MAKING OBJECTS

- Views of a Class
- Defining Your Own Class
- Declaring Instance Variables
- Declaring Methods
- Sending Messages

Example: The Car

- We get a new car for the semester: the CSCI331Mobile!
- Being the object-oriented people, we think of the CSCI331Mobile as an object with properties and capabilities
- We will create a *class* to model the CSCI331Mobile and then make an *instance* of that class

Specifications of the CSCI331Mobile

- We come up with the following basic (ok, ridiculously simple) specification for the CSCI331Mobile:
  - it should have an engine and wheels to move
  - it should have doors so people can get in and out
  - it should be able to move forward and backward
  - it should be able to turn left and right

- What are the CSCI331Mobile's *properties*?
  - engine, wheels, doors

- What are the CSCI331Mobile's *capabilities*?
  - move forward, move backward, turn left, turn right
  - don't forget the constructor — all objects have the ability to construct themselves (when sent a message to do so by another object)

- What would this look like in Java?
  - remember, properties are represented by instance variables
  - capabilities are represented by methods

- Let's

Simple Syntax for `CSCI331Mobile`

Note: The point of this is to show an outline of what a generic class definition looks like. Some functionality has been elided with
`// comments`.

Three parts to class definition:

- declaration, list of properties, list of methods (capabilities)

```
package Demos.Car;

/**
 * This class models a vehicle that
 * can move and turn.
 */
public class CSCI331Mobile { // declare class

      // start class definition by declaring
      // instance variables
      private Engine _engine;
      private Door _driverDoor,
                   _passengerDoor;
      private Wheel _frontDriverWheel,
                    _rearDriverWheel,
                    _frontPassengerWheel,
                    _rearPassengerWheel;

      public CSCI331Mobile() { // declare constructor
```

Constructor for `CSCI331Mobile`

Note: The point of this is to show an outline of what a generic class definition looks like. Some functionality has been elided with
`// comments`.

Three parts to class definition:

- declaration, list of properties, list of capabilities

```
package Demos.Car;

/**
 * This class models a vehicle that
 * can move and turn.
 */
public class CSCI331Mobile { // declare class

          public CSCI331Mobile() { // 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 CSCI331Mobile
```

Methods for **CSCI331Mobile** (cont.)

Object capabilities (methods)

```
// declare and define methods

public void moveForward() {
  // code to move CSCI331Mobile forward
}

public void moveBackward() {
  // code to move CSCI331Mobile backward
}

public void turnLeft() {
  // code to turn CSCI331Mobile left
}

public void turnRight() {
  // code to turn CSCI331Mobile right
}

} // end of class CSCI331Mobile
```

`CSCI331Mobile` Syntax Explained (1 of 5)

```
package Demos.Car;
```

- `package` keyword tells Java that this class should be part of a package
- in this case, package is `Demos.Car`

```
/* ... */
```

- everything between `/*` and `*/` is a *block comment*
  - useful for explaining specifics of classes
  - the compiler ignores comments
  - comment to make code more readable for ourselves and the users of the class

```
/**
 * This class models a vehicle that
 * can move and turn.
 */
```

- comment before class definition is called a *header comment*
  - appears at top of a class
  - explains purpose of a class

```
public class CSCI331Mobile {
```

- *declares* that we are about to create a class named `CSCI331Mobile`
- `public` indicates that any other object can create an instance of this class

`CSCI331Mobile` Syntax Explained (2 of 5)

- Everything associated with a class must appear within curly braces!
  - all instance variables and methods;
  - *no code may appear outside curly braces*: { }

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

```
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, **`CSCI331Mobile`** "encapsulates" its **`_engine`**
- remember, *properties can be 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

`CSCI331Mobile` Syntax Explained (3 of 5)

- name of instance variable is **`_engine`**
  - CSCI331 convention: prefix all instance variables with an underscore: "**`_`**"

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

- we can declare multiple reference variables of the same type by separating them with commas
- **`_driverDoor`** and **`_passengerDoor`** are both instance variables of type **`Door`**
- NOTE: these instance variables are not pointing to any objects yet!

