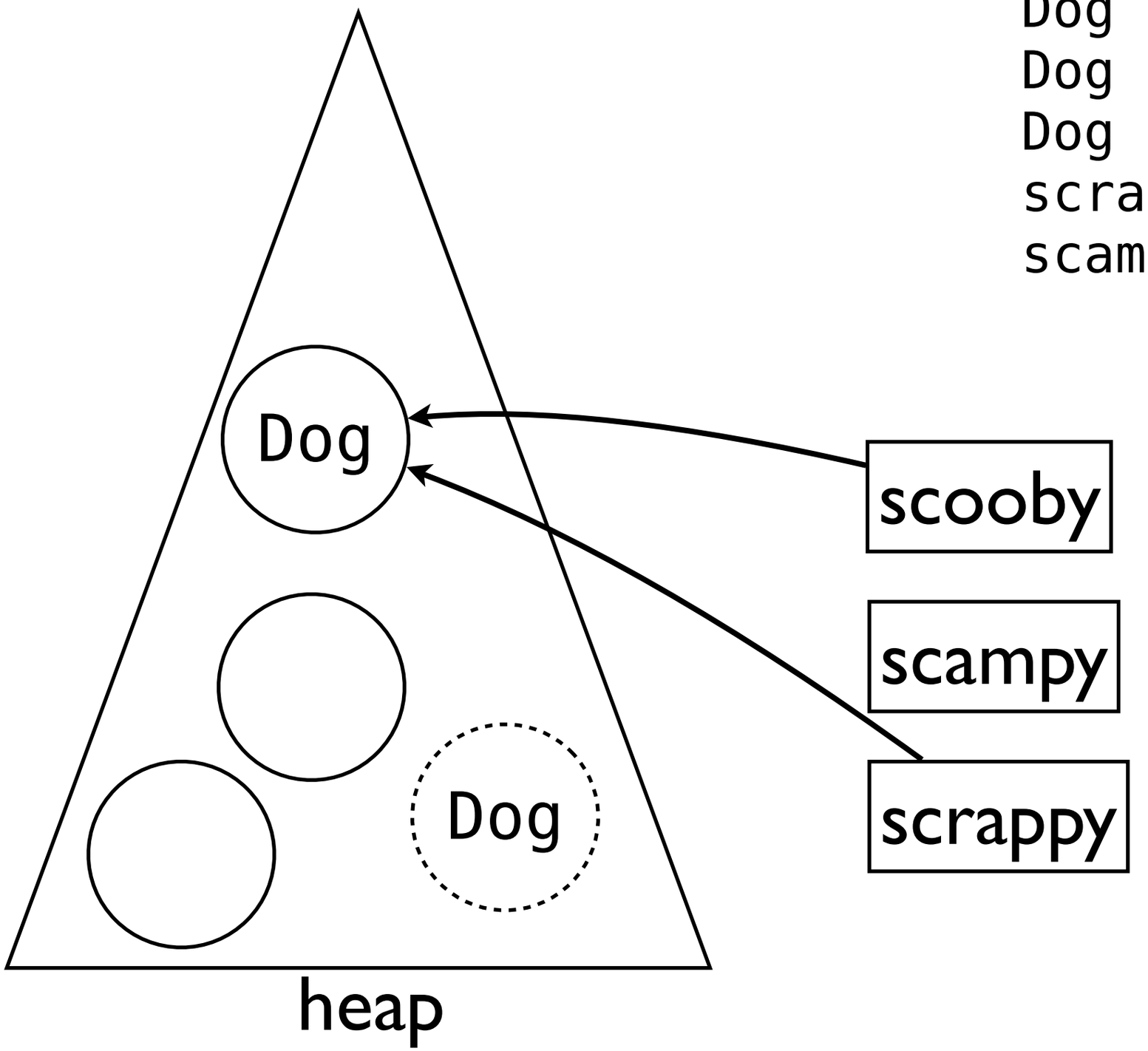
The Life of Objects

```
Dog scooby  = new Dog();
Dog scrappy = new Dog();
Dog scampy  = scooby;
scrappy     = scooby;
```



