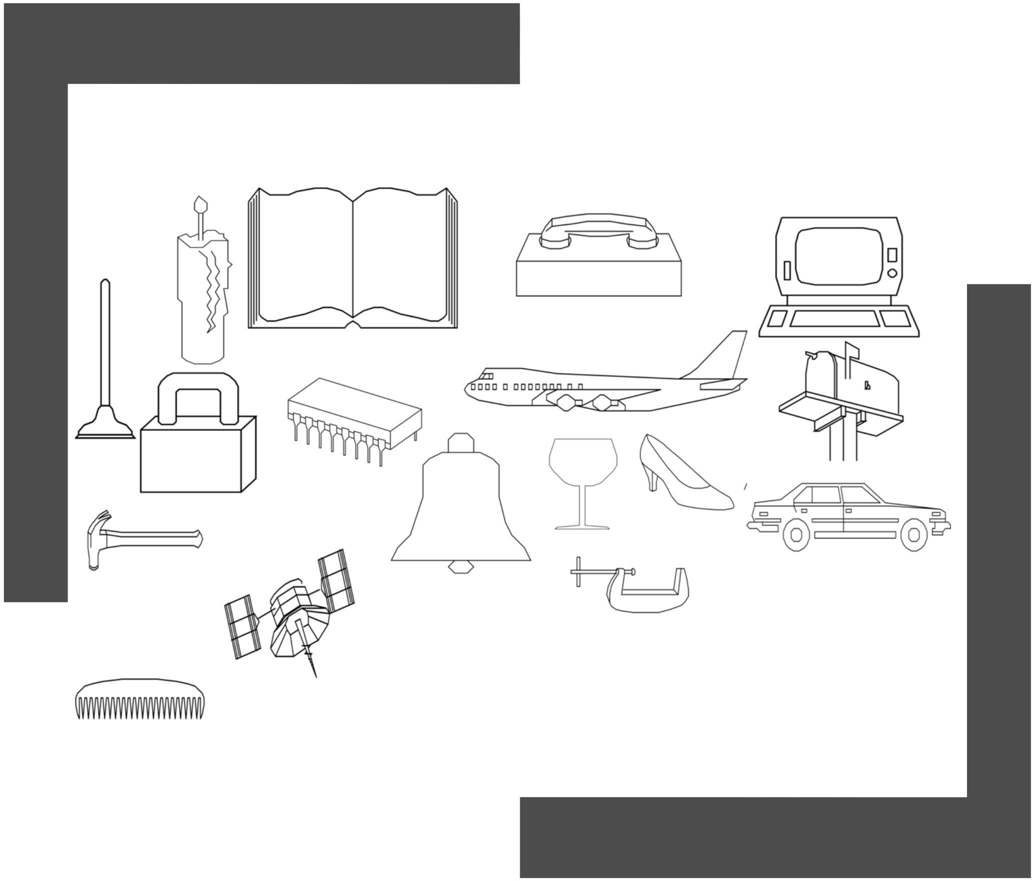
Classes and objects

Lecture 3

by Marina Barsky

Software objects



- Real objects in the real world have
  - things that they can do (*actions, methods*)
  - things that describe them (*attributes, properties*)

- **In programming, we have the same kind of thing**

Change of perspective

What sounds more natural?

```
cook (microwave, chicken)
```

```
microwave.cook (chicken)
```

- The functionality of real-world objects tends to be tightly bound up **inside the objects themselves**
- We will learn how to bundle together data and actions inside a single software construct called *object*

With objects we can model anything

- Physical objects: *House*, *Room*
- Persons: *Student*, *Patient*
- Abstract concepts: *Time*, *Relationship*
- Processes: *Simulation*, *GamePlay*

Everything in Java must have a type (*typed language*)

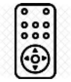
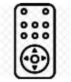
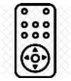
- Before creating any new objects, we must first define a new type or a class of objects

Here is one:

```
class Dog{
    String name;
    String breed;
    int size;
    double weight;
}
```

