Defining Classes and Methods

Chapter 5

Modified by James O’Reilly

*JAVA: An Introduction to Problem Solving & Programming, 7*th Ed. By Walter Savitch
ISBN 0133862119 © 2015 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

Class and Method Definitions

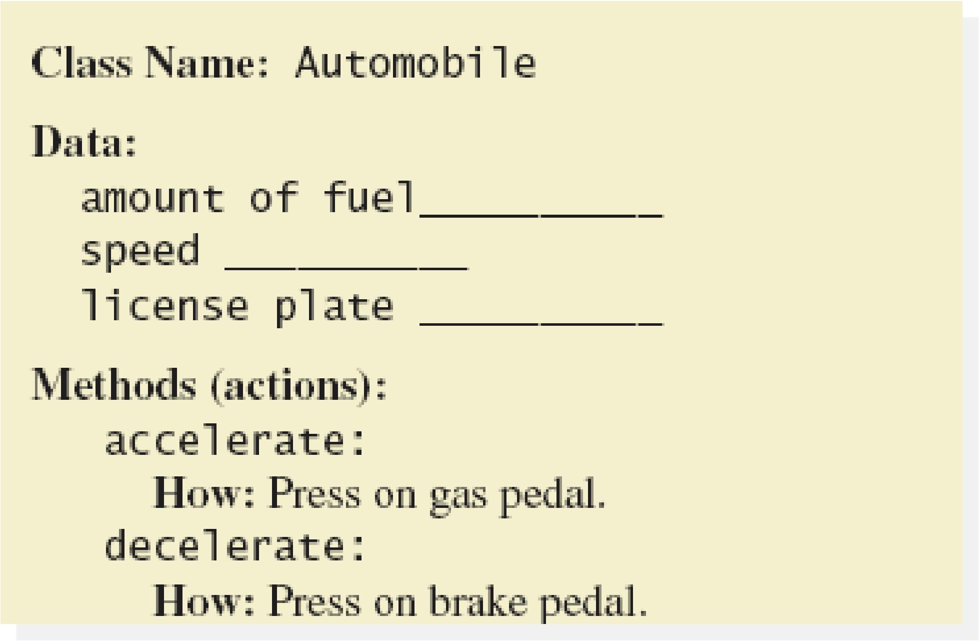
- OOP- Object Oriented Programming – Big Idea***s***:
  - Group data and related functions (methods) into Objects (Encapsulation)
  - Objects are normally "Noun" concepts which have class types
  - Objects can be made from (composed of) primitive data types and Objects
  - Objects can often be treated as abstractions (interface separate from implementation– information hiding)
  - Objects can ***inherit*** *traits from other Objects* (one is a subtype of the other)
- Java programs typically consist of multiple objects of class types
  - The Objects interact with one another where necessary
  - These Objects can make it easier to understand the interactions between parts of a program – a Person class stores information about people, Car about cars…
- Program objects can represent Objects in real world and Abstractions

Class Files and Separate Compilation

- Each `Java` class definition usually in a file by itself
  - File begins with name of the class
  - Ends with `.java`
- Class can be compiled separately
- Helpful to keep all class files used by a program in the same directory

Class and Method Definitions (the *Type*)

- Figure 5.1 A class as a blueprint



Class Name: Automobile

Data:

amount of fuel__________
speed __________
license plate __________

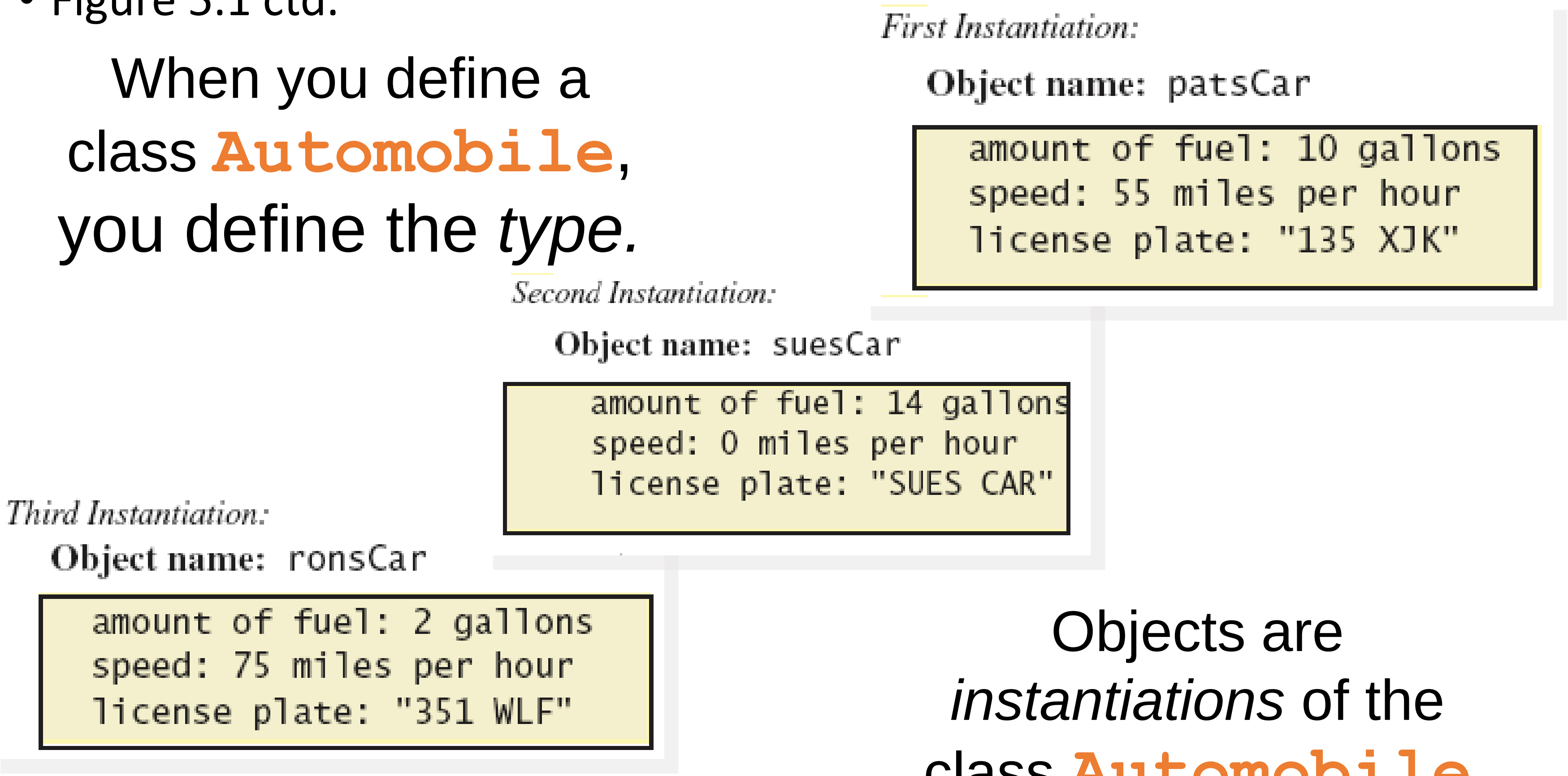
Methods (actions):

