Defining Classes and Methods

Chapter 5

Class and Method Definitions: Outline

- Class Files and Separate Compilation
- Instance Variables
- Methods
- The Keyword this
- Local Variables
- Blocks
- Parameters of a Primitive Type

Class and Method Definitions

- Java program consists of objects which interact with one another
  - Objects of class types (String, Scanner)
  - Objects have both data and methods
- Program objects can represent
  - Objects in real world
  - Abstractions

Class and Method Definitions

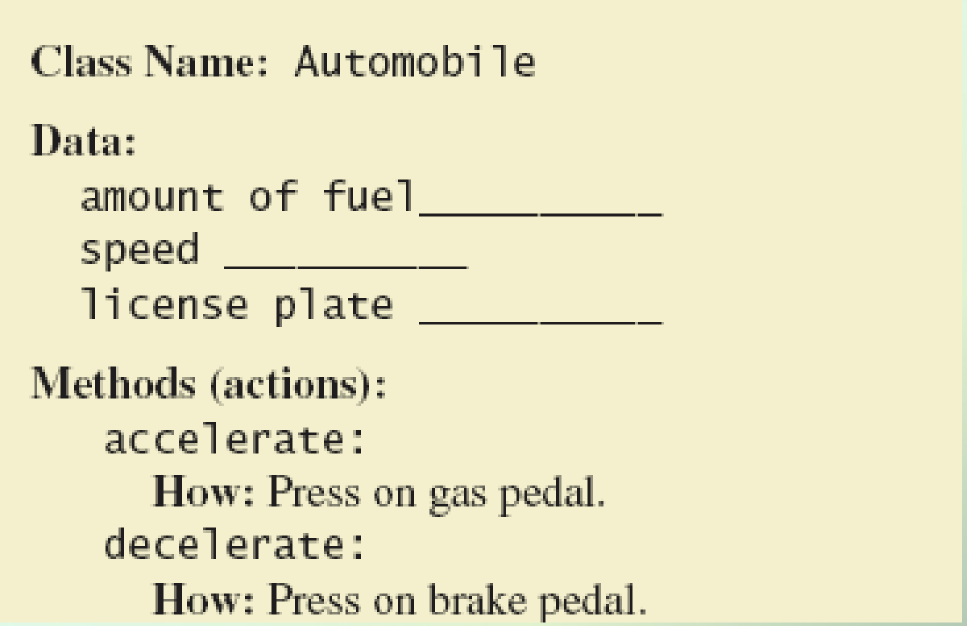
- A **class definition** is a **template** or **blueprint** for creating objects
- A class definition is like a cookie-cutter
- A cookie cutter is not a cookie, but it can be used to create cookies
- Each cookie created by a particular cookie-cutter will have **the same attributes** (thickness, decoration), but **different values for those attributes** (3mm, “#1 Luke”)

Class and Method Definitions

- An **instance** of a class is an object of that class type

Class and Method Definitions

- Figure 5.1 A class as a blueprint



Class Name: Automobile

Data:

amount of fuel__________
speed __________
license plate __________

Methods (actions):

accelerate:

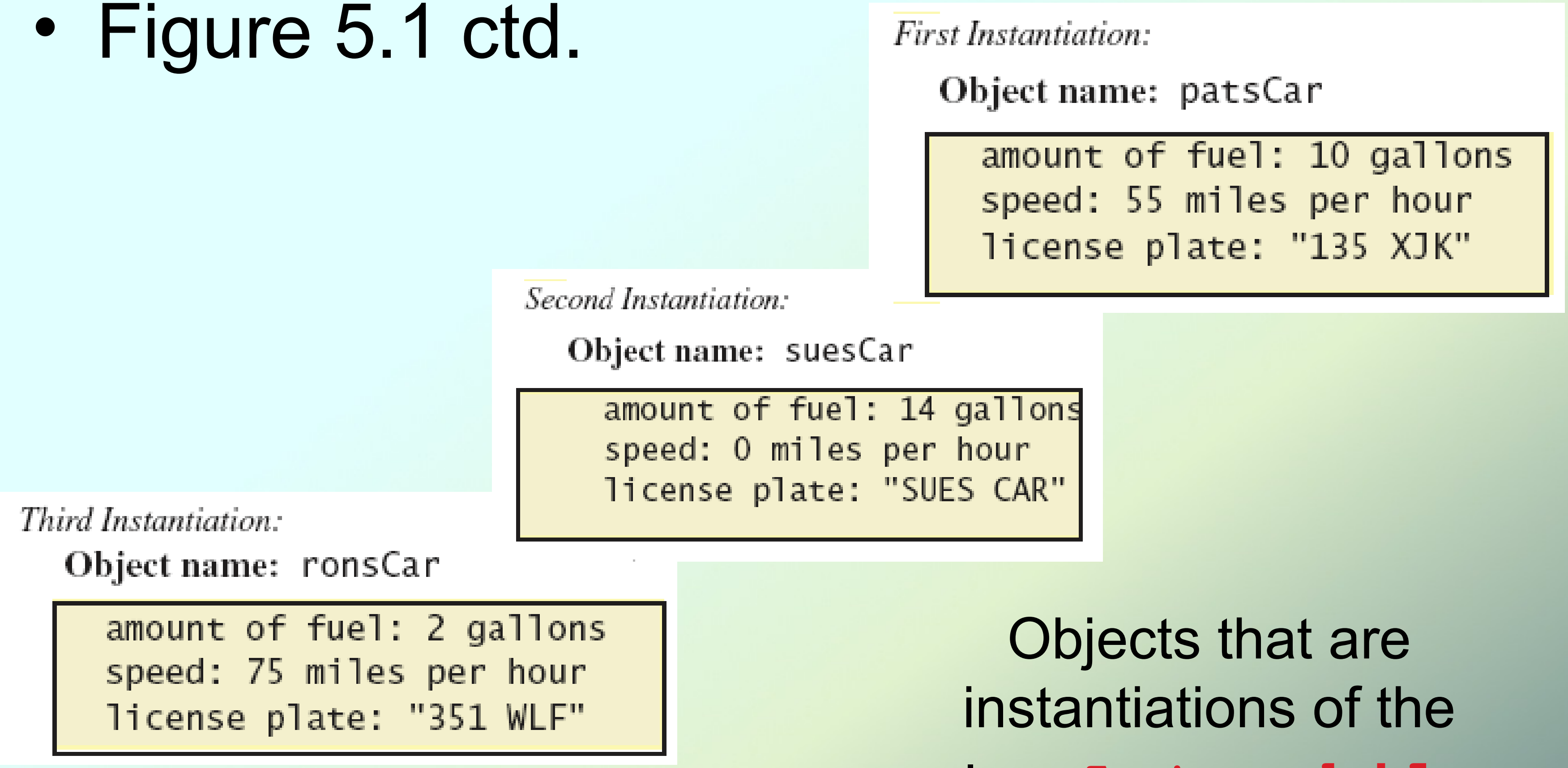
How: Press on gas pedal.

decelerate:

How: Press on brake pedal.

Class and Method Definitions

- Figure 5.1 ctd.



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:

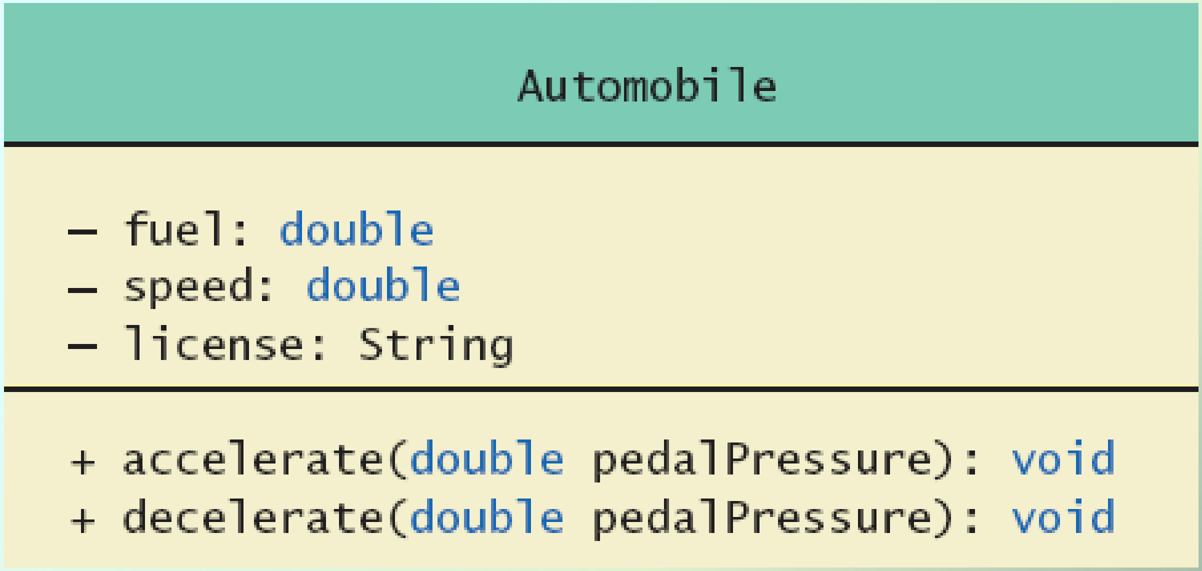
Object name: ronsCar

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

Objects that are instantiations of the class `Automobile`

Class and Method Definitions

- Figure 5.2 A class outline as a UML class diagram



Automobile

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

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

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

`Dog` class and Instance Variables

- View `Dog.java` and `DogDemo.java`
- Note `Dog` has
  - Three pieces of data (instance variables)
  - Two behaviors (methods)
- Each instance of this type has its own copies of the data items
- Use of `public`
  - No restrictions on how variables used
  - Later will replace with `private`

