CHAPTER 5 - CLASSES

CSC111

DR. SULTAN ALFARHOOD

sultanf@ksu.edu.sa

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
  - Objects of class types
  - Objects that interact with one another
- Program objects can represent
  - Objects in real world
  - Abstractions

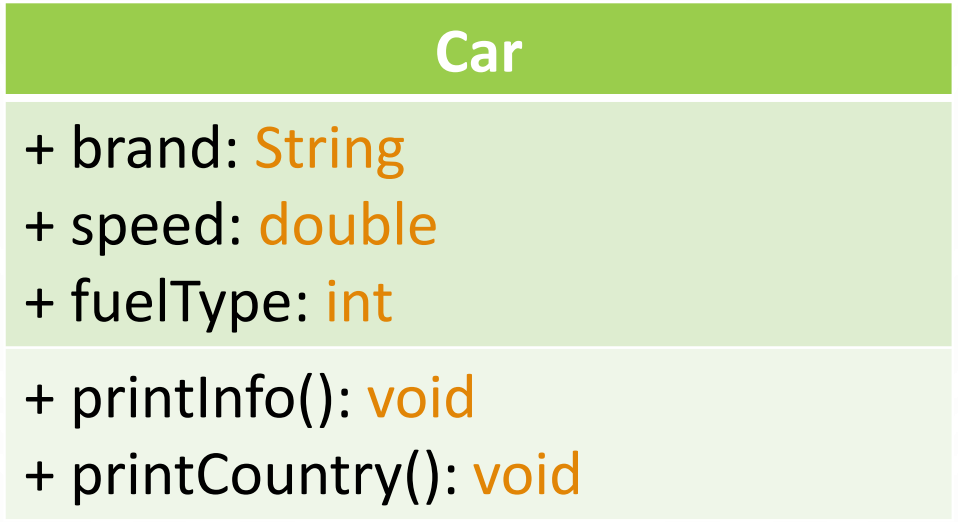
CLASS AND METHOD DEFINITIONS

A class as a blueprint

Class Name: Car
Data: • Brand • Speed • Fuel type
Methods (Actions): • Show Info • Show Country of Manufacture

CLASS AND METHOD DEFINITIONS

A class outline as a UML class diagram



CLASS AND METHOD DEFINITIONS

Object Name: c1

Brand: Toyota
speed: 240
fuelType: 95

Object Name: c2

Brand: Ford
speed: 180
fuelType: 91

Object Name: c3

Brand: Nissan
speed: 200
fuelType: 95

Objects that are instantiations of the class `Car`

```
class Car
{
    public String brand;
    public double speed;
    public int fuelType;

    public void printInfo()
    {
        System.out.println("--------------------");
        System.out.println("Car brand: "+brand);
        System.out.println("Car top speed: "+speed);
        System.out.println("Car fuel type: "+fuelType);
    }

    public void printCountry()
    {
        if(brand.equalsIgnoreCase("toyota") || brand.equalsIgnoreCase("nissan"))
            System.out.println("This car is manufactered in Japan");
        else if(brand.equalsIgnoreCase("ford") || brand.equalsIgnoreCase("chevrolet"))
            System.out.println("This car is manufactered in the USA");
        else
            System.out.println("Sorry, I don't know where this car is manufactered.");
    }
}
```

```
public class CarDemo
{
    public static void main(String args[])
    {
        Car c1=new Car();
        c1.brand="Toyota";
        c1.speed=240;
        c1.fuelType=95;

        c1.printInfo();
        c1.printCountry();

        Car c2=new Car();
        c2.brand="Ford";
        c2.speed=180;
        c2.fuelType=91;

        c2.printInfo();
        c2.printCountry();
    }
}
```

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

LISTING 5.1 Definition of a Dog Class

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

Later in this chapter we will see that the modifier `public` *for instance variables should be replaced with* `private`.

DOG CLASS AND INSTANCE VARIABLES

- `class Dog`
- Note class has
  - Three pieces of data (instance variables)
  - Two behaviors
- 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`

LISTING 5.2 Using the Dog Class and Its Methods

```
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.");
    }
}
```

Sample Screen Output

```
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
  - 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
- Calling a void method
  - Write the invocation followed by a semicolon
  - Resulting statement performs the action defined by the method

DEFINING `VOID` METHODS

- Consider method `writeOutput` from

```
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`**

THIRD EXAMPLE – SPECIES CLASS

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

LISTING 5.3 **A Species Class Definition—First Attempt** *(part 1 of 2)*

We will give a better version of this class later in this chapter.

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

    public void readInput()
    {
        Scanner keyboard = new Scanner(System.in);
        System.out.println("What is the species' name?");
        name = keyboard.nextLine();
        System.out.println("What is the population of the " +
                           "species?");
        population = keyboard.nextInt();
```

Later in this chapter you will see that the modifier `public` *for instance variables should be replaced with* `private`.

```
        System.out.println("Enter growth rate " +
                           "(% increase per year):");
        growthRate = keyboard.nextDouble();
    }
    public void writeOutput()
    {
        System.out.println("Name = " + name);
        System.out.println("Population = " + population);
        System.out.println("Growth rate = " + growthRate + "%");
    }
    public int getPopulationIn10()
    {
        int result = 0;
        double populationAmount = population;
        int count = 10;
        while ((count > 0) && (populationAmount > 0))
        {
            populationAmount = populationAmount +
                               (growthRate / 100) *
                               populationAmount;
            count--;
        }
        if (populationAmount > 0)
                result = (int)populationAmount;
        return result;
    }
}
```

LISTING 5.4 **Using the Species Class and Its Methods** *(part 1 of 2)*

```
public class SpeciesFirstTryDemo
{
    public static void main(String[] args)
    {
        SpeciesFirstTry speciesOfTheMonth = new SpeciesFirstTry();
        System.out.println("Enter data on the Species of "+
                           "the Month:");

        speciesOfTheMonth.readInput();
        speciesOfTheMonth.writeOutput();
        int futurePopulation =
            speciesOfTheMonth.getPopulationIn10();
        System.out.println("In ten years the population will be "
                           + futurePopulation);
        //Change the species to show how to change
        //the values of instance variables:
        speciesOfTheMonth.name = "Klingon ox";
        speciesOfTheMonth.population = 10;
        speciesOfTheMonth.growthRate = 15;
        System.out.println("The new Species of the Month:");
        speciesOfTheMonth.writeOutput();
        System.out.println("In ten years the population will "
                "be " + speciesOfTheMonth.getPopulationIn10());
    }
}
```

Sample Screen Output

```
Enter data on the Species of the Month:
What is the species' name?
Ferengie fur ball
What is the population of the species?
1000
Enter growth rate (% increase per year):
-20.5
Name = Ferengie fur ball
Population = 1000
Growth rate = 20.5%
In ten years the population will be 100
The new Species of the Month:
Name = Klingon ox
Population = 10
Growth rate = 15.0%
In ten years the population will be 40
```

THE KEYWORD `THIS`

- 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

- 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 and declared in different methods are different variables

LOCAL VARIABLES

- `class BankAccount`
- `class LocalVariablesDemoProgram`
- 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

LISTING 5.5 Local Variables

This class definition is in a file named `BankAccount.java`.