accelerate:
How: Press on gas pedal.
decelerate:
How: Press on brake pedal.

Class Definitions and Instantiations

- Figure 5.1 ctd.

When you define a class **`Automobile`**, you define the *type.*



First Instantiation:

Object name: `patsCar`

```
amount of fuel: 10 gallons
speed: 55 miles per hour
license plate: "135 XJK"
```

Second Instantiation:

Object name: `suesCar`

```
amount of fuel: 14 gallons
speed: 0 miles per hour
license plate: "SUES CAR"
```

Third Instantiation:

Object name: `ronsCar`

```
amount of fuel: 2 gallons
speed: 75 miles per hour
license plate: "351 WLF"
```

Objects are *instantiations* of the class **`Automobile`**

Encapsulation

- Consider example of driving a car
  - We see and use break pedal, accelerator pedal, steering wheel – know <u>what</u> they do
  - We do <u>not</u> see mechanical details of <u>how</u> they do their jobs
- Encapsulation divides class definition into
  - Class interface
  - Class implementation
- (good example of *abstraction* too),
- (possibly good example of info hiding)

Encapsulation

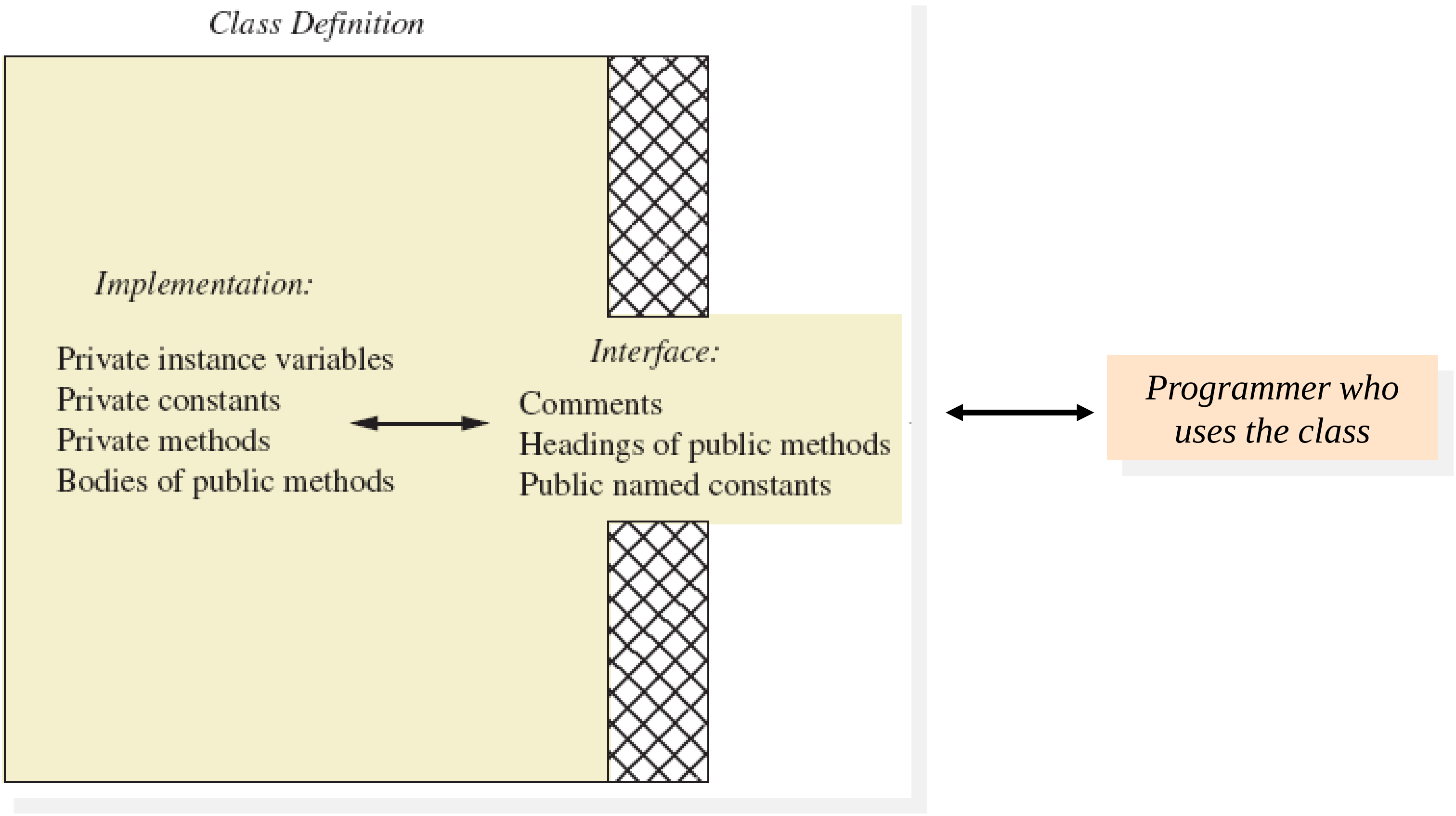
- A *class interface*
  - Tells what the class does/provides
  - Gives headings for public methods and comments about them
  - Helps manage complexity as a project grows
  - Can always make a member **public** easily, which is not true for **private**
- A *class implementation*
  - Contains private variables
  - Includes definitions of public and private methods