Methods

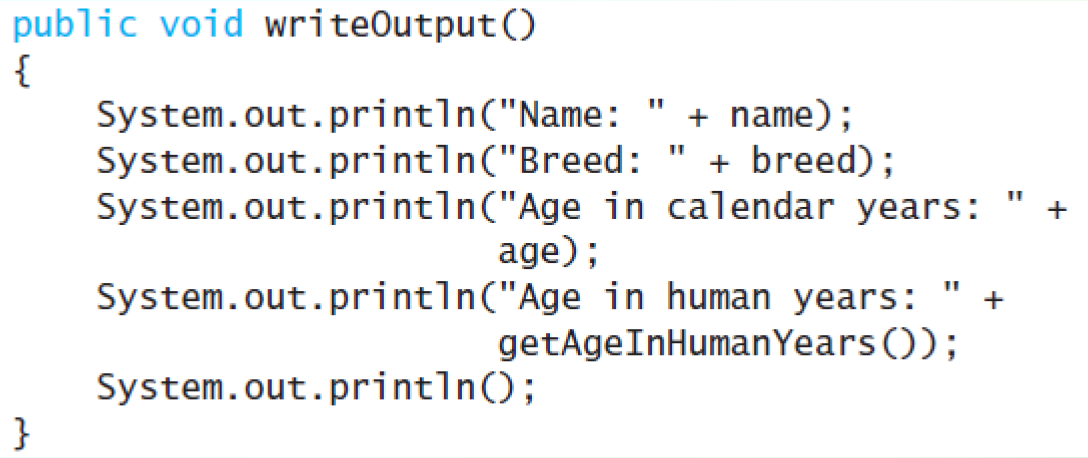
- When you use a method you "invoke" or "call" it
- Two kinds of Java methods
  - Return a single item
  - Perform some other action – a **`void`** method
- The method **`main`** is a **`void`** method
  - Invoked by the system
  - Not by the application program

Methods

- Calling a method that returns a quantity
  - Use anywhere a value can be used
    - `if (keyboard.nextInt() > 0) ...`
- Calling a void method
  - Write the invocation followed by a semicolon
  - Resulting statement performs the action defined by the method
    - `System.out.println("hello");`

Defining **void** Methods

- Consider method **writeOutput** from **Dog**



```
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();
}
```

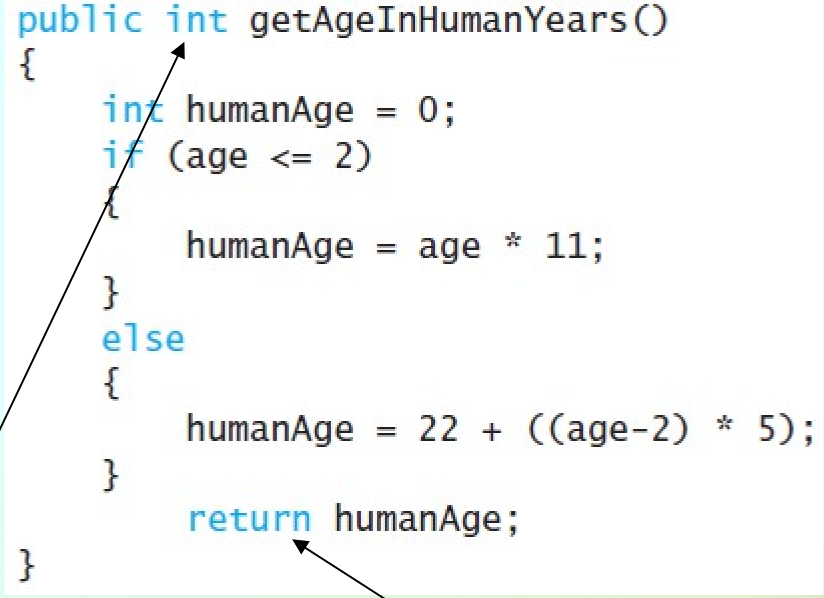
- Method definitions appear inside class definition
    - Can be used only with objects of that class

Defining `void` Methods

- Most method definitions we will see as `public`
- Method does not return a value
  - Specified as a `void` method
- Heading includes parameters
- Body enclosed in braces `{ }`
- Think of method as defining an action to be taken

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`

Example: Species Class

- Class designed to hold records of endangered species
- View **`SpeciesFirstTry.java`**
  - Three instance variables, three methods
  - Will expand this class in the rest of the chapter
- View **`SpeciesFirstTryDemo.java`**

Naming Methods

- Use a verb (or verb phrase) to name a void method
  - Examples: writeOutput
- Use a noun (or noun phrase) to name a method that returns a value
  - Example: nextInt
- All method names should start with a lowercase letter

Referring to Instance Variables

- Referring to instance variables outside the class – must use
  - Name of an object of the class
  - Followed by a dot
  - Name of instance variable
- Inside the class,
  - Use name of variable alone
  - The object (unnamed) is understood to be there

The Keyword `this`

- Inside the class the unnamed object can be referred to with the name `this`
- Example

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

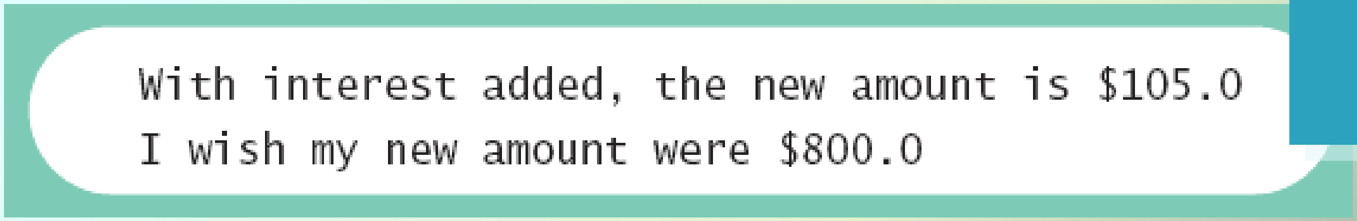
- The keyword `this` stands for the receiving object
  - Can usually be omitted
- We will seem some situations later that require the `this`

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`**
- Local variables having the same name inside a different method are considered different variables

Local Variables

- View **BankAccount.java** and **LocalVariablesDemoProgram.java**
- Note two different variables **newAmount**
  - Note different values output



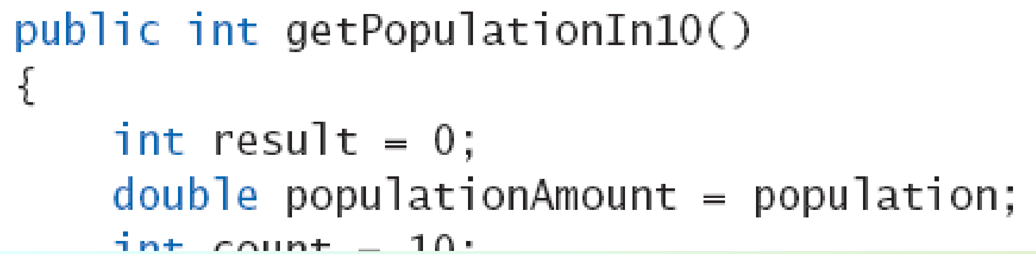
```
With interest added, the new amount is $105.0
I wish my new amount were $800.0
```

Sample screen output

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
- In general: the portion of a program in which a variable has meaning is known as the variable's scope

Parameters of Primitive Type



```
public int getPopulationIn10()
{
    int result = 0;
    double populationAmount = population;
    int count = 10;
```

- Recall method declaration in **SpeciesFirstTry**
  - Note it only works for 10 years
  - We can make it more versatile by giving the method a parameter to specify how many years
- Download **SpeciesSecondTry.java** and **SpeciesSecondTryDemo.java**

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 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
- Automatic type conversion performed

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

Information Hiding

- Programmer using a class method need not know details of implementation
  - Only needs to know *what* the method does
- Information hiding:
  - Designing a method so it can be used without knowing details
- Also referred to as *abstraction*
- Method design should separate *what* from *how*

The `public` and `private` Modifiers

- Type specified as `public`
  - Any other class can directly access that object by name
- Classes generally specified as `public`
- Instance variables usually not `public`
  - Instead specify as `private`
- View `SpeciesThirdTry.java`

JAVA: An Introduction to Problem Solving & Programming, 6th Ed. By Walter Savitch
ISBN 0132162709 © 2012 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

Programming Example

- Demonstration of need for private variables
- Download `Rectangle.java`
- Statement such as

```
    box.width = 6;
```

  is illegal since `width` is `private`

  - Keeps remaining elements of the class consistent in this example

Programming Example

- Another implementation of a Rectangle class
- Download **`Rectangle2.java`**
- Note **`setDimensions`** method
  - This is the only way the **`width`** and **`height`** may be altered outside the class