How to create an array of Dogs

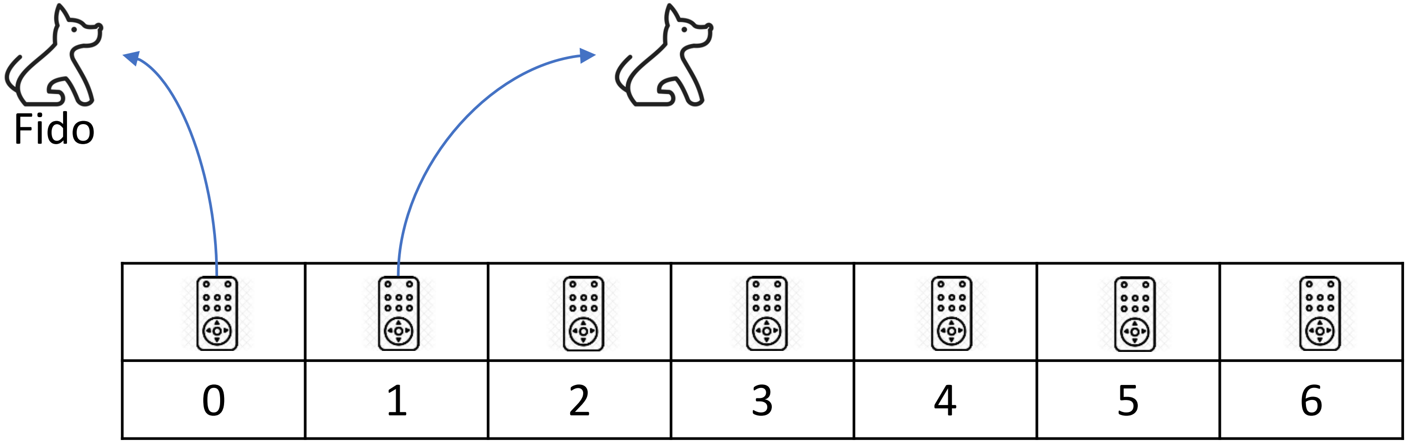
```
Dog pets = new Dog[7];
```

						
0	1	2	3	4	5	6

- This is array of references not array of dogs!
- What is missing?
- Actual dogs

How to create an array of Dogs

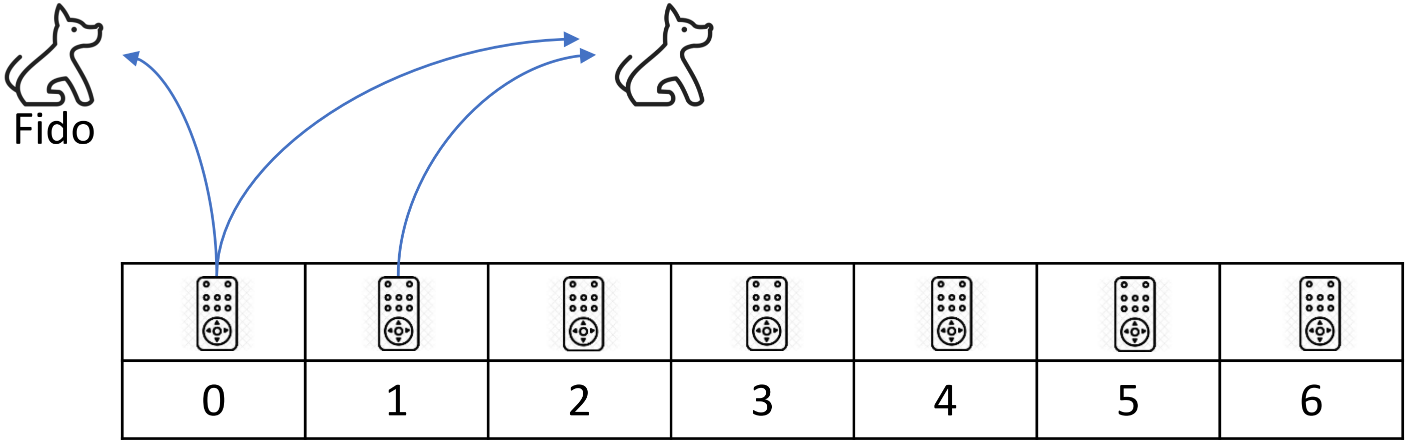
```
Dog pets = new Dog[7];
```



```
pets[0] = new Dog();
pets[1] = new Dog();
pets[0].name = "Fido";
```