```
/**
This class is used in the program LocalVariablesDemoProgram.
*/
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);
    }
}
```

This does not change the value of the variable `newAmount` *in* `main`.

Two different variables named `newAmount`

This program is in a file named `LocalVariableDemoProgram.java`.

```
/**
A toy program to illustrate how local variables behave.
*/
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);
    }
}
```

Screen Output *Chapter 6 will fix the appearance of dollar amounts.*

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

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

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

- Recall method declaration
  - Note it only works for 10 years
  - We can make it more versatile by giving the method a parameter to specify how many years
- **`class SpeciesSecondTry`**

LISTING 5.6 A Method That Has a Parameter

We will give an even better version of the class later in the chapter.

```
import java.util.Scanner;
public class SpeciesSecondTry
{
    <The declarations of the instance variables name, population,
    and growthRate are the same as in Listing 5.3.>
    <The definitions of the methods readInput and writeOutput
    are the same as in Listing 5.3.>
    /**
    Returns the projected population of the receiving object
    after the specified number of years.
    */
    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

- 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

- `class SpeciesSecondClassDemo`

LISTING 5.7 Using a Method That Has a Parameter

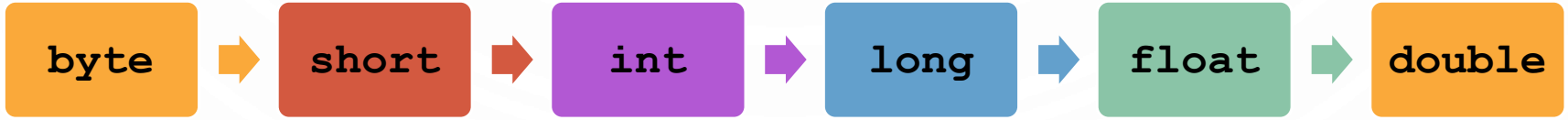
```
/**
Demonstrates the use of a parameter
with the method predictPopulation.
*/
public class SpeciesSecondTryDemo
{
    public static void main(String[] args)
    {
        SpeciesSecondTry speciesOfTheMonth = new
                                    SpeciesSecondTry();
        System.out.println("Enter data on the Species of the " +
                           "Month:");
        speciesOfTheMonth.readInput();
        speciesOfTheMonth.writeOutput();
        int futurePopulation =
            speciesOfTheMonth.predictPopulation(10);
        System.out.println("In ten years the population will be " +
                           futurePopulation);
        //Change the species to show how to change
        //the values of instance variables:
        speciesOfTheMonth.name = "Klingon ox";
        speciesOfTheMonth.population = 10;
        speciesOfTheMonth.growthRate = 15;
        System.out.println("The new Species of the Month:");
        speciesOfTheMonth.writeOutput();
        System.out.println("In ten years the population will be " +
                           speciesOfTheMonth.predictPopulation(10));
    }
}
```

Sample Screen Output

The output is exactly the same as in Listing 5.4.

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



INFORMATION HIDING, ENCAPSULATION: OUTLINE

- Information Hiding
- Pre- and Postcondition Comments
- The public and private Modifiers
- Methods Calling Methods
- Encapsulation
- Automatic Documentation with `javadoc`
- UML Class Diagrams

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
- 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()
```

PRE- AND POSTCONDITION COMMENTS

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

```
/**
 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`
- `class SpeciesThirdTry`

LISTING 5.8 A Class with Private Instance Variables

```
import java.util.Scanner;
public class SpeciesThirdTry
{
    private String name;
    private int population;
    private double growthRate;
    <The definitions of the methods readInput, writeOutput, and
    predictPopulation are the same as in Listing 5.3 and
    Listing 5.6.>
}
```

We will give an even better version of this class later in the chapter.

PROGRAMMING EXAMPLE

- Demonstration of need for private variables
- Statement such as

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

  is illegal since width is `private`

  - Keeps remaining elements of the class consistent in this example

LISTING 5.9 A Class of Rectangles

```
/**
Class that represents a rectangle.
*/
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;
    }
}
```

PROGRAMMING EXAMPLE

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

LISTING 5.10 Another Class of Rectangles

```
/**
Another class that represents a rectangle.
*/
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;
    }
}
```

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 with accessor and mutator methods
- `class SpeciesFourthTry`
- Note the mutator method
  - `setSpecies`
- Note accessor methods
  - `getName`, `getPopulation`, `getGrowthRate`

LISTING 5.11 A Class with Accessor and Mutator Methods

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

    <The definitions of the methods readInput, writeOutput, and
     predictPopulation go here. They are the same as in Listing
     5.3 and Listing 5.6.>
    public void setSpecies(String newName, int newPopulation,
                           double newGrowthRate)
    {
        name = newName;
        if (newPopulation >= 0)
            population = newPopulation;
        else
        {
                System.out.println(
                          "ERROR: using a negative population.");
                System.exit(0);
        }
        growthRate = newGrowthRate;
    }
    public String getName()
    {
        return name;
    }
    public int getPopulation()
    {
        return population;
    }
    public double getGrowthRate()
    {
        return growthRate;
    }
}
```

Yes, we will define an even better version of this class later.

A mutator method can check to make sure that instance variables are set to proper values.

ACCESSOR AND MUTATOR METHODS

- Using a mutator method
- **classSpeciesFourthTryDemo**

```
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

LISTING 5.12 Using a Mutator Method *(part 1 of 2)*

```
import java.util.Scanner;
/**
Demonstrates the use of the mutator method setSpecies.
*/
public class SpeciesFourthTryDemo
{
    public static void main(String[] args)
    {
        SpeciesFourthTry speciesOfTheMonth =
                                    new SpeciesFourthTry();
        System.out.println("Enter number of years to project:");
        Scanner keyboard = new Scanner(System.in);
        int numberOfYears = keyboard.nextInt();

        System.out.println(
                    "Enter data on the Species of the Month:");
        speciesOfTheMonth.readInput();
        speciesOfTheMonth.writeOutput();

        int futurePopulation =
        speciesOfTheMonth.predictPopulation(numberOfYears);
        System.out.println("In " + numberOfYears +
                           " years the population will be " +
                           futurePopulation);
        //Change the species to show how to change
        //the values of instance variables:
        speciesOfTheMonth.setSpecies("Klingon ox", 10, 15);
        System.out.println("The new Species of the Month:");
        speciesOfTheMonth.writeOutput();

        futurePopulation =
              speciesOfTheMonth.predictPopulation(numberOfYears);
        System.out.println("In " + numberOfYears +
                           " years the population will be " +
                           futurePopulation);
    }
}
```

Sample Screen Output

```
Enter number of years to project:
10
Enter data on the Species of the Month:
What is the species' name?
Ferengie fur ball
```