Accessor and Mutator Methods

- When instance variables are `private` the class must provide methods to access values stored there
  - Typically named `getSomeValue`
  - Referred to as accessor methods
- Must also provide methods to change the values of the `private` instance variable
  - Typically named `setSomeValue`
  - Referred to as mutator methods

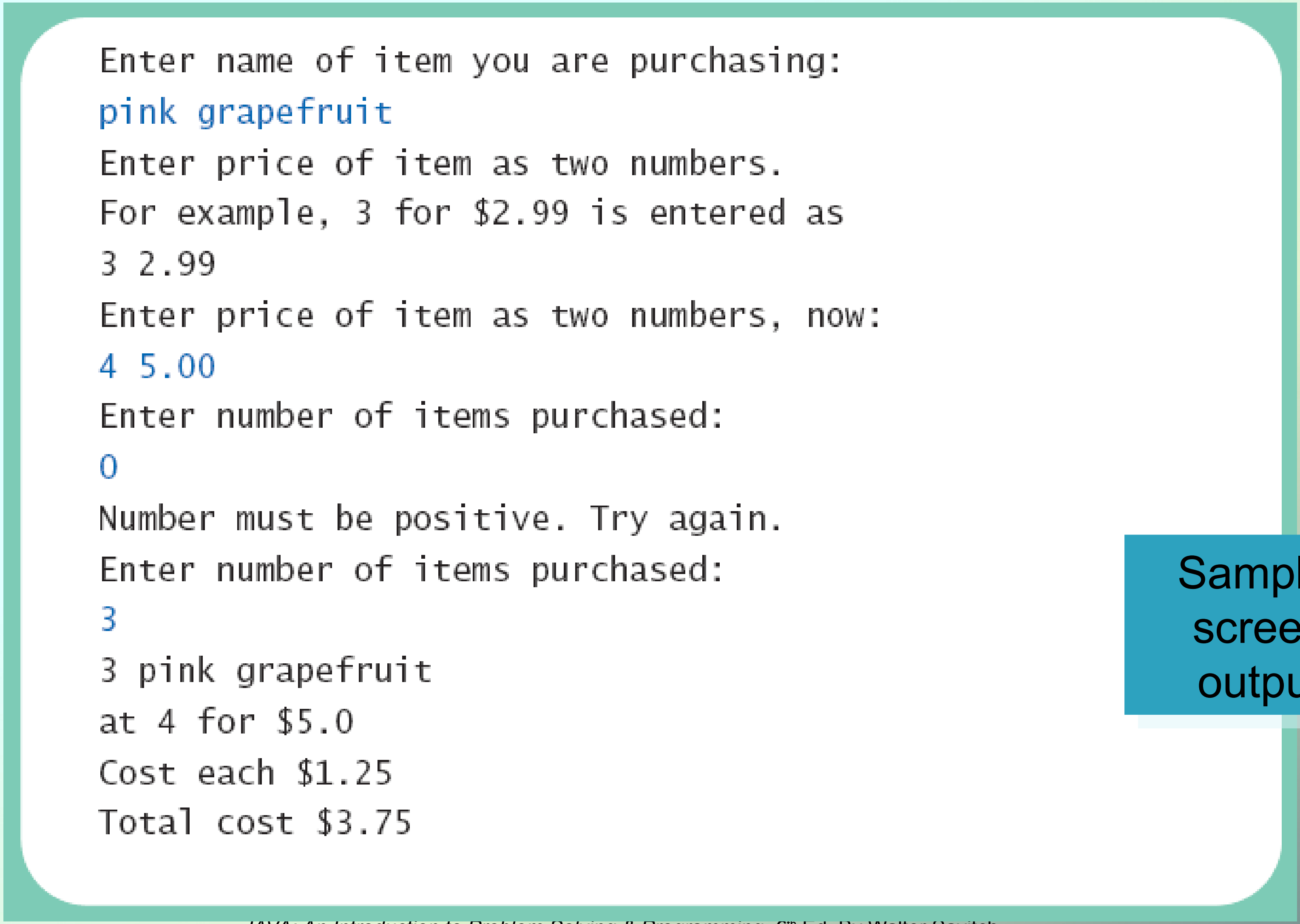
Accessor and Mutator Methods

- Consider an example class with accessor and mutator methods
- Download `SpeciesFourthTry` and `SpeciesFourthTryDemo`
- Note the mutator method
  - `setSpecies`
- Note accessor methods
  - `getName`, `getPopulation`, `getGrowthRate`

Programming Example

- A Purchase class
- Download `Purchase` and `PurchaseDemo`
    - Note use of `private` instance variables
    - Note also how mutator methods check for invalid values

Programming Example



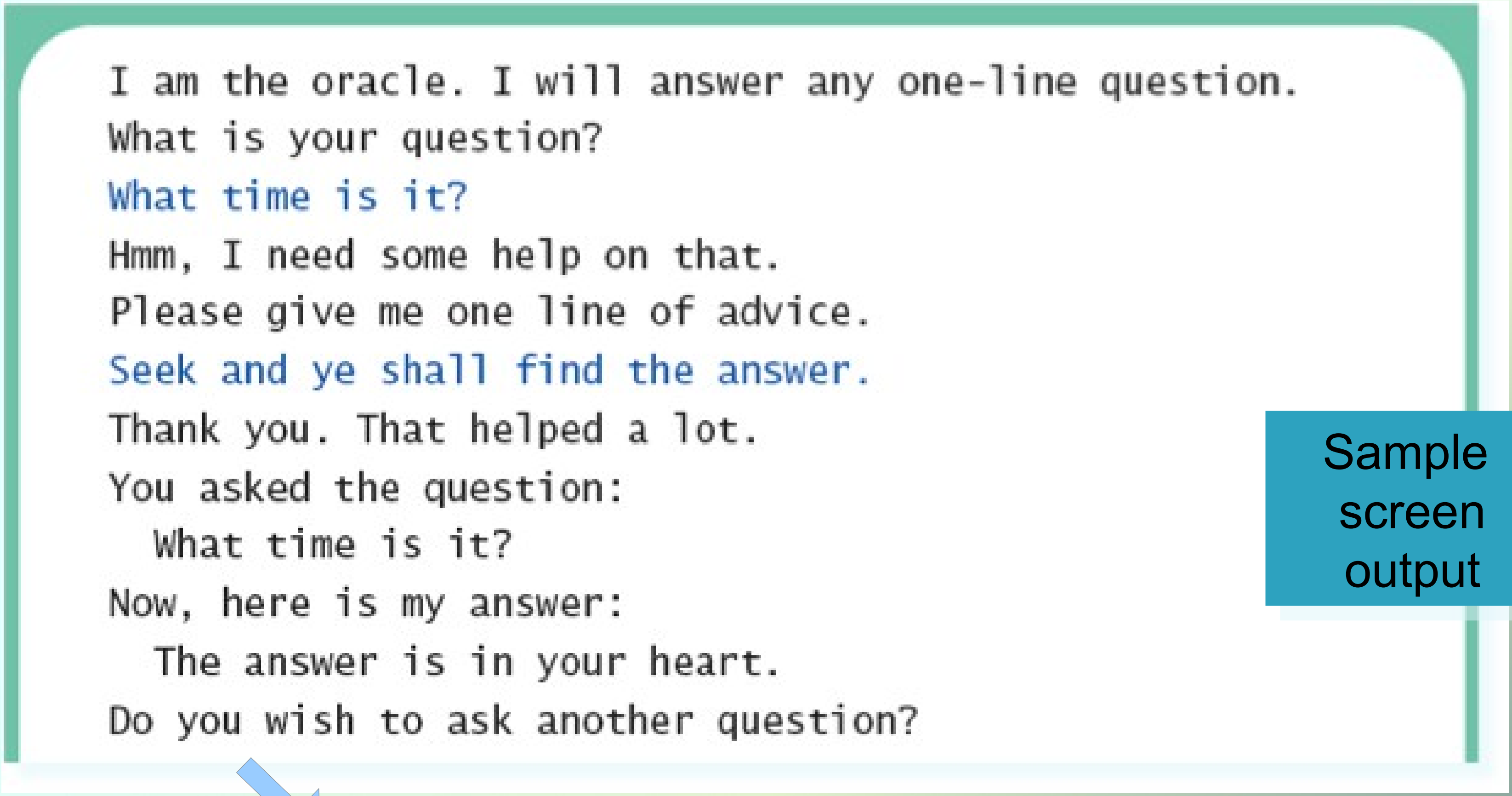
```
Enter name of item you are purchasing:
pink grapefruit
Enter price of item as two numbers.
For example, 3 for $2.99 is entered as
3 2.99
Enter price of item as two numbers, now:
4 5.00
Enter number of items purchased:
0
Number must be positive. Try again.
Enter number of items purchased:
3
3 pink grapefruit
at 4 for $5.0
Cost each $1.25
Total cost $3.75
```

Sample screen output

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
- Download `Oracle` and `OracleDemo`