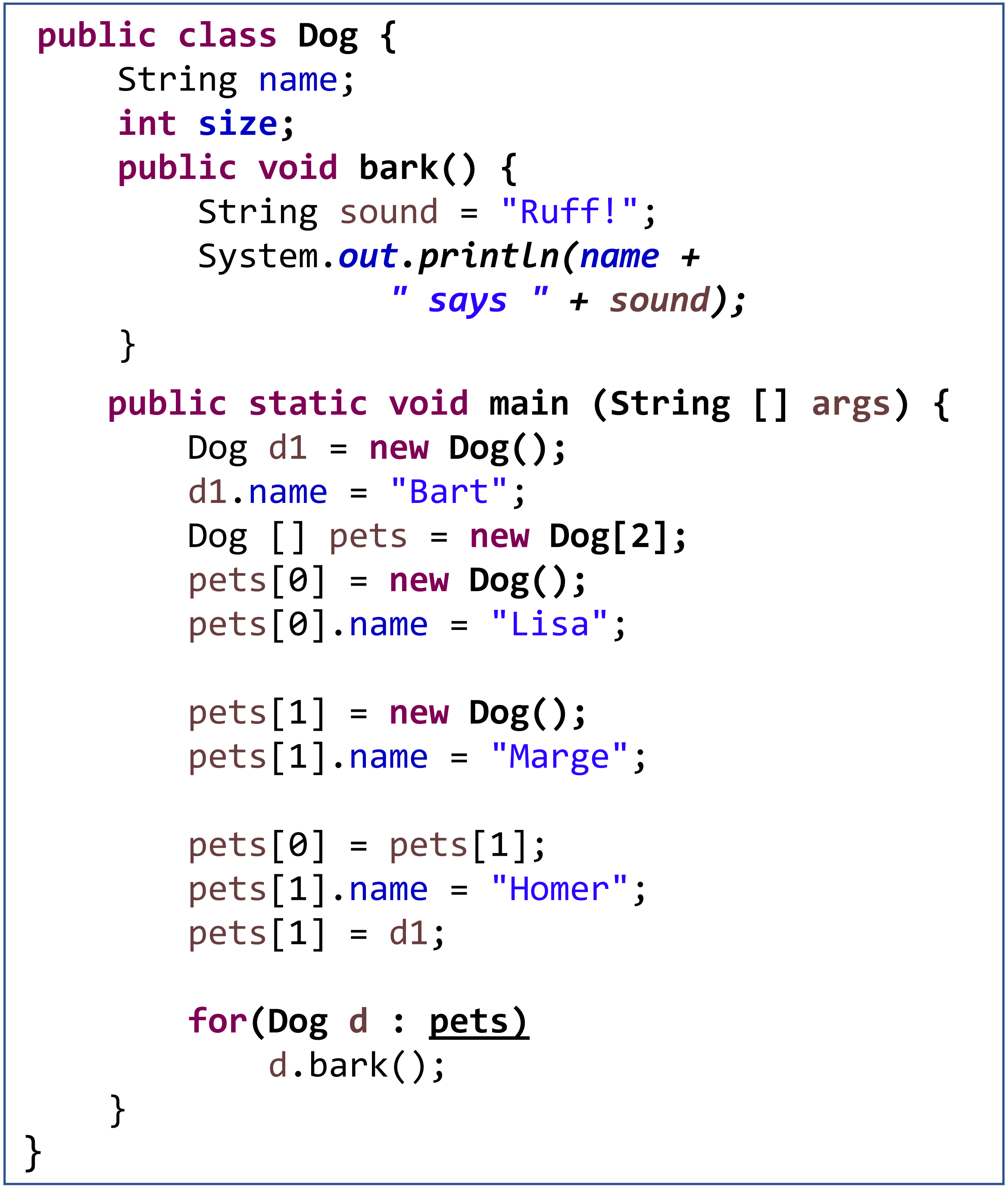
How to create an array of Dogs

```
Dog pets = new Dog[7];
```



```
pets[0] = new Dog();
pets[1] = new Dog();
pets[0].name = "Fido";
pets[0] = pets[1];
```

- Who references “Fido”?
- What is stored in pets[2]?
- What is it pointing to?



```
public class Dog {
    String name;
    int size;
    public void bark() {
        String sound = "Ruff!";
        System.out.println(name +
                 " says " + sound);
    }
    public static void main (String [] args) {
        Dog d1 = new Dog();
        d1.name = "Bart";
        Dog [] pets = new Dog[2];
        pets[0] = new Dog();
        pets[0].name = "Lisa";

        pets[1] = new Dog();
        pets[1].name = "Marge";

        pets[0] = pets[1];
        pets[1].name = "Homer";
        pets[1] = d1;

        for(Dog d : pets)
            d.bark();
    }
}
```

- What is printed?

