Chapter 5

Methods & Classes

Part 2: Classes

Adapted from notes by Pat Baker based on Java Foundations by Lewis, Chase, & DePasquale

Chapter Topics

- Working with methods
- Identifying classes and objects
- Structure and content of classes
- Instance data
- Visibility modifiers
- Constructors
- Relationships among classes
- Static methods and data

Classes & Objects

The programs written in previous examples have used classes already defined in the Java API. All of our code has been located inside the **`main()`** method.

Scanner Class:

```
Scanner scan = new Scanner(System.in);      //scan is the object of type Scanner
System.out.print("Enter your age:");
int age = scan.nextInt();     //scan object invokes nextInt() method defined in Scanner class
```

Random Class:

```
Random gen = new Random();        //gen is the object of type Random
int price = gen.nextInt(11);      //gen object invokes nextInt(n) method defined in Random class
```

Java Programs

- A **program** is made up of one or more classes
- A **class** contains one or more methods
- A **method** contains one or more statements that performs a specific task

- A Java **application** contains a method called main
  - So far, this is likely the only method you have worked with

Now it is time to:

- Define **your own** classes
- Create objects of that class type

What is a Class?

- A class is a blueprint from which individual objects are created
- Defines the **state** and **behavior** of an object

- All classes will *define*:
  - **state** (data/attributes/fields associated with a given object)
    - instance variables
  - **behavior** (methods/operations - what can objects do)
    - Methods
    - May change the state

What is an Object?

- An object is a specific **instance** of a class
- A bundle of related **state** and **behavior**

- All objects have:
  - **state** (data/attributes/fields associated with a given object)
    - instance variables
  - **behavior** (methods/operations - what can objects do)
    - Methods
    - May change the state

Example Class: `Dog`

Class `Dog` - defines a blueprint for all dog objects

- **state** (attributes/fields) - data associated with a given dog object
  - `name`
  - `color`
  - `breed`
- **behavior** (operations/methods - what can all dogs do)
  - `bark`
  - `wagTail`

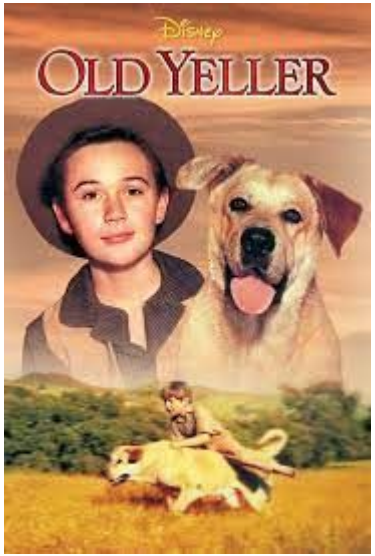
Example `Dog` Object #1

- Instance of class **`Dog`** - represents a specific dog object
- **state** (attributes/fields) - data associated with this **`Dog`** object
  - **`name`** - Beethoven
  - **`color`** - brown/white
  - **`breed`** - St. Bernard
- **behavior** (operations/methods)
  - **`bark`**
  - **`wagTail`**

Example `Dog` Object #2

- Instance of class **`Dog`** - represents a specific dog object
- **state** (attributes/fields) - data associated with this **`Dog`** object
  - **`name`** - Old Yeller
  - **`color`** - gold
  - **`breed`** - Labrador
- **behavior** (operations/methods)
  - **`bark`**
  - **`wagTail`**

Note that the *state* is different, but the *behaviors* are the same!



Example Class: `Car`

Class `Car` - defines a blueprint for all car objects

- **state** (attributes/fields) - data associated with a given car object
  - `make`
  - `model`
  - `year`
  - `speed`
- **behavior** (operations/methods) - what can all cars do
  - `applyBrake`
  - `accelerate`

Example `Car` Object #1

- Instance of class **`Car`** - represents a specific car object
- **state** (attributes/fields) - data associated with this **`Car`** object
  - **`make`** - Chevrolet
  - **`model`** - Corvette Sting Ray
  - **`year`** - 1963
  - **`speed`** - 0 (initially not moving)
- **behavior** (operations/methods)
  - **`applyBrake`**
  - **`accelerate`**

Example `Car` Object #2

- Instance of class **`Car`** - represents a specific car object
- **state** (attributes/fields) - data associated with this **`Car`** object
  - **`make`** - McLaren
  - **`model`** - Senna
  - **`year`** - 2018
  - **`speed`** - 0 (initially not moving)
- **behavior** (operations/methods)
  - **`applyBrake`**
  - **`accelerate`**

Object-Oriented Programming

- Java is an **Object-Oriented Programming** language
  - Earlier languages were **procedural** - data and methods were separate
- The fundamental entity is the "**object**"
  - An object has some information & some operations
  - Represents some real-world entity
    - A particular employee in a company
    - A window in a GUI
    - A character in a game
  - Handles its own processing and data management

Procedural Program Example

Review the `SphereCalculations` code from the previous slide. Could we implement `Sphere` as a class instead?