Information Hiding

- Programmer using a class method need not know details of implementation
  - Only needs to know *what* the method does
  - Can mark items as **private** (and others…) to indicate who should access
- Information hiding:
  - Designing a method so it can be used without knowing details
- Also related to *abstraction* and *encapsulation*
- Method design should separate *what* from *how,* this allows changes to methods to be done without modifying dependent code – great for fixes and optimization
- Abstraction: the parts that are hidden can be ignored by programmers using – not modifying -- the class. The generally visible public parts represent a simplification of the whole.

Encapsulation, Info. Hiding, Abstraction

- Figure 5.3 A well encapsulated class definition
- Remember that the interface may represent a form of *abstraction*



Class Definition

Implementation:

Private instance variables
Private constants
Private methods
Bodies of public methods

Interface:

Comments
Headings of public methods
Public named constants

Programmer who uses the class

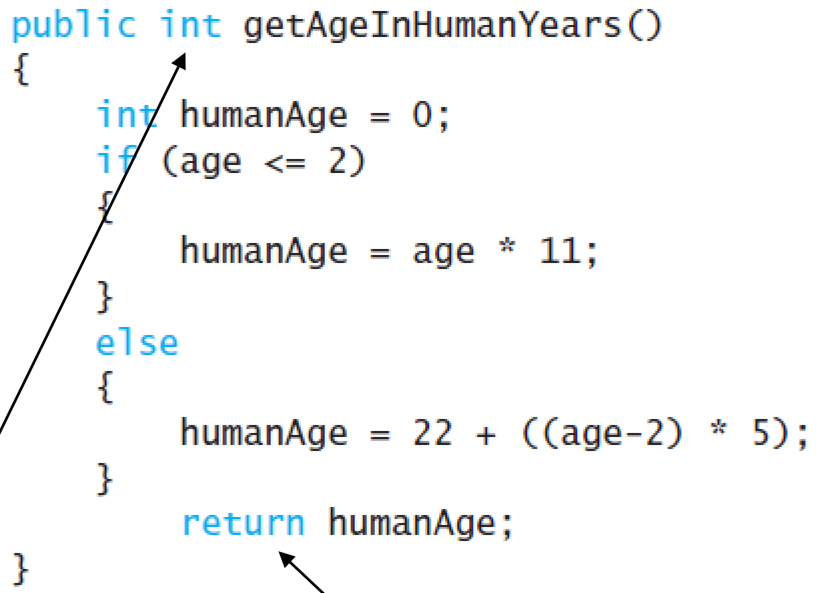
*JAVA: An Introduction to Problem Solving & Programming, 7*th Ed. By Walter Savitch
ISBN 0133862119 © 2015 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

Methods

- When you use a method you "invoke" or "call" it
- Generally a verb (it is an *action*)
- Two kinds of Java methods
  - Return a single item (can be primitive, array or other Object). Can be used to get a value (e.g. String methods)
  - Perform some other action – a `void` method. Will do something but not return a value (a method should do something or return something or be deleted).
- The method `main` is a `void` method
  - Invoked by the system
  - Not by the application program (the general case)

Methods That Return a Value

- Consider method **getAgeInHumanYears( )**

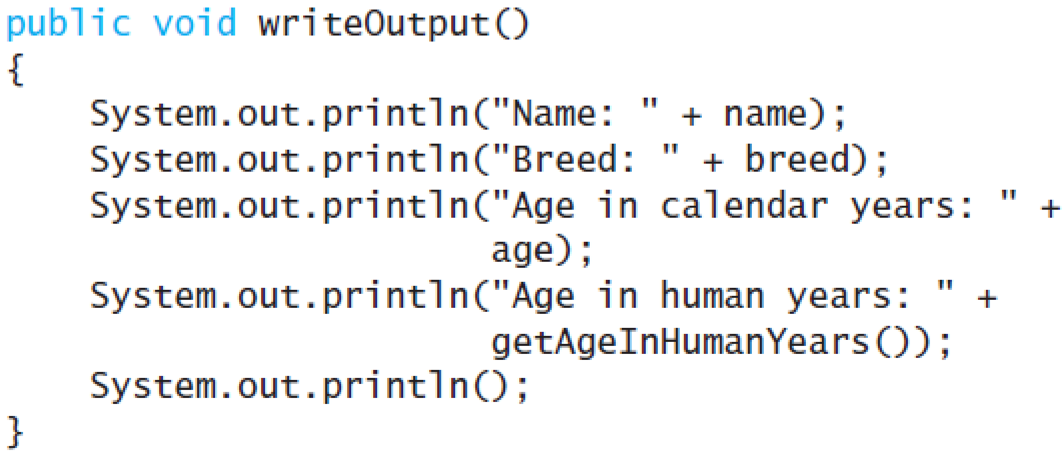


```
public int getAgeInHumanYears()
{
    int humanAge = 0;
    if (age <= 2)
    {
        humanAge = age * 11;
    }
    else
    {
        humanAge = 22 + ((age-2) * 5);
    }
        return humanAge;
}
```

