6~7. Defining Classes and Methods

[ECE20016/ITP20003] Java Programming

Agenda

- Class and Method Definitions
- Information Hiding and Encapsulation
- Objects and References

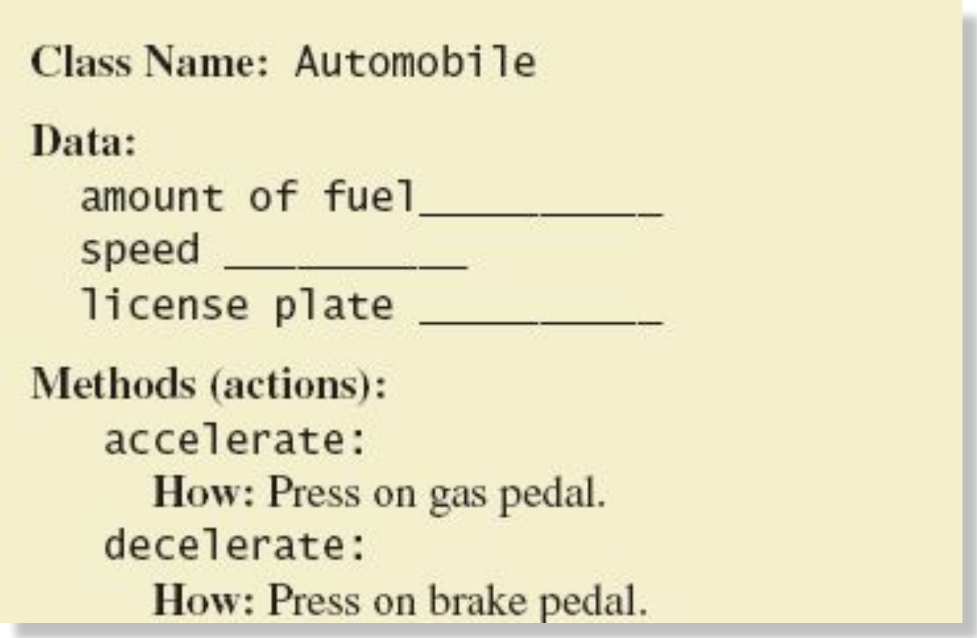
Class and Method Definitions

- Java program consists of objects
  - Objects of class types
  - Objects that interact with one another

- Program objects can represent
  - Objects in real world
  - Abstractions

Class and Method Definitions

Ex) A class as a blueprint



Class Name: Automobile

Data:

amount of fuel__________
speed __________
license plate __________

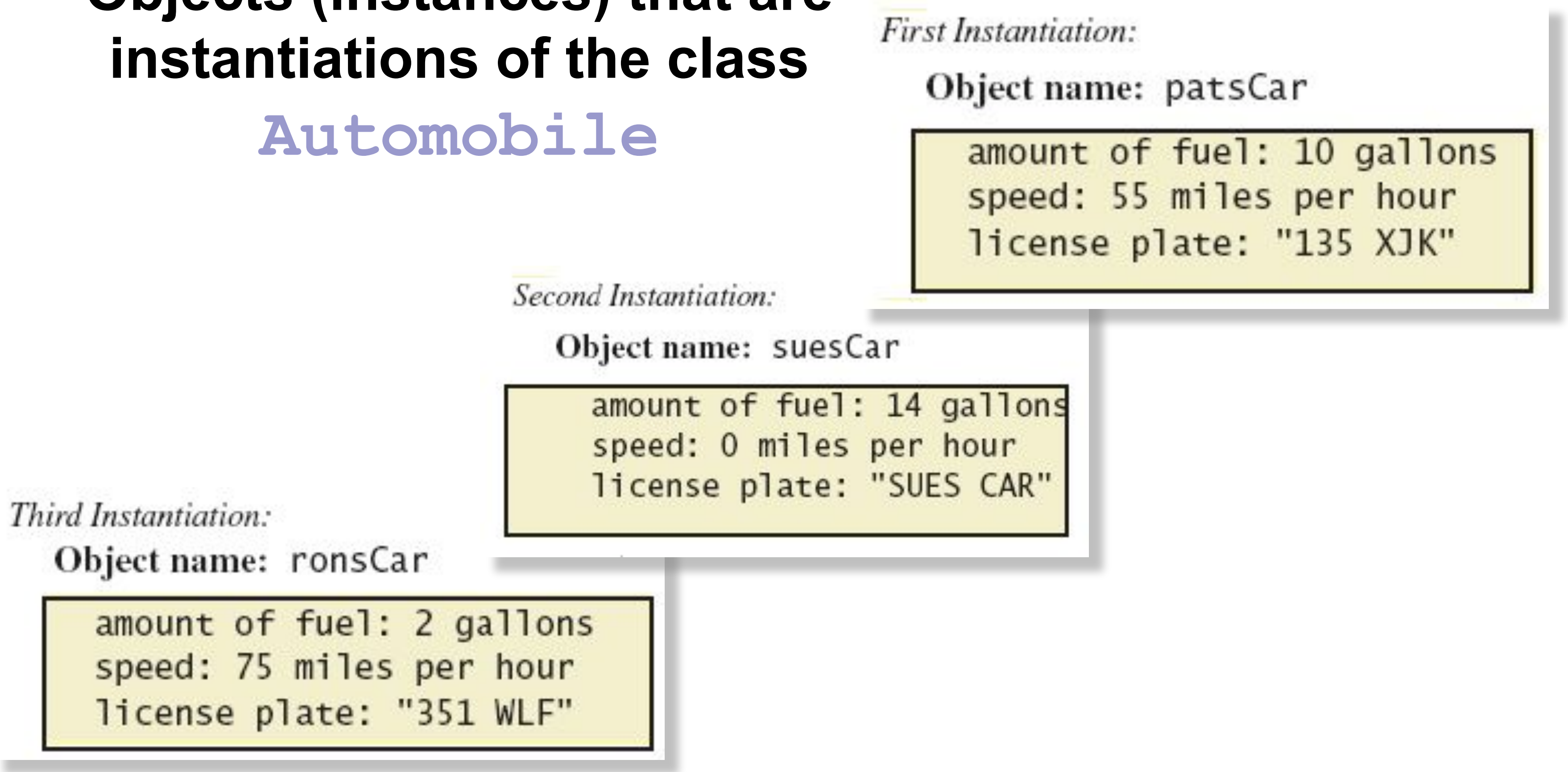
Methods (actions):

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

Class and Method Definitions

Objects (instances) that are instantiations of the class

`Automobile`



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:

