Object Oriented Programming

Creating Classes with State and Behavior

Alwin Tareen

Object Oriented Programming

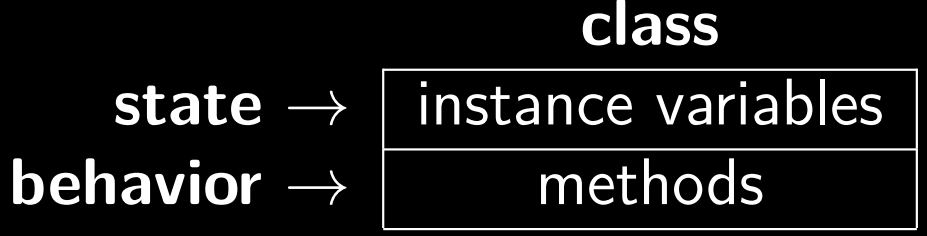
What is a class?

- As software systems become more complex, programmers look for better ways to develop software.
- One particular way is to divide a programming problem into discrete **classes** where each class has a specific task to perform, in solving the problem.
- A **class** is a description, model, or blueprint from which an object is created.

Object Oriented Programming

A class describes 2 characteristics of an object:

- It describes what data an object stores, known as an object's attributes. These are defined through the instance variables.
- It describes what an object does, known as an object's behavior. This is defined through the methods.



- The process of combining state and behavior into a single class is called **encapsulation**.

Object Oriented Programming

- Suppose you wanted to write a program that simulated the rolling of a single die.

- You could make a class called `Die` that would define the behavior for an object that represented a single six-sided die.
- Then, you could make another class called `DieTest` that created an object from the `Die` class, and simulated the rolling of a die.

Object Oriented Programming

```
public class Die
{
    private int faceValue;

    public Die()
    {
        faceValue = 1;
    }

    public void roll()
    {
        faceValue = (int) (Math.random() * 6) + 1;
    }

    public int getFaceValue()
    {
        return faceValue;
    }
}
```

Object Oriented Programming

```
public class DieTest
{
    public static void main(String[] args)
    {
        Die cube = new Die();
        cube.roll();
        System.out.println(cube.getFaceValue());
    }
}
```

Instance Variables

- These are variables that describe the state of an object, also known as its attributes.
- They are **always** declared `private`.
- You can use them in any method in the class.
- Don't initialize them, because they are always automatically assigned default values.

`int` $\rightarrow$ 0
`double` $\rightarrow$ 0.0
`boolean` $\rightarrow$ `false`
object references $\rightarrow$ `null`

Constructors

- A constructor is a special method within a class, that has the same name as the class.
- The primary purpose of a constructor is to assign initial values to the class' instance variables.
- When defining a constructor, you must not specify a return type.

```
public class Person
{
    private String name;

    public Person()
    {
        name = "";
    }
}
```

Constructors

- A constructor with no parameters is called the **default constructor**.
- A class can have more than one constructor. Providing multiple constructors makes a class more flexible and easy to use.
- When using multiple constructors, the parameter list of each constructor within a class must be unique.
- Parameter lists must differ by either the number of parameters defined, or by the parameter type.

Constructors

- The following constructors differ in the **number** of parameters.

```
public Person()
public Person(String n)

public Student(String n)
public Student(String n, int age)
```

- The following constructors differ in the **type** of parameters.

```
public Area(int length, int width)
public Area(double length, double width)
```

Constructors

- Constructors are invoked or called when you **construct** an object using the keyword `new`.
- The follwing code instantiates two `Student` objects.
- The first statement uses the `Student` class' default constructors.
- The second statement uses the `Student` class' constructor that takes 2 parameters.

```
Student alice = new Student();
Student bob = new Student("Bob", 17);
```

- Note that if a class contains **no** constructors, then `Java` will **automatically** provide a default constructor for the class.