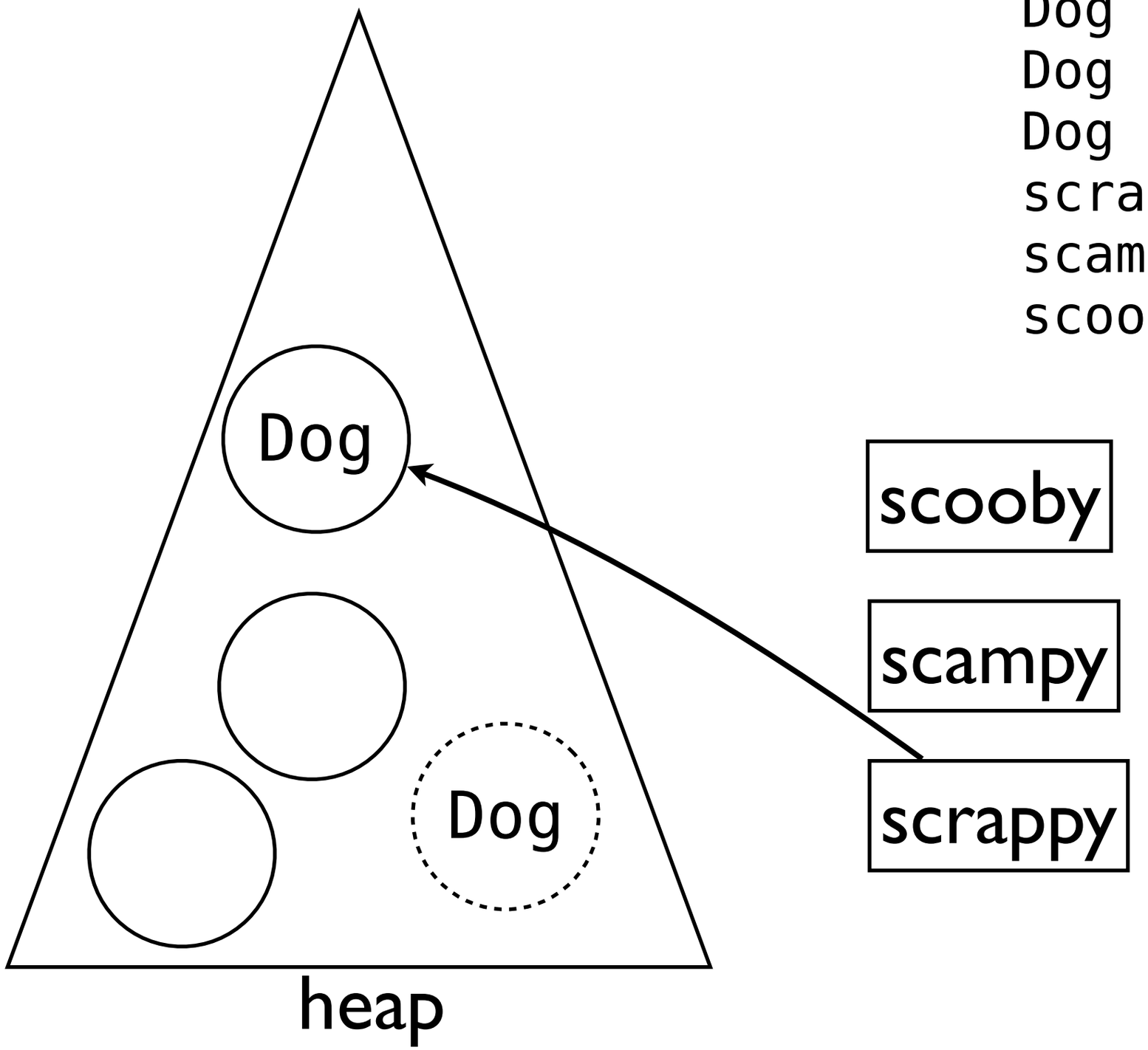
The Life of Objects

```
Dog scooby  = new Dog();
Dog scrappy = new Dog();
Dog scampy  = scooby;
scrappy     = scooby;
scampy      = null;
```