- Heading declares type of value to be returned
- Last statement executed is **return**

Defining **void** Methods

- Consider method **writeOutput** from Listing 5.1



```
public void writeOutput()
{
    System.out.println("Name: " + name);
    System.out.println("Breed: " + breed);
    System.out.println("Age in calendar years: " +
                       age);
    System.out.println("Age in human years: " +
                       getAgeInHumanYears());
    System.out.println();
}
```

- Method definitions appear inside class definition
  - Can be used only with objects of that class (or the class name, for **static** methods)
  - Can **return** in a void method (just **return;**), ends execution of that method.

The Keyword `this`

- Referring to instance variables outside of the class – must use
  - Name of an object of the class
  - Followed by a dot
  - Name of instance variable

```
keyboard.nextLine();
```

- Inside the class,
  - Use name of variable alone
  - The object (unnamed) is understood to be there from the context
  - Do not use within **static** methods (such as **main()**)
- Inside the class the unnamed object can be referred to with the name `this`
- Example

```
this.name = keyboard.nextLine();
```

- The keyword `this` stands for the receiving object
- We will see some situations later that require the `this`

Local Variables

- Variables declared inside a method are called *local* variables
  - May be used only inside the method
  - All variables declared in method `main` are local to `main`
  - Must be initialized before being read (other variables have defaults)
- Local variables having the same name and declared in different methods are different variables

Blocks

- Recall compound statements
  - Enclosed in braces { }
- When you declare a variable within a compound statement
  - The compound statement is called a *block*
  - The scope of the variable is from its declaration to the end of the block
- Variable declared outside the block usable both outside and inside the block

Parameters of Primitive Type

- Note the declaration

```
public int predictPopulation(int years)
```

  - The *formal* parameter is `years`

- Calling the method

```
int futurePopulation =
 speciesOfTheMonth.predictPopulation(10);
```

  - The *actual parameter*, also called the *argument*, is the integer 10

Parameters of Primitive Type

- Parameter names are local to the method
- When method invoked
  - Each parameter initialized to value in corresponding actual parameter
  - Primitive actual parameter cannot be altered by invocation of the method – pass by *value*
- Automatic type conversion performed

```
byte -> short -> int ->
      long -> float -> double
```

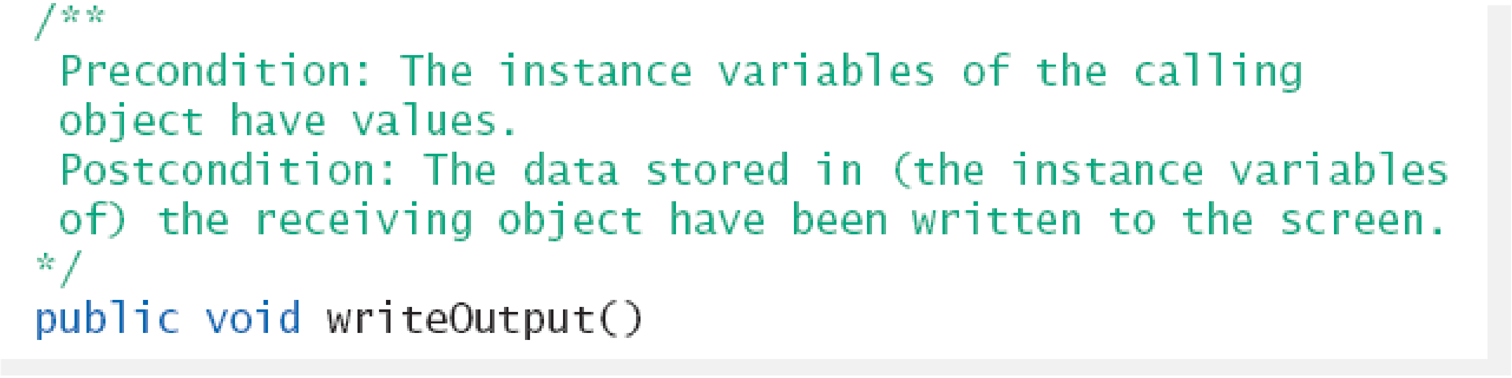
Automatic Documentation `javadoc`

- Generates documentation for class interface
- Comments in source code must be enclosed in `/**   */`
- Utility `javadoc` will include
  - These comments
  - Headings of public methods
- Output of `javadoc` is HTML format (webpage format).