Consider the following:

- What data (state) do you need for a sphere?
- What operations (behavior) do you perform with that data?
    - Should any of the behavior change the state of the sphere?

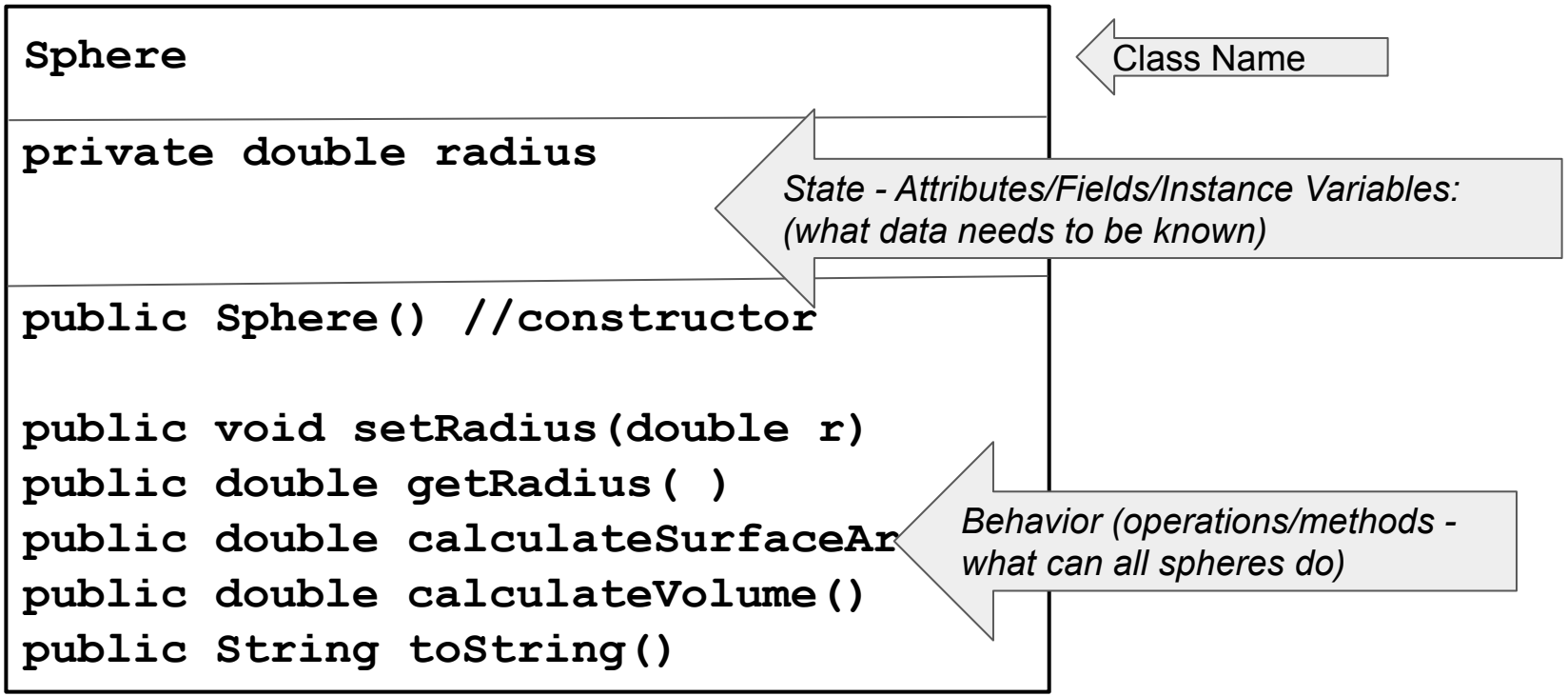
Convert to Object-Oriented Programming

- Class Name: **`Sphere`**

- State - Attributes/Fields:
  - What data needs to be known about the **`Sphere`**?

- Behavior - Operations/Methods:
  - What do **`Sphere`** objects need to do ?