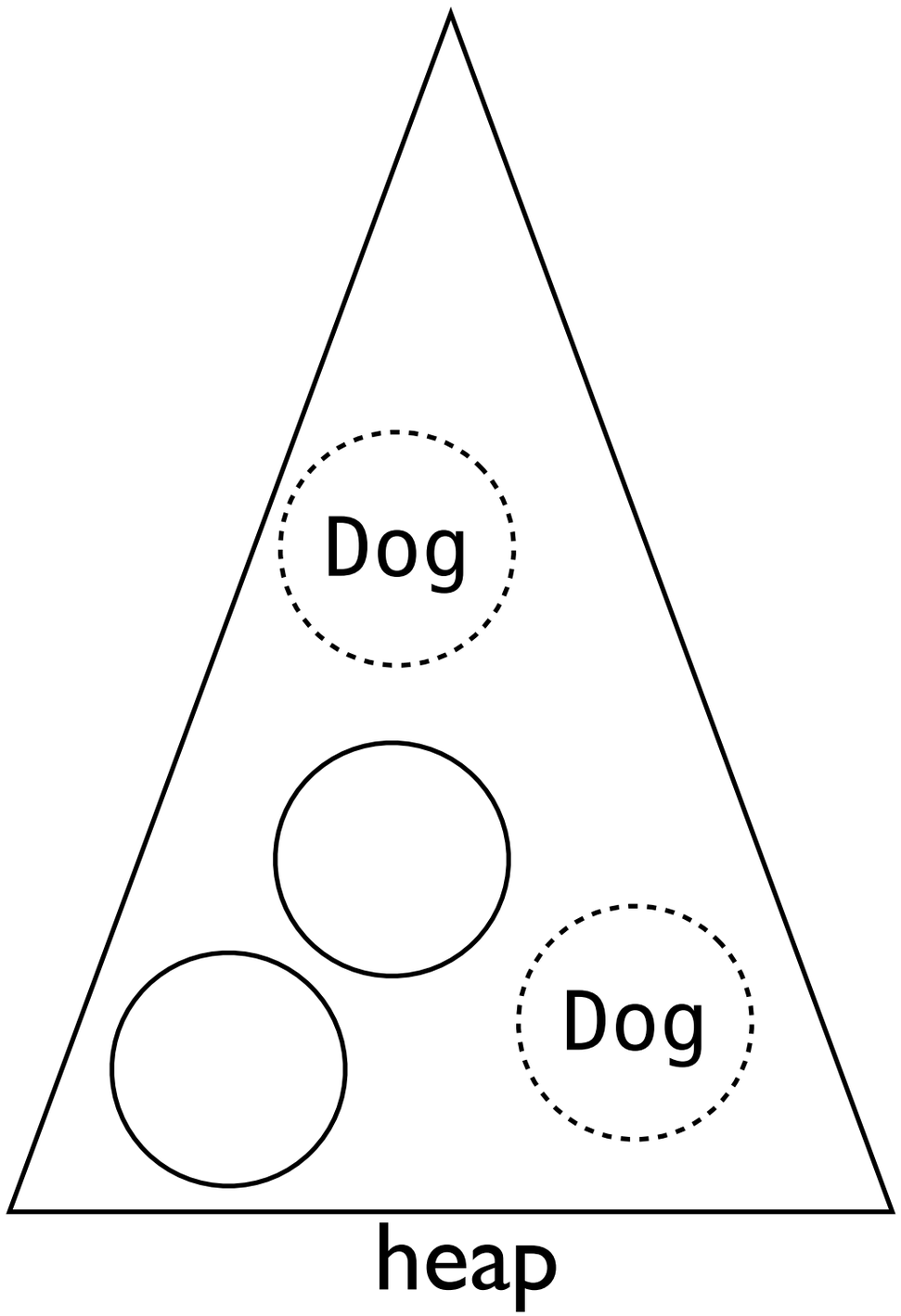


The Life of Objects

```
Dog scooby  = new Dog();
Dog scrappy = new Dog();
Dog scampy  = scooby;
scrappy     = scooby;
scampy      = null;
scooby      = null;
```



