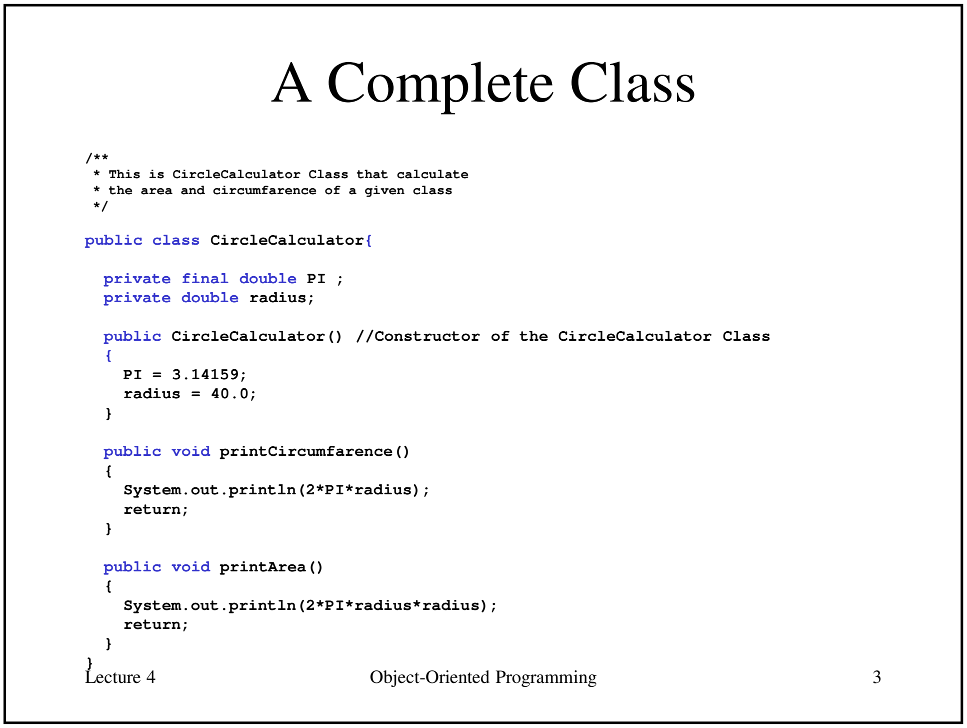
Making Classes and Objects

Lecture 4

Object-Oriented Programming

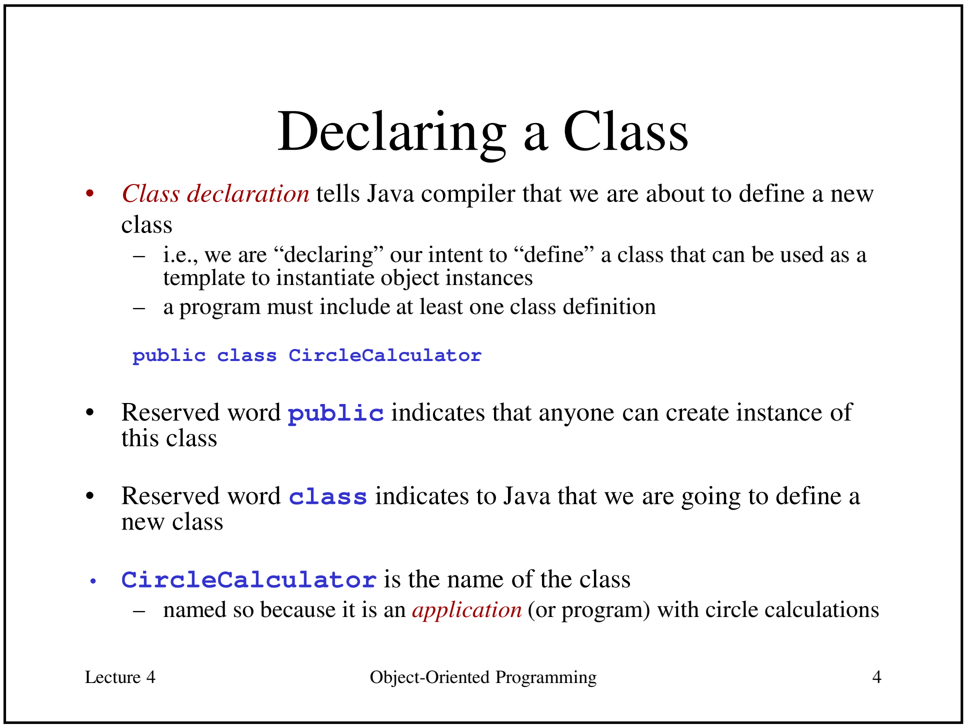
Agenda

- A Complete Class example
- Declaring a Class
- Defining a Class
- Declaring Variables
- Constructors
- Declaring Constructors
- Comments
- Class with Other Classes as Objects example
- Syntax Explanation
- Object Relationships
- Representing Relationship in UML
- Readings



A Complete Class

```
/**
 * This is CircleCalculator Class that calculate
 * the area and circumfarence of a given class
 */

public class CircleCalculator{

  private final double PI ;
  private double radius;

  public CircleCalculator() //Constructor of the CircleCalculator Class
  {
    PI = 3.14159;
    radius = 40.0;
  }

  public void printCircumfarence()
  {
    System.out.println(2*PI*radius);
    return;
  }

  public void printArea()
  {
    System.out.println(2*PI*radius*radius);
    return;
  }
}
```

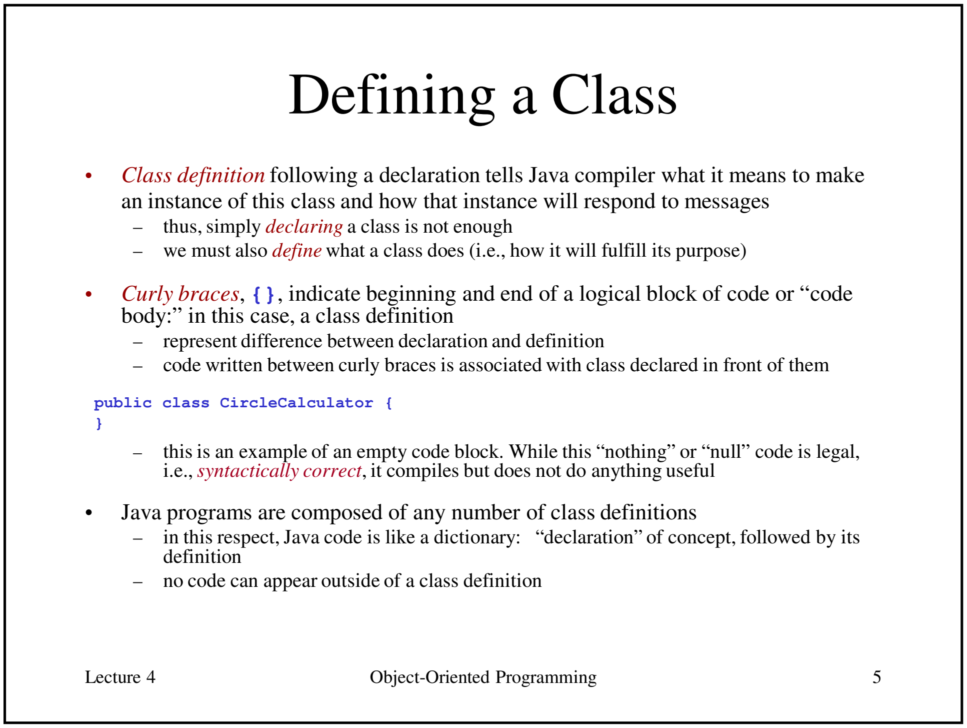


Declaring a Class

- *Class declaration* tells Java compiler that we are about to define a new class
  - i.e., we are “declaring” our intent to “define” a class that can be used as a template to instantiate object instances
  - a program must include at least one class definition

  ```
  public class CircleCalculator
  ```