Code Example: The Dog Class

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

    public Dog()
    {
        size = 0;
        name = "";
    }

    public Dog(int dogSize, String dogName)
    {
        size = dogSize;
        name = dogName;
    }
```

Code Example: The Dog Class, Continued

```
    public int getSize()
    {
        return size;
    }

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

Accessor Methods

- Instance variables must be declared `private`, as demonstrated in the following `Student` class:

```
public class Student
{
    // instance variables
    private String name;
    private int age;
```

Accessor Methods

- By declaring the instance variables as `private`, client programs that create objects from the class are not allowed to access the instance variables directly, using the dot operator:

```
public class StudentTest
{
    public static void main(String[] args)
    {
        Student pupil = new Student();
        pupil.name = "George" // ERROR
    }
}
```

Accessor Methods

- However, client programs often need the ability to see the contents of the instance variables of an object.
- For this reason, classes are often designed with a special type of method called an **accessor method**.
- Methods defined in a class which allow clients to observe instance variables(but not modify them) are called accessor methods.
- Remember, client programs do not have direct access to these instance variables, because they are declared `private`.
- The only way that client programs can view the values of the instance variables, is if there are accessor methods that provide them with this information.

Code Example: The Student Class

```
public class Student
{
    private String name;
    private int age;

    public Student()
    {
        name = "";
        age = 0;
    }

    public String getName()
    {
        return name;
    }

    public int getAge()
    {
        return age;
    }
}
```

Accessor Methods

- The purpose of an accessor method is to allow a client program to see the value of an instance variable.
- For example, the `getName()` accessor method from the `Student` class allows clients to see the contents of the `name` instance variable.
- Accessor methods are declared with a return type that corresponds to the data type of the instance variable being accessed.

```
private String name;

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

Accessor Methods

- Note: A common practice is to define accessor methods with the word **get** in front of their name, followed by the name of the instance variable they are accessing.
- For example, `getName()`, `getLength()`, `getWidth()`, `getScore()`, `getTemperature()`, etc.

```
Student pupil = new Student("Bob", 17);

System.out.println(pupil.getName());
System.out.println(pupil.getAge());
```

Mutator Methods

- Methods in a class that allow clients to modify an object's instance variables are called **mutator methods**.
- If the instance variables of a class are declared `private`, then clients who instantiate objects of this class do not have direct access to its instance variables.
- If you wish for clients to have the ability to change the value of a particular instance variable, then you must provide a mutator method for that variable in the class implementation.

Mutator Methods

- Consider the following mutator method that is defined for the `Student` class:

```
public void setName(String n)
{
    name = n;
}
```

- This method, when called, will change the value of the `name` instance variable to the value specified by the parameter `n`.
- This method allows clients to **mutate** or change the contents of the variable `name`.

Mutator Methods

- If you don't want a client to have the ability to modify a particular instance variable, then don't provide a mutator method for that variable.
- Mutator methods are defined with a return type of `void`, since they do not return a value.
- Note: A common practice is to define mutator methods with the word `set` in front of their name, followed by the name of the instance variable they are modifying.
- For example, `setLength()`, `setWidth()`, `setScore()`, `setName()`, `setTemperature()`, etc.

```
Student pupil = new Student();
pupil.setName("Alice");
pupil.setAge(17);
```

Code Example: The Cat Class

```
public class Cat
{
    private String name;
    private int size;

    public Cat()
    {
        name = "";
        size = 0;
    }

    public Cat(String n, int s)
    {
        name = n;
        size = s;
    }
```

Code Example: The Cat Class, Continued

```
    // accessor methods
    public String getName()
    {
        return name;
    }

    public int getSize()
    {
        return size;
    }
```

Code Example: The Cat Class, Continued

```
    // mutator methods
    public void setName(String n)
    {
        name = n;
    }

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

The toString() Method

- The purpose of the `toString()` method is to provide client programs with an easy way to print the contents of the instance variables of a class.
- It can also be used to print other information within an object, such as the results of method calls.
- The `toString()` method of an object is activated by enclosing the object name within a `println()` statement:

```
Student pupil = new Student("Bob", 17);
System.out.println(pupil);
```

The toString() Method

- Any class can include a `toString()` method in its implementation. The method must use the following format:

```
public String toString()
{
    ...
}
```

- Within the body of `toString()`, a `String` is defined and returned to the `println()` method of the client program.
- The `String` is often built using a series of concatenation operators, so the `String` can include more than one variable.

The toString() Method

- Labels are often included within the `String` to make the output easily readable by the user.
- The escape sequence `\n` is also used to embed newline characters within the `String`, so the output can be displayed on multiple lines.

```
public String toString()
{
    String result = "";
    result += "Name: " + name + "\n";
    result += "Age: " + age;
    return result;
}
```

The End