```
public CSCI331Mobile() {
```

- *constructor* for class **`CSCI331Mobile`**
- **remember:** constructor is first message sent to a newly created object
- must have the same identifier (name) as its class
- **`()`** makes it a method

`CSCI331Mobile` Syntax Explained (4 of 5)

```
_engine = new Engine();
```

- the most common use of constructors is to *initialize* instance variables
  - i.e., construct its initial state
  - that's just what we're doing here!

- **note:** Constructor `CSCI331Mobile()` refers directly to instance variable `_engine`
  - all methods, including constructor, have direct access to all of their class' instance variables

- the rest of the instance variables are initialized in the same way

`CSCI331Mobile` Syntax Explained (5 of 5)

```
public void moveForward() {
```

- *declares* 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 ("call **`moveForward`** on that instance")

- reserved word **`void`** indicates that this method does not return a result when called
  - some methods return values to the object which called the method
  - constructor declaration does not include return value (because constructors always return a new object instance!)
  - more on return values next lecture

- **`moveForward`** is name of method
  - CSCI331 convention: method names start with lowercase letter, and all following words in method name are capitalized

- anything inside curly braces **`{ }`** is part of method definition's body

`CSCI331Mobile`

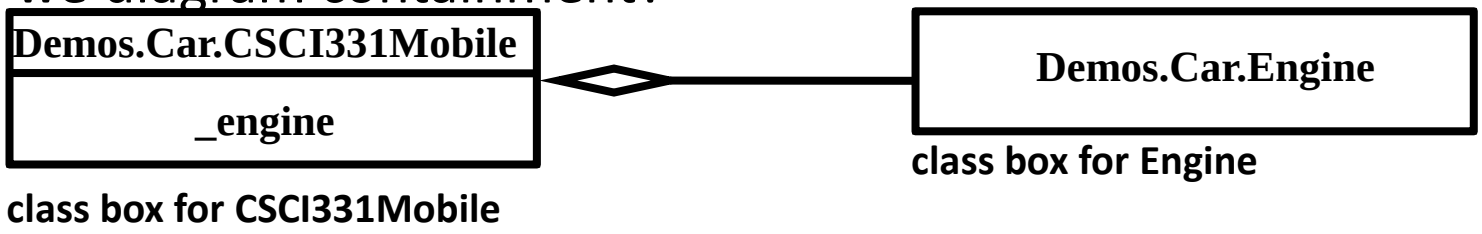
- That's it for basic skeleton of class **`CSCI331Mobile`**!

- Now you know how to write a class with properties (instance variables) and capabilities (methods).

- In a few weeks, you would be able to write the full **`CSCI331Mobile`** class!
  - you would be able to fully define methods
  - you would add a few more instance variables and change methods a little
  - but *basic structure will be the same!*

- Next we look at the representation of objects' three types of properties. These are:
  - components
  - associations with other objects
  - attributes

- In our description, we said the **`CSCI331Mobile`** had an engine, doors, and wheels; these are its *components*.
- We say that the **`CSCI331Mobile`** *is composed of* its engine, doors, and wheels.
- *Containment* is when one class is a component of another.
- How do you determine containment?