- Reserved word `public` indicates that anyone can create instance of this class
- Reserved word `class` indicates to Java that we are going to define a new class
- `CircleCalculator` is the name of the class
  - named so because it is an *application* (or program) with circle calculations

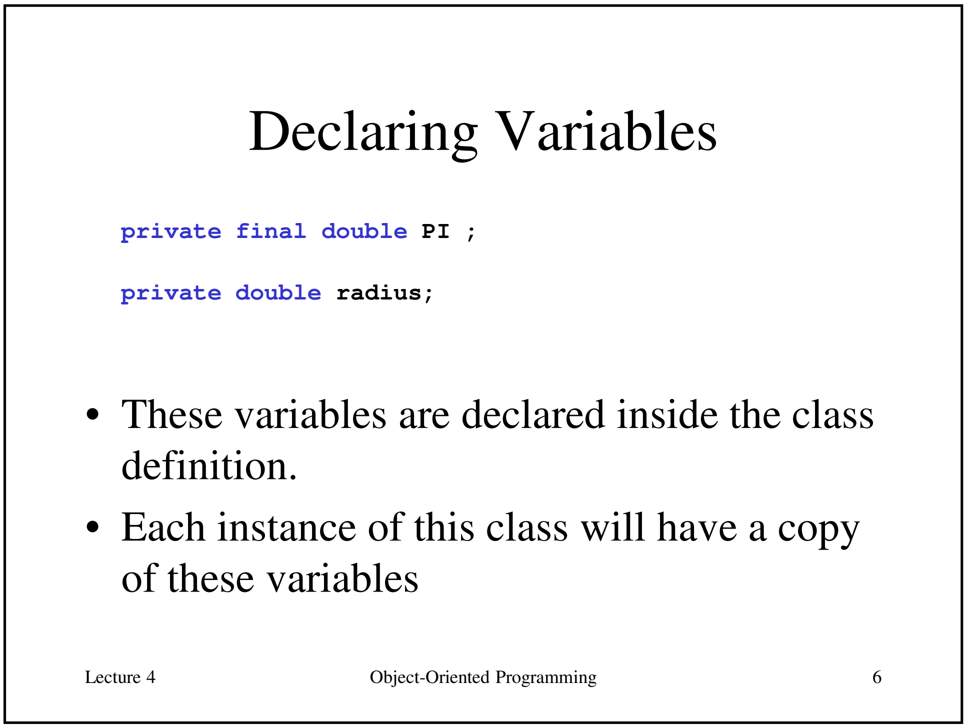


Defining a Class

- *Class definition* following a declaration tells Java compiler what it means to make an instance of this class and how that instance will respond to messages
  - thus, simply *declaring* a class is not enough
  - we must also *define* what a class does (i.e., how it will fulfill its purpose)
- *Curly braces*, **{ }**, indicate beginning and end of a logical block of code or "code body:" in this case, a class definition
  - represent difference between declaration and definition
  - code written between curly braces is associated with class declared in front of them

```
public class CircleCalculator {
}
```

  - this is an example of an empty code block. While this "nothing" or "null" code is legal, i.e., *syntactically correct*, it compiles but does not do anything useful
- Java programs are composed of any number of class definitions
  - in this respect, Java code is like a dictionary: "declaration" of concept, followed by its definition
  - no code can appear outside of a class definition



Declaring Variables

```
private final double PI ;

private double radius;
```

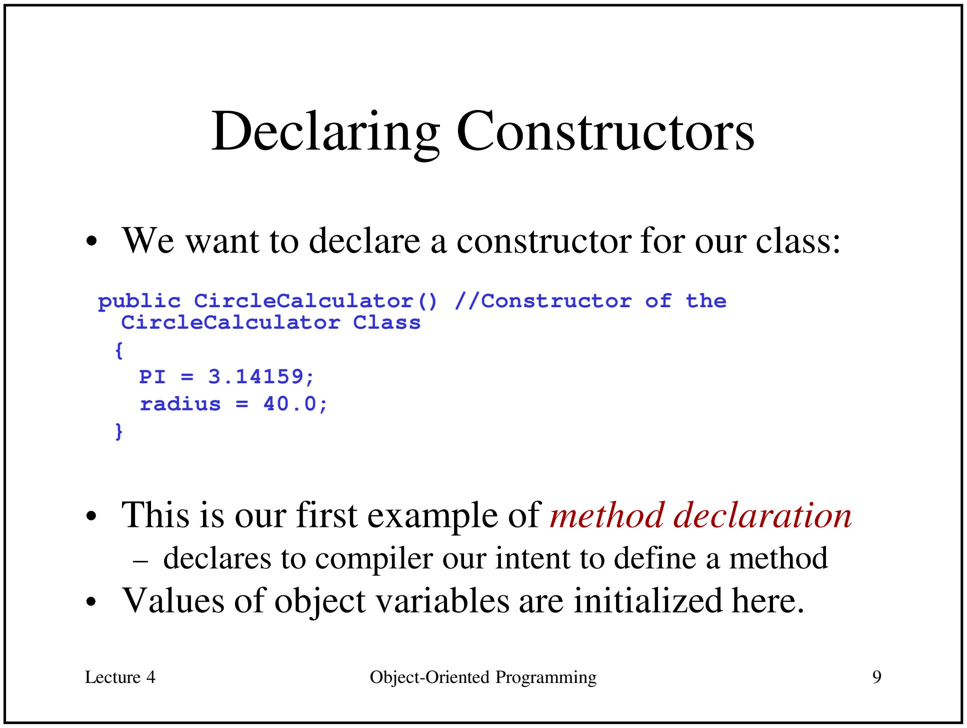
- These variables are declared inside the class definition.
- Each instance of this class will have a copy of these variables

Constructors

- Now we need to have instances of our class to do something useful
- *Constructor* is a special method that is called whenever a class is instantiated (created)
  - another object sends a message that calls a constructor
  - A constructor is the first message an object receives and cannot be called subsequently
  - *establishes initial state of properties* for instance

Constructor (cont’d)

- If you do not define any constructors for class, Java writes one for you
  - called *default* constructor
  - default constructor will initialize each instance variable to its default value
  - This is not a good idea
    - ALWAYS write your own constructor for each class
    - ALWAYS give each instance variable an initial value