A

```
Lisa says Ruff!
Homer says Ruff!
```

B

```
Homer says Ruff!
Bart says Ruff!
```

C

```
Lisa says Ruff!
Marge says Ruff!
```

D

```
Bart says Ruff!
Bart says Ruff!
```

E NONE OF THE ABOVE

```
public class Dog {
    String name;
    int size;
    public void bark() {
        String sound = "Ruff!";
        System.out.println(name +
                  " says " + sound);
    }

   public static void main (String [] args) {
       Dog d1 = new Dog();
       d1.name = "Bart";
       Dog [] pets = new Dog[2];
       pets[0] = new Dog();
       pets[0].name = "Lisa";

       pets[1] = new Dog();
       pets[1].name = "Marge";

       pets[0] = pets[1];
       pets[1].name = "Homer";
       pets[1] = d1;

       for(Dog d : pets)
           d.bark();
   }
}
```

- How many references?

  3

- How many total objects allocated on the heap?

  3

- How many abandoned objects?

  1

- What is the name of an abandoned Dog?

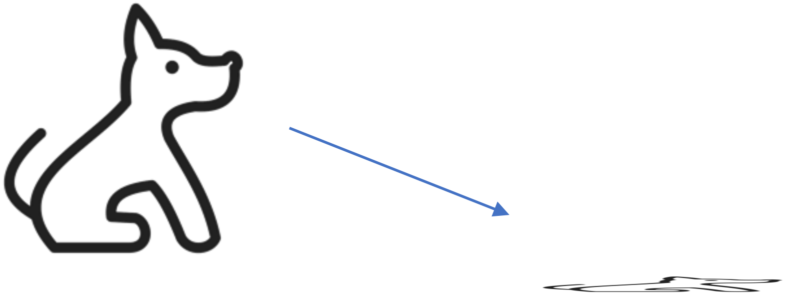
  “Lisa”

Bad Idea: exposing instance variables

```
public class BadDog {
    public String name;
    public int height;
    public void bark() {
        …
    }


    public static void main (String [] {
        BadDog d = new BadDog();
        d.height = 0;
    }
}
```

- We should never allow direct access to instance variables
- See what may happen!



Access Modifiers

- `public`, `private`, and `protected` are called ***access modifiers***
- They control access of other classes to instance variables and methods of a given class
  - `public`: Accessible to all other classes
  - `protected`: Accessible to the class declaring it and its subclasses
  - `no modifier`: Accessible to the class declaring it and all classes in the same package
  - `private`: Accessible only to the class declaring it

Data-Hiding Principle (*Encapsulation*)

- Make instance variables **`private`**
- Use **`public`** methods to access/modify object data

- The methods are called accessors/mutators
- We will call them getters/setters
  - Getter: get some value back
  - Setter: set value of some instance variable

Example of Data Hiding

```
public class GoodDog {
    private String name;
    private int height;


    public void setHeight (int h) {
        if (height > 0)
            height = h;
    }

    public int getHeight() {
        return height;
    }
}
```