Pre- and Postcondition Comments

- Precondition comment
  - States conditions that must be true before method is invoked
- Postcondition comment
  - Tells what will be true after method executed

- Example



```
/**
 Precondition: The instance variables of the calling
 object have values.
 Postcondition: The data stored in (the instance variables
 of) the receiving object have been written to the screen.
*/
public void writeOutput()
```

Access Modifiers

- For general use: specified as `public`
  - Any other class can directly access that object by name
  - Classes generally specified as `public`
- Only class should modify/access: specify `private`
  - Instance variables usually `private`
  - <u>Make all member variables private unless you have a good reason not to.</u>
- Also, two others
  - `<package-private>:` only visible within package (collection of related files)
  - `protected:` ”The protected modifier specifies that the member can only be accessed within its own package (as with package-private) and, in addition, by a subclass of its class in another package.”[1]

Programming Example

- Another implementation of a Rectangle class
- View sample code, listing 5.10
  `class Rectangle2`
- Note **`setDimensions`** method
  - This is the only way the **`width`** and **`height`** may be altered outside the class

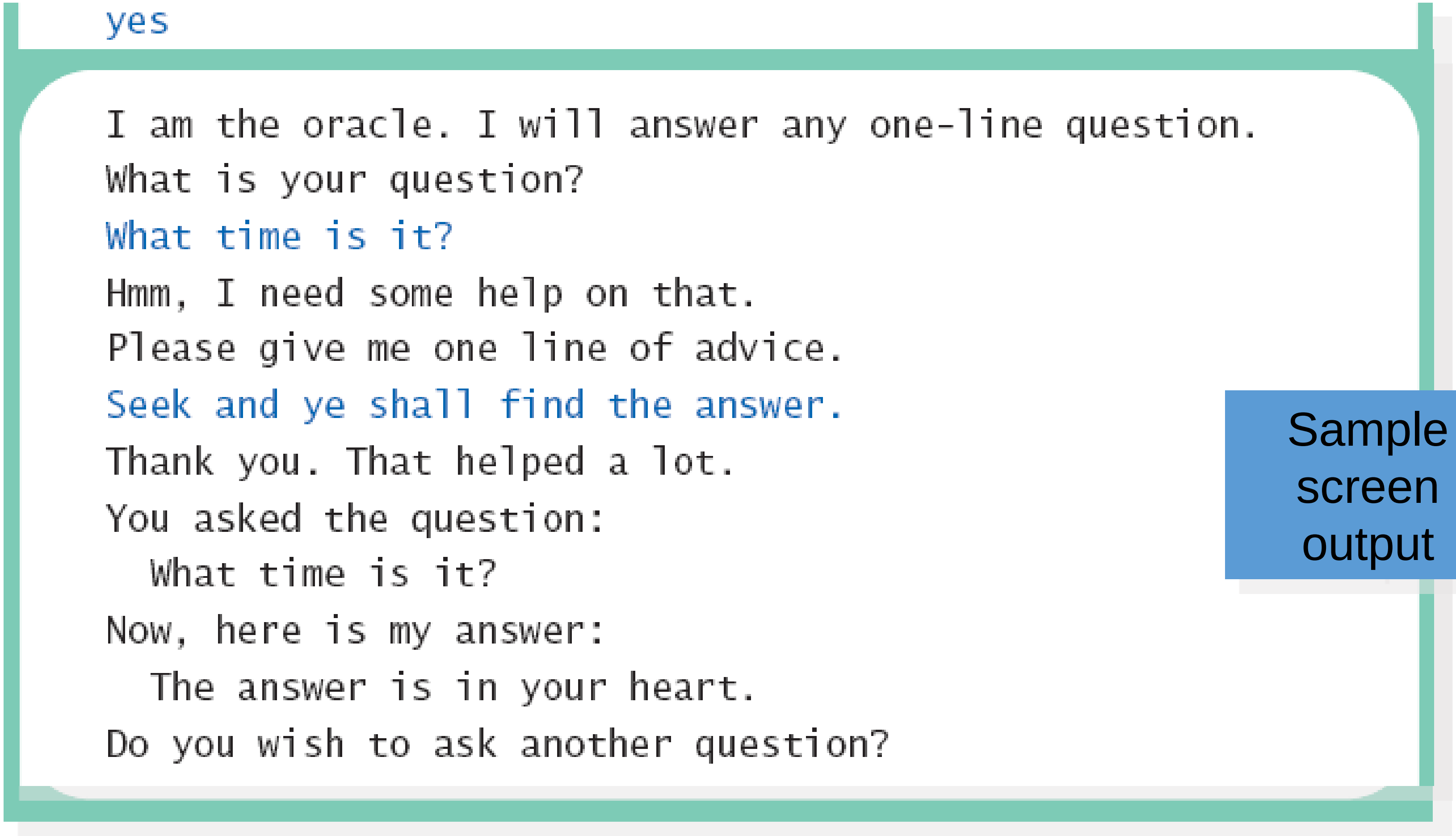
Encapsulation with Information Hiding (a recipe for making a class)

- Preface class definition with comment on how to use class
- Declare all instance variables in the class as private.
- Provide public accessor methods to retrieve data
- Provide public methods manipulating data
  - Such methods could include public mutator methods.
- Place a comment before each public method heading that fully specifies how to use method.
- Make any helping methods private.
- Write comments within class definition to describe implementation details.