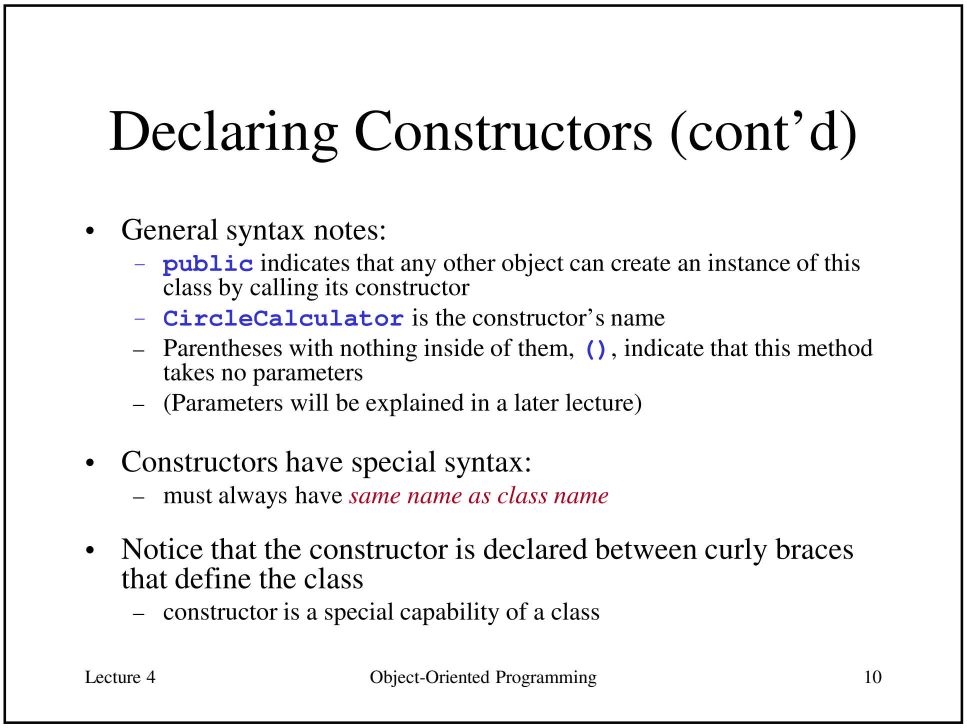


Declaring Constructors

- We want to declare a constructor for our class:

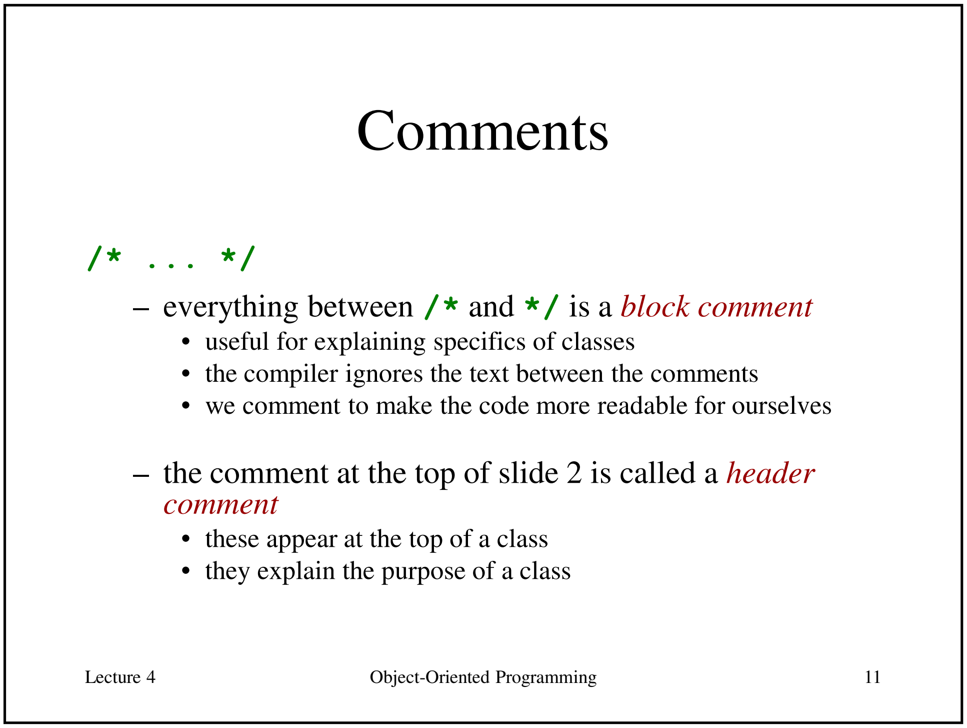
```
public CircleCalculator() //Constructor of the
  CircleCalculator Class
 {
   PI = 3.14159;
   radius = 40.0;
 }
```

- This is our first example of *method declaration*
  - declares to compiler our intent to define a method
- Values of object variables are initialized here.



Declaring Constructors (cont’d)

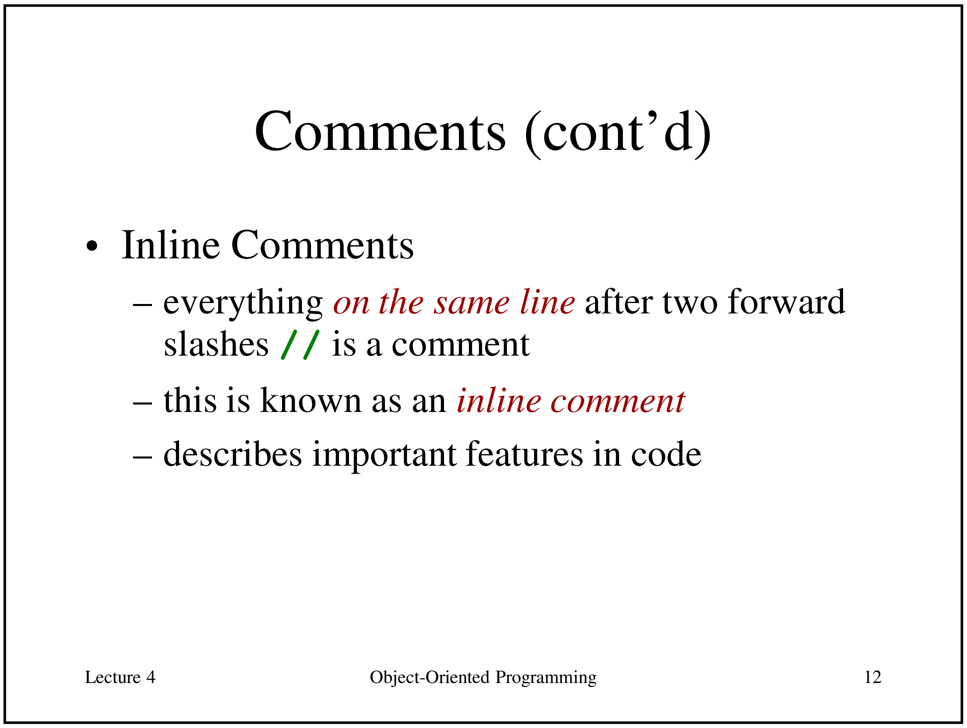
- General syntax notes:
  - **`public`** indicates that any other object can create an instance of this class by calling its constructor
  - **`CircleCalculator`** is the constructor’s name
  - Parentheses with nothing inside of them, **`()`**, indicate that this method takes no parameters
  - (Parameters will be explained in a later lecture)
- Constructors have special syntax:
  - must always have *same name as class name*
- Notice that the constructor is declared between curly braces that define the class
  - constructor is a special capability of a class