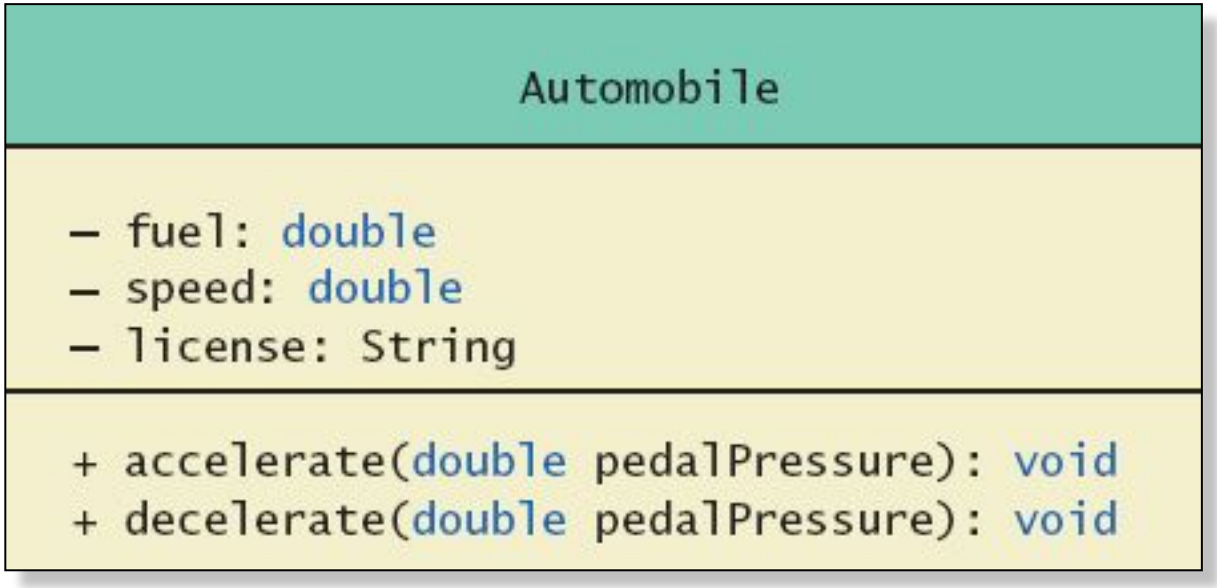
Object name: `ronsCar`

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

Class and Method Definitions

- A class outline as a UML class diagram

  cf. UML: Unified Modeling Language

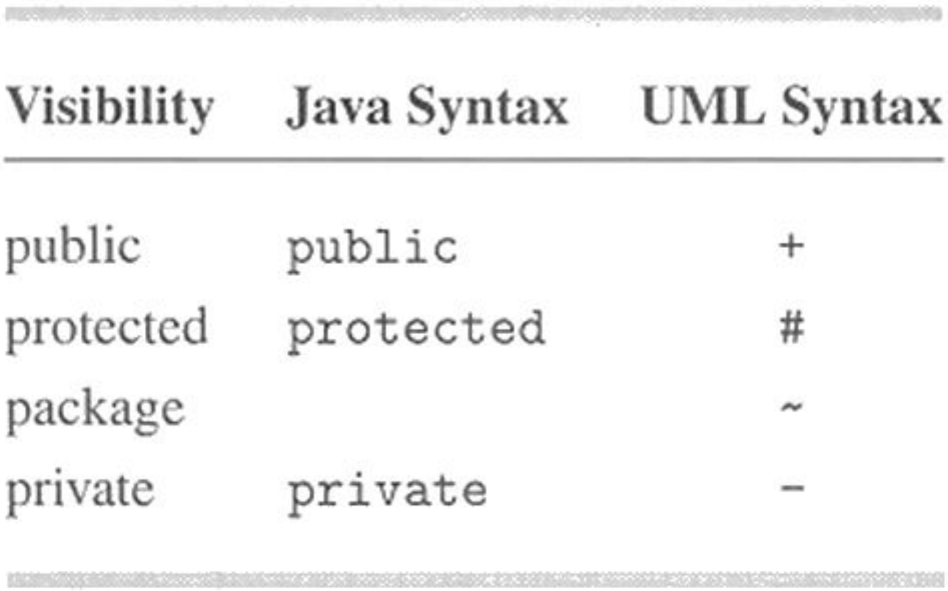


Automobile

− fuel: double
− speed: double
− license: String

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

UML Visibility



Visibility	Java Syntax	UML Syntax
public	public	+
protected	protected	#
package		~
private	private	-

Class Files and Separate Compilation

- Each Java class definition usually in a file
  - The filename should be *ClassName*.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

```
public class Dog
{
        public String name;
        public String breed;
        public int age;

        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();
        }

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

Dog class and Instance Variables

- The Dog class has
  - Three pieces of data (instance variables)
  - Two behaviors (methods)

- Each instance of this type has its own copies of the data items.

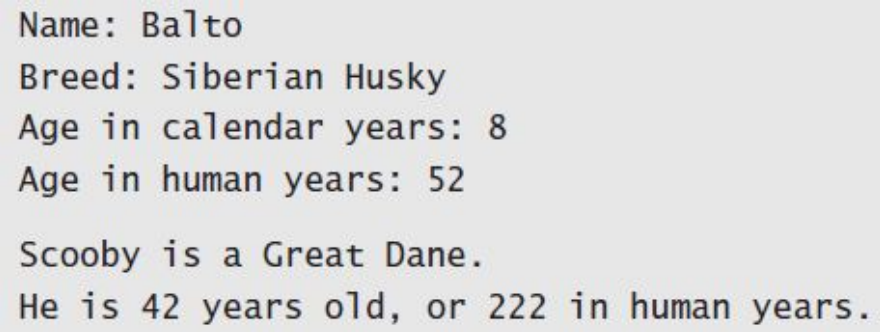
- Use of public
  - No restrictions on how variables used
  - Can be replaced with private

Java Access Modifiers

	public	protected	default	private
same class	O	O	O	O
same package	O	O	O	
derived classes	O	O		
other	O			

DogDemo

```
public class DogDemo
{
    public static void main(String[] args)
    {
        Dog balto = new Dog();
        balto.name = "Balto";
        balto.age = 8;
        balto.breed = "Siberian Husky";
        balto.writeOutput();

        Dog scooby = new Dog();
        scooby.name = "Scooby";
        scooby.age = 42;
        scooby.breed = "Great Dane";
        System.out.println(scooby.name + " is a " + scooby.breed + ".");
        System.out.print("He is " + scooby.age + " years old, or ");
        int humanYears = scooby.getAgeInHumanYears();
        System.out.println(humanYears + " in human years.");
    }
}
```



```
Name: Balto
Breed: Siberian Husky
Age in calendar years: 8
Age in human years: 52

Scooby is a Great Dane.
He is 42 years old, or 222 in human years.
```

Methods

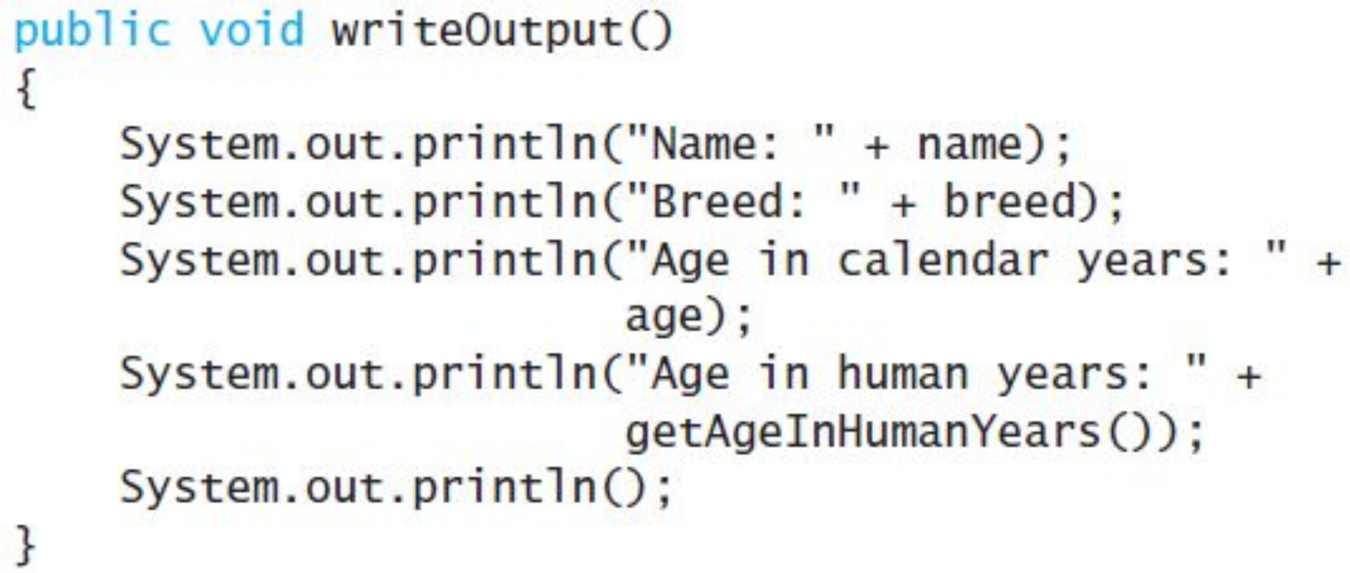
- When you use a method you "invoke" or "call" it
- Two kinds of Java methods
  - Return a single item
    - Use anywhere a value can be used
  - Perform some other action – a void method
    - Resulting statement performs the action defined by the method
- The method *main*

  public static void main(String[] args)

  - A void method
  - Invoked by the system

Defining Methods

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



```java
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();
}
```

Defining Methods

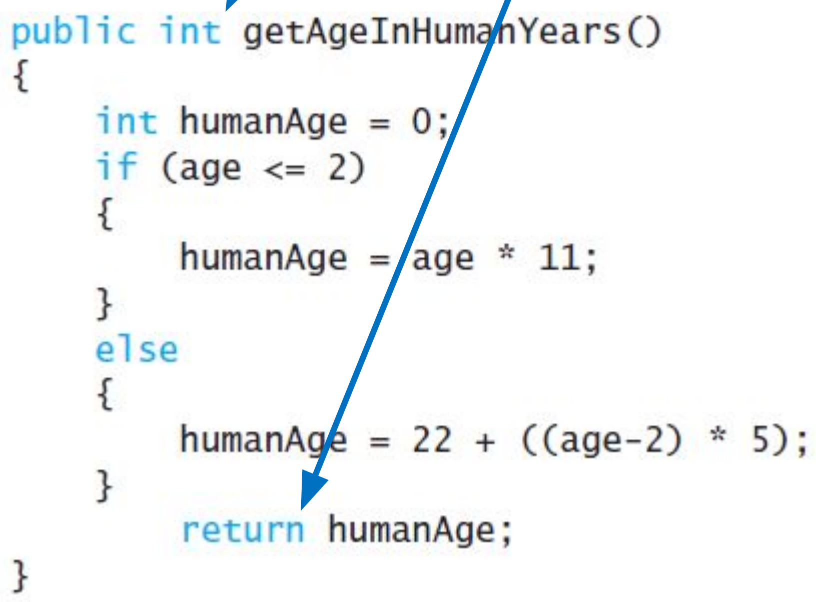
- Most method definitions we will see as public
  - Void method does not return a value

- Head
  - Method name + parameters

- Body
  - Enclosed in braces { }
  - Think of method as defining an action to be taken

Methods That Return a Value

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



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

The Keyword *this*

- Referring to instance variables outside the class

  Syntax) *ObjectName.VariableName*

- Referring to instance variables inside the class
  - Use *VariableName* alone
    - The object (unnamed) is understood to be there.

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

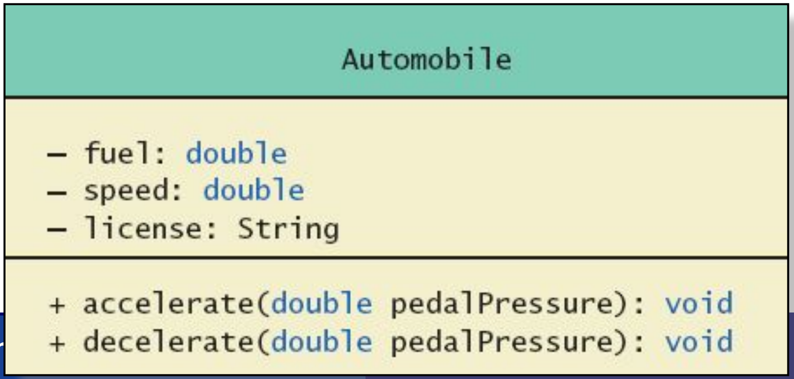
  Ex) this.name = keyboard.nextLine();

  - The keyword *this* stands for the receiving object

Class and Object

- Class definition
  - Similar to drawing a blueprint.