Methods Calling Methods

- A method body may call any other method
- If the invoked method is within the same class
  - Need not use prefix of receiving object
- View demo program, listing 5.16
  class OracleDemo

Methods Calling Methods



```
yes

I am the oracle. I will answer any one-line question.
What is your question?
What time is it?
Hmm, I need some help on that.
Please give me one line of advice.
Seek and ye shall find the answer.
Thank you. That helped a lot.
You asked the question:
  What time is it?
Now, here is my answer:
  The answer is in your heart.
Do you wish to ask another question?
```

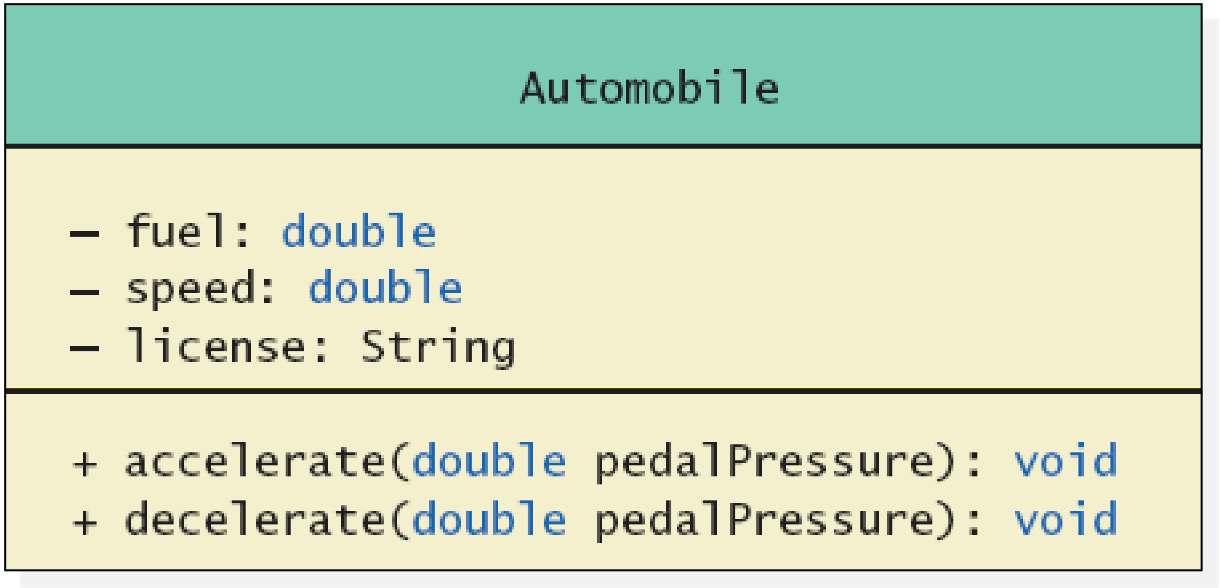
Sample screen output

UML Class Diagrams

- Contains more than interface, less than full implementation
- Usually written *before* class is defined – a good design before implementation prevents rewrites
- Used by the programmer defining the class
  - Contrast with the interface used by programmer who uses the class

Class and Method Definitions

- Figure 5.2 A class outline as a UML class diagram
- + and – indicate public/private, respectively (later – Access mods)



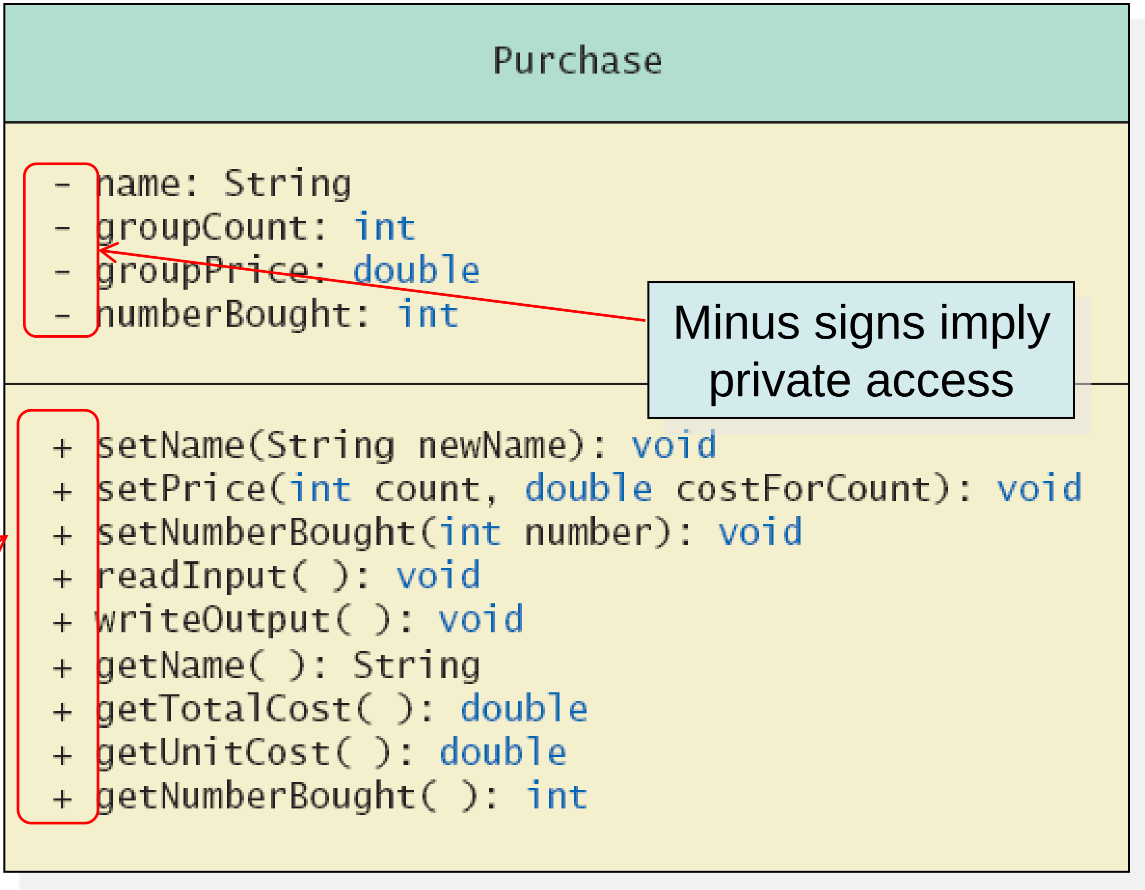
```
Automobile

– fuel: double
– speed: double
– license: String

+ accelerate(double pedalPressure): void
+ decelerate(double pedalPressure): void
```