Comments

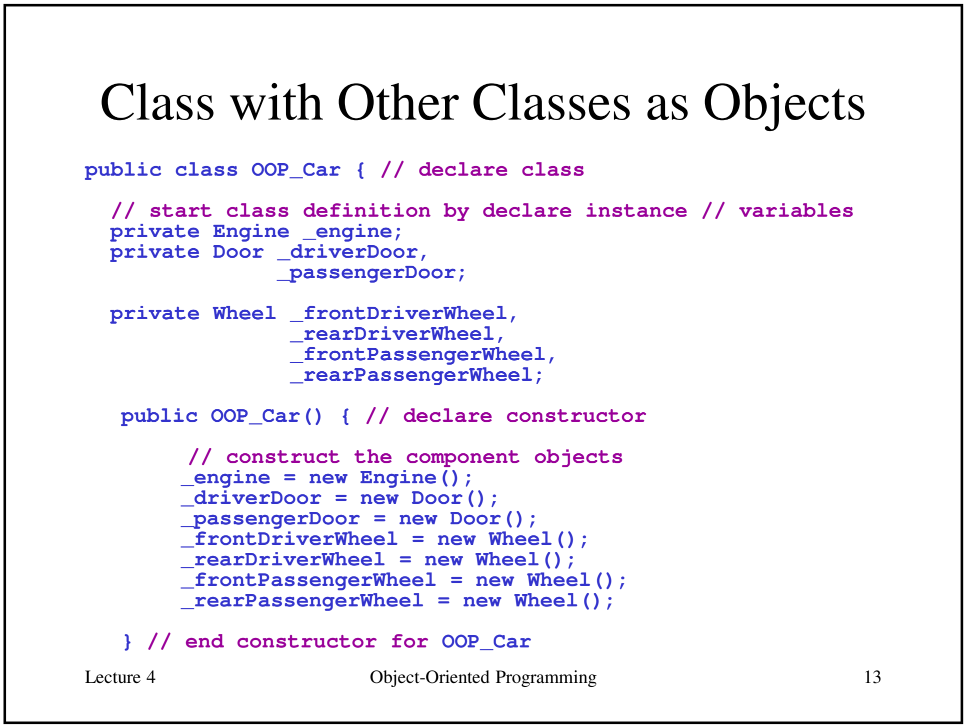
`/* ... */`

- everything between `/*` and `*/` is a *block comment*
  - useful for explaining specifics of classes
  - the compiler ignores the text between the comments
  - we comment to make the code more readable for ourselves
- the comment at the top of slide 2 is called a *header comment*
  - these appear at the top of a class
  - they explain the purpose of a class



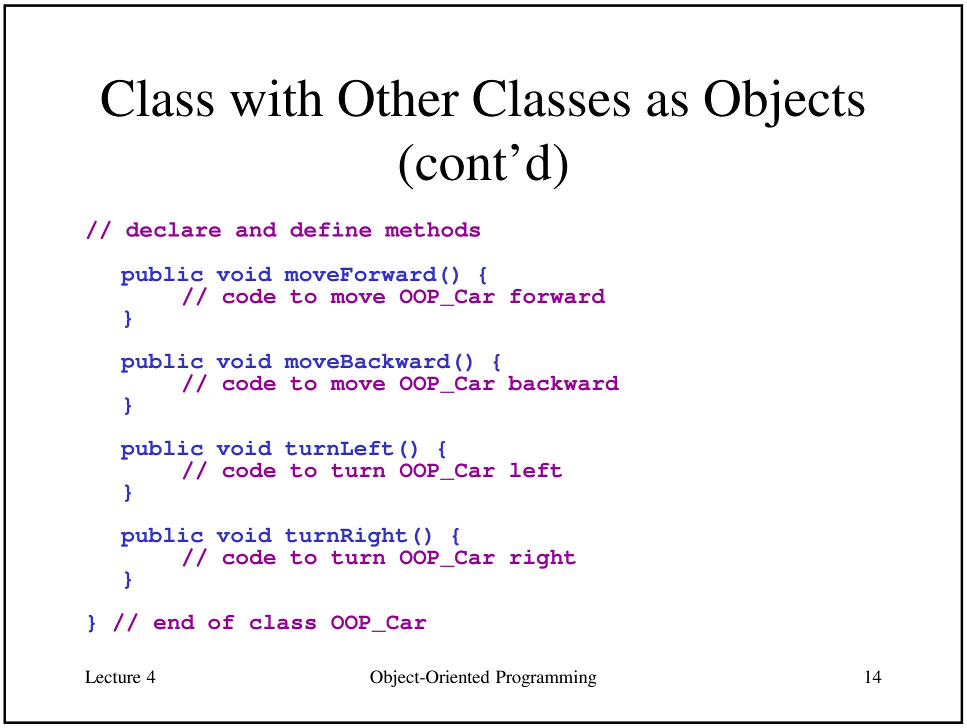
Comments (cont’d)

- Inline Comments
  - everything *on the same line* after two forward slashes `//` is a comment
  - this is known as an *inline comment*
  - describes important features in code



Class with Other Classes as Objects

```
public class OOP_Car { // declare class

  // start class definition by declare instance // variables
  private Engine _engine;
  private Door _driverDoor,
               _passengerDoor;

  private Wheel _frontDriverWheel,
                _rearDriverWheel,
                _frontPassengerWheel,
                _rearPassengerWheel;

   public OOP_Car() { // declare constructor

        // construct the component objects
        _engine = new Engine();
        _driverDoor = new Door();
        _passengerDoor = new Door();
        _frontDriverWheel = new Wheel();
        _rearDriverWheel = new Wheel();
        _frontPassengerWheel = new Wheel();
        _rearPassengerWheel = new Wheel();

   } // end constructor for OOP_Car
```



Class with Other Classes as Objects (cont'd)

```
// declare and define methods

   public void moveForward() {
        // code to move OOP_Car forward
   }

   public void moveBackward() {
        // code to move OOP_Car backward
   }

   public void turnLeft() {
        // code to turn OOP_Car left
   }

   public void turnRight() {
        // code to turn OOP_Car right
   }

} // end of class OOP_Car
```

Syntax Explanation

`private Engine _engine;`

- *declares* an instance variable named `_engine` of type `Engine`
- reserved word `private`
  - indicates that instance variable will be available only to methods within this class
  - other objects do not have access to `_engine`
  - thus, `OOP_Car` "encapsulates" its `_engine`
- remember, *properties are objects* themselves
  - every object must be an instance of some class
  - the class of an instance variable is called its *type* which determines what messages can be sent to this property
- name of instance variable is `_engine`