declared
as private

setter

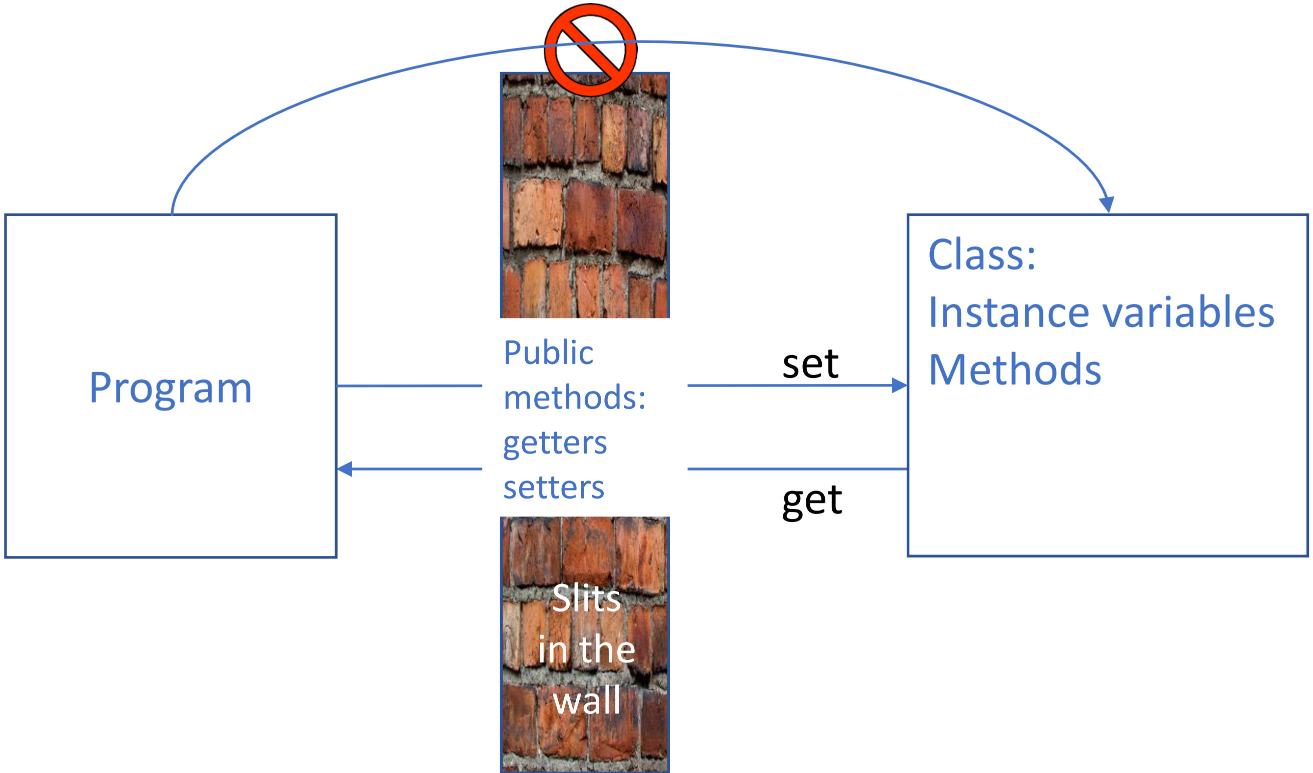
getter

Build an impenetrable wall around your data

- Programs that use your classes should NOT:

  be able to change the value of the instance variables directly

- Restrict the access to an object's data so you can only get it or change it by using methods



Advantages of Data Hiding

With Data Hiding and Encapsulation we can:

- validate the parameter passed to the method
- reject unacceptable values (such as negative year): ignore them or throw an exception
- round the value to the closest valid or default value
- change method and make it faster/safer without changing any code that uses our class

Setting up initial values

```
public class PoorDog {
    private String name;
    private int height;

    …

    public int getHeight() {
        return height;
    }

    public String getName() {
        return name;
    }
}
```

- We do not want flattened dog with name *null*!
- How do we ensure that this never happens?
- Where do we perform object setup – where do we set the initial object state?

- Inside main:

```
PoorDog d = new PoorDog();
System.out.println("dog’s height is: "+ d.getHeight());
System.out.println("dog’s name is: "+ d.getName());
```

Three steps of object creation

```
Dog d = new Dog();
```

1 Declare reference variable

```
Dog d = new Dog();
```

2 Create new Dog object

```
Dog d = new Dog();
```

3 Connect reference to object

```
public class Dog {
    private String name;
    private int height;


    public void setHeight (int h) {
        if (height > 0)
            height = h;
    }

    public int getHeight() {
        return height;
    }
}
```

Dog() is called a constructor

```
Dog d = new Dog();
```

- Are we calling some method named **Dog()**?
- Where is this method defined?