UML Class Diagrams

- Note Figure 5.4 for the **Purchase** class



```
Purchase

- name: String
- groupCount: int
- groupPrice: double
- numberBought: int

+ setName(String newName): void
+ setPrice(int count, double costForCount): void
+ setNumberBought(int number): void
+ readInput( ): void
+ writeOutput( ): void
+ getName( ): String
+ getTotalCost( ): double
+ getUnitCost( ): double
+ getNumberBought( ): int
```

Minus signs imply private access

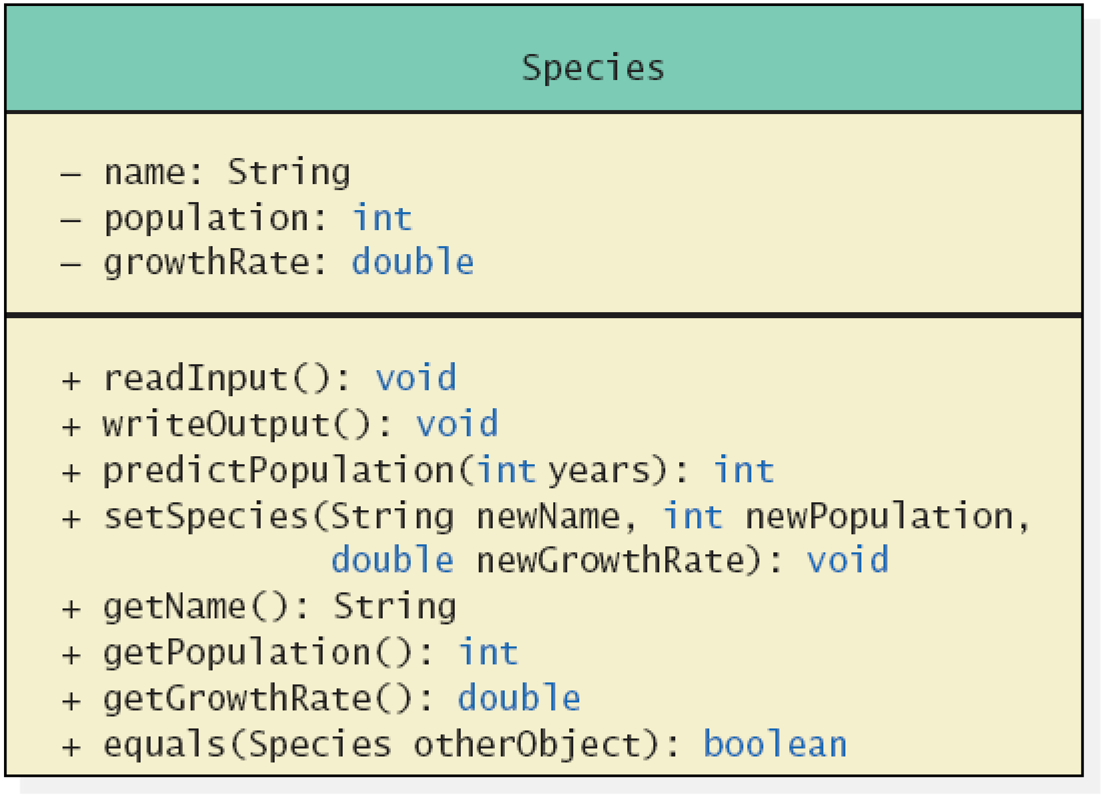
Plus signs imply public access

Variables of a Class Type

- All variables are implemented as a memory location
- Data, the actual value, of *primitive type* stored in the memory location assigned to the variable
- Variable of *class type* contains memory address of object named by the variable
- Address called the *reference* to the variable
- A *reference type* variable holds references (memory addresses) , not all the data

Complete Programming Example

- View sample code, listing 5.19
  class **Species**
- Figure 5.7
  Class Diagram
  for the class
  Species
  in listing 5.19



Species

```
– name: String
– population: int
– growthRate: double
```

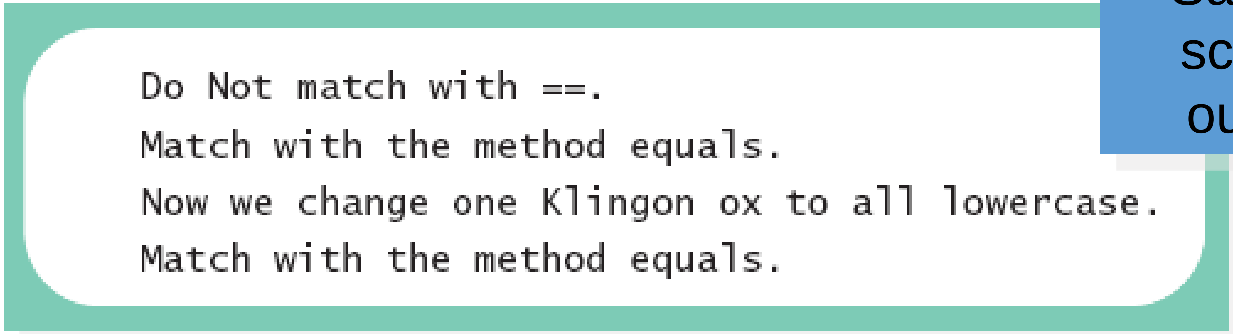
```
+ readInput(): void
+ writeOutput(): void
+ predictPopulation(int years): int
+ setSpecies(String newName, int newPopulation,
             double newGrowthRate): void
+ getName(): String
+ getPopulation(): int
+ getGrowthRate(): double
+ equals(Species otherObject): boolean
```