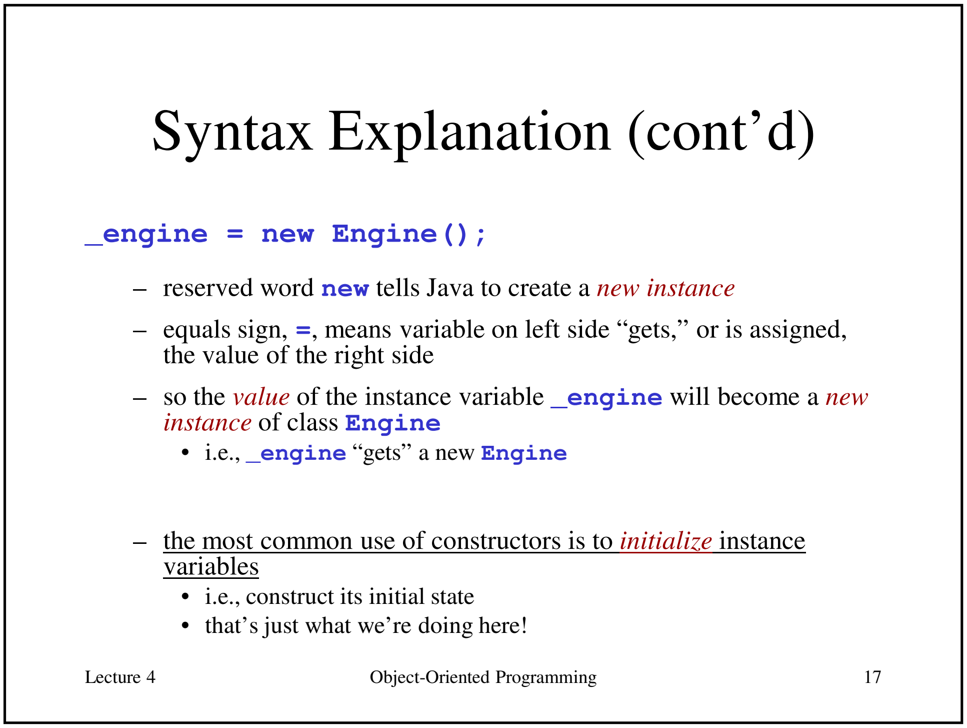
Syntax Explanation (cont'd)

```
private Door  _driverDoor,
                _passengerDoor;
```

- we can declare multiple instance variables of the same type by separating them with commas
- `_driverDoor` and `_passengerDoor` are both instance variables of type `Door`

```
public OOP_Car() {
```

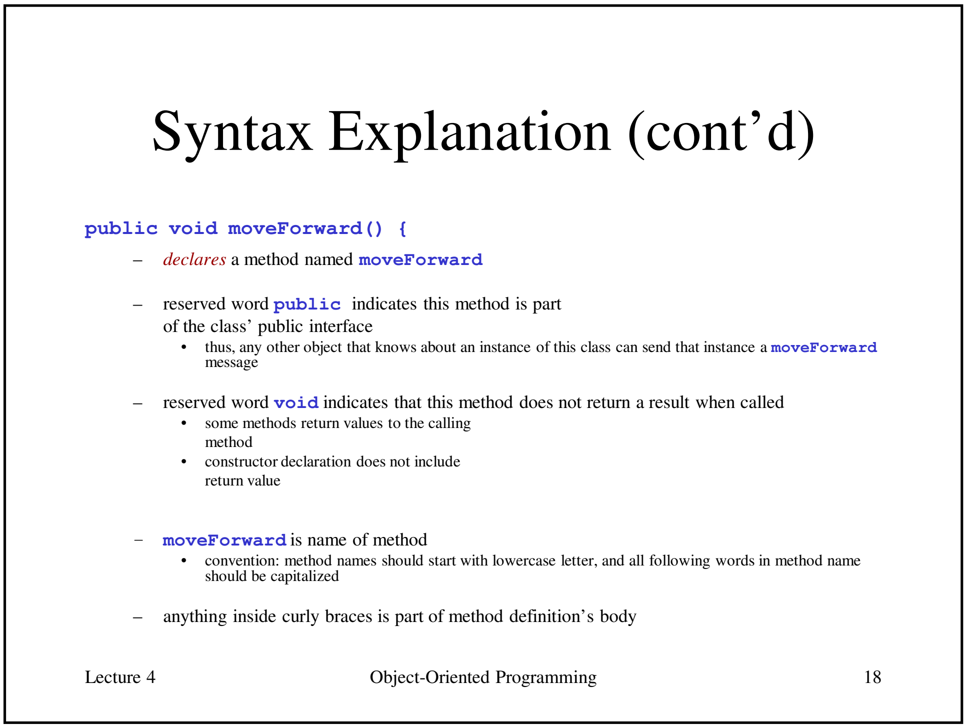
- *constructor* for class `OOP_Car`
- remember: constructor is the first message sent to a newly created object
- must have the same identifier (name) as its class
- `()` makes it a method



Syntax Explanation (cont'd)

`_engine = new Engine();`

- reserved word **`new`** tells Java to create a *new instance*
- equals sign, **`=`**, means variable on left side "gets," or is assigned, the value of the right side
- so the *value* of the instance variable **`_engine`** will become a *new instance* of class **`Engine`**
  - i.e., **`_engine`** "gets" a new **`Engine`**
- <u>the most common use of constructors is to *initialize* instance variables</u>
  - i.e., construct its initial state
  - that's just what we're doing here!



Syntax Explanation (cont'd)

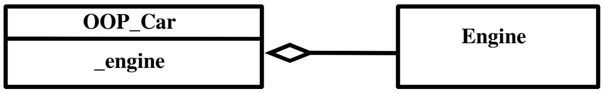
`public void moveForward() {`

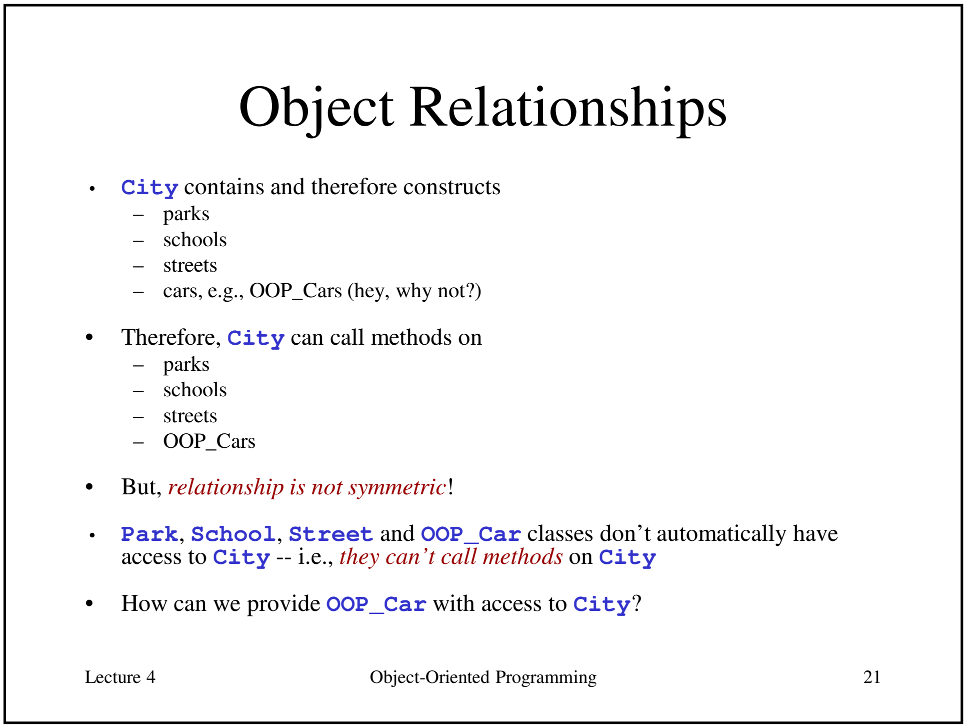
- *declares* a method named **`moveForward`**
- reserved word **`public`** indicates this method is part of the class' public interface
  - thus, any other object that knows about an instance of this class can send that instance a **`moveForward`** message
- reserved word **`void`** indicates that this method does not return a result when called
  - some methods return values to the calling method
  - constructor declaration does not include return value
- **`moveForward`** is name of method
  - convention: method names should start with lowercase letter, and all following words in method name should be capitalized
- anything inside curly braces is part of method definition's body