The Life of Objects



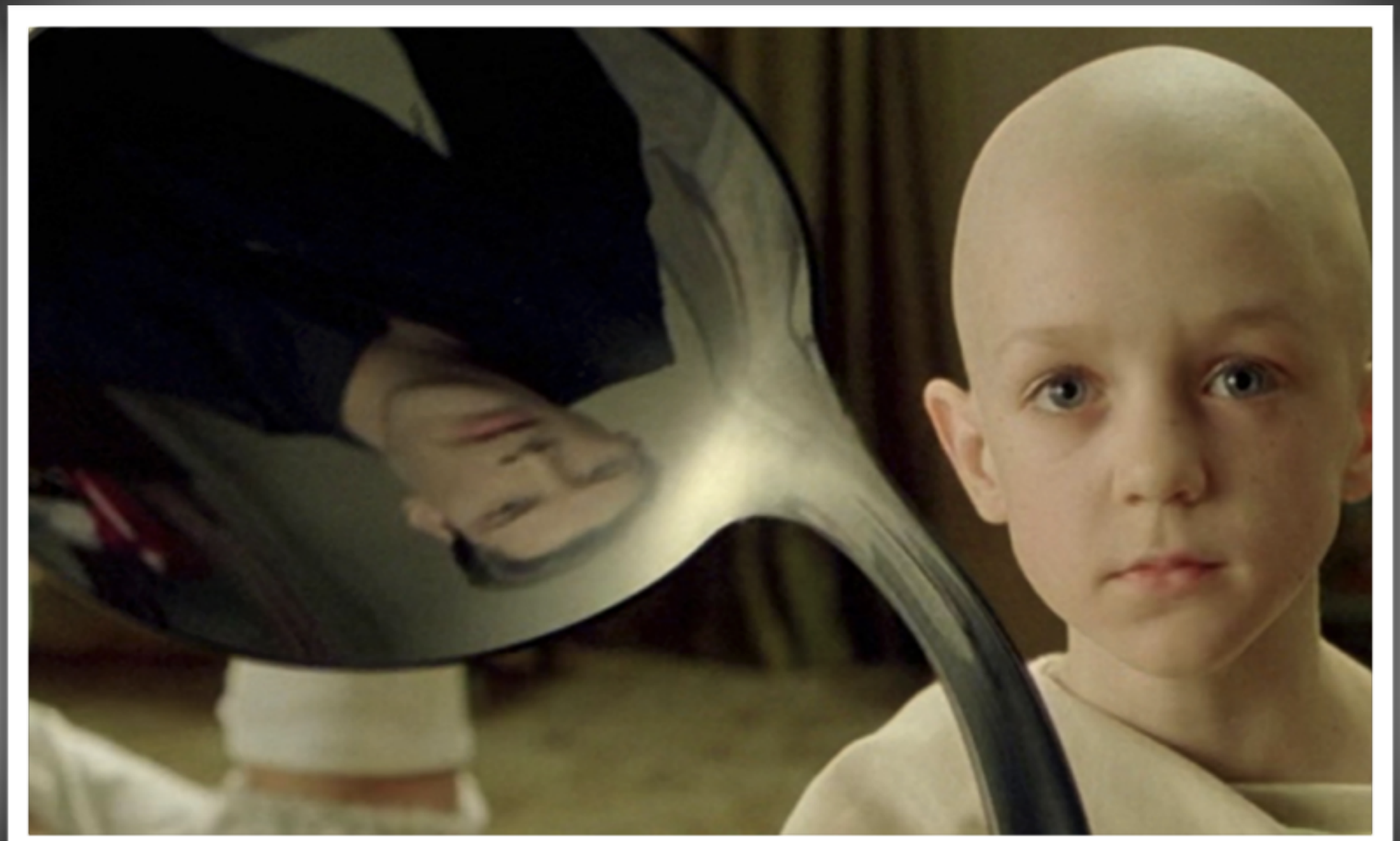

```
Dog scooby  = new Dog();
Dog scrappy = new Dog();
Dog scampy  = scooby;
scrappy     = scooby;
scampy      = null;
scooby      = null;
scrappy     = null;
```

scooby

scampy

scrappy

References



References

```
class Spoon {
   …
}

Spoon spoon = new Spoon ();
```

- There is no spoon?
- Well, spoon isn't really an *object*
- It's a *reference* to a spoon, and we can do this:

```
Spoon spoon' = spoon;
```

Null References

- You can also create a reference to nothing

```
Spoon spoon = null;
```

- I wish I didn't have to tell you about these (I nearly didn't!)
- What could possibly go wrong?

Null References

- You can also create a reference to nothing

```
Spoon spoon = null;
```

- I wish I didn't have to tell you about these (I nearly didn't!)
- What could possibly go wrong?

```
spoon.scoop();
```

Null References

- You can also create a re

```
Spoon spoon
```

- I wish I didn't have to (I nearly didn't!)
- What could possibly

```
spoon.scoop();
```

Null References

"I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years."

— Tony Hoare

Constructors

Object Instantiation

- Creating new objects can be done with the **new** keyword:

```
Dog scooby = new Dog();
```

- Without parameters, this creates a default object, which can then be populated with data:

```
scooby.name  = “Scooby”;
scooby.color = new Color(139, 69, 19);
```

Object Instantiation

- Creating new objects can be done with the **new** keyword:

```
Dog scooby = new Dog();
```

- Without parameters, this creates a default object, which can then be populated with data:

```
scooby.name  = “Scooby”;
scooby.color = new Color(139, 69, 19);
```

- That works just fine, so why is the following preferable?

```
Color brown = Color (139, 69, 19);
scooby      = new Dog(“Scooby”, brown);
```

Object Instantiation

- Creating new objects can be done with the **new** keyword:

```
Dog scooby = new Dog();
```

- Without parameters, this creates a default object, which can then be populated with data:

```
scooby.name  = “Scooby”;
scooby.color = new Color(139, 69, 19);
```

- That works just fine, so why is the following preferable?

```
Color brown = Color (139, 69, 19);
scooby      = new Dog(“Scooby”, brown);
```

- Sometimes the “default” simply makes no sense!