Defining an **equals** Method

- As demonstrated by previous figures
  - We cannot use == to compare two objects
  - We must write a method for a given class which will make the comparison as needed
- View sample code, listing 5.17
  class Species
- The **equals** for this class method used same way as **equals** method for **String**

Demonstrating an **`equals`** Method

- View sample program, listing 5.18
  `class SpeciesEqualsDemo`
- Note difference in the two comparison methods **`==`** versus **`.equals( )`**



```
Do Not match with ==.
Match with the method equals.
Now we change one Klingon ox to all lowercase.
Match with the method equals.
```

Sample screen output

Accessor and Mutator Methods

- When instance variables are private must provide methods to access values stored there
  - Typically named `getSomeValue`
  - Referred to as an accessor method
- Must also provide methods to change the values of the private instance variable
  - Typically named `setSomeValue`
  - Referred to as a mutator method
  - Allows us to check the values (e.g. negative width doesn't make sense normally)

Accessor and Mutator Methods

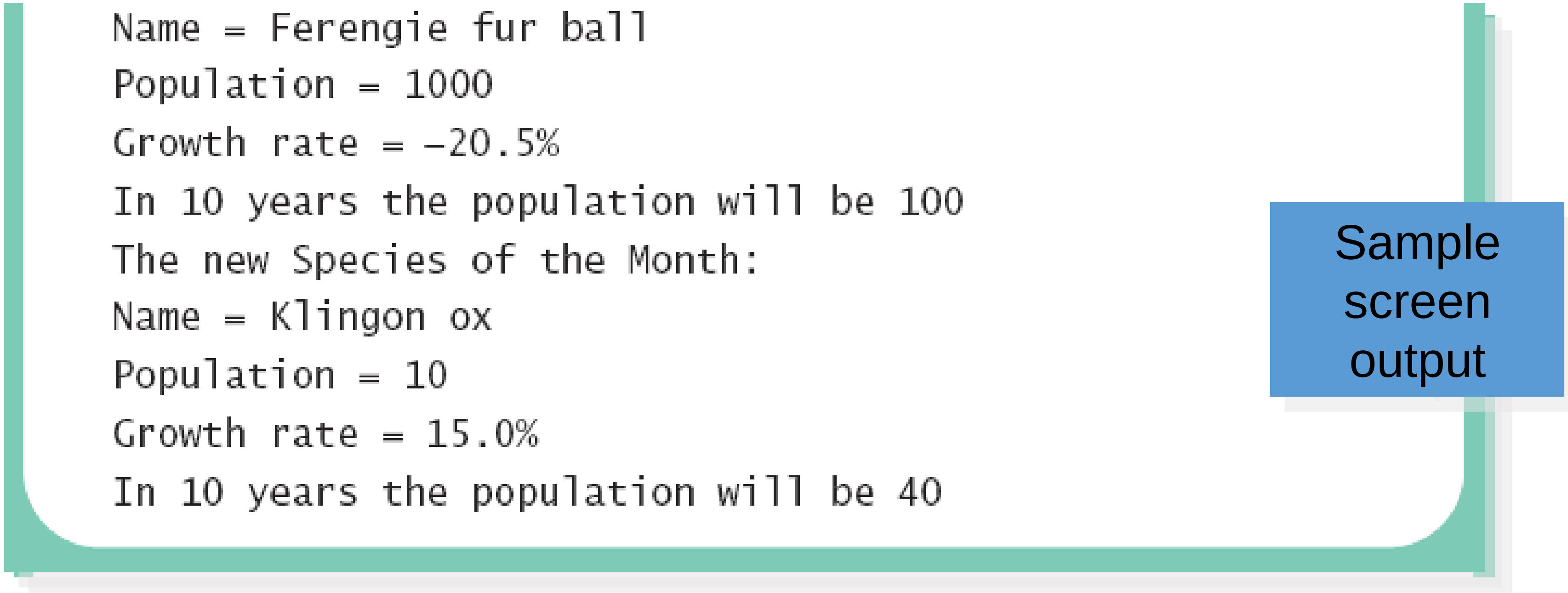
- Consider an example class with accessor and mutator methods
- View sample code, listing 5.11

  `class Species`
- Note the mutator method
  - **`setSpecies`**
- Note accessor methods
  - **`getName`**, **`getPopulation`**, **`getGrowthRate`**

Accessor and Mutator Methods

- Using a mutator method
- View sample program, listing 5.12

`class Species`



```
Name = Ferengie fur ball
Population = 1000
Growth rate = -20.5%
In 10 years the population will be 100
The new Species of the Month:
Name = Klingon ox
Population = 10
Growth rate = 15.0%
In 10 years the population will be 40
```

Sample screen output

Unit Testing

- A methodology to test correctness of individual units of code
    - Typically methods, classes
- Collection of unit tests is the **test suite**
- The process of running tests repeatedly after changes are make sure everything still works is **regression testing**

Method Parameters of a Class Type

- When assignment operator used with objects of class type
  - Only memory address is copied
- Similar to use of variables of class type
  - Memory address of actual parameter passed to formal parameter
  - Formal parameter may access public elements of the class
  - Actual parameter thus can be changed by class methods

References

- [1] https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html (last accessed 10/19)