PROGRAMMING EXAMPLE

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

LISTING 5.13 The Purchase Class *(part 1 of 3)*

```
import java.util.Scanner;
/**
Class for the purchase of one kind of item, such as 3 oranges.
Prices are set supermarket style, such as 5 for $1.25.
*/
public class Purchase
{
    private String name;
    private int groupCount;     //Part of a price, like the 2 in
                                //2 for $1.99.
    private double groupPrice;  //Part of a price, like the $1.99
                                // in 2 for $1.99.
    private int numberBought;   //Number of items bought.
    public void setName(String newName)
    {
        name = newName;
    }
    /**
    Sets price to count pieces for $costForCount.
    For example, 2 for $1.99.
    */
    public void setPrice(int count, double costForCount)
    {
        if ((count <= 0) || (costForCount <= 0))
        {
            System.out.println("Error: Bad parameter in " +
                               "setPrice.");
            System.exit(0);
        }
        else
        {
            groupCount = count;
            groupPrice = costForCount;
        }
    }
```

```
public void setNumberBought(int number)
{
    if (number <= 0)
    {
        System.out.println("Error: Bad parameter in " +
                           "setNumberBought.");
        System.exit(0);
    }
    else
        numberBought = number;
}
```

```
/**
Reads from keyboard the price and number of a purchase.
*/
public void readInput()
{
    Scanner keyboard = new Scanner(System.in);
    System.out.println("Enter name of item you are purchasing:");
    name = keyboard.nextLine();
    System.out.println("Enter price of item as two numbers.");
    System.out.println("For example, 3 for $2.99 is entered as");
    System.out.println("3 2.99");
    System.out.println("Enter price of item as two numbers, " +
                       "now:");
    groupCount = keyboard.nextInt();
    groupPrice = keyboard.nextDouble();

    while ((groupCount <= 0) || (groupPrice <= 0))
    { //Try again:
        System.out.println("Both numbers must " +
                           "be positive. Try again.");
        System.out.println("Enter price of " +
                           "item as two numbers.");
        System.out.println("For example, 3 for " +
                           "$2.99 is entered as");
        System.out.println("3 2.99");
        System.out.println(
                         "Enter price of item as two numbers, now:");
        groupCount = keyboard.nextInt();
        groupPrice = keyboard.nextDouble();
    }
    System.out.println("Enter number of items purchased:");
    numberBought = keyboard.nextInt();

    while (numberBought <= 0)
    { //Try again:
        System.out.println("Number must be positive. " +
                           "Try again.");
        System.out.println("Enter number of items purchased:");
        numberBought = keyboard.nextInt();
    }
}
```

```
    /**
    Displays price and number being purchased.
    */
    public void writeOutput()
    {
        System.out.println(numberBought + " " + name);
        System.out.println("at " + groupCount +
                           " for $" + groupPrice);
    }
    public String getName()
    {
        return name;
    }
    public double getTotalCost()
    {
        return (groupPrice / groupCount) * numberBought;
    }
    public double getUnitCost()
    {
        return groupPrice / groupCount;
    }
    public int getNumberBought()
    {
        return numberBought;
    }
}
```

LISTING 5.14 Use of the Purchase Class

```
public class PurchaseDemo
{
    public static void main(String[] args)
    {
        Purchase oneSale = new Purchase();
        oneSale.readInput();
        oneSale.writeOutput();
        System.out.println("Cost each $" + oneSale.getUnitCost());
        System.out.println("Total cost $" +
                oneSale.getTotalCost());
    }
}
```

Sample Screen Output

```
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
```

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
- class Oracle
- class OracleDemo

LISTING 5.15 Methods Calling Other Methods

```
import java.util.Scanner;
public class Oracle
{
    private String oldAnswer = "The answer is in your heart.";
    private String newAnswer;
    private String question;

    public void chat()
    {
        System.out.print("I am the oracle. ");
        System.out.println("I will answer any one-line question.");
        Scanner keyboard = new Scanner(System.in);
        String response;
        do
        {
            answer();
            System.out.println("Do you wish to ask " +
                "another question?");
            response = keyboard.next();
        } while (response.equalsIgnoreCase("yes"));
        System.out.println("The oracle will now rest.");
    }
```

```
    private void answer()
    {
        System.out.println("What is your question?");
        Scanner keyboard = new Scanner(System.in);
        question = keyboard.nextLine();
        seekAdvice();
        System.out.println("You asked the question:");
        System.out.println(" " + question);
        System.out.println("Now, here is my answer:");
        System.out.println(" " + oldAnswer);
        update();
    }
    private void seekAdvice()
    {
        System.out.println("Hmm, I need some help on that.");
        System.out.println("Please give me one line of advice.");
        Scanner keyboard = new Scanner(System.in);
        newAnswer = keyboard.nextLine();
        System.out.println("Thank you. That helped a lot.");
    }
    private void update()
    {
        oldAnswer = newAnswer;
    }
}
```

LISTING 5.16 Oracle Demonstration Program *(part 1 of 2)*

```
public class OracleDemo
{
    public static void main(String[] args)
    {
        Oracle delphi = new Oracle();
        delphi.chat();
    }
}
```

Sample Screen Output

```
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?
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.
```