Methods Calling Methods

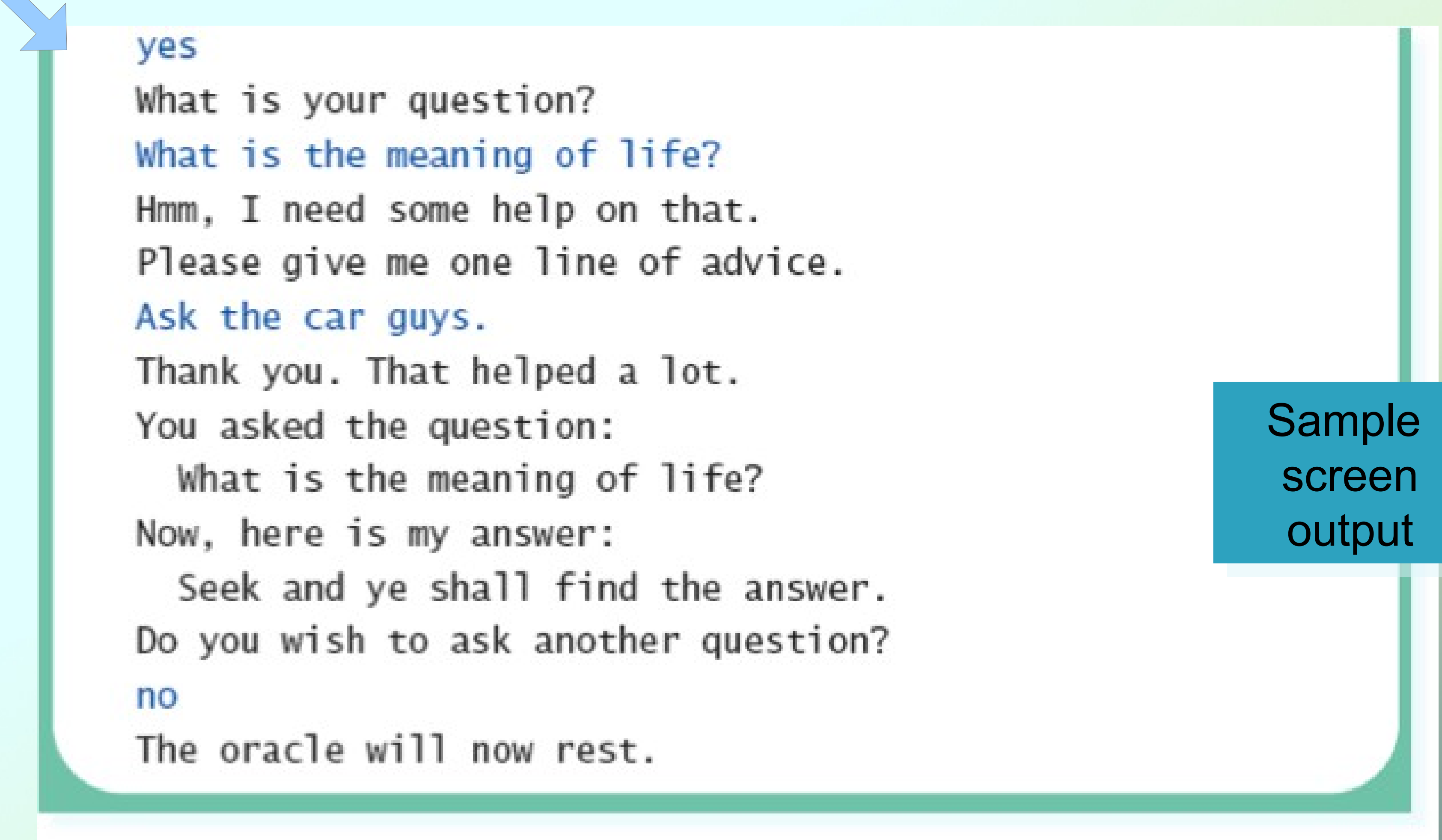


```
I am the oracle. I will answer any one-line question.
What is your question?
What time is it?
Hmm, I need some help on that.
Please give me one line of advice.
Seek and ye shall find the answer.
Thank you. That helped a lot.
You asked the question:
  What time is it?
Now, here is my answer:
  The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

Sample screen output

Cont. next slide

Methods Calling Methods



```
yes
What is your question?
What is the meaning of life?
Hmm, I need some help on that.
Please give me one line of advice.
Ask the car guys.
Thank you. That helped a lot.
You asked the question:
  What is the meaning of life?
Now, here is my answer:
  Seek and ye shall find the answer.
Do you wish to ask another question?
no
The oracle will now rest.
```

Sample screen output

Encapsulation

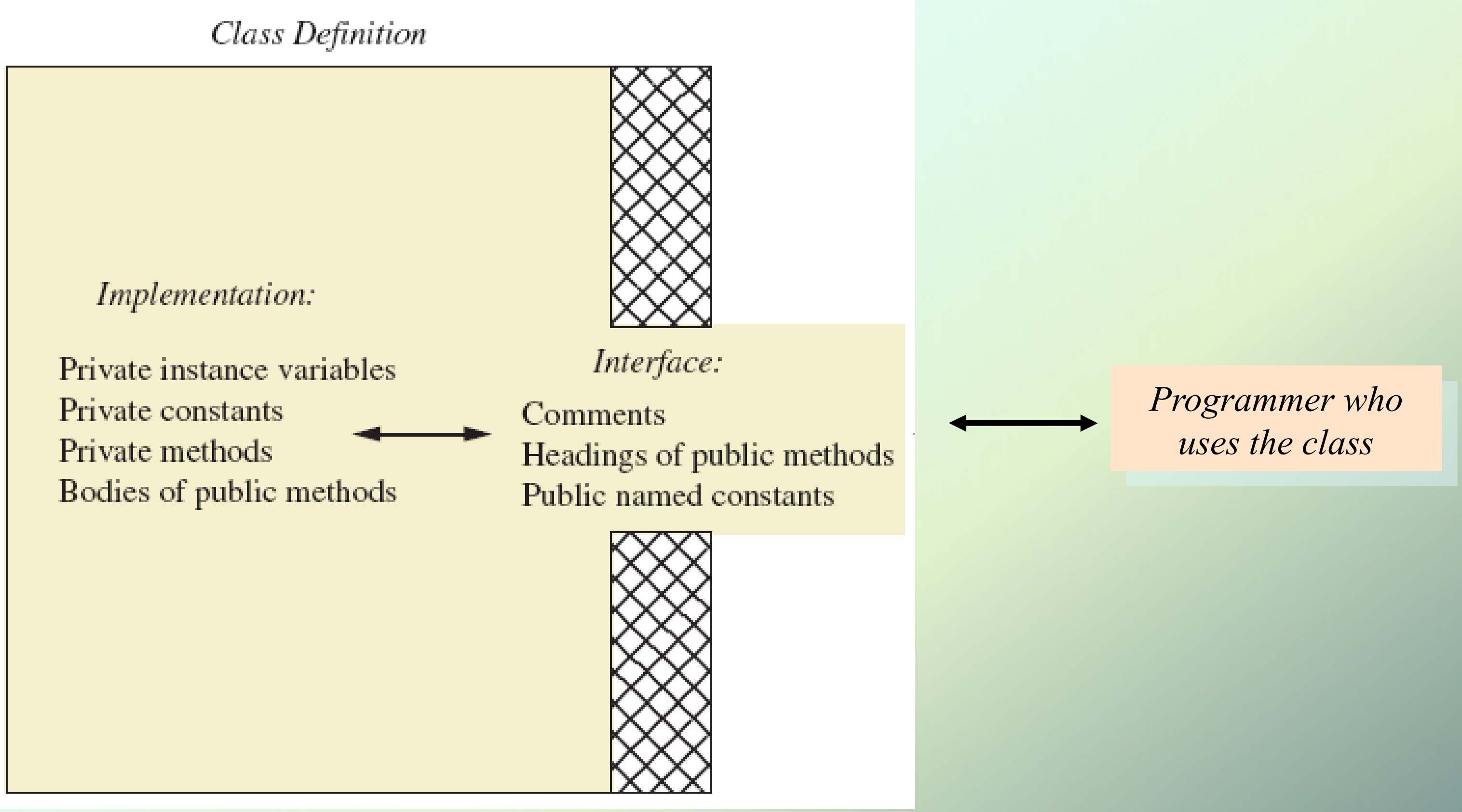
- Consider example of driving a car
  - We see and use break pedal, accelerator pedal, steering wheel – know what they do
  - We do not see mechanical details of how they do their jobs
- Encapsulation divides class definition into
  - Class interface
  - Class implementation

Encapsulation

- A *class interface*
  - Tells <u>what</u> the class does
  - Gives headings for public methods and comments about them
- A *class implementation*
  - Contains private variables
  - Includes definitions of public and private methods

Encapsulation

- Figure 5.3 A well encapsulated class definition



Encapsulation

- 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.

Automatic Documentation `javadoc`

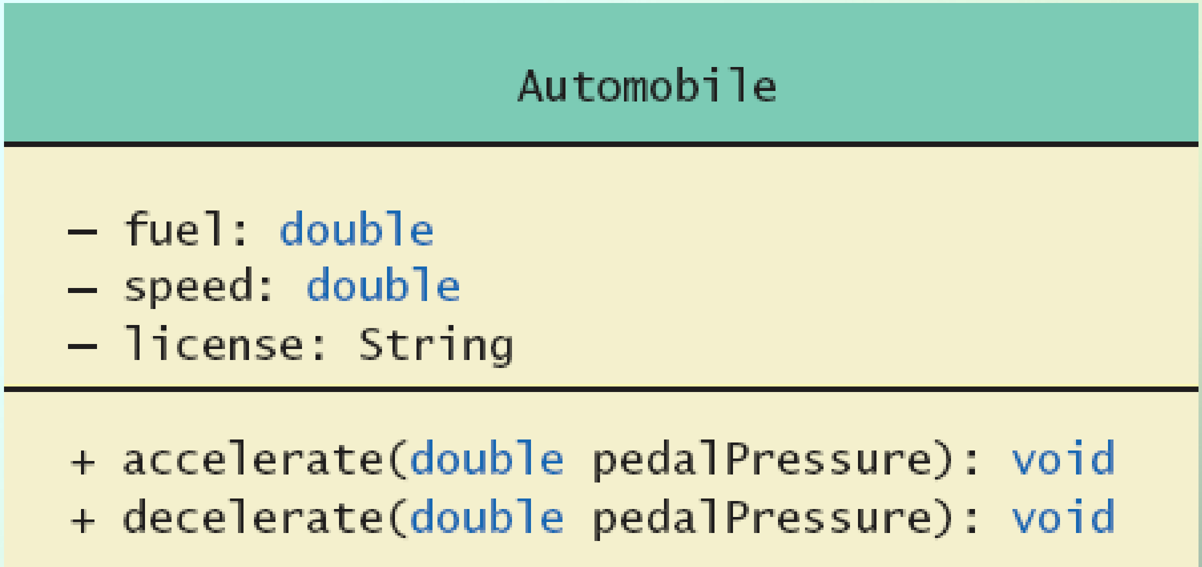
- Generates documentation for class interface
- Comments in source code describing a class/method must be enclosed in `/** */`
  - `@param` for each parameter of a method
  - `@return` for describing what method returns
- Utility `javadoc` will include these comments and headings of public methods
- Output of `javadoc` is HTML format

Automatic Documentation `javadoc`

- Add `javadoc` comments to the `Rectangle` class
- In DrJava
  - Tools -> Javadoc -> Preview Javadoc for Current Document
  - May have to set browser first:
    - Edit -> Preferences -> Resource Locations
    - Enter browser command (firefox,...)

UML Class Diagrams

- Recall Figure 5.2 A class outline as a UML class diagram



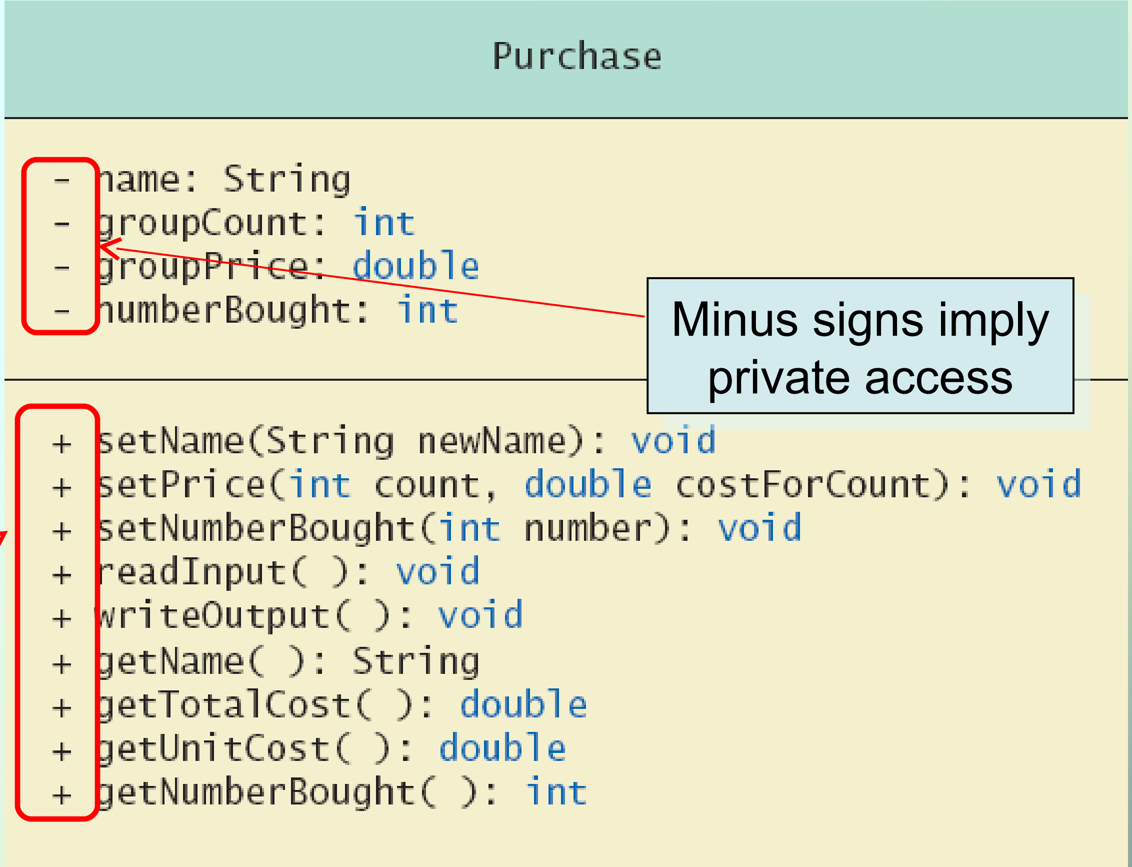
```
Automobile