Convert to Object-Oriented Programming



Object-Oriented Programming

- Usually multiple ways to represent a problem
- Usually no “one right answer”

- OOP itself is not always the answer

More about Objects

- Think of calling a method as “sending a message” that asks the object to do something
- The message contains the operations name and arguments
- The client doesn’t care how the message is handled, only that it produces an expected result

Objects vs Classes

- A class represents a abstract concept

- An object is the realization of a class
  - We instantiate an object of a specific class using new
  - An object is an instance of a particular class

- There can be multiple objects of a given class, but each object is an instantiation of a single class

The Math & Random Classes

- `java.util` package includes a `Random` class that generates pseudorandom numbers
- `java.lang` includes a `Math` class with various math functions

- You should familiarize yourself with both classes

Wrapper Classes

- Each primitive type has a corresponding wrapper class
- `int` → `Integer`
- `char` → `Character`

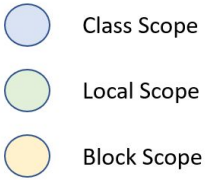
- Wrapper classes also static methods that help manage objects of that type
- Autoboxing automatically converts a primitive to a corresponding wrapper object

Data Scope

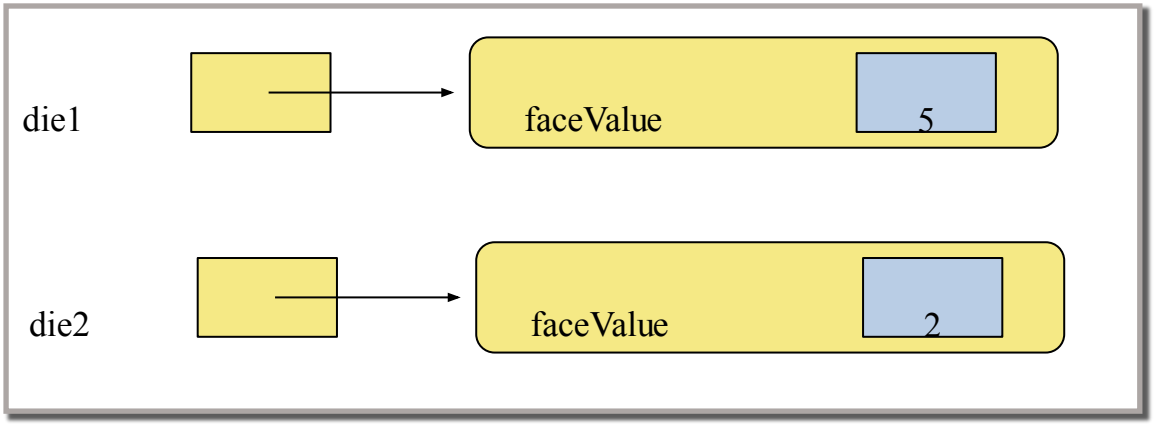
- The scope of data is the part of the program in which it can be referenced
- Data members of a class can be referenced by all methods in that class
- Local variables in a method can only be referenced in that method
- Data declared in a code block can only be referenced in that block

Data Scope

```
public class ScopeExample {
    int classScope;

    public ScopeExample(int classScope) {
        this.classScope = classScope;
    }

    public void myMethod() {
        int localScope = 0;

        while (localScope < 10) {
            int blockScope = 1;
            System.out.println(classScope + " : " + localScope + " : " + blockScope);
            localScope++;
            blockScope *= localScope;
        }
    }
}
```



Instance Data

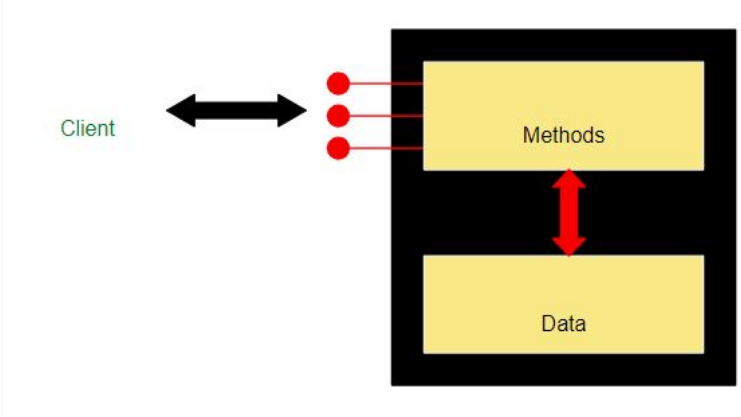


Each object maintains its own `faceValue` variable (and thus its own state)

Encapsulation

- When we define a class we need to be concerned with the details of variables and methods
- Encapsulation allows the object to protect its data
- The users of a class only need to know the services that the object provides
- Therefore, specific details should be encapsulated in the class
- The services the class provide describe its interface/API

Encapsulation



- The object can be thought of a black box
- The client only cares about the available operations, not the specifics of how the data is stored internally
- The client invokes the interface methods of the object
- The object manages its own data

Accessors & Mutators

- A class should not allow public access to its data

- Getter/setter methods can be provided to allow clients to changed the data under the supervision of the object

- Your class should not necessarily provide getter/setters for each of their attributes.