  - class **`CSCI331Mobile`** has an instance variable of type **`Engine`**
  - class **`CSCI331Mobile`** *creates* an instance of type **`Engine`**
  - therefore, **`CSCI331Mobile`** *contains* an **`Engine`**
- How do we diagram containment?



- Diagramming covered in the next lecture

- Let's say we have a (very self-aware) `City` object.

- `City` contains and therefore constructs
  - parks
  - schools
  - streets
  - cars, e.g., CSCI331Mobiles (hey, why not?)

- Therefore, `City` can call methods on
  - parks
  - schools
  - streets
  - CSCI331Mobiles

- But, *this relationship is not symmetric*!

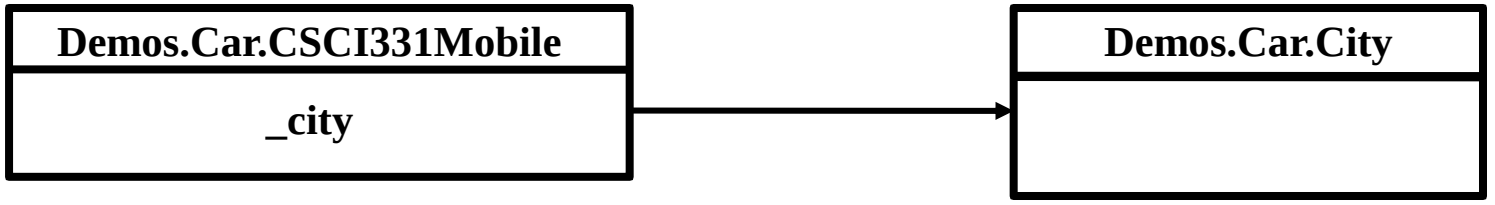
- **`Park`**, **`School`**, **`Street`** and **`CSCI331Mobile`** classes cannot create new cities, but they may need to know about some properties of the city, for example to avoid collision of a car with the building

- Let's focus on our **`CSCI331Mobile`**: how can we provide **`CSCI331Mobile`** with access to **`City`**?

- Answer: *Associate* the **`CSCI331Mobile`** with its **`City`**

- How do you determine association relationship?
  - we'll add to class **`CSCI331Mobile`** a reference variable of type **`City`**
  - since class **`CSCI331Mobile`** ***doesn't create*** an instance of type **`City`**, **`City`** will not be contained by **`CSCI331Mobile`**
  - we say: class **`CSCI331Mobile`** "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?



Attributes

- The **`CSCI331Mobile`** has certain attributes
  - color, size, position, etc.

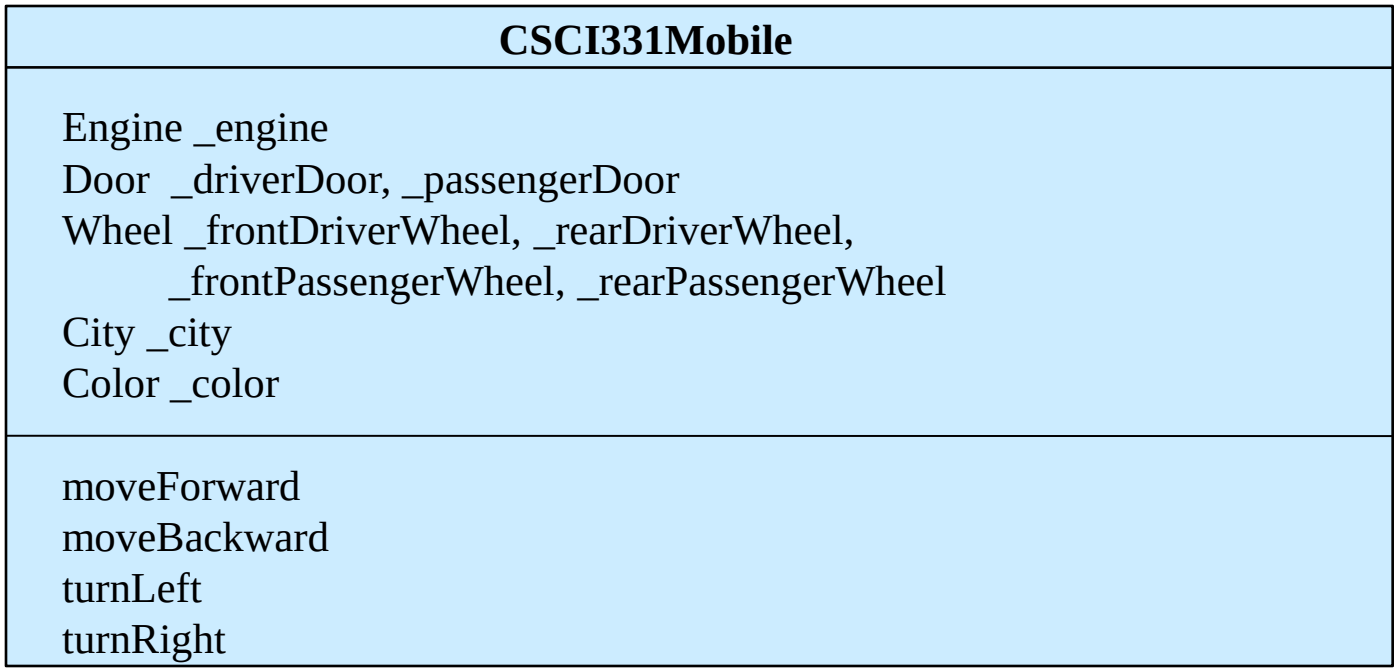
- Attributes are properties that *describe* the **`CSCI331Mobile`**
  - we'll add to class **`CSCI331Mobile`** an instance variable of type **`Color`**
  - **`CSCI331Mobile`** ***is described by*** its **`Color`**
  - this is different from an "***is composed of***" relationship
  - class **`CSCI331Mobile`** ***doesn't contain*** its **`Color`**, ***nor is it associated*** with it
  - we say: **`Color`** is an ***attribute*** of class **`CSCI331Mobile`**
  - class **`CSCI331Mobile`** may set its own **`Color`**, or another class may call a method on it to set its **`Color`**
  - the actual color of the **`CSCI331Mobile`** is an attribute, but it is also an instance of the **`Color`** class
    - **all instance variables are instances!**

- How do we diagram an attribute?

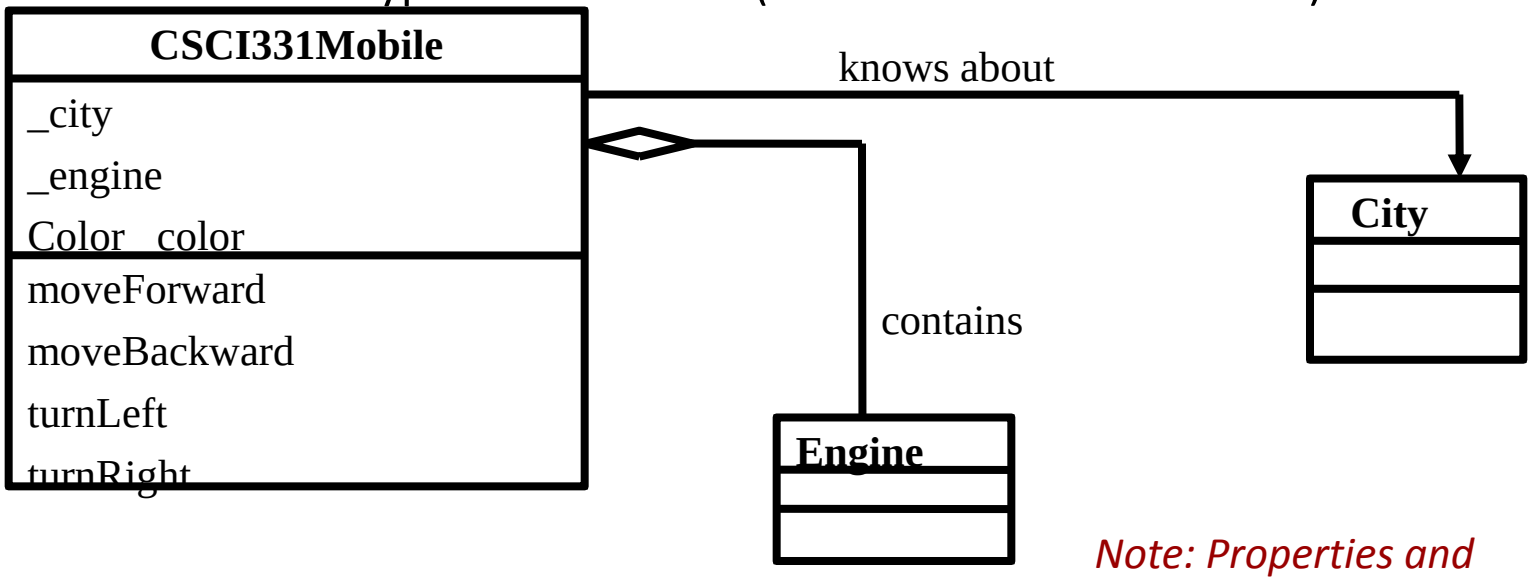
Class Box

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