- The compiler writes a default constructor method for you if you did not define it:

```
public Dog(){
   //do nothing
}
```

```
public class Dog {
    private String name;
    private int height;


    public void setHeight (int h) {
        if (h > 0)
            height = h;
    }

    public int getHeight() {
        return height;
    }
}
```

How is constructor different from a normal method?

```
public class Dog {
    private String name;
    private int height;

    public Dog(){
    }

    public void setHeight (int h) {
        if (height > 0)
            height = h;
    }

    public int getHeight() {
        return height;
    }
}
```

A. There is no return type

B. The name is exactly the same as the name of the class

C. There are no method parameters

D. All of the above

E. Only A and B are true

Constructor

- The code in constructor runs **before** the object is assigned to the reference variable
- This is our chance to initialize everything that needs to be initialized
- In most cases: we initialize instance variables

```
public class Dog {
    private String name;
    private int height;

    public Dog(){
        height = 10;
        name = "Unnamed“;
    }

    public void setHeight (int h) {
        if (h > 0)
            height = h;
    }

    public int getHeight() {
        return height;
    }
}
```

Constructors with parameters

- We can force the user of our class to pass parameters during object creation

- Both constructors require that at least the `height` of the Dog is specified

- Each overloaded constructor must have a different signature

```
public class Dog {
   private String name;
   private int height;

   public Dog(int height){
       this.height = height;
       this.name = "Unnamed“;
    }

    public Dog(int height, String name){
       this.height = height;
       this.name = name;
    }

    public void setHeight (int h) {
        if (h > 0)
            height = h;
    }
```

Does compiler always make a default constructor? **NO!**

- If we explicitly defined at least one constructor in our code, we do not have a default constructor (without parameters) anymore:

```
Dog d = new Dog(); ✖
```

- This will not compile: there is no constructor without parameters

```
public class Dog {
   private String name;
   private int height;

   public Dog(int height){
       this.height = height;
       this.name = "Unnamed“;
    }

    public Dog(int height, String name){
       this.height = height;
       this.name = name;
    }

    public void setHeight (int h) {
        if (h > 0)
            height = h;
    }
```

You must add default constructor explicitly

```
Dog d = new Dog();
```

- This will work now

```
public class Dog {
    private String name;
    private int height;

     public Dog(int height){
        this.height = height;
        this.name = "Unnamed“;
    }

    public Dog(int height, String name){
        this.height = height;
        this.name = name;
    }

    public Dog(){
        this.height = 10;
        this.name = "Unnamed“;
    }
```

Defining a new type (class):

We need:

- Data fields = attributes = instance variables
- Capabilities = methods
- Constructor(s): setting up default values

Encapsulation

- Data hiding and protection of object’s data from illegal changes is a part of a very important principle in OOP: *encapsulation*
- The implementation and object data should be hidden from the outside world
- Only public method signatures are outward-facing and are accessible from outside. This is called object interface

Objects: summary

- We can model real world objects by **abstracting** selected properties and actions of these objects, ignoring details.
- The **Object-oriented program** is a system of collaborating objects. They collaborate by sending messages (calling each other’s methods).
- The outside objects should not know how object A does its thing or stores its data. Object A **encapsulates** its methods, and exposes only method signatures – **interface**.

Static Variables

- Variables can either be “attached” to the class or to instances of the class (objects).
- Static variables **are not** associated with any one object’s state. They are usually properties or definitions.
- Non-static variables are called instance variables because they are tied to exactly one instance of an object. They can be accessed with the keyword ‘this’.

Static or No Static?

- When deciding if variable should be static:

Ask yourself: Is it possible that the value of this variable will vary across different objects?

- Consider:

`Rectangle` **class** :

`numSides;` static (all rectangles have 4 sides)

`height;` not static (rectangles can have different dimensions)

Static Methods

- Methods also can either be “attached” to the class or to instances of the class.
- Static methods **do not** depend on the state of the object.
- They can be answered without anything that could reference the keyword “this”. Called using the class name.
- Non-static methods rely on an object’s state, often depending on the values of instance variables. Called on an instance.

Static or No Static?

- To decide if your method should be static:

Ask yourself: Does this method depend on the state of the object, or is it always the same regardless?

- Consider a `Rectangle` class:

```
getArea();
```

not static (depends on a particular rectangle's dims)

```
calculateArea(int h, int w);
```

static (formula; all info provided as inputs)