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

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

UML Class Diagrams

- UML 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

JAVA: An Introduction to Problem Solving & Programming, 6th Ed. By Walter Savitch
ISBN 0132162709 © 2012 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

UML Class Diagrams

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

Variables of a Class Type

- All variables are implemented as a memory location
- Data 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

Variables of a Class Type

- Object itself not stored in the variable
  - Stored elsewhere in memory
  - Variable contains address of where it is stored
- Address called the *reference* to the variable
- A *reference type* variable holds references (memory addresses)
  - This makes memory management of class types more efficient

Variables of a Class Type

- `=` example with primitive type variables (works as expected):

```
int n = 42;
int m = n;
n = 99;
System.out.println(n + " and " + m);
```

- Output:

```
99 and 42
```

Variables of a Class Type

- `=` example with class type variables:

```
SpeciesFourthTry klingonSpecies =
                   new SpeciesFourthTry();
SpeciesFourthTry earthSpecies =
                   new SpeciesFourthTry();
klingonSpecies.setSpecies("Klingon", 10, 15);
earthSpecies.setSpecies("Rhino", 11, 2);
earthSpecies = klingonSpecies;
earthSpecies.setSpecies("Elephant", 100, 12);
System.out.println("earthSpecies:");
earthSpecies.writeOutput();
System.out.println("klingonSpecies:");
klingonSpecies.writeOutput();
```

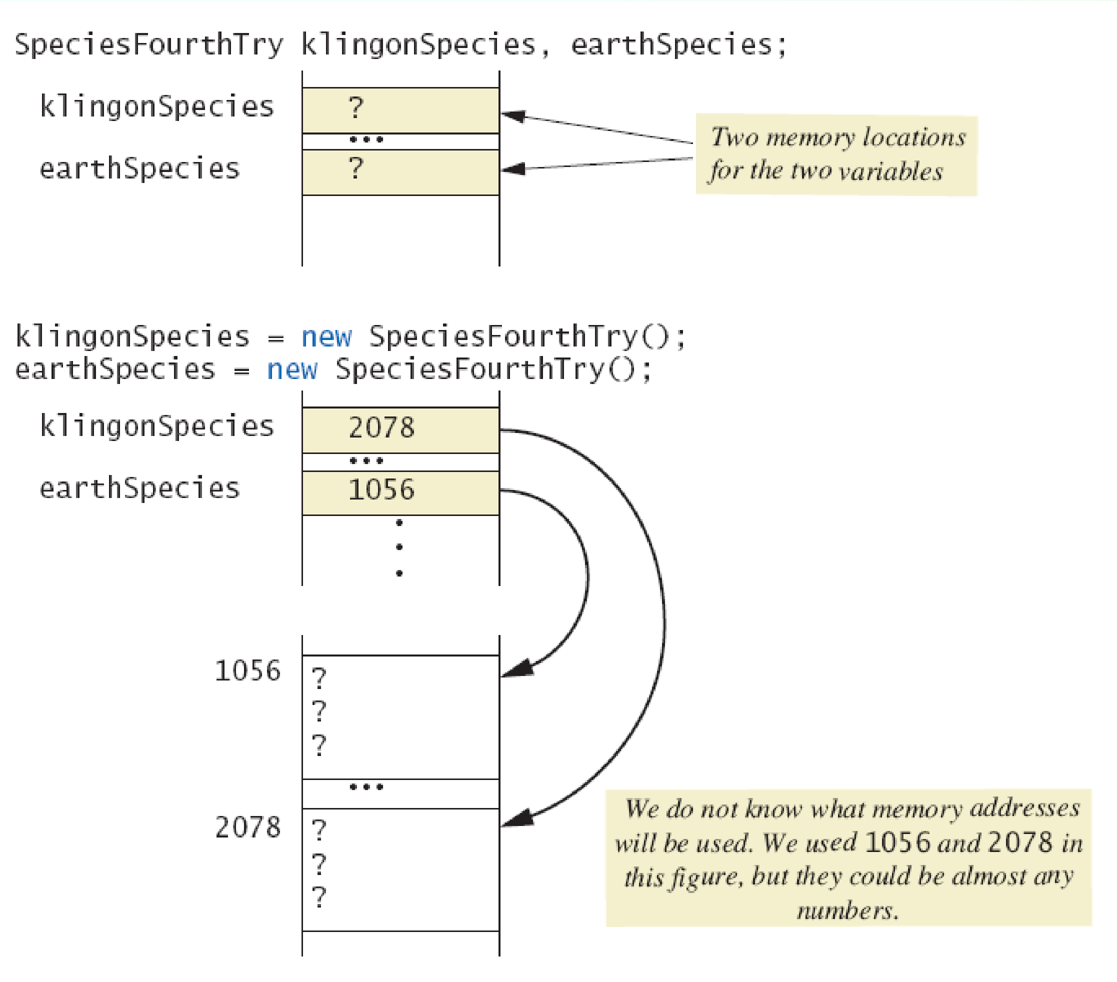
Variables of a Class Type

- `=` example with class type variables (ctd.), output:

```
earthSpecies:
Name = Elephant
Population = 100
Growth rate = 12%
klingonSpecies:
Name = Elephant
Population = 100
Growth rate = 12%
```