Object Relationships

- In our description, we said the `OOP_Car` had an engine, doors, and wheels; these are its *components*
- It can be said that the `OOP_Car` *is composed of* its engine, doors, and wheels
- *Containment* is when one class is a component of the other
- How do you determine containment?
  - class `OOP_Car` has an instance variable of type `Engine`
  - class `OOP_Car` *creates* an instance of type `Engine`
  - therefore, `OOP_Car` *contains* an `Engine`

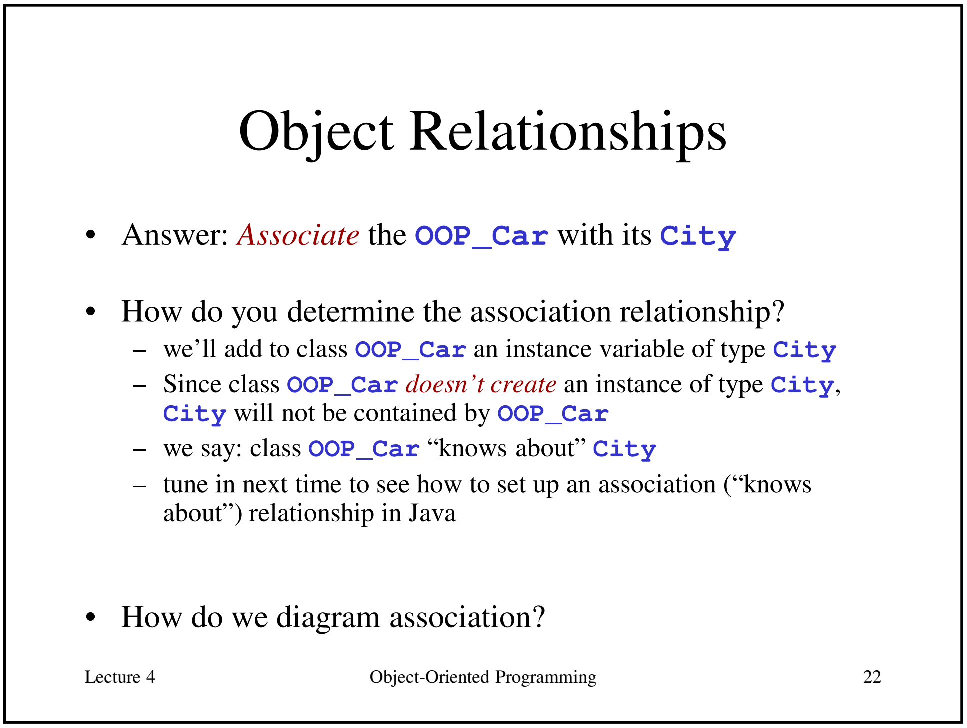
Representing Relationship in UML





Object Relationships

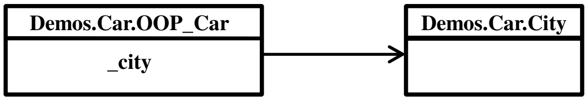
- `City` contains and therefore constructs
  - parks
  - schools
  - streets
  - cars, e.g., OOP_Cars (hey, why not?)
- Therefore, `City` can call methods on
  - parks
  - schools
  - streets
  - OOP_Cars
- But, *relationship is not symmetric*!
- `Park`, `School`, `Street` and `OOP_Car` classes don't automatically have access to `City` -- i.e., *they can't call methods* on `City`
- How can we provide `OOP_Car` with access to `City`?



Object Relationships

- Answer: *Associate* the `OOP_Car` with its `City`
- How do you determine the association relationship?
  - we'll add to class `OOP_Car` an instance variable of type `City`
  - Since class `OOP_Car` *doesn't create* an instance of type `City`, `City` will not be contained by `OOP_Car`
  - we say: class `OOP_Car` "knows about" `City`
  - tune in next time to see how to set up an association ("knows about") relationship in Java
- How do we diagram association?

Object Relationships



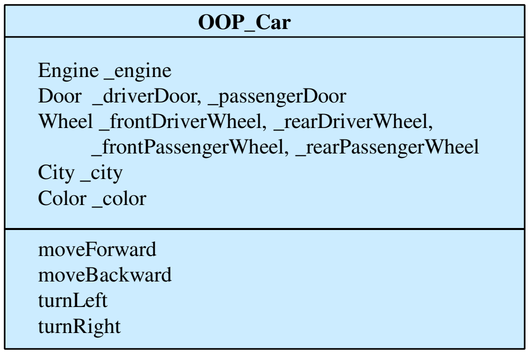
Object Relationships

- The OOP_Car has certain attributes
  - color, size, position, etc.
- Attributes are properties that *describe* the OOP_Car
  - we'll add to class OOP_Car an instance variable of type Color
  - OOP_Car ***is described by*** its Color
  - this is different than "***is composed of***" relationship
  - class OOP_Car ***doesn't contain*** its Color**,** ***nor is it associated*** with it
  - we say: Color is an ***attribute*** of class OOP_Car
  - class OOP_Car may set its own Color or another class may call a method on it to set its Color
  - the actual color of the OOP_Car is an attribute, but it is also an instance of the Color class
    - **all instance variables are instances!**

Representing Classes

- A rectangle is drawn to represent an individual class schematically
  - at the top is the class name
  - the next section lists the properties of the class (instance variable names are optional)
  - below the properties are listed the capabilities of the class
    - note that the constructor is assumed and is not listed under capabilities

A Class Representation



OOP_Car

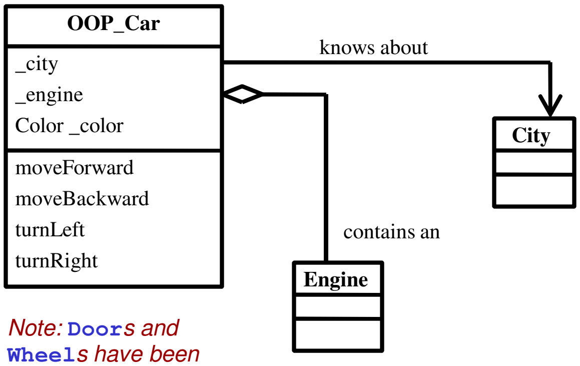
Engine _engine
Door _driverDoor, _passengerDoor
Wheel _frontDriverWheel, _rearDriverWheel,
_frontPassengerWheel, _rearPassengerWheel
City _city
Color _color

moveForward
moveBackward
turnLeft
turnRight

Class Diagram

- A *class diagram* shows how classes relate to other classes
  - rectangles represent classes
  - relationships between classes are shown with lines
  - important properties have their name
  - with reference to class boxes representing their type
  - attributes have type and identifier (but don't show references)