- Example of class **`CSCI331Mobile`** with the added properties just discussed:



- A *class diagram* shows how classes relate to other classes (as shown briefly on previous slides)
  - rectangles represent classes
  - relationships between classes are shown with lines
  - Association and containment properties have their names with reference to class boxes representing their type
  - attributes have type and identifier (but don't show references)



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

Note: Properties and Capabilities of `City` and `Engine` have been elided for clarity

Packages and Accessing Classes

- **`CSCI331Mobile`** is in package **`Demos`**

- It is in its own sub-package
  - so its *qualified* (complete) name is: **`Demos.Car.CSCI331Mobile`**
  - qualified name of a class includes names of all packages it belongs to (e.g., **`Demos`** and its sub-package **`Car`**)

- To access a class, you can always refer to it by its qualified name

- But if class you want to access is in the same package as the current class, you can omit the package name
  - **`Engine`** is in package **`Demos.Car`**
  - package **`Demos.Car`** at the top of **`CSCI331Mobile`** class definition makes **`CSCI331Mobile`** part of the **`Demos.Car`** package
  - therefore, **`CSCI331Mobile`** can refer to **`Demos.Car.Engine`** as, simply, **`Engine`**

Working With Variables

- Remember **`CSCI331Mobile`**? 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 reference variables a default value of **`null`**
  - i.e., it has no useful value
  - **`null`** is another reserved word in Java
  - means a non-existent memory address

Uninitialized Variables and `null`

- If you forget to give your reference variables initial values, Java VM will reward you with a *runtime error* in the form of a
  `NullPointerException`
  - runtime errors are problems that occur while your program is running
  - i.e., your program compiled successfully, but it does not execute successfully
  - for now, when runtime errors occur, your program is usually stopped by Java VM

- **`NullPointerException`**
  - if you get such an error, make sure you have initialized all of your object's instance variables!
  - most common occurrence of a **`NullPointerException`** is trying to send a message to an uninitialized variable

WATCH OUT!

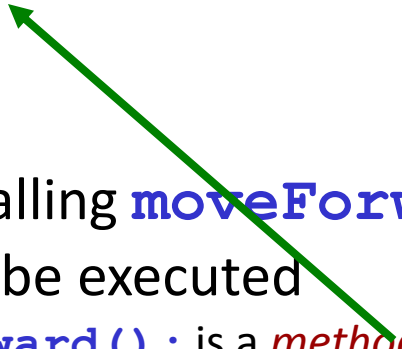
Working With Methods

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