```
public class Automobile{
      private double fuel;
      private double speed;
      private String license;
      public void accelerate(…) {
         ...
      }
      public void deaccelerate(…) {
         ...
      }
      …
}
```



Automobile

− fuel: double
− speed: double
− license: String

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

- Object creation
  - Similar to making a product.

```
Automobile suesCar = new Automobile();
/*
    statements to set attributes of suesCar
*/
```

suesCar
-fuel = 14
-speed = 0
-license = “SUES CAR”
+accelerate(…): void
+deaccelerate(…): void

- In suesCar.accelerate(…),
  - fuel means suesCar.fuel
  - speed means suesCar.speed
  - this means suesCar

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*

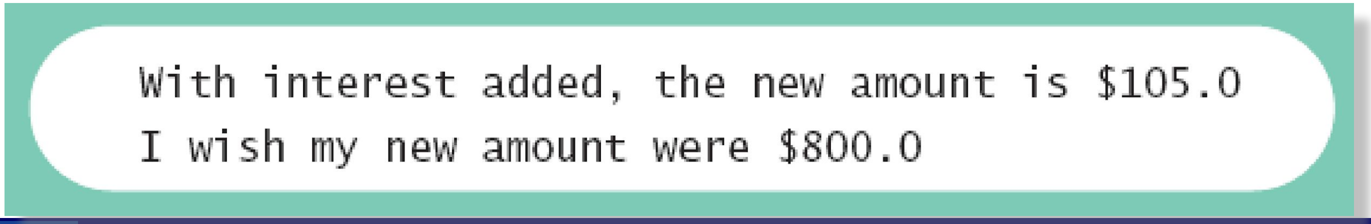
- Local variables having the same name and declared in different methods are different variables

BankAccount

```
public class BankAccount
{
    public double amount;
    public double rate;
    public void showNewBalance ()
    {
        double newAmount = amount + (rate / 100.0) * amount;
        System.out.println ("With interest added, the new amount is $“
                                                + newAmount);
    }
}
```

LocalVariablesDemoProgram

```
public class LocalVariablesDemoProgram
{
    public static void main (String [] args)
    {
        BankAccount myAccount = new BankAccount ();
        myAccount.amount = 100.00;
        myAccount.rate = 5;
        double newAmount = 800.00;
        myAccount.showNewBalance ();
        System.out.println ("I wish my new amount were $“
                                                        + newAmount);
    }
}
```



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

Blocks

- Blocks or compound statements
  - Statements enclosed in braces { }

- When you declare a variable within a compound statement
  - 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

```
class SpeciesSecondTry {
    …
    public int predictPopulation (int years)
      {
        int result = 0;
        double populationAmount = population;
        int count = years;
        while ((count > 0) && (populationAmount > 0))
        {
            populationAmount = (populationAmount +
                    (growthRate / 100) * populationAmount);
            count - - ;
        }
        if (populationAmount > 0)
            result = (int) populationAmount;
        return result;
      }
}
```

Parameters of Primitive Type

- 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
```

Agenda

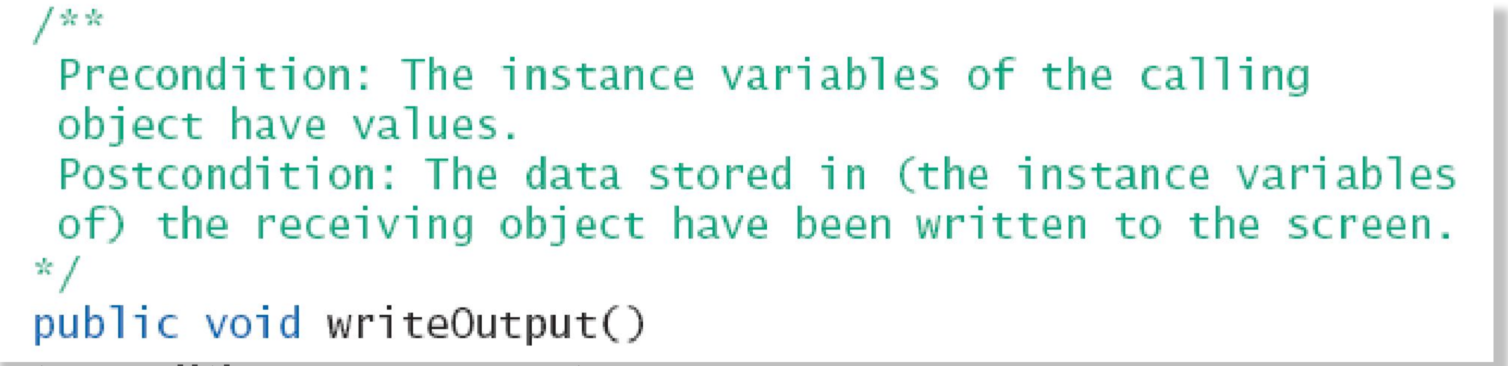
- Class and Method Definitions
- **Information Hiding and Encapsulation**
- Objects and References

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*

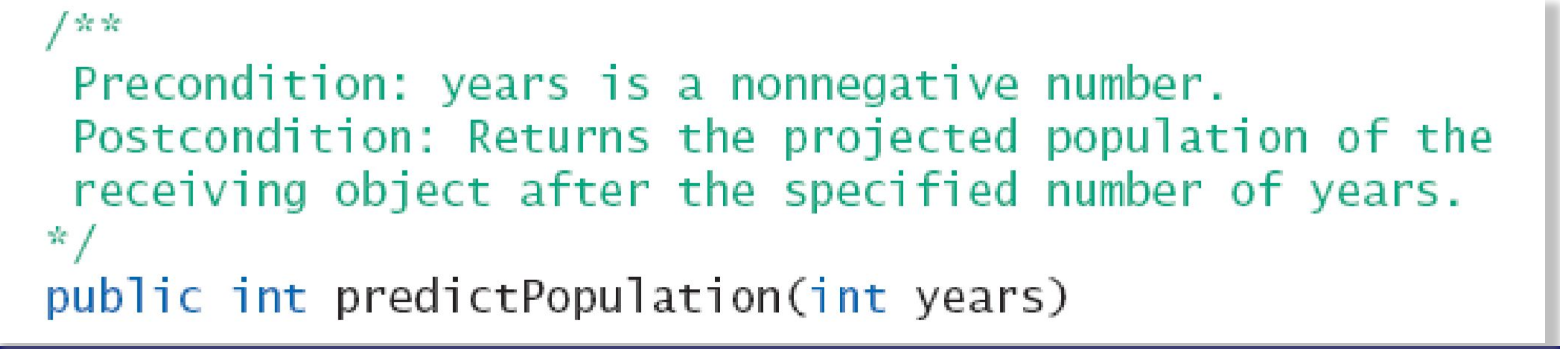
Pre- and Postcondition Comments

- Precondition comment
  - States conditions that must be true before method is invoked



```
/**
 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()
```

- Postcondition comment
  - Tells what will be true after method executed



```
/**
 Precondition: years is a nonnegative number.
 Postcondition: Returns the projected population of the
 receiving object after the specified number of years.
*/
public int predictPopulation(int years)
```

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*

Programming Example

```
public class Rectangle
{
    private int width;
    private int height;
    private int area;
    public void setDimensions (int newWidth, int newHeight)
    {
        width = newWidth;
        height = newHeight;
        area = width * height;
    }
    public int getArea ()
    {
        return area;
    }
}
```

➔ Statement such as “box.width = 6;” is illegal.

Programming Example

```
public class Rectangle2
{
    private int width;
    private int height;

    public void setDimensions (int newWidth, int newHeight)
    {
        width = newWidth;
        height = newHeight;
    }

    public int getArea ()
    {
        return width * height;
    }
}
```

- setDimensions() method is the only way the width and height may be altered outside the class.