ENCAPSULATION

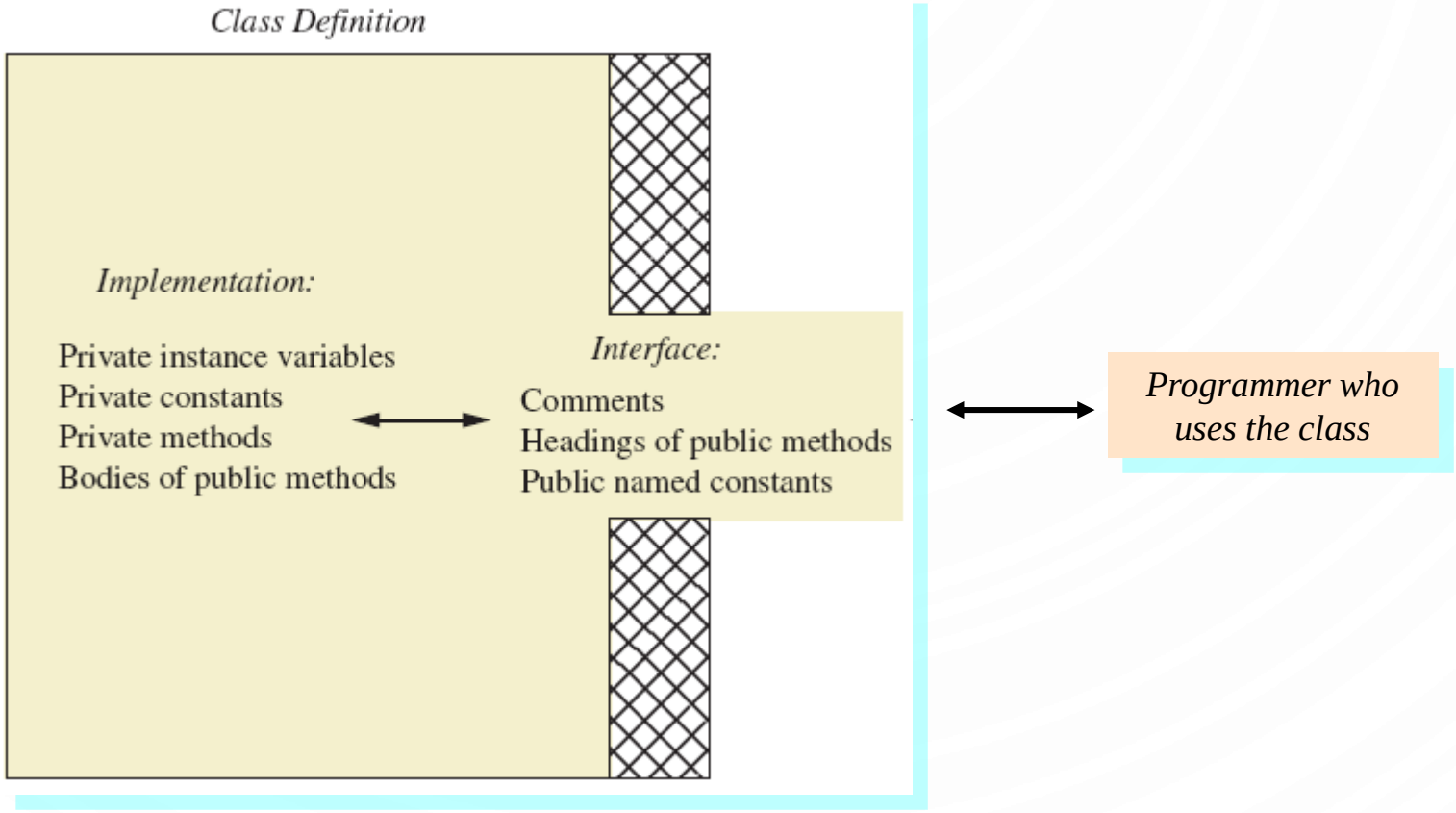
- Consider example of driving a car
  - We see and use break pedal, accelerator pedal, steering wheel – know what they do
  - We do not see mechanical details of how they do their jobs
- Encapsulation divides class definition into
  - Class interface
  - Class implementation

ENCAPSULATION

- A *class interface*
  - Tells 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



ENCAPSULATION

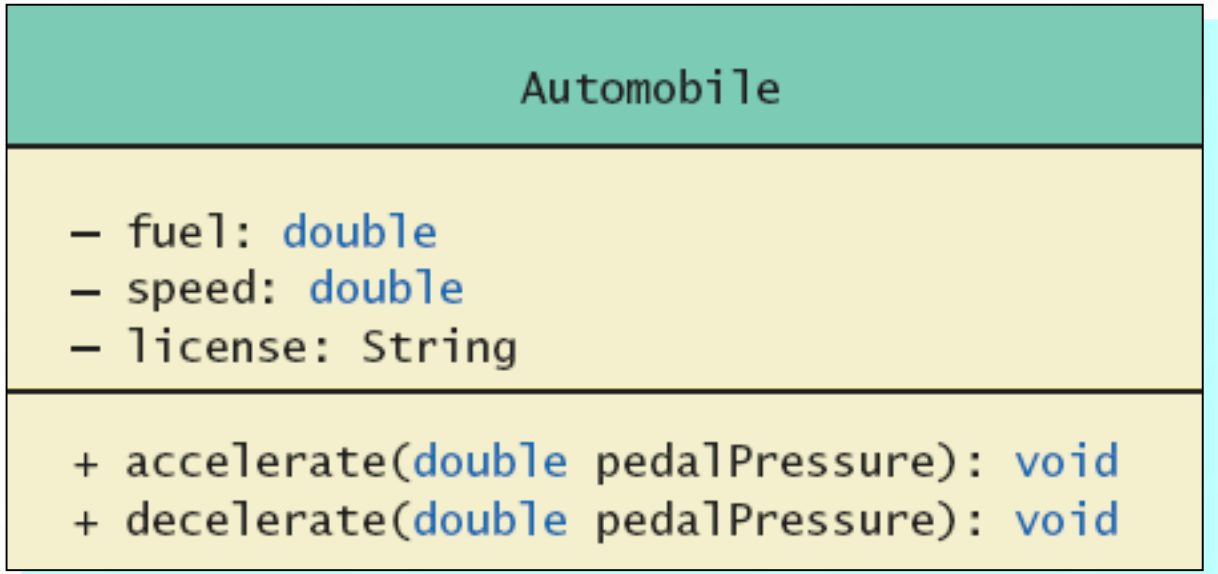
- Preface class definition with comment on how to use class
- Declare all instance variables in the class as private.
- Provide public accessor methods to retrieve data Provide public methods manipulating data
  - Such methods could include public mutator methods.
- Place a comment before each public 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