Class Diagram



OOP_Car

_city
_engine
Color _color

moveForward
moveBackward
turnLeft
turnRight

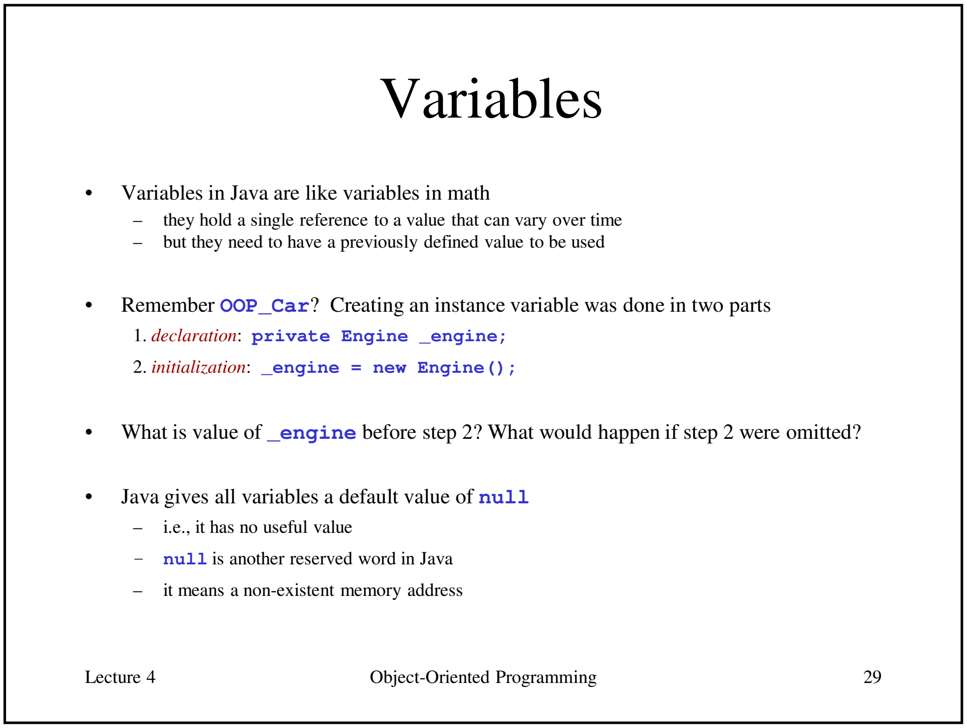
knows about

City

contains an

Engine

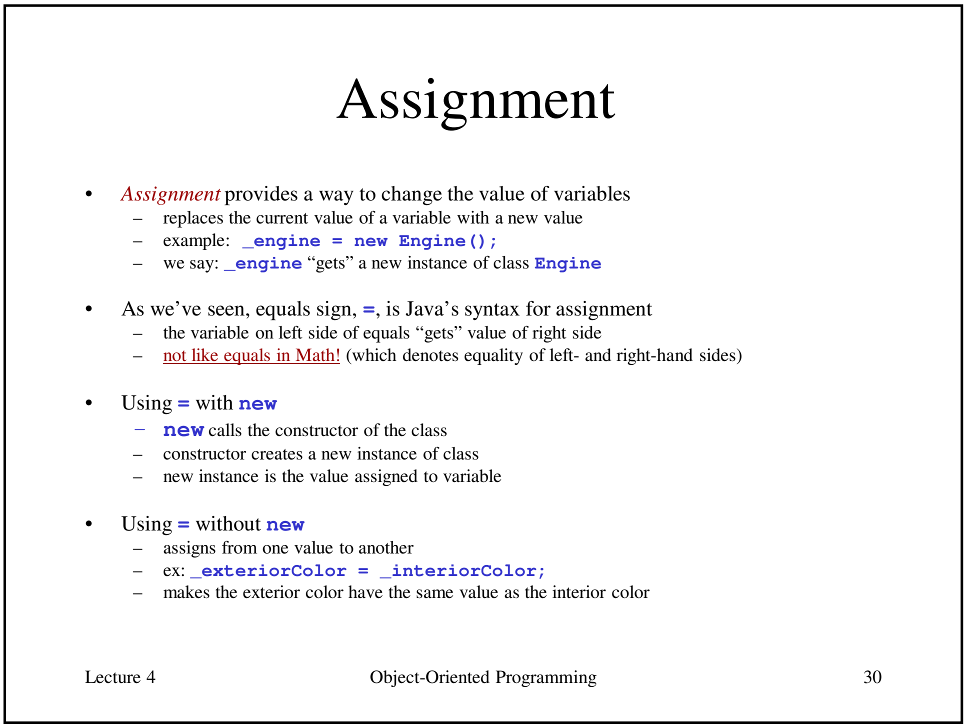
Note: `Doors` *and* `Wheels` *have been elided for clarity*



Variables

- Variables in Java are like variables in math
  – they hold a single reference to a value that can vary over time
  – but they need to have a previously defined value to be used
- Remember `OOP_Car`? Creating an instance variable was done in two parts
  1. *declaration*: `private Engine _engine;`
  2. *initialization*: `_engine = new Engine();`
- What is value of `_engine` before step 2? What would happen if step 2 were omitted?
- Java gives all variables a default value of `null`
  – i.e., it has no useful value
  – `null` is another reserved word in Java
  – it means a non-existent memory address

Lecture 4 Object-Oriented Programming 29



Assignment

- *Assignment* provides a way to change the value of variables
  – replaces the current value of a variable with a new value
  – example: `_engine = new Engine();`
  – we say: `_engine` "gets" a new instance of class `Engine`
- As we've seen, equals sign, `=`, is Java's syntax for assignment
  – the variable on left side of equals "gets" value of right side
  – not like equals in Math! (which denotes equality of left- and right-hand sides)
- Using `=` with `new`
  – `new` calls the constructor of the class
  – constructor creates a new instance of class
  – new instance is the value assigned to variable
- Using `=` without `new`
  – assigns from one value to another
  – ex: `_exteriorColor = _interiorColor;`
  – makes the exterior color have the same value as the interior color

Lecture 4 Object-Oriented Programming 30

Calling Methods

- We know how to declare methods, but how do we call them? How can we send messages between objects?
- Syntax is: `<variableIdentifier>.<methodIdentifier>();`

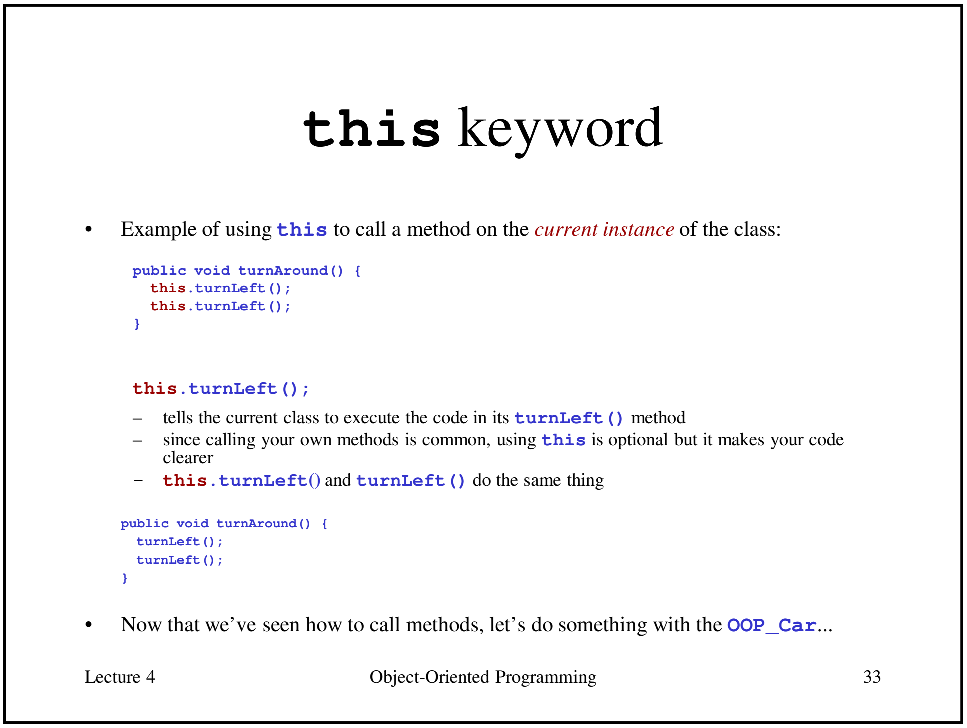
```
public class City {

  private OOP_Car _15mobile;

  public City() {
    _15mobile = new OOP_Car();
    _15mobile.moveForward();
  }
}
```