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

Accessor and Mutator Methods

- Consider an example class (Projector) with accessor and mutator methods
  - Note the mutator method
    - setTemperature(int temperature)
  - Note accessor methods
    - getTemperature(), getDescription();

Methods Calling Methods

- A method body may call any other method
  - If the invoked method is within the same class, object name can be omitted.

- In Projector class

```
public void turnOn() {
        System.out.println("My project is turing on!");
        getLampTemparature();
}

private void getLampTemparature() {
        System.out.println("My projector temparature is :" + mLampTemparature);
}
```

Encapsulation

- Consider example of driving a car
  - We see and use brake 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

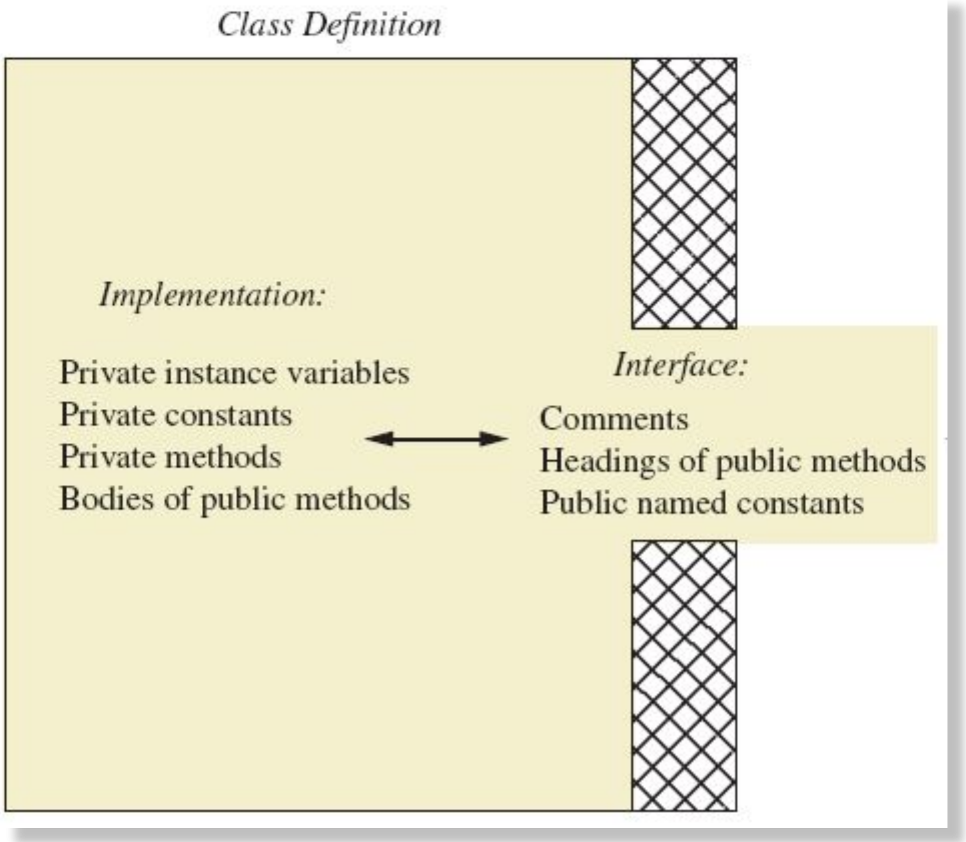
Encapsulation

- A *class interface*
  - Tells what 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

- A well encapsulated class definition



Programmer who uses the class

Encapsulation

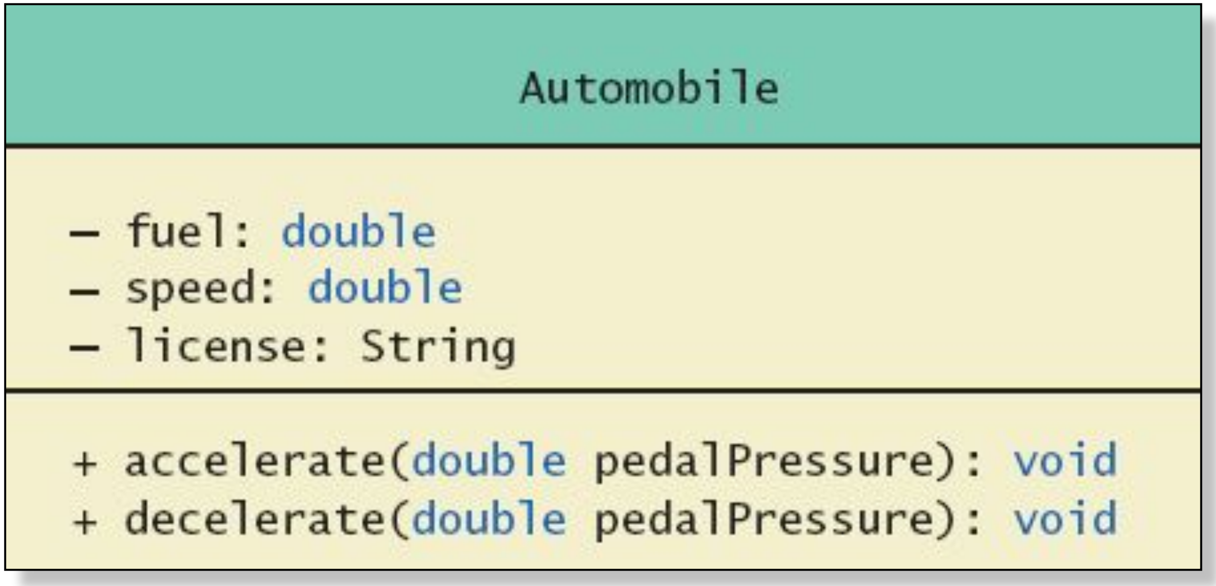
- Preface class definition with comment on how to use class
- Declare all instance variables in the class as private.
- Provide public accessor methods to retrieve data
- Provide public methods manipulating data
  - Such methods could include public mutator methods.
- Place a comment before each public 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 must be enclosed in /** */
- Utility *javadoc* will include
  - These comments
  - Headings of public methods
- Output of *javadoc* is HTML format