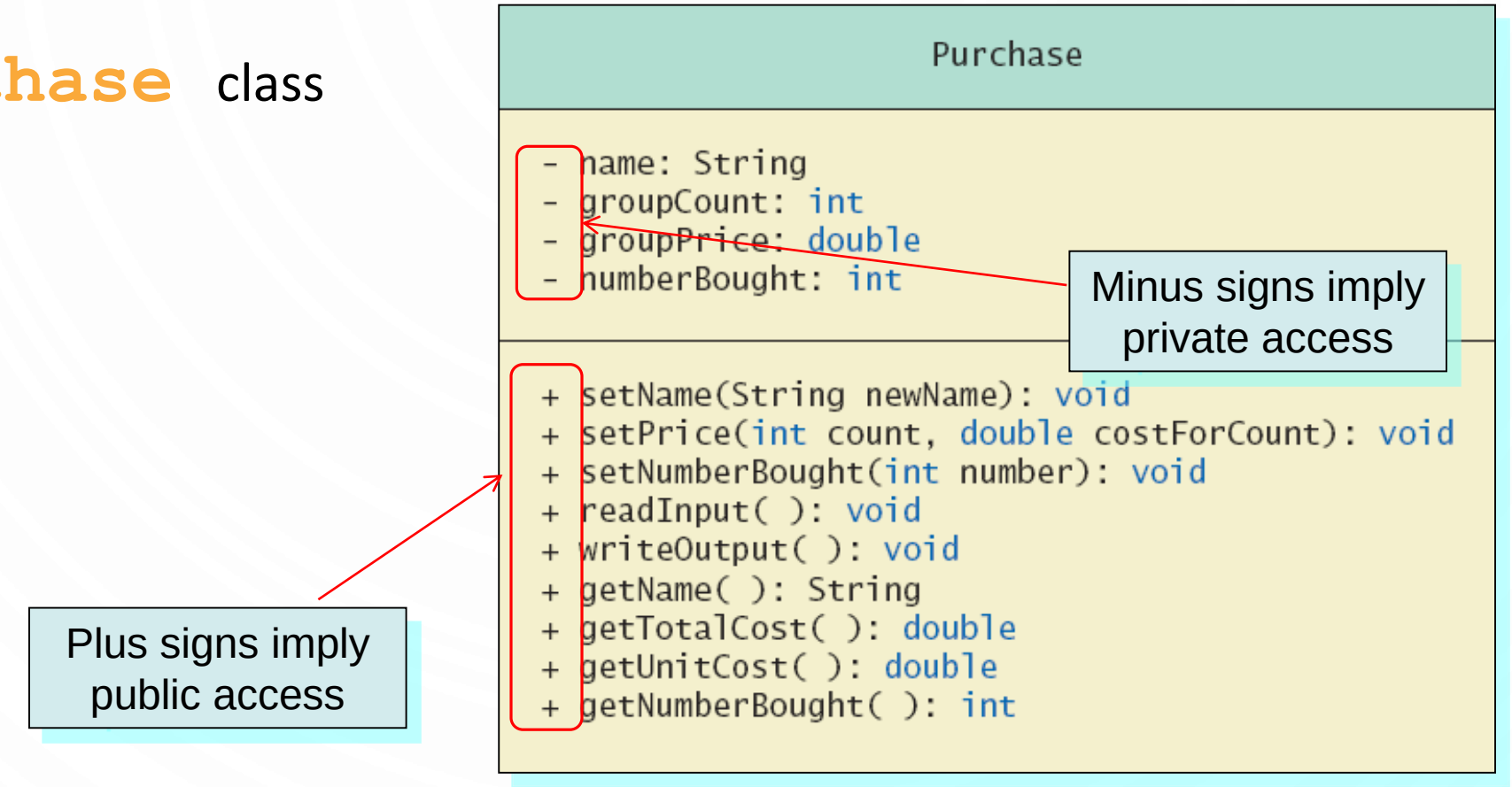
UML CLASS DIAGRAMS

- A class outline as a UML class diagram



UML CLASS DIAGRAMS

- **`Purchase`** class



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

OBJECTS AND REFERENCES: OUTLINE

- Variables of a Class Type
- Defining an equals Method for a Class
- Boolean-Valued Methods
- Parameters of a Class Type

VARIABLES OF A CLASS TYPE

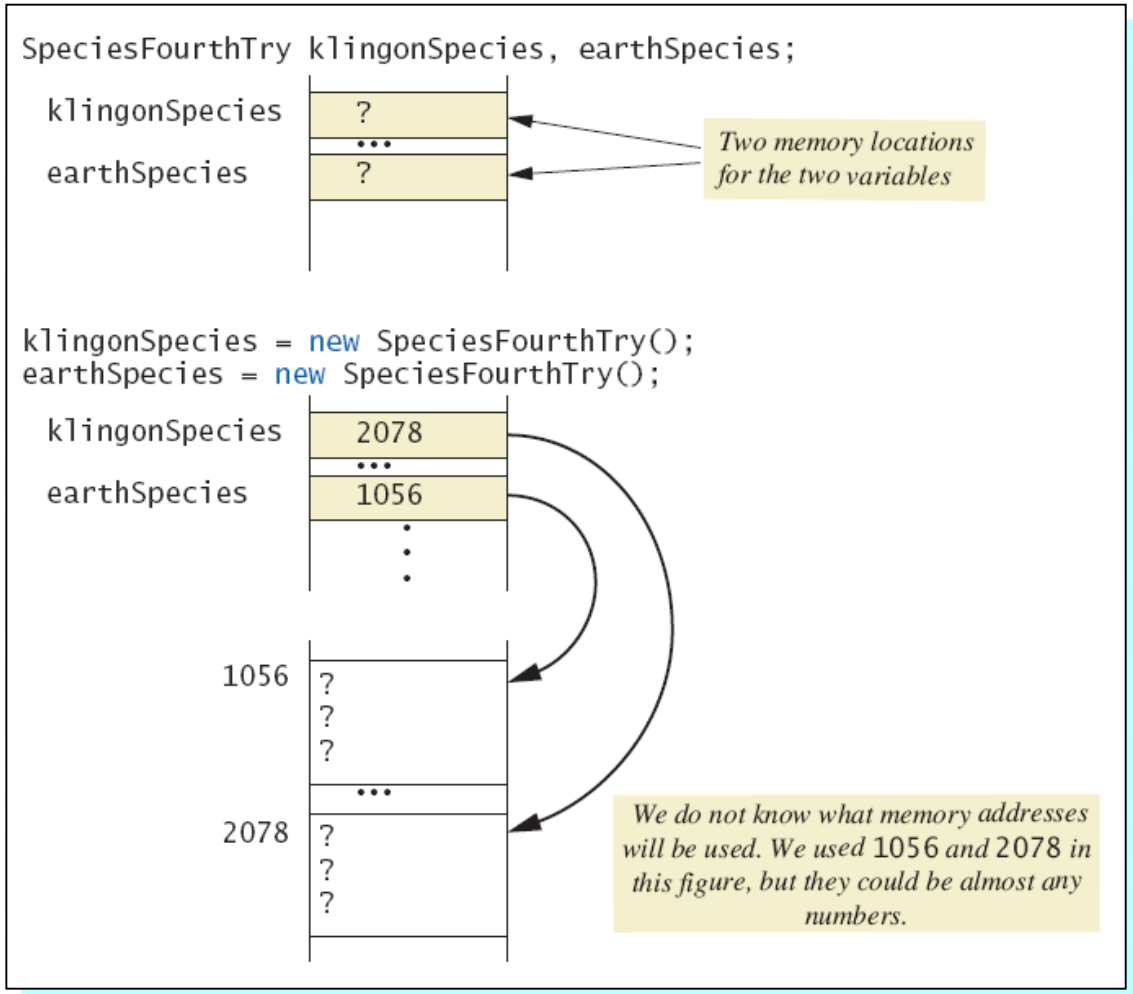
- All variables are implemented as a memory location
- Data of *primitive type* stored in the memory location assigned to the variable
- Variable of *class type* contains memory address of object named by the variable