Constructors

- Create a new Dog.java file
- Demonstrate the creation of scooby
- Add a constructor method
- Show that parameters are now *required*

This Reference

```
class Dog {
  String name;
  String color;

  Dog(String name, String color) {
    this.name  = name;
    this.color = color;
  }

  void talk () {
    System.out.println("My name is " + name);
  }


  public static void main (String[] args) {
    Dog scooby = new Dog("Scooby", "Great Dane");
    scooby.talk();
  }
}
```

the constructor method:

- named after the class
- no return value

This Reference

```
class Dog {
  String name;
  String color;

  Dog(String name, String color) {
    this.name  = name;
    this.color = color;
  }

  void talk () {
    System.out.println("My name is " + name);
  }


  public static void main (String[] args) {
    Dog scooby = new Dog("Scooby", "Great Dane");
    scooby.talk();
  }
}
```

"this" refers to the current object

Arrays

Arrays

- Arrays in Java are objects too! It might be tempting to write:

```
Dog[] dogs;
```

but of course we risk invoking the wrath of null pointer exceptions.

- What would we expect the constructor to look like?

```
Dog[] dogs = new Dog[]();
Dog[] dogs = new Dog[](4);
```

```
Dog[] dogs = new Dog[];
Dog[] dogs = new Dog[4];
```

Arrays

- New arrays need a size on initialisation, and use special notation that mirrors C:

```
Dog[] dogs = new Dog[4];
```

- Alas, this still invokes the wrath of null pointer exceptions. Why?

Arrays

- New arrays need a size on initialisation, and use special notation that mirrors C:

```
Dog[] dogs = new Dog[4];
```

- Alas, this still invokes the wrath of null pointer exceptions. Why?

- Consider the following:

```
Dog[] dogs = new Dog[4];
dogs[0].bark();
```

Arrays

- New arrays need a size
  use special notation tha

```
Dog[] dogs = new
```

- Alas, this still invokes t
  pointer exceptions. W

- Consider the followin

```
Dog[] dogs = new Dog[4];
dogs[0].bark();
```

Arrays

- To avoid the wrath of nulls, arrays also need to have their contents initialised!

```
Dog[] dogs = new Dog[4];
dogs[0] = scooby;
dogs[1] = scrappy;
dogs[2] = scampy;
dogs[3] = new Dog(“Scratchy”, “Pug”);
```

Arrays

- To avoid the wrath of nulls, arrays also need to have their contents initialised!

```
Dog[] dogs = new Dog[4];
dogs[0] = scooby;
dogs[1] = scrappy;
dogs[2] = scampy;
dogs[3] = new Dog(“Scratchy”, “Pug”);
```

Can we do any better?

Arrays

- To avoid the wrath of nulls, arrays also need to have their contents initialised!

```
Dog[] dogs = new Dog[4];
dogs[0] = scooby;
dogs[1] = scrappy;
dogs[2] = scampy;
dogs[3] = new Dog(“Scratchy”, “Pug”);
```

- If you’re especially pedantic, you’d write …

```
Dog[] dogs = new Dog[]
  { scooby
  , scrappy
  , scampy
  , new Dog(“Scratchy”, “Pug”) };
```

Arrays

- To avoid the wrath of nulls, arrays also need to have their contents initialised!

```
Dog[] dogs = new Dog[4];
dogs[0] = scooby;
dogs[1] = scrappy;
dogs[2] = scampy;
dogs[3] = new Dog(“Scratchy”, “Pug”);
```

- If you’re especially pedantic, you’d write …

```
Dog[] dogs = new Dog[]
  { scooby
  , scrappy
  , scampy
  , new Dog(“Scratchy”, “Pug”) };
```

(this part is optional)

Arrays

- You can always find out the size of an array

```
n = dogs.length;
```

- Arrays can be iterated over without counters

```
for (Dog dog : dogs) {
  dog.bark()
}
```

this is simply awesome

- This goes through the array, each time through the loop it assigns to dog

Awesome Calculator

- Make a calculator, but this time iterate over the args, rather than use a counter