UML Class Diagrams

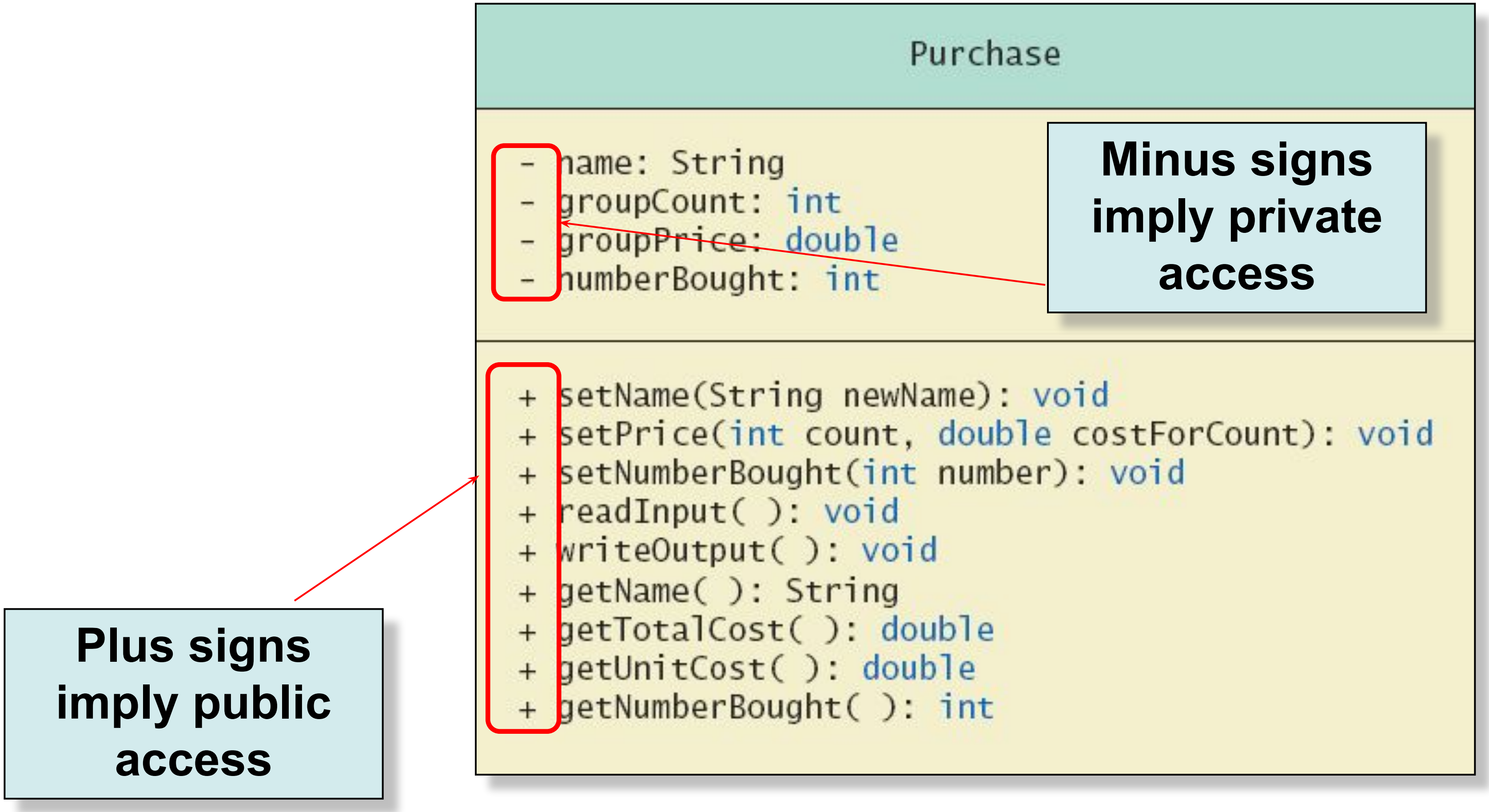
- A class outline as a UML class diagram



```
Automobile

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

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

UML Class Diagrams

- 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

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

Agenda

- Class and Method Definitions
- Information Hiding and Encapsulation
- **Objects and References**

Variables of a Class Type

- All variables are implemented as a memory location

- Variable of *primitive type* contains data in the memory location assigned to the variable

  Ex) int i;

- Variable of *class type* contains memory address of object named by the variable

  Ex) MyClass obj = new MyClass();

Variables of a Class Type

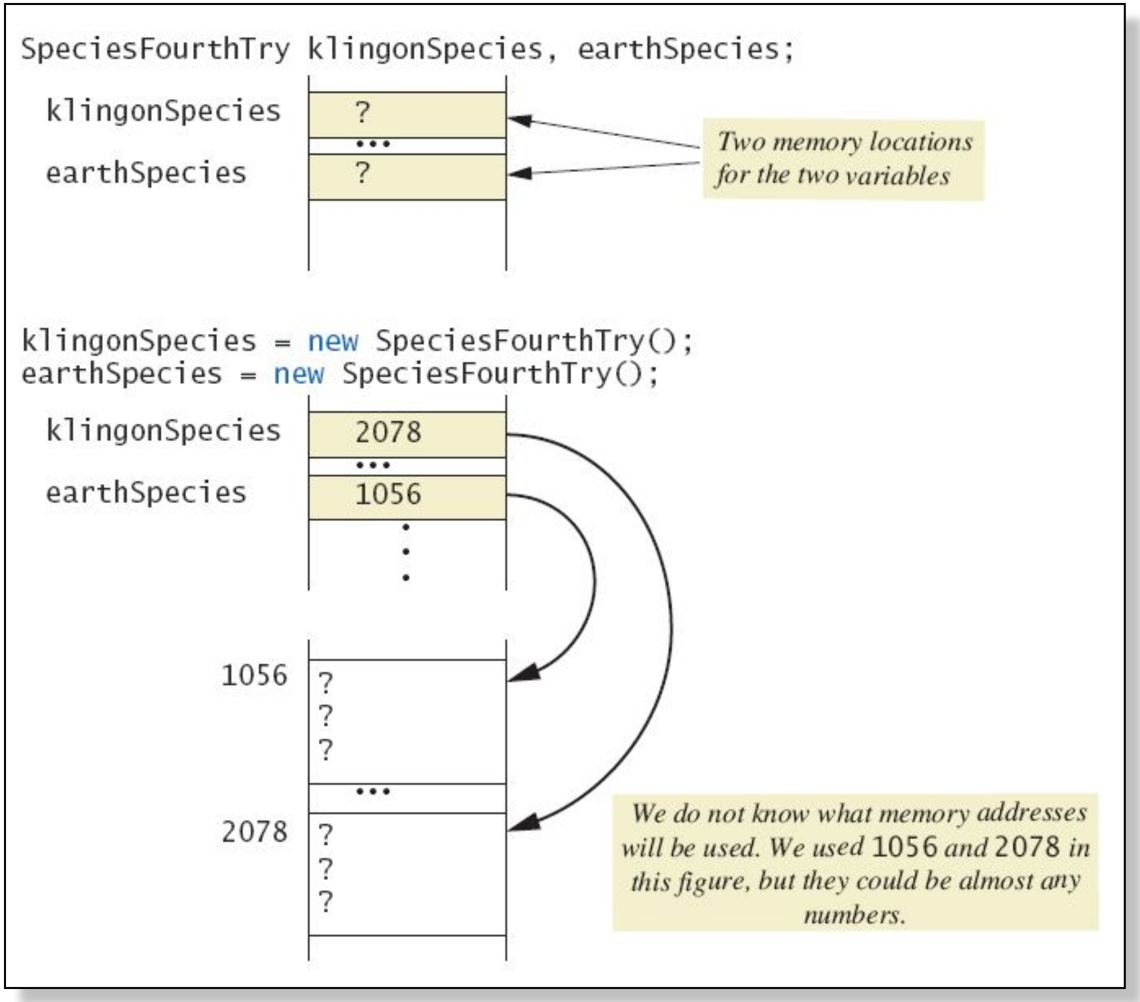
- Object itself not stored in the variable
  - Stored elsewhere in memory
  - Variable contains address of where it is stored

- Address is 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

- Behavior of class variables



```
SpeciesFourthTry klingonSpecies, earthSpecies;
```

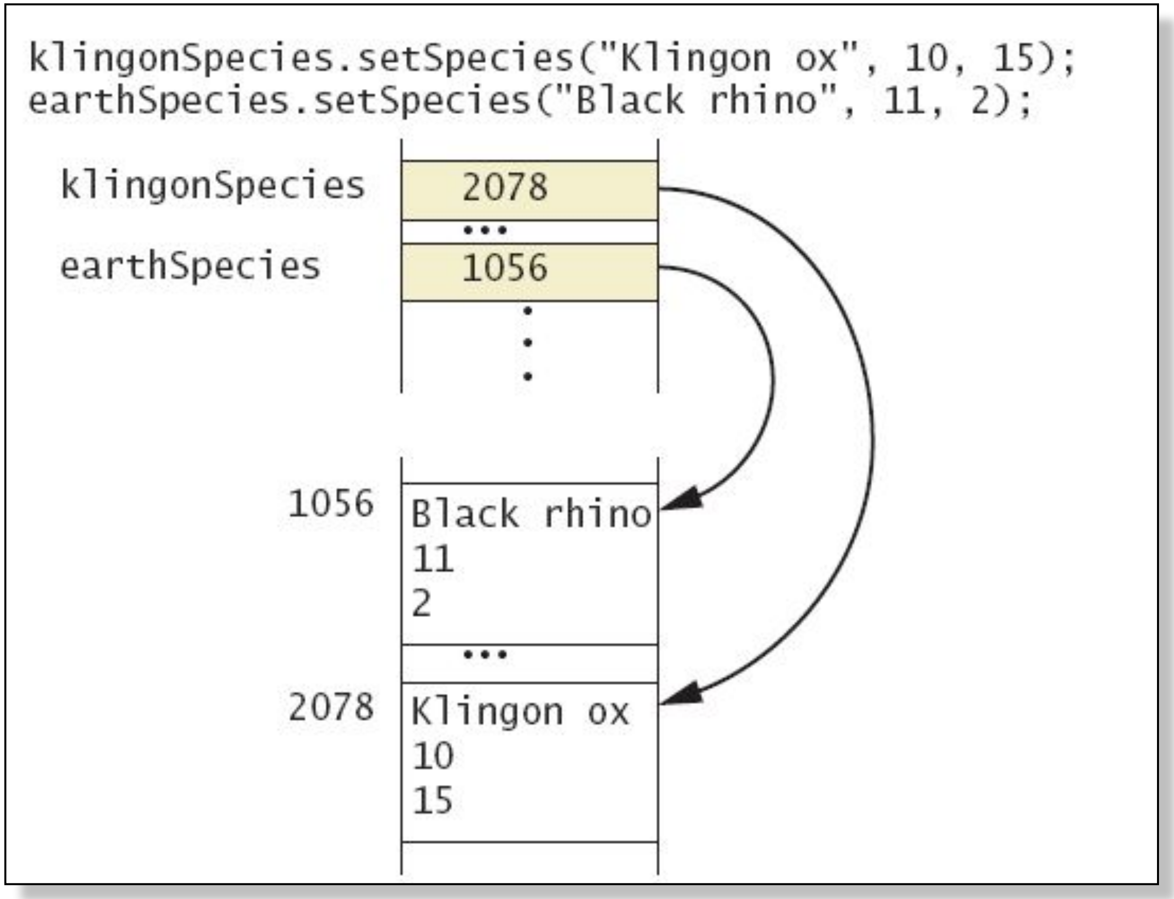
Two memory locations for the two variables