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

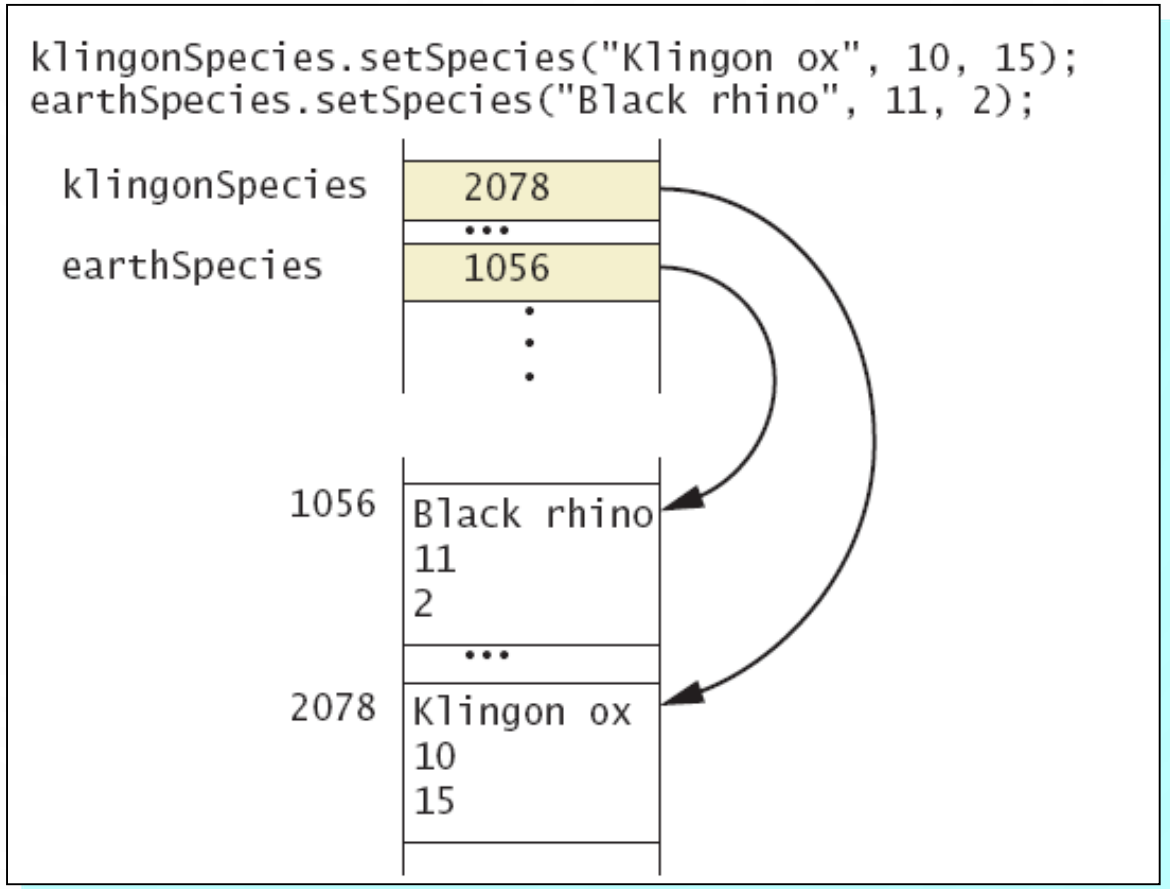
VARIABLES OF A CLASS TYPE

- Behavior of class variables



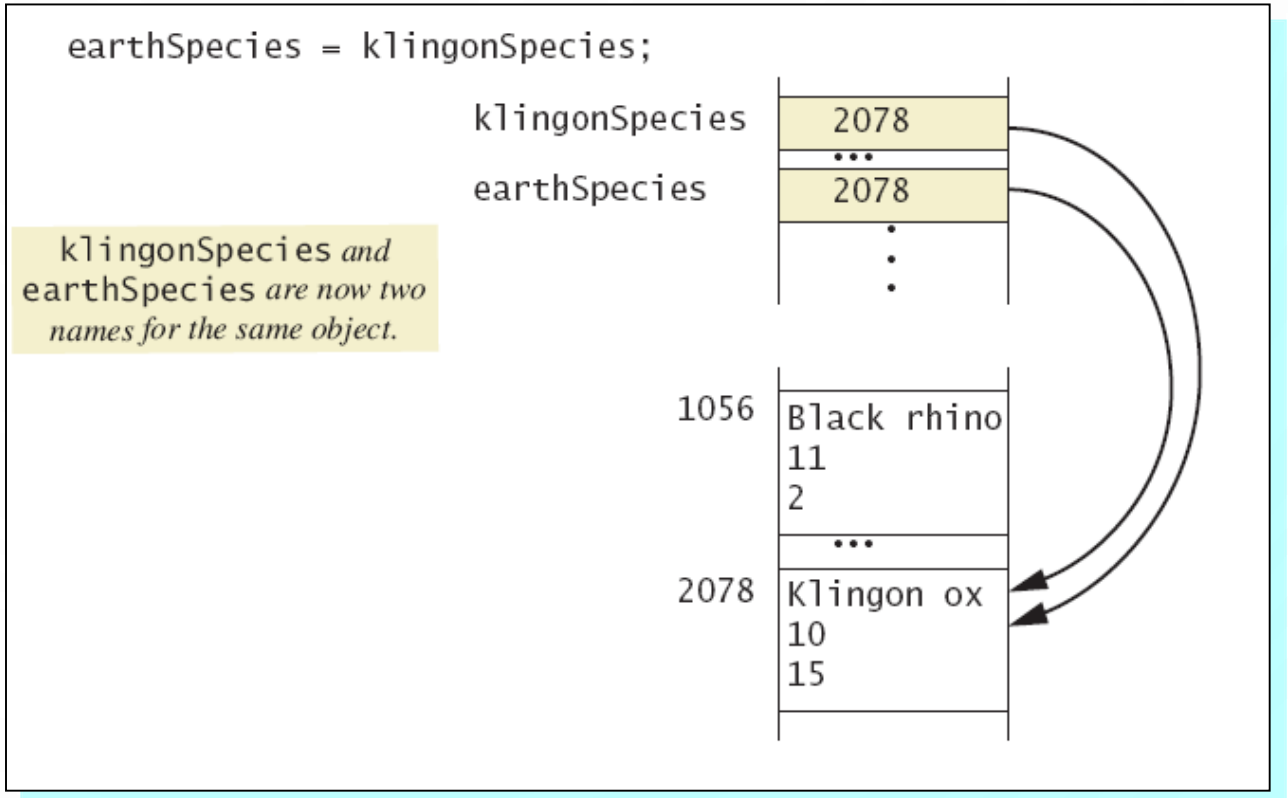
VARIABLES OF A CLASS TYPE

- Behavior of class variables



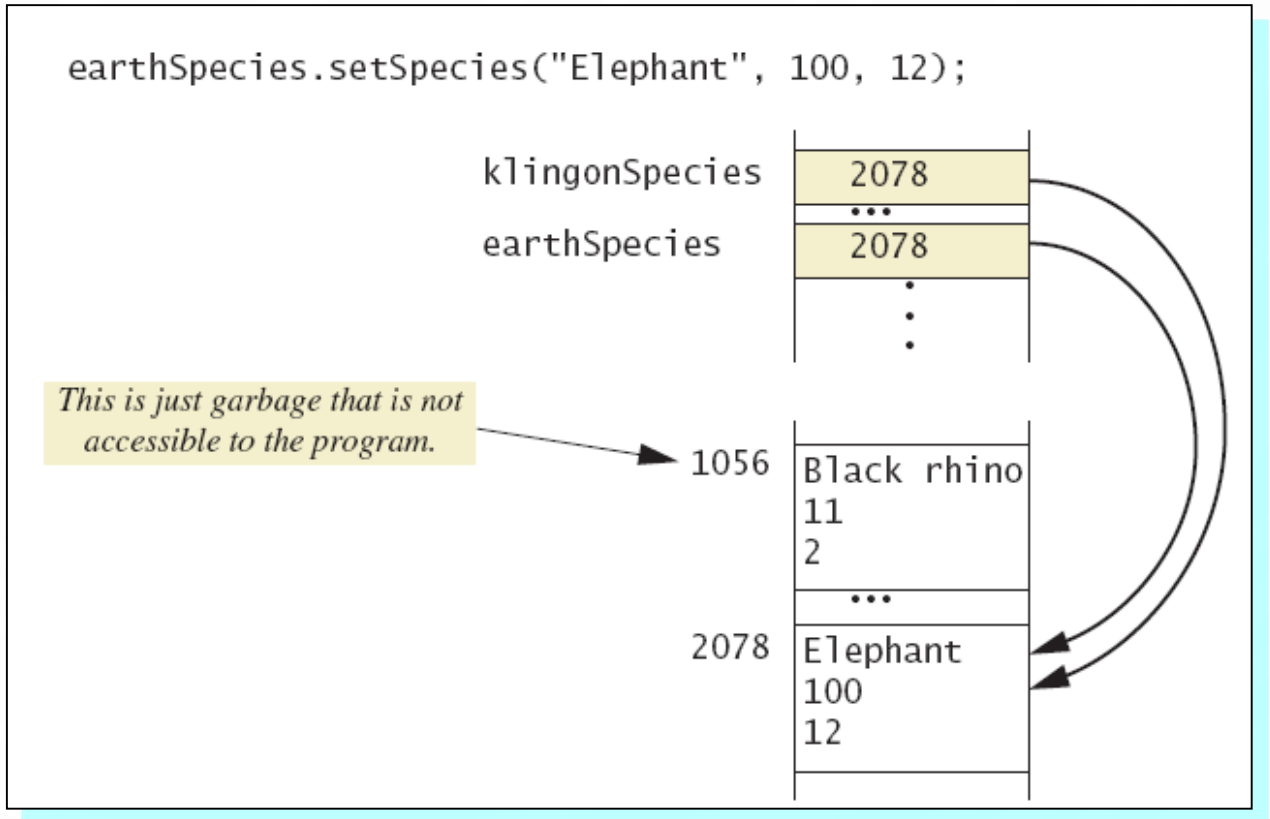
VARIABLES OF A CLASS TYPE

- Behavior of class variables



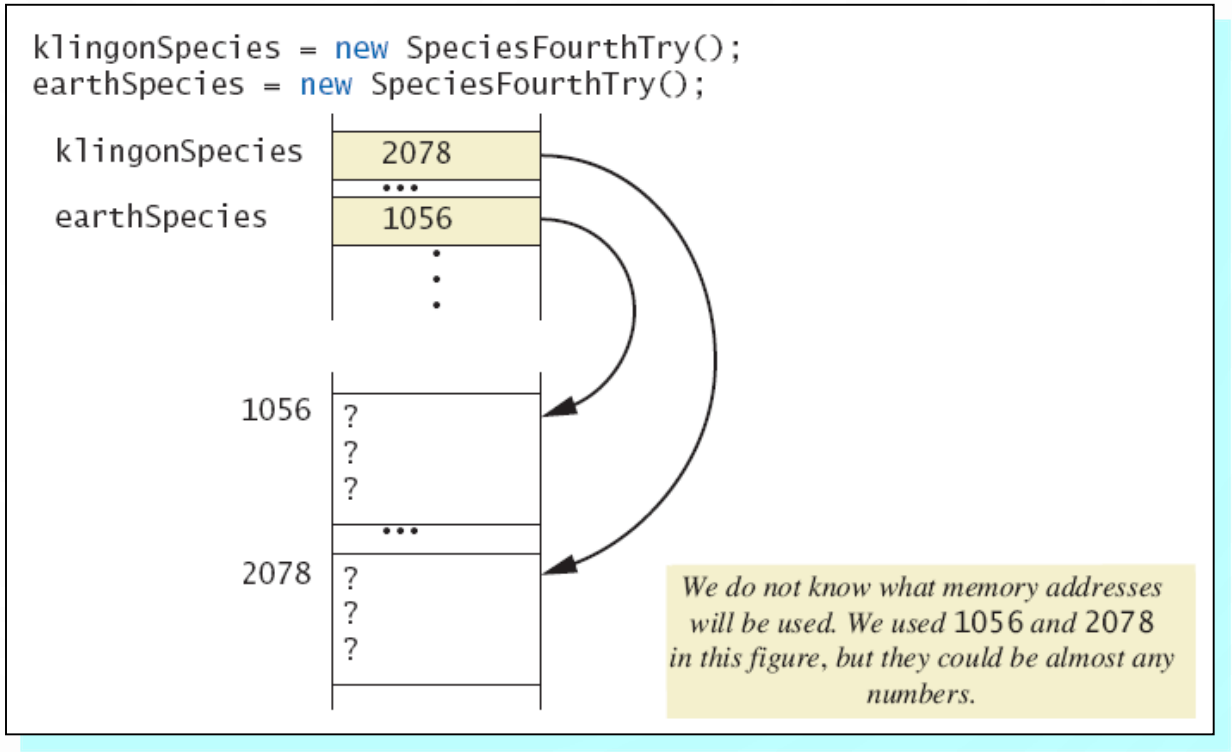
VARIABLES OF A CLASS TYPE

- Behavior of class variables



VARIABLES OF A CLASS TYPE

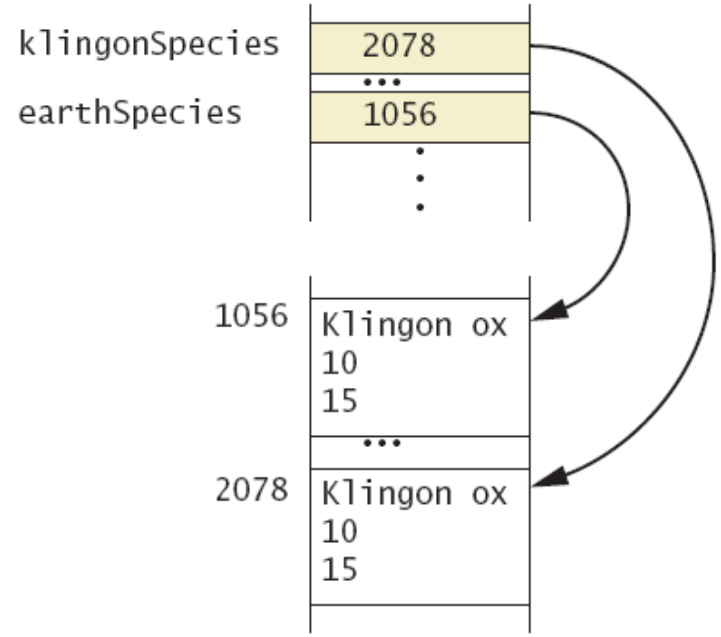
- Dangers of using == with objects



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

LISTING 5.17 Defining an `equals` Method

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

    <The definition of the methods readInput, writeOutput, and
     predictPopulation go here. They are the same as in
     Listing 5.3 and Listing 5.6.>
    <The definition of the methods setSpecies, getName,
     getPopulation, and getGrowthRate go here. They are the
     same as in Listing 5.11.>

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

`equalsIgnoreCase` *is a method of the class* `String`.