Variables of a Class Type

- Behavior of class variables



```
SpeciesFourthTry klingonSpecies, earthSpecies;
```

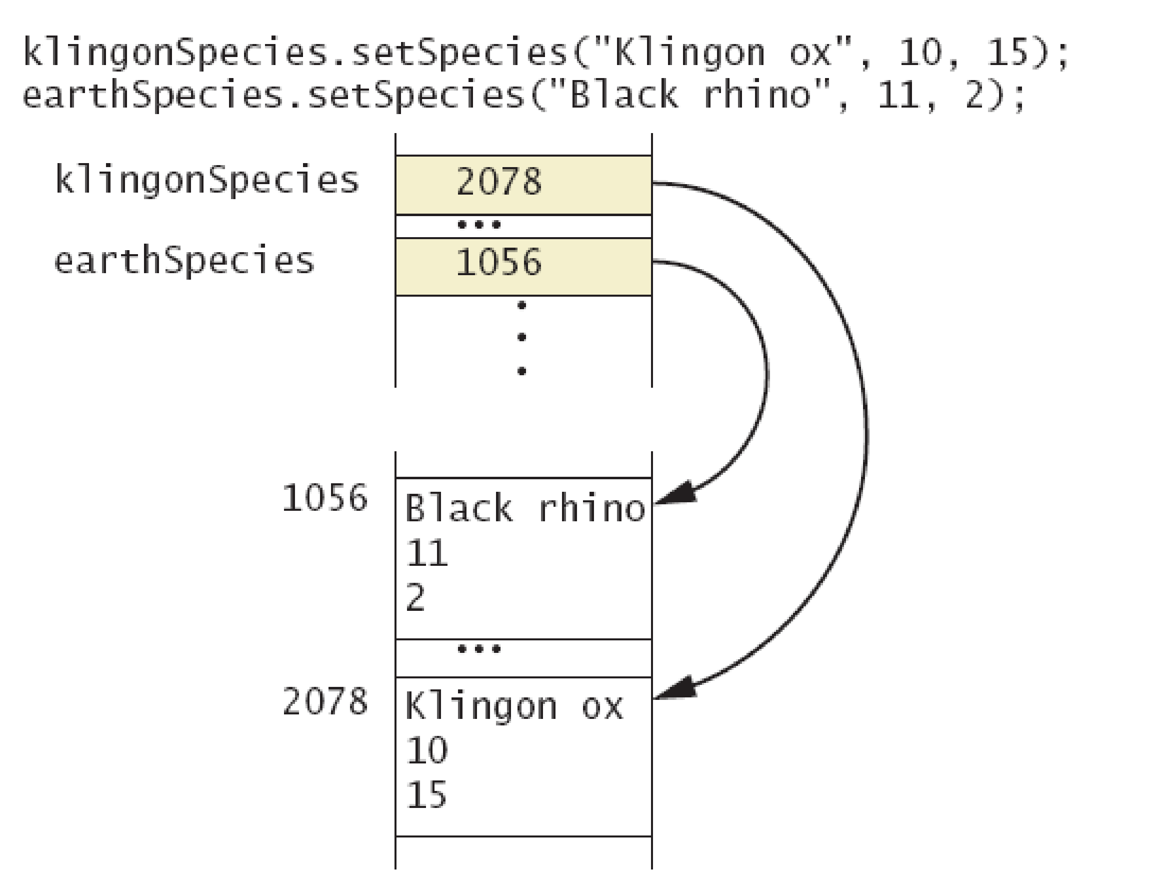
Two memory locations for the two variables

```
klingonSpecies = new SpeciesFourthTry();
earthSpecies = new SpeciesFourthTry();
```

We do not know what memory addresses will be used. We used 1056 *and* 2078 *in this figure, but they could be almost any numbers.*

Variables of a Class Type

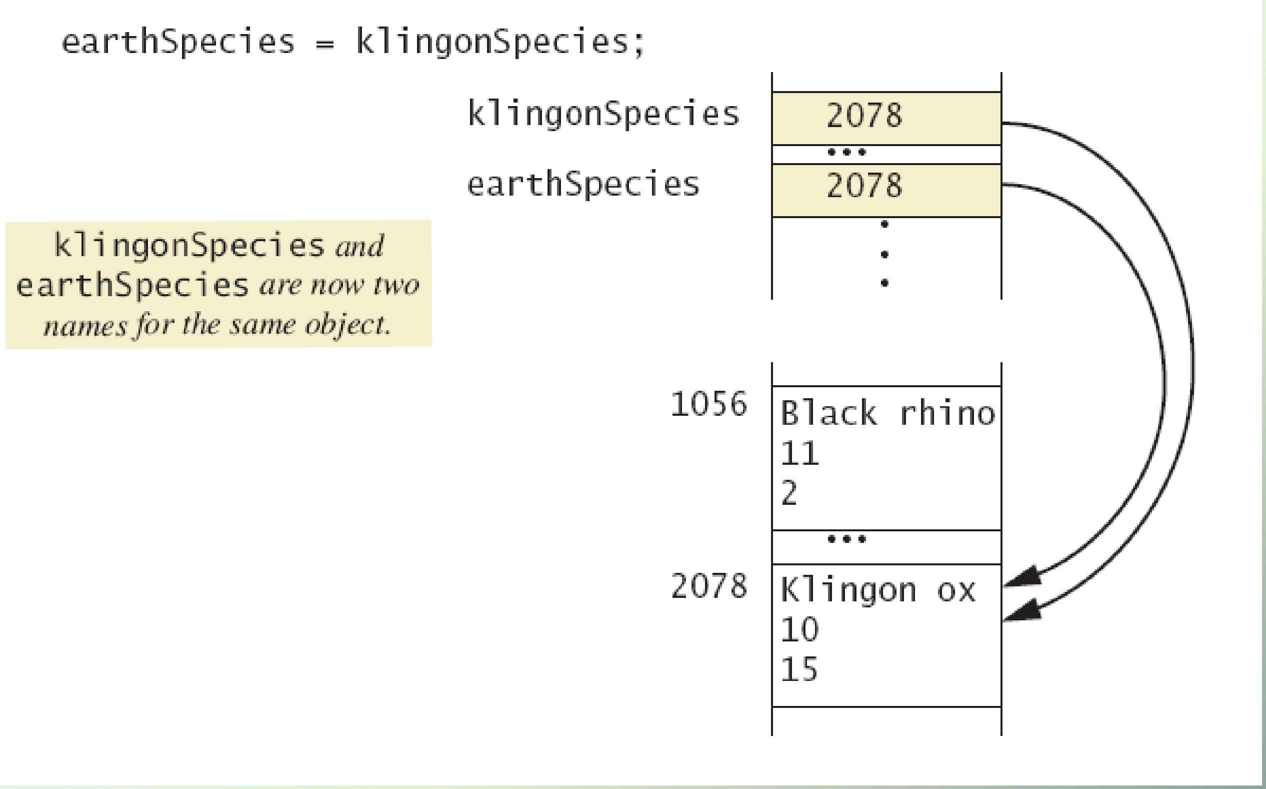
- Behavior of class variables



```
klingonSpecies.setSpecies("Klingon ox", 10, 15);
earthSpecies.setSpecies("Black rhino", 11, 2);
```