```
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

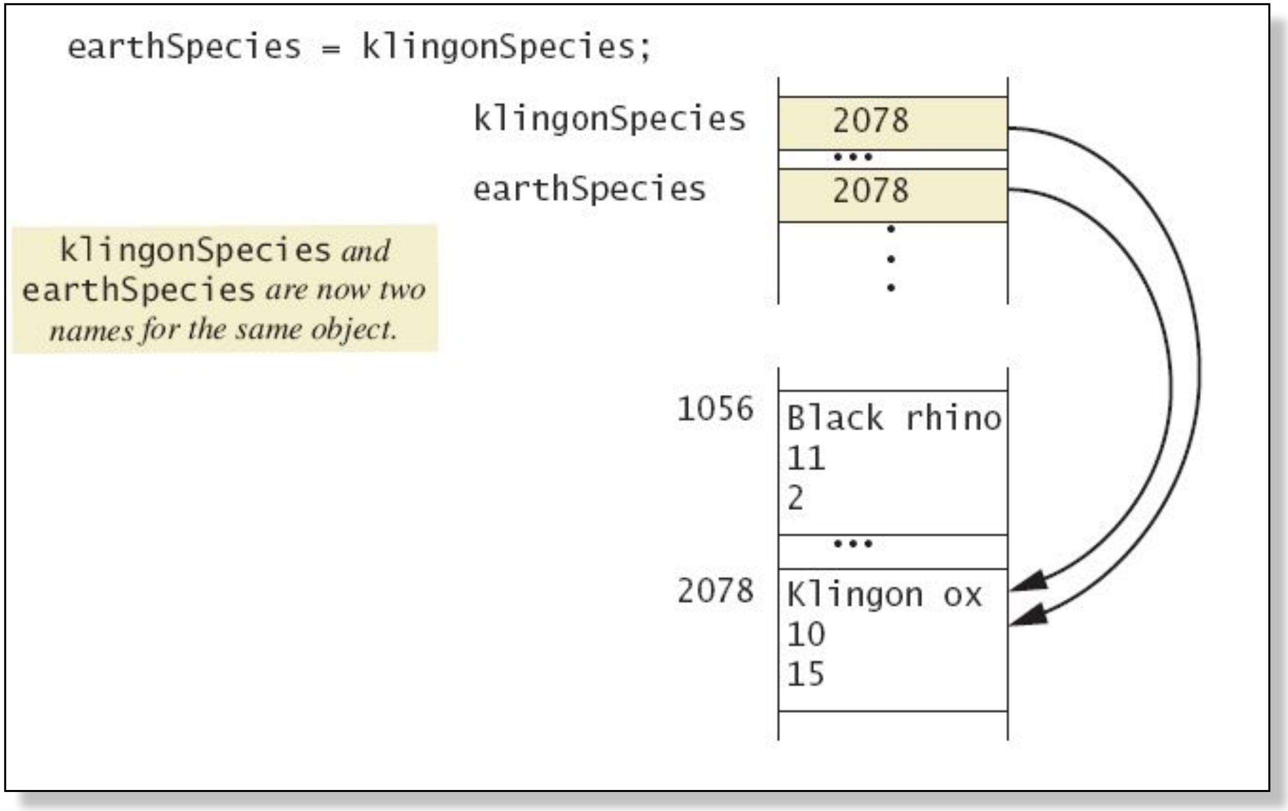
- Behavior of class variables



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

Variables of a Class Type

- Behavior of class variables

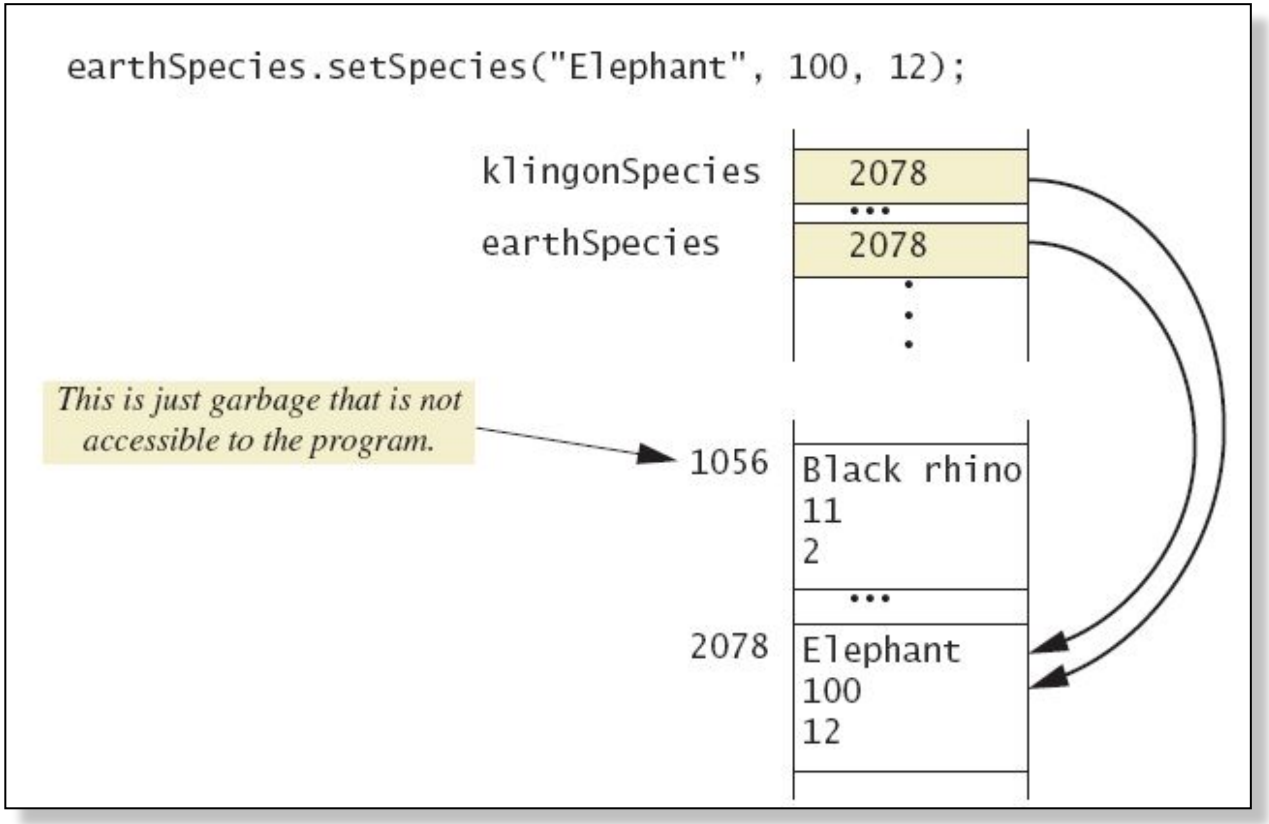


```
earthSpecies = klingonSpecies;
```

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

Variables of a Class Type

- Behavior of class variables

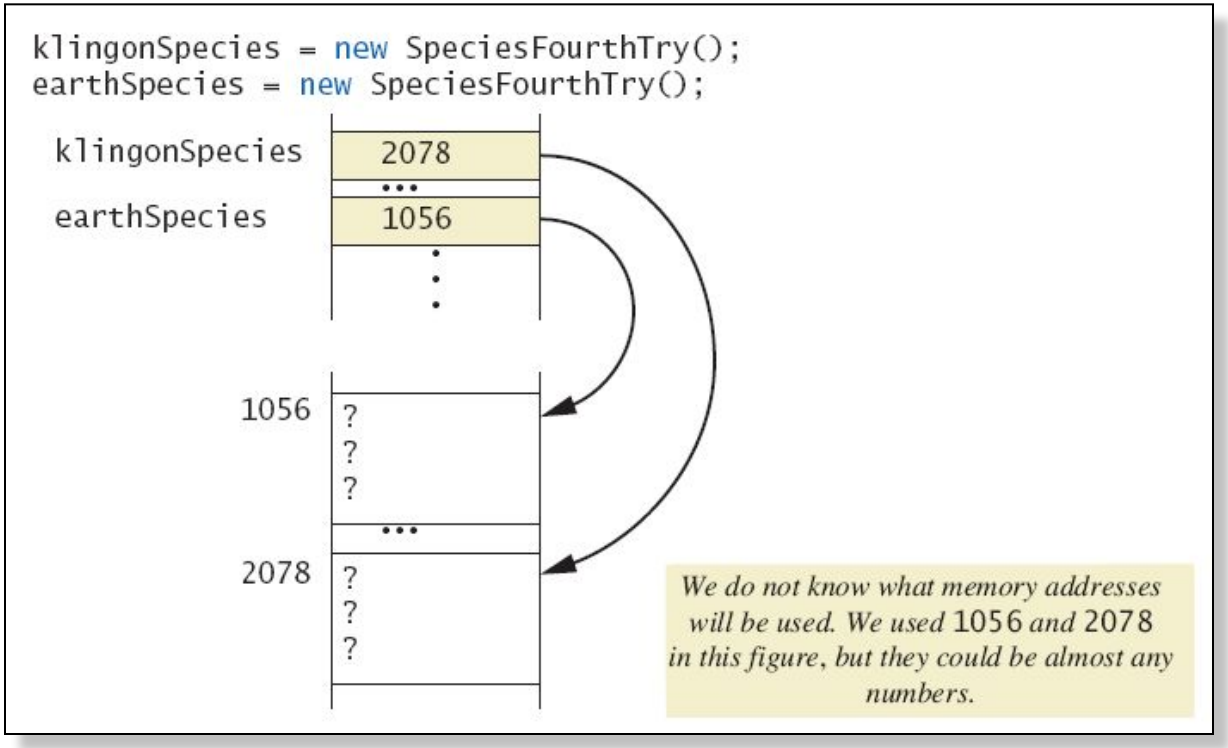


```
earthSpecies.setSpecies("Elephant", 100, 12);
```