Encapsulation

- A better way to think of encapsulation is that you should strive to encapsulate behavior, not just data
- For example:
  - `setAccount(1000000)` vs `deposit(1000000)`
  - `setDecor()` vs `decorate()`
- The user of the class shouldn't make decision about how the state gets changed!
- Instead of setters that simply set the value of an attribute, think of behaviors that can be provided as methods

Visibility Modifiers

- `public`
- `private`
- `protected`
- `final`

Constructors

- Each class can have a constructor that sets up objects of that class
- The constructor is a method that has the same name as the class
- The constructor can be overridden

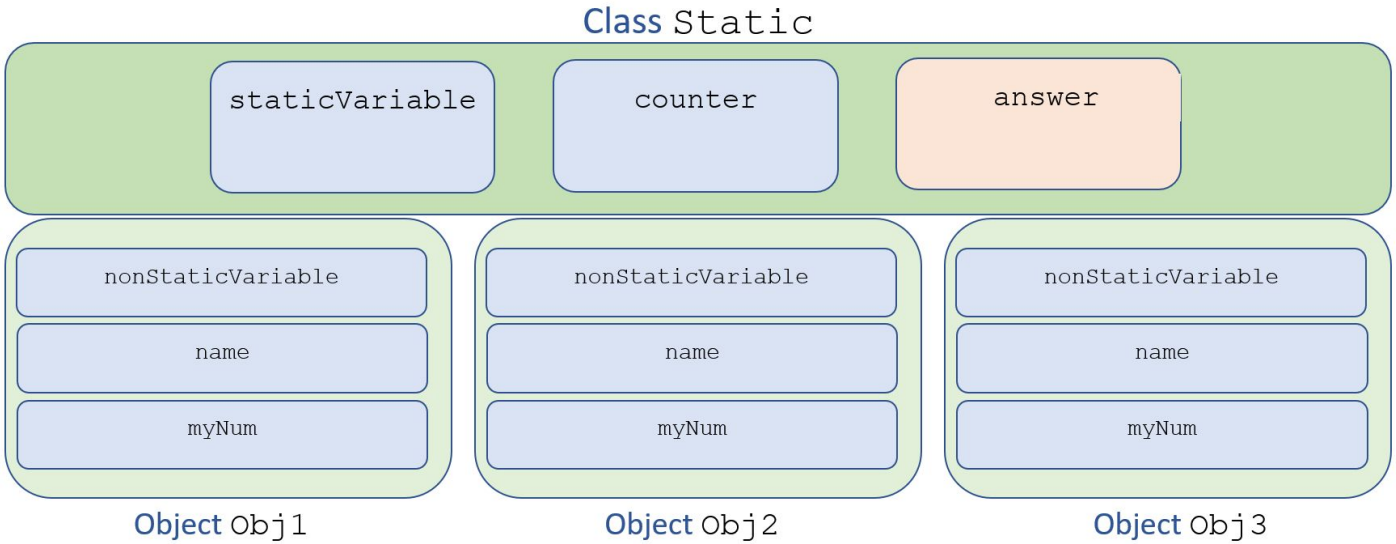
Static Class Members

- Static members are associated with the class as a whole
- Non-static members are associated with individual objects

- Static class variables have one value shared among all objects of that class
- Static class methods can be called without instantiating an object
  - Can only access static data members
- Deciding which members to make static is an important design decision

Static Class Members

```
private static int staticVariable
private int nonStaticVariable;
private String name;
private static final int answer = 42;

private int myNum;
private static int counter = 0;
```



Class Relationships

One of the powerful features of OOP is that objects can use other objects in different way. Suppose you are writing a class called `MyClass` that can take advantage features of another class called `OtherClass`

- Dependency: `MyClass` uses `OtherClass`
- Aggregation/Composition: `MyClass` has-a `OtherClass`
- Inheritance: `MyClass` is-a `OtherClass`

The this reference

- The `this` reference allows an object to refer to itself

- Can be useful to make methods more clear, especially when another object of the same type is involved

Designing a Class

- Consider a six-sided die
  - What is its state?
  - What is its behavior?

- Strive for reusability
- Not all programs that use the class will use all behavior/state of the class

Now go write some code!