Variables of a Class Type

- Behavior of class variables

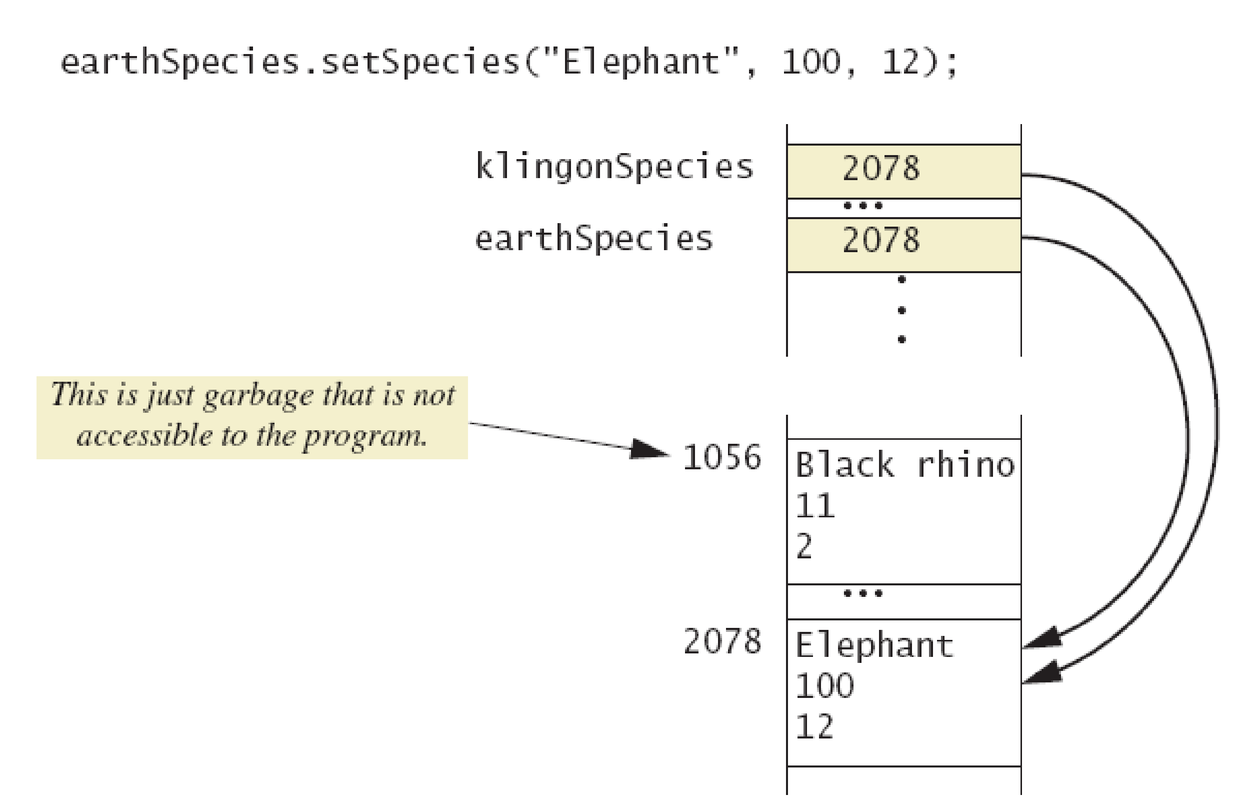


```
earthSpecies = klingonSpecies;
```

klingonSpecies *and* earthSpecies *are now two names for the same object.*

Variables of a Class Type

- Behavior of class variables

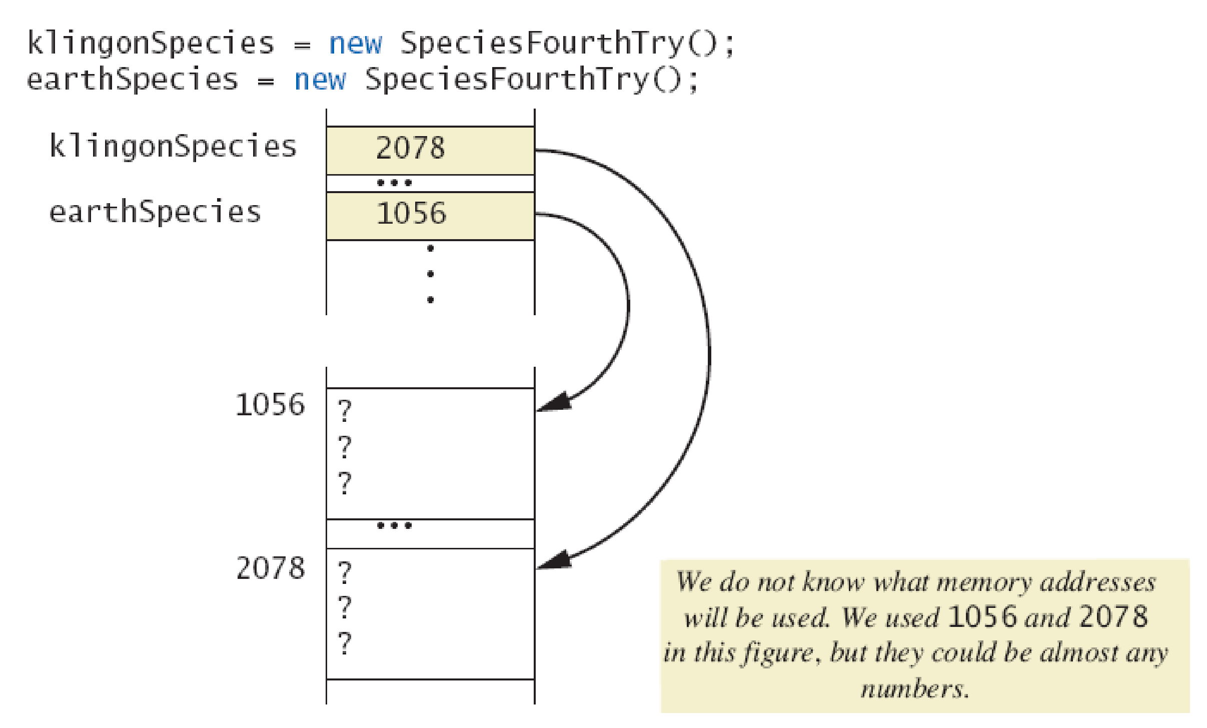


```
earthSpecies.setSpecies("Elephant", 100, 12);
```

This is just garbage that is not accessible to the program.

Variables of a Class Type

- Dangers of using == with objects

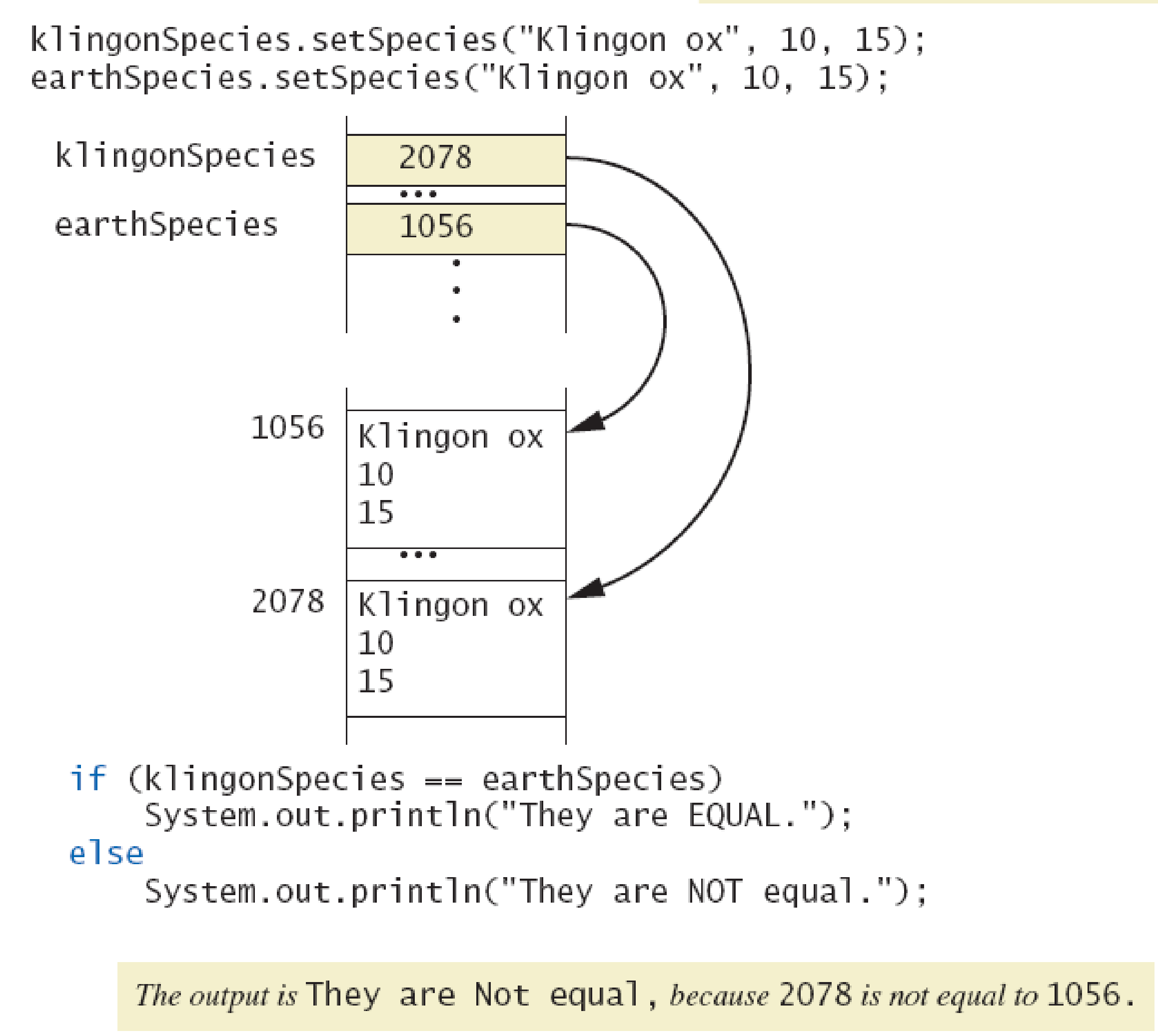


```
klingonSpecies = new SpeciesFourthTry();
earthSpecies = new SpeciesFourthTry();
```