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

Variables of a Class Type

- Dangers of using **==** with objects

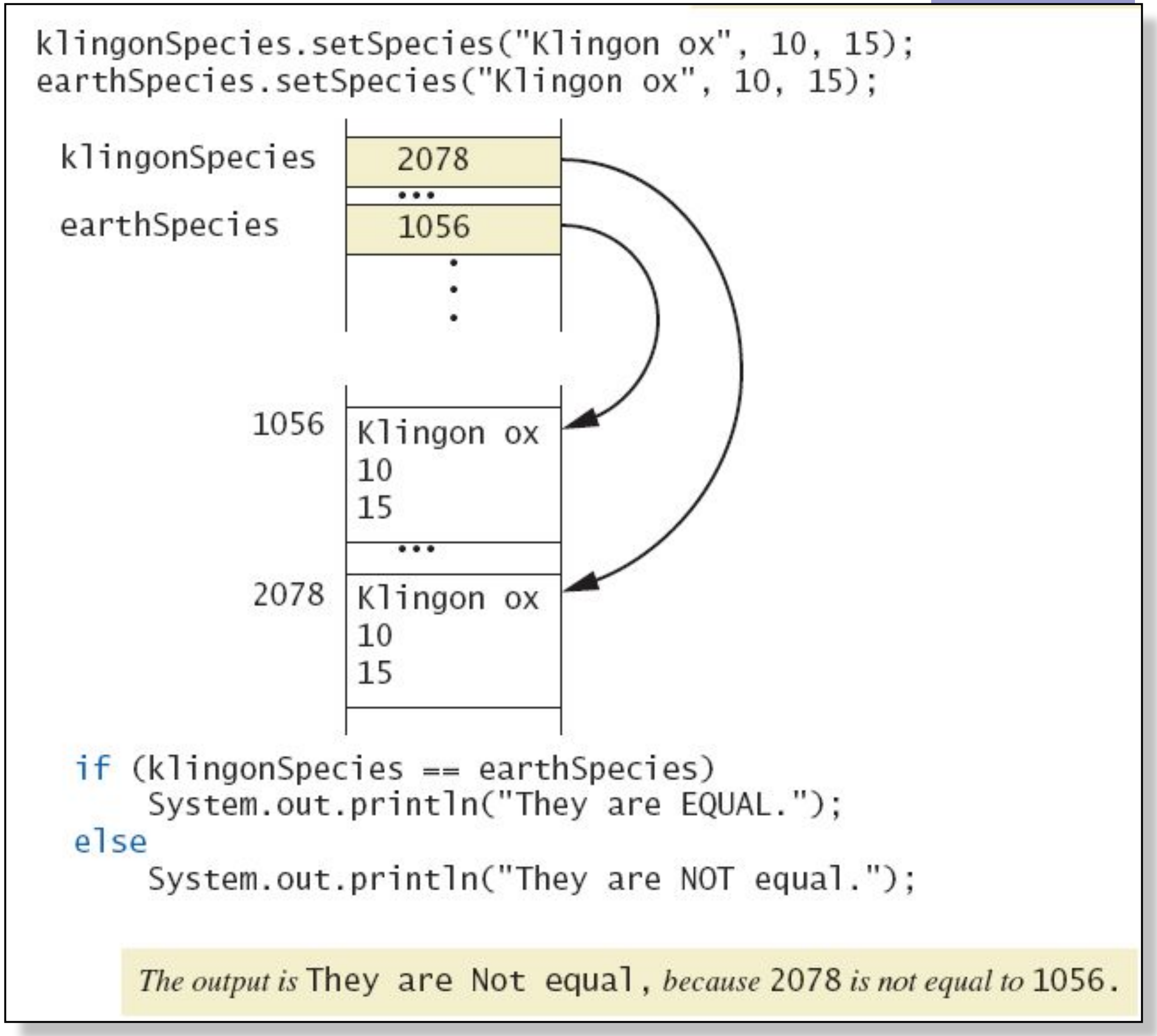


```
klingonSpecies = new SpeciesFourthTry();
earthSpecies = new SpeciesFourthTry();
```

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

- We CANNOT use == to compare two objects
- We must write a method for a given class which will make the comparison as needed

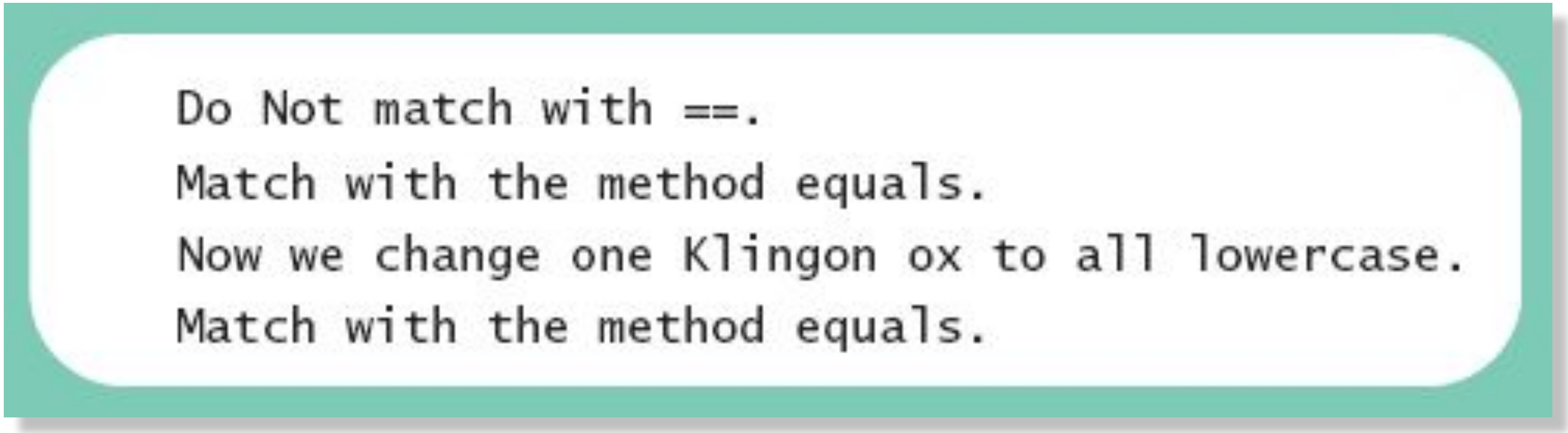
```
import java.util.Scanner;
public class Species
{
    private String name;
    private int population;
    private double growthRate;

         …

    public boolean equals (Species otherObject)
    {
        return (this.name.equalsIgnoreCase (otherObject.name)) &&
            (this.population == otherObject.population) &&
            (this.growthRate == otherObject.growthRate);
    }
}
```

Demonstrating an *equals* Method

```
public class SpeciesEqualsDemo
{
    public static void main (String [] args)
    {
        Species s1 = new Species (), s2 = new Species ();
        s1.setSpecies ("Klingon ox", 10, 15);
        s2.setSpecies ("Klingon ox", 10, 15);
        if (s1 == s2)
            System.out.println ("Match with ==.");
        else
            System.out.println ("Do Not match with ==.");
        if (s1.equals (s2))
            System.out.println ("Match with the method equals.");
        else
            System.out.println ("Do Not match with the method equals.");
        System.out.println ("Now we change one Klingon ox to all lowercase.");
        s2.setSpecies ("klingon ox", 10, 15); //Use lowercase
        if (s1.equals (s2))
            System.out.println ("Match with the method equals.");
        else
            System.out.println ("Do Not match with the method equals.");
    }
}
```