```
public class City {

  private CSCI331Mobile _15mobile;

  public City() {
     _mobile = new CSCI331Mobile();
     _mobile.moveForward();
  }
}
```



- Sending a message (calling **`moveForward`** on **`_mobile`**) causes the method's code to be executed
  - **`_mobile.moveForward();`** is a *method call*
  - **`_mobile`** is the message's *receiver* (the instance being told to move)
  - dot ("**`.`**") separates receiver from method name
  - **`moveForward`** is the name of message to be sent
  - **`()`** denotes parameters sent to the message

`this` keyword (1 of 2)

- What if we want one method in a class to call another method in the same class?
  - say we want the **`CSCI331Mobile`** to have a **`turnAround()`** method
  - want **`turnAround()`** method to call **`CSCI331Mobile`**'s own **`turnLeft()`** or **`turnRight()`** method twice

- In order for *current instance* to be 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 (2 of 2)

- 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 current class to execute 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 same thing

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

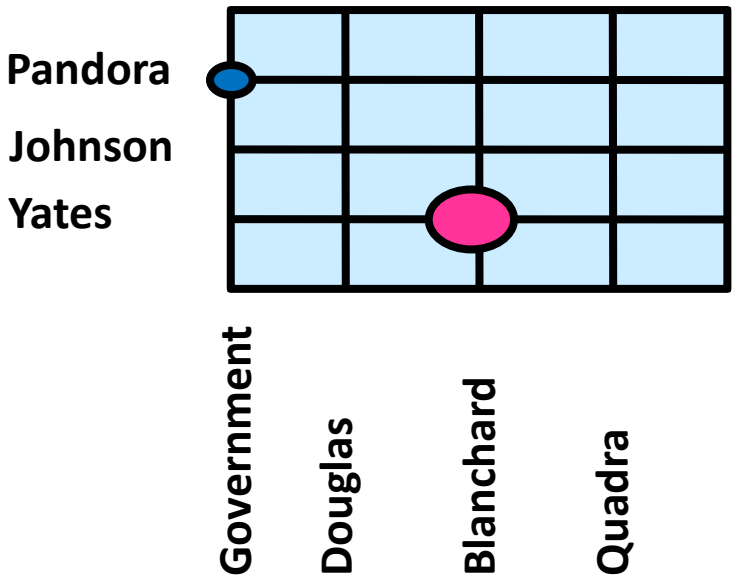
may be shorter, but not as clear

Example: Driving Around Victoria

- Imagine a "world" where the **`CSCI331Mobile`** moves only along the roads defined by a regular grid:
  - simplified city map: the streets of Victoria are all the same length and go only horizontally and vertically (also, they are all 2-way)
  - **`CSCI331Mobile`** can move forward in the direction that it is facing and can turn 90 degrees left or right
  - can move only one block at a time

Example: Driving Around Victoria

- How do we get **`CSCI331Mobile`** to a movie theatre (corner of Yates and Blanchard) and back, given:
  - **`CSCI331Mobile`** starts at corner of Pandora and Government facing north (initial conditions)



We're Done!

- It's *that* simple!
- Now you know how to create and use a class
- Next time: Customizing methods and setting up associations between objects!