We do not know what memory addresses will be used. We used 1056 *and* 2078 *in this figure, but they could be almost any numbers.*

Variables of a Class Type

- Dangers of using == with objects



```
klingonSpecies.setSpecies("Klingon ox", 10, 15);
earthSpecies.setSpecies("Klingon ox", 10, 15);
```

```
if (klingonSpecies == earthSpecies)
    System.out.println("They are EQUAL.");
else
    System.out.println("They are NOT equal.");
```

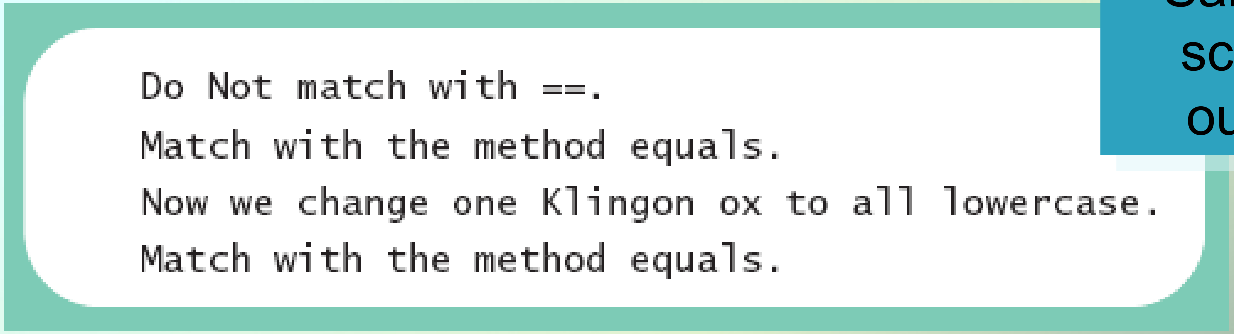
The output is They are Not equal, *because* 2078 *is not equal to* 1056.

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
- Download `Species`
- The `equals` for this class method used same way as `equals` method for `String`

Demonstrating an **equals** Method

- Download **SpeciesEqualsDemo**
- Note difference in the two comparison methods **==** versus **.equals( )**

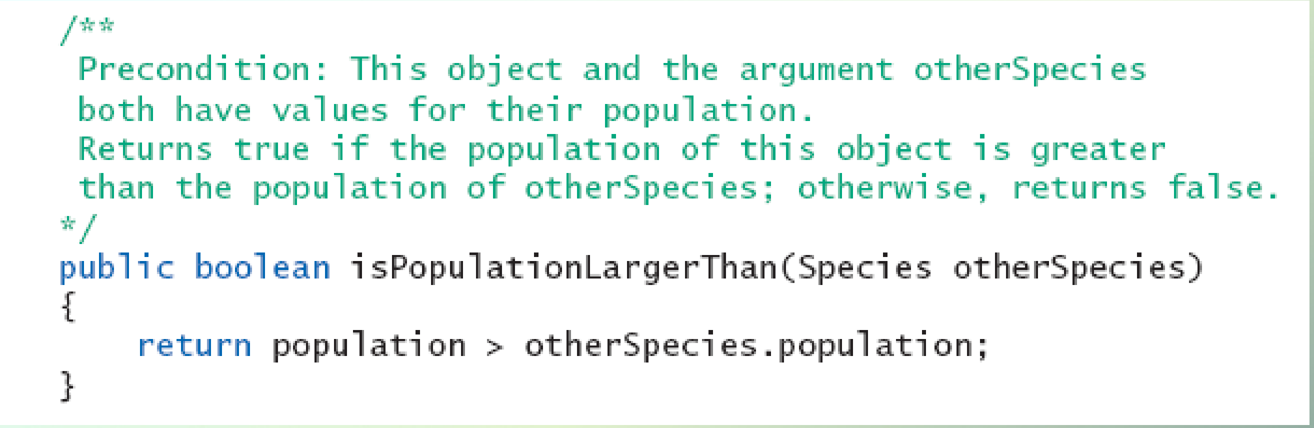


```
Do Not match with ==.
Match with the method equals.
Now we change one Klingon ox to all lowercase.
Match with the method equals.
```

Sample screen output

Boolean-Valued Methods

- Methods can return a value of type **`boolean`**
- Use a **`boolean`** value in the **`return`** statement
- Add this method to the **`Species`** class



```
/**
 Precondition: This object and the argument otherSpecies
 both have values for their population.
 Returns true if the population of this object is greater
 than the population of otherSpecies; otherwise, returns false.
*/
public boolean isPopulationLargerThan(Species otherSpecies)
{
    return population > otherSpecies.population;
}
```

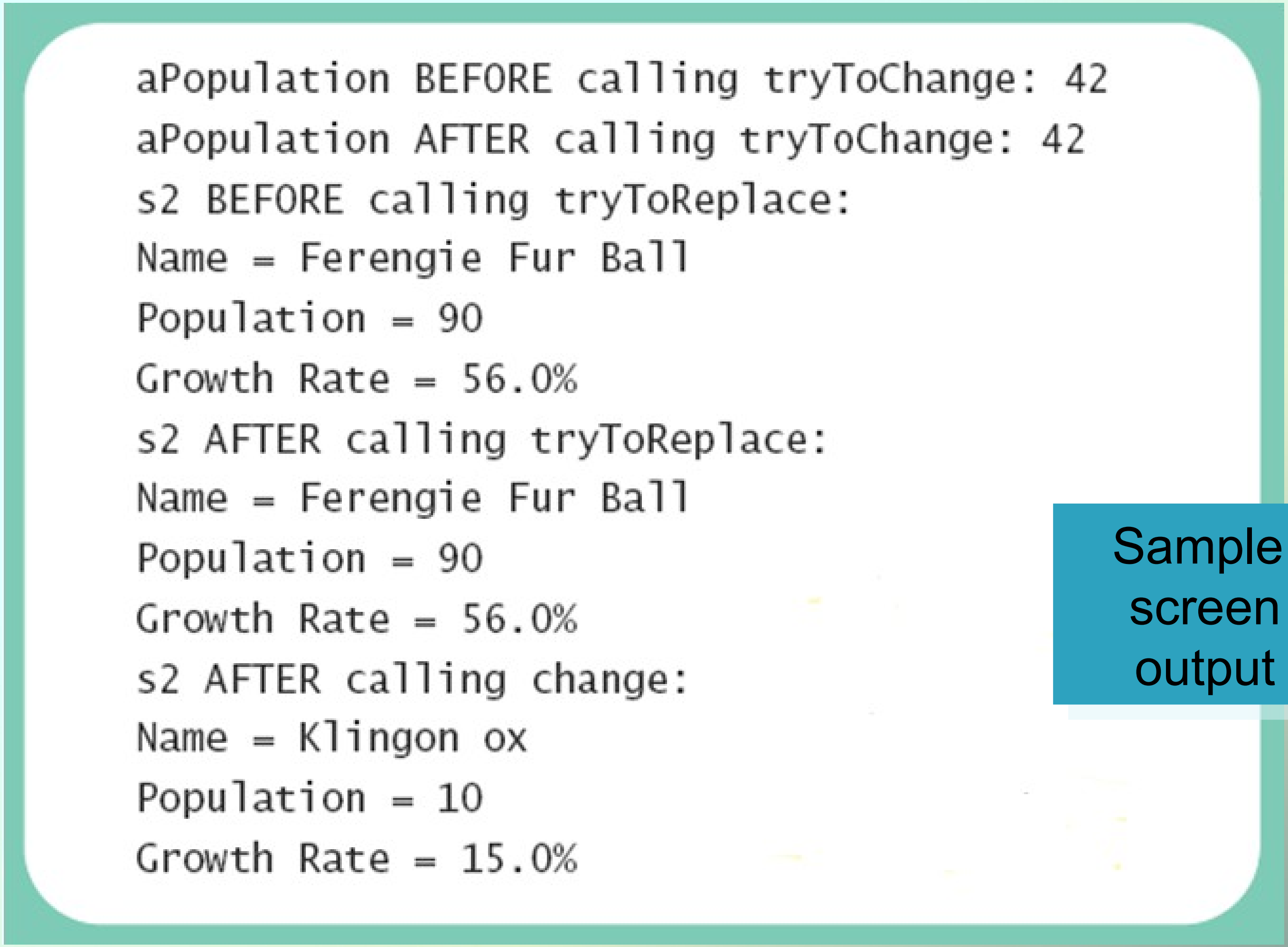
Parameters of a Class Type

- When assignment operator used with objects of class type
  - Only memory address is copied
- Similar to use of parameter 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

Programming Example

- Download `DemoSpecies`
  - Note different parameter types and results
- Download `ParametersDemo`
  - Parameters of a class type versus parameters of a primitive type

Programming Example



```
aPopulation BEFORE calling tryToChange: 42
aPopulation AFTER calling tryToChange: 42
s2 BEFORE calling tryToReplace:
Name = Ferengie Fur Ball
Population = 90
Growth Rate = 56.0%
s2 AFTER calling tryToReplace:
Name = Ferengie Fur Ball
Population = 90
Growth Rate = 56.0%
s2 AFTER calling change:
Name = Klingon ox
Population = 10
Growth Rate = 15.0%
```

Sample screen output