DEMONSTRATING AN `EQUALS` METHOD

- `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

LISTING 5.18 Demonstrating an `equals` Method *(part 1 of 2)*

```
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 change one Klingon ox to " +
                           "lowercase.");
```

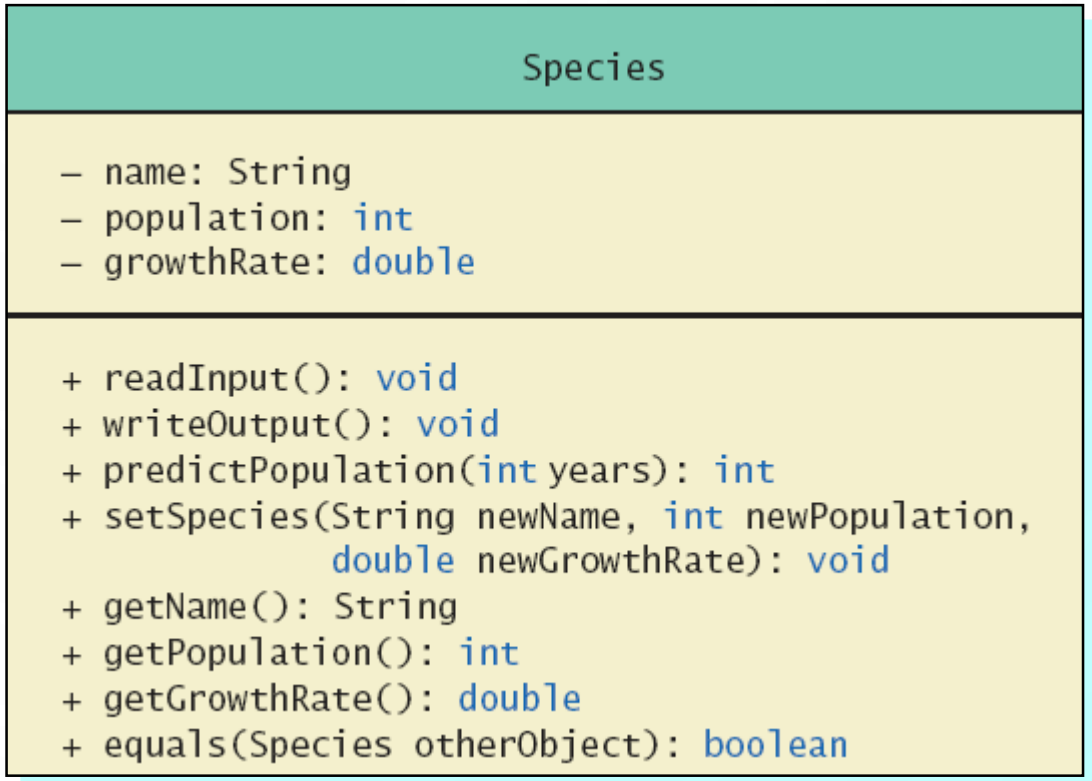
```
        s2.setSpecies("klingon ox", 10, 15); //Use lowercase
        if (s1.equals(s2))
            System.out.println("Match with the method equals.");
        else
            System.out.println("Do Not match with the method " +
                               "equals.");
    }
}
```

Screen Output

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

COMPLETE PROGRAMMING EXAMPLE

- class **Species**
- Class Diagram for the class **Species**



LISTING 5.19 The Complete Species Class *(part 1 of 2)*

This is the same class definition as in Listing 5.17, but with all the details shown.

```
import java.util.Scanner;
/**
Class for data on endangered species.
*/
public class Species
{
    private String name;
    private int population;
    private double growthRate;

    public void readInput()
    {
        Scanner keyboard = new Scanner(System.in);
        System.out.println("What is the species' name?");
        name = keyboard.nextLine();

        System.out.println(
                        "What is the population of the species?");
        population = keyboard.nextInt();
        while (population < 0)
        {
            System.out.println("Population cannot be negative.");
            System.out.println("Reenter population:");
            population = keyboard.nextInt();
        }
        System.out.println(
                        "Enter growth rate (% increase per year):");
        growthRate = keyboard.nextDouble();
    }
```

```
public void writeOutput()
{
    System.out.println("Name = " + name);
    System.out.println("Population = " + population);
    System.out.println("Growth rate = " + growthRate + "%");
}
/**
Precondition: years is a nonnegative number.
Returns the projected population of the receiving object
after the specified number of years.
*/
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;
}
```

```
    public void setSpecies(String newName, int newPopulation,
                           double newGrowthRate)
    {
        name = newName;
        if (newPopulation >= 0)
            population = newPopulation;
        else
        {
            System.out.println("ERROR: using a negative " +
                               "population.");
            System.exit(0);
        }
        growthRate = newGrowthRate;
    }
    public String getName()
    {
        return name;
    }
    public int getPopulation()
    {
        return population;
    }

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

This version of `equals` *is equivalent to the version in Listing 5.17. Here, the keyword* `this` *is understood to be there implicitly.*

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;
}
```

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**

LISTING 5.20 Sample Tests for the Species Class

```
public class SpeciesTest
{
    public static void main(String[] args)
    {
        Species testSpecies = new Species();

        // Test the setSpecies method
        testSpecies.setSpecies("Tribbles", 100, 50);
        if (testSpecies.getName().equals("Tribbles") &&
            (testSpecies.getPopulation() == 100) &&
            (testSpecies.getGrowthRate() >= 49.99) &&
            (testSpecies.getGrowthRate() <= 50.01))
        {
             System.out.println("Pass: setSpecies test.");
        }
        else
        {
             System.out.println("FAIL: setSpecies test.");
        }

        // Test the predictPopulation method
        if ((testSpecies.predictPopulation(-1) == 100) &&
            (testSpecies.predictPopulation(1) == 150) &&
            (testSpecies.predictPopulation(5) == 759))

        {
             System.out.println("Pass: predictPopulation test.");
        }
        else
        {
             System.out.println("FAIL: predictPopulation test.");
        }
    }
}
```

Sample Screen Output

```
Pass: setSpecies test.
Pass: predictPopulation test.
```

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

- `class DemoSpecies`
  - Note different parameter types and results

- Parameters of a class type versus parameters of a primitive type

`class ParametersDemo`

LISTING 5.21 A Demonstration Class *(part 1 of 2)*

```
import java.util.Scanner;
/**
This version of the class Species is only a toy example designed
to demonstrate the difference between parameters of a class type
and parameters of a primitive type.
*/
public class DemoSpecies
{
    private String name;
    private int population;
    private double growthRate;
    /**
    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,
    which is unchanged.
    */
```

```
    public void change(DemoSpecies otherObject)
    {
        otherObject.name = this.name;
        otherObject.population = this.population;
        otherObject.growthRate = this.growthRate;
    }
    <The rest of the class definition is the same as that of the class
    Species in Listing 5.19.>
}
```

LISTING 5.22 Parameters of a Class Type Versus Parameters of a Primitive Type

```
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();
    }
}
```

Screen Output

```
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%
```

An argument of a primitive type cannot change in value.

An argument of a class type cannot be replaced.

An argument of a class type can change in state.