- Sending a message ("calling `moveForward` on `_15mobile`") causes the method's code to be executed

  `_15mobile.moveForward()` is a *method call*
  - `_15mobile` is the message's *receiver* (the instance being told to move)
  - dot ("`.`") separates receiver from method name
  - `moveForward` is the name of method to be sent
  - `()` denotes parameters to the message
  - more on parameters next lecture! woo hoo!

Calling Methods

- What if we want one method in a class to call another method in the same class?
  – let's say we want the `OOP_Car` to have a `turnAround()` method
  – we will want the `turnAround()` method to call the `OOP_Car`'s own `turnLeft()` or `turnRight()` method twice
- In order for the *current instance* to be a receiver of message, *we need a way to refer to it*
- Reserved word `this` is shorthand for "this instance"
  – `this` allows an instance to *send a message to itself*



this keyword

- Example of using **this** to call a method on the *current instance* of the class:

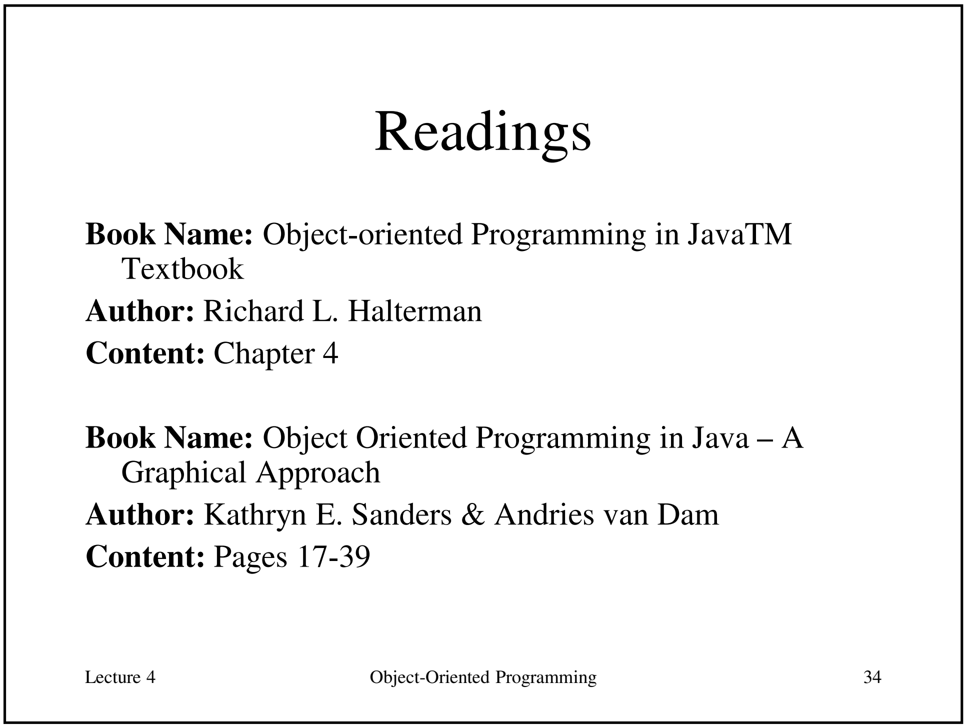
```
public void turnAround() {
  this.turnLeft();
  this.turnLeft();
}
```

this.turnLeft();

- tells the current class to execute the code in its **turnLeft()** method
- since calling your own methods is common, using **this** is optional but it makes your code clearer
- **this.turnLeft()** and **turnLeft()** do the same thing

```
public void turnAround() {
  turnLeft();
  turnLeft();
}
```

- Now that we’ve seen how to call methods, let’s do something with the **OOP_Car**...



Readings

Book Name: Object-oriented Programming in JavaTM Textbook

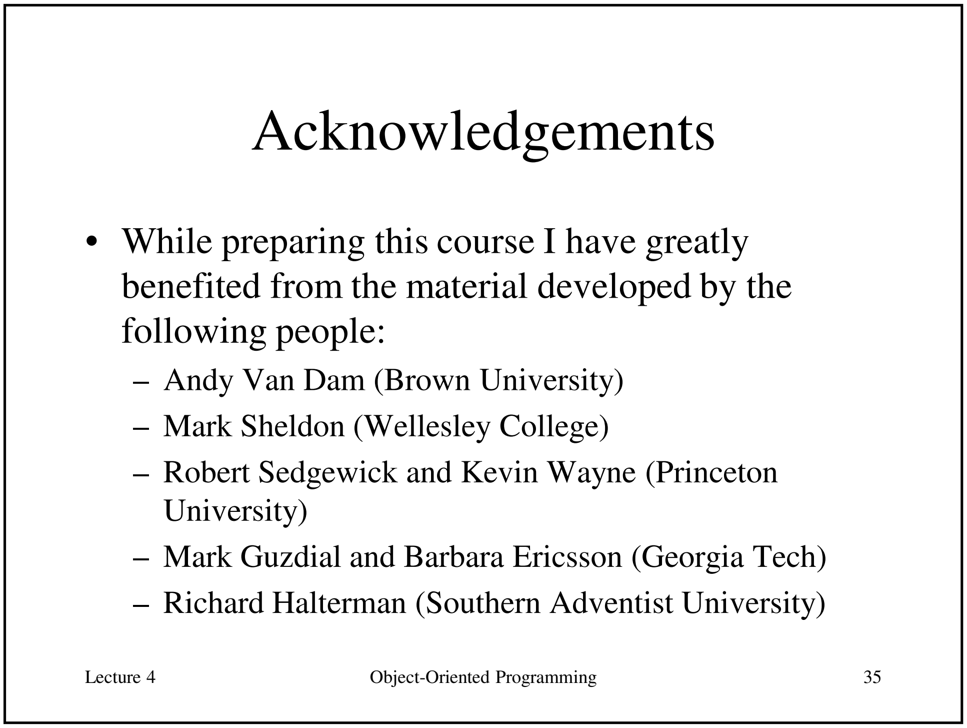
Author: Richard L. Halterman

Content: Chapter 4

Book Name: Object Oriented Programming in Java – A Graphical Approach

Author: Kathryn E. Sanders & Andries van Dam

Content: Pages 17-39



Acknowledgements

- While preparing this course I have greatly benefited from the material developed by the following people:
  - Andy Van Dam (Brown University)
  - Mark Sheldon (Wellesley College)
  - Robert Sedgewick and Kevin Wayne (Princeton University)
  - Mark Guzdial and Barbara Ericsson (Georgia Tech)
  - Richard Halterman (Southern Adventist University)