Demonstrating an *equals* Method

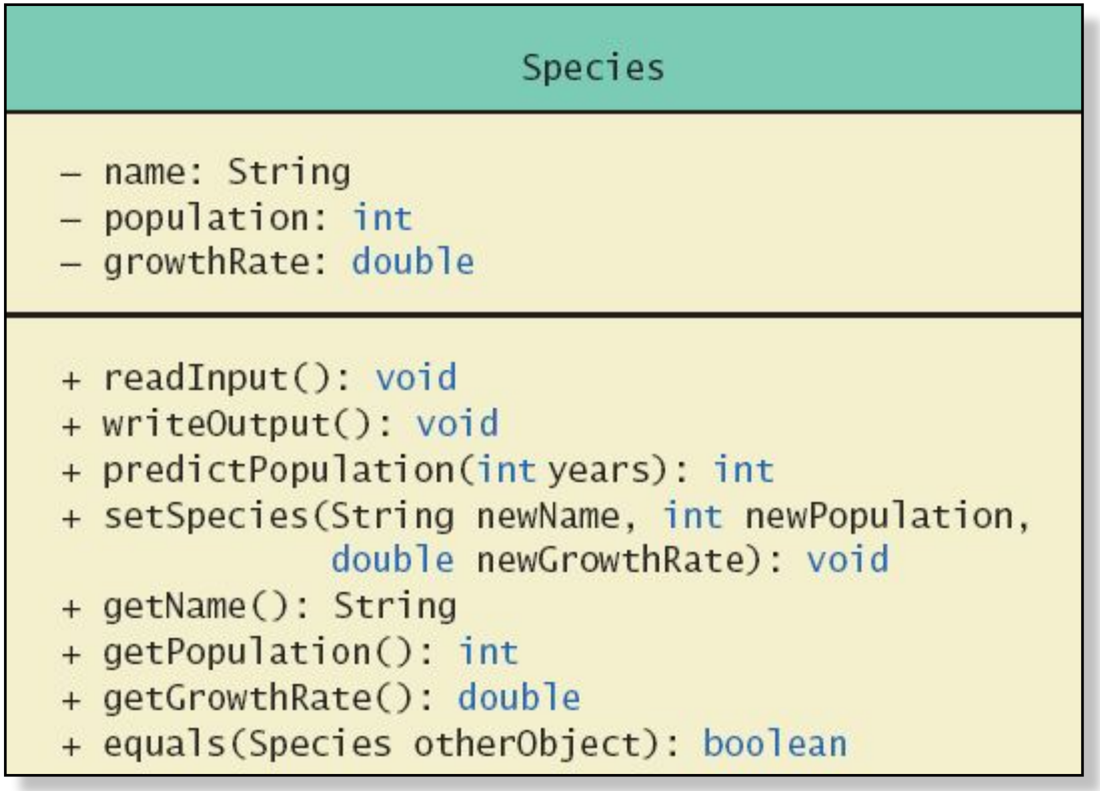
- Result



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

Can you code Species class?

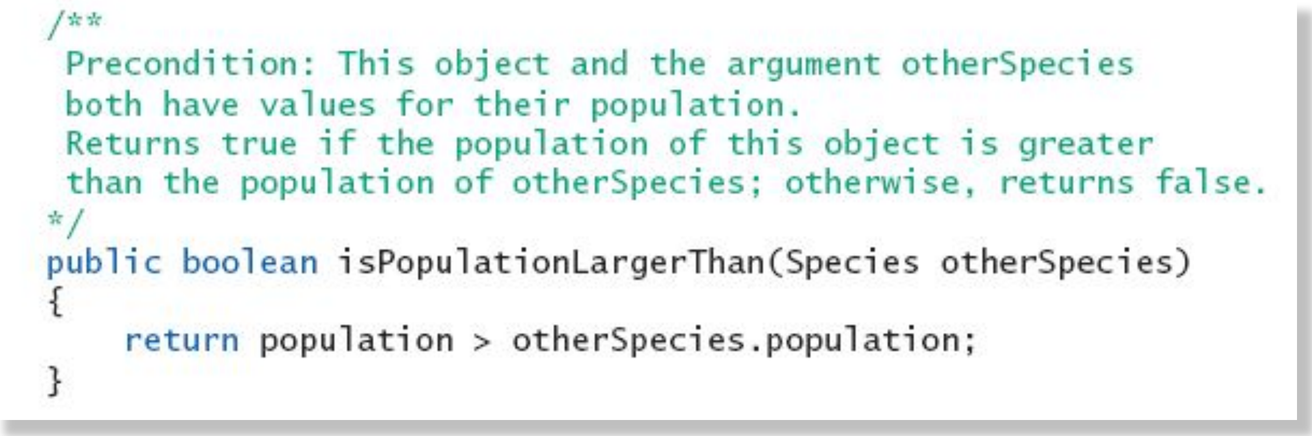
- Class Diagram for the class Species



```
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
```

Boolean-Valued Methods

- Methods can return a value of type boolean
- Use a boolean value in the return statement



```
/**
 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

DemoSpecias

- Tries to set intVariable equal to the population of this object. But arguments of a primitive type cannot be changed.

```
public void tryToChange (int intVariable)
{
    intVariable = this.population;
}
```

- Tries to make otherObject reference this object. But arguments of a class type cannot be replaced.

```
public void tryToReplace (DemoSpecies otherObject)
{
    otherObject = this;
}
```

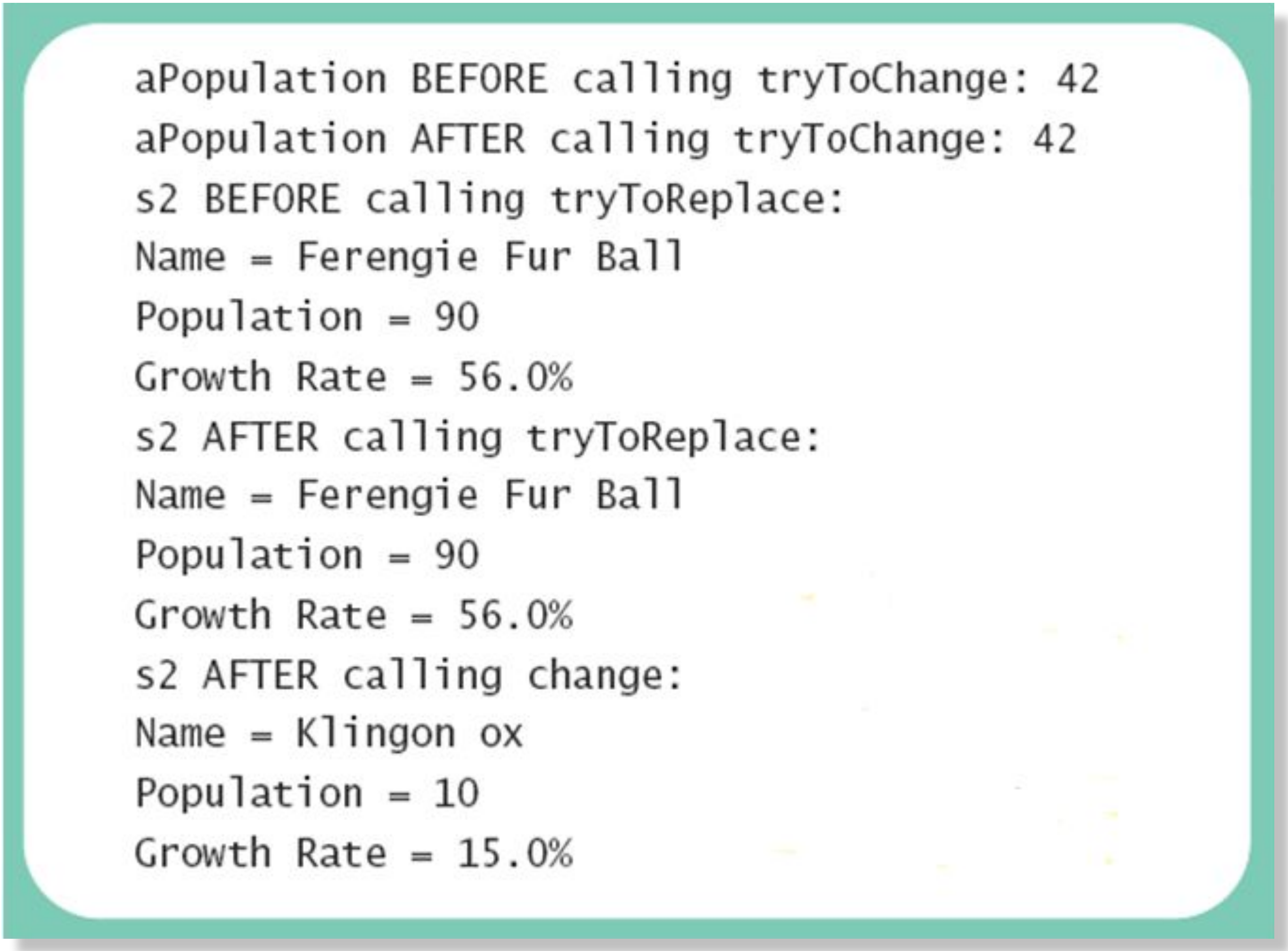
- Changes the data in otherObject to the data in this object.

```
public void change (DemoSpecies otherObject)
{
    otherObject.name = this.name;
    otherObject.population = this.population;
    otherObject.growthRate = this.growthRate;
}
```

ParametersDemo

```
public class ParametersDemo
{
    public static void main (String [] args)
    {
        DemoSpecies s1 = new DemoSpecies (),  s2 = new DemoSpecies ();
        s1.setSpecies ("Klingon ox", 10, 15);
        int aPopulation = 42;
        System.out.println ("aPopulation BEFORE calling tryToChange: “ + aPopulation);
        s1.tryToChange (aPopulation);
        System.out.println ("aPopulation AFTER calling tryToChange: “ + aPopulation);

        s2.setSpecies ("Ferengie Fur Ball", 90, 56);
        System.out.println ("s2 BEFORE calling tryToReplace: ");
        s2.writeOutput ();
        s1.tryToReplace (s2);
        System.out.println ("s2 AFTER calling tryToReplace: ");

        s2.writeOutput ();
        s1.change (s2);
        System.out.println ("s2 AFTER calling change: ");
        s2.writeOutput ();
    }
}
```

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%
```